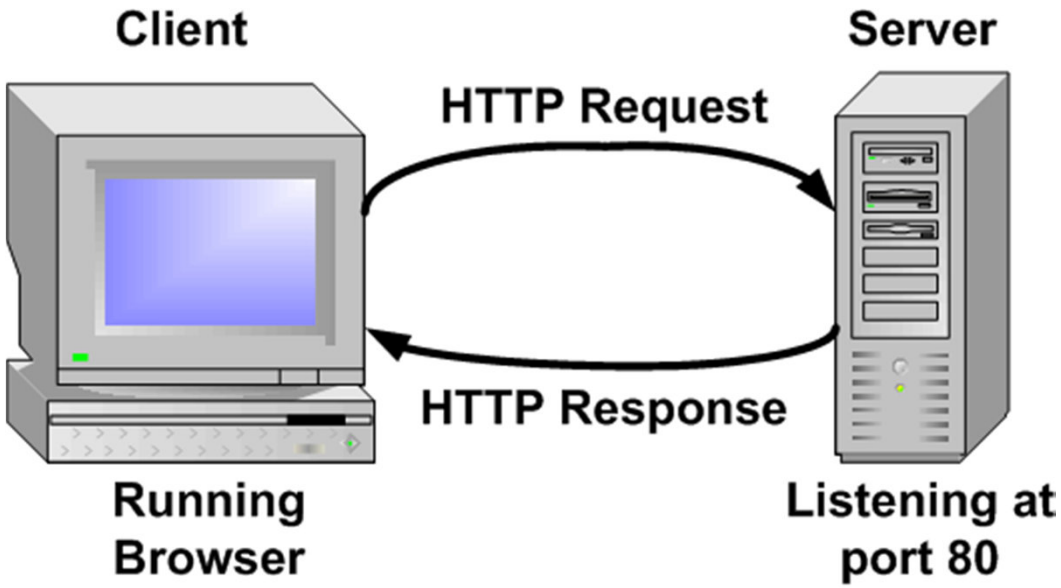
Seguridad de Recursos y Aplicaciones Web

HTTP Security

HTTP Fundamentals I



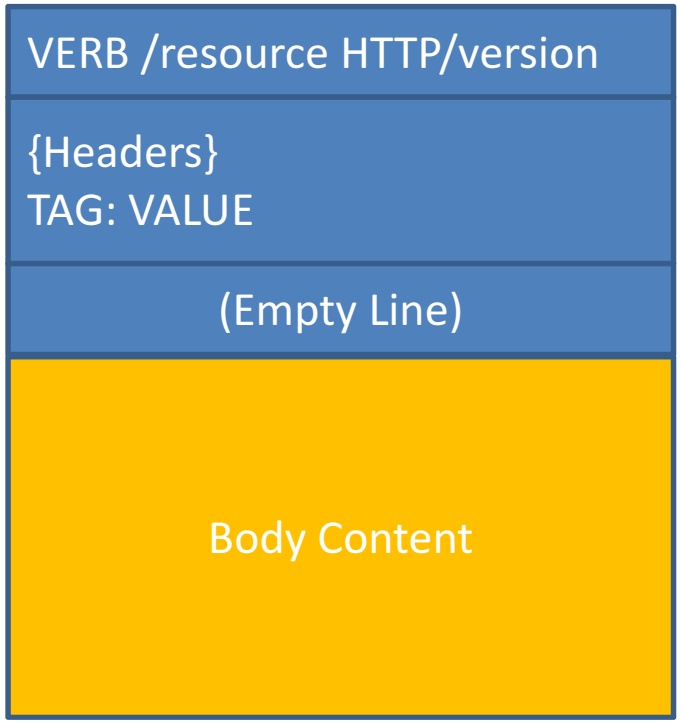
HTTP Request Anatomy I

- HTTP Request
  - GET: Retrieve Resource
  - POST :Upload Information
  - PUT: Upload File
  - DELETE: Delete Resource

GET Example

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Google Chrome 4.5
(Empty Line)
```

- HTTP Request - Shape



HTTP Request Anatomy II

- HTTP Request Useful Headers:

Cookie	an HTTP cookie previously sent by the server with Set-Cookie (below)	Cookie: $Version=1; Skin=new;
Content-Length	The length of the request body in octets(8-bit bytes)	Content-Length: 348
Content-Type	The MIME type of the body of the request (used with POST and PUT requests)	Content-Type: application/x-www-form-urlencoded
Host	The domain name of the server and the TCP port number	Host: en.wikipedia.org:80
Origin	Initiates a request for cross-origin resource sharing (asks server for an 'Access-Control-Allow-Origin' response header) .	Origin: http://www.example-social-network.com
User-Agent	The user agent string of the user agent	User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:12.0) Gecko/20100101 Firefox/12.0
Accept	Content-Types that are acceptable for the response	Accept: text/plain
Authorization	Authentication credentials for HTTP authentication	Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Cache-Control	Used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain	Cache-Control: no-cache

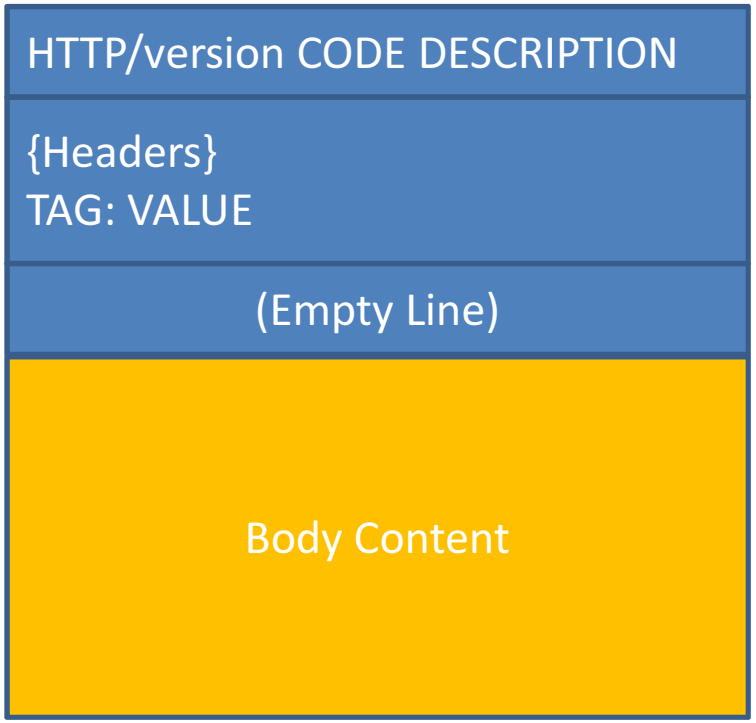
GET Resource URL Encoding

