Web Security

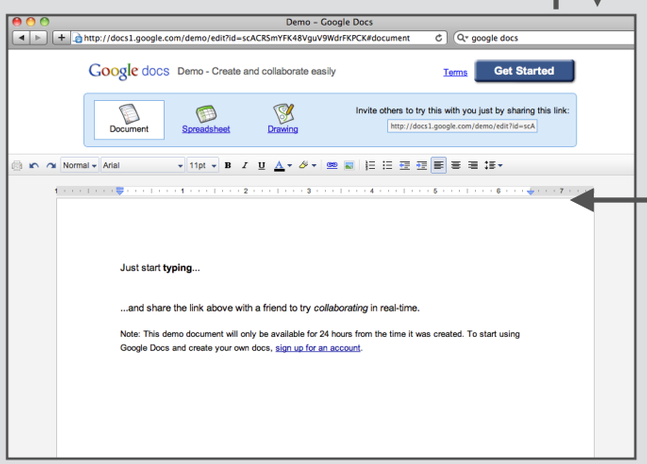
Thierry Sans

The big Picture

DNS server(s)

1.2.3.4

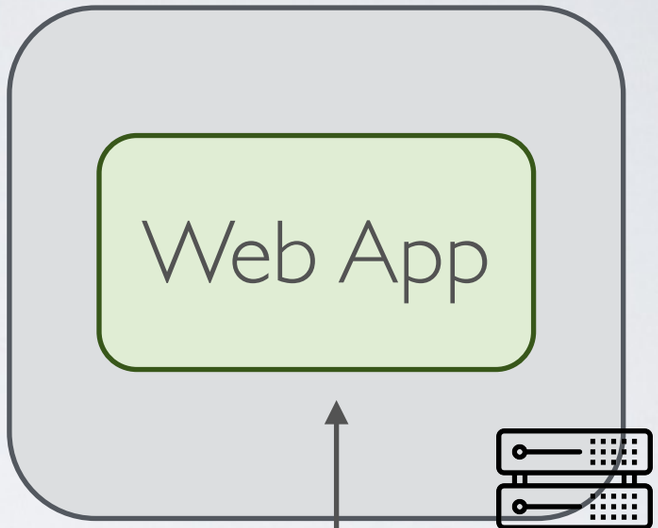
Who is example.com?



HTTP(S)

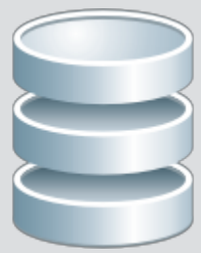
port 80/443

HTTP(S)



Third-Party API

5.6.7.8



1.2.3.4

Securing the web architecture means securing ...

- The domain name
- The server hosting the Web application
- The third-party applications (database for instance)
- The third-party access (API credentials for instance)
- The web application code (frontend and backend)

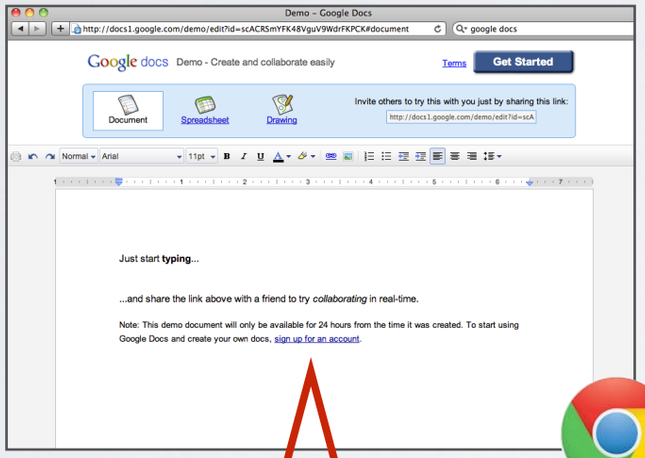
Our focus here!

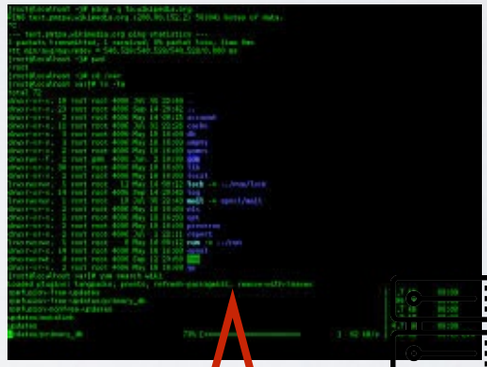
What is a web application?

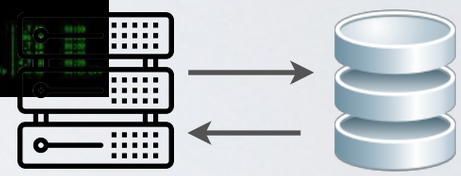
program running on the browser

+

program running on the server







The Threats

What we now about HTTP (from the networking challenges)

HTTP Request

- the query string (a.k.a the URL)
- the HTTP method (`GET`, `POST`, `PUT`, `PATCH`, `DELETE`)
- the headers
- the (optional) body

HTTP Response

- the status code
- the headers
- the body

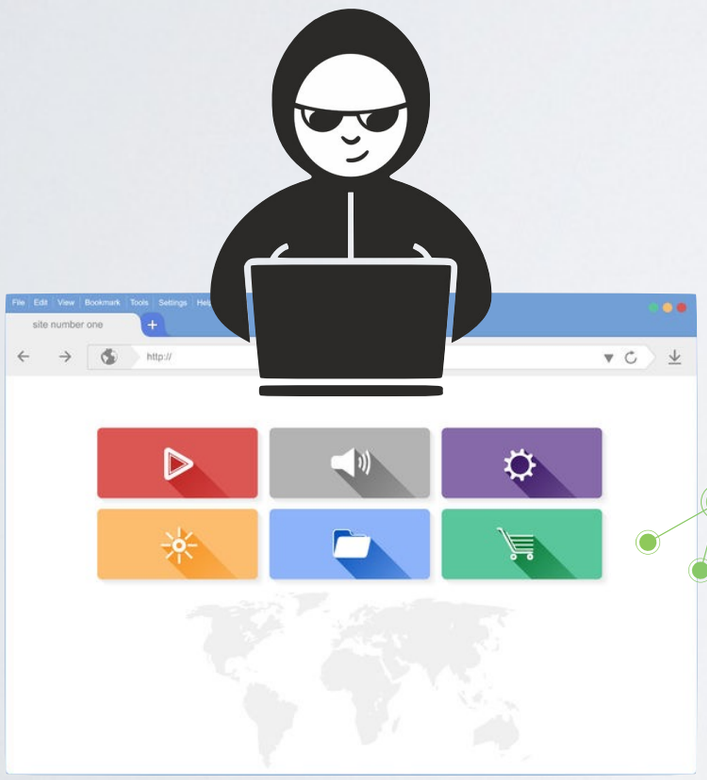
- An attacker might want to **eavesdrop or spoof** the content of HTTP(S) requests and responses

