# Web Security

Jose L. Muñoz-Tapia

Universitat Politècnica de Catalunya (UPC)

# Web Security

# Outline

Symmetric Cryptography

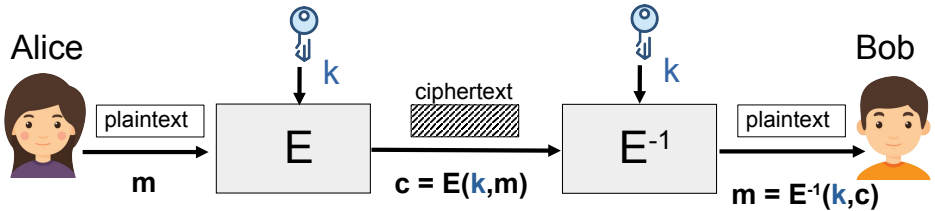
Public Key Cryptography

Hybrid Cryptography

Digital Certificates

HTTPS Concept

CORS




- Symmetric cryptography is the basic tool for implementing confidentiality.
- $E$ and $E^{-1}$ are identical in many symmetric algorithms… but not always.
- So, why then is this scheme called **symmetric**?

# Outline

Symmetric Cryptography

Public Key Cryptography

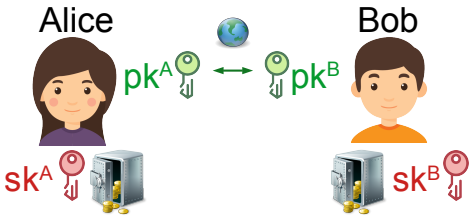
Hybrid Cryptography

Digital Certificates

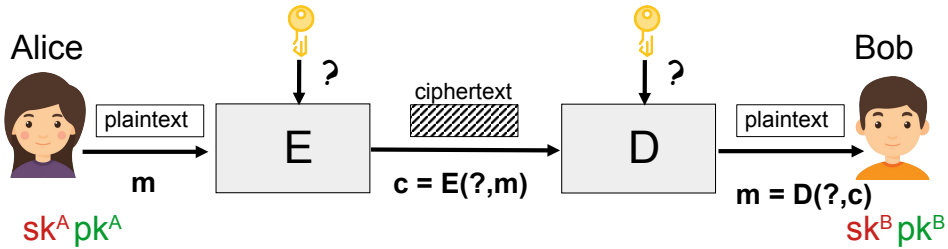
HTTPS Concept

CORS

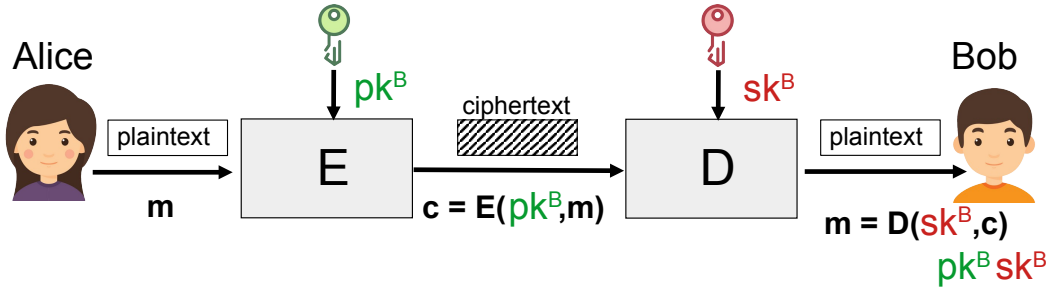
In Public Key Cryptography (PKC), the sender and receiver do **not need to share a secret**.




- Each of them has a key pair:
  - One key is PUBLIC (everybody must know that it belongs to the user).
  - The other is PRIVATE (kept secret by the user).
- One key is used for encryption and the other for decryption (asymmetric algorithm).

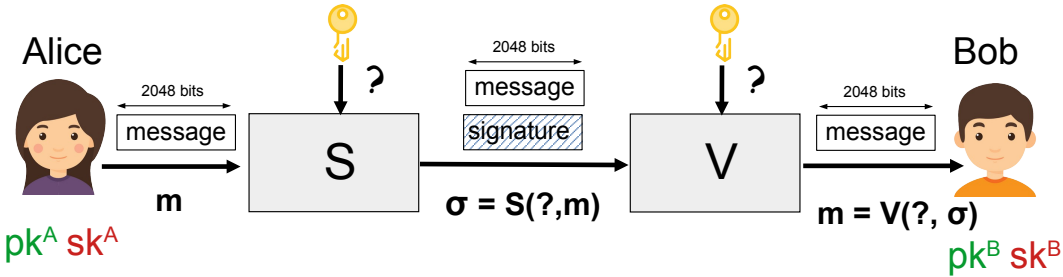



Which keys should we use to send a confidential message from Alice to Bob?