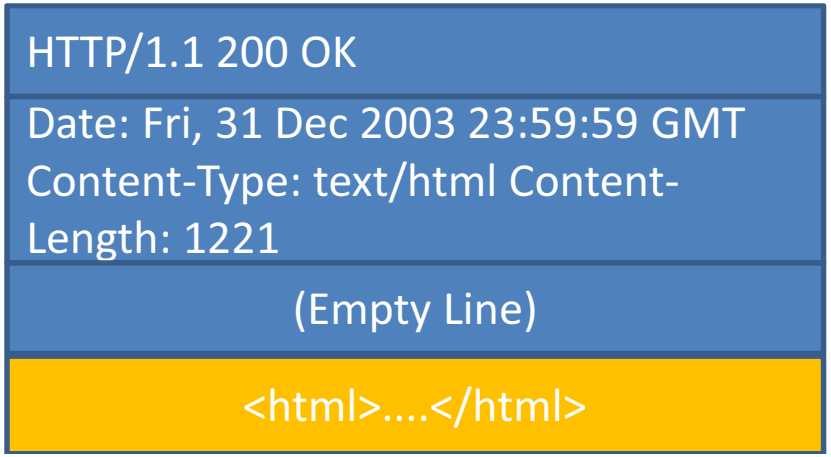
- “?” symbol used to separate params
  - /resource?params
- Each param encoded as
- “&” symbol used to separate each param

```
GET /index.html?user=jmalcaraz&size=100 HTTP/1.1
Host: www.example.com
User-Agent: Google Chrome 4.5
(Empty Line)
```

HTTP Response Anatomy I

- HTTP Response
  - 1xx Message
  - 2xxx Success
  - 3xx Redirection
  - 4xx Client-side Error
  - 5xx Server-side Error





HTTP Response Anatomy II

- HTTP Useful Headers:

Location	Used in redirection, or when a new resource has been created.	Location: http://www.w3.org/pub/WWW/People.html
Refresh	Used in redirection, or when a new resource has been created. This refresh redirects after 5 seconds.	Refresh: 5; url=http://www.w3.org/pub/WWW/People.html
Server	A name for the server	Server: Apache/2.4.1 (Unix)
Set-Cookie	An HTTP cookie	Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1
WWW-Authenticate	Indicates the authentication scheme that should be used to access the requested entity.	WWW-Authenticate: Basic
Access-Control-Allow-Origin	Specifying which web sites can participate in cross-origin resource sharing	Access-Control-Allow-Origin: *
Cache-Control	Tells all caching mechanisms from server to client whether they may cache this object. It is measured in seconds	Cache-Control: max-age=3600
Content-Length	The length of the response body	Content-Length: 348
Content-Type	The MIME type of this content	Content-Type: text/html; charset=utf-8
Date	The date and time that the message was sent	Date: Tue, 15 Nov 1994 08:12:31 GMT

Types of Authentication

- *Something you know* (eg. a password).
