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

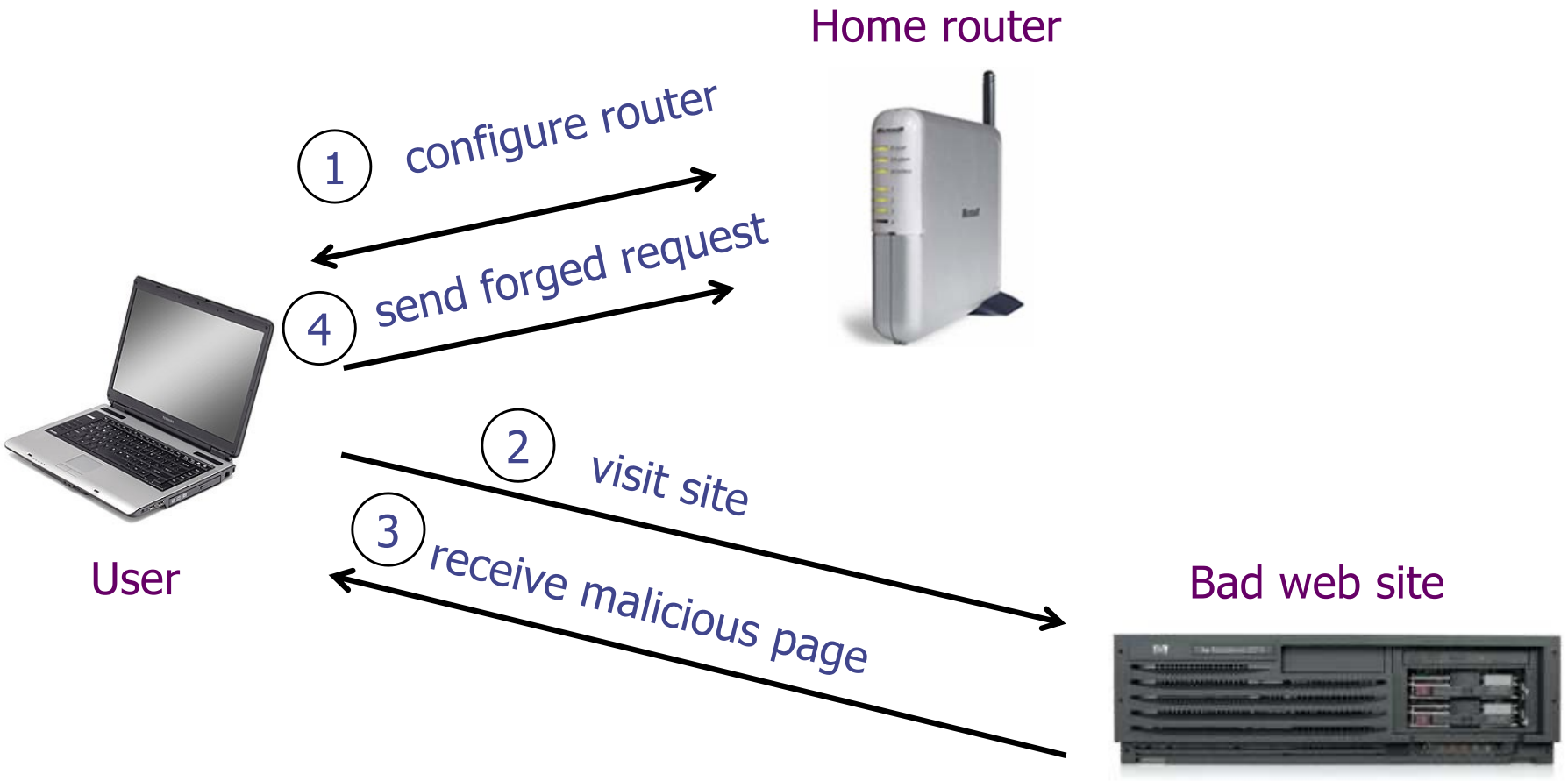
More OWASP

- « 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 based authentication …)

Not just Internet Web Servers: Attacks on Home Router



Not just Internet 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

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

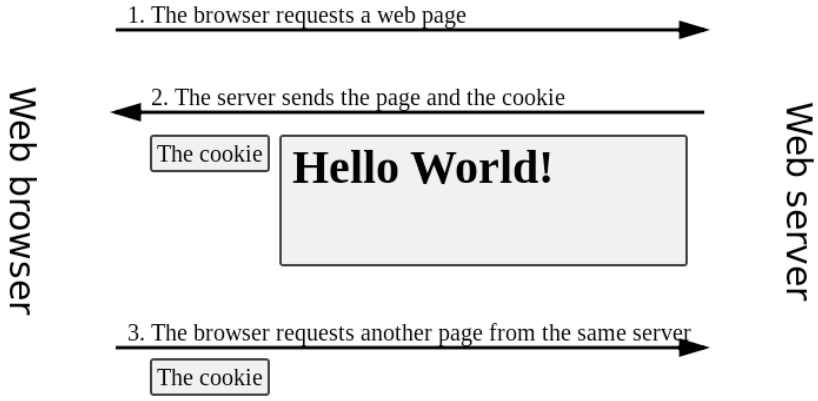
Maintaining Client State

- 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 (and more): 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
- More and more: controlling browser and server behavior / security features
  - E.g., ”Secure” attribute instructs that cookie should only be sent over HTTPS (confidentiality to prevent man-in-the-middle attack)