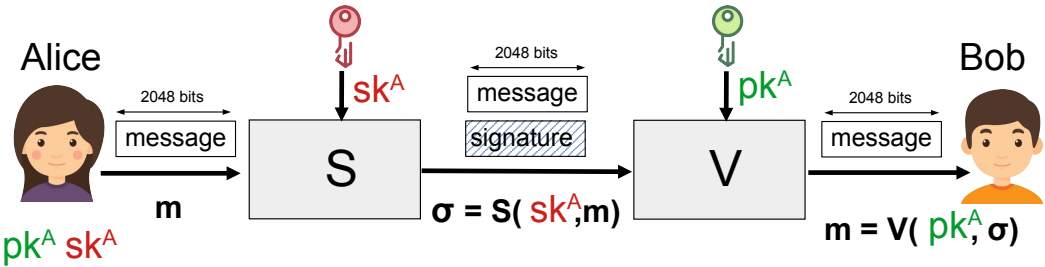



As mentioned, we need to use the keys from **the same pair**.

Public key cryptography is also called **asymmetric cryptography** because different keys are used for encryption and decryption.




Which keys should we use to send the message and its digital signature from Alice to Bob?




Digital signatures provide two security services: integrity and authentication.

**Authentication** is the act of proving an assertion, in this case the identity of the sender which is represented by a public key.

Notice that by the moment, we are not interested in confidentiality.

# Outline

Symmetric Cryptography

Public Key Cryptography

Hybrid Cryptography

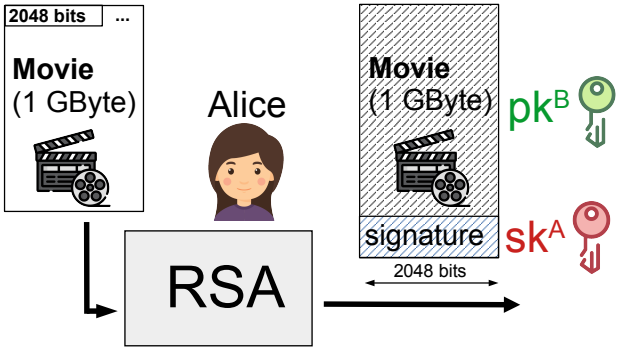
Digital Certificates

HTTPS Concept

CORS

# How to Confidentially Send Our Movie

1. We can split the file in blocks.
2. Encrypt block by block with the destination's public key.

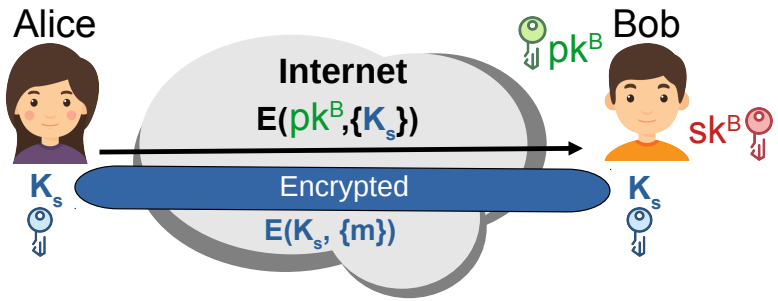



Is this a good solution?

Why symmetric cryptography is still used today?

# Performance & Hybrid Cryptography i

- Public key encryption is much more costly than symmetric encryption
  - Several orders of magnitude more expensive in terms of CPU.
  - Keys should be larger to provide the same level of security (AES 128 bits, RSA 3072).
- The solution to get the best from both types of cryptography is to mix them:
  This approach is called **hybrid cryptography**.




- In **hybrid cryptography**, we use asymmetric cryptography to exchange a symmetric session key.
- Then, the bulk data is symmetrically encrypted.

# Outline

Symmetric Cryptography

Public Key Cryptography

Hybrid Cryptography

Digital Certificates

HTTPS Concept

CORS

# Using PKC Safely

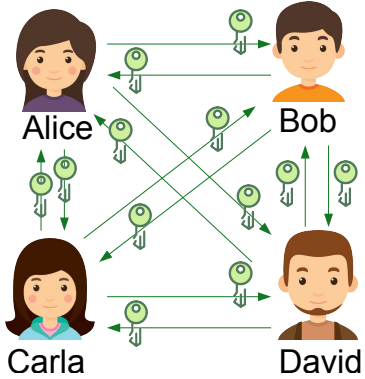
To use public key cryptography safely, we need:

1. To keep secret our secret key.
2. We must be sure that the public key that we are using to send confidential data or to verify a digital signature **belongs to the intended identity**.

**How we manage the binding between an identity and a public key?** This is the problem of public keys announcements.

# Web of Trust Approach

- As time goes on, you will accumulate keys from other people that you may want to designate as trusted **introducers**.
- Everyone else will each choose their own trusted introducers.
- Everyone will gradually accumulate and distribute with their key a collection of certifying signatures from other people.

