Lecture 26 – Browser Security

Stephen Checkoway
Oberlin College
Some slides from Bailey's ECE 422

Documents

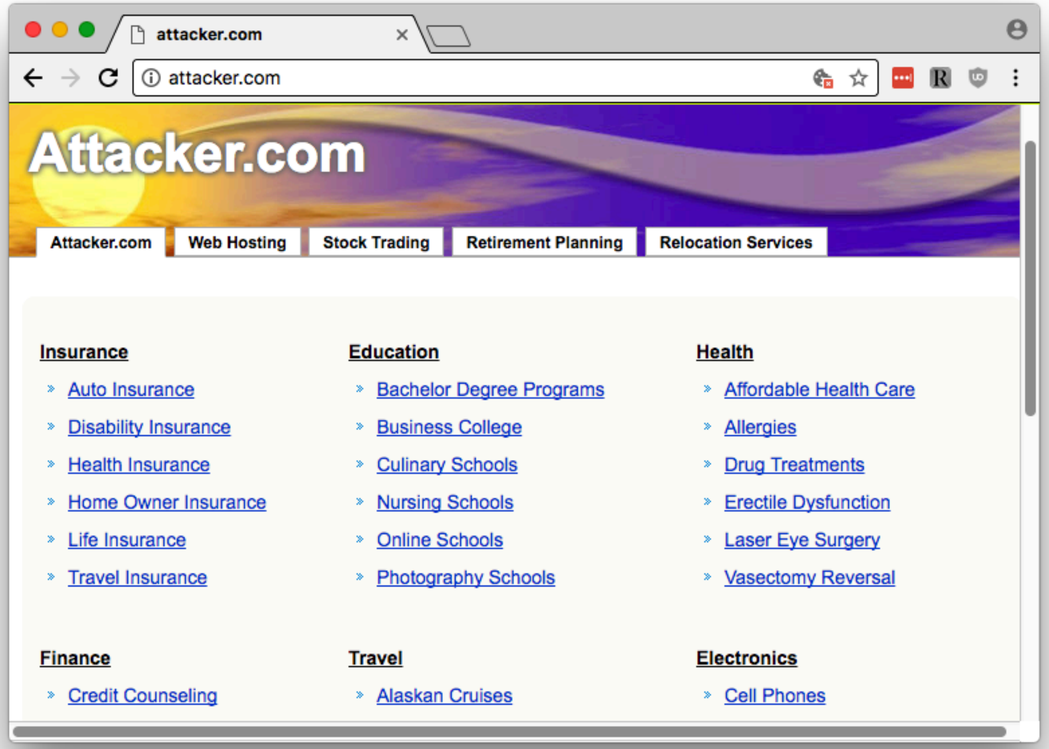
- Browser's fundamental role is to display documents comprised of
  - HTML
  - JavaScript
  - Style sheets (CSS)
  - Images
  - Sounds
  - Movies
  - ~~Plugin content~~
    - ~~Flash~~
    - ~~SilverLight~~
    - ~~QuickTime~~
    - ~~...~~

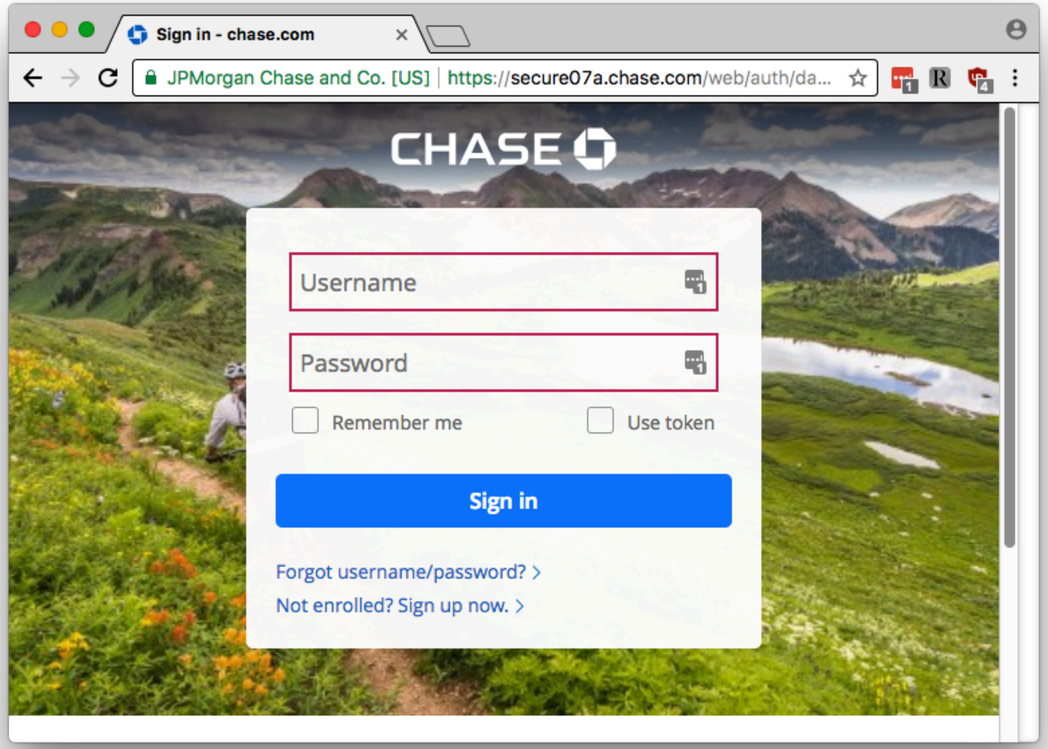
Document Object Model (DOM)

- The browser allows scripts to
  - add/modify/delete/style the DOM elements
  - make changes in response to user actions (e.g., clicks)
  - submit forms
  - browse to a new document altogether
- Scripts in one document can modify another document
  - We say that Document A *scripts* Document B

Scripting other documents

- Very powerful capability and without constraints would be dangerous
- Consider having attacker.com open while logging into chase.com
- If attacker.com can script chase.com, what could happen?





Clearly we need separation

- This is Risk #3 from last time
- Same Origin Policy (SOP)
  - Goal: Partition documents into equivalence classes that can script each other (including reading each others' content)
  - Each document is assigned an origin and documents can script other documents in the same origin
  - We construct the origin from the URL

From URLs to Origins

- General form of a URL
  scheme://user:pass@host:port/path?querystring#fragment
- Most parts are optional giving URLs like
  https://www.cs.oberlin.edu/~csmc/
  https://google.com?q=hello+world
- Origins are the triple (scheme, host, port)
  What's the origin for https://www.cs.oberlin.edu/~csmc/?

  What's the origin for https://google.com?q=hello+world?

From URLs to Origins

- General form of a URL
  scheme://user:pass@host:port/path?querystring#fragment
- Most parts are optional giving URLs like
  https://www.cs.oberlin.edu/~csmc/
  https://google.com?q=hello+world
- Origins are the triple (scheme, host, port)
  What's the origin for https://www.cs.oberlin.edu/~csmc/?

  (https, www.cs.oberlin.edu, 443)

  What's the origin for https://google.com?q=hello+world?

From URLs to Origins

- General form of a URL
  scheme://user:pass@host:port/path?querystring#fragment
- Most parts are optional giving URLs like
  https://www.cs.oberlin.edu/~csmc/
  https://google.com?q=hello+world
- Origins are the triple (scheme, host, port)
  What's the origin for https://www.cs.oberlin.edu/~csmc/?

  (https, www.cs.oberlin.edu, 443)

  What's the origin for https://google.com?q=hello+world?

  (https, google.com, 443)

Origins

Origins

- Why does the origin include the host?

Origins

- Why does the origin include the host?
  - To prevent attacker.com from scripting bank.com

Origins

- Why does the origin include the host?
  - To prevent attacker.com from scripting bank.com
- Why does the origin include the scheme?

Origins

- Why does the origin include the host?
  - To prevent attacker.com from scripting bank.com
- Why does the origin include the scheme?
  - If not, then http://bank.com can script https://bank.com. An "on-path" attacker could inject <script>…</script> into http://bank.com which affects https://bank.com

Origins

- Why does the origin include the host?
  - To prevent attacker.com from scripting bank.com
- Why does the origin include the scheme?
  - If not, then http://bank.com can script https://bank.com. An "on-path" attacker could inject <script>…</script> into http://bank.com which affects https://bank.com
- Why does the origin include the port?

Origins

- Why does the origin include the host?
  - To prevent attacker.com from scripting bank.com
- Why does the origin include the scheme?
  - If not, then http://bank.com can script https://bank.com. An "on-path" attacker could inject <script>…</script> into http://bank.com which affects https://bank.com
- Why does the origin include the port?
  - Think about multiple web servers run by different users on the same machine. Without including the port, https://host.com:8443 could script the entirely unrelated https://host.com

Not the end of the story

- Documents (and thus scripts) can load elements from other origins including images, scripts, style sheets, and ~~flash objects~~
  - Loading these elements endorses their content and the included elements are considered to be in the loading document's origin
- Conversely, documents (and thus scripts) can submit forms which sends data from the document to some server
  - Submitting forms declassifies the data sent
- Cross-Origin Resource Sharing (CORS) can enable cross-origin requests

Web Review | HTTP

gmail.com

Web Review | HTTP

```
GET / HTTP/1.1
Host:  gmail.com
```

gmail.com

Web Review | HTTP

```
GET / HTTP/1.1
Host:  gmail.com
```

```
HTTP/1.1 200 OK
…
<html>
  <head>
     <script>alert(‘Hi!’)</script>
  </head>
<img src=“//gmail.com/img.png”/>
```

gmail.com

Web Review | HTTP

```
GET / HTTP/1.1
Host:  gmail.com
```


```
HTTP/1.1 200 OK
…
<html>
  <head>
     <script>alert('Hi!')</script>
  </head>
<img src="//gmail.com/img.png"/>
```

Web Review | HTTP

```
GET / HTTP/1.1
Host:  gmail.com
```


```
HTTP/1.1 200 OK
…
<html>
  <head>
     <script>alert('Hi!')</script>
  </head>
<img src="//gmail.com/img.png"/>
```


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

Web Review | HTTP

```
GET / HTTP/1.1
Host:  gmail.com
```


```
HTTP/1.1 200 OK
…
<html>
  <head>
     <script>alert('Hi!')</script>
  </head>
<img src="//gmail.com/img.png"/>
```


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

```
HTTP/1.1 200 OK
…
<89>PNG^M …
```

Web Review | HTTP

```
GET / HTTP/1.1
Host:  gmail.com
```


```
HTTP/1.1 200 OK
…
<html>
  <head>
     <script>alert('Hi!')</script>
  </head>
<img src="//gmail.com/img.png"/>
```


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

```
HTTP/1.1 200 OK
…
<89>PNG^M …
```

Web Review | AJAX (jQuery style)

gmail.com

Web Review | AJAX (jQuery style)

```
GET / HTTP/1.1
Host:  gmail.com
```

gmail.com

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

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

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

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

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

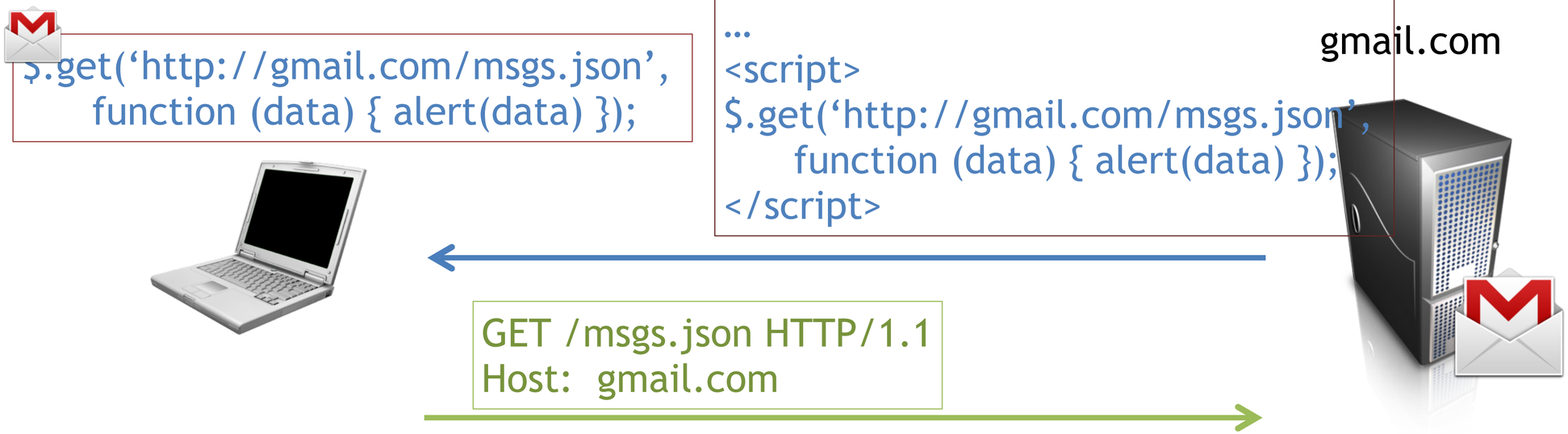
```
GET /msgs.json HTTP/1.1
Host:  gmail.com
```

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

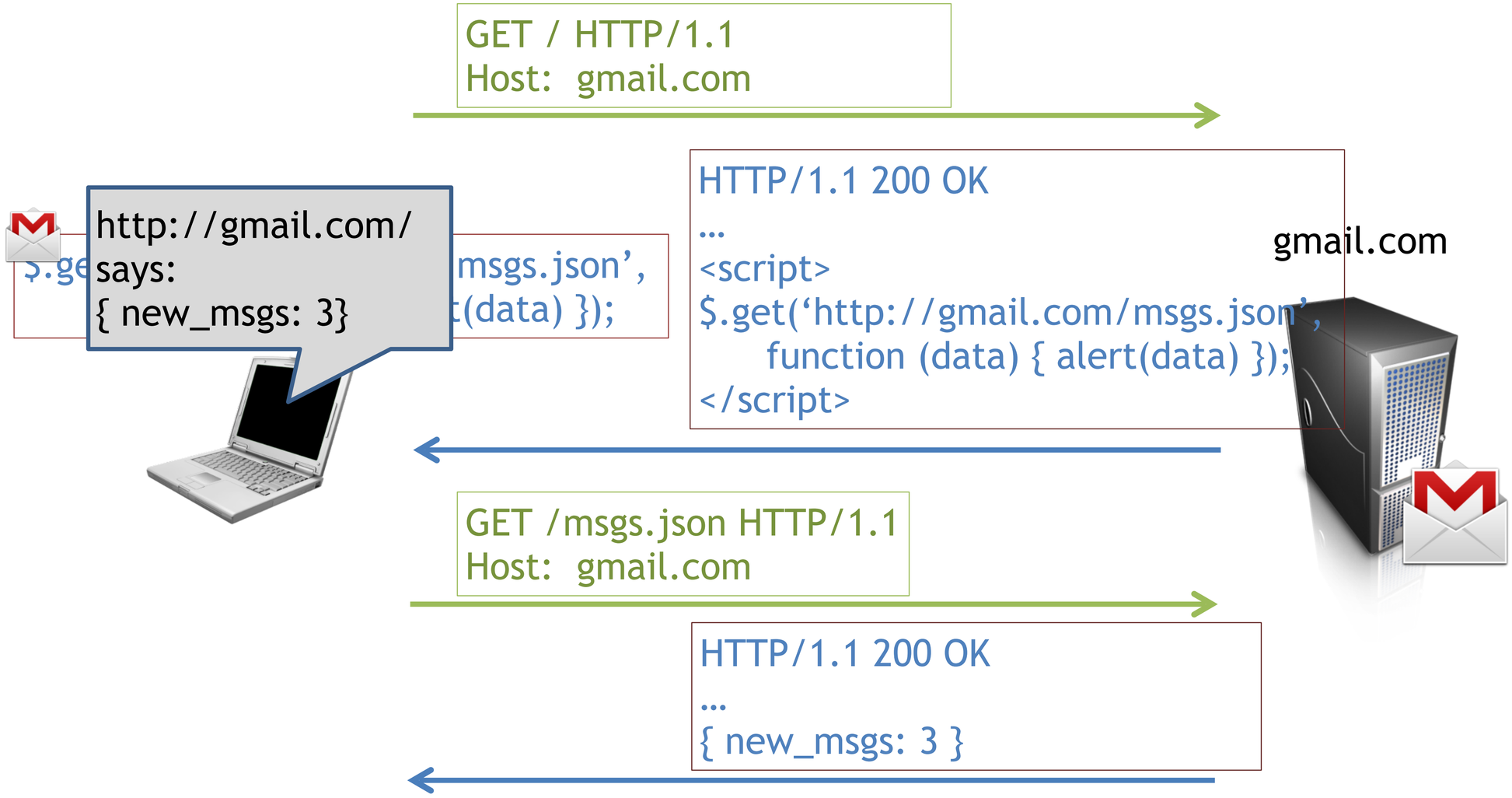


$.get('http://gmail.com/msgs.json',
function (data) { alert(data) });

