Web Security

Thierry Sans

1991

Sir Tim Berners-Lee

info.cern.ch/hypertext/WWW/TheProject.html

World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
: Pointers to the world's online information, subjects , W3 servers, etc.

Help
: on the browser you are using

Software Products
: A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)

Technical
: Details of protocols, formats, program internals etc

Bibliography
: Paper documentation on W3 and references.

People
: A list of some people involved in the project.

History
: A summary of the history of the project.

How can I help ?
: If you would like to support the web..

Getting code
: Getting the code by anonymous FTP , etc.

Web Portals

2014

Customer Resources Managemen

E-Health

Accounting and Billing

E-Learning

Collaboration

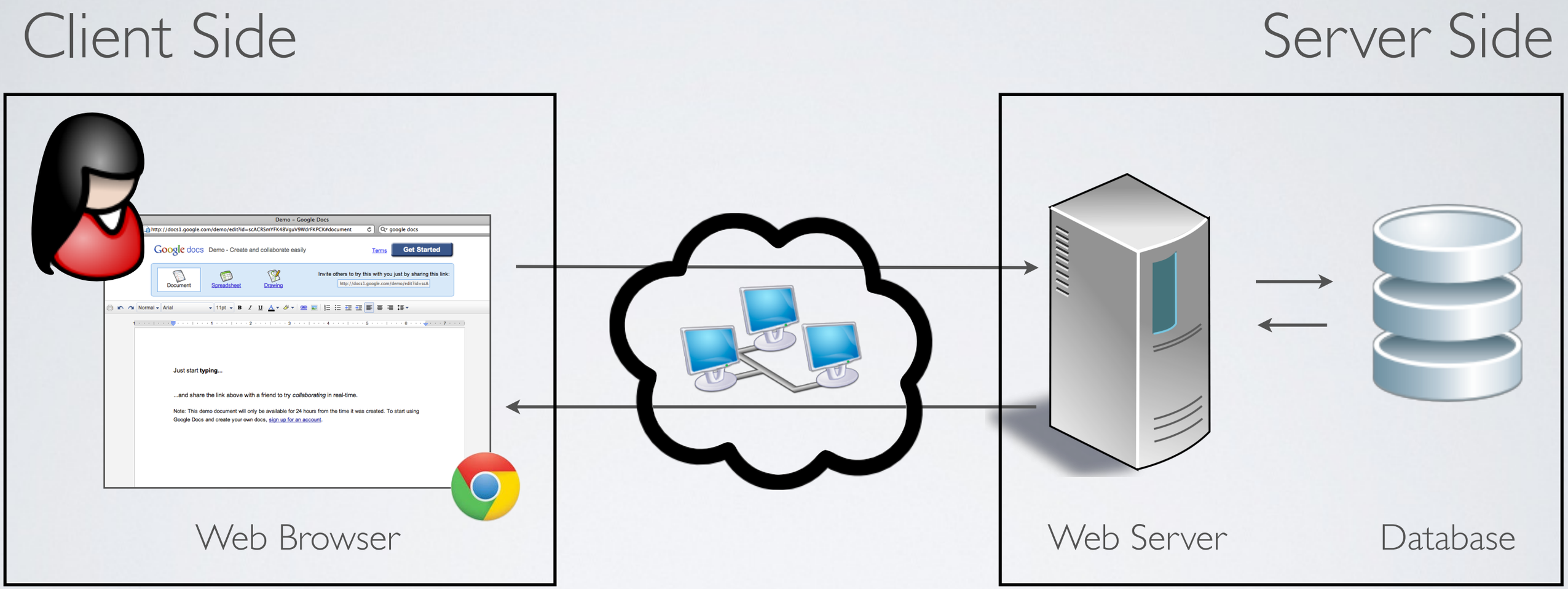
Content Management

Social Networks

Publishing

The Big Picture

The web architecture



Securing the web architecture means securing ...

- The network
- The DNS (Domain Name System)
- The web server operating system
- The web server application (*Apache* for instance)
- The database application (*Oracle* for instance)
- The web application

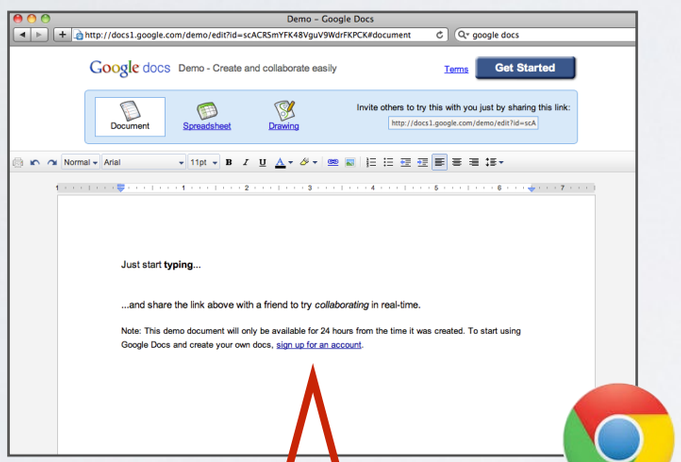
Our focus here!

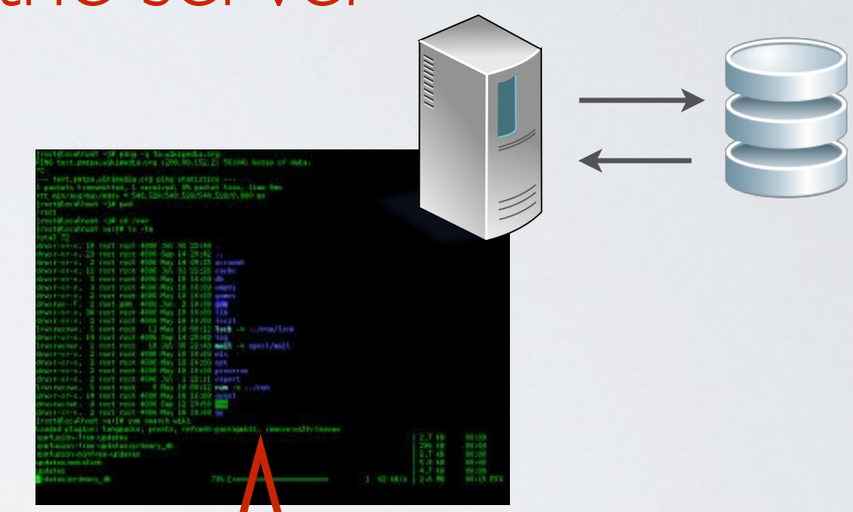
What is a web application?

program running on the browser

+

program running on the server





JS

HTML

CSS

python

RAILS

php

node

Java

Microsoft ASP.net

Anatomy of a web application

The HTTP protocol

Network protocol for requesting/receiving data on the Web

- Standard TCP protocol on **port 80** (by default)
- **URI/URL** specifies what resource is being accessed
- Different **request methods**