The Attacker Model

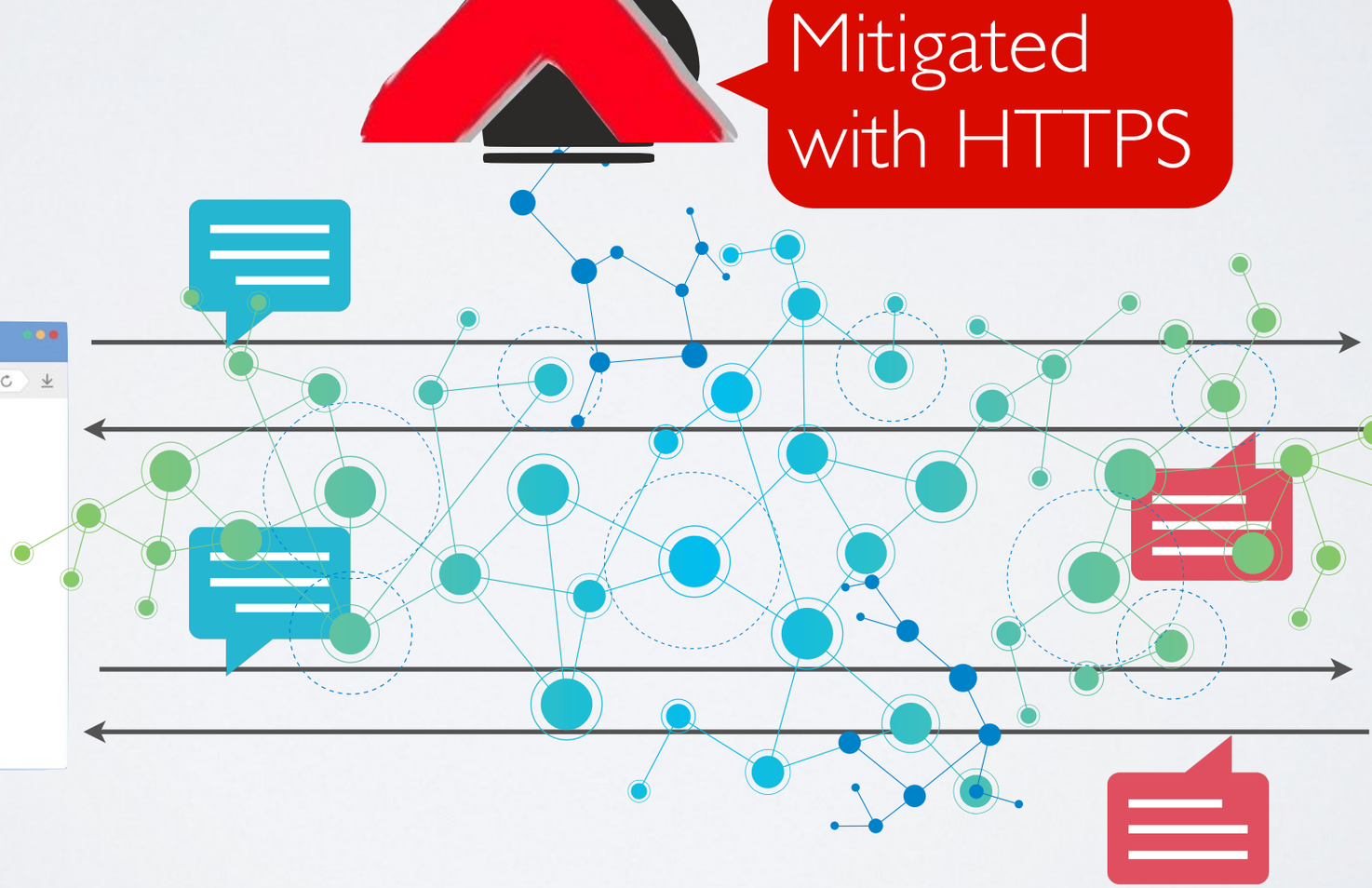
Man-in-the-Middle

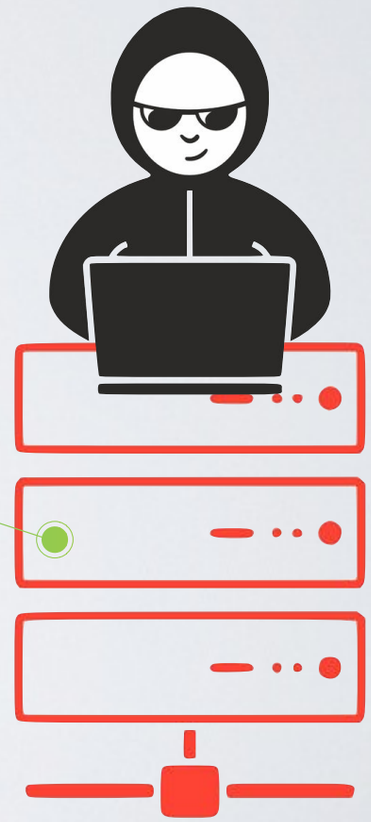
Man-in-the-Browser

Man-on-the-Server



Mitigated with HTTPS





Vulnerabilities By Types/Categories

CVEdetails.com assigns types/categories to vulnerabilities using CWE ids and keywords.

