Internet Engineering

## TAGS USED:

**LSS = LONG STORY SHORT**

**OOL = OUT OF LESSON**

#### PSD = PREVIOUS SESSION DISCUSSION

**BYD = BY DEFAULT**

**DFW = DIFFERENCE WITH**

**ATST = AT THE SAME TIME**

**MQUESTION = MY QUESTION**

**OQUESTION = OTHER'S QUESTION**

**E.G. = FOR EXAMPLE**

# // ------ ------ ------ FIRST SESSION:

https://sbu-ce.github.io/IE-lecture/

parham.alvani@gmail.com

WWW

public information sharing

hypertext (linking documents to each other)

## Client-Server

### Server Processes:

* file retrieval
* computation/processing (DB lookup, transaction, ...)

refresh without refresh without refreshing the whole website.

kheyli jadidan kam pish miad ke ja be ja beshim beyne safahat (baste shodane safhe ghabli va baz shodane safhe jadid) hamanja ba taghire component ha taghire mohtavaye safhe ro mibinid

## engineering questions:

* how is it organized
* client-server interaction
* how representation is described
* how webpages interaction
* user interaction
* how to update portion of web pages
* encoded data transfer
* how server processes queries/requests
* how to develope complex web applications
* how gmail offline works
* how other applications use gmail

static web --> a webpage that has no reaction to the client.

internet protocols,

javascript with interactive webpages,

--> dynamic webpages in server side logic. with different appearances in multiple

meaning the web is not loaded predetermined.

also the webpage can interact with other websites.--> JSON.

#### Server side famous architectures.

## web services:

--> sabte ahval --> banke melli (national code validation)

gets the apis maybe --> google weather.

a web service which documented a character in a movie. (function invocation over web --> function call in web)

## course contents:

* introduction
* HTTP (internet protocols)
* HTML
* CSS
* Javascript
* JSON
* Golang
* Web Applications
* Virtualization
* React --> helps the dev of Frontend (TA)

## TA :

* Git
* HTTP Client
* docker (maybe?)

### Basic Concepts:

* Protocol, Port, Header, ...

### Programming

* Java/C

### Database

* SQL

## levels :

Lecture,

HomeWorks,

Industry,

Google, FaceBook

# // ------ ------ ------ SECOND SESSION

# PROTOCOL HTTP:

RFC == request for comment

best practice==standard --> no math standard

Internet community task force --> sharing and commenting on them (for each version of the protocol)

different versions of HTTPs.

transfer not only text but also all sorts of medias and texts and application like pdf and zips

CLIENT-SERVER model --> one person requests and one person responds.

is an application layer for users.

operates over **reliable communication** (there is no back up plan / plan b)

over TCP (reliable)

default server port : 80 (**id** of the connection between app and network protocol)

(several applications using network so each gets a specific port to connect)

#### ----------- these are discussed highly in the network course.

## HTTP IS STATELESS

http doesn't remember the clients just yields the files and documents. (protocol alone can't remember sessions)

(MQUESTION --> what informations are stored in session, which makes a person identifiable. ANSWERED : [1] : second session)

#### each URL is a URI but URIs are more general e.g. STID

## URL AND URI:

## uniform resource Identifier

uniform resource Locator (uniqely locate a file (locate and identify **ATST**)) (URL <= URI)

URN <= URI--> (uniform resource name)unique name. e.g. ISBN and STID (doesn't locate but identifies the entity uniquely)

URL not only identifies but also locates the files and documents.

## // ---- URL FORMAT:

#### <protocol || scheme >:// <user> : <pass> @ <host> : <port> / <path> ? <query> # <frag>

## SCHEME PART IN URL:

URL doesn't necessarily use http protocol and port is extracted if nothing, it extracts the default port from the scheme (HTTP --> 80)

different schemes : HTTP, HTTPS, FTP (File Transfer Protocol), File(browser **STD** dependant, it opens a local file), javascript

## PATH PART IN URL:

the address of the object file in the destination file system. (by default a file in the server.) (directory in dest)

note : web server root is /var/www/ (equivalent to windows C directory) <=> domain (no access is allowed to the root directory)

in todays it doesn't need to be file anymore.