  - This is the most common kind of authentication used for humans.
  - Unfortunately, something that you know can become something you just forgot. And if you write it down, then other people might find it.
  - **Weaknesses**: Length, Character set & Randomness.
- *Something you have* (eg. a smart card).
  - This form of human authentication removes the problem of forgetting something you know, but some object now must be with you any time you want to be authenticated. And such an object might be stolen and then becomes something the attacker has.
- *Something you are* (eg. a fingerprint).
  - Base authentication on something intrinsic to the principal being authenticated. It's much harder to lose a fingerprint than a wallet. Unfortunately, biometric sensors are fairly expensive and (at present) not very accurate.
- Weak Authentication: 1 method
- Strong Authentication: 2 methods or more!

Authentication Flows -> Password Strength

Weak HTTP Authentication Methods

(Ordered according to the security level)

R1: HTTP Basic Authentication

Works: Client-side Browser, Client-side Services

1. Client asks GET /secret

2. Server: *401* WWW-Authenticate: Basic realm="Codegram"

3. Client sends credentials:
Authorization: "Basic" base64encode(username+":"+password)

4. Server returns *200 OK* with the document if the user and pass match in the DB.

General > HTTP Authentication

- Hint! Use Developer Tools -> Chrome!

Wrong! Why?

- Attacker can obtain **user's plain text password**
- The server **must know user's password in plain text**
- **Replay**: If intercepted, requests can be reproduced on the future
- **Reflection attack**: Attacker could fake server, get authorization, and fake client
- **Man-in-the-middle**: Attacker could fake identity and modify requests

R2: HTTP Form-based Authentication?

- 1º Option).
  - You implement your form
  - You implement your authentication methods
    - server-side app (servlet, cgi, similar)
    - DB
  - **Every resource** has to check if user is authenticated
  - **REALLY TEDIOUS TASK!**
- 2º Option)
  - You implement your form
  - Use Tomcat authentication & DB
  - Resources do not need to check anything!

Wrong! Why?

- Attacker can obtain **user's plain text password**
- The server **must know user's password in plain text**
- **Replay**: If intercepted, requests can be reproduced on the future
- **Reflection attack**: Attacker could fake server, get authorization, and fake client
- **Man-in-the-middle**: Attacker could fake identity and modify requests

R3: Use API token in the header

Works: Client-side Services

1. Client asks GET /secret

2. Server: *401* WWW-Authenticate: Basic realm="Codegram"

3. Client sends credentials:
X-AuthToken = "secret_long_string"

4. Server returns *200 OK* with the document in the token is a valid token in the DB.

Wrong! Why?

- ~~Attacker can obtain **user's plain text password** - still has access to the token~~
- ~~The server **must know user's password in plain text** - still the token...~~
- **Replay**: If intercepted, requests can be reproduced on the future
- **Reflection attack**: Attacker could fake server, get authorization, and fake client
- **Man-in-the-middle**: Attacker could fake identity and modify requests

R4: Use SSL

- **Hypothesis:** If the channel can be trusted, then the underlying authentication mechanism shouldn't be something to worry about.

OK … that's fine … but … is it really neccesary?

- SSL **Handshake is expensive**
- SSL communication has a **performance hit** (cyphering/decyphering)
- We just **lost HTTP shared cache** and probably client-side cache
- Useful **ONLY** when you have to exchange sensible information (non-related to authentication)
- Do you really need SSL for your personal blog?

