Lecture 13: Web & Cloud Wrap-Up

Professor Adam Bates
CS 461 / ECE 422
Fall 2019

Goals for Today

- Learning Objectives:
  - Understand how cross-site scripting attacks violate the same origin policy (Web Wrap-Up)
  - Explore why containers “do not contain”

- Announcements, etc:
  - **Midterm October 9th, 7pm, 1404 Siebel**
  - Grade distributions for MP1 checkpoints will be released after regrade requests are processed.
  - MP2 Checkpoint #1: **Due Sept 25 at 6pm**
  - MP2 Checkpoint #2: **Due Oct 7 at 6pm**

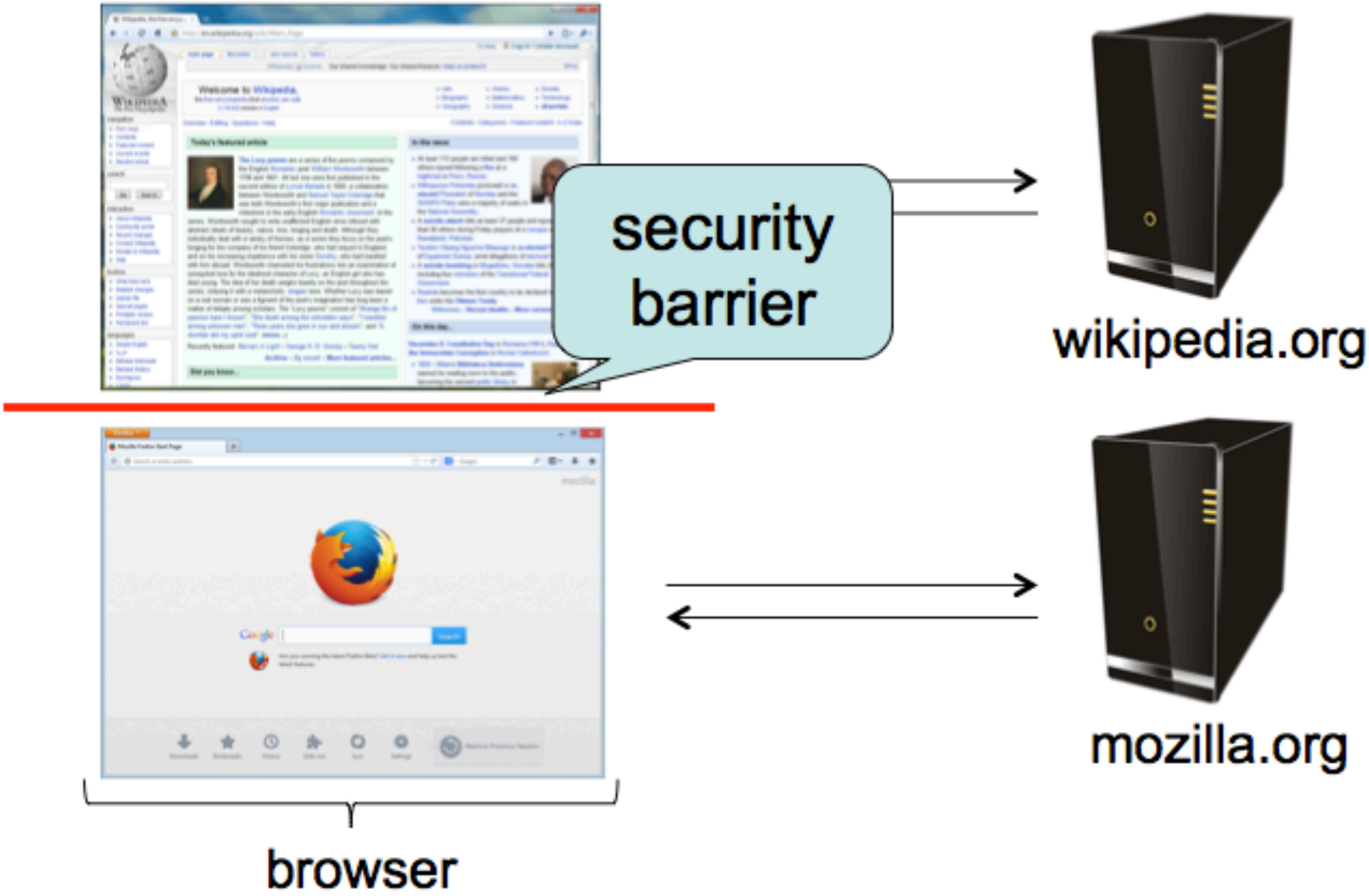
Reminder: Please put away devices at the start of class