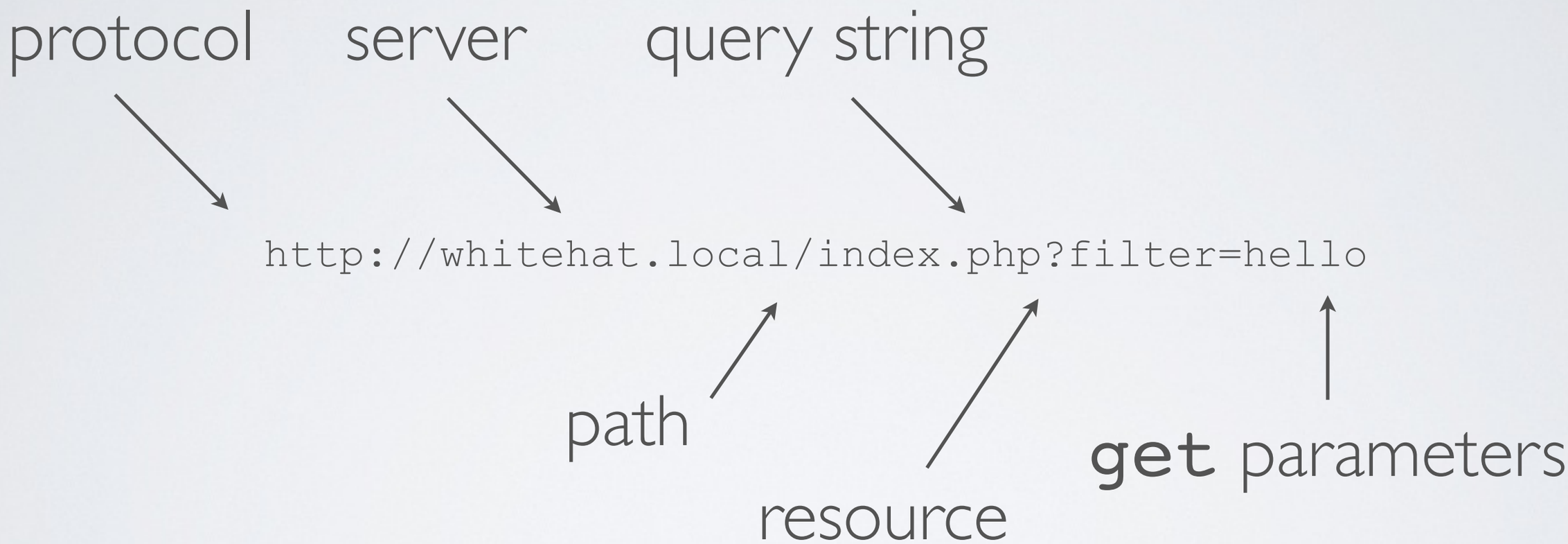
Let's look at what a web server does

telnet to a web server

```
> telnet whitehat.local 80
GET /
```

enter HTTP requests

Anatomy of a URL



Authentication and Authorization

- ✓ Authentication
  - Who are the authorized users?
- ✓ Authorization
  - Who can access what and how?

The simple recipe for user authentication

1. **Ask the user for a login and password** and send it to the server (HTTP/POST request)
2. **Verify the login/password** based on information stored on the server (usually in the database)
3. **Start a session** once the user has been authenticated
4. **Grant access to resources** according to the session

The concept of session

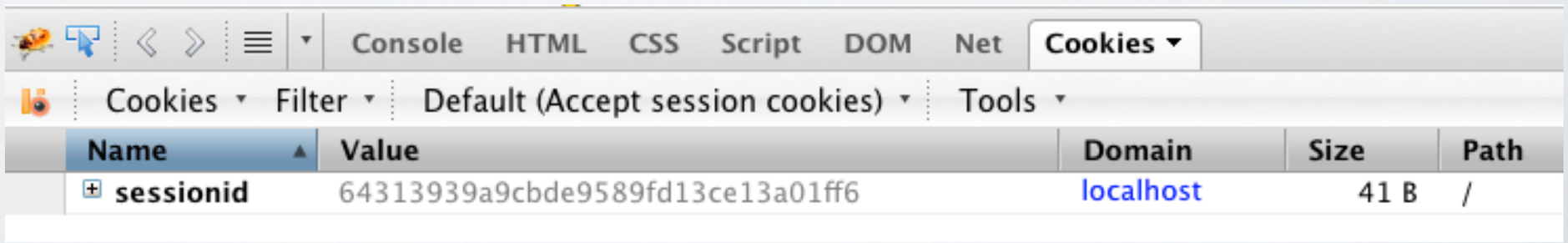
There is a **session id** (aka token)
between the browser and the web application

This session id should be **unique** and **unforgeable**
(usually a long random number or a hash)