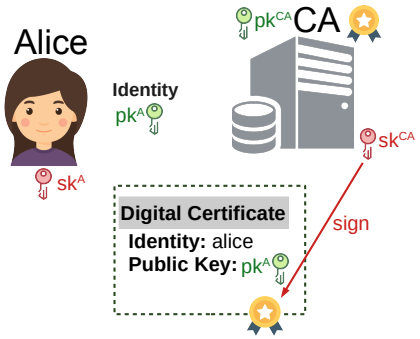



- The expectation is that anyone receiving it will trust at least one or two of the signatures.
- This will cause the emergence of a decentralized fault-tolerant web of confidence for all public keys.

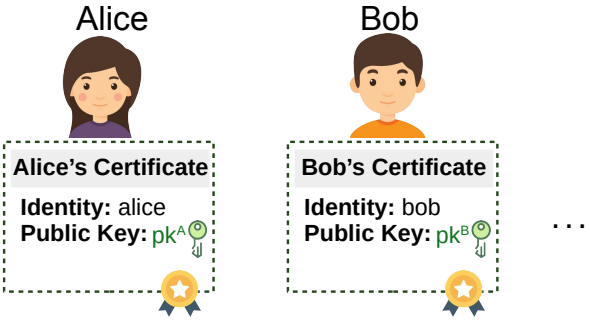
# Certification and Digital Certificates

- The web of trust has not been universally used.
- Users prefer to delegate the management of public keys.
- Specifically, the certification process is delegated to a Certification Authority (CA).
- The CA issues a digitally signed document binding an identity with a public key.

  - This document is known as **digital certificate**.
  - In general, digital certificates are not secret documents.

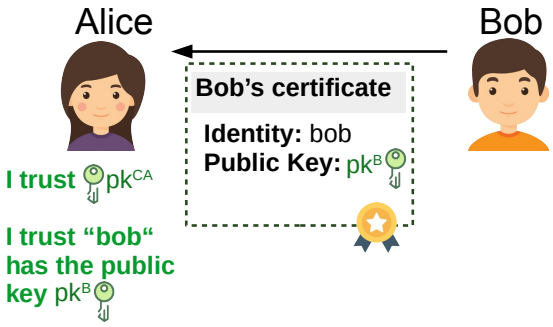


# Check a Digital Certificate i




1. If the CA is trusted by Alice (which means that it certifies correctly making sure that the identity is correct before signing the certificate)
2. if Alice correctly knows the public key of the CA
3. if Alice, using the public key of the CA, checks that the signature of Bob's certificate is correct

# Check a Digital Certificate ii



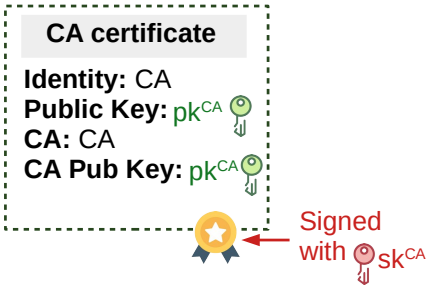

Then, **Alice can trust the identity of Bob**.

Notice that Alice just needs to trust **one public key** (CA's public key).

**By sending a certificate, is Bob authenticated?**

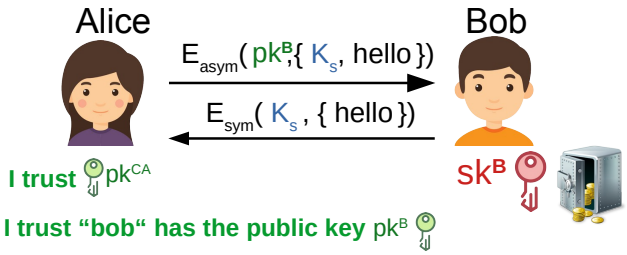
# Auto-signed, Root or CA Certificates

- Instead of distributing the public keys of certification authorities as a string of bits, CA public keys are also distributed using certificates.
- In this case, the certificate is signed with the private key of the public key that is being certified.




- These types of certificates are also called ”root” or CA certificates.
- This is because they define the root of the trust (other certificates are trusted because they are signed by another entity).

# Authentication with a Digital Certificate




- To authenticate itself, Bob needs to prove that he knows the private key associated with the public key in the certificate.
- As a simple example, Alice can send a symmetric session key together with something that needs to be encrypted like "hello" and Bob has to return this symmetrically encrypted.

How is a certificate? how can we express an identity?

# The X.509 Standard

- The X.509 standard defines how to create a digital certificate.
- Certificates include:
  - Fields to **describe the identity**.
  - A serial number.
  - A validity period.
  - The algorithm of the public key.
  - CA public key.
  - Algorithm used to sign the certificate.
  - Fields for the purpose of the certificate and more.

# X.509 Identity Fields

- The **identity** is expressed with:
  - Country Name.
  - State or Province Name.
  - Locality Name.
  - Organization Name. E.g, company.
  - Organizational Unit Name. E.g, section.
  - **Common Name**. E.g, server FQDN or YOUR name.
    For example, in Internet: `CN = www.mybank.com`
  - **Subject Alt Names**. These are also important, they are **alternative names**.