Web Vulnerabilities

Year	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	File Inclusion	CSRF	XXE	SSRF	Open Redirect	Input Validation
2015	343	1093	216	773	146	3	248	49	8	46	0
2016	418	1096	85	476	90	4	85	39	15	28	0
2017	2469	1539	505	1500	281	154	334	109	57	97	930
2018	2076	1729	503	2039	569	112	479	188	118	85	1228
2019	1202	2005	544	2387	485	126	559	136	103	121	894
2020	1216	1846	464	2201	436	108	414	119	130	100	807
2021	1656	2509	741	2723	546	89	520	126	188	133	666
2022	1782	2848	1762	3370	686	86	766	123	230	137	663
2023	1594	2007	2116	5101	742	108	1392	124	239	168	513
2024	1723	2359	2646	7434	923	244	1433	110	372	113	99
2025	1947	2419	3485	7670	893	493	1702	99	479	141	0
Total	16426	21450	13067	35674	5797	1527	7932	1222	1939	1169	5800

source: cvedetails.com

https://owasp.org/Top10/

The 2021 OWASP Top 10 list

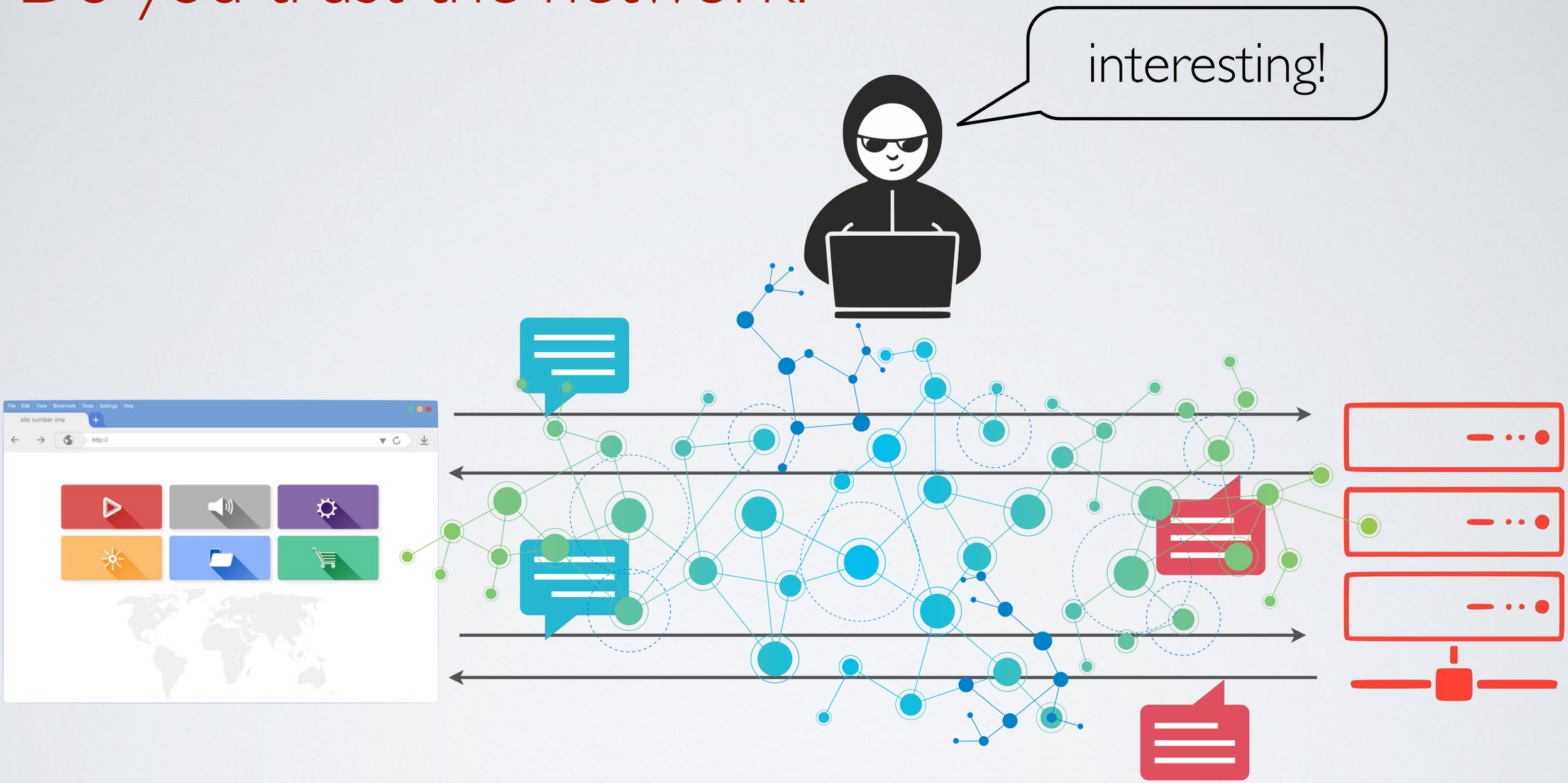
A01:2021 Broken Access Control	**A02:2021** Cryptographic Failures	**A03:2021** Injection	**A04:2021** Insecure Design	**A05:2021** Security Misconfiguration
A06:2021 Vulnerable and Outdated Components	**A07:2021** Identification and Authentication Failures	**A08:2021** Software and Data Integrity Failures	**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