## QUERY PART IN URL:

a mechanism to pass information from client to active pages. starts with ?, separated by '&'

(filling a form and sending info by query)

### example

#### https://httpbin.org/get?name=Parham&family=Alvani

#### path = get

#### query = "name = Parham"

## FRAG PART IN URL:

a fragment/section in the page which is supported by browsers only.

## DOMAIN NAME (/HOST) PART IN URL:

case insensitive (the rest of URL can be case insensitive or not dependent to the web server)

## URL IN ACTION:

1) user asks for retrieval

2) browser finds IP address of host (DNS lookup)

3) creates TCP and connects to the host and with the port.

4) browser sends http request

5) the server responds and the browser receives and shows (processes the response).

all of URL is not used in server. (host and port is for browser to lookup in the DNS server.)

TBREVIEWED LATER HERE AGAIN

## URL ENCODING:

URLs encoded with ascii codes. (old times)

### ASCII

---------------------------

unsafe chars (e.g. space)

reserved chars (e.g. '&')

non-ascii chars

characters which are not safe are sent by the example in the top (with %) ~ --> 126 --> %7E (1byte for each character if non-ascii -> utf8/16/32)

parsers in servers assumed that there is no space in URL.

domains are now getting support for UTF8 (other language)

older days space and newLine weren't distinguished.

so space is an unsafe character.

## URL ENDED

## TRANSACTIONS:

HTTP 1.1 & 1 --> one request only one response

methods --> requests

status code --> indicates the response

HTTP1 --> it's like batch systems (sequentially request and response.)

#### webpage includes:

main skeleton is HTML in the page

js / css and etc.

displaying web page: HTML page

rendering

linked resources (MQUESTION?)

get resources --> transactions

## how to do TRANSACTIONs:

http just shows the frame of transaction

( (HOW TOs) shape, size of data is TCP's job)

## NON-PERSISTENT:

assume TCP as a tunnel for simplicity (create tunnel for each transaction and close it after it's done

CON : alot of overhead ! resource overhead)

PRO : you can send simultaneous transactions (for example IDM sends a transaction for each 100MB of file)

note : overhead is for server but the client feels so good.

## PERSISTENT:

one tunnel only (one TCP) --> sequentially doing transactions

PRO : performance in the server.

CON : the performance in client is suffering. (serial manner)

HTTP 1.1 : speed up in client side --> pipelining. (sending all transactions at once instead of sending one by one)

#### pipelining --> IMPOSSIBLE

there is no identifier for each response to tell which request is it for. --> the only identifying manner was the sequence.

even parallel processing still would be useless because server should maintain the order of transactions. (one response can block all the following coming after.)

## HTTP2:

multiplexing --> identifying each response -> request.

#### PROS :

piece and partially sending responses

send all responses simultaneously

simple implementation

#### an example:

some don't have header because there is a caching discussion.

### ANSWERED MQUESTIONS:

[1]: a number stored in database which tells this is for person folani. (simplest method) CON: a retrieval from database for each request.

time Session (clears session after 10 mins)

example Session 1 month by mistake --> block session. (cookie)

JSON web token (more common service.)

JWT : all info is given to client, for avoiding editing the info by user, there is a signature under it. if you change info --> the signature cannot be copied. --> invalid session

# // ------ ------ ------ THIRD SESSION:

review from prev session

HTTP : client-server protocol / 1.0 / 1.1 / 2

## HTTP message structure (MAIN TOPIC OF TODAY)

relative address to base address (/var/www/)

tunnel needed to message (TCP) (->needs server address (postal code of server) :: IP)

-> linked to the port in server

non-persistant: meaning connection is not held and each time we produce a new connection (tunnel (TCP))

parallel connection is allowed

persistent: parallelism is not allowed (sequence is imp)

MQUESTION: the number for multiplexing is distinguishable but for who? the client can't understand between the numbers (no answers yet)

HTTP MESSAGES:

HTTP is text-based protocol (there is also binary-based protocols)

what is inside response and requests

composed of some lines

Req Line | Status Line (methods and status)

General Header

Req Header | Response Header

Entity Header

(these last three are extra explanation for our message, useful, extendable by time --> standard protocols, (today a text tomorrow an image))

Empty Line

Message/entity Body

headers are a bunch of names and values

e.g. request:

GET /index.html HTTP/1.1

HOST: www.aut.ac.ir

User-Agent: Mozilla/36.0

Accept-Language: en-us

Connection: keep-alive

general req structure:

Method<sp>Path<sp>Version<CLRF>

Header-Field:Header-Value<CLRF>

...

Header-Field:Header-Value<CLRF>

<CLRF>

Entity-Body

e.g. response

HTTP/1.1 200 OK

Date: Sun, 2 oct ... (today's date)

Server: Apache/2.2.2 (the server)

Last-Modified: Mon, ... (last mod date)

Connection: keep-alive

Content Length: 3000

data data data ...

response structure

Version<sp>Code<sp>Reason<CLRF>

Header-Field:Header-Value<CLRF>

...

Header-Field:Header-Value<CLRF>

<CLRF>

Entity-Body

code :: the number of response status

reason :: readable response status

Code:200 --> Reason:OK

Code:404 --> Reason:File not Found

MQUESTION: mesale vaghe E mishe az header ha bezanin az tu ye site?

OQUESTION (K. KHAVARI):

answer:

sol1: payload ->there is no limit all the data into one message,

sol2: but the network divides the message into some chunks and does it chunks

another solution was to do alot of connections and download piece by piece.

HTTP METHODS:

actions that client asks

GET (wanting to retrieve file)

Safe Operations:

get resource from server

send data to server

Unsafe:

delete resource

create/replace resource

these can be reversed.

Debugging methods:

get info

check what has got from client

list of operations applicable on a resource

GET: retrieve resource from server

HEAD: similar to get but resource is not retrieved but the headers of response (for debugging purposes)

POST: submit data (another sol is to send parameters from query parameters.)

DELETE: remove the resource

PUT: add message body

TRACE: echo back the request from client (for debugging purposes)

OPTIONS: list of supported methods by server on the resource

when implementing the server ourselves these methods can be redefined (e.g. GET 96243077 retireves the information about a student whose number is that. and does not necessarily retrieves a file)

http responses --> it is not common to change the responses' standard

Sucess

failure{

bad request,

server problem,

...

}

redirection to other resources

2xx:

200 : OK

201 : created

204 : no content

(all meaning success)

4xx: (fault from client)

400 : bad request,

401 : unathorized, (login solves it)

403 : forbidden, (Iran sanctions, admin files)

404 : not found,

405 : method not allowed

5xx: (fault from server)

500 : internal server error (server crashed)

501 : not implemented (beta)

503 : service unavailable

(maintainance mode, load of requests)

3xx:(caching and redirection)

301 : moved permanently, browser is cached (soft98.ir --> soft98.com)

307 : move temporarily, (you should check for it in another place) (browser does not cache it) (internet service buying page example --> google.com --> mokhaberat)

bitly shortens the URLs (not permanently move) (load on the bitly server goes high up, charge the server for every milion views of the site, each time the server moves the bitly site requests/redirects to the new site even if the client is using the old URL domain/path)

304 : not modified

telnet creates a connection: (is not a tool used in companies)

example:

telnet google.com 80

// ------ ------ ------ FOURTH SESSION:

review in HTTP headers

text based protocol benefits

most of headers are opt (for both responses and requests)

info about client/server (coming from server/client)

info about requested resource

info about response

security/authentication

there are few headers which are mandatory

these headers talk about the body of the message.

general headers (both req and response)

request headers

response headers

entity headers

entension headers (these are the headers not standard(future standards) or (specific for a company/project))

GENERAL HEADERS:

HEADER::Date header (time and date of created message)

HEADER::Connection header (both from client|server)

close : non-persistent connection

keep-alive : persistent connection

(server can close if he sees the need for non-persistent connection) --> opposing server to client request.

optional header:

HEADER::via : information

proxy nodes (non-client-server architecture)

REQUEST HEADERS:

(most important&mandatory)HEADER::Host: the name of server

was not required in telnet example. so why is it mandatory?

reason:

no normal people buys server for personal use. (server is the minimum server but still too big for the purpose)

there are alot of pages in the server.

--> the HOST header name will be looked up in the web server to determine which one to talk to.

not buying server --> buying hosts.

multiple websites on the server --> the it is mandatory.

note : browser always puts the host header independent of the need.

HEADER::referer: a redirection (names the site you came from) --> e.g. the shopping website. which website customers mostly come from?

client info headers:

HEADER::user-agent (client program (browser))

HEADER::accept the media supported (e.g. only html supported)

HEADER::accept-encoding (e.g. zip is not supported)

HEADER::accept-language (if there is french version in the website it'll give it to you)

these are not standard and are not mandatory so there are different behaviors from different websites toward this.

note : the HTTP request is really ez to manipulate/create --> some websites don't base everything on headers

HEADER::Range: range of bytes (parallel connection)

HEADER::Authorization

HEADER::Cookie

HEADER::If-Modified-Since (want the resource if modified/changed in this x time interval, used in browser caching)

RESPONSE HEADER:

HEADER::Server (server info)

HEADER::WWW-Athenticate (resource which login/adminstration is needed)

HEADER::Proxy-Authenticate

HEADER::Set-Cookie

HEADER::Location: the redirected response tells you where the new destination is.

HEADER::Last-Modified (caching use)

HEADER::Content-Range: (e.g. IDM, tells you which range of data requested it is.)

HEADER::Expires: (caching use, tells you when the content changes/is valid)

ENTITY HEADERS:

tells you about the body of the message.

HEADER::Content-Length (important header, check the length recieved and the expected length to be recieved is EQ)

HEADER::Content-Type (IMP) e.g. image/gif, text/xml, text/java

HEADER::Allow: in response to OPTIONS method, it gives you the list of allowed methods callable on this resource.

at the end of URL doesn't necessarily has the extension.

MQUESTION: akhe mage request ro nemidunim? response esh bara hamon requeste ma bude dige partial answer at the end of session 4 : [1]

EXTENSIION HEADERS:

not standardized yet/specific for a company/project

old times prefixed with X-

deprecated in 2012 : new suggestion :

<organization name><dash><header>

e.g.

Example-foo

VND.ExampleInc.foo

or put domain name :

com.example.foo

http://example.com/foo

note : no limit for characters usable in headers.

x- prefix is still used by many despite of deprecation

example:

restaurant puts the user-phoneNum in the header.

HTTP TOOLS:

cURL (CLI)

Postman (not free) -open source-> Postwoman

Hoppscotch

HTTPie(CLI)

usage : for testing websites and parameters. terminal login in golestan :D

postman --> tests the 1000 request to the website.

end of HTTP INTRODUCTION

EXTENSIONS TO HTTP:

STATELESS PROBLEM:

stateless protocol (HTTP) doesn't recognize you anymore(cannot personilize website) / cannot remember you

game website --> systems --> color change

refresh page --> no memory of client --> all settings reset.

sol 1 : X-Forwarded-For header, put identifier for browser, which this solution is not used because there are better solutions.

sol 2 : IP address of client --> bad thing about this sol (change in position and other issues.)

sol 3 : COOKIE

an information/id which server gives to browser.

browser gets an ID from server (not necessarily an ID, can be everything)

Set-Cookie header in response which requests the client to use this header for later identification. browser sends in later requests this cookie. this is standard, it is not mandatory though.

for privacy issues the websites ask for cookie permission

two types of cookies:

Session cookies : deleted by browser termination.

Persistent cookies : permanently available.

any structure in cookie is OK.

request example:

GET /cookies/set?name=parham&family=alvani HTTP/1.1

...

response :

Set-Cookie: name=parham; Path=/

Set-Cookie: family=alvani; Path=/

request example:

GET /cookies HTTP/1.1

Cookie : name=parham; family=alvani

response example:

headers:

body:

{

"cookies": {

"family" : "alvani",

"name" : "parham"

}

}

MQUESTION: the cookie is determined by the server? or in the url given by the request from the browser? (answered by slide in cookie limitations)

Limitations of cookie:

cannot be used to store large data

limitations for browser : store at least x cookies

at least x bytes for each cookie

at least x cookies for each host

note : min requirements from the browser and max stats from the server-side

Cookies are any text/string (not executable so it is safe in case of virus spreading)

the server does not read cookies, if browser sees the need for cookie it will send the appropriate cookie.

--> not mandatory from server.

(incognitive mode example)

Client can control cookies:

delete

save

view the values

create new cookie

(not every info is sent by cookie because client can change which is unsafe for critical information)

the only thing server can do with cookies:

controlling cookies by attributes. (e.g. path attribute)

expiration time,

domain,

path,

security,

etc

COOKIE ATTRIBUTES:

ATT::Expire(an absolute date) and ATT::Max-Age (a duration later should be deleted)

if any of the attributes browser sets => the cookie is permanent (browser closed the cookie remains until the expiration)

if a passed expiration date is sent the cookie is immediately deleted.

ATT::Secure (browser only sends by HTTPS protocol)

example:

login use case

server sends 73 cookie to the alvani user.

there is a mapping in the server side. 73:object

these types of cookie which are security important are session cookies.

attack: takes the request and sends the request for the server himsef (takes the identity, man in the middle attack)

there are a lot of solutions. in cookie and also in other solutions

another attack: sends a random number of cookie.

session cookies, are mostly used for logins, and are a very big number and an info from the user, to lower the likeliness.

some websites, say remember me.

HTTP connection some guy can come and read the contents of messages. HTTPS layers an encryption not to let anybody read the cookie.

ATT::HttpOnly

javascript can access cookies and browser can limit the things js can access. control of browser.

ATT::SameSite

only first-party websites. and the cookie be sent on certain conditions. (e.g. example.com using image from github, github is a 3rd party website)

Lax: top-level navigation, GET reqs to 3rd parties. (this is default)

Strict: only first-party

None: no restrictions.

possible attack, using cookie to do something in background from third prties (e.g. remove course).

fetches are normally nothing executable and normally images and browsers are really sensitive in the first place.

review:

COOKIE has some attributes,

the use cases

can contain sensitive information, e.g. the login usage.

CSRF ATTACK

ANSWERED MQUESTIONS:

[1]: if the content-type is not specified the rendering in browser might come with issue and sometimes browsers understand the extension by the 3 first characters in the file (e.g. jpeg, png, etc.)

// ------ ------ ------ FIFTH SESSION:

ATT::HttpOnly: prevents from javascript files access cookies

ATT::SameSite: main purpose: ctrl cookie for which domains will it be send/not

MODE.Lax: direct/redirect web access the cookies will be sent. but the cookies will not be sent if the page has some resource to be requested from other websites.

MODE.Strict: only first-party

MODE.None: for tracking purposes the cookies will be sent to third-party websites as well. (Secure attribute is a new restriction force)

CSFR ATTACK: login in golestan -> cookies set. a link from other sites (malicious site). a webpage that has a click to make a method to the previous site (golestan) --> samesite.strict

cookie validation --> clients aren't always good. (up to now we assumed third-party thieves and bad guys)

cookie be sent for which domain/path (browser sets cookie for domain but not the subdomains by default)

ATT::Domain: setting the domain explicitly the sub-domains will also be set from the browser.

ATT::Path: (sets cookie for path and all sub-paths)

validity check for domain and path.

old standards : <dot> .example.com

new standard : example.com (the dot behind is eliminated)

constraint 1 : hello.com request cannot get a cookie for example.com (the domains for request and cookie attribute should match)

constraint 2 : sub-domain cookie set is not allowed -->

e.g. for example.com not the sub-domain : hello.example.com

constraint 3 : you can set domain for higher domains but not top level domain (.com)

constraint 4: you can set cookie for path including the requested path.

using library for editting servers.

review of cookie

info about client storing to remember

login uses.

risk for storing info in cookie to reduce server processing load (bc client could change the cookie)

PROXY AND CACHING

http not always client-server architecture --> here!

proxy : some intermediate entity between client and server (as a client for server and as a server for client) (IP changing using proxy to turnaround filterings)

REVERSE PROXY: -> haproxy and NGINX

nothing that you usually encounter day to day life when you are a client. (maybe just when you are an employee in a company)

(the abstraction layer on server. the server gets to be anonymous and the client doesn't know which server it is requesting to.)

pros:

doing some similar works between the multiple servers. reduces redundancy.

e.g.

load balance,

(cloudflair:) drop the request rate from the proxy. or stop the users from iran.

compression to optimize speed and latency

caching in reverse proxy is also available.

note : reverse proxy is in the control of the server-side.

FORWARD PROXY:

(abstraction layer on client. client gets to be anonymous.)

// AKA normal proxy

authentication for client or some certain servers.

FP:AUTHENTICATION: (td better methods)

limit access to internet OR access web server

--> only when login. proxy checks user pass and authenticates the client.

FP:ACCOUNTING: (td better methods)

log activities (they could check your activities in your proxy in internet access in uni)

FP:SECURITY: (td better methods)

firewalls checking for attacks

FP:FILTERING: (td better methods)

limit access

FP:ANONYMIZER:

request from another server for us. (iran sanctions 403 error turnaround)

FP:CACHING

(td better methods) -> today there are better methods to do these. because the http proxy should be set in your pc so it has some issues and problems.

CACHING :

keeping a copy of the resource and use it instead of request --> browser's local cache

pros:

server load dec

latency dec

net bottleneck dec

redundant data dec

no cache

local/private cache

shared cache (in company server/university/computer)

how to be sure that the source available in cache is still valid/unchanged/unmodified?

CACHING ALGO:

simple form: request for object in local cache -> there's no such object -> request to server

item available fresh:

Cache-Control:

general header

case1-> max-age = <seconds> (from server)

case2-> no-store (no caching is allowed)

case3-> no-cache (request for validation before caching is required)

REQUEST:

if (in case1&expired) or in case3:

using conditional request:

If-Modified-Since: false:

the object is not retrieved and if true the new object is retrieved -> in anyways it is faster.

If-None-Match: using tag instead of time-modified --> (e.g. content is not changed just the file renamed or was moved to antoher directory) --> so the tag uses a version name of the object

OOL : (out of lesson) : (ETAG for wikipedia )

RESPONSE:

304 unmodified the cache is fine.

-> set new expire time

200 OK

server retrieves the new version of object and the caching process is done again.

caching is more meaningful media files

# // ------ ------ ------ SIXTH SESSION:

proxy discussion and caching and reverse proxy (PSD = previous session discussion)

how to control access to our server and HTTP role

stateless protocol and cookies need for authentication

authentication --> the need bc of stateless http

## authentication and authotization difference

authentication --> the right for connection to the server

authorization --> the accessible/permitted items for the user.

authorization is after authentication

## AUTHENTICATION:

(the need to limit access)

BASIC : Base64 encoded user:pass (user not itself can have colon)

DIGEST : plain user name + digest of pass

#### steps:

1. request for resource.
2. resource access fail --> 401
3. client side ask user-pass from user.
4. client sends user and pass.
5. server authenticates and allows.

issue : this way every request requires these steps -> reason : stateless protocol.

base64 -> binary-to-text encoding schemes (e.g. farsi) represent binary data in an ASCII string

non-printable characters in ascii is problematic

#### solution:

6-bit,6-bit seperating instead of 8-bit seperation (each 6-bit is mapped to a printable ascii code.) (e.g. 0000H is null character so it is not printable --> instead 000,000b is mapped to 'A' and etc.)

independant of the content of the binary represented message. ALSO USED IN BASIC AUTHENTICATION but it's a general thing

## AUTHORIZATION HEADER:

Basic <something in base64 representation>

#### problem :

bi-directional transformation --> possible attack finding the protocol and can use it.

(wire shark in network course capturing message)

#### solution:

hashing -> one way transformation (the pass is not attainable)

#### problem2 (reply attack!)

a possible attack : hash is usable by the attacker too

#### solution for p2:

Nonce --> a random number per request. to use in the hashing for the digest authentication.

### solution2:

### Secure http (HTTPS) --> not even seeing the protocol context.

HTTPS (encoding the whole message.)

digest authentication example (gettin old):

### error 401 server respose:

WWW-Authenticate:

Digest realm="me@kennethreitz.com", nonce="9e97866f5

note: server error 401 in Digest is really important which wasn't in Basic Authentication. (sending request ahead without the 401 error)

but in Digest 401 sends you the nonce parameter which is necessary.

### this is clients' response/request.

realm, digestURI and nonce comes from server (parameters)

user pass is from client.

HA1 = MD5(username:realm:password) // hashes.

HA2 = MD5(method:digestURI)

response = MD5(HA1:nonce:HA2)

// hash(pass, nonce)

the server expects the nonce to be attainable and check with the hash sent by user. (another request has another nonce.)

even if the hacker obtains the nonce it'll be useless because in order to send the hash in request he must know the client's user pass.

MQUESTION: nonce be ezaye har request miad ye addade random mide?(answered [1])

digest method for authentication is expired

another example --> the server gives us a bunch more parameters.

also using different methods to make the unhashing process harder. (even though it's so complicated it is still possible to unhash it. <pro>-> the more time it takes the hash expires.)

## IN THE WILD :

plain text password sending using HTTPS (the load for pc is alot but nowadays and free certificates it is so easy to send HTTPS now)

second problem solution:

using cookies for keeping the login and we don't need to do the authentication everytime.

header::secure : true

http has a method for authorization but it is not imp and sending the user pass plain text is more common nowadays.

## BEARER AUTHENTICATION:

#### new idea:

### set cookie for client

and take user pass plain text

--> http authentication useless

authorization header (which is useless) => goal is to give client access through cookie.

no more user pass is needed

giving the token authentication (the bearer of the cookie is authenticated.)

but the authorization header instead of cookie is better.

## but problems of cookie:

1. user can change the cookie
2. the cookie has limited capacity

using the authorization header we don't more worry about the browser handling the cookie. this way the capacity limitation is gone. but the client can still change this.

(we can keep it in the app in client side)

JWT Standard idea JSON Web Tokens to generate the tokens in the authorization header.

QUESTION:(solution is JWT) how to make sure that the client is not making changes to the token.

ONE SOLUTION: is to give the client a number and when given back, the number's invalid if the client’s made changes.

and the info about the client is kept in the server side app.

#### THE PROBLEM WITH THIS SOL:

simple apps which we don't need to keep the client's information (e.g. the weather news website)

JWT --> give needed information to client and the client give back in every request. (one time saved in DB (a cheap rather DB) JWT has no load on the DB to check if the given back info is still valid.) --> signature. (an encrypted signature, which has a low porbabilty for client to create valid signature. so if modified the modified signature i likely to be invalid)

LSS : validate signature only --> not checking by server database

JWT reduces load for databases so it is a very common methods nowadays.

#### problem:

bad users --> we have no choice but to declare the token to be destructive.

## HTTPS:

HTTP problem data is available in HTTP (only the pass is encoded in digest form only)

filtering --> query in google was readable.

#### China FIREWALL

#### Iran firewall

there is no specific method to read the content of the message

## SSL handshake process

a public key from the server

we create an encrypted message with the key

we have a complete encrypted message because only the server with key can read the message.

MQUESTION : didan yaani chi? chejori dar http mituneste bebine? ke alan https aslan ejaze nemide aslan chizi dide beshe va connection secure mide?

#### FILTERING TOPICS

#### SSL

[1] : client also sends the nonce to the server. the protection wouldn't reduce.