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

HTTP/1.1 200 OK
…
{ new_msgs: 3 }

Web Review | AJAX (jQuery style)



Web Review | Same-Origin Policy (SOP)

(evil!)
facebook.com

gmail.com

Web Review | Same-Origin Policy (SOP)

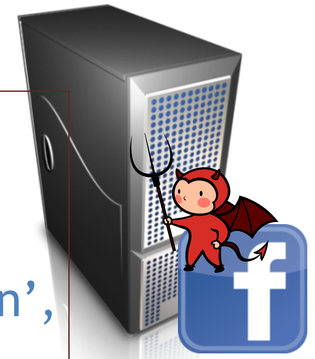
GET / HTTP/1.1
Host: facebook.com

(evil!)
facebook.com

gmail.com

Web Review | Same-Origin Policy (SOP)

(evil!)
facebook.com



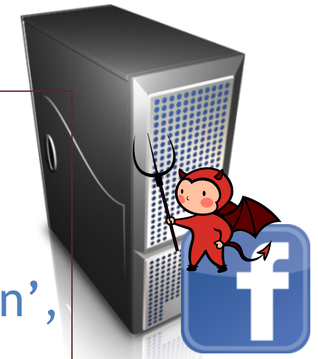
```
GET / HTTP/1.1
Host:  facebook.com
```

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

gmail.com

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>
```

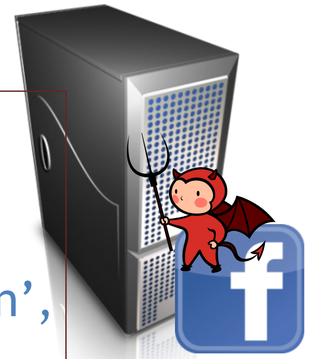
```
$.get('http://gmail.com/msgs.json',
    function (data) { alert(data); }
```

gmail.com

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

(evil!)
facebook.com



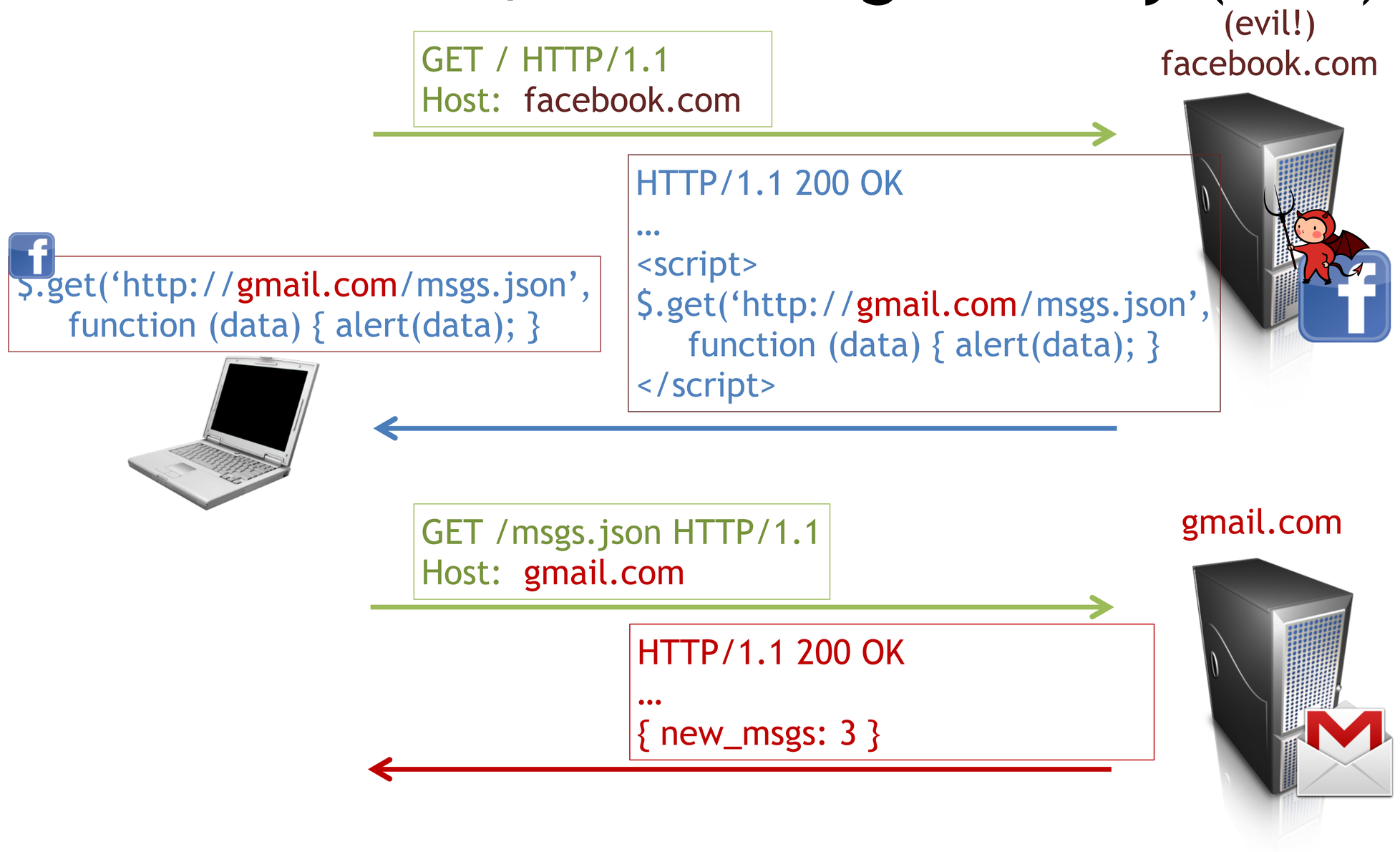
HTTP/1.1 200 OK
…
<script>
$.get('http://gmail.com/msgs.json',
function (data) { alert(data); }
</script>

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

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

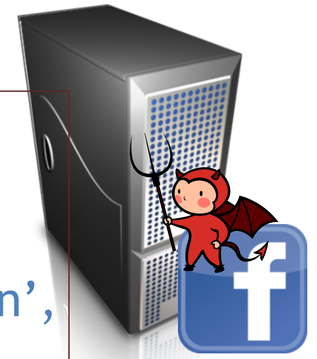
gmail.com

Web Review | Same-Origin Policy (SOP)



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); }
```

gmail.com

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

```
HTTP/1.1 200 OK
…
{ new_msgs: 3 }
```

Web Review | Same-Origin Policy (SOP)

facebook.com

gmail.com

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

facebook.com

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
…

?

gmail.com

Web Review | Same-Origin Policy (SOP)

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

facebook.com

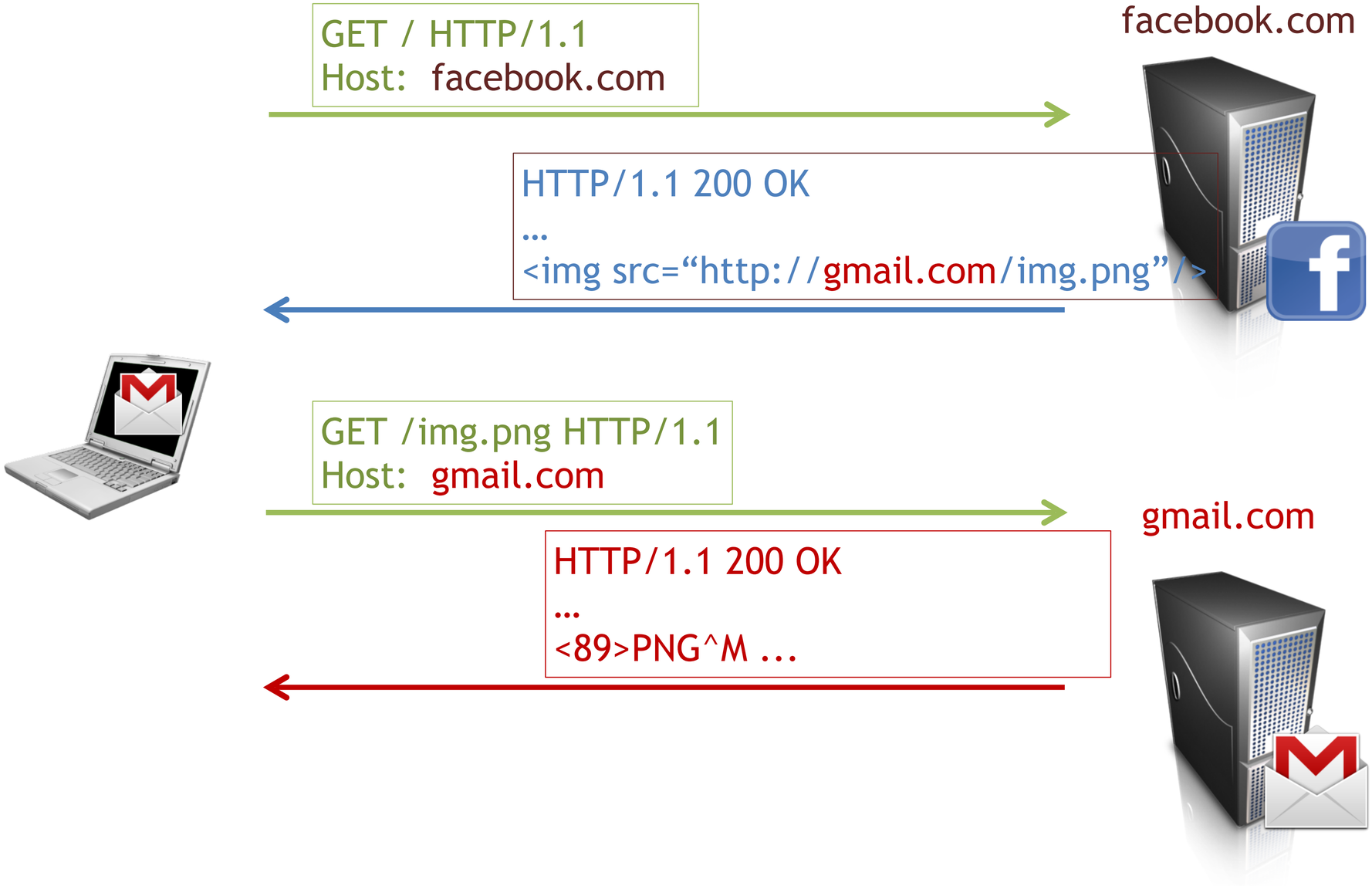
HTTP/1.1 200 OK
…

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)



Web Review | Same-Origin Policy (SOP)

facebook.com

gmail.com

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

facebook.com

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"/>

gmail.com

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

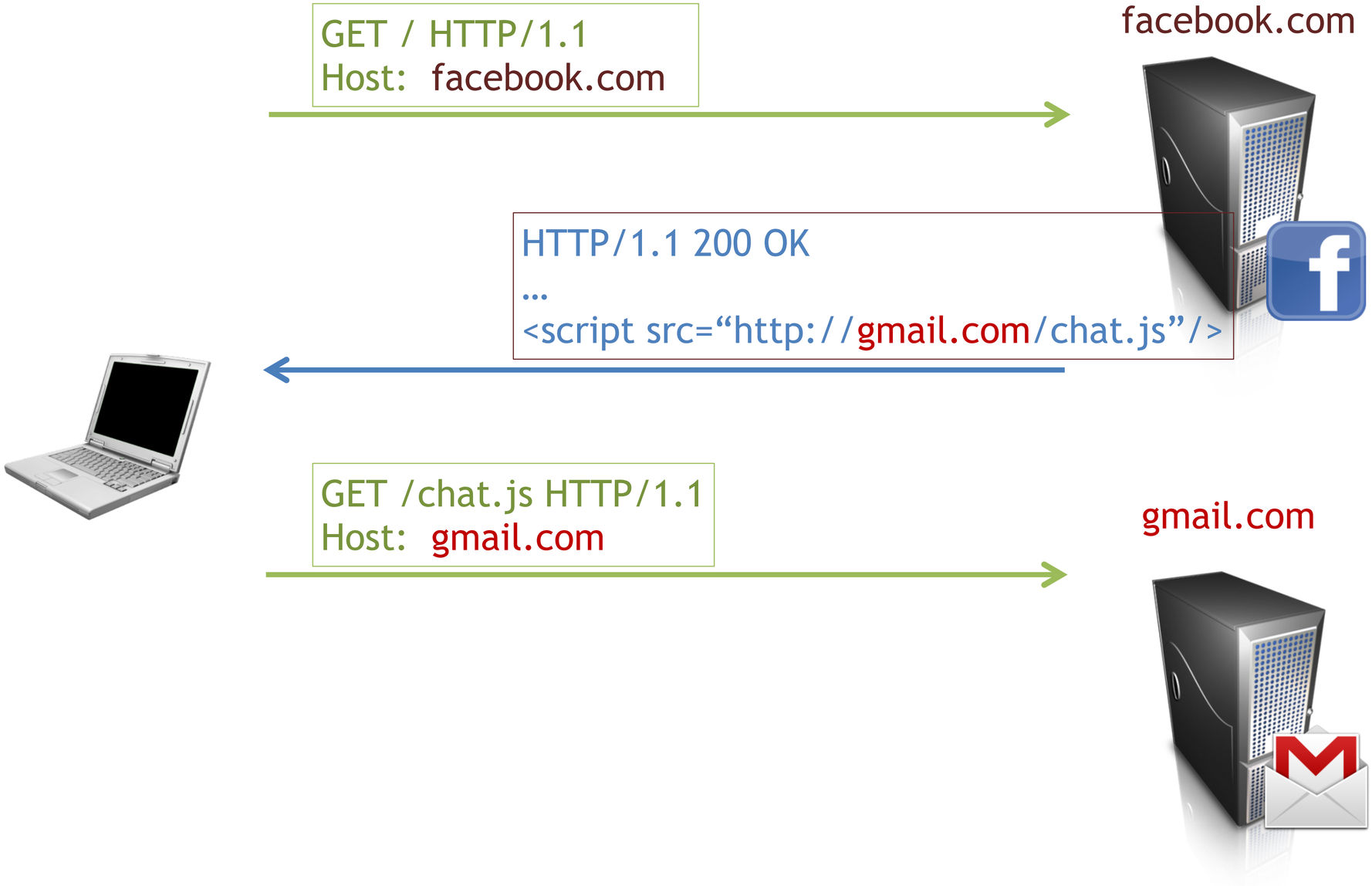
facebook.com

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

?

gmail.com

Web Review | Same-Origin Policy (SOP)



Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

facebook.com

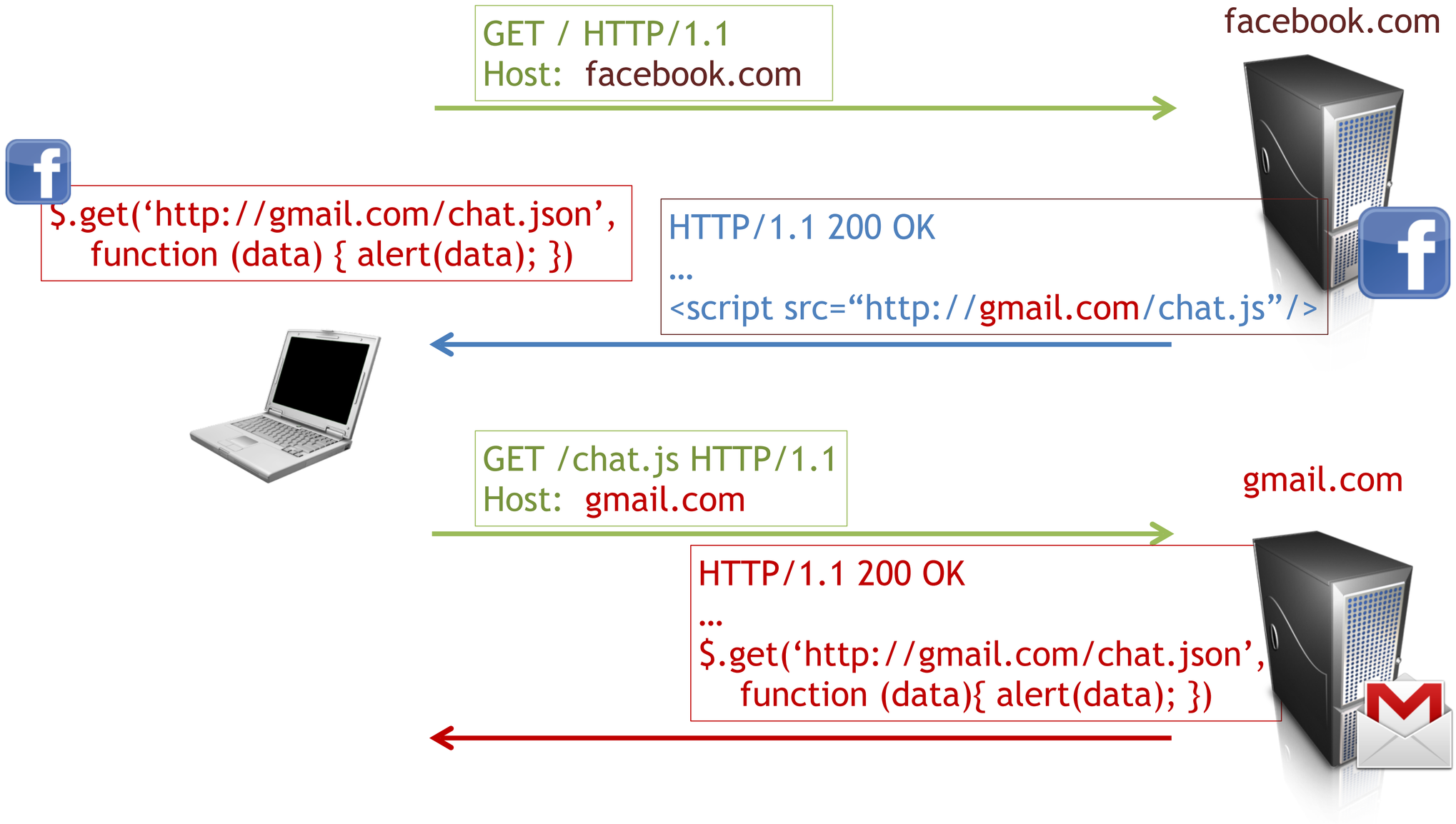
HTTP/1.1 200 OK
…
<script src="http://gmail.com/chat.js"/>