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

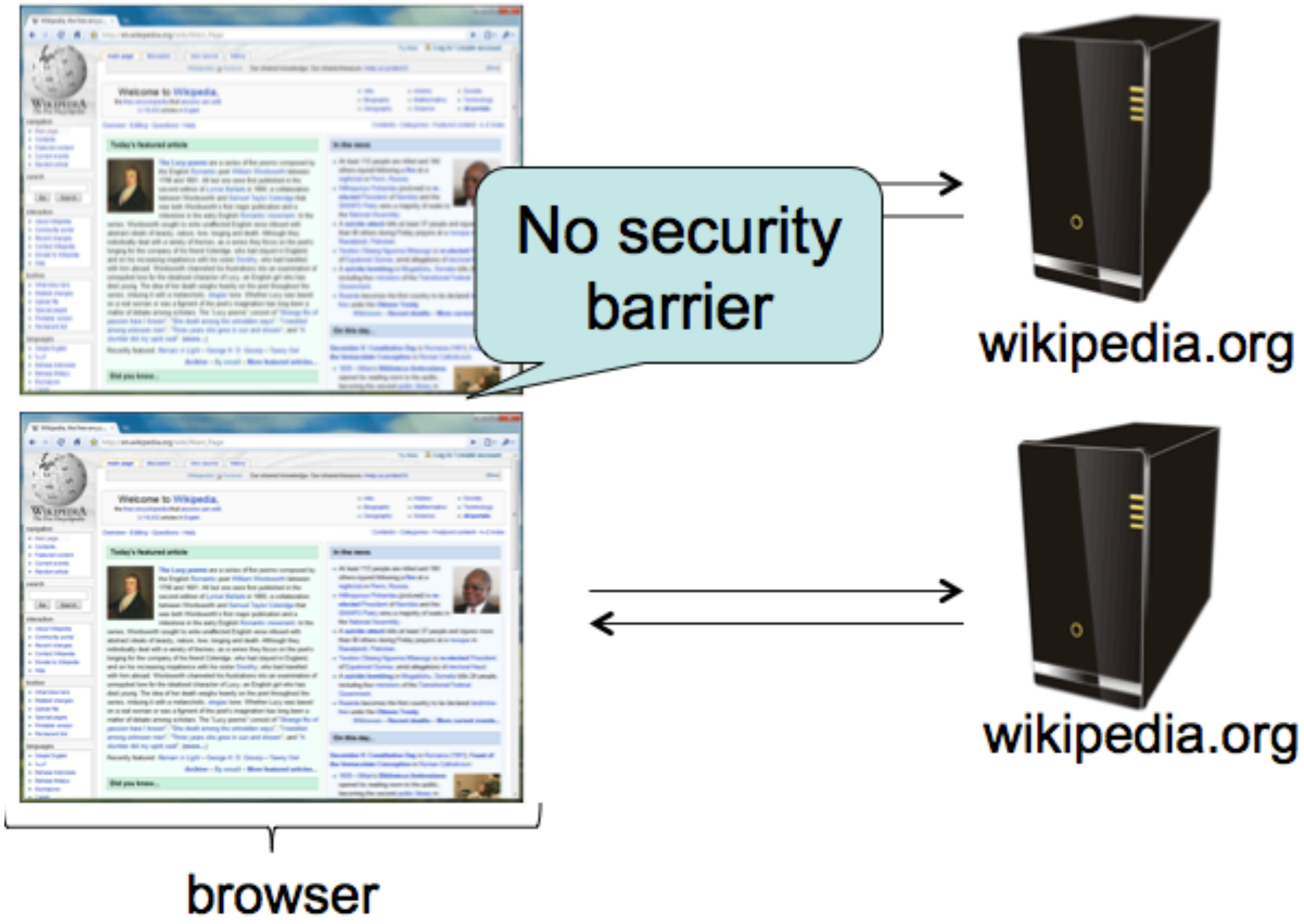
Same-origin policy

- Each site is isolated from all others



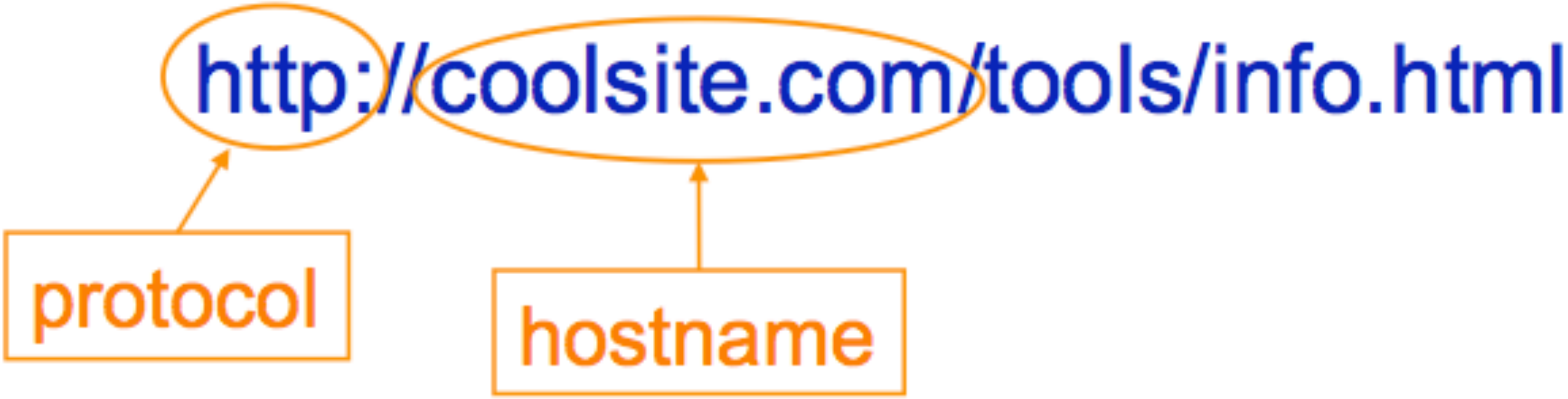
Same-origin policy

- Multiple pages from same site aren’t isolated



Same-origin policy

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



http://coolsite.com/tools/info.html

protocol

hostname

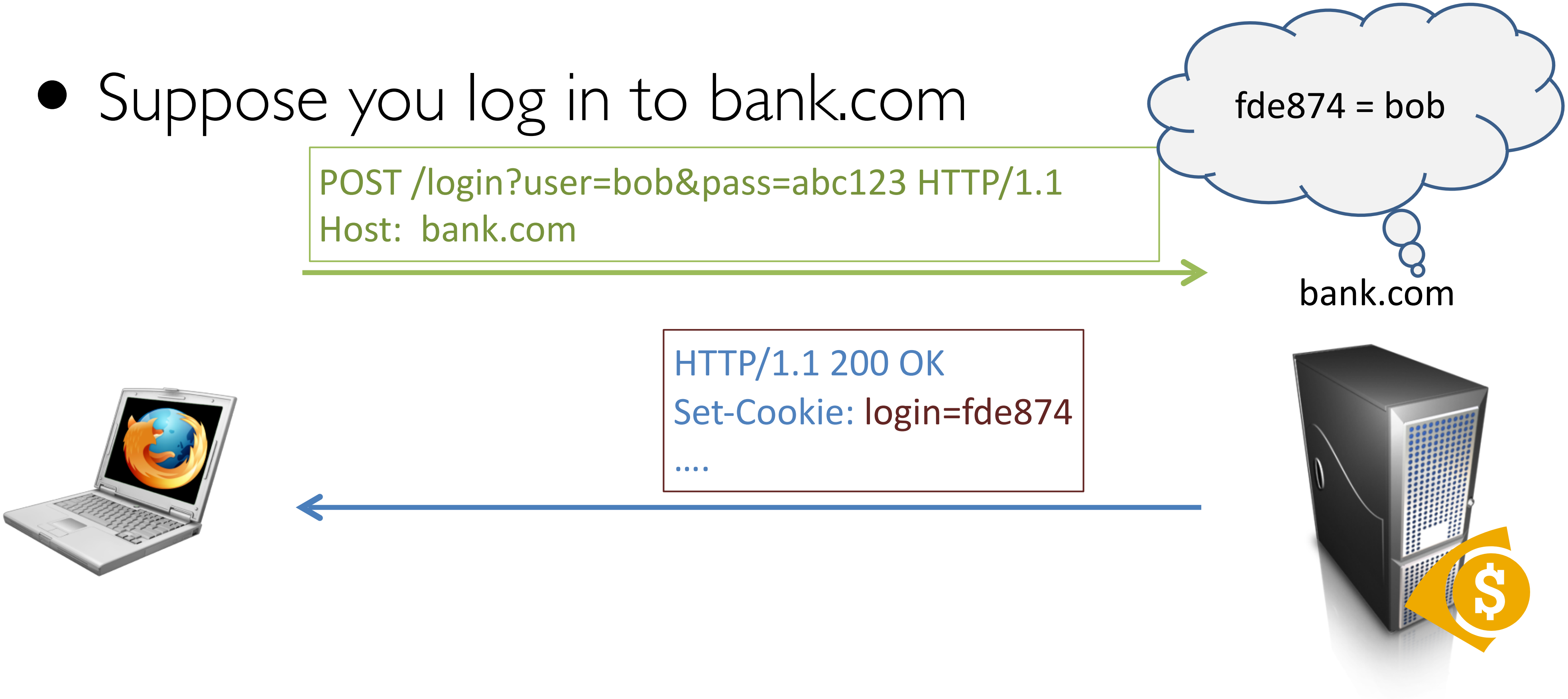
- Javascript on one page can read, change, and interact freely with all other pages from the same origin

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**

Cross-site Request Forgery (CSRF)