R5: HTTP Digest Authentication

1. **Client ask GET /secret.html**
2. **Server returns:**

WWW-Authenticate: Digest
realm="testrealm@host.com",
nonce="dcd98b7102dd2f0e8b1", (random generated)
opaque="5ccc069c403ebaf9f01“ (random generated)

3. The client responses:

Authorization: Digest
realm="testrealm@host.com", (same as in request)
response="6629fae49393a",
opaque="5ccc069c403ebaf9f01“(same as in request)
The response is calculated as:

- **HA1** = MD5(username:realm:password) «jose:testrealm@host.com:Circle of Life»
- **HA2** = MD5(method:URI) «GET:/secret.html»
- response = *MD5(HA1:nonce:HA2)*

4. The server calculates “*response” using HA1 stored in the DB -> If it match return the resource*

5. Important -> Then, the server discard the once

Better .. Isn’t it?

- ~~Attacker can obtain **user's plain text password** - still has access to the token~~ -> MD5
- ~~The server **must know user's password in plain text** - still the token…~~ -> MD5
- ~~**Replay**: If intercepted, requests can be reproduced on the future~~ -> NONCE Mechanism
- ~~**Reflection attack**: Attacker could fake server, get authorization, and fake client~~ -> HA1 never sent
- **Man-in-the-middle**: Attacker could fake identity and modify requests

- Problem? -> The client cannot pipeline requests!

R6: HTTP Extended Digest Authentication I

1. **Client ask GET /secret.html**
2. **Server returns:**

WWW-Authenticate: Digest
realm="testrealm@host.com",
nonce="dcd98b7102dd2f0e8b1", (random generated)
opaque="5ccc069c403ebaf9f01“ (random generated)
qop="auth,auth-int",

3. The client responses:

Authorization: Digest
username="Mufasa",
realm="testrealm@host.com", (same as in request)
nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093", (same as in request)
qop=auth, (one selected from the options in request) (see next slide)
nc=00000001, (number of attempts)
cnonce="0a4f113b", (nonce in the client side)
response="6629fae49393a",
opaque="5ccc069c403ebaf9f01“

The response is calculated as:

- response = *MD5(HA1:nonce:cnonce:HA2)*

R6: HTTP Extended Digest Authentication II

***4. The server calculates** "response" using HA1 stored in the DB -> If it match return the resource*

5. Important -> Then, the server discard the server once

Note

- **Subsequent requests** can use the same *nonce* but must increment*nc* (nonce count)
- The server **must keep track** of the number of uses of a *client nonce*.
- With qop-value = auth-int it also includes a hash of the request body. -> **Overhead**! (but less than SSL)

Digest without qop="auth-init"

- ~~Attacker can obtain **user's plain text password** - still has access to the token~~ -> MD5
- ~~The server **must know user's password in plain text** - still the token...~~ -> MD5
- ~~**Replay**: If intercepted, requests can be reproduced on the future~~ -> NONCE Mechanism