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

gmail.com

Web Review | Same-Origin Policy (SOP)

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

```
GET /chat.json HTTP/1.1
Host:  gmail.com
```

gmail.com

Web Review | Same-Origin Policy (SOP)

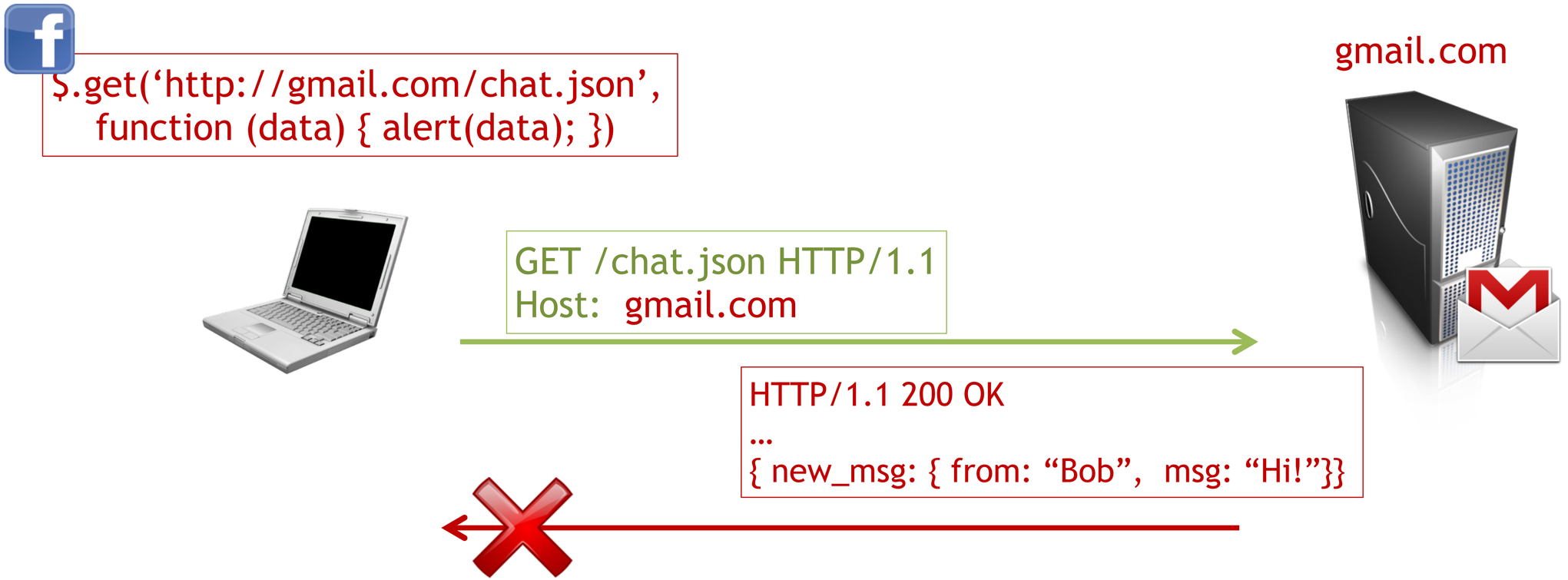
```
$.get('http://gmail.com/chat.json',
    function (data) { alert(data); })
```

gmail.com

```
GET /chat.json HTTP/1.1
Host:  gmail.com
```

```
HTTP/1.1 200 OK
…
{ new_msg: { from: "Bob",  msg: "Hi!"}}
```

Web Review | Same-Origin Policy (SOP)



iframes

- Complete document inside a document
  <iframe src="https://somewhere.com/page.html"></iframe>
- The contents of each iframe belong to its source origin (https, somewhere.com, 443) for the iframe above
- The iframe element itself belongs to its containing document
- iframes obey the SOP

Web Review | Same-Origin Policy (SOP)

facebook.com

gmail.com

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

facebook.com

gmail.com

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)

GET / HTTP/1.1
Host: facebook.com

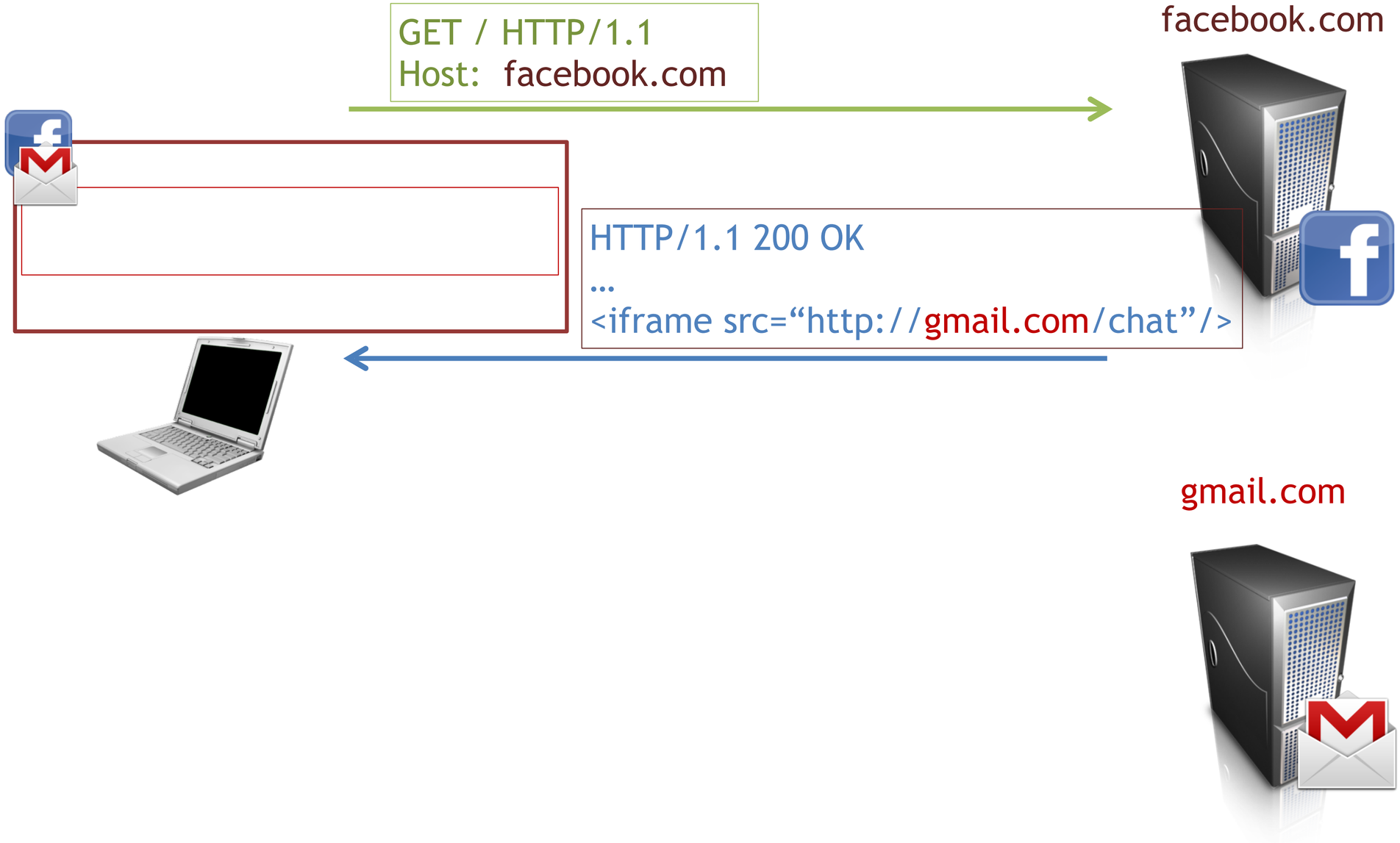
facebook.com

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

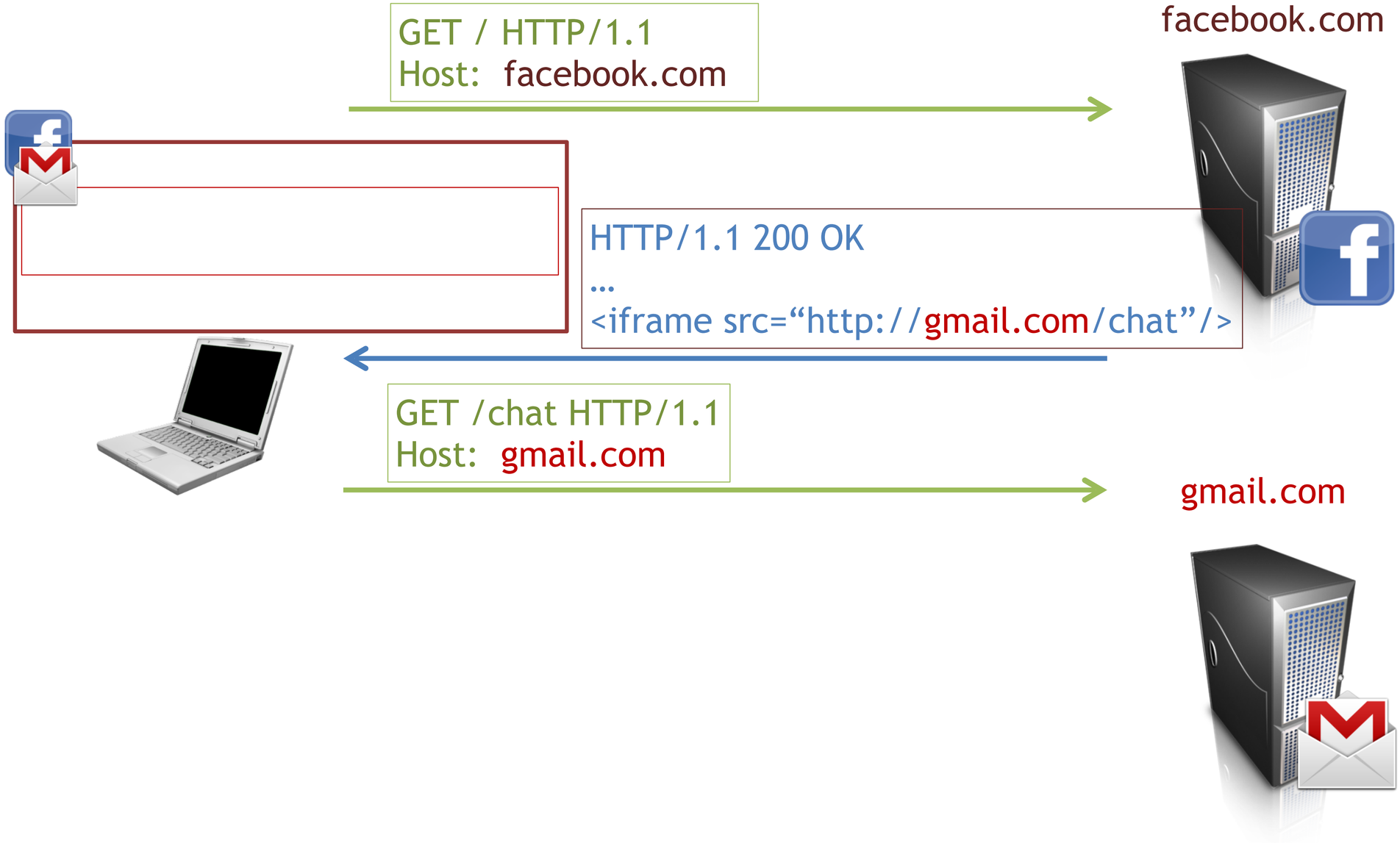
?

gmail.com

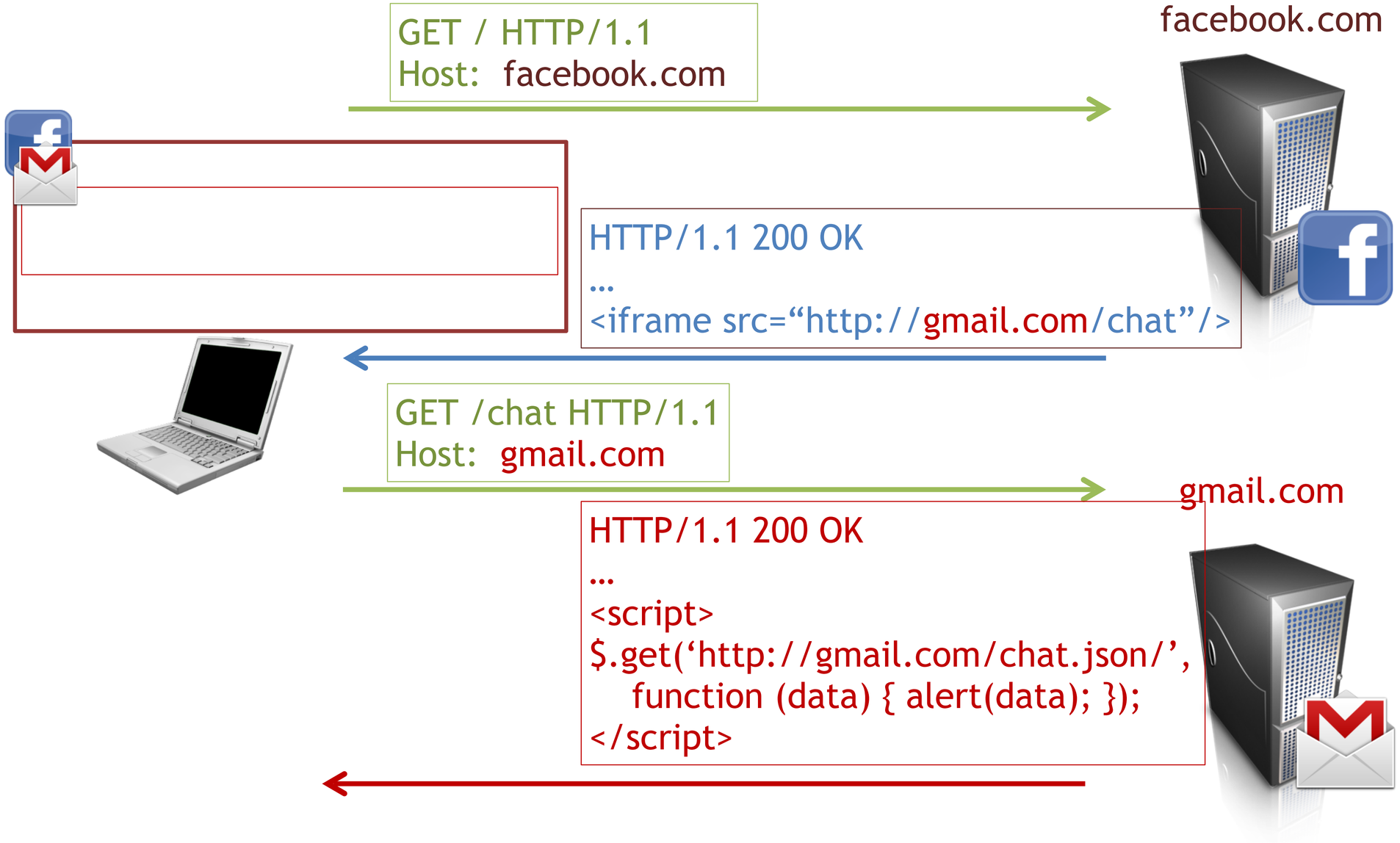
Web Review | Same-Origin Policy (SOP)