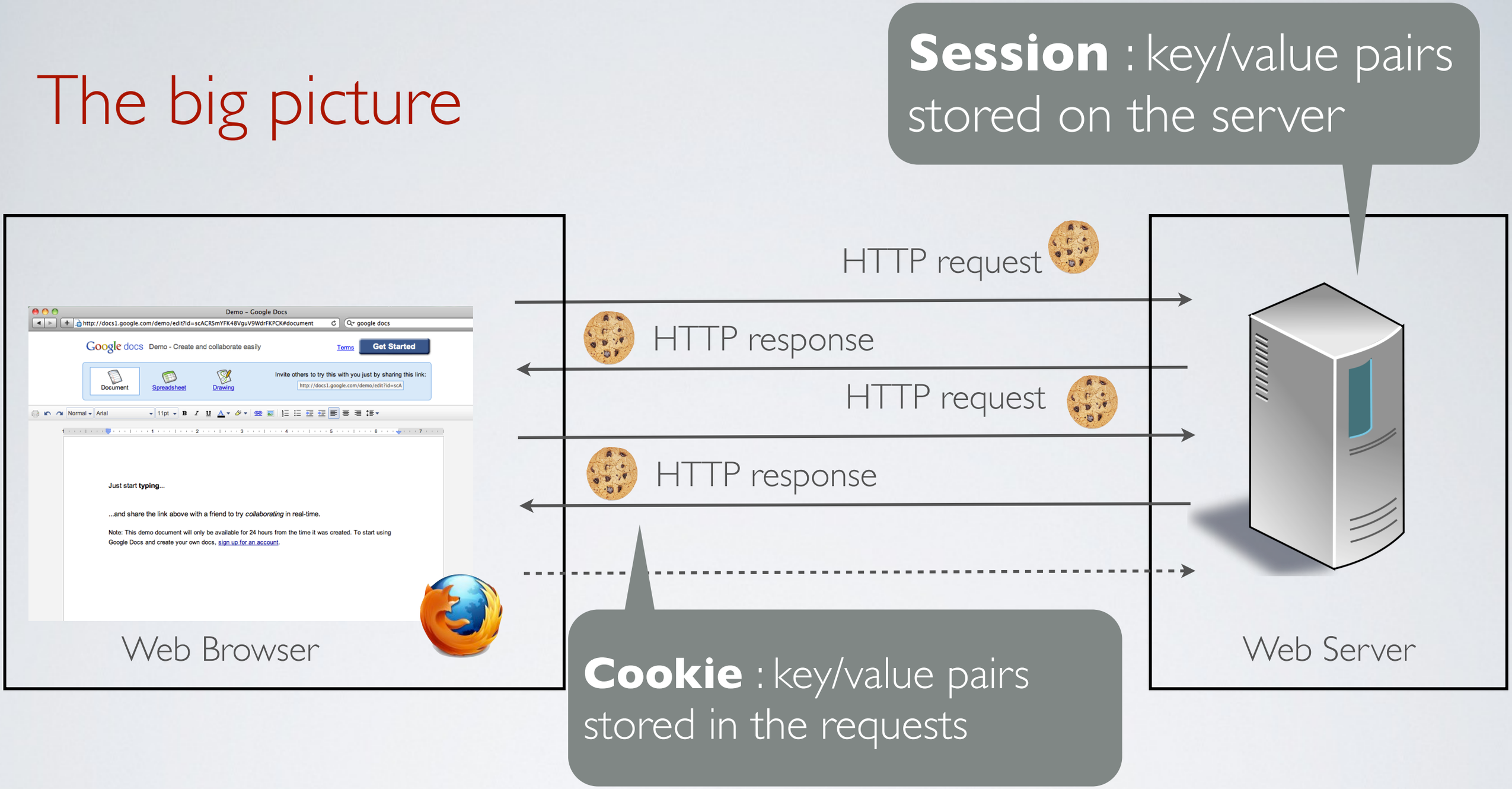
➡ Stored in the cookie

The session id is bind to **key/value pairs data**

➡ Stored on the server



The big picture



The user can **create, modify, delete** the session ID in the cookie

But **cannot access** the key/value pairs stored on the server

Insufficient Transport Layer Protection

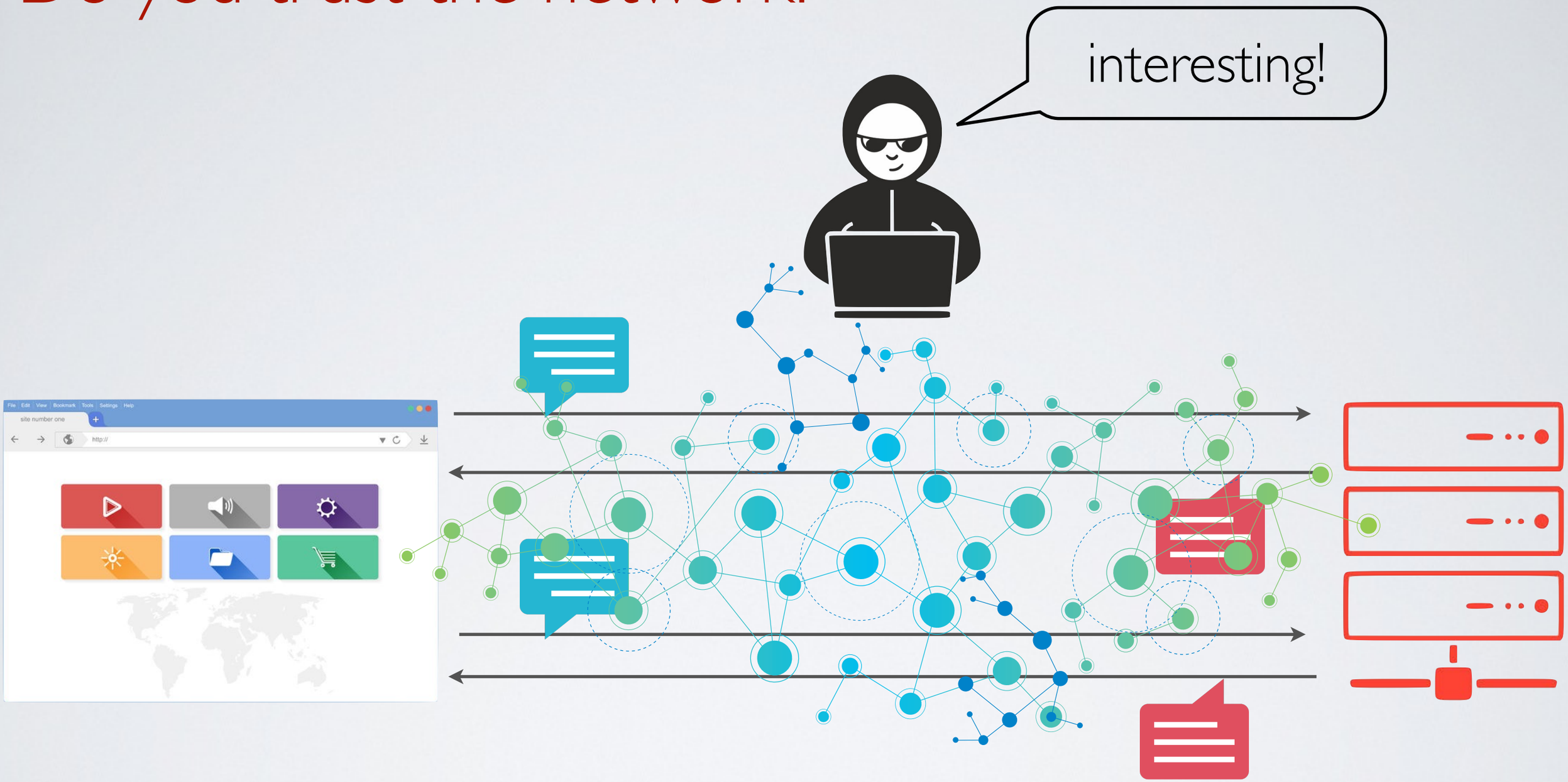
a.k.a the need for HTTPs

How to steal user's credentials

- ➡ Brute force the user's pa[illegible]word or ses[illegible]n ID

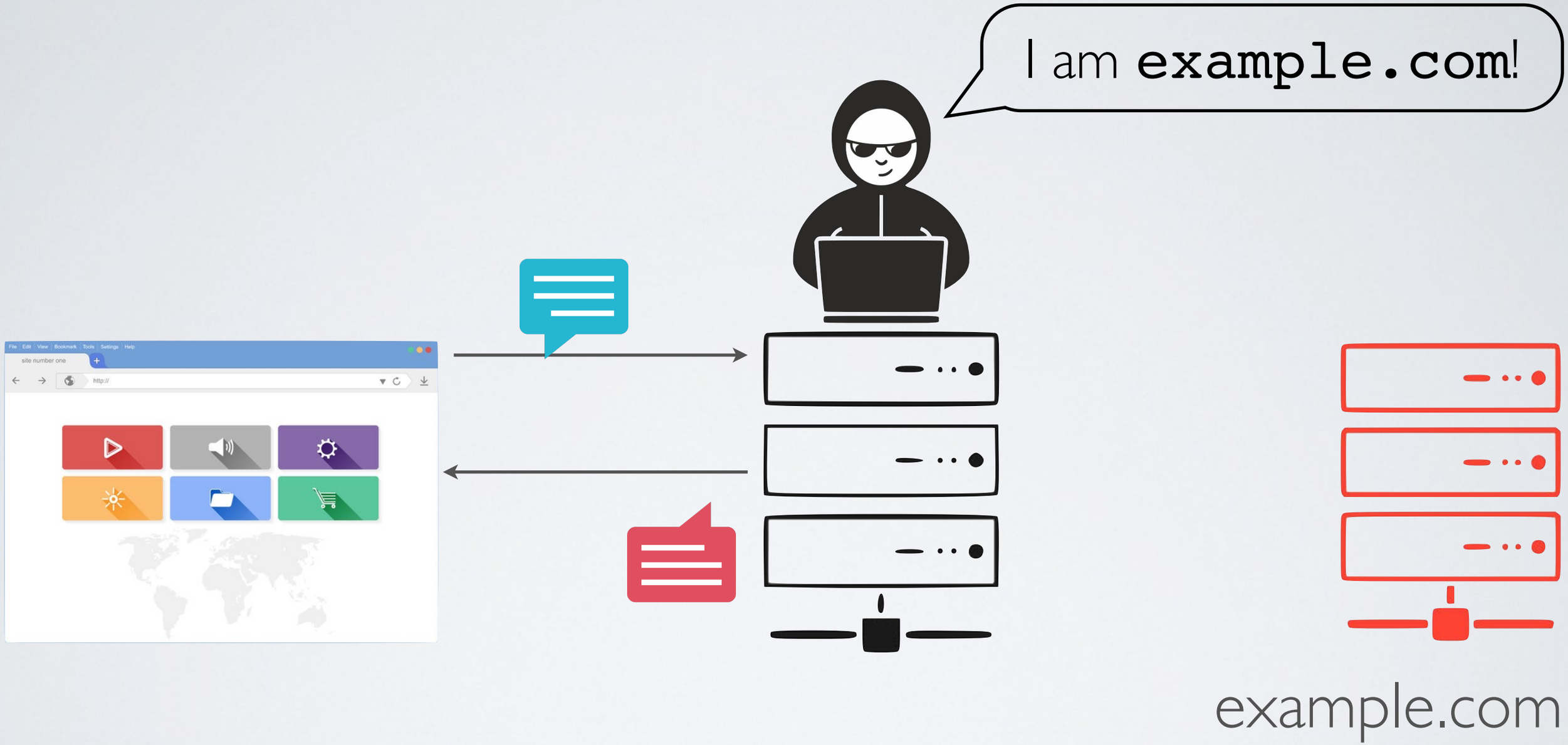
- ➡ Steal the user's password or session ID

Do you trust the network?