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

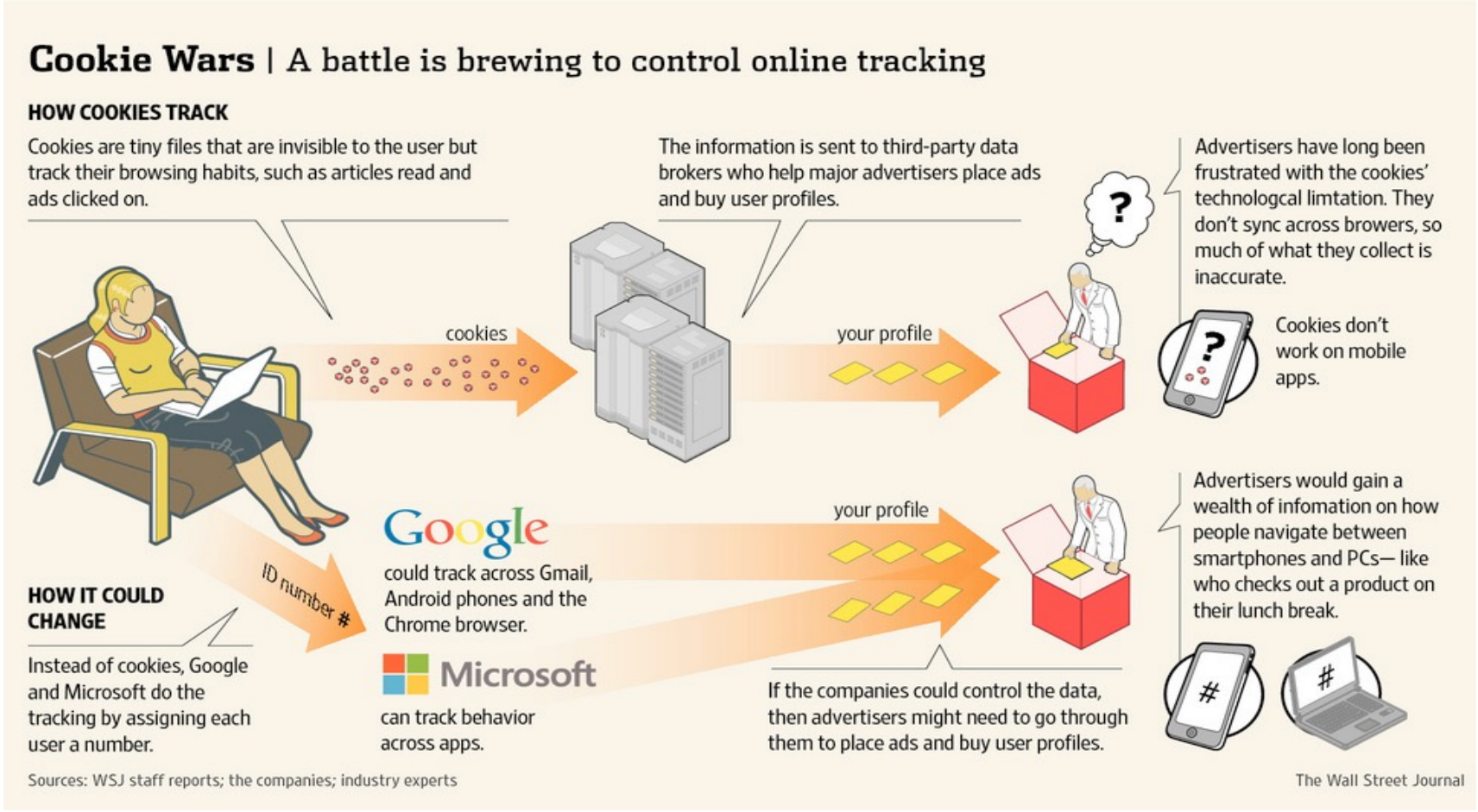
State management: URL rewriting

- 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/Cons: 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 browser for removing them