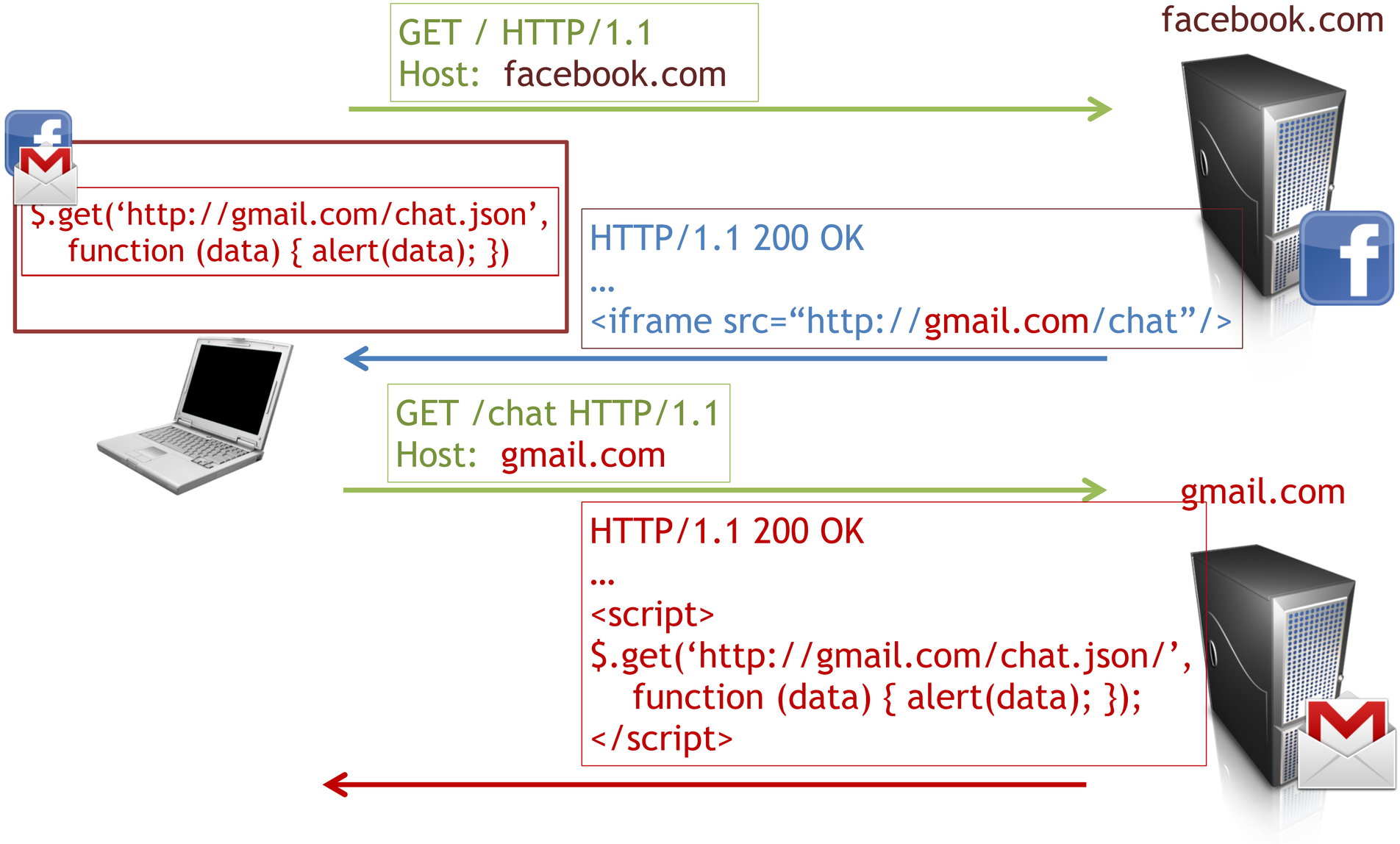
Web Review | Same-Origin Policy (SOP)



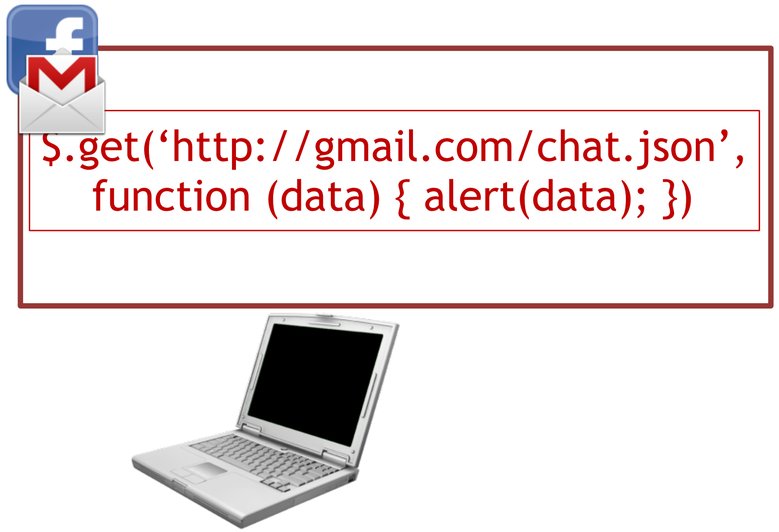
Web Review | Same-Origin Policy (SOP)



Web Review | Same-Origin Policy (SOP)

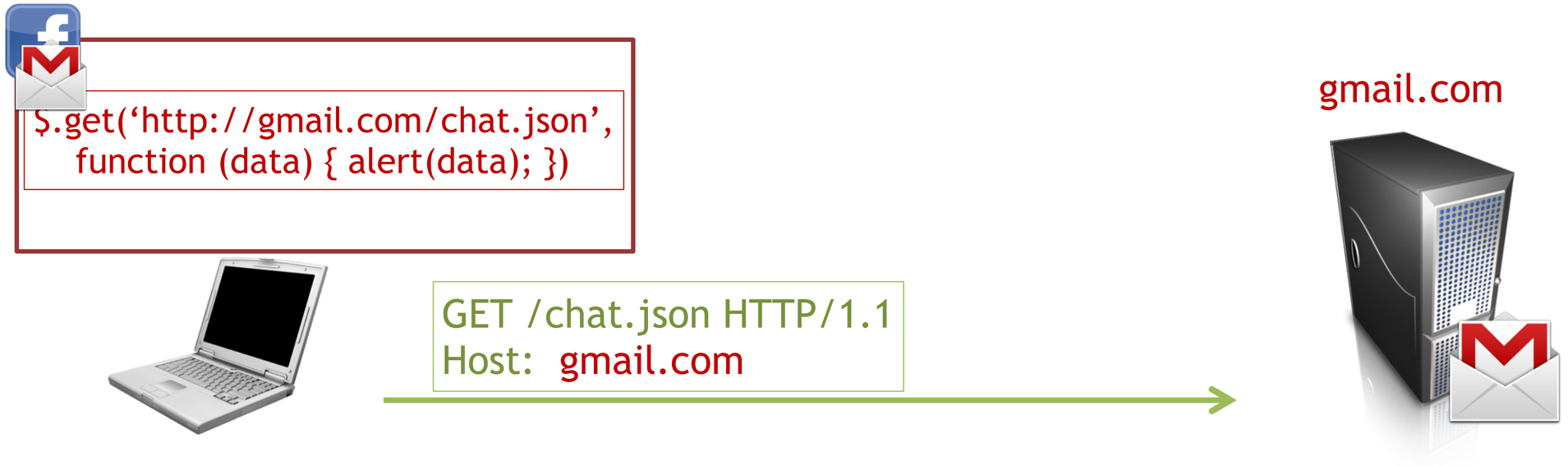


Web Review | Same-Origin Policy (SOP)

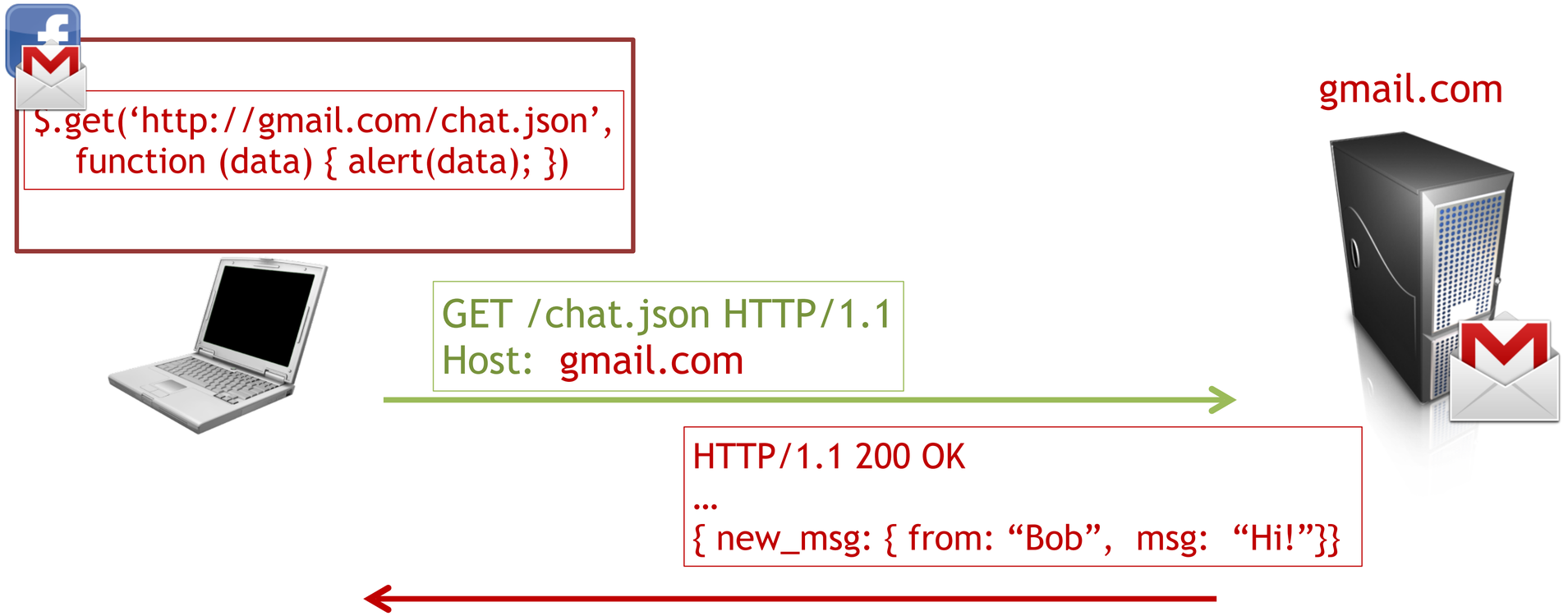


gmail.com

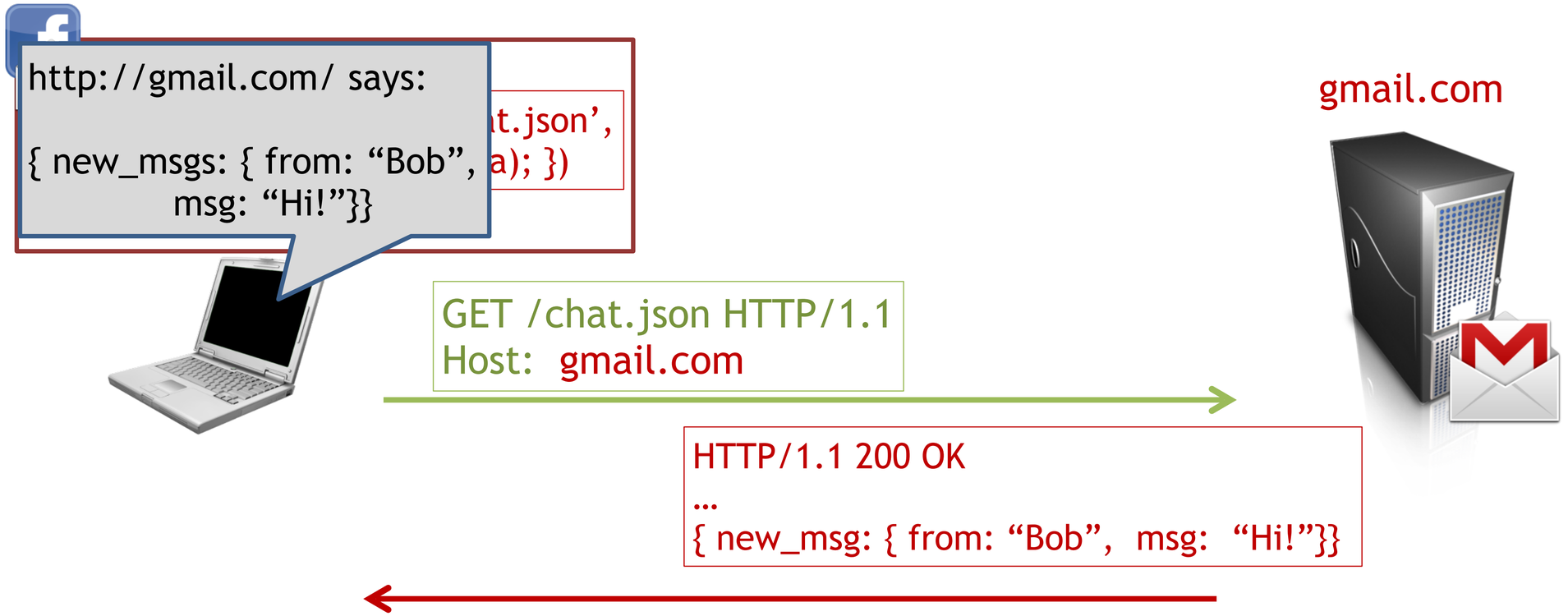
Web Review | Same-Origin Policy (SOP)



Web Review | Same-Origin Policy (SOP)



Web Review | Same-Origin Policy (SOP)



Beware finer-grained origins

- Not all web features respect the SOP
- Example: Cookies can include a path
  - In order to read a cookie with a path, the path of the document's URL must extend the path of the cookie

    Cookie path: /a/b/c

    Document path: /a/b <- Cannot read the cookie

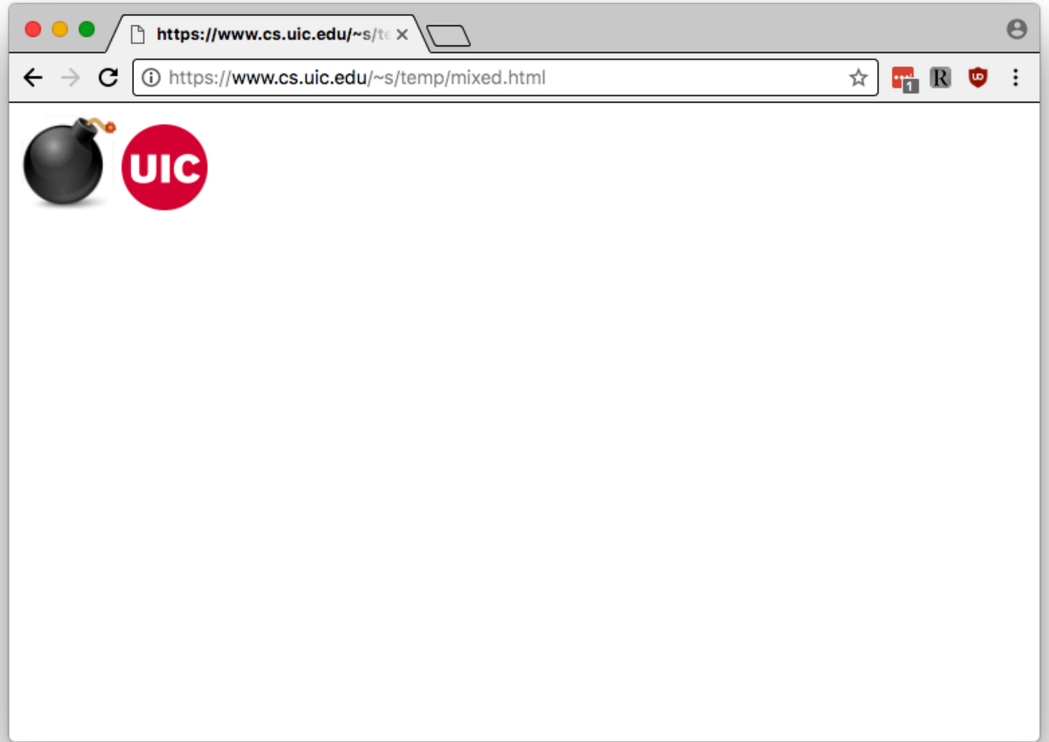
    /a/b/c/d <- Can read the cookie
  - This is "finer-grained" than the standard SOP
  - Is this a problem?