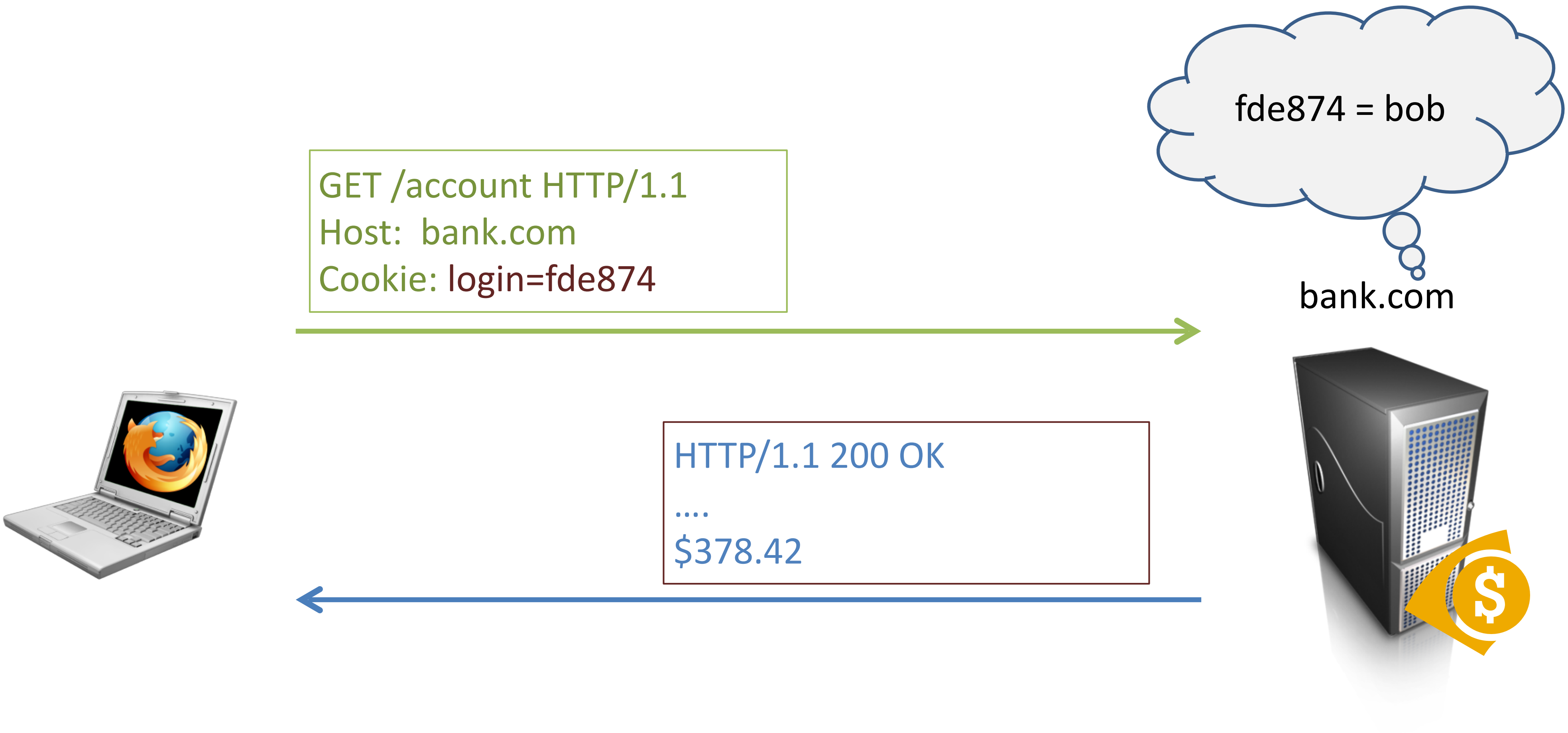
- Suppose you log in to bank.com



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

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

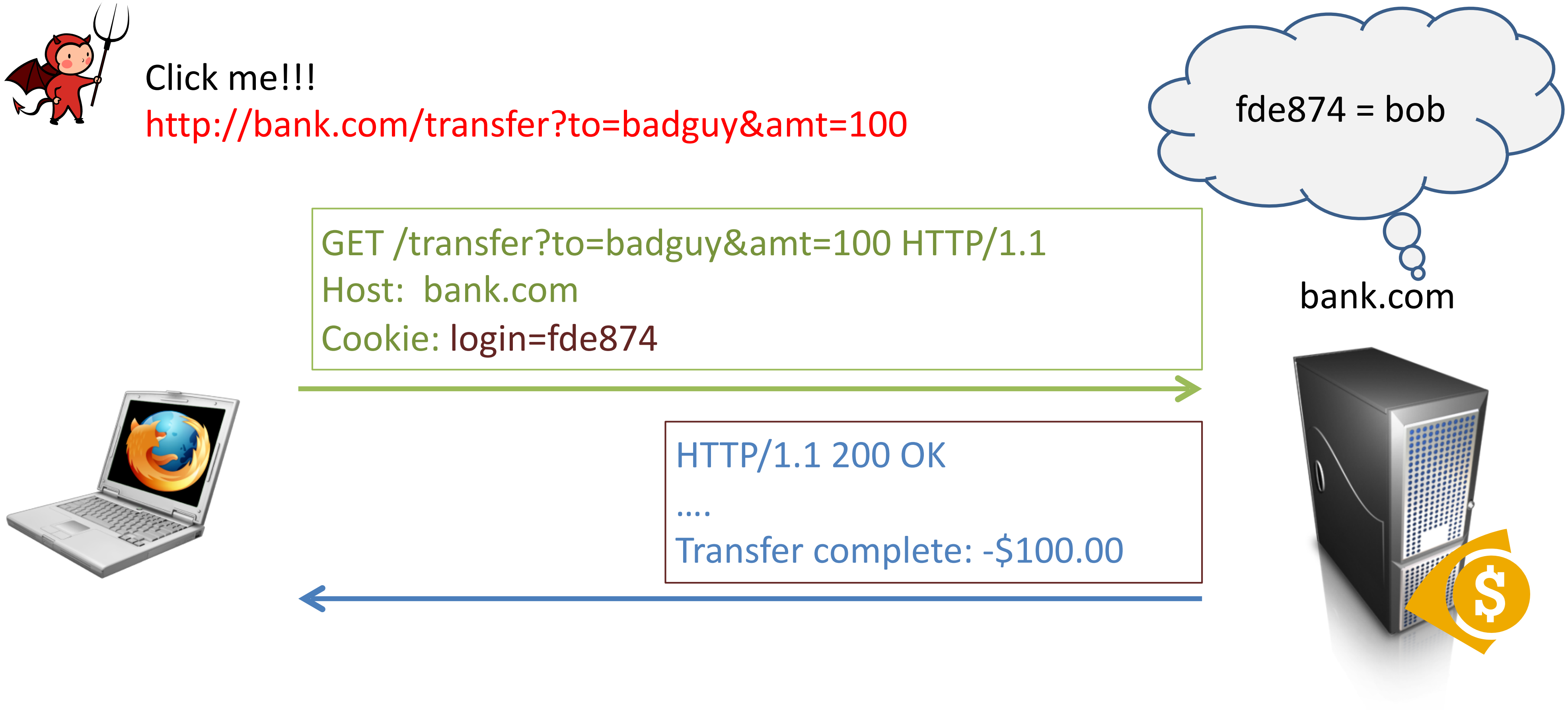
Cross-site Request Forgery (CSRF)



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

```
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
```