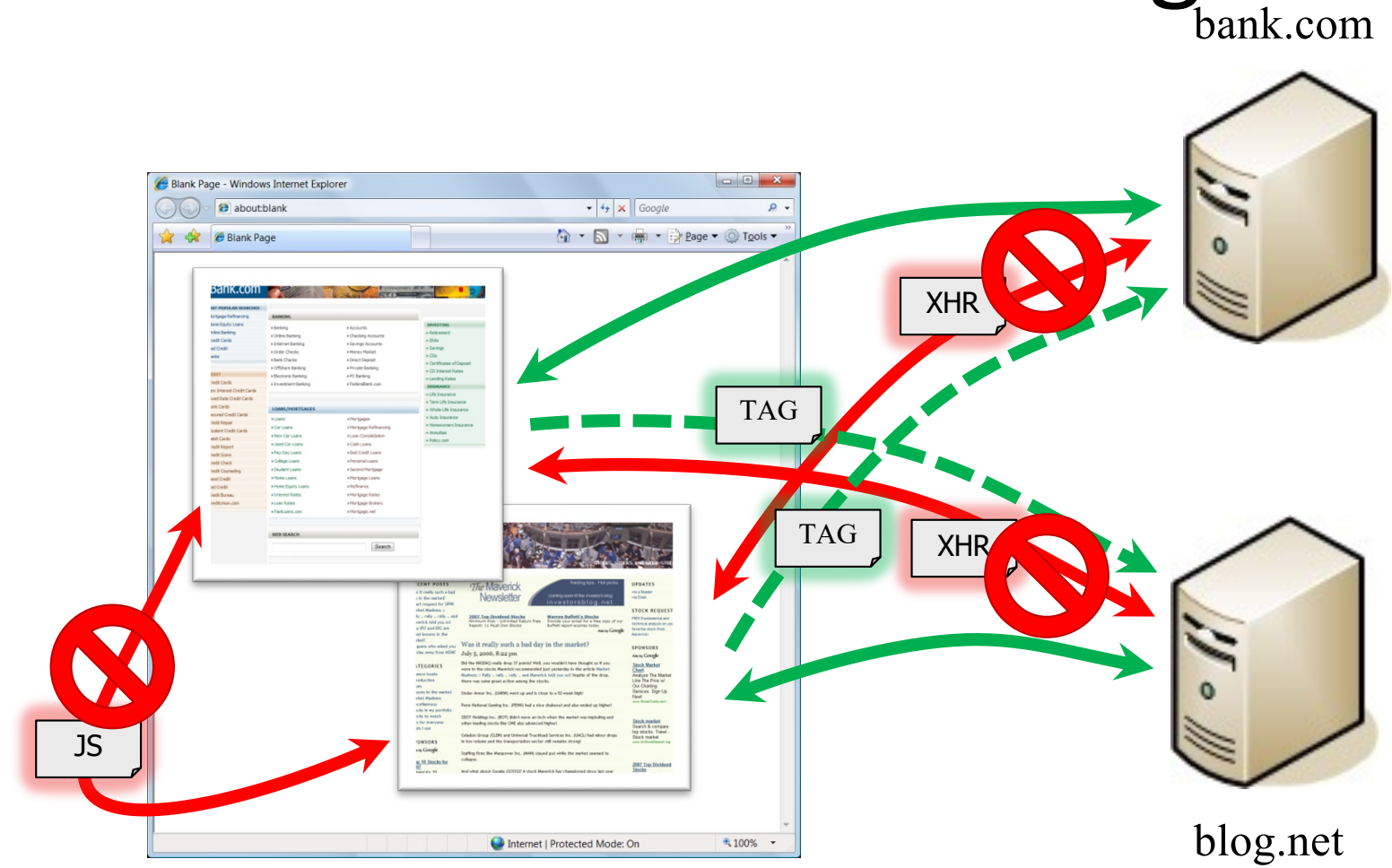
HTTP Cookies Security: History

- 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 stateful session with HTTP usage (Cookie / Set-Cookie)
- 2002: HttpOnly (XSS)
- 2011: RFC6265 – updates RFC 2965
- 2017-ongoing: RFC 6265bis (draft) - SameSite

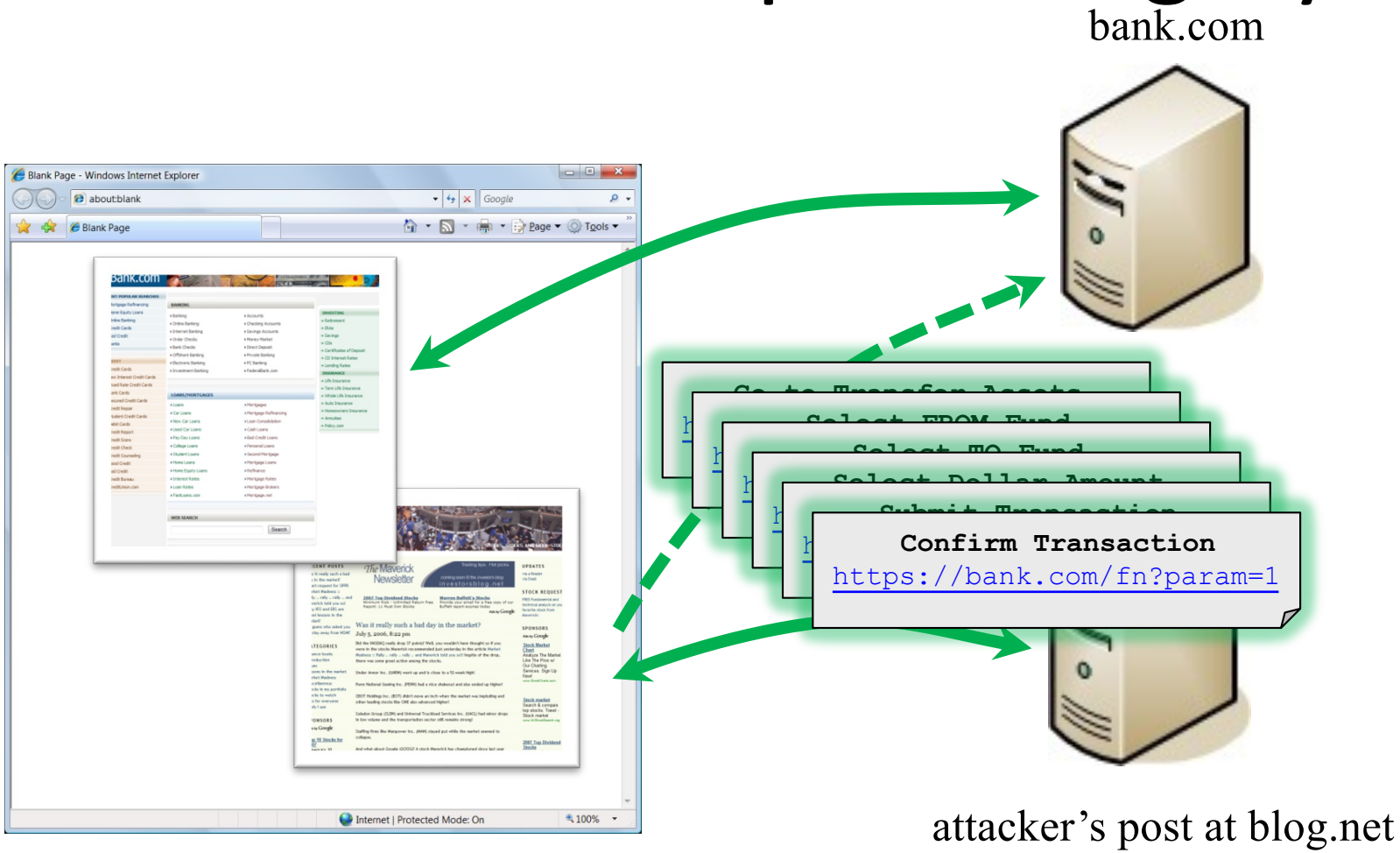
The Browser “Same Origin” Policy (SOP)