- ~~**Reflection attack**: Attacker could fake server, get authorization, and fake client~~ -> HA1 never sent
- **Man-in-the-middle**: Attacker could fake identity and modify requests

- ~~Problem? -> The client cannot pipeline requests!~~

Digest with qop="auth-init". Awesome!

- ~~Attacker can obtain **user's plain text password** - still has access to the token~~ -> MD5
- ~~The server **must know user's password in plain text** - still the token...~~ -> MD5
- ~~**Replay**: If intercepted, requests can be reproduced on the future~~ -> NONCE Mechanism
- ~~**Reflection attack**: Attacker could fake server, get authorization, and fake client~~ -> HA1 never sent
- ~~**Man-in-the-middle**: Attacker could fake identity and modify requests~~

- ~~Problem? -> The client cannot pipeline requests!~~
- **Bonus**: Safe from **cryptoanalysis**

Insecure Comm > *Insecure Login*

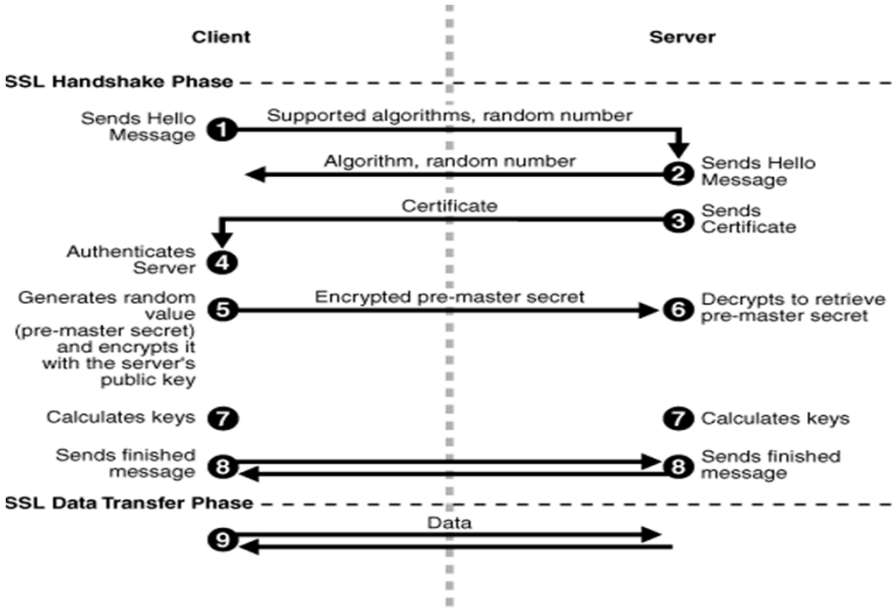
<u>EJERCICIO 1</u> (2 horas)

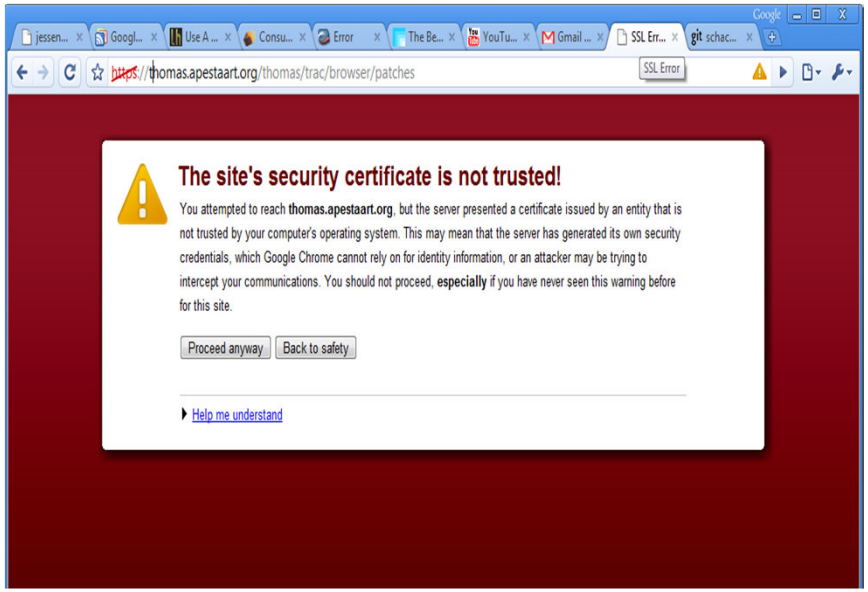
- O bien en LINUX, o bien con una maquina virtual en Windows
- Configurar Tomcat con SSL
- Arrancar WebGoat
- Arrancar wireshark in la maquina virtual
- Desde la maquina fisica, acceder al servidor webgoat
- Cuando sepan las respuestas
- **Hacer el ejercicio en nuestro servidor web!**
- **1 Semana!**

HTTP Strong Authentication Methods

R7: Now Default SSL

- Secure Channel -> Something you have (priv), Something you know (secret)
- SSL by default **only authenticates the server** using certificates
  - If the server has not a valid certificate -> Alert!





R8: Client-Side Certificate Authentication

But, client can be also authenticated!

1. **Client ask GET /secret.html over HTTPS**
2. **Server returns:**

WWW-Authenticate: Client-Cert

3. **Client** Validates Server certificate store
4. **User (client)** select a valid certificate
5. **Server** check client's cert
6. **SSL is established using such certificate**

R9: Electronic ID (eDNI)

- Modified version of Client-Side Certificate Authentication

1. **Client ask GET /secret.html over HTTPS**
2. **Server returns:**

WWW-Authenticate: Client-Cert

3. **Client** Validates Server certificate store
4. **User (client)** select a valid e-DNI certificate
5. Client check the presence of the eDNI in the smartcard and send the SSL challenge to the SmartCard
6. SmartCard ask for the passphrase (protocol PKCS#11)
7. Client insert passphrase
8. If valid, SmartCard sign using the private key stored in the card
9. **Server** check client's cert
6. **SSL is established using such certificate**

Authentication Flaws -> Forgot Password

References

- http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_1-1/ssl.html
- http://people.rit.edu/dsbics/546/lecture/https.html
- http://talks.codegram.com/http-authentication-methods#/intro
- http://www.cs.cornell.edu/courses/cs513/2005fa/nnlauthpeople.html
- Falta un buen libro
- Falta strong auth y weak auth