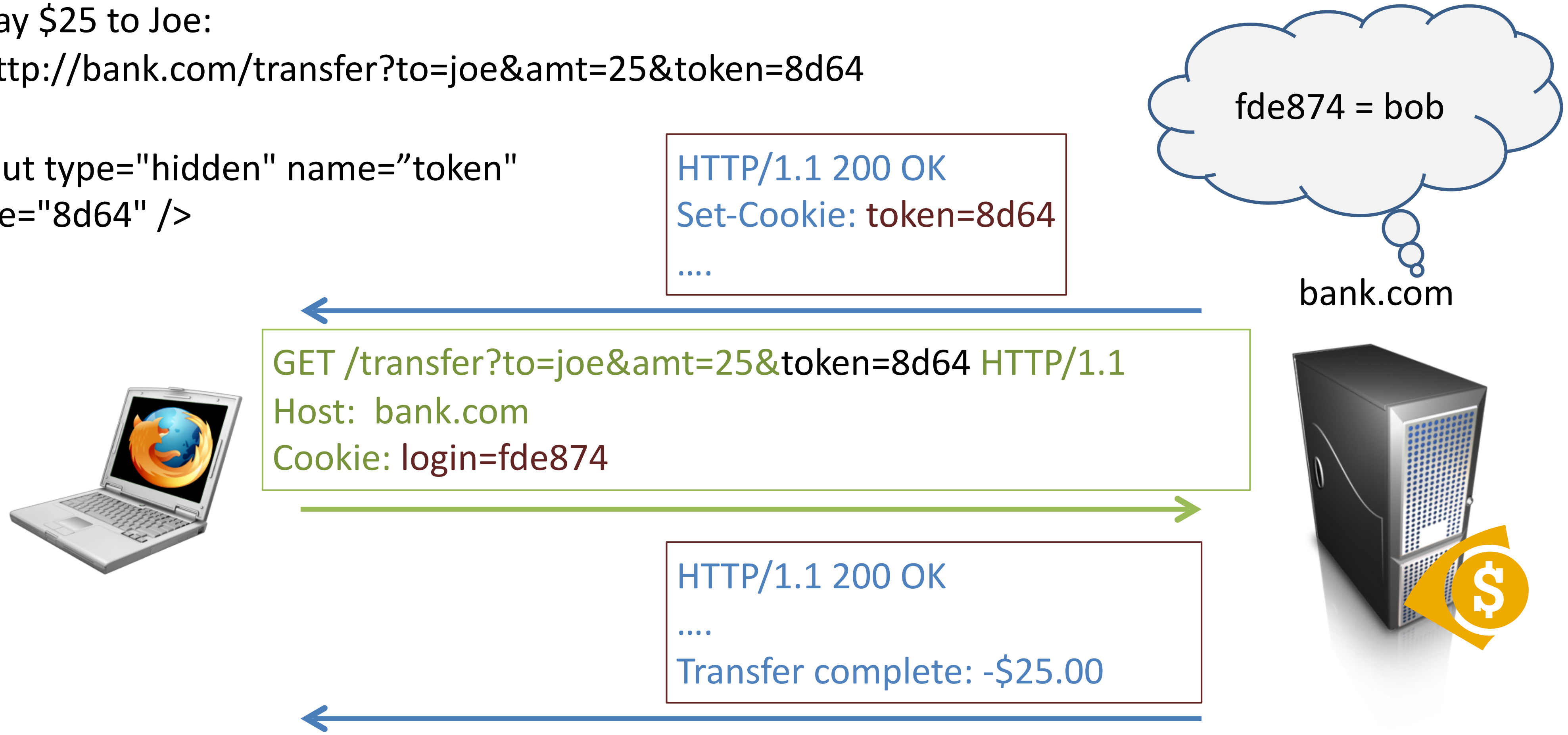
CSRF Defenses

- Need to “authenticate” 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
....
```

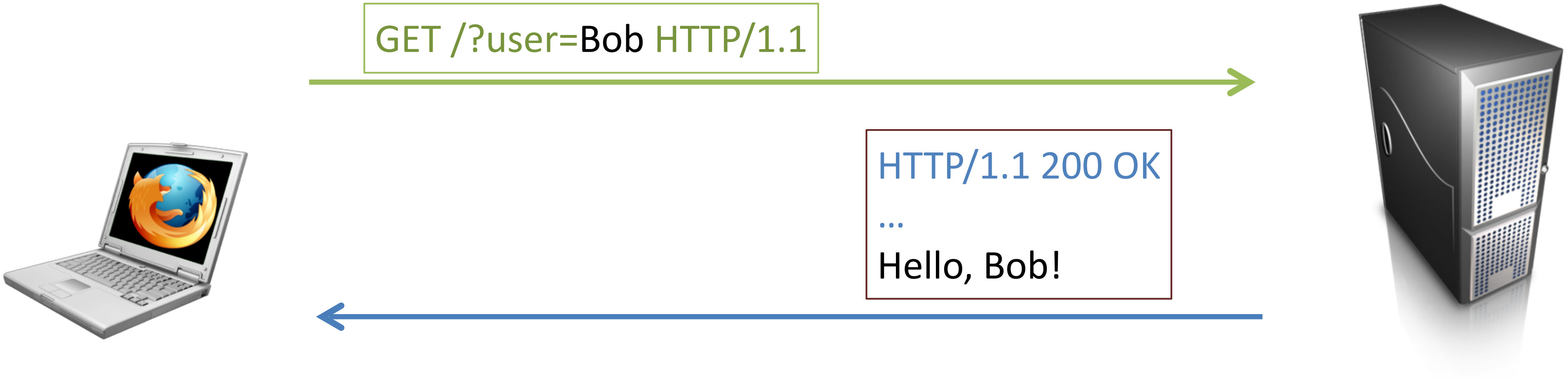
```
GET /transfer?to=joe&amt=25&token=8d64 HTTP/1.1
Host:  bank.com
Cookie: login=fde874
```

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

Cross-Site Scripting (XSS)

```
<?php

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



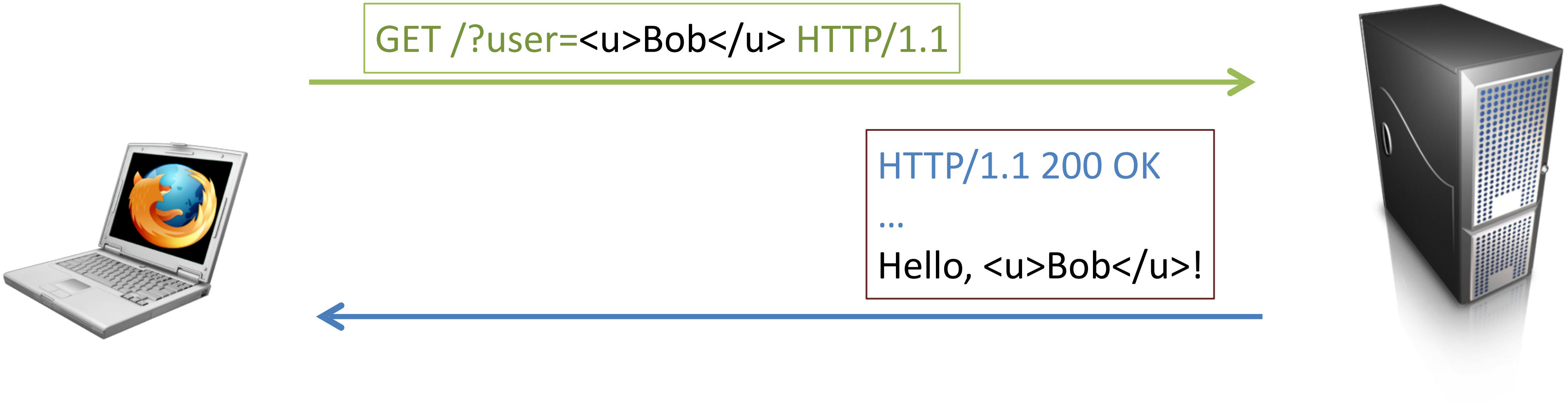
GET /?user=Bob HTTP/1.1

HTTP/1.1 200 OK
…
Hello, Bob!

Cross-Site Scripting (XSS)

```
<?php

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



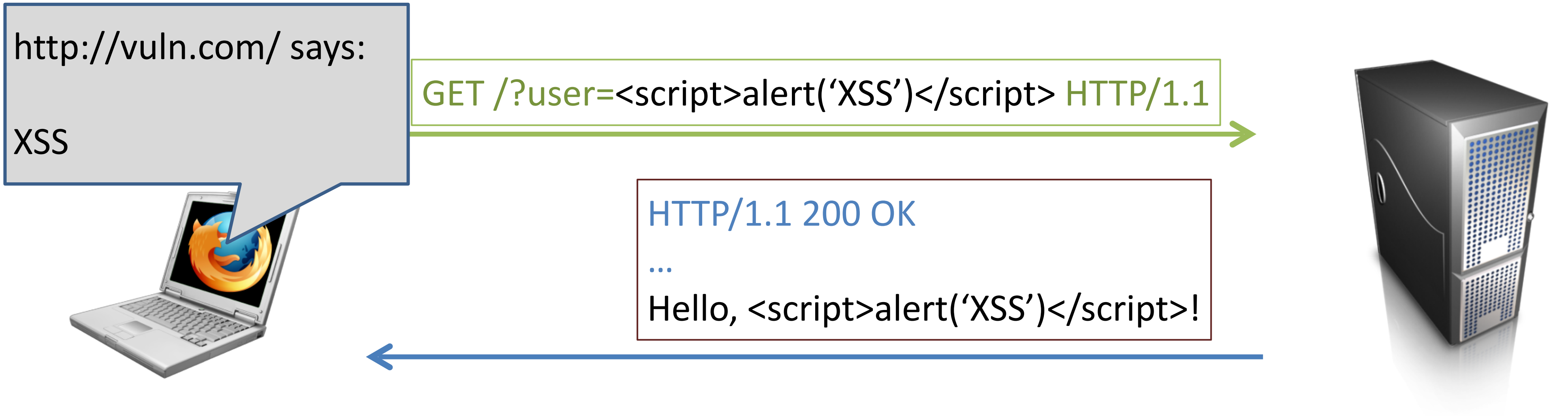
GET /?user=<u>Bob</u> HTTP/1.1

HTTP/1.1 200 OK
…
Hello, <u>Bob</u>!

Cross-Site Scripting (XSS)

```
<?php

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



http://vuln.com/ says:

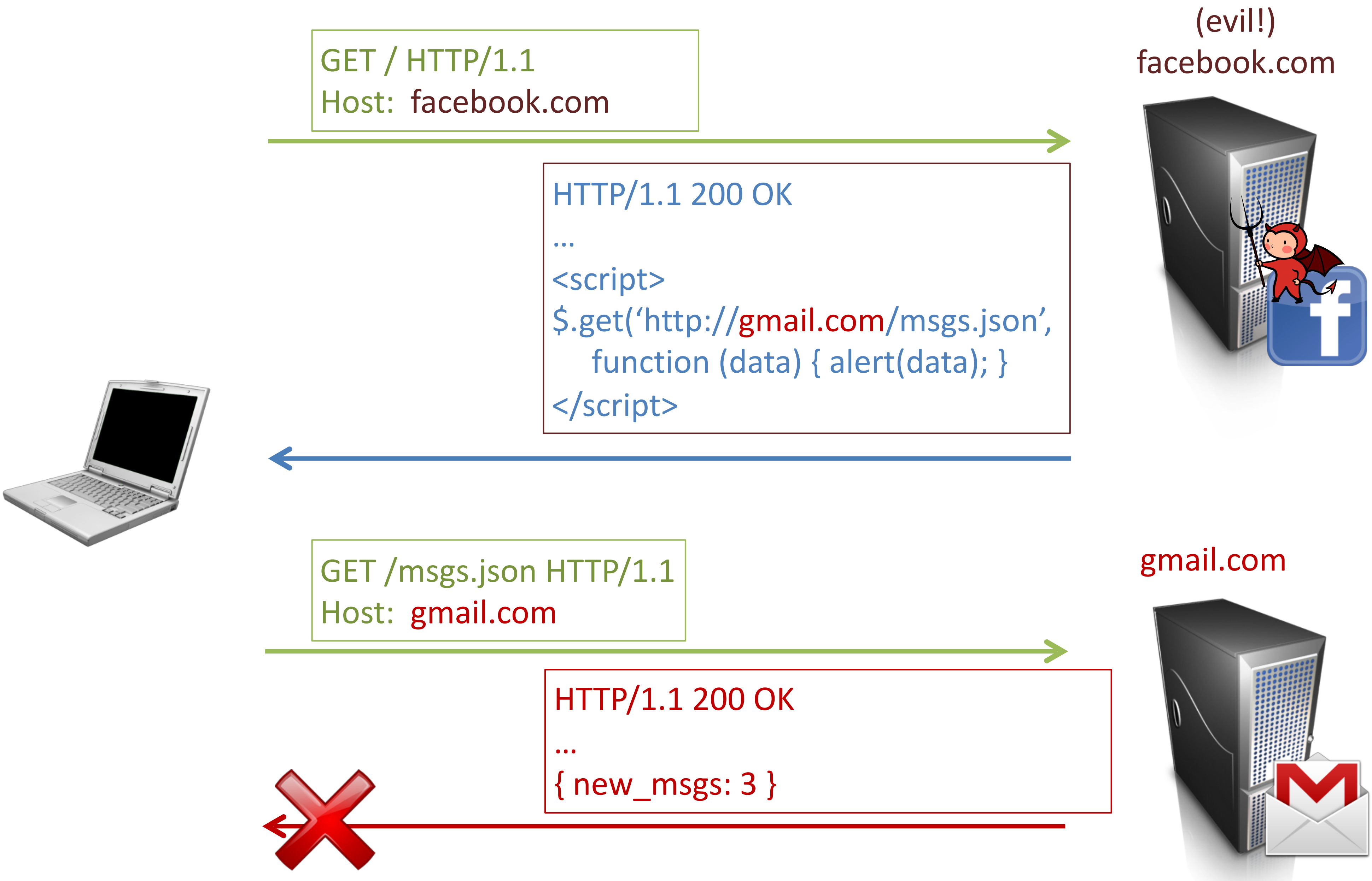
XSS

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>

Web Review | Same-Origin Policy (SOP)



(evil!)
facebook.com

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

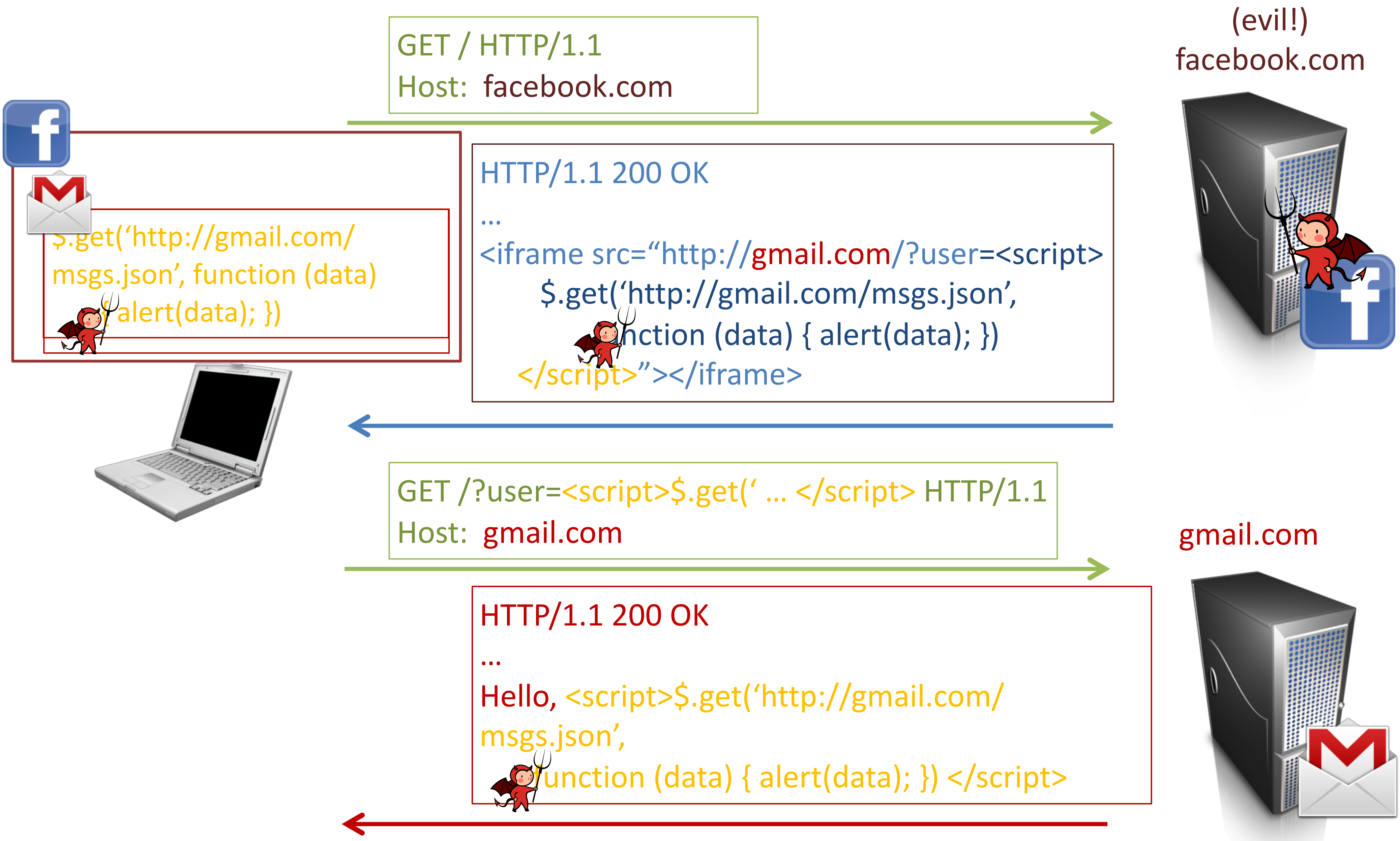
```
HTTP/1.1 200 OK
…
<script>
$.get('http://gmail.com/msgs.json',
    function (data) { alert(data); }
</script>
```