- 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)

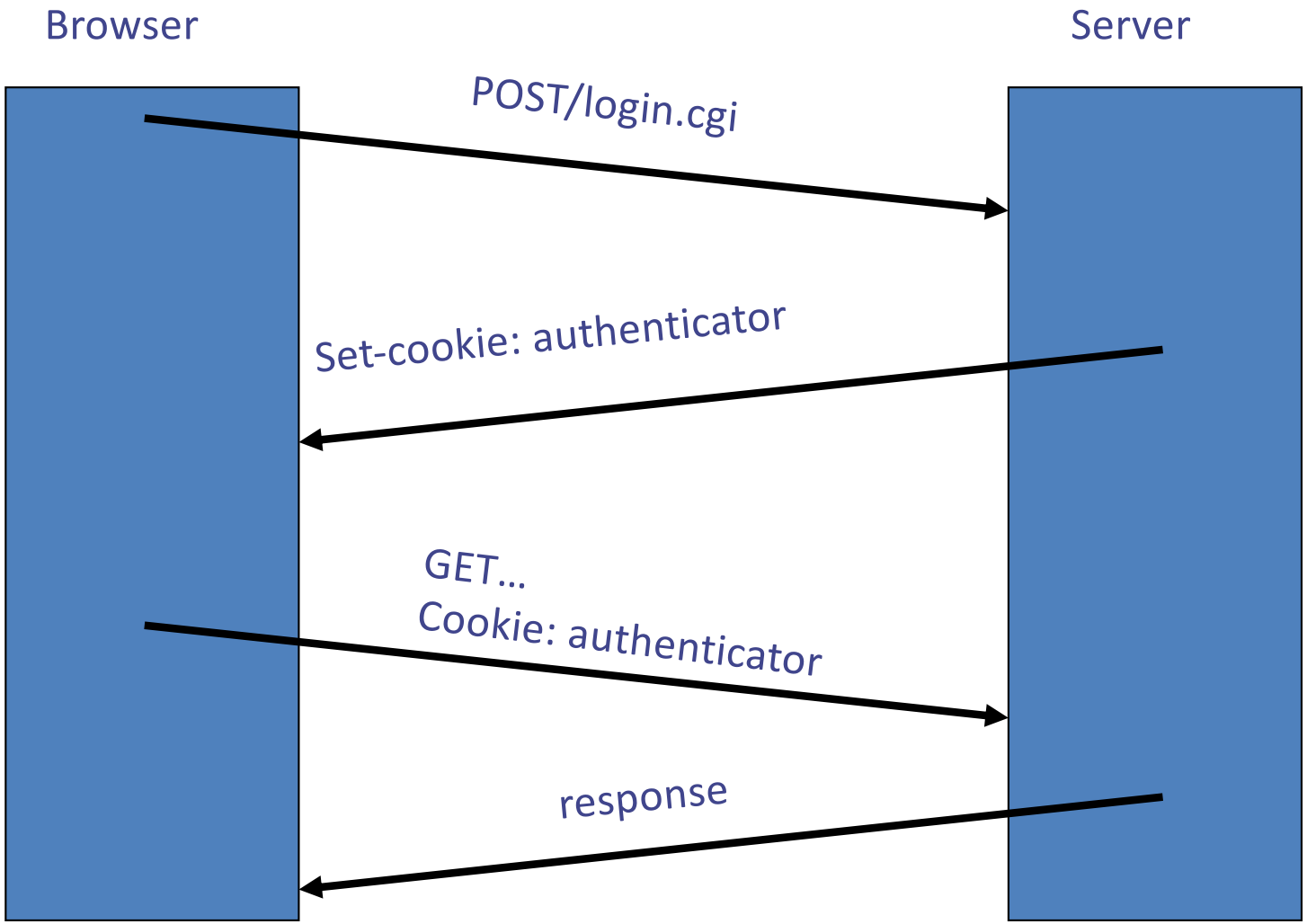
- XmlHttpRequest (AJAX)
  - Normally subject to same origin policy
  - But poorly managed CORS (Cross-Origin Resource Sharing) may relax these constraints …
  - May be fooled by a proxy too