# Outline

Symmetric Cryptography

Public Key Cryptography

Hybrid Cryptography

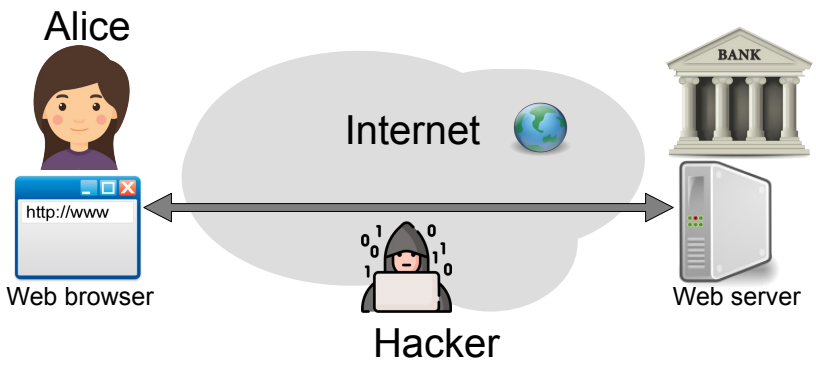
Digital Certificates

HTTPS Concept

CORS

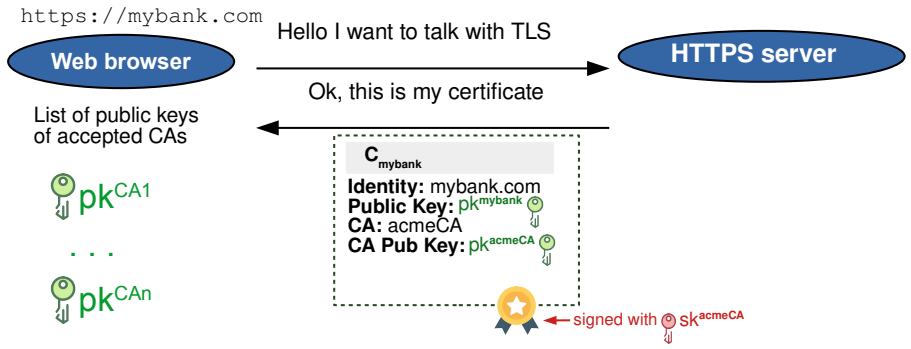
# How to Talk to our Bank?

Most of us use a Web application to establish a communication with our bank.




These applications are secured with hybrid cryptography and digital certificates.

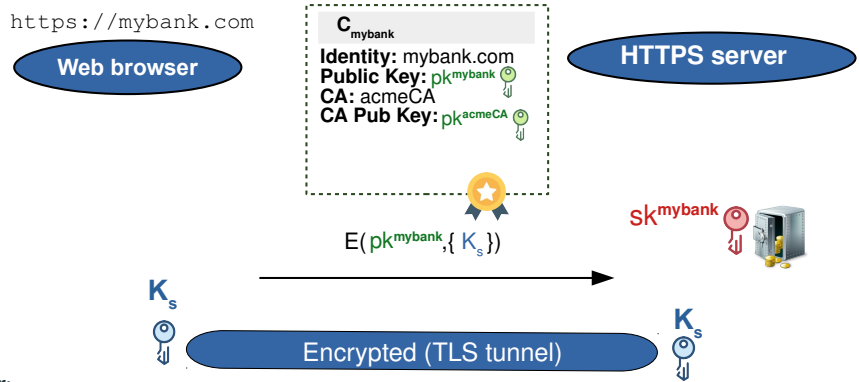
# Server Certificate




- The browser performs several security checks:
  1. Is the certificate signature correct?
  2. Do I recognize the public key of the CA?
  3. Is the certificate expired?
  4. Is the URL the same as the one in the certificate?
- You can test these checks in `https://badssl.com`

Are the previous checks enough to authenticate the server?

# The ”TLS Tunnel”




The browser:

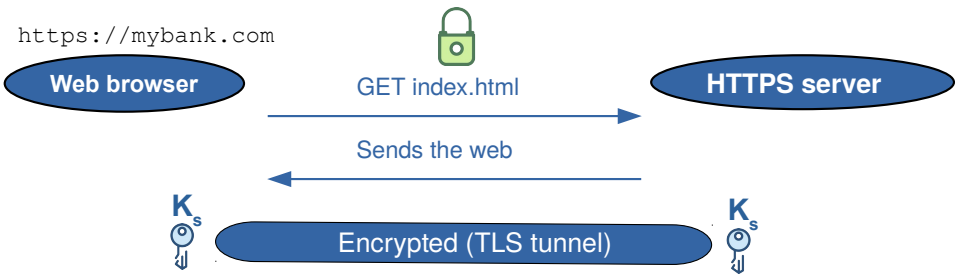
- Generates a random symmetric session key $K_S$ and sends it to the server.
- The session key is encrypted with the public key present in the server’s certificate.