gmail.com

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

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

Cross-Site Scripting (XSS) Attack



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

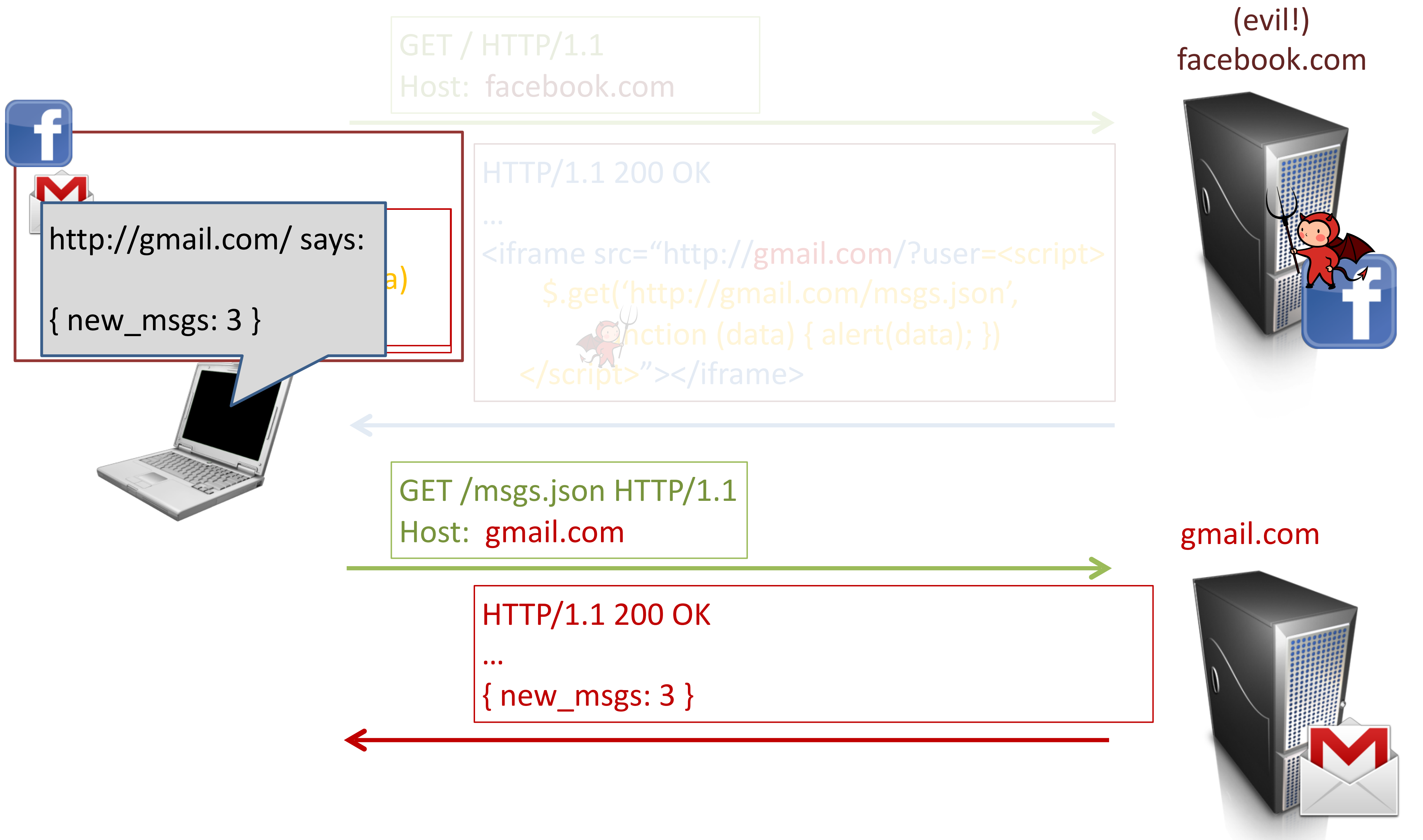
```
HTTP/1.1 200 OK
…
<iframe src=“http://gmail.com/?user=<script>
      $.get(‘http://gmail.com/msgs.json’,
          function (data) { alert(data); })
   </script>”></iframe>
```

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

```
GET /?user=<script>$.get(‘ … </script> HTTP/1.1
Host:  gmail.com
```

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

Cross-Site Scripting (XSS) Attack



```
GET / HTTP/1.1
Host:  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>
```

http://gmail.com/ says:

{ new_msgs: 3 }

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

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

XSS Defenses

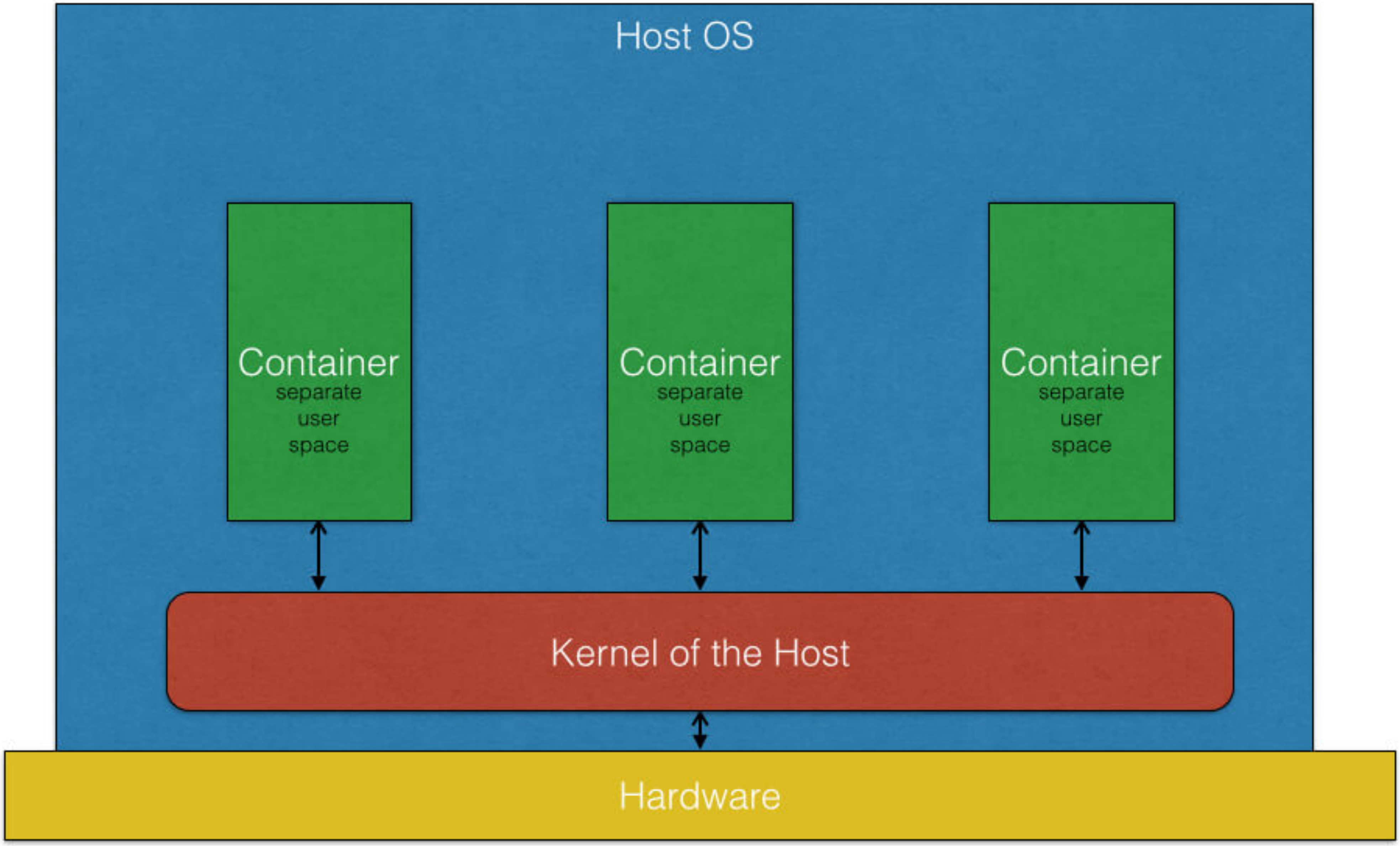
- Make sure **data** gets shown as **data**, not executed as code!
- Escape special characters
  - Which ones? Depends what context your $data is presented
    - Inside an HTML document? <div>$data</div>
    - Inside a tag? <a href=“http://site.com/$data”>
    - Inside Javascript code? var x = "$data";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