Broader view of CSRF

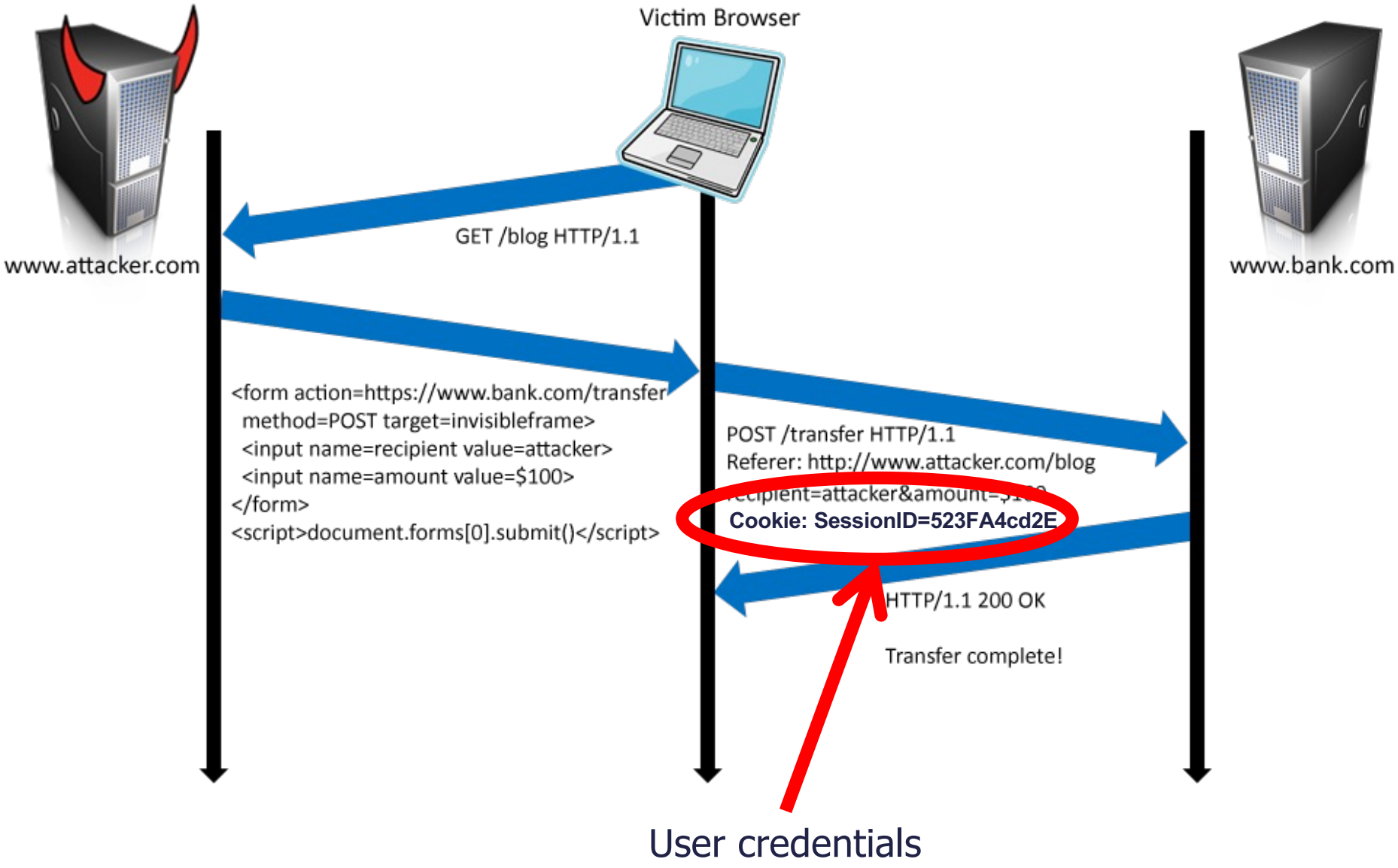
- Abuse of cross-site data export feature
  - From user's browser to honest server
  - Disrupts integrity of user's session
- Why mount a CSRF attack?
  - Network connectivity
  - Read browser state
  - Write browser state
- Not just "session riding"

