



Web Security

SQL Injection, CSRF, XSS



ECEN 5032
Sep 16, 2016

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 | Cookies

POST /login HTTP/1.1

Host: gmail.com

user=alice&pass=s3cre7



HTTP/1.1 200 OK

Server: gws

Set-Cookie: foo="bar"

Set-Cookie: token="8kFmCe..."

<html>

...

gmail.com



GET / HTTP/1.1

Host: gmail.com

Cookie: foo="bar"; token="8k..."

Ah, it's
alice!



Web Review | AJAX (jQuery style)

GET / HTTP/1.1
Host: gmail.com

http://gmail.com/ says:

{ new_msgs: 3 }

msgs.json',
alert(data) });

HTTP/1.1 200 OK

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

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)

(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)



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



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

gmail.com



HTTP/1.1 200 OK

...

{ new_msg: { from: "Bob", msg: "Hi!" } }



Web Review | Same-Origin Policy (SOP)

GET / HTTP/1.1
Host: facebook.com

facebook.com



HTTP/1.1 200 OK

...

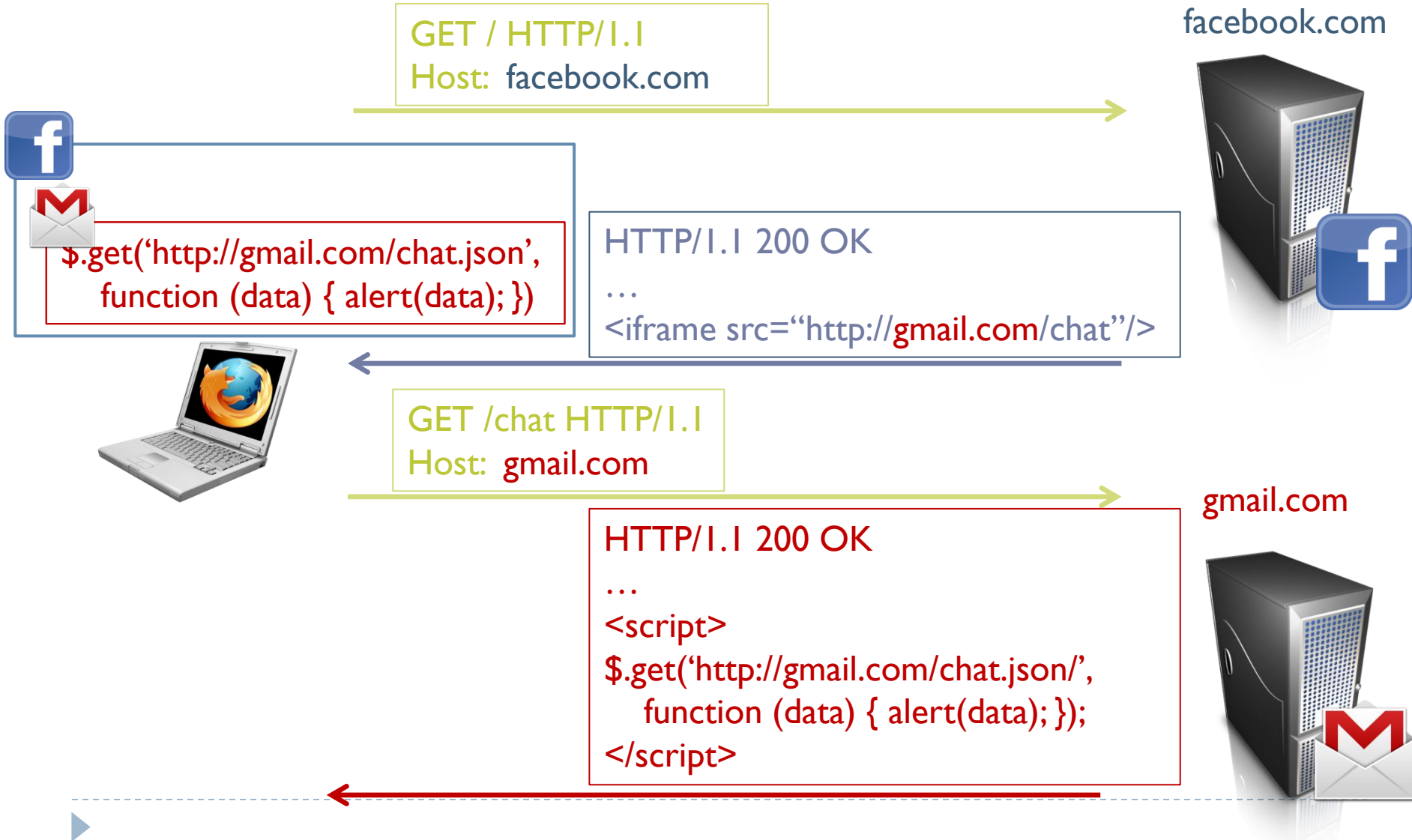
<iframe src="http://gmail.com/chat"/>



gmail.com



Web Review | Same-Origin Policy (SOP)