- Threat 1 : an attacker **can eavesdrop** messages sent back and forth

Do you *really* trust the network?



- Threat 2 : an attacker **can tamper with** messages sent back and forth

Confidentiality and Integrity

- Threat 1 : an attacker **can eavesdrop** messages sent back and forth

  Confidentiality: how do exchange information <u>secretly?</u>

- Threat 2 : an attacker **can tamper** messages sent back and forth

  Integrity: How do we exchange information <u>reliably?</u>

Why and when using HTTPS?

HTTPS = HTTP + TLS

➡ TLS provides

- confidentiality: end-to-end secure channel
- integrity: authentication handshake

➡ HTTPS protects any data send back and forth including:

- login and password
- session ID

✓ **HTTPS everywhere**
HTTPS must be used during the entire session

Be careful of mixed content

Mixed-content happens when:

1. an HTTPS page contains elements (ajax, js, image, video, css ...) served with HTTP

2. an HTTPS page transfers control to another HTTP page within the same domain

- authentication cookie will be sent over HTTP
- Modern browsers block (or warn of) mixed-content

`Secure` cookie flag

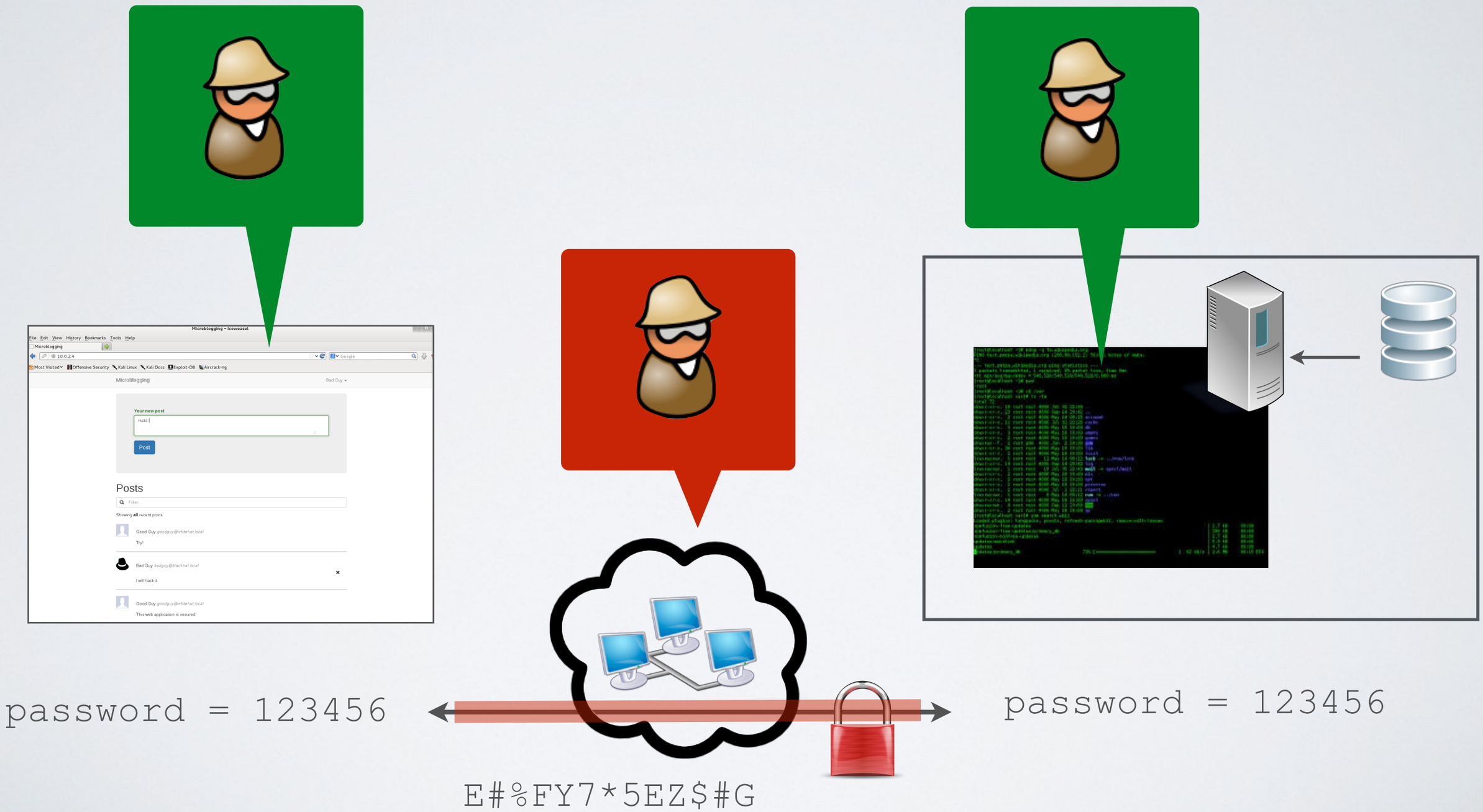
✓ The cookie will be sent over HTTPS exclusively

➡ Prevents authentication cookie from leaking in case of mixed-content

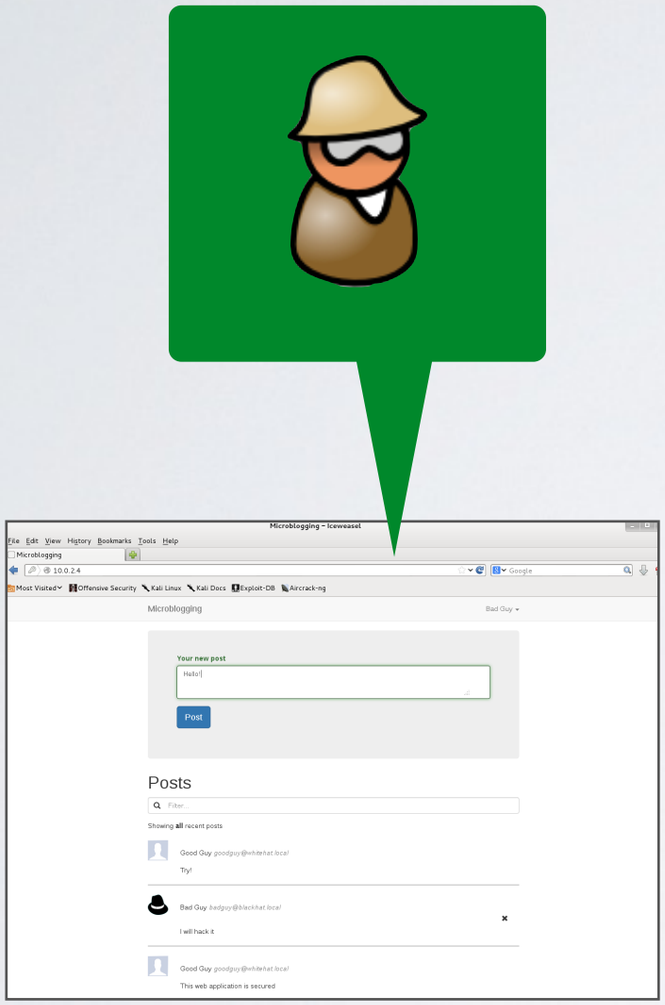
Do/Don't with HTTPS

- Always use HTTPS exclusively (in production)
- Always have a valid and signed certificate (no self-signed cert)
- Always avoid using absolute URL (mixed-content)
- Always use **`secure`** cookie flag with authentication cookie