Authentication: session using cookies

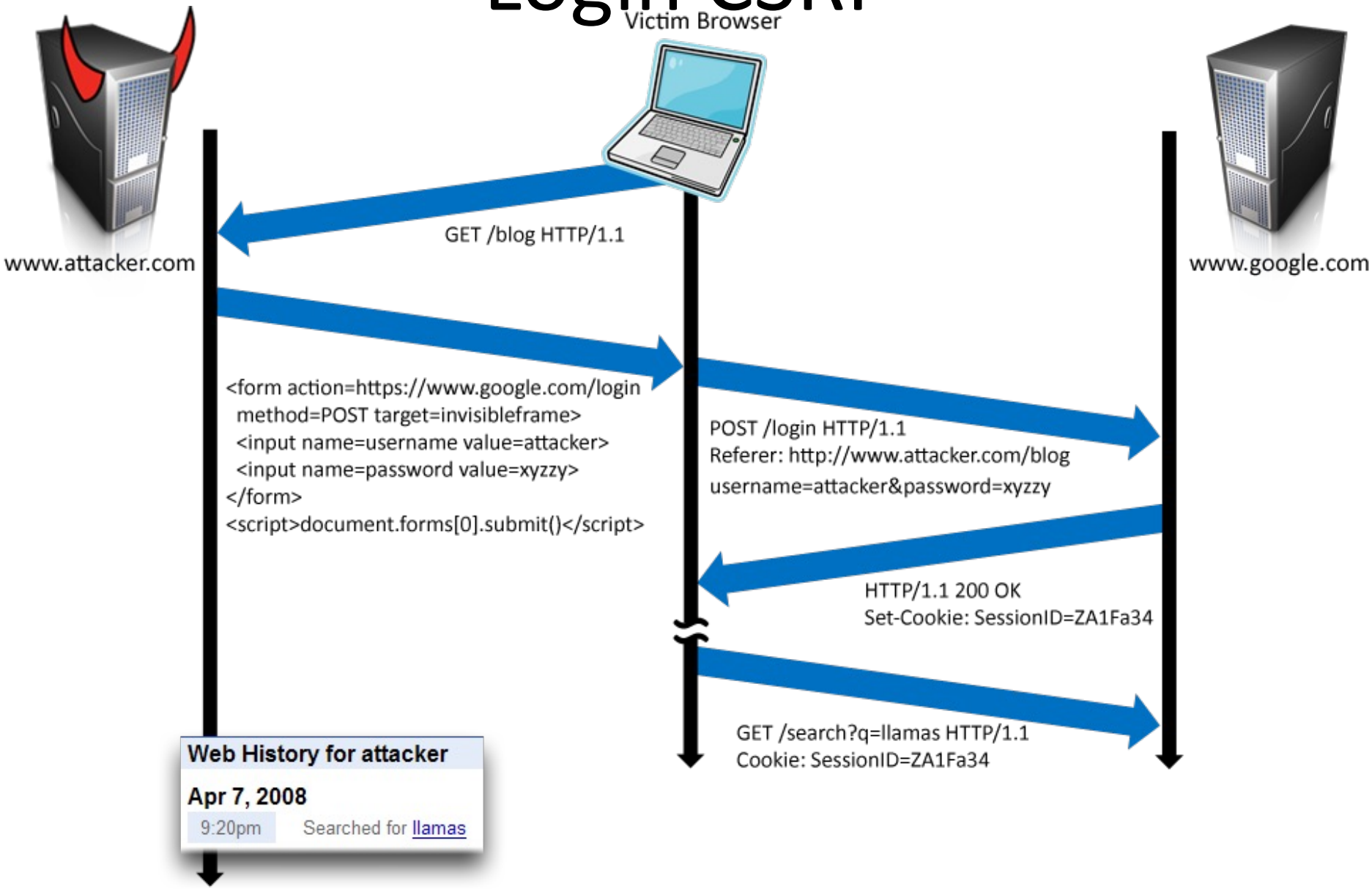
- Browser behavior: automatically attaches cookie previously set by server