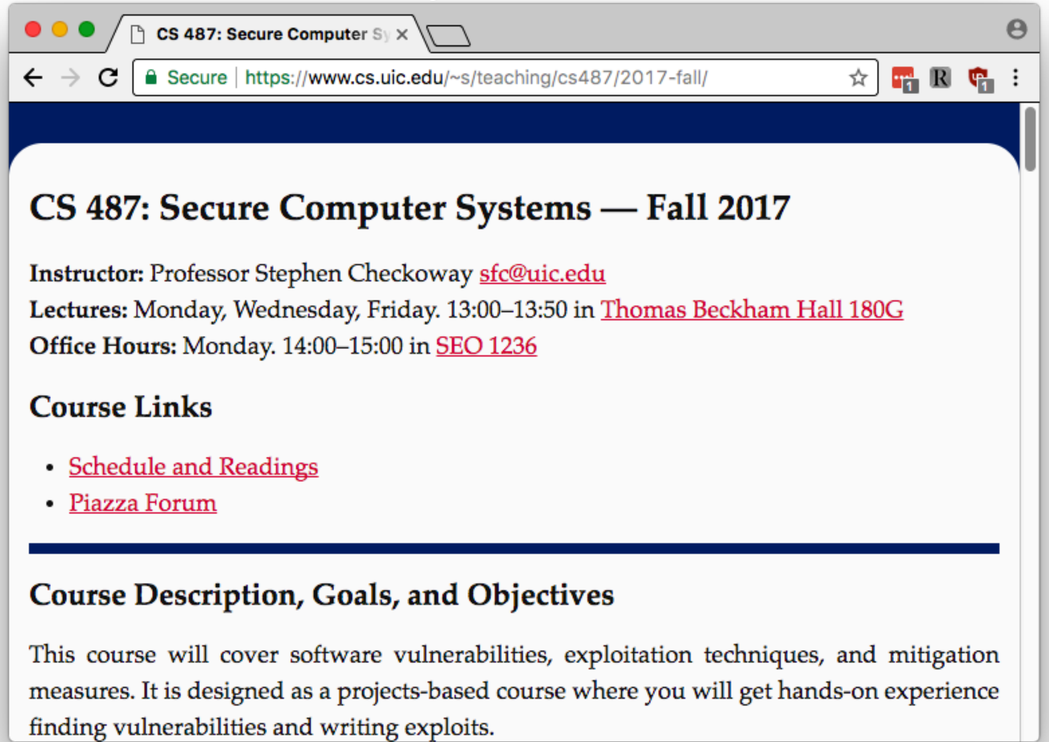
Cookie paths example cont.

- Since documents in the same page can script each other, page /a/b can still read the cookie:
  - Create an iframe with src set to /a/b/c/d (where this the path of some real document that can read the cookie value)
  - Since the iframe is in the same origin, page /a/b can inject a script element into the iframe's document
  - The injected script reads the cookie value and sends it back to the containing page
- Cookie paths should **not** be used as a security boundary

Mixed content

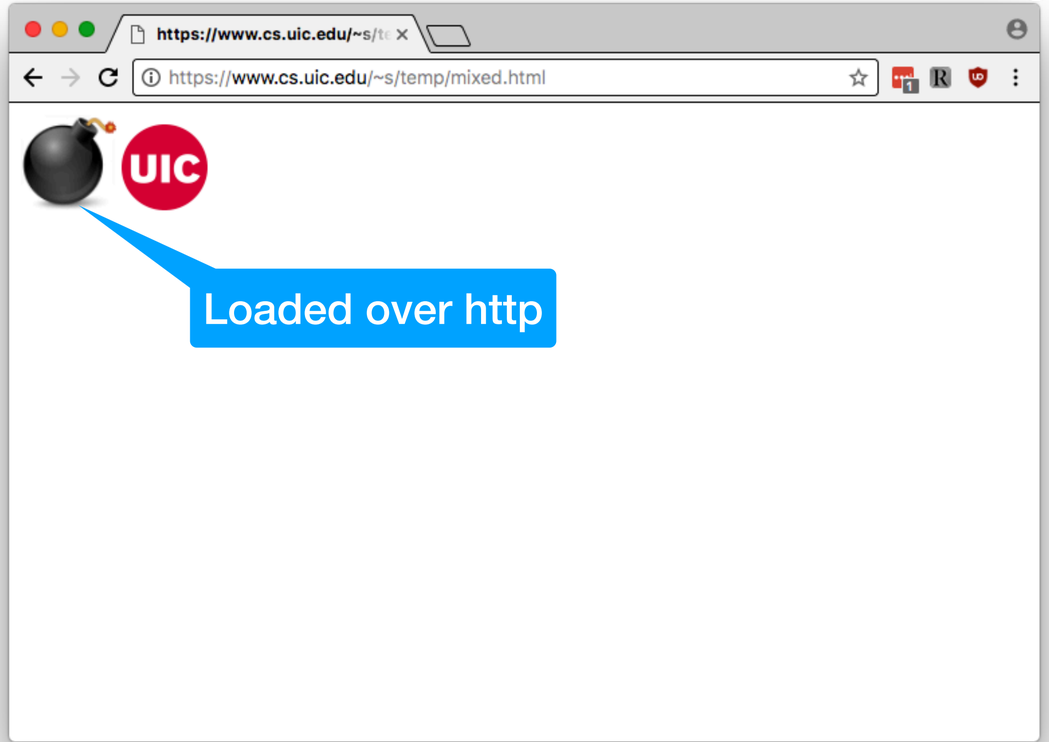
- Documents can contain elements loaded over both http and https
- Browsers indicate that this is insecure (by not displaying a lock icon) on the page with mixed content
- Other documents in the same origin are not similarly marked as insecure

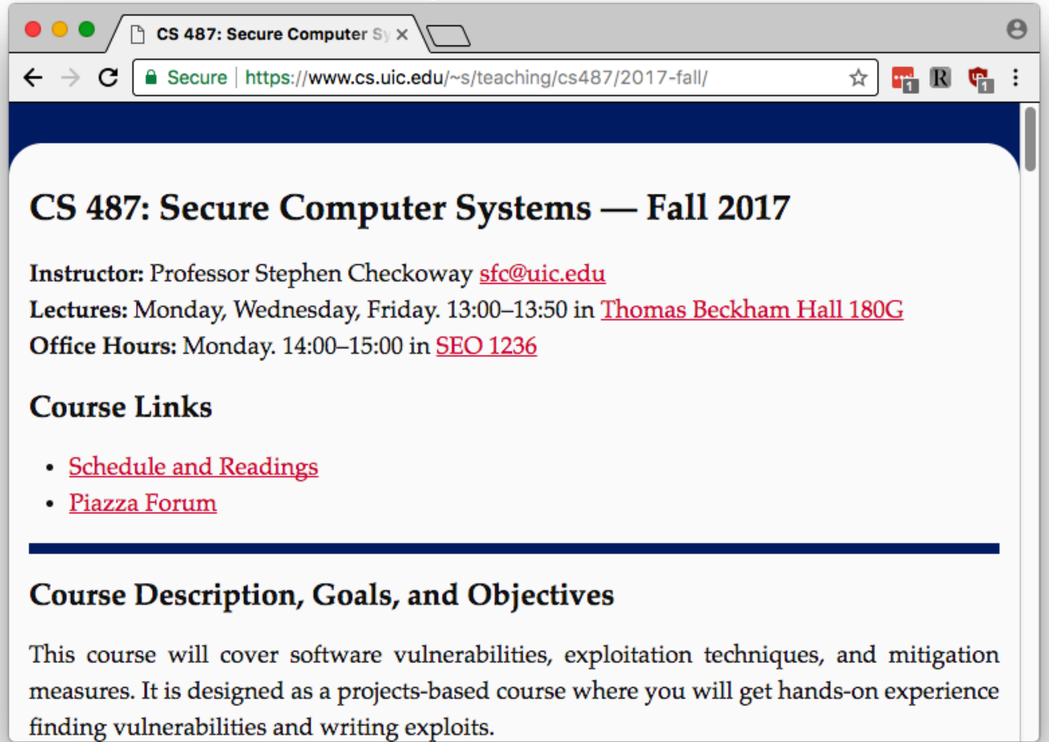




Mixed content

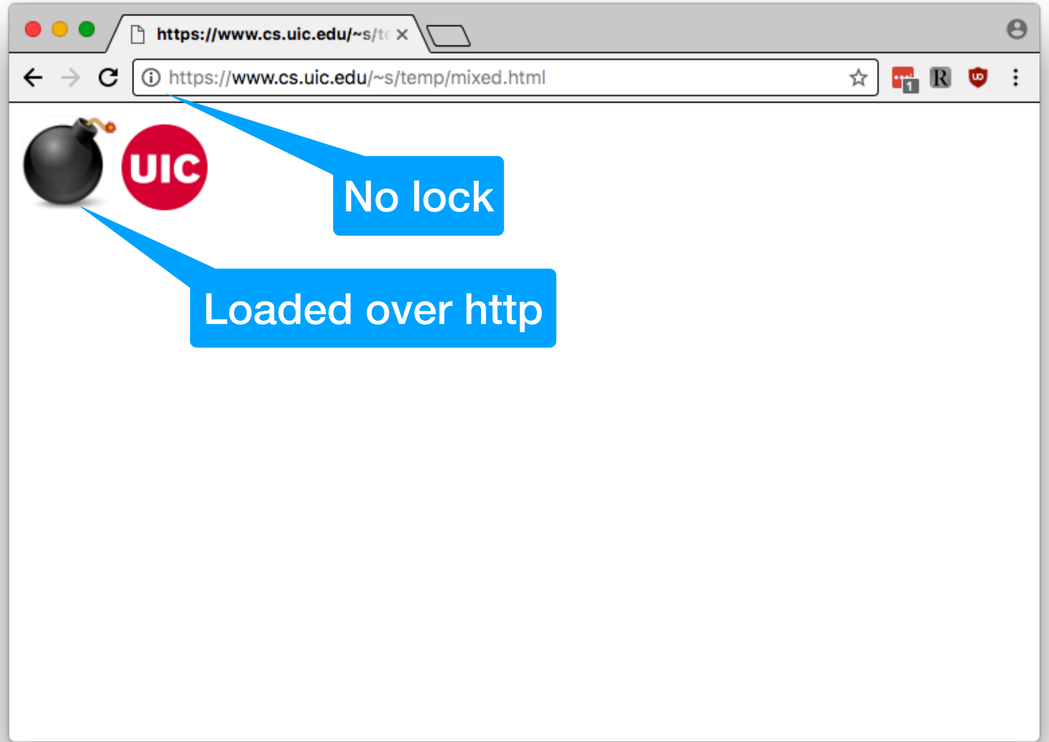
- Documents can contain elements loaded over both http and https
- Browsers indicate that this is insecure (by not displaying a lock icon) on the page with mixed content
- Other documents in the same origin are not similarly marked as insecure

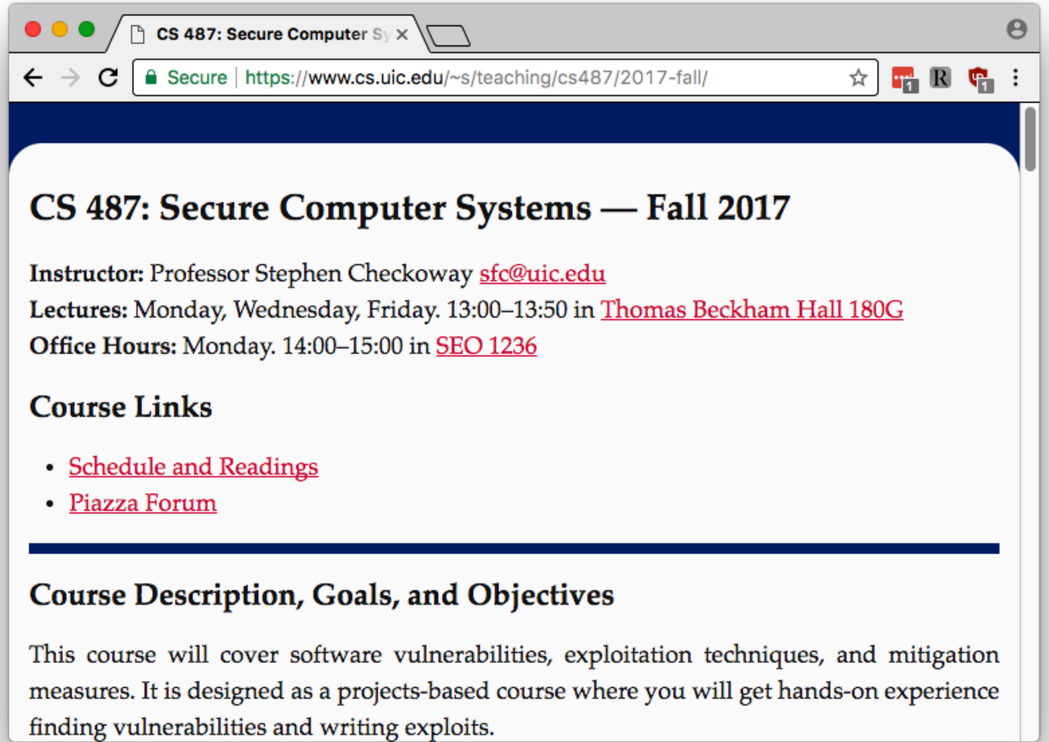




Mixed content

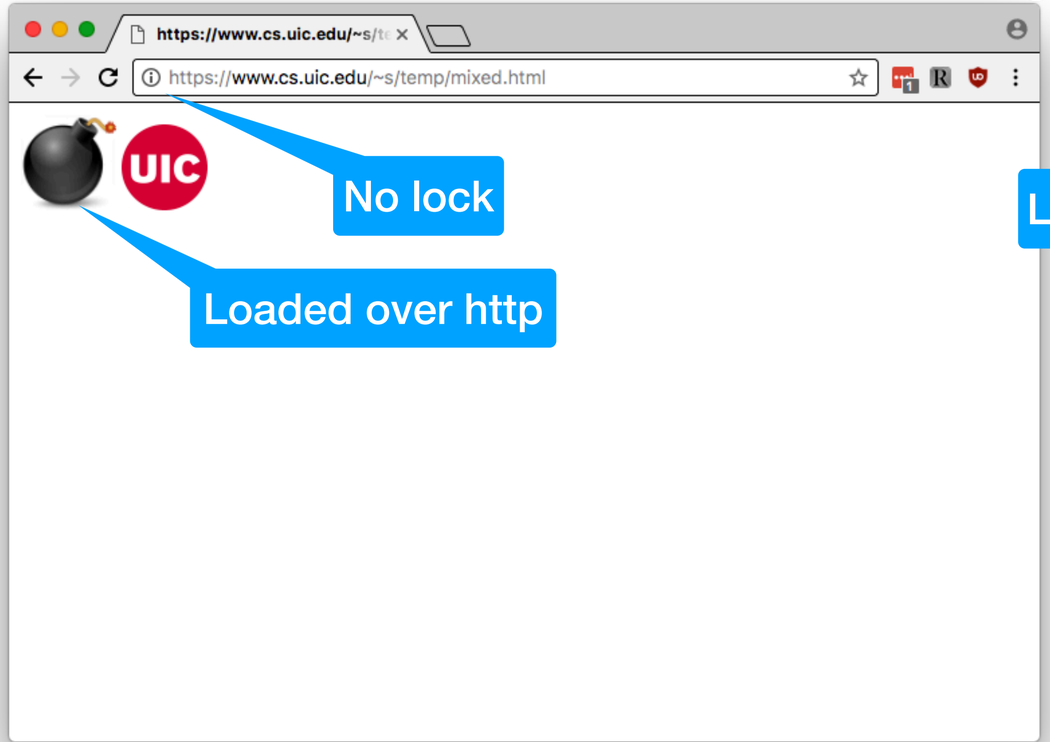
- Documents can contain elements loaded over both http and https
- Browsers indicate that this is insecure (by not displaying a lock icon) on the page with mixed content
- Other documents in the same origin are not similarly marked as insecure