A02 Cryptographic Failure

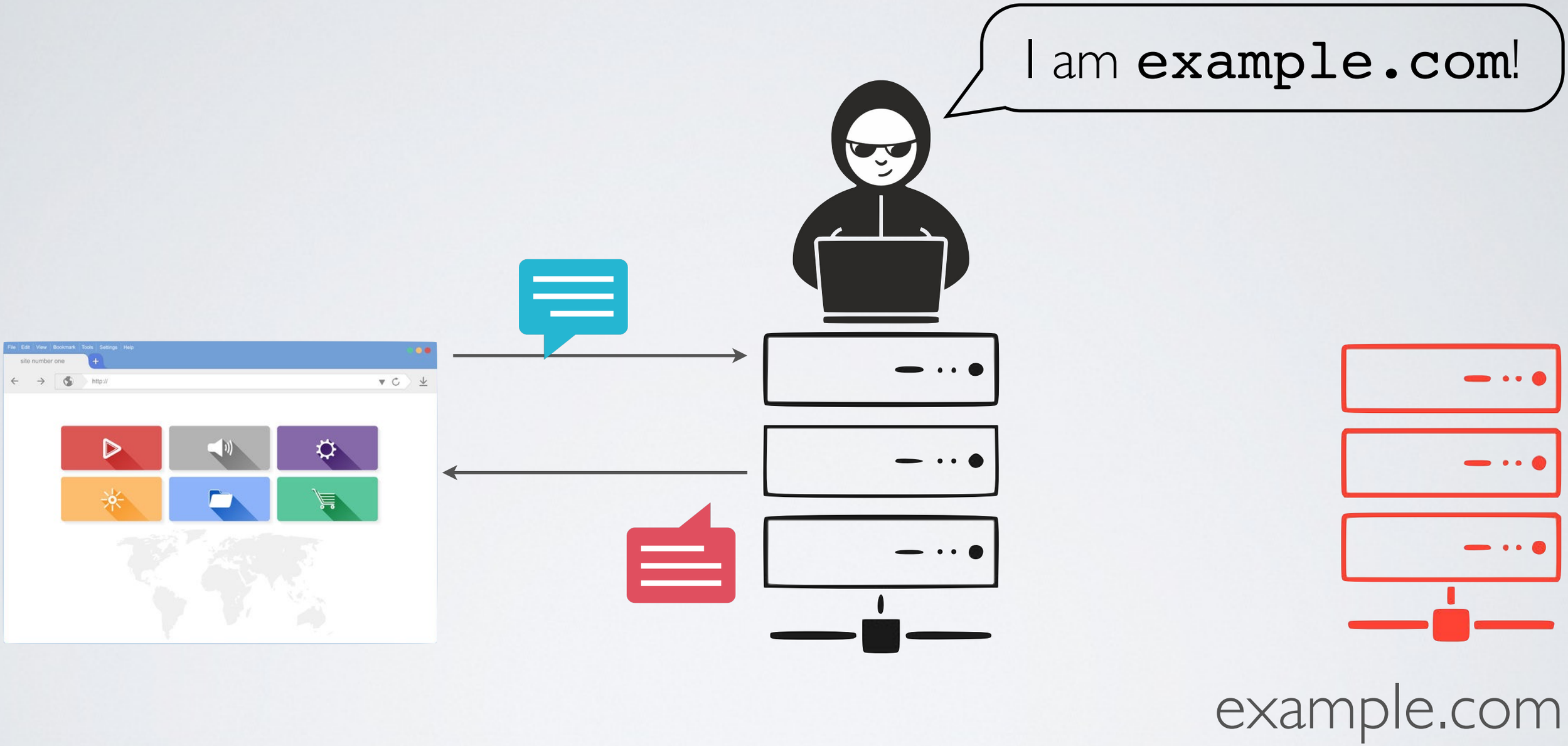
Insufficient Transport Layer Protection

Do you trust the network?



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

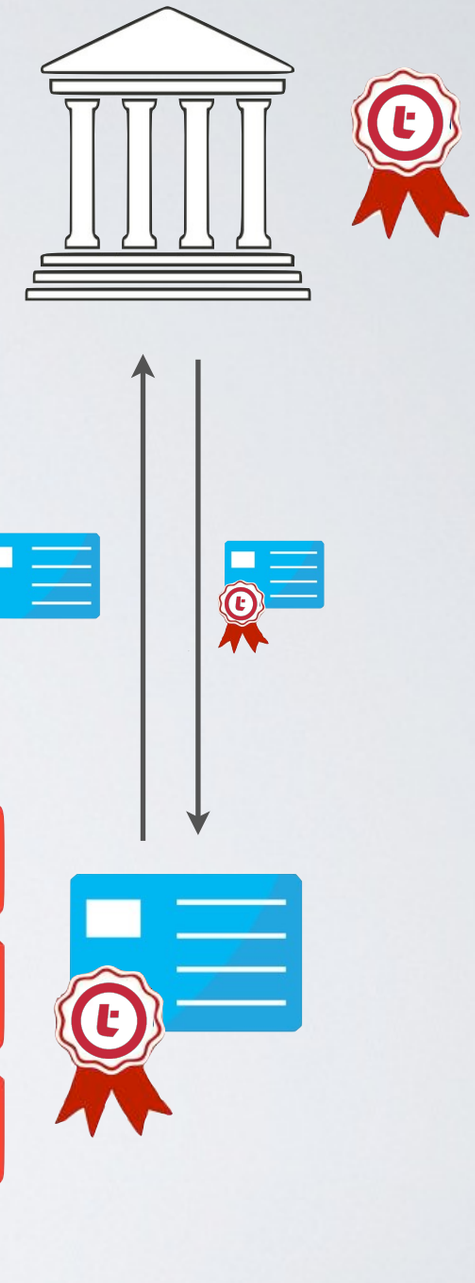
Do you *really* trust the network?

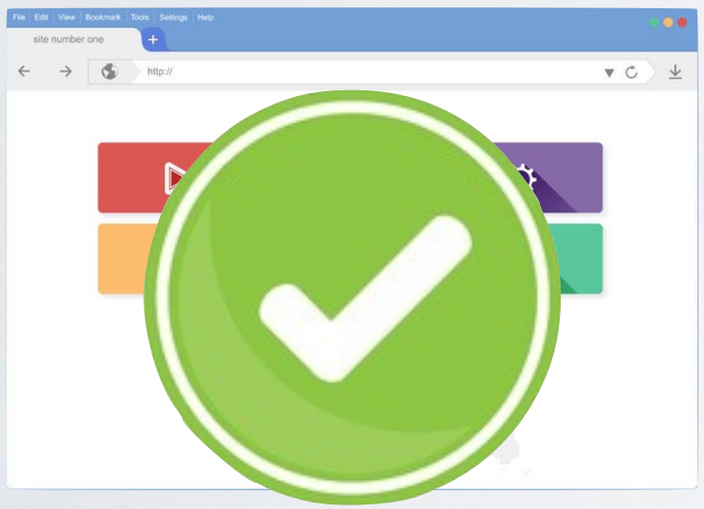


- Threat 2 : an attacker **can tamper with** messages sent back and forth

HTTPS

Certificate Authority (CA)





TLS Handshake

AE Session

HTTPS in practice

The Web Application owner must

1. Acquire a domain name through a *Domain Name Registrar*
2. Configure the DNS to associate the domain name with IP address of the server
3. Generate a certificate and request the *Certificate Authority* to sign it

   ➡ The CA will verify the domain ownership

   ➡ The certificate must be renewed periodically (usually every year)

Why and when using HTTPS?