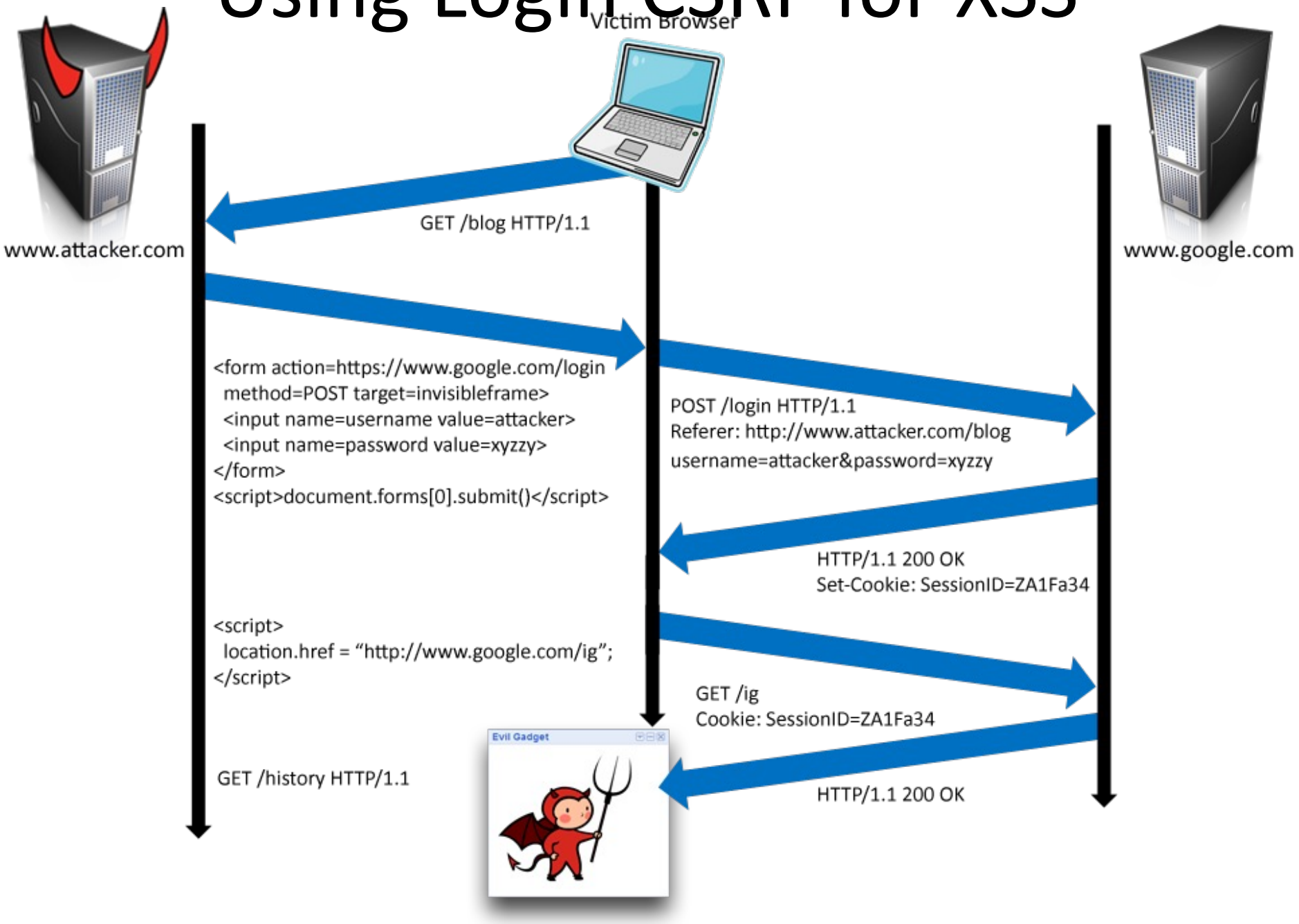
CSRF: Form post with cookie



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

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 Defense: secret tokens

- Persistent authentication (login/session data) validated for each HTTP request
  - Hard to guess secret (unguessability replaces unforgeability)
  - Sent through hidden field, instead of cookie, to prevent theft !!

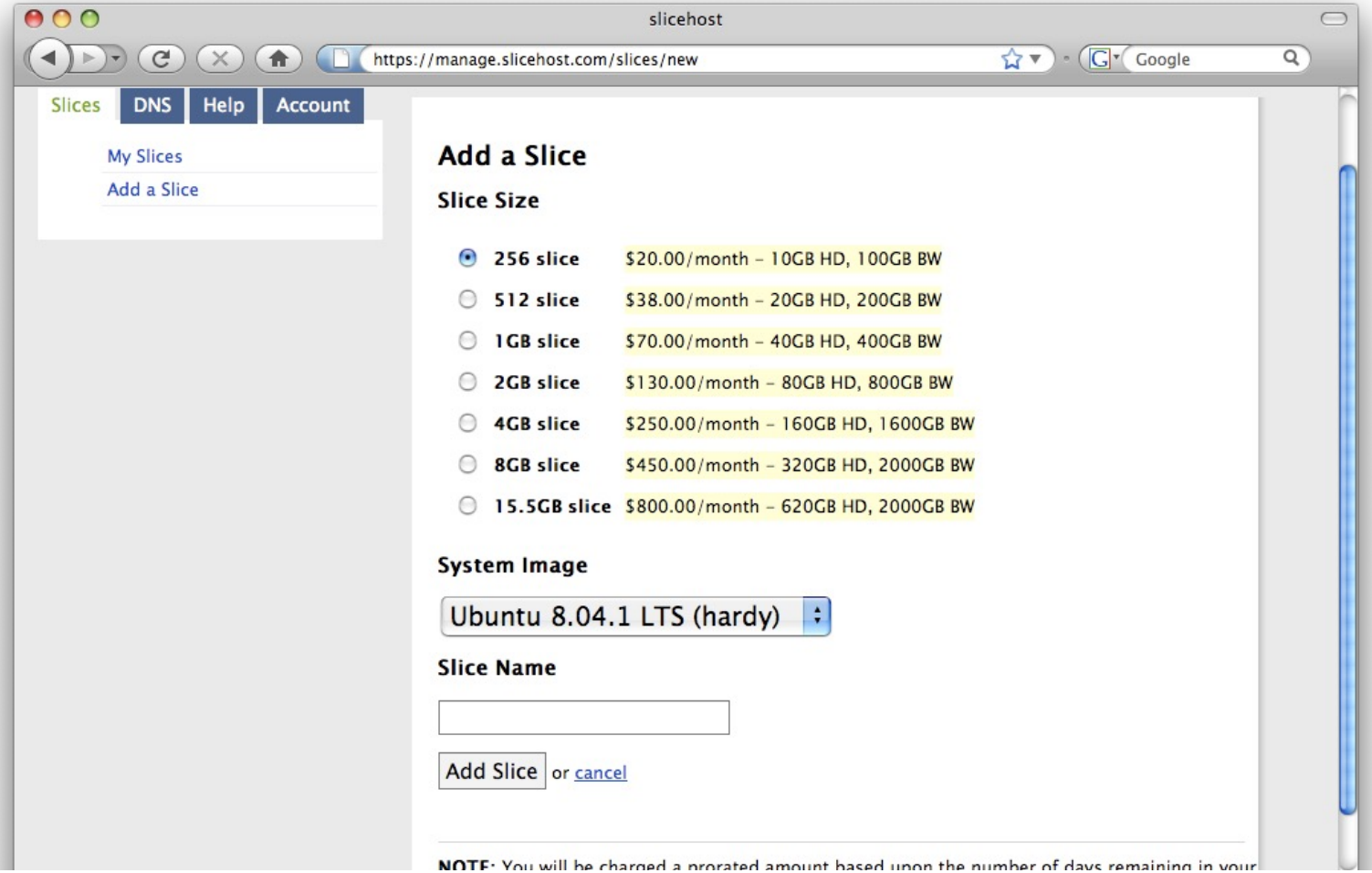
```
<input type=hidden value=23a3af01b>
```