and now, cloud wrap-up

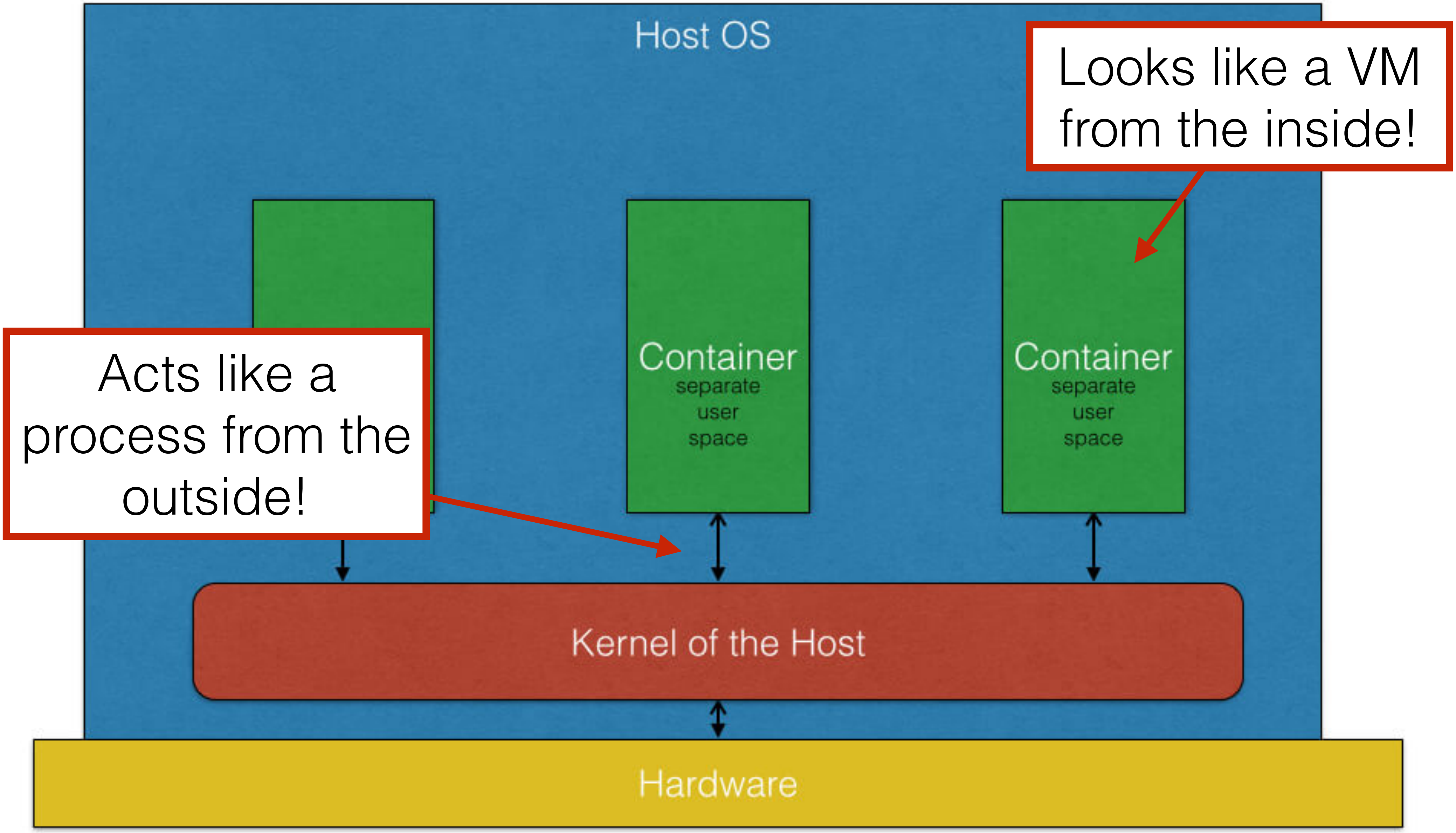
Enter Containers

- Rather than virtualize both user space and kernel space… why not just 'virtualize' user space?
- Meets the needs of most customers, who don't require significant customization of the OS.
- Sometimes called 'operating system virtualization,' which is highly misleading…
- Running natively on host, containers enjoy bare metal performance without reliance on advanced virtualization support from hardware.

Enter Containers



Enter Containers



Looks like a VM from the inside!

Acts like a process from the outside!

OS Support for Containers

- Linux Containers (LXC):
  - chroot
  - Kernel Namespaces
    - PID, Network, User, IPC, uts, mount
  - cgroups for HW isolation
  - Security profiles and policies
    - Apparmor, SELinux, Seccomp

containers = chroot on steroids

- `chroot` changes the apparent root directory for a given process and all of its children
- An old idea! POSIX call dating back to 1979
- Not intended to defend against privileged attackers… they still have root access and can do all sorts of things to break out (like `chroot`'ing again)
- Hiding the true root FS isolates a lot; in *nix, file abstraction used extensively.
- Does not completely hide processes, network, etc., though!

Namespaces

- The key feature enabling containerization!
- Partition practically all OS functionalities so that different process domains see different things
- Mount (mnt): Controls mount points
- Process ID (pid): Exposes a new set of process IDs distinct from other namespaces (i.e., the hosts)
- Network (net): Dedicated network stack per container; each interface present in exactly 1 namespace at a time.
- ….

Namespaces

- The key feature enabling containerization!
- Partition practically all OS functionalities so that different process domains see different things
- Interprocess Comm. (IPC): Isolate processes from various methods of POSIX IPC.
    - e.g., no shared memory between containers!
- UTS: Allows the host to present different host/domain names to different containers.
- There's also a User ID (user) and cgroup namespace

User Namespace

- Like others, can provide a unique UID space to the container.
- More nuanced though — we can map UID 0 inside the container to UID 1000 outside; allows processes inside of container to think they're root.
- Enables containers to perform administration actions, e.g., adding more users, while remaining confined to their namespace.

cgroups

- Limit, track, and isolate utilization of hardware resources including CPU, memory, and disk.
- Important for ensuring QoS between customers! Protects against bad neighbors
- Operate at the namespace granularity, not per-process
- Features:
    - Resource limitation
    - Prioritization
    - Accounting (for billing customers!)
    - Control, e.g., freezing groups
- The cgroup namespace prevents containers from viewing or modifying their own group assignment

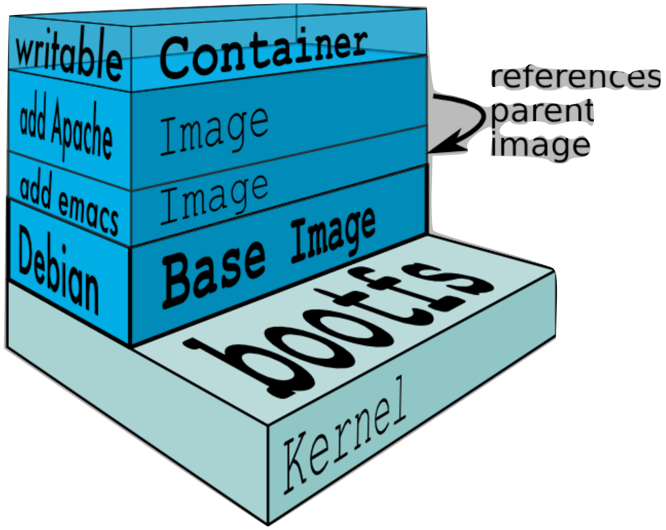
Container Security?

"Containers do not contain." - Dan Walsh (SELinux contributor)

- In a nutshell, it's *real hard* to prove that every feature of the operating system is namespaced.
  - /sys? /proc? /dev? LKMs? kernel keyrings?
- Root access to any of these enables pwning the host
- Solution? Just don't forget about access control; SELinux now offers pretty good support for namespace labeling.
- SELinux and Namespaces actually synergize nicely; *much* easier to express a correct isolation policy over a coarse-grained namespace than, say, individual processes