HTTPS = HTTP + TLS

➡ TLS provides

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

✓ **HTTPS everywhere**
HTTPS must be used for all requests and response without any exception

The problem of **Mixed-Content**

Mixed-content happens when:

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

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

- Authentication credentials (a.k.a cookies) will be sent over HTTP
- Modern browsers block (or warn about) 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

Beyond HTTPS - Attacking the Web Application

Attacking the **Frontend** (Man-in-the-Browser)

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

Attacking the **Backend** (Man-on-the-Server)

- **Broken Access Control** (a.k.a incomplete mediation)
- **SQL Injection**

A01 Broken Access Control

Incomplete Mediation

Broken Access Control

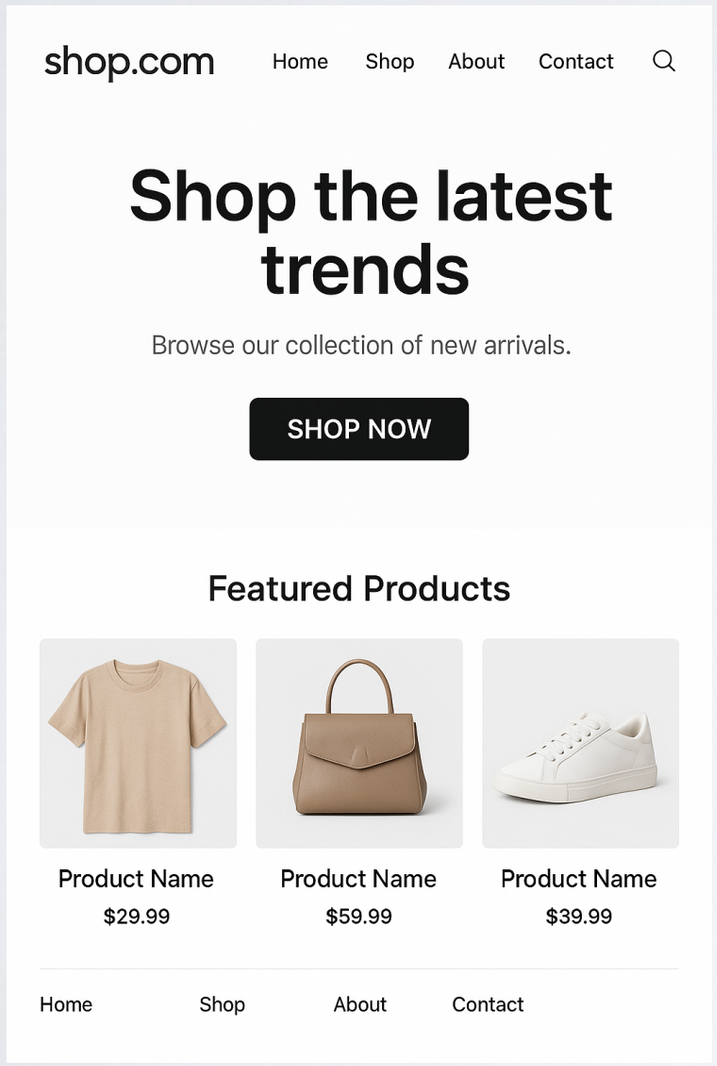
"AT&T Inc. apologized to Apple Inc. iPad 3G tablet computer users whose **e-mail addresses were exposed during a security breach** disclosed last week."

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*

Incomplete Mediation - The Shopping Cart Attack



The total is calculated by a script on the client

The order is generated based on the request

shop.com

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

Thank you for your order!