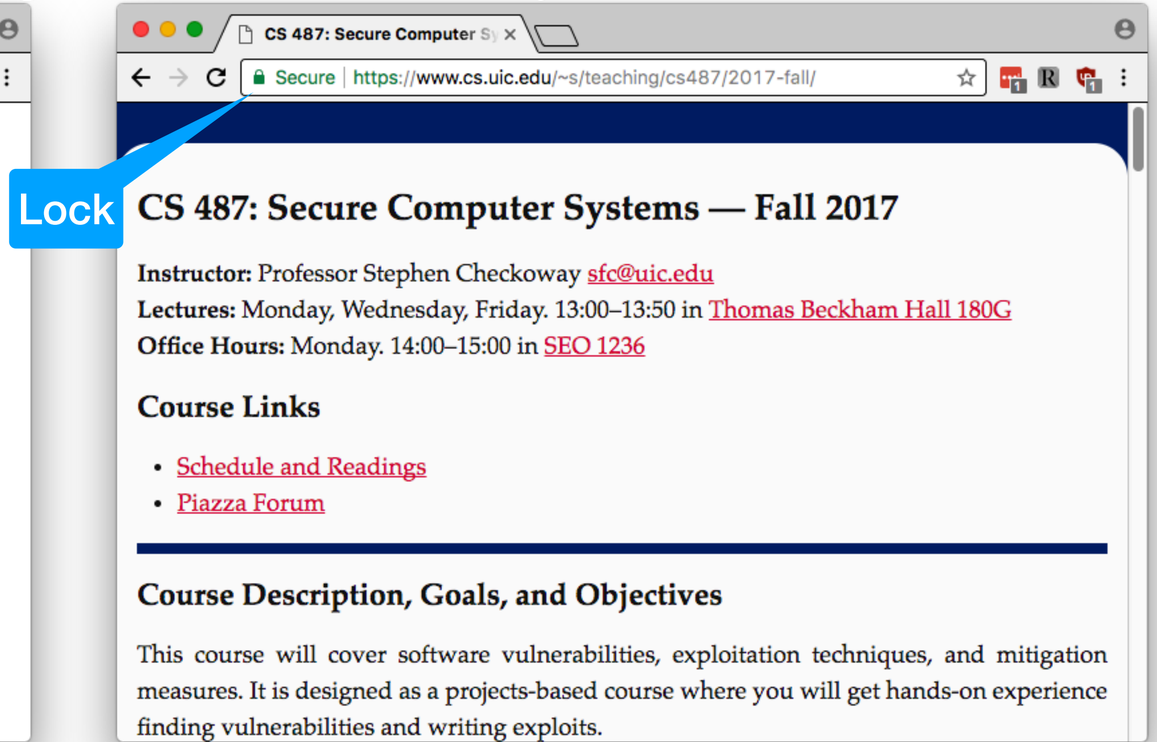




Mixed content

- Documents can contain elements loaded over both http and https
- Browsers indicate that this is insecure (by not displaying a lock icon) on the page with mixed content
- Other documents in the same origin are not similarly marked as insecure





Mixed content

Mixed content

- Is that an issue?

Mixed content

- Is that an issue?
- Yes, script injected from the element loaded over http could script other pages in the same origin…

Mixed content

- Is that an issue?
- Yes, script injected from the element loaded over http could script other pages in the same origin…
- …except modern browsers explicitly do not run scripts loaded via http in an https page, so not really any more

Cross-origin attacks

Setup

- Web attacker
  - Controls one or more domains (e.g., attacker.com, evil.com)
  - Can cause the victim to browse to a page serving JavaScript at one of these domains
- Victim is logged in to bank.com (or any other interesting site)

Quick review

Quick review

- Can the attacker's JavaScript read bank.com?

Quick review

- Can the attacker's JavaScript read bank.com?
  - No. Same origin policy

Quick review

- Can the attacker's JavaScript read bank.com?
  - No. Same origin policy
- The attacker's script uses XMLHttpRequest("https://bank.com") which causes the browser to fetch https://bank.com and return its contents. Can the attacker's script read the response?

Quick review

- Can the attacker's JavaScript read bank.com?
  - No. Same origin policy
- The attacker's script uses XMLHttpRequest("https://bank.com") which causes the browser to fetch https://bank.com and return its contents. Can the attacker's script read the response?
  - No. Same origin policy

Quick review

- Can the attacker's JavaScript read bank.com?
  - No. Same origin policy
- The attacker's script uses XMLHttpRequest("https://bank.com") which causes the browser to fetch https://bank.com and return its contents. Can the attacker's script read the response?
  - No. Same origin policy
- Can the attacker's script use XMLHttpRequest("https://bank.com/transfer?from=victim&to=attacker")?

Quick review

- Can the attacker's JavaScript read bank.com?
  - No. Same origin policy
- The attacker's script uses XMLHttpRequest("https://bank.com") which causes the browser to fetch https://bank.com and return its contents. Can the attacker's script read the response?
  - No. Same origin policy
- Can the attacker's script use XMLHttpRequest("https://bank.com/transfer?from=victim&to=attacker")?
  - Yes! Same origin policy doesn't prevent this. The script just cannot read the response

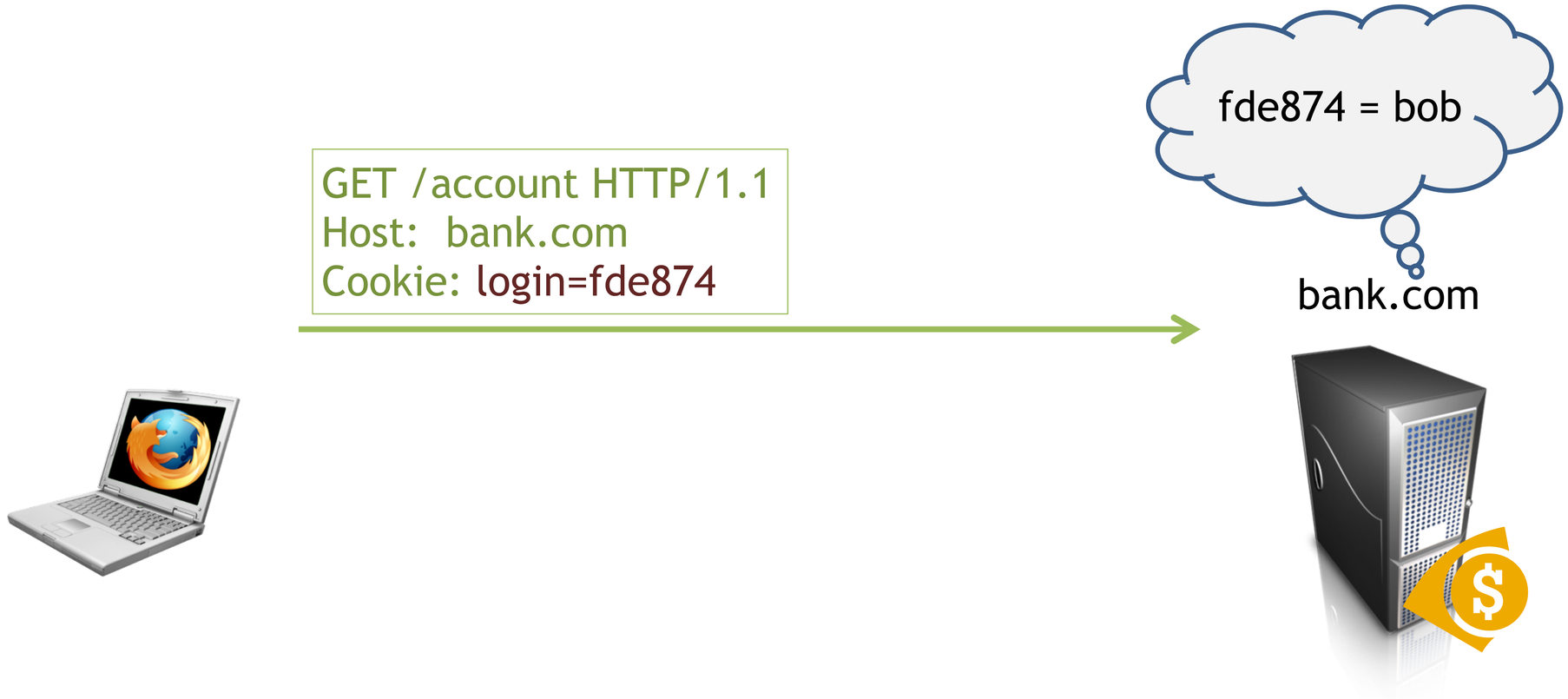
Cross-site request forgery (CSRF)

- The attacker's site instructs the victim's browser to make a request to an honest site (e.g., using XMLHttpRequest or even just an enticing link)
- An XMLHttpRequest allows both GET and POST
- The browser sends all relevant cookies, including any sessions cookies identifying the logged in victim
- From the server's perspective, it looks exactly like a normal request from the victim's browser

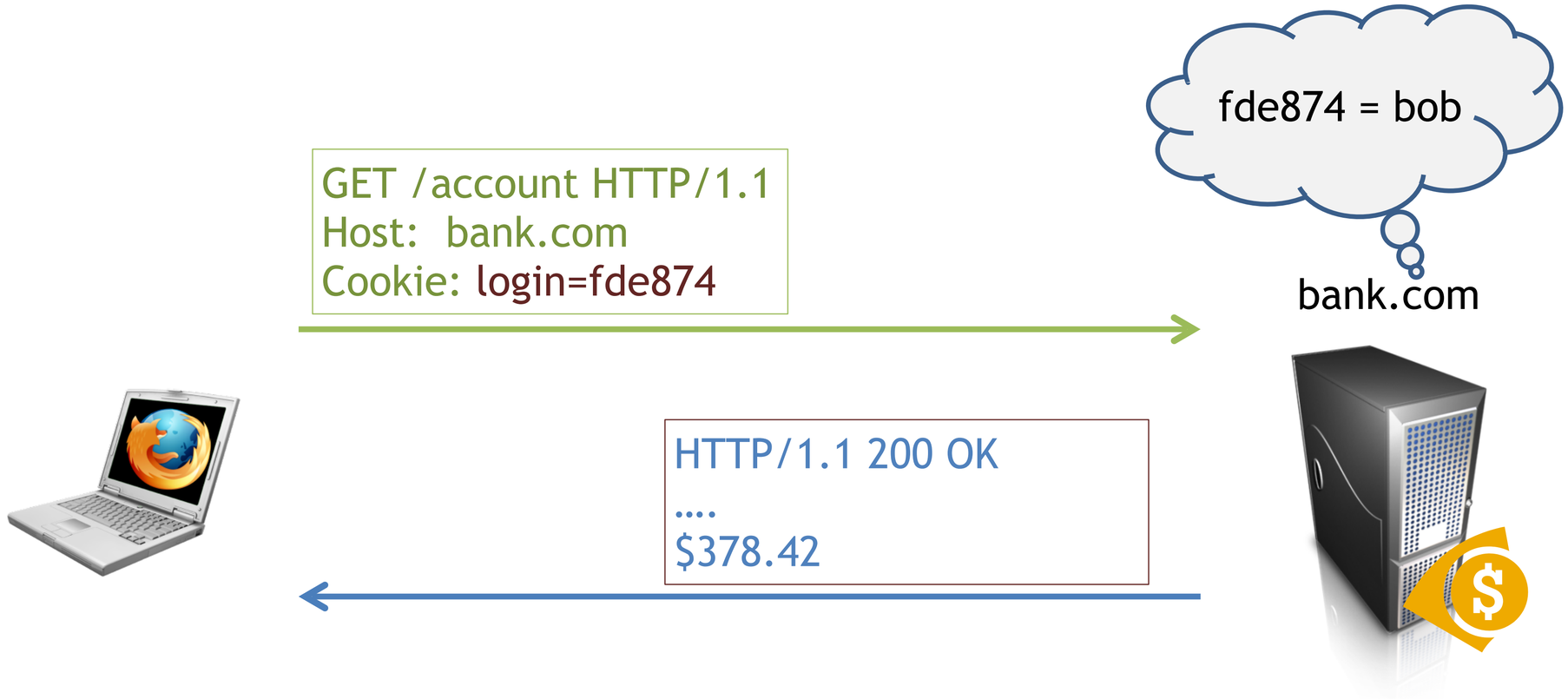
Cross-site Request Forgery (CSRF)

Cross-site Request Forgery (CSRF)

Cross-site Request Forgery (CSRF)

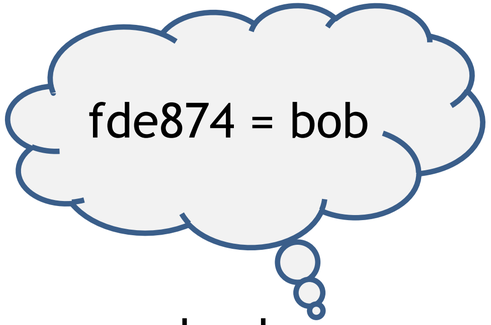


Cross-site Request Forgery (CSRF)



Cross-site Request Forgery (CSRF)

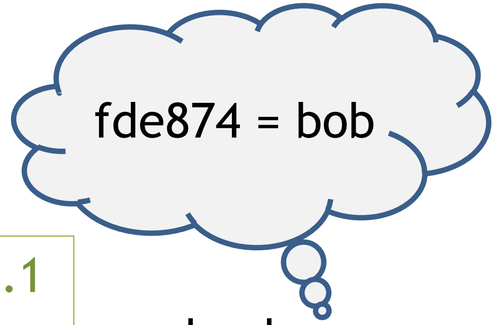
Click me!!!
http://bank.com/transfer?to=badguy&amt=100



bank.com

Cross-site Request Forgery (CSRF)

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

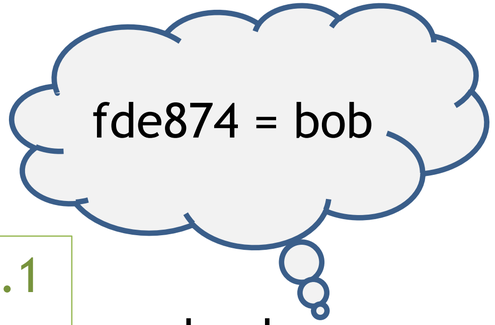



```
GET /transfer?to=badguy&amt=100 HTTP/1.1
Host:  bank.com
Cookie: login=fde874
```

bank.com

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

Why not make requests directly?

- Use the browser's state: The browser sends cookies, client certificates, basic auth credentials in the request
- Set the browser's state: The browser parses and acts on responses, even if the JavaScript cannot read the responses
- Leverage the browser's network connectivity: The browser can connect to servers the malicious site cannot reach (e.g., those behind a firewall)

CSRF Defenses

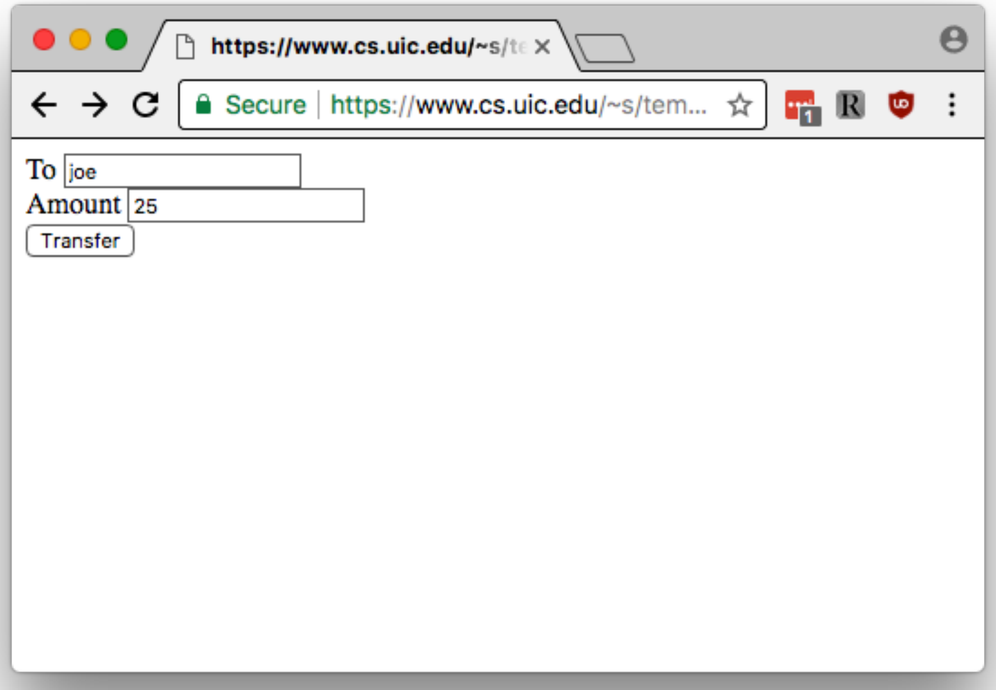
- Need to “authenticate” each user action originates from the legitimate site
- Only needed for actions that change state (E.g., POST but not GET)
  - Why isn't it needed for GET?
- Possibilities
  - Secret token
  - HTTP Referer header (yes, Referer not Referrer, it was misspelled)
  - Custom HTTP header
  - Origin header

Secret token

- Hidden form field with the token value
- The token should be unpredictable to attackers
- Random numbers work, but then need to be stored server side
- Using crypto, we can do better (HMAC)
- The token should be sent along with every POST and checked by the server
- This is a hassle for dynamically-generated content since it needs to include the tokens
- What prevents malicious script from fetching the page (e.g., with XMLHttpRequest), reading the token, and then sending a response with the token?