Limitation of HTTPS



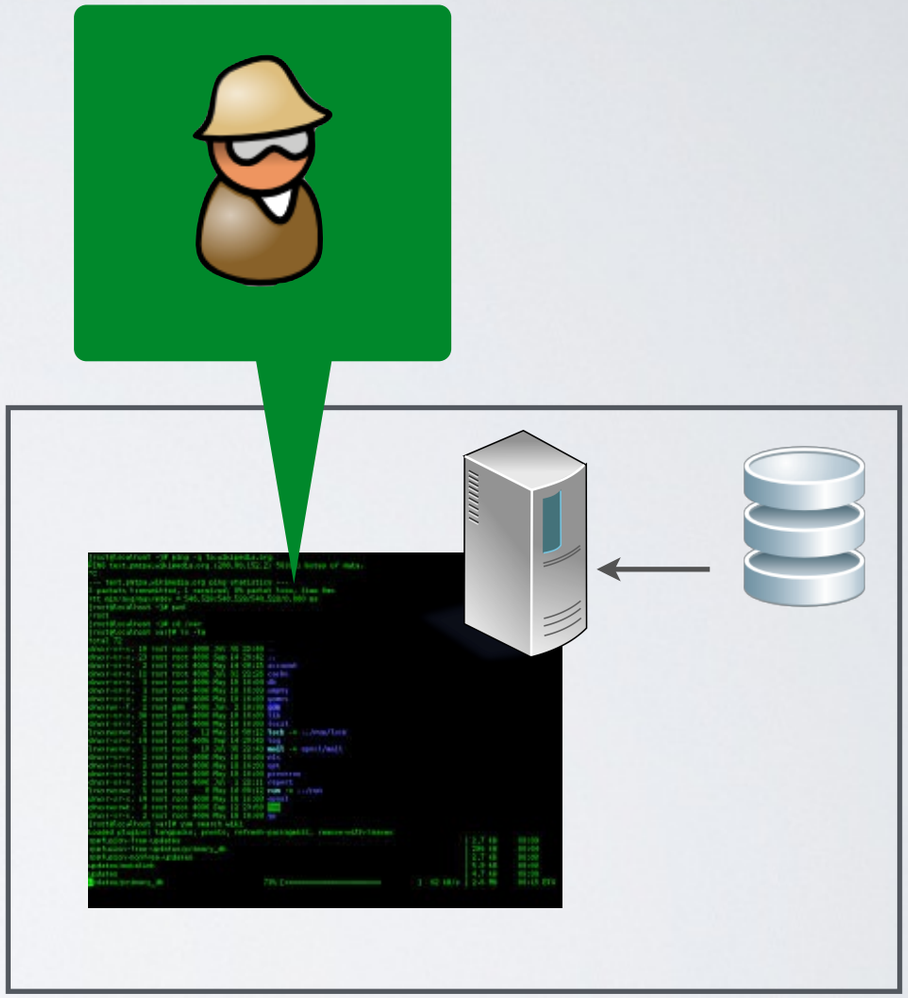
Stealing passwords from the client



- Social engineering - Phishing
- Keyloggers (keystroke logging)
- Data mining (emails, logs)
- Hack the client's code

Stealing passwords from the server

- Hack the server
- Hack the server's side code



Beyond HTTPS - attacking the web application

Frontend Vulnerabilities

- Content Spoofing
- **Cross-Site Scripting**
- **Cross-site Request Forgery**

Backend Vulnerabilities

- **Incomplete Mediation**
- Broken Access Control
- **SQL Injection**

Backend Vulnerability

Broken Access Control

Information Leakage

"AT&T Inc. apologized to Apple Inc. iPad 3G tablet computer users whose **e-mail addresses were exposed during a security breach** disclosed last week."

source *Business Week - June 14 2010*

"There's no hack, no infiltration, and no breach, **just a really poorly designed web application** that returns e-mail address when ICCID is passed to it."

source *Praetorian Prefect - June 9 2010*

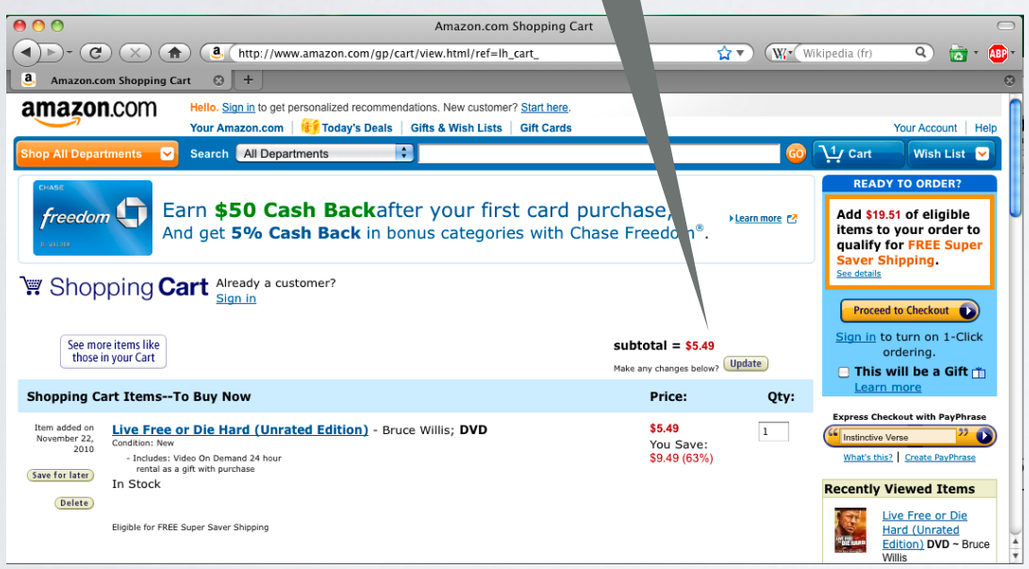
Backend Vulnerability

Incomplete Mediation

Incomplete Mediation - The Shopping Cart Attack

The total is calculated by a script on the client

The order is generated based on the request

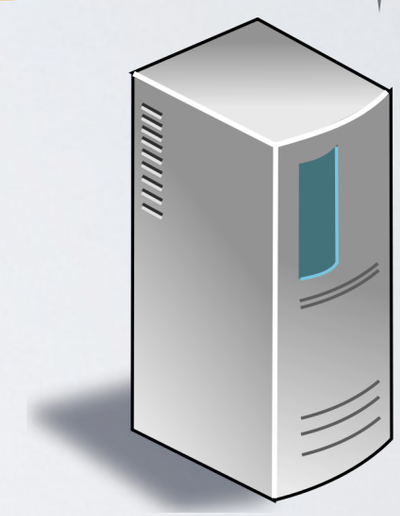


*

amazon.com

order=(#2956,10,1,10)

Thank you for your order!



Client Trusted Domain

Server Trusted Domain

* *Notice that Amazon is **not** vulnerable to this attack*

The backend is the **only trusted domain**

- ◉ Data coming from the frontend cannot be trusted
- ✓ Sensitive operations must be done on the backend

Backend Vulnerability

SQL Injection

Problem

➡ An attacker can inject SQL/NoSQL code

- Retrieve, add, modify, delete information
- Bypass authentication

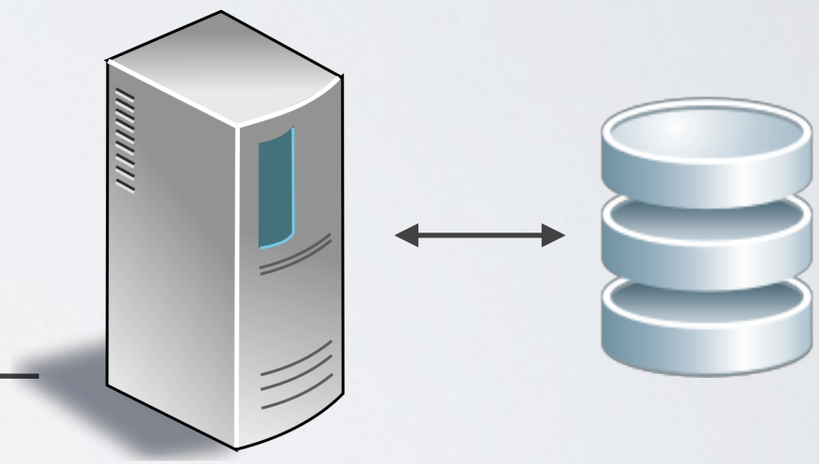
Checking password

signin.html

/signin/

name=Alice&pwd=pass4alice

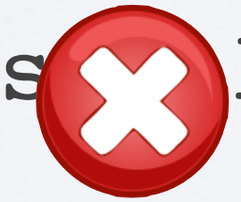
Access Granted!