The backend is the **only trusted domain**

- ◉ Data coming from the frontend cannot be trusted

- ✓ Sensitive operations must be done on the backend

A03 Injection

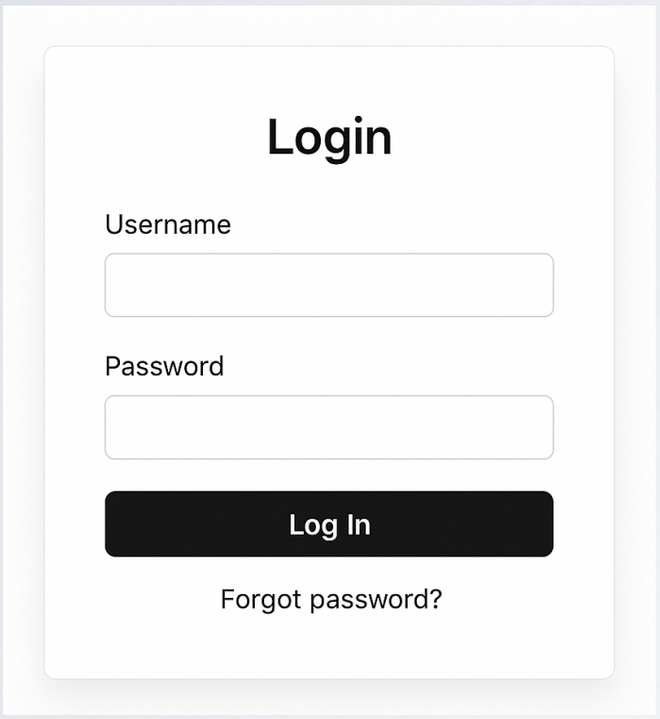
SQL Injection

Problem

➡ An attacker can inject SQL/NoSQL code

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

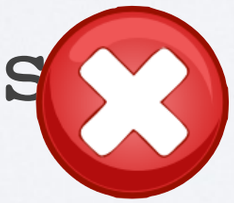
Checking password



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    });
```

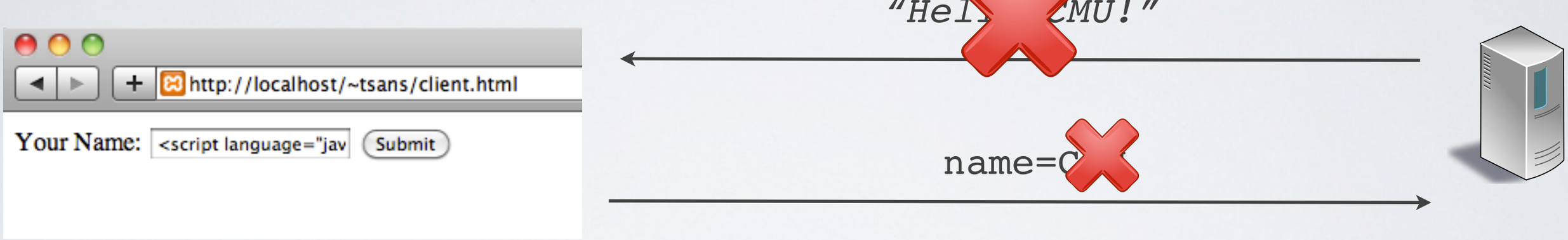
```
username: alice
password: pas lice
```

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

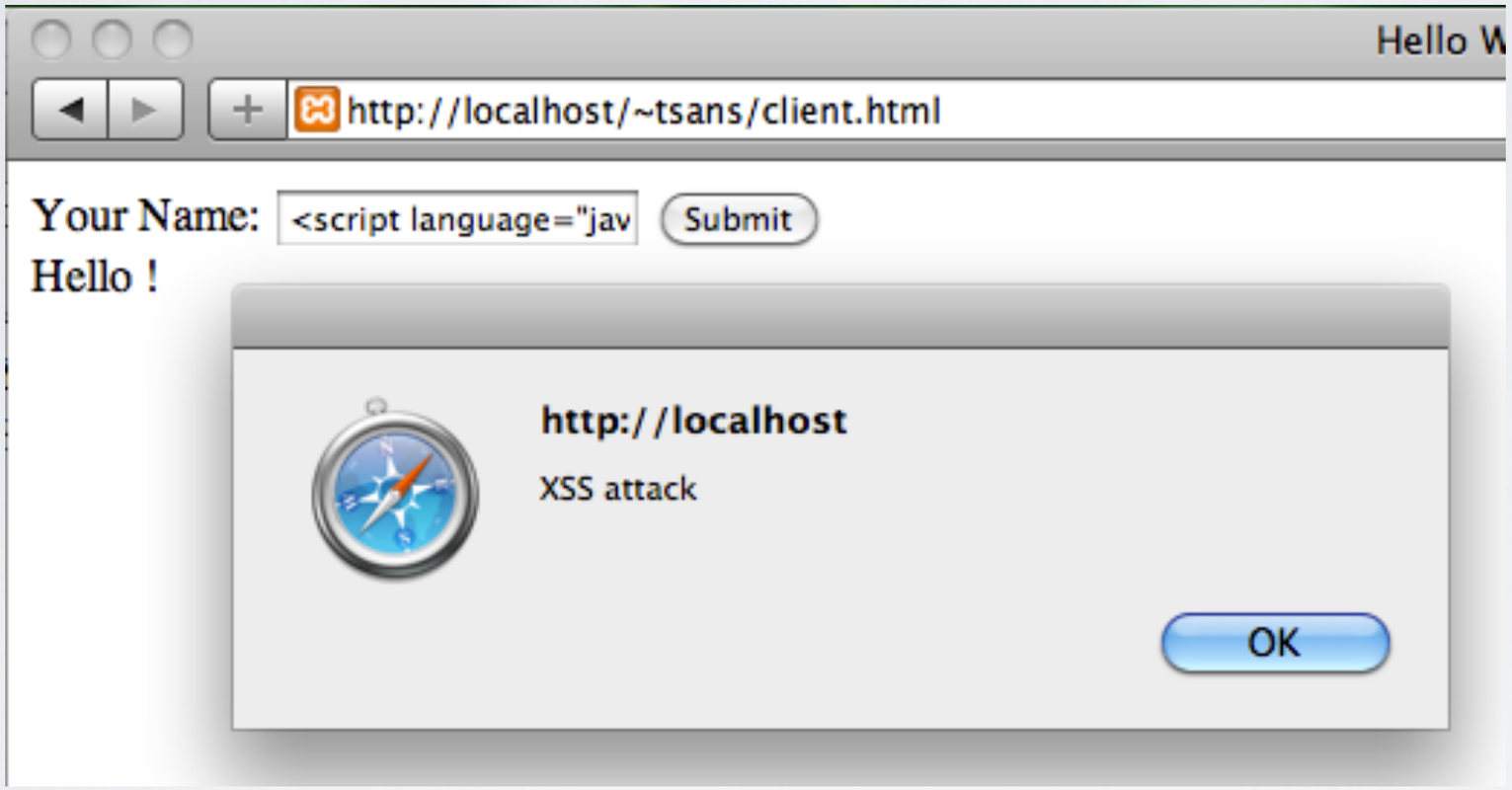
A03 Injection

Cross-Site Scripting (XSS)

Cross-Site Scripting Attack (XSS attack)



XSS Attack = Javascript Code Injection

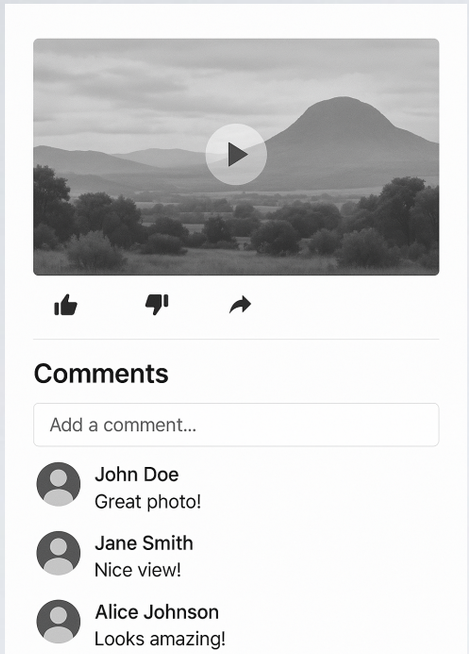


Problem

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

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

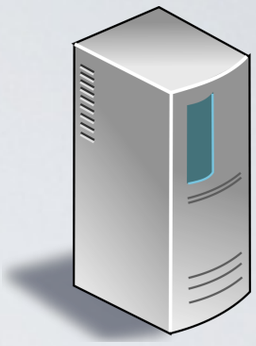
Forging a perfect scam

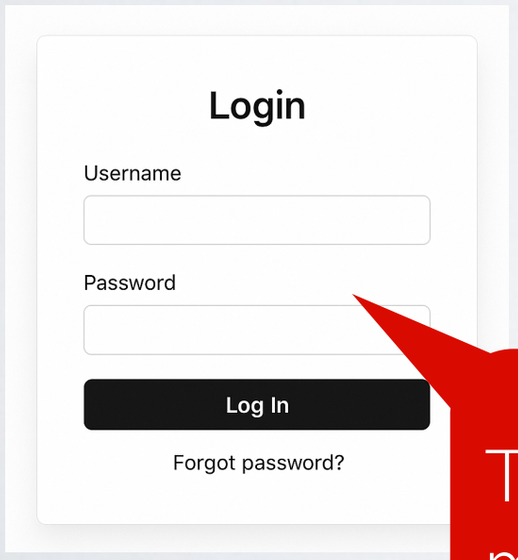


GET /?videoid=527

<html ...

comment = “<script> ...

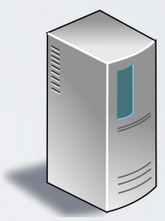




GET /?videoid=527

<html ...

login=Alice&password=123456



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

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

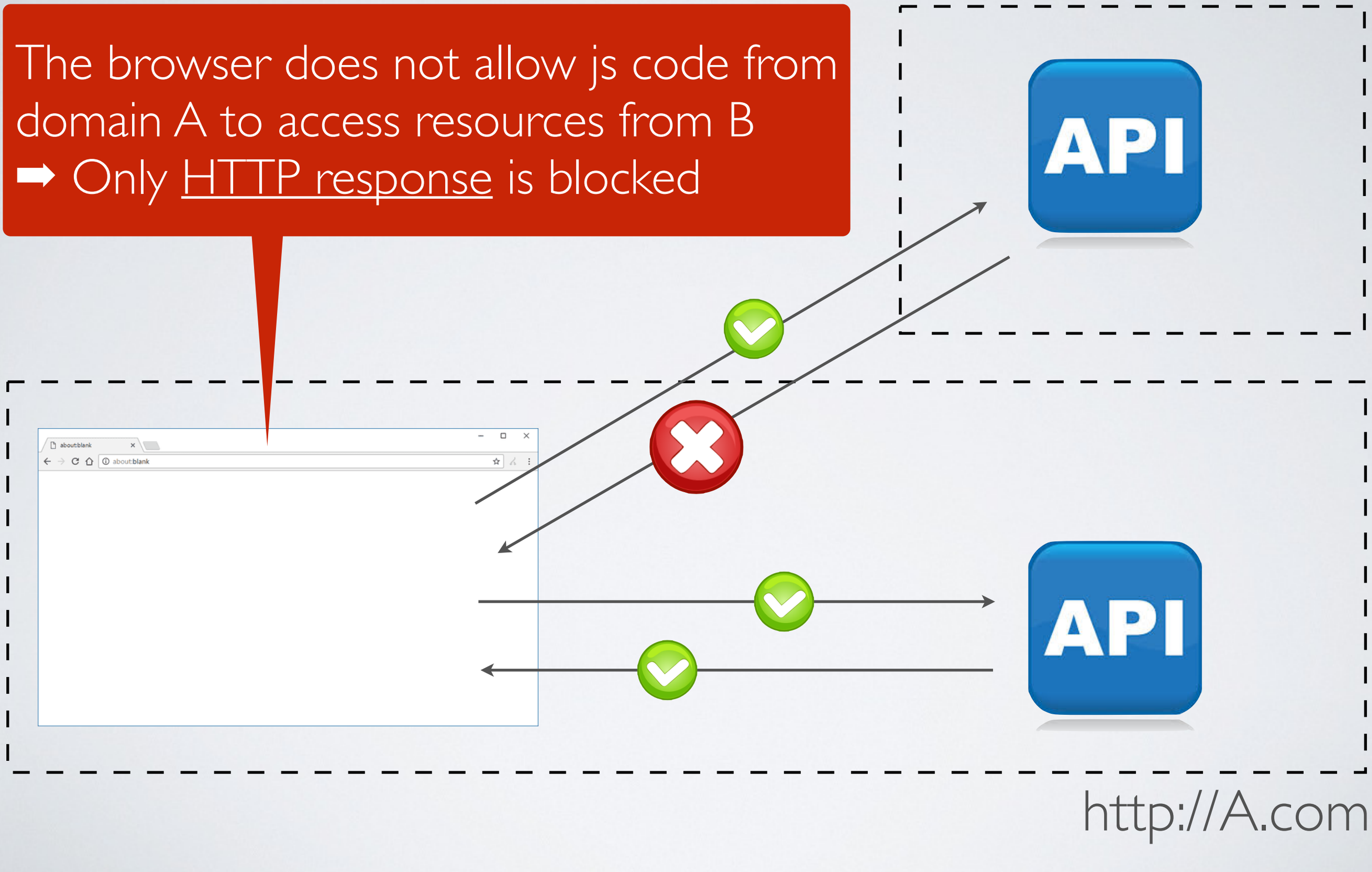
A01 Broken Access Control

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

- Fetch requests
- Form actions

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

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

Examples

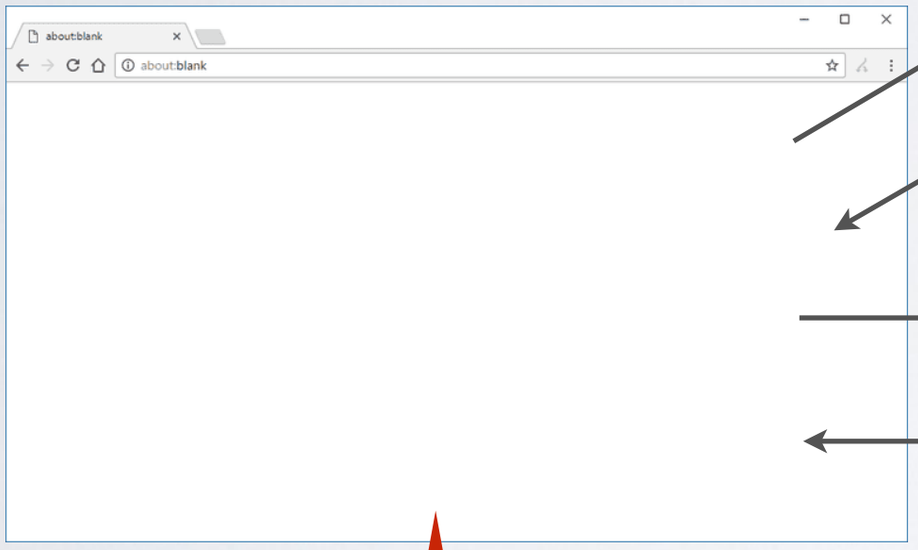
	client	server
same protocol, port and host	`http://example.com`	`http://example.com`
	`http://user:pass@example.com`	`http://example.com`
top-level domain	`http://example.com`	`http://example.org`
host	`http://example.com`	`http://other.com`
sub-host	`http://www.example.com`	`http://example.com`
sub-host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

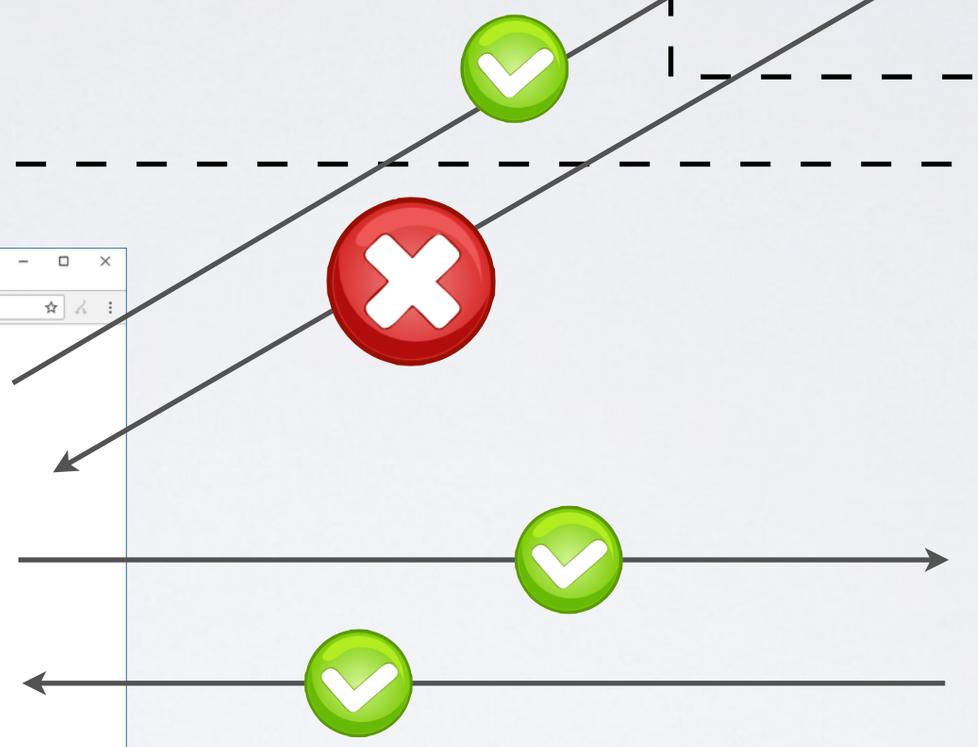
Cross-Site Request Forgery

http://B.com

Examples

```
POST /transfer/ {to: Mallory, amount: $1000)
DELETE /account/ {owner: Mallory}
```





http://A.com

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

`SameSite` cookie flag

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

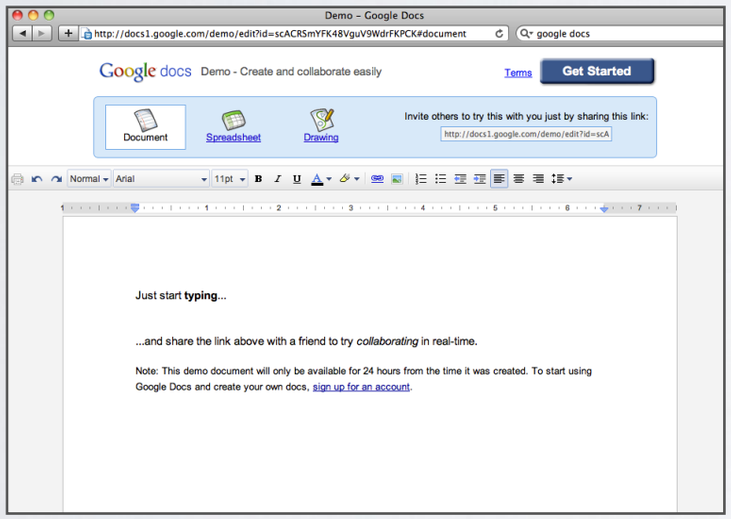
➡ Prevents forwarding the authentication cookie over cross-origin requests

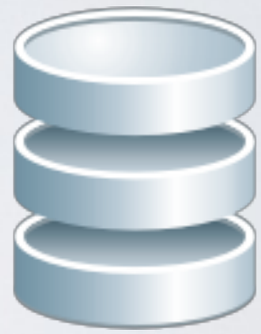
Conclusion

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