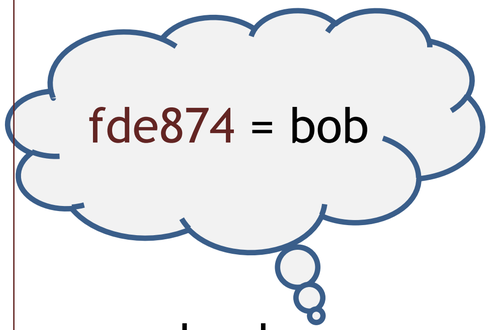
Example CSRF token

```
<form action="/transfer" method="post">
  <input type="hidden" name="token" value="8d64">
  To <input type="text" name="to"><br>
  Amount <input type="text" name="amount"><br>
  <input type="submit" value="Transfer">
</form>
```



CSRF Defenses

```
HTTP/1.1 200 OK
Set-Cookie: login=fde874

<form action="/transfer" method="post">
  <input type="hidden" name="token" value="8d64">
...
```

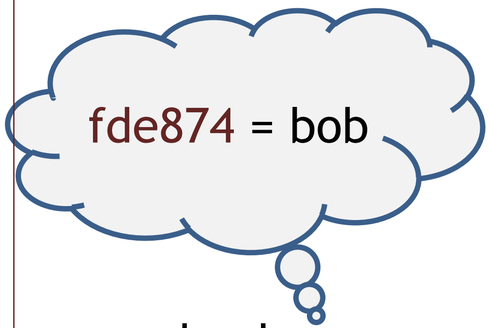


bank.com

This is not actually how POST data is encoded and sent, but the principle is the same

CSRF Defenses

```
HTTP/1.1 200 OK
Set-Cookie: login=fde874

<form action="/transfer" method="post">
  <input type="hidden" name="token" value="8d64">
...
```



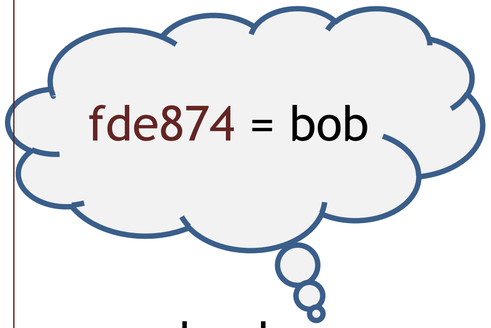
bank.com

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

This is not actually how POST data is encoded and sent, but the principle is the same

CSRF Defenses

```
HTTP/1.1 200 OK
Set-Cookie: login=fde874

<form action="/transfer" method="post">
  <input type="hidden" name="token" value="8d64">
...
```



bank.com

```
POST /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
```

This is not actually how POST data is encoded and sent, but the principle is the same

Referer header

- Sent by the browser and contains the URL of the page containing the link that was clicked or form that was submitted
- Easy to handle server side, just check that the request comes with the correct Referer header
- However, it is frequently stripped by the browser or middle boxes (for privacy reasons)
- It's stripped less often over HTTPS since middle boxes can't modify content

Custom HTTP header

- XMLHttpRequest supports adding custom headers but browsers disallow them on cross-origin requests
- Server can check that the custom header is present

Origin header

- The evolution of the Referer header but only contains the scheme, host, and port, not the full URL
- As with the Referer and custom headers, the server checks the Origin is correct
- Supported by all major browsers
- Unlike custom headers, it's part of the standard

Cross-site scripting (XSS)

- XSS is a method for attackers to embed content (often JavaScript) in another page
- Two basic types
  - Reflected XSS
  - Stored XSS

Reflected XSS

- Web attacker causes the victim to click a link to a legitimate page where the link contains some script
- The server includes the script verbatim in the legitimate page which is sent back to the browser
- The browser interprets it as script coming from the legitimate origin

Cross-Site Scripting (XSS)

```
<?php

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

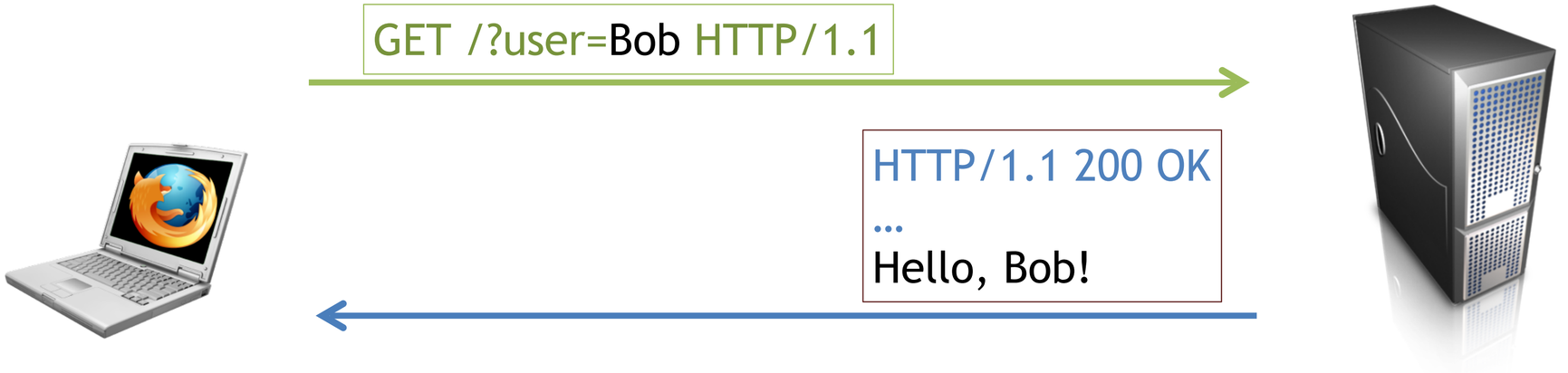
Cross-Site Scripting (XSS)

```
<?php

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

GET /?user=Bob HTTP/1.1

Cross-Site Scripting (XSS)

```
<?php

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



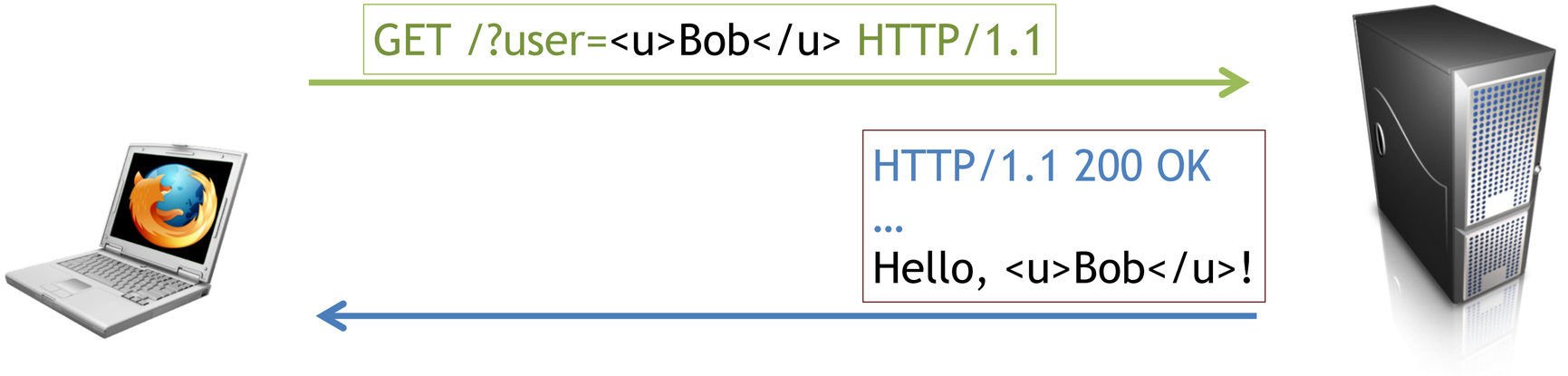
Cross-Site Scripting (XSS)

```
<?php

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

GET /?user=<u>Bob</u> HTTP/1.1

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

Cross-Site Scripting (XSS)

```
<?php

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

Cross-Site Scripting (XSS)

```
<?php

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

Cross-Site Scripting (XSS)

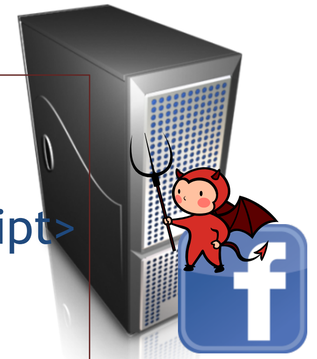
```
<?php

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

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

Cross-Site Scripting (XSS) Attack

GET / HTTP/1.1
Host: facebook.com

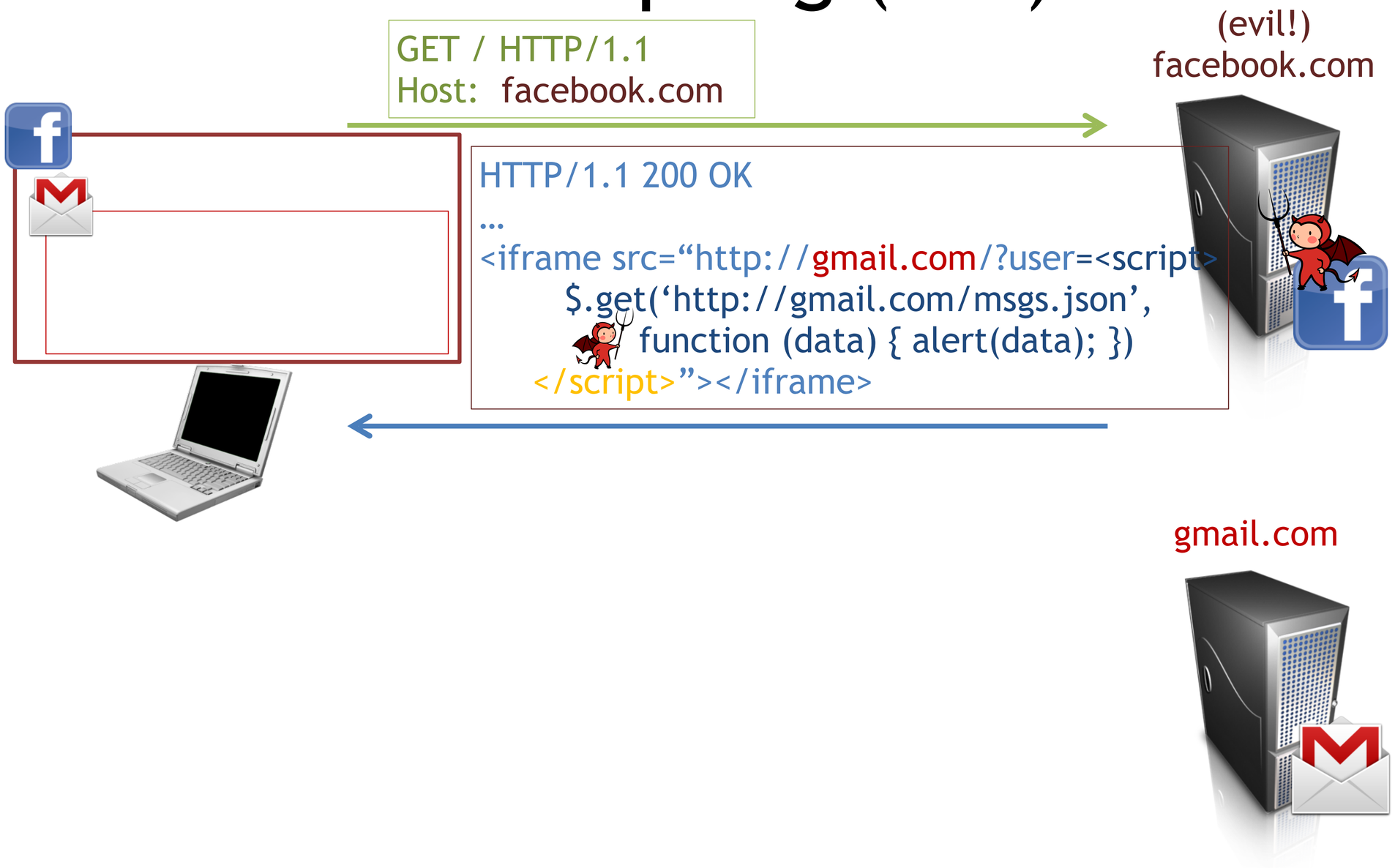
(evil!)
facebook.com



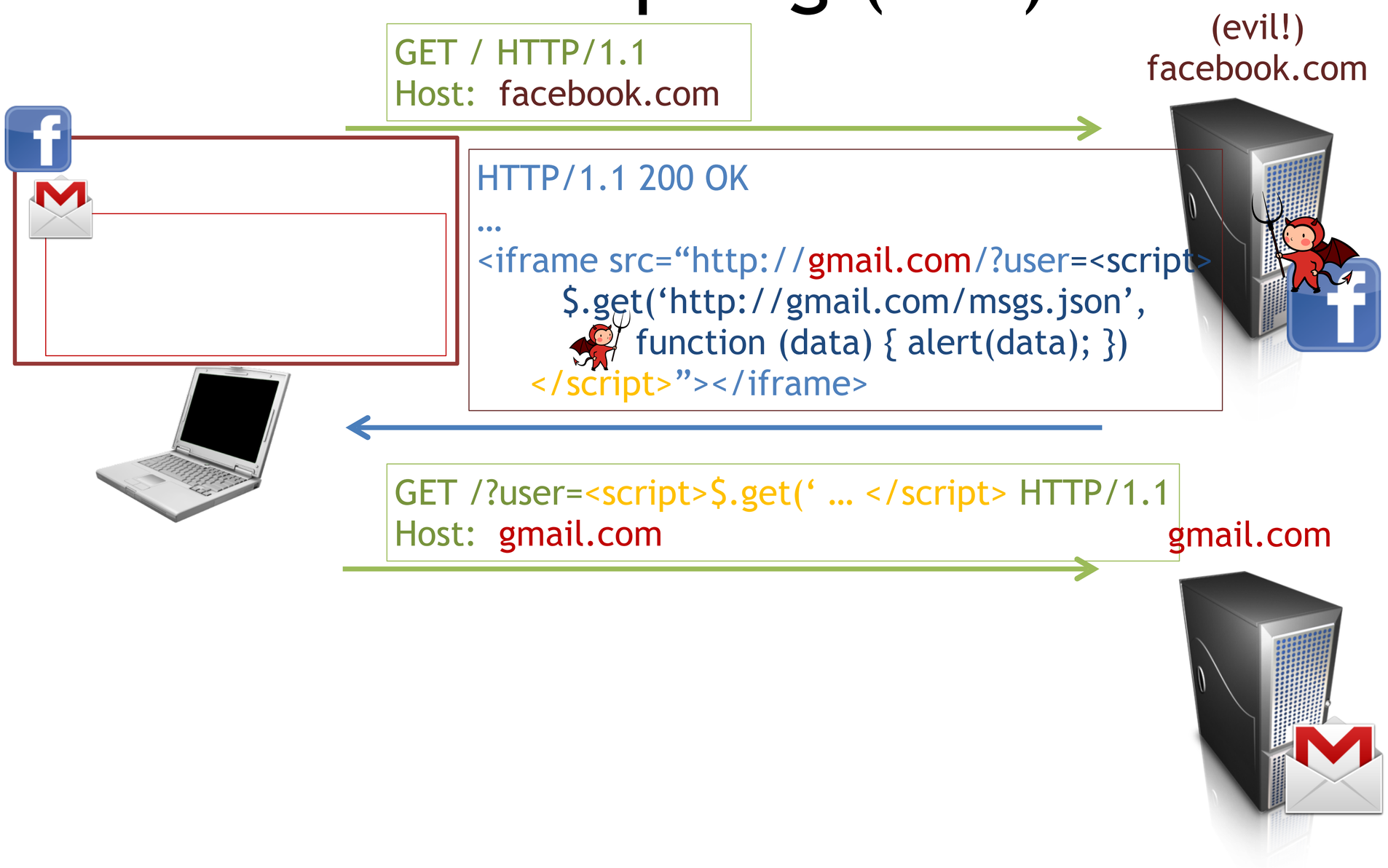
HTTP/1.1 200 OK
…
<iframe src="http://gmail.com/?user=<script>
$.get('http://gmail.com/msgs.json',
function (data) { alert(data); })
</script>"></iframe>

gmail.com

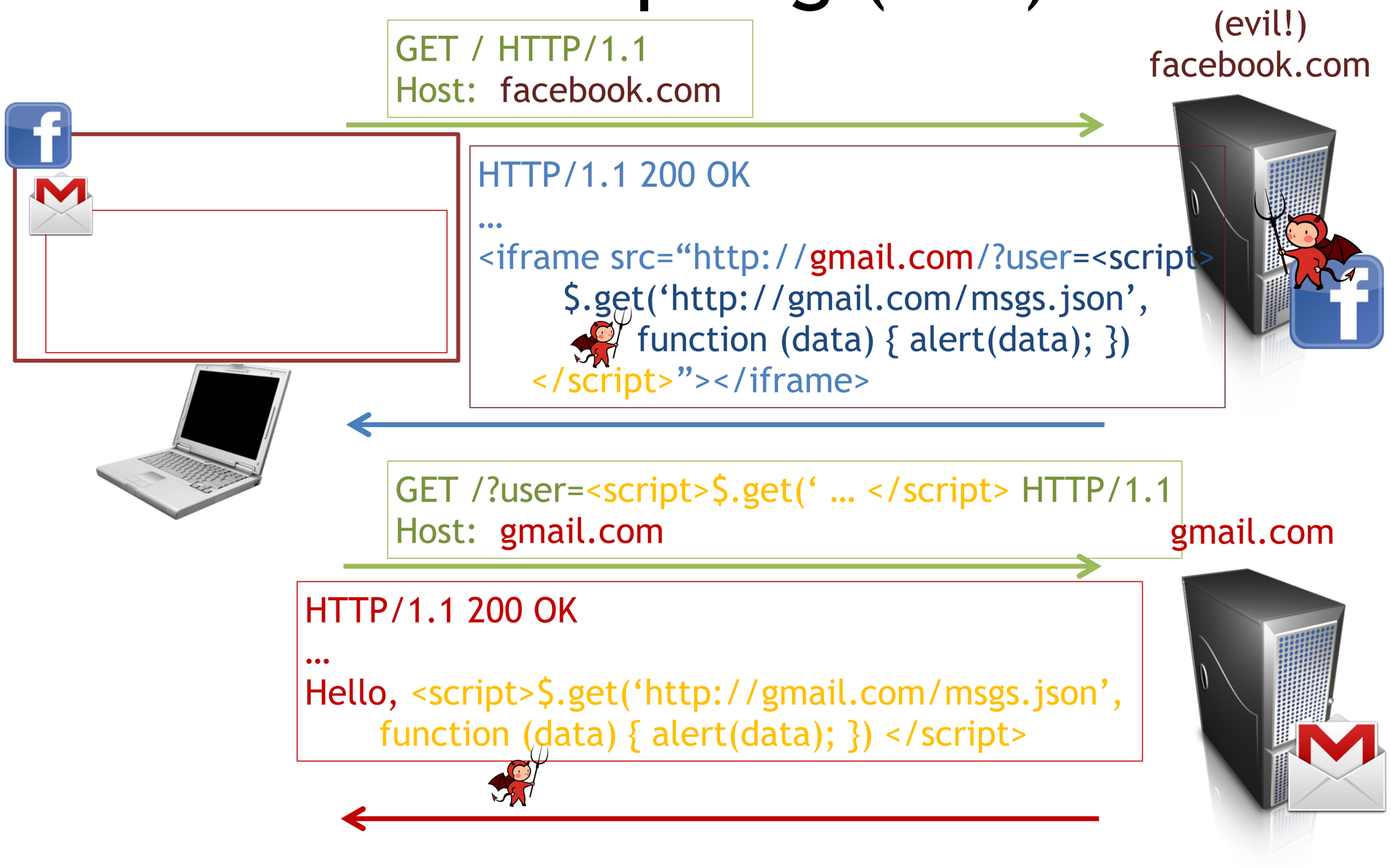
Cross-Site Scripting (XSS) Attack



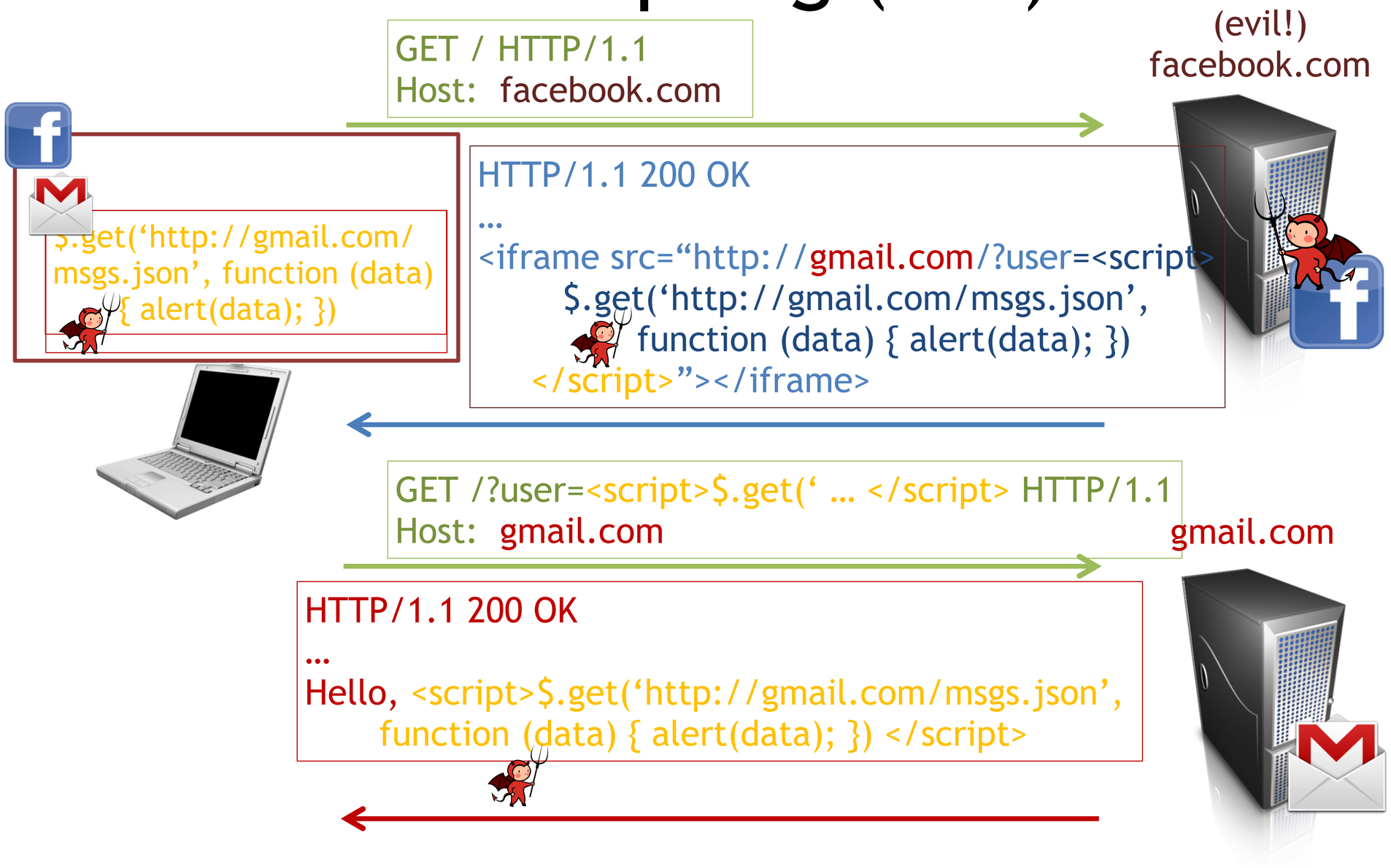
Cross-Site Scripting (XSS) Attack



Cross-Site Scripting (XSS) Attack



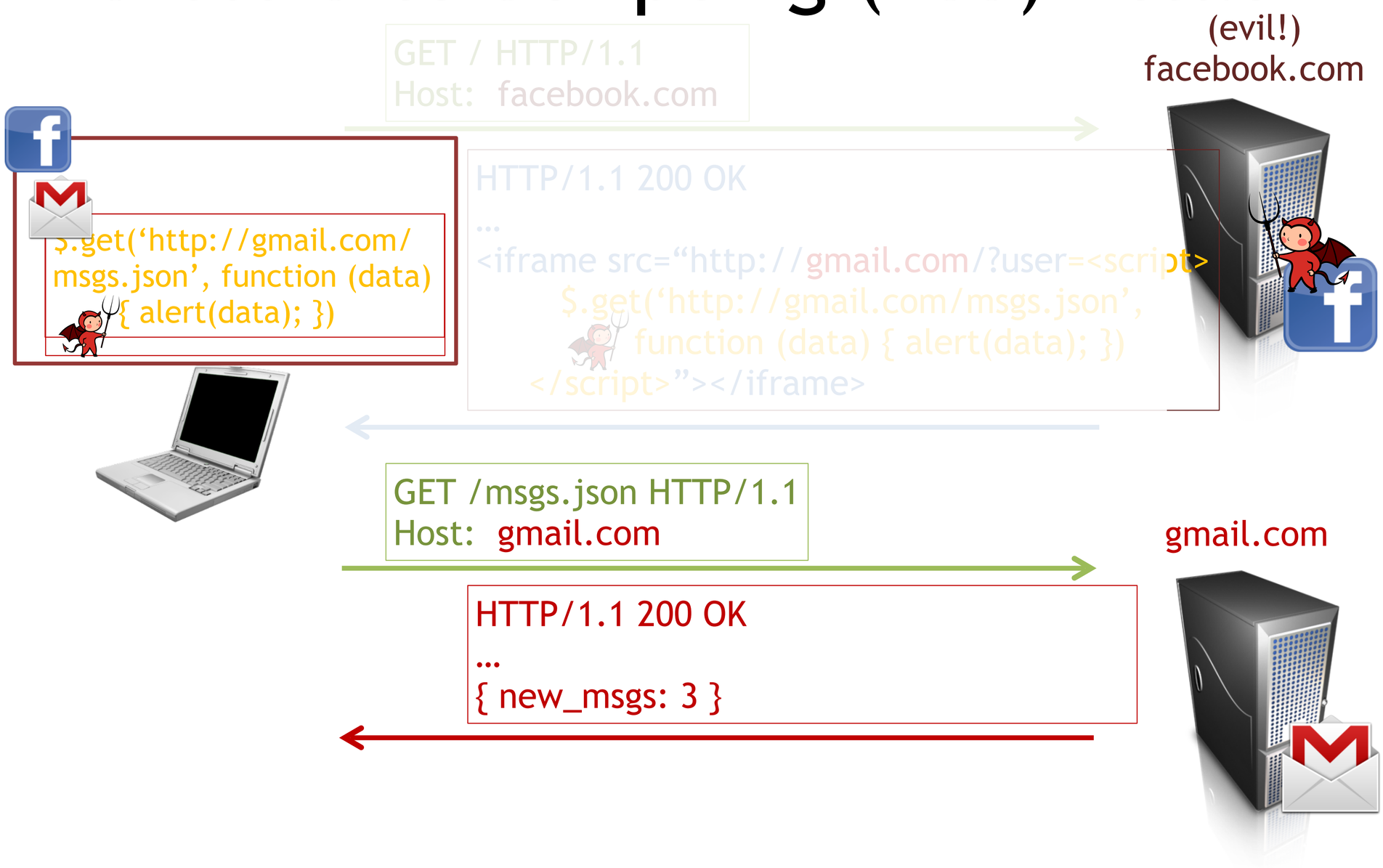
Cross-Site Scripting (XSS) Attack



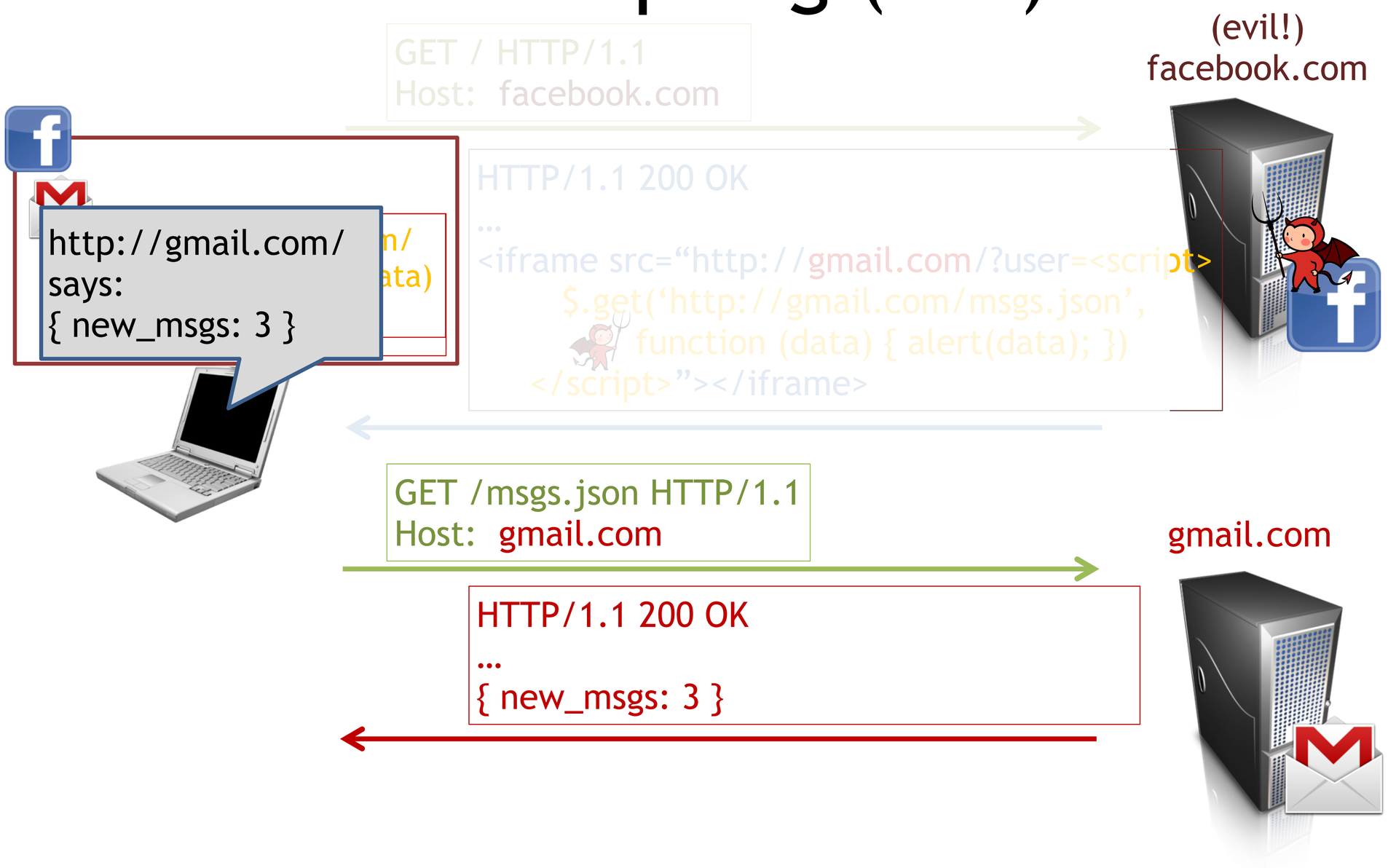
Cross-Site Scripting (XSS) Attack

Cross-Site Scripting (XSS) Attack

Cross-Site Scripting (XSS) Attack



Cross-Site Scripting (XSS) Attack



XSS capabilities

- Execute arbitrary scripts in the context (i.e., Origin) of the vulnerable server
- Manipulate the DOM of the vulnerable page
- Submit/read forms (including any CSRF tokens)
- Read cookies
- Install event handlers
- In essence, anything that JavaScript can do!

Stored XSS

- Some web sites serve user-generated content but fail to properly sanitize the user's input
- The attacker POSTs some HTML with JavaScript on the page (e.g., a post on a forum)
- When victims visit the page, the attacker's script is served and the browser (not realizing it came from the attacker) executes it as normal
- The script can do anything JavaScript can do!

Example: Samy worm

- Myspace allowed users to insert HTML in their profiles, but disallowed <script>
- Some browsers support JavaScript inside CSS

- Myspace disallowed the word javascript but Internet Explorer (at the time anyway) allowed
  java
  script
  which bypassed their filter
- Other filters were bypassed by using eval()

Example: Samy worm

- Samy Kamkar discovered this and put some script in his profile
- When his page was viewed by a victim, the victim's browser would run the script which would modify the victim's profile to include "but most of all, samy is my hero" as well as the script itself
- Within 20 hours, over one million people's profiles were infected
- Myspace had to go offline to fix the problem
- Kamkar pleaded guilty to a felony and got 3 years probation, a fine, and restricted computer use (now he makes cool YouTube videos!)