Bypassing password check

```
db.run("SELECT * FROM users
WHERE USERNAME = '" + username + "'
  AND PASSWORD = '" + password + "'")
```

```
username: alice
password: pas lice
```



```
blah' OR '1'='1
```

NoSQL Injection

```
db.find({  username: username,
           password: password   });
```

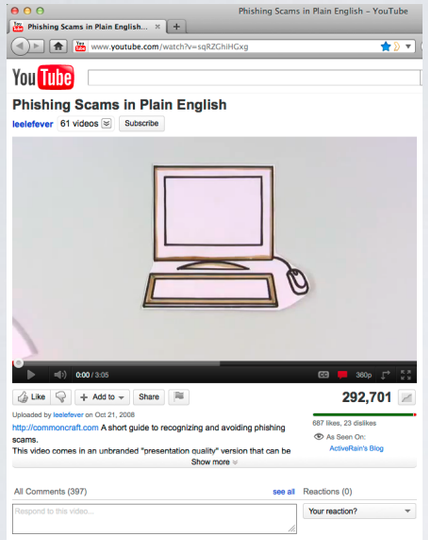
```
username: alice
password: pas lice
```

```
{gt: ""}
```

Frontend Vulnerability

Content Spoofing

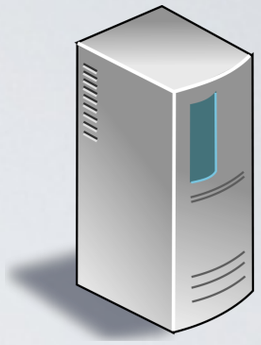
Content Spoofing

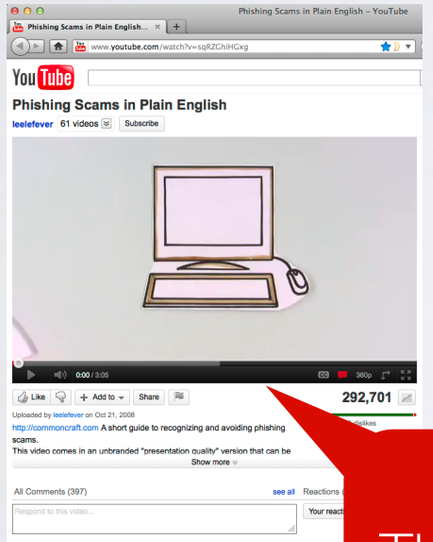


GET /?videoid=527

<html ...

comment = “<a href=”myad.com”>Fun stuff ...





GET /?videoid=527

<html ...

The page contains the attacker’s ad.

** Notice that Youtube is **not** vulnerable to this attack*

Problem

- ➡ An attacker can inject HTML tags in the page
  - Add illegitimate content to the webpage (ads most of the time)

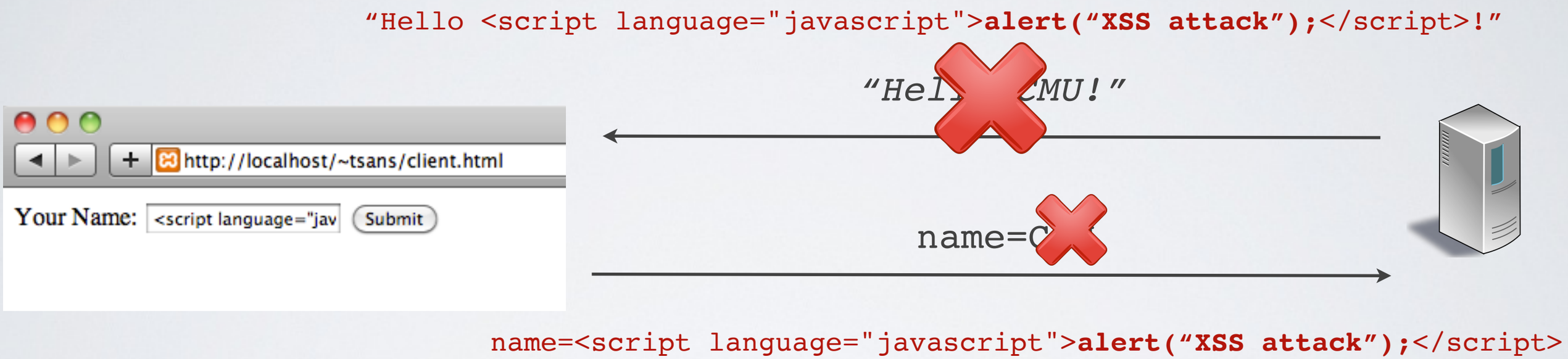
Generic Solution

- ✓ Data inserted in the DOM must be validated

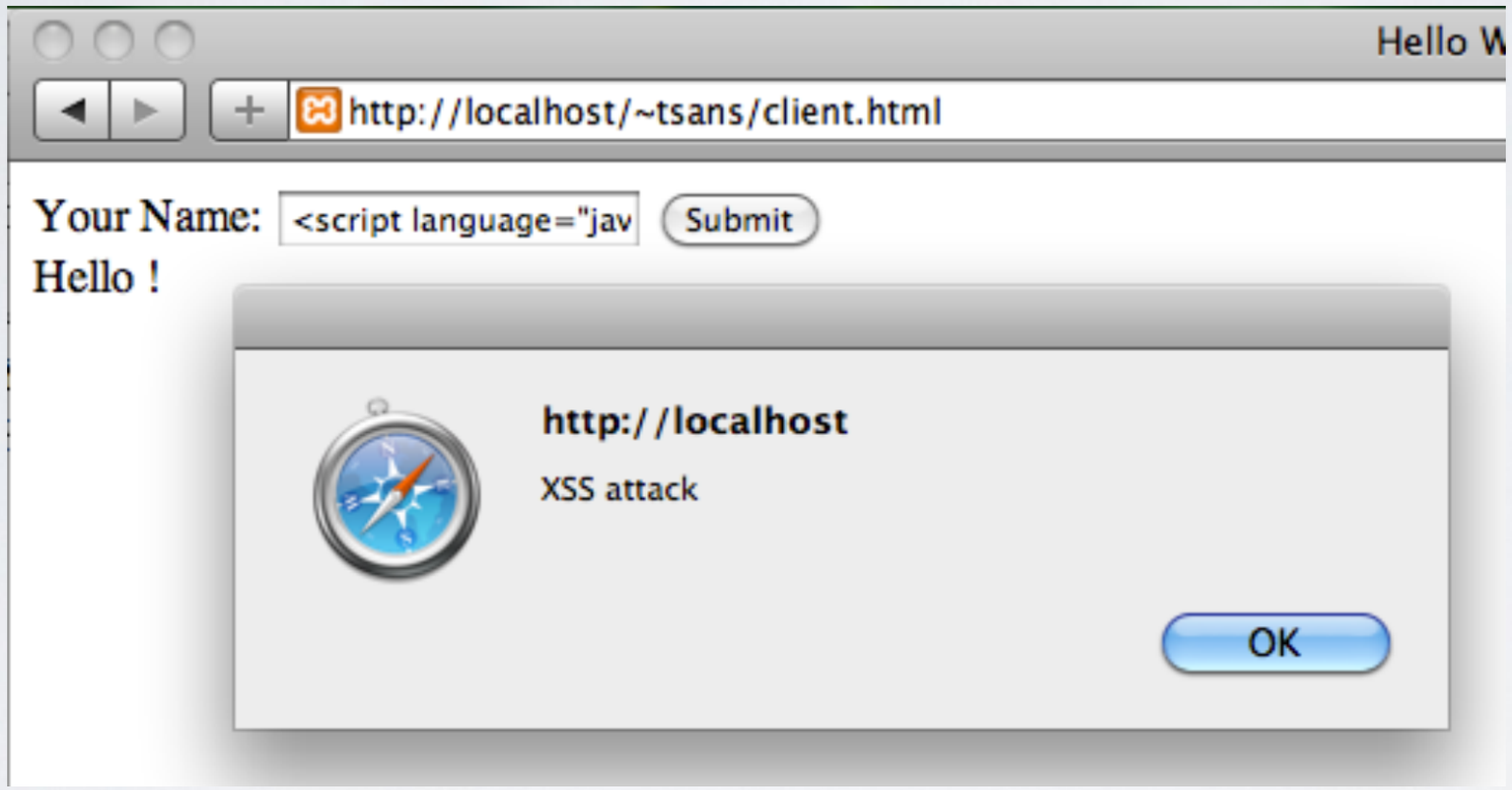
Frontend Vulnerability

Cross-Site Scripting (XSS)

Cross-Site Scripting Attack (XSS attack)



XSS Attack = Javascript Code Injection

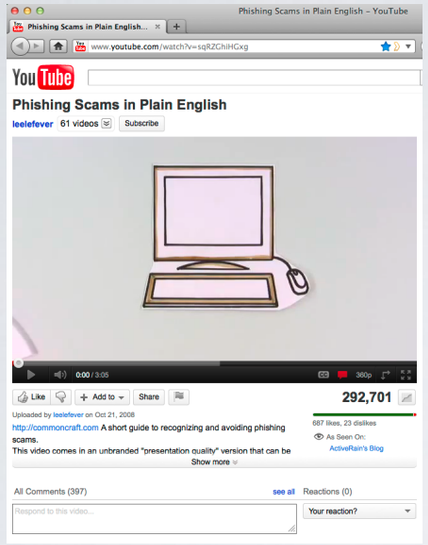


Problem

➡ An attacker can inject **arbitrary javascript code** in the page that will be executed by the browser

- **Inject illegitimate content** in the page (same as content spoofing)
- **Perform illegitimate HTTP requests** through Ajax (same as a CSRF attack)
- **Steal Session ID** from the cookie
- **Steal user's login/password** by modifying the page to forge a perfect scam

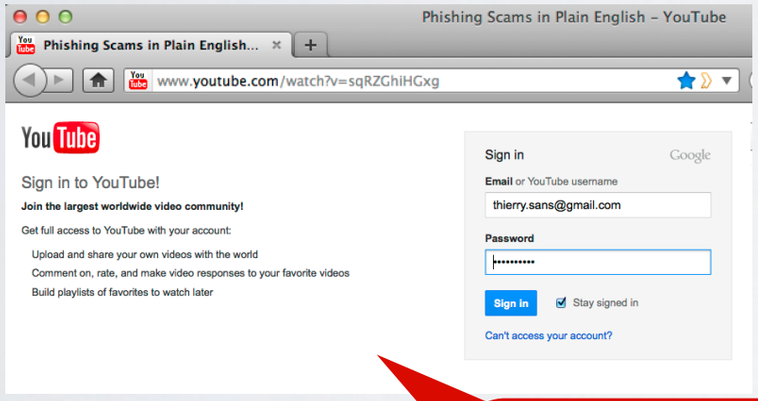
Forging a perfect scam



GET /?videoid=527

<html ...

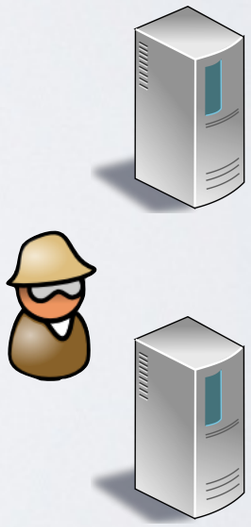
comment = “<script> ...



GET /?videoid=527

<html ...

login=Alice&password=123456



The script contained in the comments modifies the page to look like the login page!

** Notice that Youtube is **not** vulnerable to this attack*

It gets worst - XSS Worms

Spread on social networks

- Samy targeting MySpace (2005)
- JTV.worm targeting Justin.tv (2008)
- Twitter worm targeting Twitter (2010)

Variations on XSS attacks

- **Reflected XSS**
  Malicious data sent to the backend are immediately sent back to the frontend to be inserted into the DOM

- **Stored XSS**
  Malicious data sent to the backend are store in the database and later-on sent back to the frontend to be inserted into the DOM

- **DOM-based attack**
  Malicious data are manipulated in the frontend (javascript) and inserted into the DOM

Solution

- ✓ Data inserted in the DOM must be validated

`HttpOnly` cookie flag

✓ The cookie is not readable/writable from the frontend

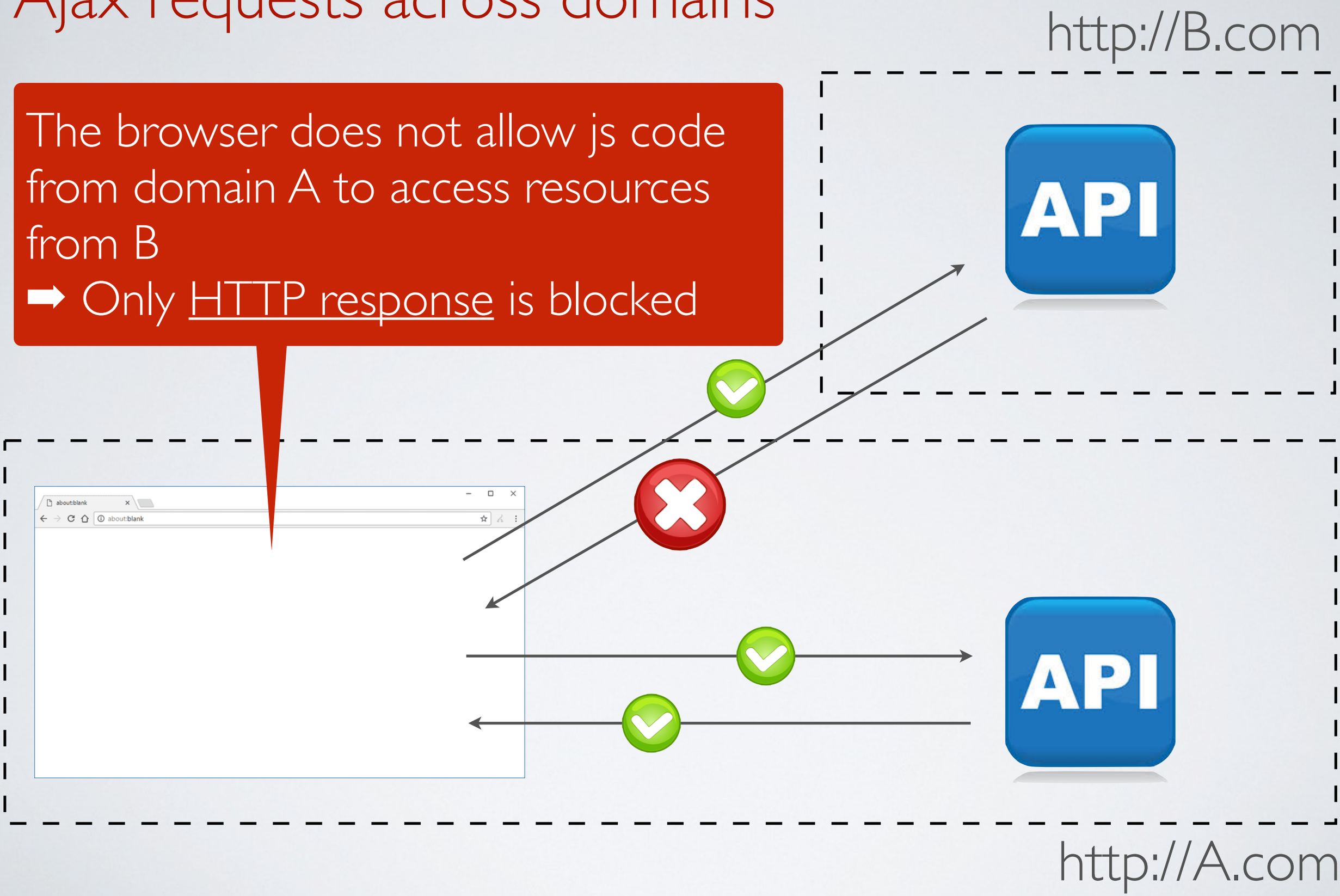
➡ Prevents the authentication cookie from being leaked when an XSS attack (cross-site scripting) occurs

Cross-Site Request Forgery

Ajax requests across domains

The browser does not allow js code from domain A to access resources from B

➡ Only HTTP response is blocked



Same origin policy

➡ **Resources must come from the same domain (protocol, host, port)**

Elements under control of the same-origin policy

- Ajax requests
- Form actions

Elements **not** under control of the same-origin policy

- Javascript scripts
- CSS
- Images, video, sound
- Plugins

Examples

	client	server
same protocol, port and host	`http://example.com`	`http://example.com`
	`http://user:pass@example.com`	`http://example.com`
top-level domain	`http://example.com`	`http://example.org`
host	`http://example.com`	`http://other.com`
sub-host	`http://www.example.com`	`http://example.com`
sub-host	`http://example.com`	`http://www.example.com`
port	`http://example.com:3000`	`http://example.com`
protocol	`http://example.com`	`https://example.com`

[digression] relaxing the same-origin policy

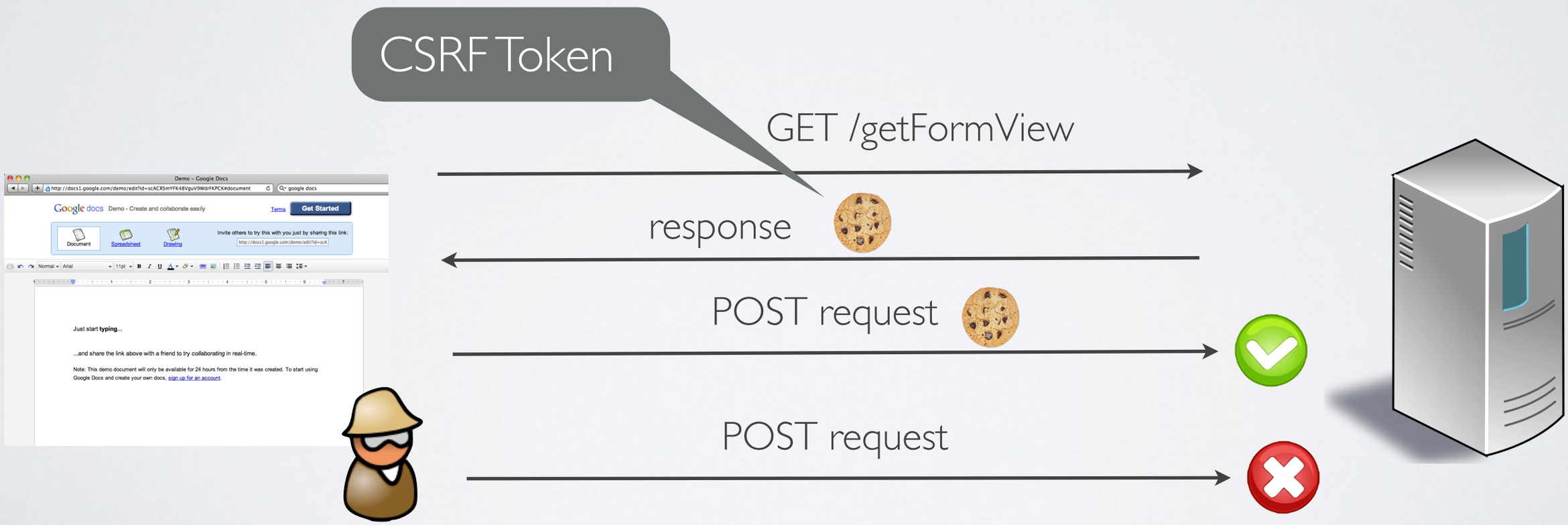
- Switch to the superdomain with javascript
  `www.example.com` can be relaxed to `example.com`
- iframe
- JSONP
- Cross-Origin Resource Sharing (CORS)

Problem

- ➡ An attacker can executes unwanted but yet authenticated actions on a web application by setting up a malicious website with cross-origin requests

Generic solution - CSRF tokens

✓ Protect legitimate requests with a CSRF token



`SameSite` cookie flag

✓ The cookie will be not be sent over cross-site requests

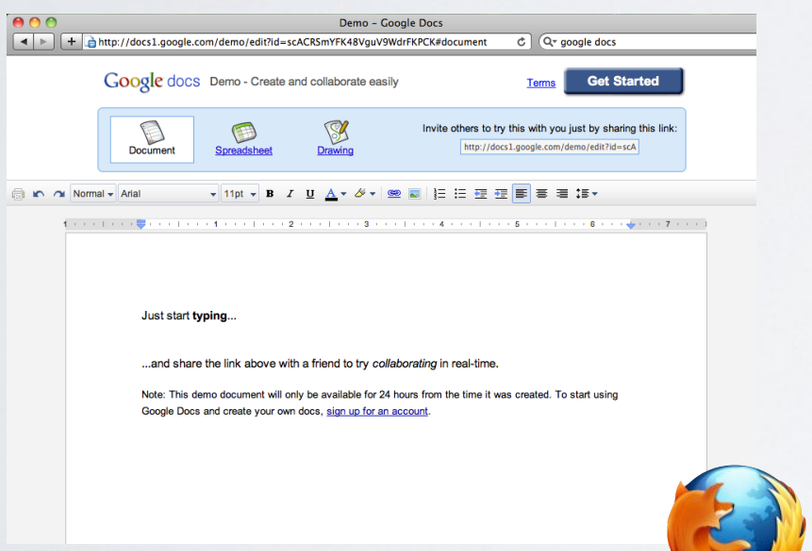
➡ Prevents forwarding the authentication cookie over cross-origin requests (cross-site request forgery)

Conclusion

You have **absolutely no control** on the client

Client Side

Server Side



Web Browser

Web Server

Database

References

- OWASP Top 10
  https://owasp.org/www-project-top-ten/
- Mozilla Secure Coding Guideline
  https://wiki.mozilla.org/WebAppSec/Secure_Coding_Guidelines

https://owasp.org/Top10/

The 2021 OWASP Top 10 list

A01:2021
Broken Access Control

A02:2021
Cryptographic Failures

A03:2021
Injection

A04:2021
Insecure Design

A05:2021
Security Misconfiguration

A06:2021
Vulnerable and Outdated Components

A07:2021
Identification and Authentication Failures

A08:2021
Software and Data Integrity Failures

A09:2021
Security Logging and Monitoring Failures

A10:2021
Server-Side Request Forgery

- ➡ Risks are ranked according to the frequency of discovered security defects, the severity of the uncovered vulnerabilities, and the magnitude of their potential impacts