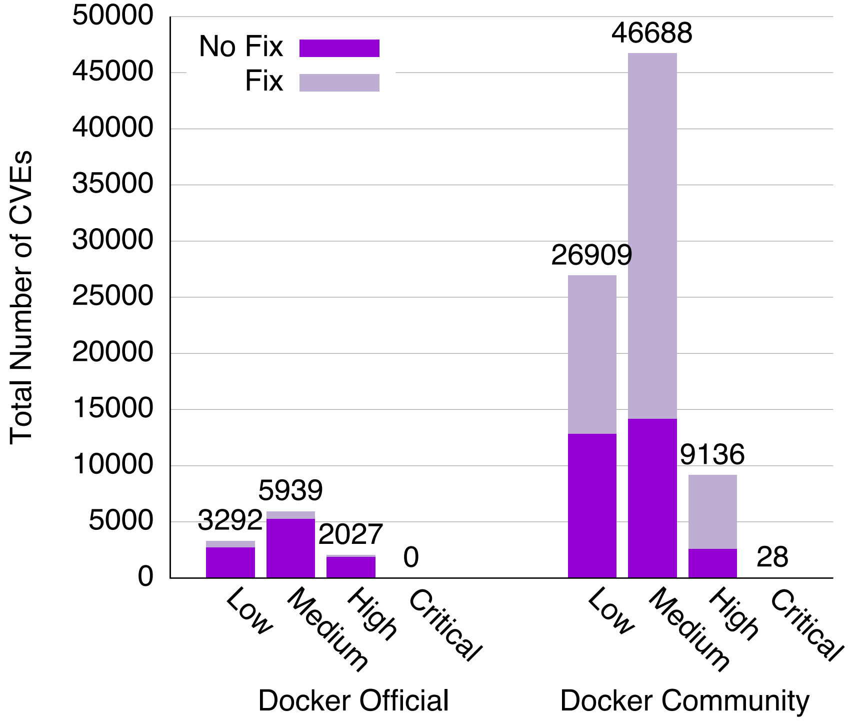
How Docker fits in

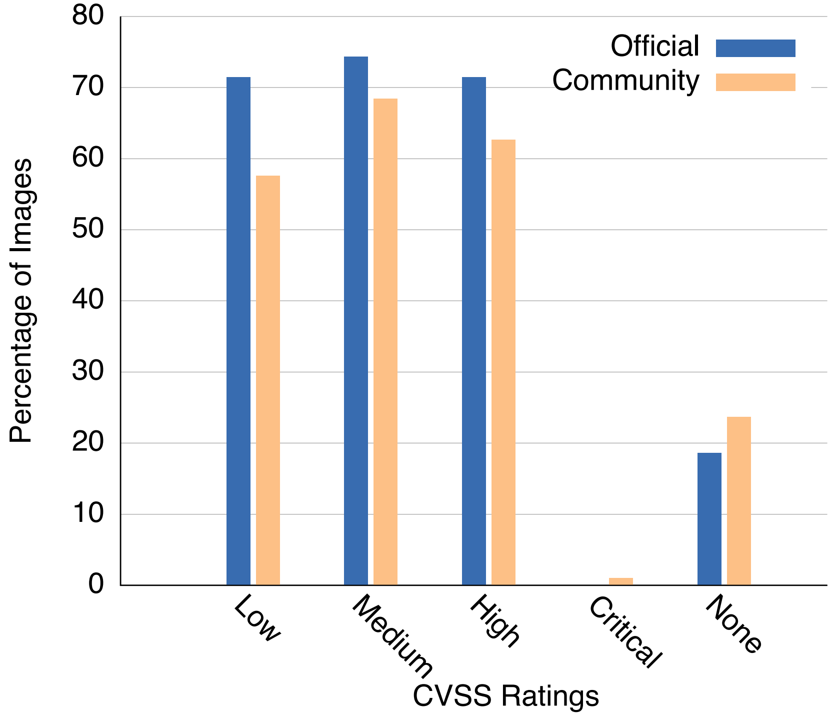
- Utilities that allow you to leverage (e.g.) LXC to build a portable, self-sufficient application using containers.
- Assures that all libraries and dependencies are packaged inside of a container image



Docker Attack Surface?

- Docker provides something analogous to an app market for building containers.
- Are the container images 'secure'?





[Hassan et al., NDSS'18]

Above the clouds…

- Container (~PaaS clouds) are strictly easier to manage than traditional IaaS VMs.
- The era of Container hype has somewhat come and gone… containers still expose more flexibility than most users need!!
- Th hype now is about Function-as-a-Service cloud; (a.k.a. serverless computing)
- Individual programs/functions executed by invocation, great for event-driven stuff.
- Enabled by containers

Serverless Computing

- Web Application is cached inside of FaaS cloud; no persistent architecture (virtual or otherwise)
- Web App components (Functions) are spawned and destroyed on a per-request basis.
- Function invocation is triggered by different event sources, e.g., web requests
- Pay *by the millisecond* for use
- Security considerations??

Serverless Security

- ***Are functions secure?***
- Standard attacks probably won't work, and if they do the function will shut down immediately
- Harder for malware to establish persistence, e.g., file system is read-only, container dies*
  - * it's supposed to, anyway…
- Harder to exfiltrate due to virtual network rules
  - reverse shell probably not available because exploitable function does not have Internet access

FaaS Threat Vectors

- Probe with standard attacks (e.g., code injection) in the hopes of breaking something, identify location in workflow that will leak the presence of a crash (e.g., confirmation email)
- Can't drop to shell; how to establish persistence?

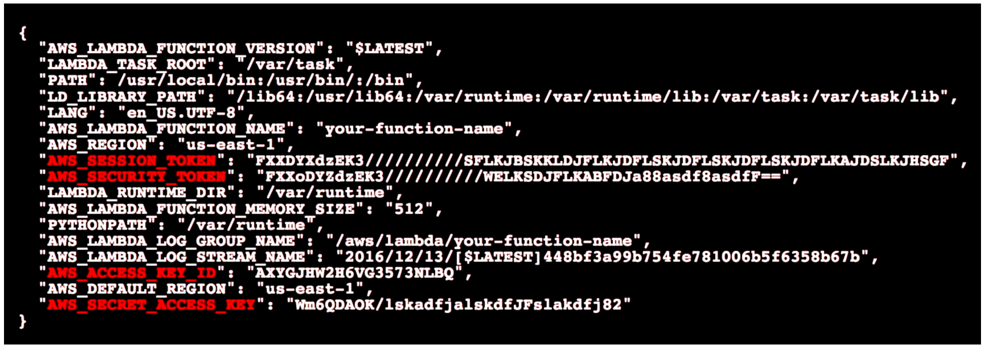
[Jones, CCC Congress'13]

Container Reuse

- Functions execute inside a container
- For performance ‘warm’ containers are sometimes re-used across multiple functions/customers
  - i.e., numerous *explicit* data channels (unlike IaaS)
- Containers/Functions are cached in memory and re-launched if a request arrives in a short enough period of time
- Attacker can leverage this to simulate persistence

FaaS Threat Vectors

- Probe with standard attacks (e.g., code injection) in the hopes of breaking something, identify location in workflow that will leak the presence of a crash (e.g., confirmation email)
- Can’t drop to shell; how to establish persistence?
  - Write malware or tool sets to /tmp
  - Issue periodic requests to keep container cached
- On AWS, customer access tokens are stored as environment variables in instance… potential for tokens to be wildly overprivileged if misconfigured.



```
{
  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  "LAMBDA_TASK_ROOT": "/var/task",
  "PATH": "/usr/local/bin:/usr/bin/:/bin",
  "LD_LIBRARY_PATH": "/lib64:/usr/lib64:/var/runtime:/var/runtime/lib:/var/task:/var/task/lib",
  "LANG": "en_US.UTF-8",
  "AWS_LAMBDA_FUNCTION_NAME": "your-function-name",
  "AWS_REGION": "us-east-1",
  "AWS_SESSION_TOKEN": "FXXDYXdzEK3//////////SFLKJBSKKLDJFLKJDFLSKJDFLSKJDFLSKJDFLKAJDSLKJHSGF",
  "AWS_SECURITY_TOKEN": "FXXoDYZdzEK3//////////WELKSDJFLKABFDJa88asdf8asdfF==",
  "LAMBDA_RUNTIME_DIR": "/var/runtime",
  "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "512",
  "PYTHONPATH": "/var/runtime",
  "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/your-function-name",
  "AWS_LAMBDA_LOG_STREAM_NAME": "2016/12/13/[$LATEST]448bf3a99b754fe781006b5f6358b67b",
  "AWS_ACCESS_KEY_ID": "AXYGJHW2H6VG3573NLBQ",
  "AWS_DEFAULT_REGION": "us-east-1",
  "AWS_SECRET_ACCESS_KEY": "Wm6QDAOK/lskadfjalskdfJFslakdfj82"
}
```

- Use token-based authorizations to access data, then exfiltrate using cloud services

[Jones, CCC Congress’16]