An honest server:

- Will have the corresponding private key.
- Decrypts the message and gets $K_S$.
- A dishonest server cannot get $K_S$.

# Server Sends Content

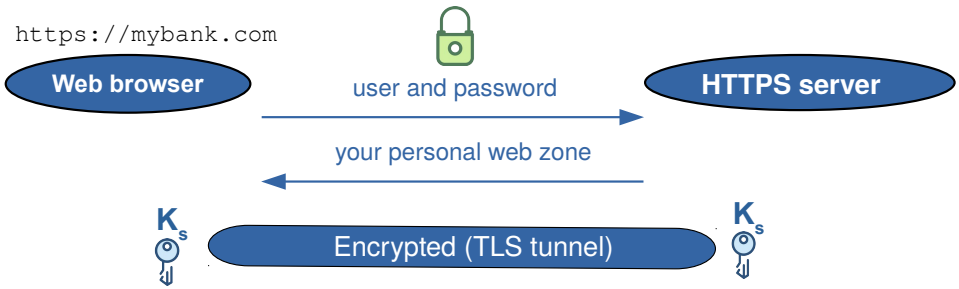
Over the TLS tunnel the server confidentially sends the Web page:




At this moment who is authenticated to who?

# Typical HTTPS User Authentication

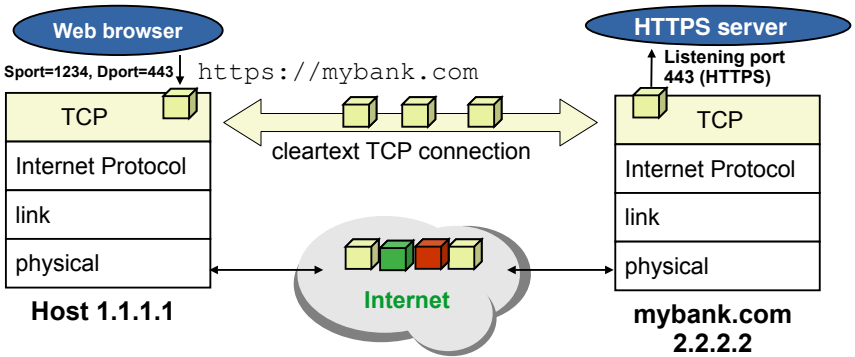
- The server was authenticated with a certificate.
- For users this is complex.
- Most solutions consists in just user and password:




- Notice that the user and password are only sent after the server is authenticated and through a secure channel (TLS tunnel).
- Be careful! look at the green padlock and that there is not any risk warning in your browser.

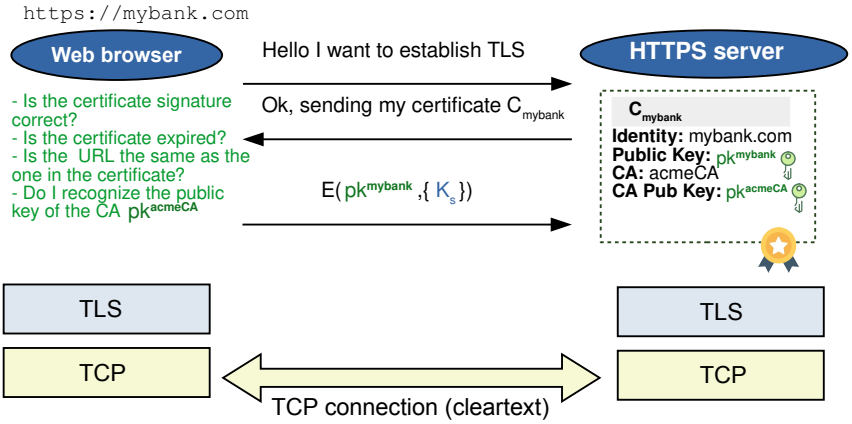
# HTTPS Protocol Layers i

First, a TCP connection is established (no encryption):



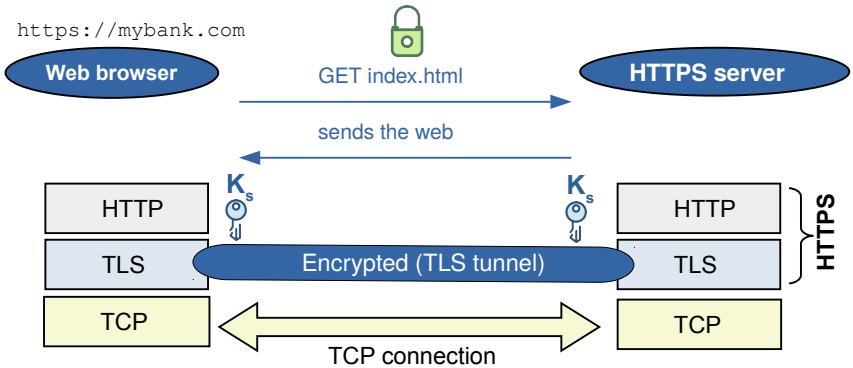
# HTTPS Protocol Layers ii

The TLS protocol establishes a secure "tunnel" over the TCP connection:



# HTTPS Protocol Layers iii

Inside the TLS tunnel the server confidentially sends the Web content:



# Outline

Symmetric Cryptography

Public Key Cryptography

Hybrid Cryptography

Digital Certificates

HTTPS Concept

CORS

# Origin

**Origin**

An origin is a tuple `(protocol, hostname, port)`

Examples (they are all different origins):

`http://foo.com/index.html (proto:HTTP,hostname:foo.com,Port:80)`

`https://foo.com (proto: HTTPS, hostname: foo.com, Port: 443)`

`http://foo.com:8080 (proto: HTTP, hostname: foo.com, Port: 8080)`

In general, JavaScript code can only access resources from its own origin, this is called **Single Origin Policy (SOP)**.

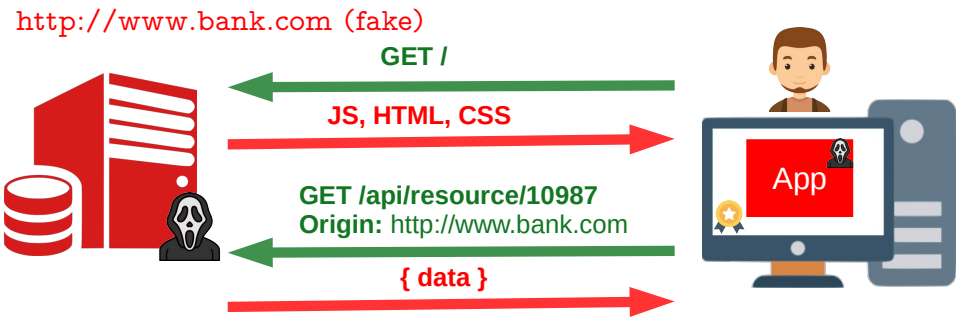
# Single Origin Policy (SOP)

- The browser is an absolute **trusted component**.
- We trust that **honest users have not hacked browsers**.
- Correct browsers know **the origin of each Web application that they are executing**.
- **Single Origin Policy (SOP)** means that a server will only provide resources to its application code.

# First Try with SOP

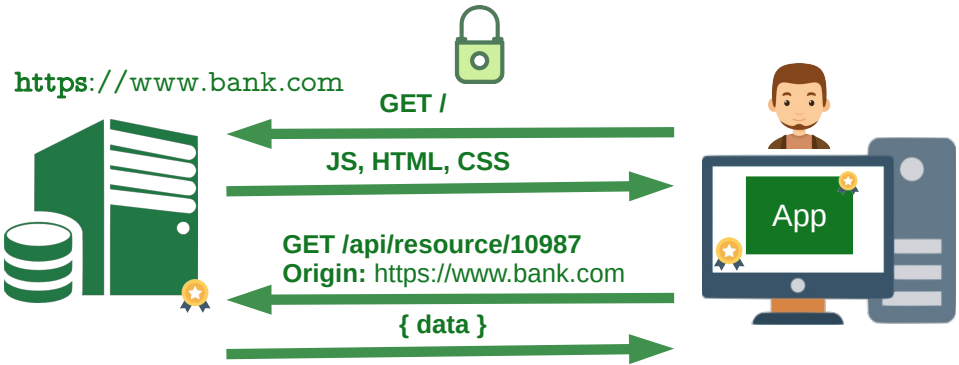
- The code of an application can ask for resources.
- The browser sends the origin of the application that is creating the request in an HTTP header called `Origin`.
- If the server recognizes itself as the "origin" then, it sends the requested resource.

If an honest user connects to a ”fake server” thinking that is an ”honest server”, the user will be in big trouble:



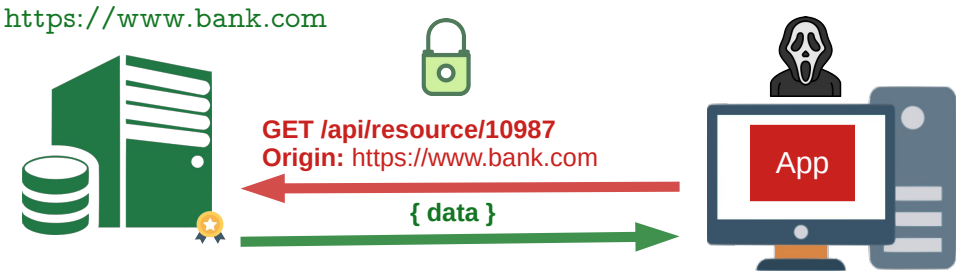
# Fake Servers ii

We prevent the fake server attack with a proper HTTPS connection:



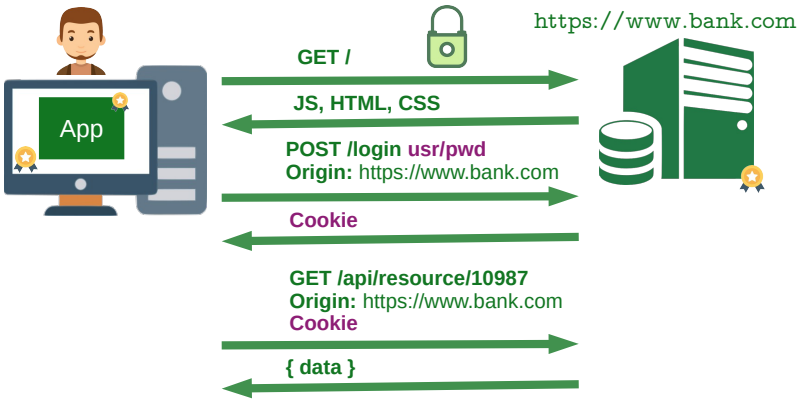
# Fake User

Without further protection, a fake user can get resources from an honest server:



# Login with a ”Session Cookie”

A valid application subscriber sends some extra information to prove that she is authorized, a typical way of doing this is to use a **cookie**:




Note. Each origin has its own (isolated) assets: cookies (which are automatically sent), local storage, etc.

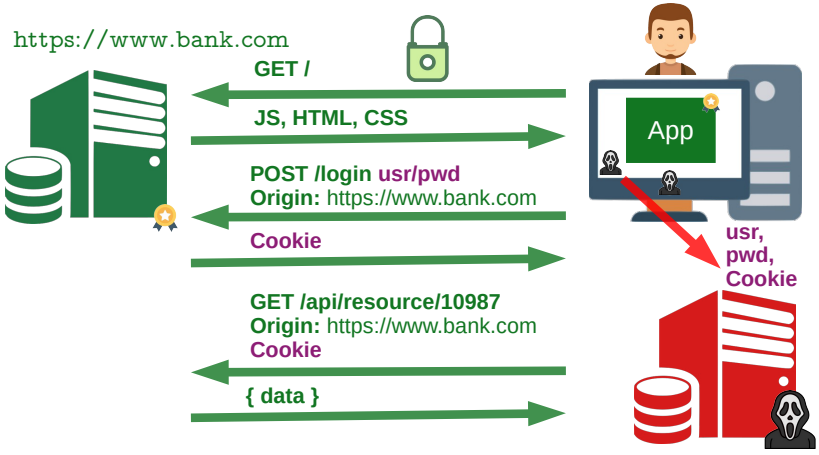
# Cookies (with ExpressJS)

```
const express = require('express');
const app = express();
const cookieParser = require('cookie-parser');
app.use(cookieParser());

app.get("/login", (request, response) => {
    response.cookie("loggedin", "true", { domain: "example.com"});
    response.send("Cookie sent!");
});

app.get("/readcookie", (request, response) => {
    if(request.cookies.loggedin == "true") {
        response.send("Logged");
    } else {
        response.send("Not logged");
    }
});

app.listen(8080, () => console.log("Express server running on port 8080") );
```

# Hacked OS

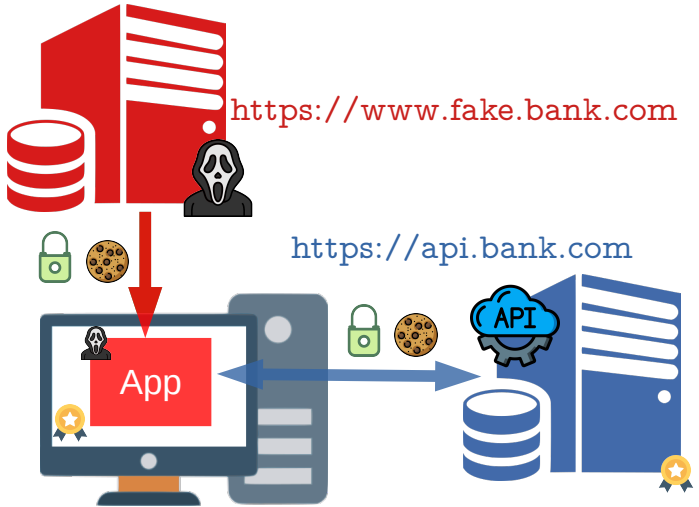
A hacked browser or operating system (OS) breaks everything.

E.g. the browser can send credentials (usr/pwd) and cookies to attacker's servers.



# Cross-origin Requests Security Issues

- Honest but sloppy users might be in trouble if a resource server that allows critical actions allows requests from other origins.
- If the user has a cookie for accessing the API server, undesirable Javascript code might access confidential resources.