- Variations
  - Session identifier
  - Session-independent token
  - Session-dependent token
  - HMAC / MD5 / SHA-1 of session identifier for integrity protection

- Tokens for Server-Side state maintenance
  - session ID + Secret Token Validation

- Tokens for stateless (client-side) state maintenance:
  - double-submission: token to be sent in header (request parameter) + cookie in body
  - Strong requirements (notably HTTPS to prevent attackers from injecting cookies, encrypted cookies)

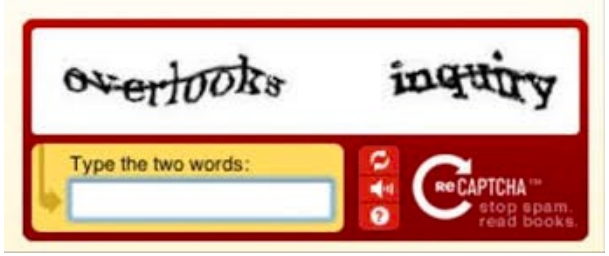
Secret Token Validation




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

Mitigation Strategies

- Check https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html
- Additional anti-CSRF HTML elements
  - Verifying request origin (referer attribute, Origin header)
  - SameSite cookie attribute (draft RFC 6265bis since 2017)
  - Custom HTTP Header (+CORS)
- 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)



Referer Validation

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

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

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>
```

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 application firewall rule (server-side)

```
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 : this generally means that you have to authenticate again each time the website/webapp is accessed
  - **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, not possible from iframe, img tag, script tag)
  - Most browsers now integrate this feature

Custom HTTP Header

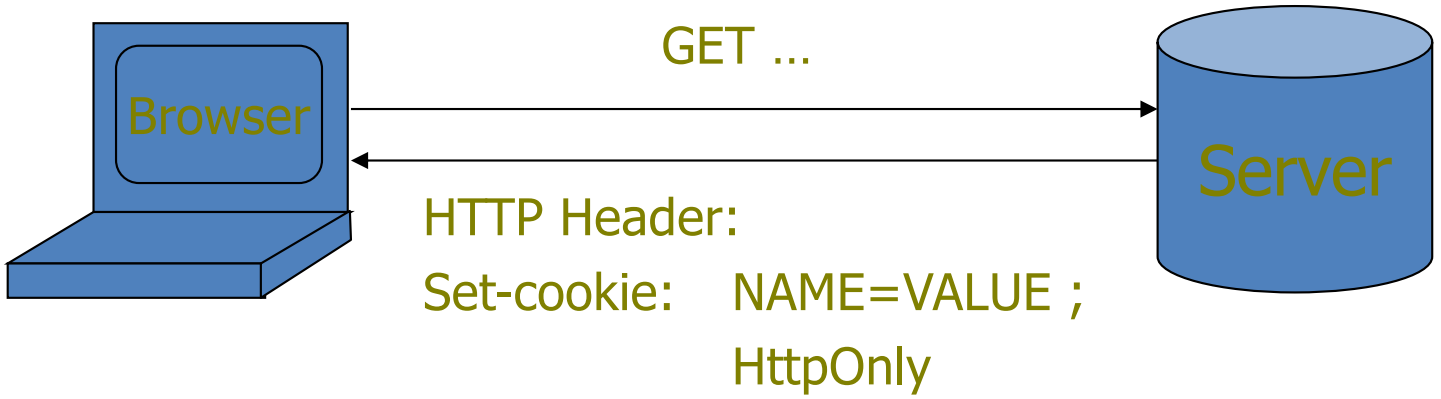
```
X-Requested-By: XMLHttpRequest
```

- SOP (browser feature):
  - HTTP requests performed via form, image, iframe, … unable to set custom HTTP headers.
  - Only accessible to XMLHttpRequest.
- Requires security policies built in to prevent attackers from sending CORS requests unless specifically allowed by (server defined) policy
- Simpler approach for AJAX/XHR

CORS: Cross-Origin Resource Sharing

- Allows access to requests from another origin
- This is allowed from the server side
- HTTP headers
  - Access-Control-Allow-Origin: http://domaine-a.localhost
  - Access-Control-Allow-Headers: X-Requested-With, Content-Type Access-Control-Allow-Methods: POST, GET

More cookies: HttpOnly against XSS



- 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 …

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 !!!

Additional solutions

Web Application Firewalls

- Help prevent some attacks we discussed:
  - Cross site scripting
  - SQL Injection
  - Form field tampering
  - Cookie poisoning

Sample products:

Imperva

Kavado Interdo

F5 TrafficShield

Citrix NetScaler

CheckPoint Web Intel

Code checking

- Blackbox security testing services:
  - Whitehatsec.com
- Automated blackbox testing tools:
  - Cenzic, **Hailstorm**
  - Spidynamic, **WebInspect**
  - eEye, **Retina**
- Web application hardening tools:
  - WebSSARI [WWW'04] : based on information flow
  - Nguyen-Tuong [IFIP'05] : based on tainting

Summary

- SQL Injection
  - Bad input checking allows malicious SQL query
  - Known defenses address problem effectively
  - But many other injections
- 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