Web Review | Same-Origin Policy (SOP)

http://gmail.com/ says:

```
{ new_msgs: { from: "Bob",  
              msg: "Hi!" } }
```



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

gmail.com



```
HTTP/1.1 200 OK
```

...

```
{ new_msg: { from: "Bob", msg: "Hi!" } }
```



Code Injection

```
<?php
```

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

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



HTTP/1.1 200 OK

...

Desktop

Documents

Music

Pictures

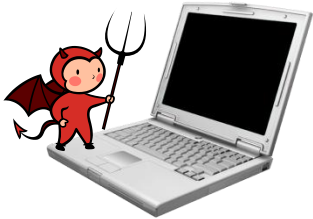


Code Injection

```
<?php
```

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

```
GET /?path=$(rm -rf /) HTTP/1.1
```



```
<?php
```

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



Code Injection

- ▶ **Confusing Data and Code**

- ▶ Programmer thought user would supply data, but instead got (and unintentionally executed) code

```
<?php
```

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



- ▶ **Common and dangerous class of vulnerabilities**

- ▶ Shell Injection
- ▶ SQL Injection
- ▶ Cross-Site Scripting (XSS)
- ▶ Control-flow Hijacking (Buffer overflows)



SQL

- ▶ Structured **Query** Language

- ▶ Language to ask (“query”) databases questions:

- ▶ How many users live in Boulder?

- ```
“SELECT COUNT(*) FROM `users` WHERE location = ‘Boulder’”
```

- ▶ 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 = "Boulder"; DELETE FROM `users` WHERE I='I'"
```

```
SELECT * FROM `users` WHERE location='Boulder';
DELETE FROM `users` WHERE I='I'
```



# SQL Injection Defense

---

- ▶ Make sure **data** gets interpreted as **data**!
  - ▶ Basic approach: escape control characters (single quotes, escaping characters, comment characters)
  - ▶ Better approach: Prepared statements – declare what is data!

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



# 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)

---



# Cross-site Request Forgery (CSRF)



Click me!!!

<http://bank.com/transfer?to=badguy&amt=100>

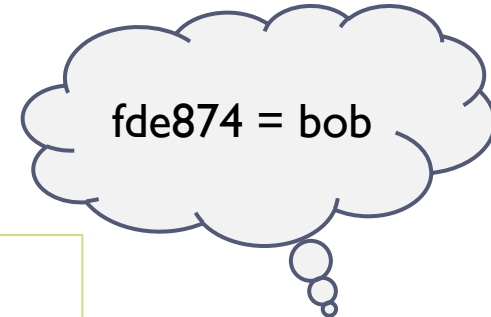


GET /transfer?to=badguy&amt=100 HTTP/1.1  
Host: bank.com  
Cookie: login=fde874

HTTP/1.1 200 OK

....

Transfer complete: -\$100.00



bank.com



# 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>

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

fde874 = bob

bank.com

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



# Cross-Site Scripting (XSS)

---

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

GET /?user=Bob HTTP/1.1



HTTP/1.1 200 OK  
...  
Hello, Bob!



# Cross-Site Scripting (XSS)

---

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

GET /?user=<u>Bob</u> HTTP/I.I



HTTP/I.I 200 OK  
...  
Hello, <u>Bob</u>!



# Cross-Site Scripting (XSS)

http://vuln.com/ says:  
XSS



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>](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

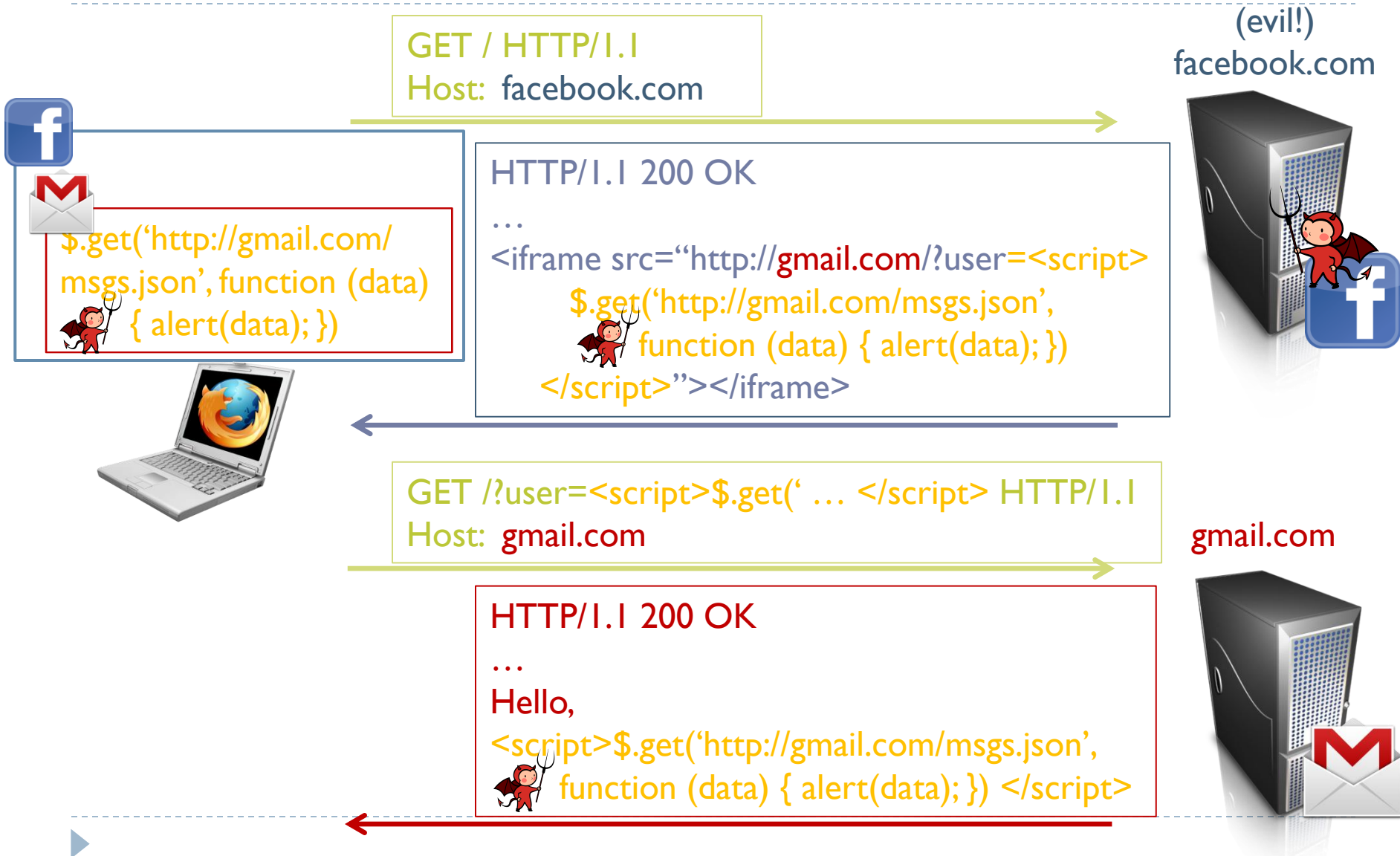
gmail.com



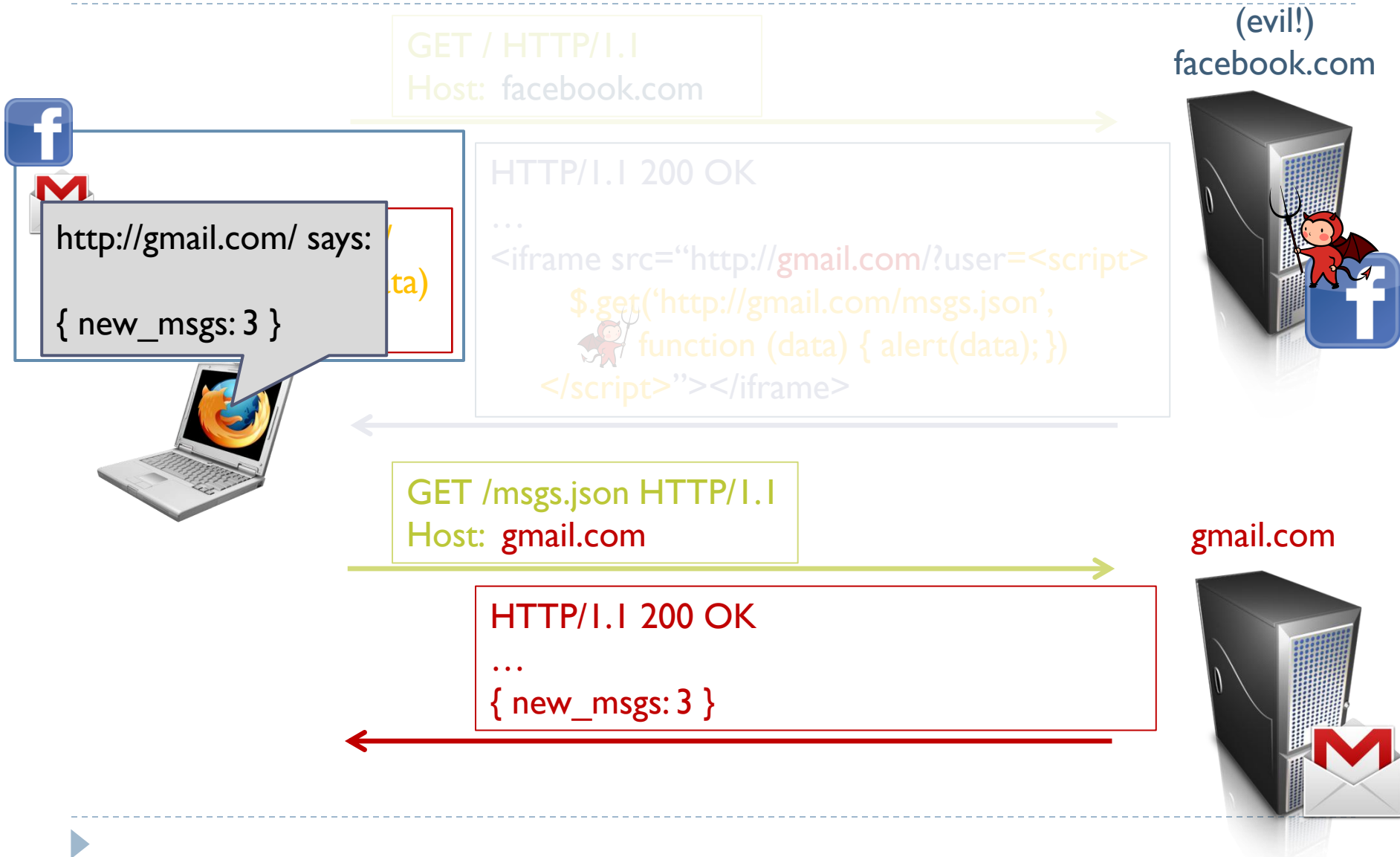
```
HTTP/1.1 200 OK
...
{ new_msgs: 3 }
```



# Cross-Site Scripting (XSS) Attack



# Cross-Site Scripting (XSS) Attack



# Types of XSS

---

## ▶ Reflected XSS

- ▶ `http://vulnerable.com/?q=<script>alert('XSS!')</script>`

## ▶ Stored XSS

- ▶ Attacker stores XSS in database

```
POST /message HTTP/1.1
```

```
Host: vulnerable.com
```

```
to=victim&message=<script>alert('XSS!')</script>
```

- ▶ Victim browses to `http://vulnerable.com/inbox ...`

```
You have 1 new message:

```

```
From: <i>attacker</i>

```

```
Message: <script>alert('XSS')</script>
```





# Cross-Site Scripting (XSS) Attack

---

- ▶ What can an attacker do with an XSS?
  - ▶ Exfiltrate data back to attacker (HTTP POST)
    - ▶ Cookies, CSRF tokens, private information
  - ▶ Perform actions on victims behalf
    - ▶ Any CSRF attacks!
    - ▶ Set cookies to attacker's choosing



# XSS Defenses

---

- ▶ Make sure **data** gets shown as **data**, not executed as code!
- ▶ Escape special characters
  - ▶ Which ones? Depends what context your `$data` is presented
    - Inside an HTML document? `<div>$data</div>`
      - `$data = <script>alert('XSS!')</script>`
    - Inside a tag? `<a href="http://site.com/$data">`
      - `$data = " onmouseover="alert('XSS!')" foo="`
    - Inside Javascript code? `var x = "$data";`
      - `$data = ";alert('XSS!');//`
    - Inside CSS code? `body { color: $data; }`
      - `$data = #000; background:url("javascript:alert('XSS!'))"`
  - ▶ Make sure to escape every last instance!
- ▶ Frameworks can let you declare what's user-controlled data and automatically escape it