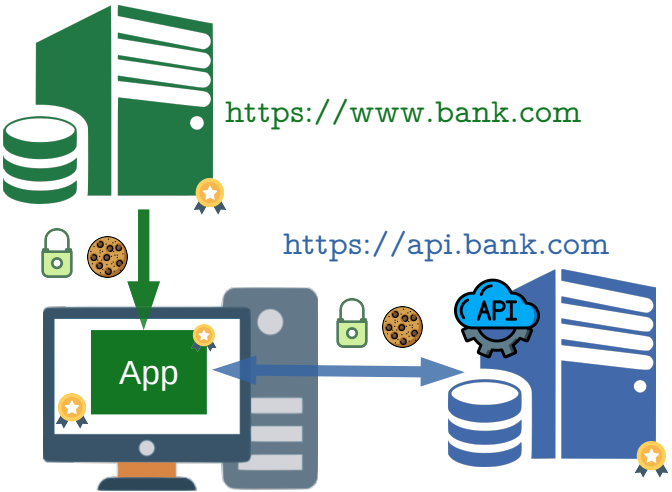
# SOP Architecture

SOP means that a server will only provide resources to its application code so it creates a **centralized architecture**:


Can potentially make our server a bottle neck and facilitate Denial of Service (DoS) attacks.

# Goal of CORS

**Cross-Origin Resource Sharing (CORS)** has the goal of decentralizing the interaction of a web application with different resource servers in a secure way.



# Basic CORS

- A web application executes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, or port) from its own.
- In its most basic form, CORS works by adding HTTP headers in cross-origin requests and responses:
  - The clients (browsers) inform about the origin that is creating the request in an HTTP header `Origin`.
  - The servers describe which origins are permitted to read information in an HTTP header called `Access-Control-Allow-Origin`.

# CORS Request

- Suppose web content at `https://www.bank.com` wishes to invoke content on domain `https://api.bank.com`.
- Then, the browser will send something like:

```
GET /public/resources/1 HTTP/1.1
Host: api.bank.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
Origin: https://www.bank.com
```

- The request header is `Origin`, which shows that the invocation is coming from `https://www.bank.com`.

# CORS Response

- In response, the server sends back an `Access-Control-Allow-Origin` header:

```
HTTP/1.1 200 OK
Date: Mon, 24 Feb 2020 00:23:53 GMT
Server: nignx
Access-Control-Allow-Origin: www.bank.com
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml
```

- In this case, the server responds with `Access-Control-Allow-Origin: www.bank.com`
- This means that the resource can be accessed only by web apps from the origin `www.bank.com`.
- If we receive `Access-Control-Allow-Origin: *` this allows the access from any origin.

# CORS Normal Case

- When dealing with sensitive resources, an origin is only allowed to get a resource over HTTPS.
- Additionally, if required by the resource's server, we can use `Cookies` or the HTTP `Authentication` header to send **authentication credentials** (usually called **tokens**).

# CORS Protection

Now we see how CORS protects the resources that should not be available to not authorized origins:

# CORS Errors

- CORS failures result in errors.
- But for security reasons:
  - Specifics about the error are not available to JavaScript.
  - All the code knows is that an error occurred.
  - The only way to determine what specifically went wrong is to look at the browser's console for details.

# CORS in Action with Express i

```
// Web 1: example.com:8080/login
const express = require('express');
const app = express();
const cookieParser = require('cookie-parser');
app.use(cookieParser());

app.get("/login", (request, response) => {
    response.cookie("loggedin", "true", { expires: new Date(Date.now() + 300000), domain: "example.com" });
    response.send("Cookie sent!");
});

app.get("/read-cookie", (request, response) => {
    if(request.cookies.loggedin == "true") { response.send("Logged"); }
    else { response.send("Not logged"); }
});

app.listen(8888, () => console.log("Express server running on port 8080") );
```

# CORS in Action with Express ii

```
// Web 2: www.test.com:8088/page.html
const express = require('express');
const app = express();
app.use(express.static(__dirname + '/public'));
app.listen(8088, () => console.log("Static server running on port 8088") );
```

```
<!-- page.html -->
<html>
  <body>
    <h1> Send AJAX </h1>
    <script>
      fetch('http://api.example.com:12345/resources', { credentials: 'include' })
       .then(response => response.text())
       .then(console.log);
    </script>
  </body>
</html>
```

# CORS in Action with Express iii

```
const express = require('express');
const app = express();
const cookieParser = require('cookie-parser');
app.use(cookieParser());
const cors = require('cors');
app.use(cors({
    origin: ["http://example.com:8080", "http://www.test.com:8088"],
    credentials: true  // send Access-Control-Allow-Credentials: true
  }));

app.get("/resources", (request, response) => {
    if(request.cookies.loggedin == "true") {
      response.send({tasks:["walk with the dog", "buy food"]});
    } else { response.send(null); }
});

app.listen(12345, () => console.log("api server started on port 12345"));
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS