Web Application Architecture

Web application are based on client-server paradigm. Clients run on web browsers. Server, instead, is often based on a complex architecture. Communication between them is made over Internet using HTTP(S) protocol. They are based on the classic three-layers architecture: Presentation, Business Logic and Data Access.

On server side, the overall architecture can be really complex. In modern applications a Database is used for data management, mostly relational. For the business logic layer, complex and various software components can be used.

On client side, instead, users exploit web browsers in order to make requests to Web sites. The Presentation Layer is implemented through browsers.

Basic ingredients for Web Applications

Web pages are exchanged within HTML pages. Modern version of HTML is HTML 5, which supports most recent web applications. It includes several technologies like CSS, JavaScript and DOM. It supports also client-side scripting in order to make interactive pages, moving part of business logic on clients exploiting their computational power. Also source of threats of course.

All modern browsers support JavaScript with built-in interpreters. JavaScript relies on a run-time environment to provide objects and methods exploited by scripts in order to interact with the environment and also the possibility to include these scripts in HTML pages. Scripts run in a sandbox where they can only perform Web-related actions. Same Origin Policy (SOP) constraints the usage of those script: script on a web site must not be able to access information about another web site.

Document Object Model (DOM) is a technology that represents HTML documents as a tree of objects. The tree's structure follows the one of the HTML document and through this objects can be programmatically accessed and manipulated in an easy way.

JavaScript, indeed, relies on the DOM in order to dynamically interact with these objects, to change, add, remove them.

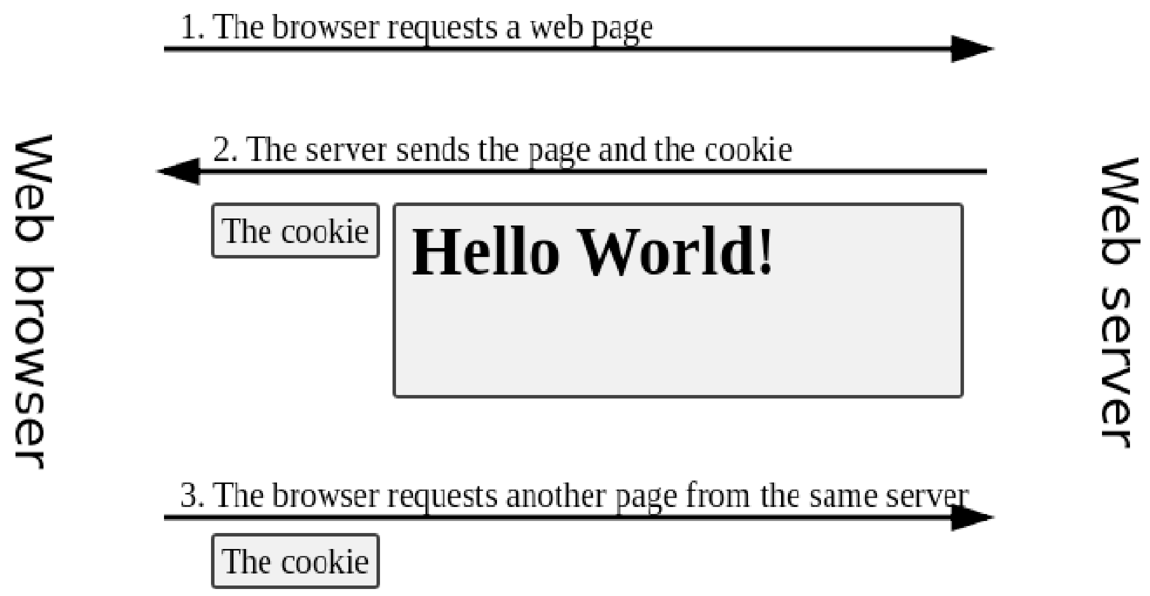
A new version of SOP is allowed and supported by web browser: Relaxing SOP. Some web sites are based on multiple subdomains. This configuration would violate the SOP. So the policy is relaxed in order to allow scripts working on multiple subdomains of the same web site.

As we said before, browsers run into containers (or sandboxes). Each tab of chrome has its own container. The idea is that different interaction of the user must be kept isolated in order to deny any kind of cross info stealing by an attacker. A vulnerable web site should not concern also other open web sites.

HTTPS is the secure version of HTTP obtained by making HTTP using a TLS connection between two hosts. All browsers support it. TLS is a set of protocols that guarantees confidentiality, data integrity, server authentication and provides some security features avoiding manipulation of data while they are in transit by a third evil party.

Security at client side

Since HTTP protocol is stateless, cookies can implement states. Browsers store cookies in dedicated storage areas.


Web browser
1. The browser requests a web page
2. The server sends the page and the cookie
The cookie
Hello World!
3. The browser requests another page from the same server
The cookie
Web server

Only two fields are mandatory: name and value. Usually more than these two values are present like expiration date and time, domain which they refer to, accessible from JavaScript or not and more.

There are different type of cookies:

- First party: a cookie received directly from the visited web site.
- Third party: a cookie received from third parties (web server not directly visited).
- Session cookies: automatically deleted when browsers exit.
- Permanent cookies: have a expiration date and time.

Browsers provide users with some support to define a personal policy on cookies:

- Accept or reject cookies.
- Accept or reject third party cookies.
- Keep or do not keep permanent cookies.
- Browse and inspect stored cookies and possibly delete a selection of them.

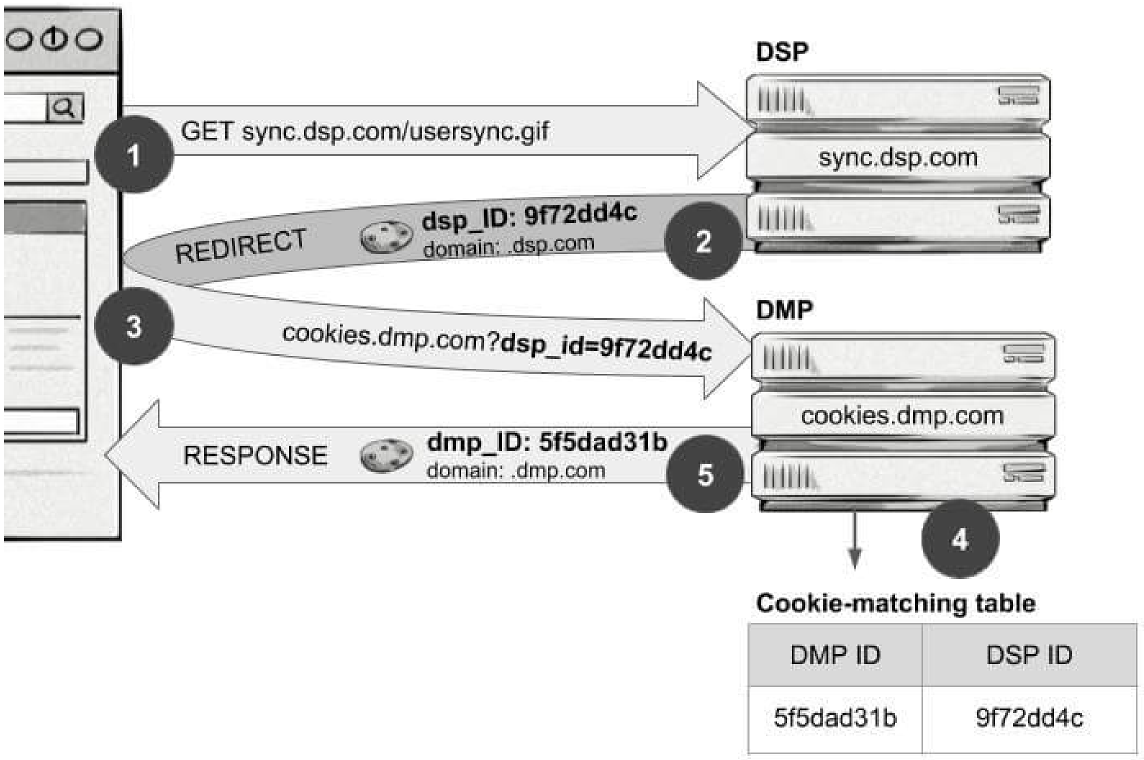
Third party cookies enable user tracking. There are many different ways how to implement it. Web tracking's goal is to determine web users interests or modeling users behavior. They want to personalized as much as possible shown advertising. This approach may violate user privacy and be used for unethical purposes.

An usage example is:

- An user connects to site A to get index.html.
- index.html asks to download image img.png from site B≠A.
- Now the user's browser connects to B to get the img.png.
- B knows the refer if no https is used and sends back one or more cookies with sensitive information.

A referrer is the URL of the previous item which led to the current request (for an image is generally the page on which it is to be displayed).

Another example is cookies syncing. As mentioned previously, cookies are domain specific, which means those created by one third-party tracker cannot be read by another third-party tracker. Therefore, in order to accurately target an audience, advertisers need to incorporate user data from various domains and sources, which happens as part of data-buying agreements and partnerships between different companies. Cookie syncing works when two different advertising systems (aka platforms) map each other's unique IDs and subsequently share information that they have both gathered about the same user.



DMP ID	DSP ID
5f5dad31b	9f72dd4c

This image explains the process: a web site asks the user's browser to download an image from DSP. A cookie (user ID) is created if the user is the first time that accesses that tracking service. Then the DSP asks for the user's browser to contact, via REDIRECT, another tracking service specifying the user ID chosen for the user. The DMP now can generate a new user ID for the current user creating a cookie and sending it back to the user. DMP can also update its own lookup table where it saves the user ID of different tracking systems. Cookie syncing is completed.

Internet is designed as a public network. Routing information is public too, i.e. IP packet headers identify source and destination end points easily being captured by an observer. Encryption does not hide identities, only content of connections. The Anonymity concept is related to the fact that a person must not be identifiable within a set of subjects. The main idea is not to reach anonymity alone, but all the users should appear the same. Unlikability, instead, means that an action is no more linkable to the identity of the user that generated it. Unobservability is really hard to achieve. Indeed, it requires that an adversary cannot even tell whether someone is using a particular system and/or protocol.

Possible attacks on Anonymity are:

- Passive traffic analysis, inferring from network traffic who is talking to whom.
- Active traffic analysis, Injecting packets or put a timing signature on packet flow.
- Compromise of network nodes, violating some internal routers, never trust network devices not under our control.

Anonymity proposal = Tor (dedicated slides).

Authentication is the process of verification that an individual, entity or web site is whom it claims to be. Commonly performed by submitting a user name or ID and one or more items of private information that only a given user should know. HTTP authentication is a method for a browser to provide a username and password when making a request. There are two mechanisms based on challenge-response paradigm:

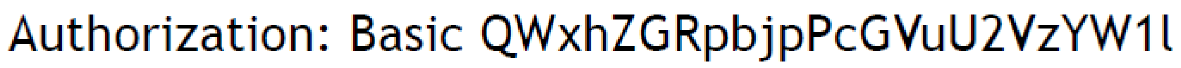
- basic authentication
- digest authentication

The HTTP protocol is a request-response protocol. It is the most used for accessing Web pages. A client requests a resource to the server, the server provides a response to it. In a HTTP session there are sequence of request-response transactions. Of course, requests and responses are formatted following a standard in order to be easily interpreted by browsers and servers. An HTTP message is composed as follows:

- First line specifies the request type or the status of a resource.
- Then we have an arbitrary amount of headers in the form "name: [values]".
- Then, after an empty line, there is the message body (an html page for example).

The request type can be: GET, HEAD, POST, DELETE, PUT, PATCH and etc. For example, GET requests a representation of the specified resource or HEAD requests only about metadata of a GET request. A status code defines the status of a resource. There are 5 set of status code: 1xx Informational, 2xx Successful, 3xx Redirection, 4xx Client Error, 5xx Server Error.

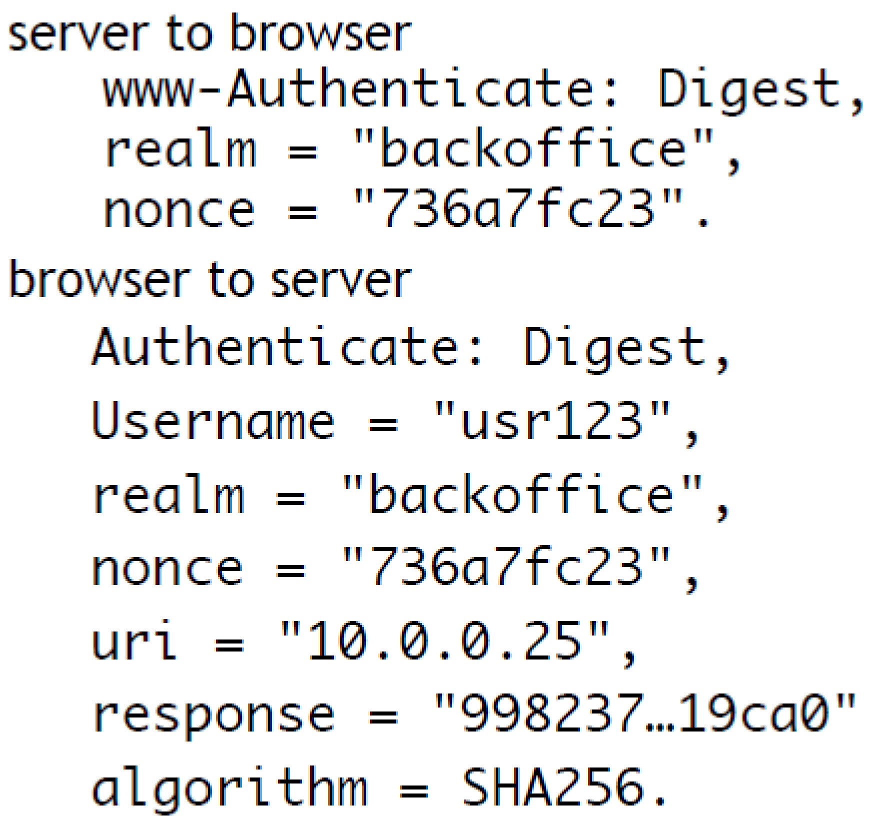
Basic Authentication transmits username and password only encoded in Base64, i.e. encryption requires TLS (HTTPS). These information are put in the Authorization header, specifying the type (Basic):



```
Authorization: Basic QWxhZGRpbjpPcGVuU2VzYW1l
```

The encoded string represents the clear text string in the format username:password.

Digest Authentication, instead, uses cryptographically secure hash function to represent username and password. The server challenges the browser sending to it a nonce. A valid response for the challenge should contains username, password, nonce, http method and URI.



```
server to browser
    www-Authenticate: Digest,
    realm = "backoffice",
    nonce = "736a7fc23".
browser to server
   Authenticate: Digest,
   Username = "usr123",
   realm = "backoffice",
   nonce = "736a7fc23",
   uri = "10.0.0.25",
   response = "998237…19ca0"
   algorithm = SHA256.
```

```
HA1 = hash(username:realm:password)
HA2 = hash(method:digestURI)
response = hash(HA1:nonce:HA2)
```

A popular way used along with HTTP to provide authentication capability is based on HTML Forms. When a browser requests a protected page, the server returns a page containing an HTML form which prompts the user for username and password, along with a submitting button. User fills the form and then presses button. The browser crafts the correct HTTP message, as before, and send it to server. Now, server is able to perform whatever checks on the retrieved data.

In HTML there is the "form" tag to create a form object in a web page. It can contain whatever input object, from text inputs to checkboxes. No encryption is guaranteed by the form object, there is needed HTTPS. In order to use a form, we need to use GET or POST HTTP methods, POST is preferred because GET shows in the URL the content of the form.

As we said previously, HTTP is stateless, i.e. if I authenticate myself only relying on HTTP, the server is not able to notice the authentication in future requests. Cookies allow server to recognize an already logon user. These cookies, called session cookies, must be protected carefully because they bypass authentication! If an attacker sends to a server a cookie of another logon user, the server will show him all the protected pages related to the user associated with the received cookie. The theft of these cookies is known as session hijacking or cookie hijacking.

Security at server side

A web server is typically an host where an HTTP daemon is running accepting connections on port 80 and 443 (possibly others). It is frequently the main goal for an attacker, i.e. the most profit one.

Unvalidated input

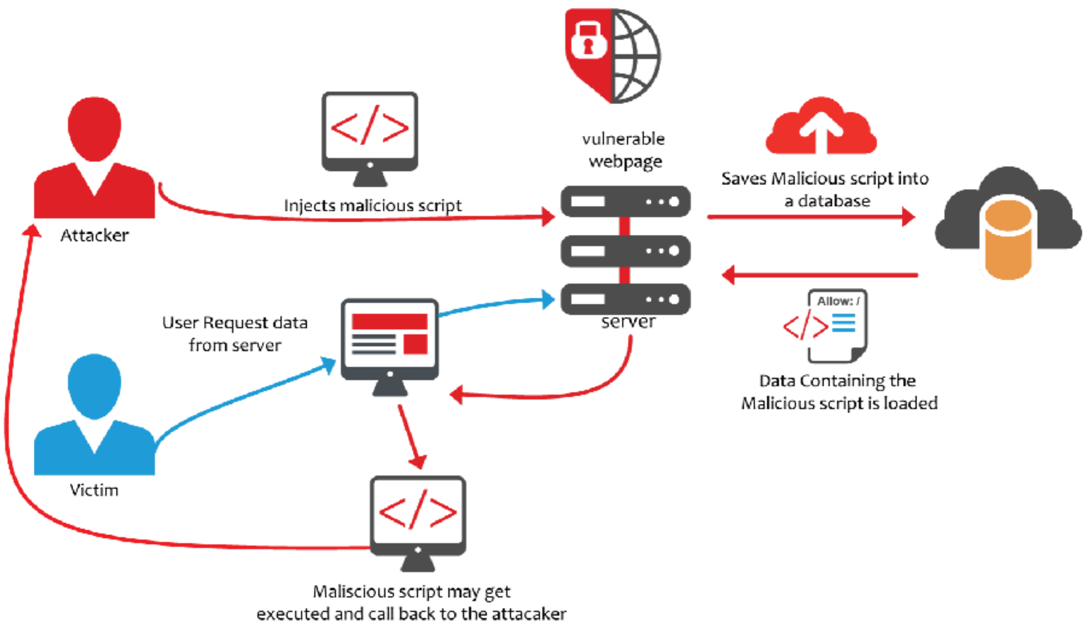
It is a general problem that affects all softwares, also web applications. In web applications, attackers try to tamper any part of an http request, starting from body, URL, headers, cookies etc. Their goal is to bypass site's security. Frequent origin of this problem is that the input is only scripted validated on client side, easy to bypass.

Validate an input means to guarantee that on server side arrive only valid and feasible inputs, rejecting non feasible and legal ones. For web application, it is not enough validate the input through JavaScript. The validation and the sanitization must be performed also on server side! This increases load on server side of course. Validation requires to check if the input passed by the user respects some formatting rules. Sanitization, instead, requires the server to prevent some user's input to crash the application/service where that input will be used, by escaping some specific and application related characters.

Buffer Overflow

Buffer overflow is a possible attack that an attacker can perform on applications that don't check about the size of the input that users passed. This technique allows attackers to manipulate the execution flow of the server side in order to access directly its memory and to execute some arbitrary code (in the worst scenario). OpenSSL was affected by HeartBleed bug for a long time: in order to keep a session alive, server required an echo reply by the client, but the length of the replied string could be increased in order to write more bytes than the actual allocated by the server.

Cross Site Scripting


Attacker
Injects malicious script
vulnerable webpage
server
Saves Malicious script into a database
Allow: /
Data Containing the Malicious script is loaded
User Request data from server
Victim
Maliscious script may get executed and call back to the attacaker

Cross Site Scripting is spreading a lot during the last years, due to an increasing usage of scripting in web pages. A typical scenario is shown in the above picture: an attacker injects some malicious script in a vulnerable web page, then this script can be stored permanently in the server, when a legit user asks for some service or resource to the server, this latter will deliver the correct resource along with the malicious script that the attacker previously injected, now on the script runs on user's browser and in most cases contacts the attacker machine sending some private information.

E.g.

```
<A HREF = "http://example.com/comment.cgi?mycomment = <SCRIPT>malicious
code</SCRIPT>">Click here</A>
```

The content between script tag will be interpreted as script and then executed. Pay attention of sanitization of the input!

The malicious code is up to the attacker, it could contain a src parameter that download an image from a malicious web site (this approach breaks SOP), it could steal session cookies, and so on so forth.

There are three main categories of XSS:

- Stored: the malicious script is stored in the vulnerable server, users are attacked when they request the resource that embeds the injected script, this is the most dangerous.
- Reflected: the script is NOT stored in the server, in this case the attacker uses other ways to deliver the script like email with malicious links (the link points to a valid web site, but it contains a script that, when the request is served and its result displayed, the malicious script will be executed).
- DOM-based: it is an XSS attack wherein the attack payload is executed as a result of modifying the DOM “environment” in the victim’s browser used by the original client side script, so that the client side code runs in an “unexpected” manner. The HTTP response of the server is left unchanged! The only side effects of this attack are on the user's browser which modifies the DOM environment wrt the injected payload.

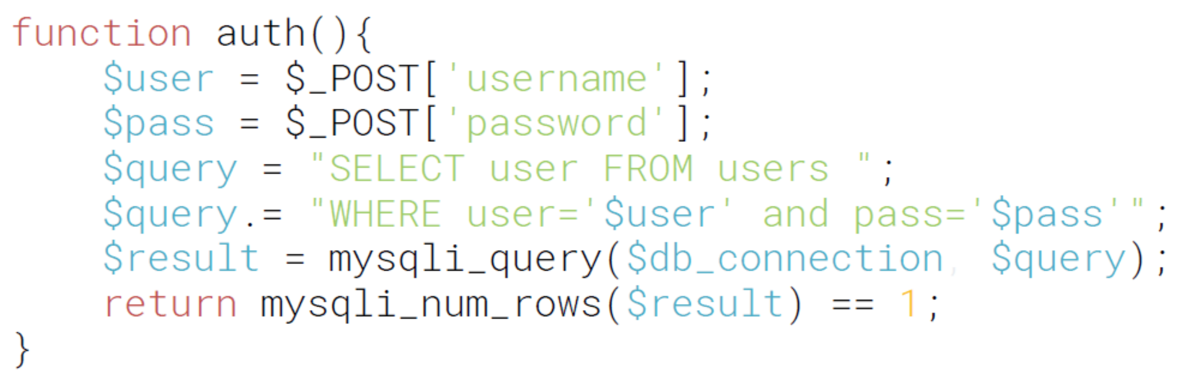
The real difference between the first two and the third one is that: stored and reflected XSS modifies the HTTP response of the server, stored because it is injected in an object retrieved from a persistent memory, reflected because it is injected in a not stored object of the response page (e.g. injected in the username and when the page shows "welcome *username*" the username will be replaced with the script), instead, the DOM-based is a script that tampers the HTTP response of the server ONLY on client side.

SQL Injection

If something is taken from user input and then stored or executed on server side, well, this is a possible injection flaw. In this case we are focusing on SQL Injection that allows an attacker to dump or tamper the database that a web application used.

The scenario is pretty simple: an attacker send some non sanitized input to the server that doesn't analyze it, then this input is used some how to execute a dynamic query in order to retrieve some database content. Pay attention on using SQL framework avoiding so concatenation of unsanitized input to the query.

E.g.



```
function auth(){
    $user = $_POST['username'];
    $pass = $_POST['password'];
    $query = "SELECT user FROM users ";
    $query.= "WHERE user='$user' and pass='$pass'";
    $result = mysqli_query($db_connection, $query);
    return mysqli_num_rows($result) == 1;
}
```

This ridicolous piece of code tries to authenticate an user basing on his username and password. The main problem is that, as I mentioned before, these inputs are concatenated to the where clause of the query done by the server without any sort of sanitization. Instead, the mysqli framework is only used to execute a raw query without asking it to check its correctness and security.

In this particular case in order to bypass the authentication control, an attacker could insert whatever value in username and this string for the password ' OR '1'=1''. Since the two conditions are in AND, both must be true, and since the second one is a boolean evaluation ALWAYS true, the dbms will answer with the row containing the specified username. (USERNAME AND (PASSWORD='' OR TRUE) == USERNAME AND TRUE).

With SQL Injection we can retrieve whatever information we want, from tables name to master's scheme. We can also inject another query, separating them with ";".

A particular type of SQL Injection is the Blind version. Sometime a direct output of the query is not available, we have to distinguish between two behavior of the web application when the query is correctly executed and when it fails. In this case, in a blind way (without seeing the output), we can infer on the success outcome of the query. So for example, in general, what we can ask to db are boolean questions, when basing on the outcome of the web application we receive an undirect answer from the db.

Let's do an example where a web application provides a search bar for images based on their ID:

- An attacker can ask for a real image and see the outcome.
- Then he can try to inject a sql statement, that asks for the same image but with an AND clause clearly false, in order to see if the web app is vulnerable to sql injection.
- Then he can retry the same injection with an AND statement clearly true in order to see if the correct image is displayed.
- Then he can write a script that cycling asks to the server some boolean condition to the server and if he receives the image the answer is true, otherwise is false.

In order to avoid sql injection a programmer can:

- validate always the input, setting some format rules and constraints.
- sanitize the input, in order to escape some application dependent characters (like ' for sql).
- use of stored procedures, defined at server side that can validate inputs.

Notice that a timing attack is always possible although the error message cannot be obtain after a sql query. Basically, if the first part of the query is correct then it is asked the db to sleep for some milliseconds, otherwise it will return immediately and we can infer that the asked information is false.

Attacks to passwords

They may be related to:

- Poor password management at server side
- Offline dictionary attack
- Guessable and reused passwords
- Combination of previous scenarios

We can prevent these attacks choosing hard and strong passwords and using multi factor authentications. A bad and weak password is often a consequence on underestimation of risks by users and laziness.
Servers should not store passwords in clear text, they must encrypt them somehow and they must store a secret deriving from a password and not the password itself (e.g. the salted hash). Remember that we suppose that the server is always secure, but it is usual that an attacker will try

to compromise it and if the environment is not well protected, all the data are compromised by consequence.

Session hijacking

A session can be hijacked by means of:

- XSS
- Session fixation
- Session sidejacking
- Malware

Using XSS is really simple: an attacker hosts (or rent) a private server around the world, when he finds some vulnerable web sites to XSS, he can craft a malicious script to be injected that will send to the attacker's remote server the session cookie of the user. So a typical scenario is:

- Attacker hosts some http service on a remote server in order to retrieve responses from malicious scripts.
- It creates a malicious scripts ad hoc for the vulnerable web application that performs an http request to its own server passing as a parameter the session cookie of the current user under attack.
- It injects this malicious script in the vulnerable server.
- Whenever a user requests that resource to the server, the malicious script will be executed on the logon user's browser and will send to the remote server the session cookie.
- Now this cookie can be embedded in a request of the attacker claiming to be the user associated to that cookie.

This general approach is mitigated by HTTPOnly parameter of the cookie, which asserts that the cookie is not accessible from outer sources except HTTP.

Session Fixation explores a limitation in the way the web application manages the session ID. When authenticating a user, it doesn’t assign a new session ID, making it possible to use an existent session ID. The attack consists of obtaining a valid session ID (e.g. by connecting to the application), inducing a user to authenticate himself with that session ID, and then hijacking the user-validated session by the knowledge of the used session ID. The attacker has to provide a legitimate Web application session ID and try to make the victim’s browser use it.

A typical scenario is:

- An attacker knows that a web site of interest is vulnerable and he is interested in hijacking session of an user that owns a valid account for that web site.
- He sends to the user a valid link to that web site, embedding in the request a proposal of the SID for the user.
- The user click on that link and logs on in the web site.
- Now the server associates that SID to that user for a duration of a session.
- Since the SID was proposed by the attacker, he knows it. Now using that seed he can logs on claiming to be the attacked user.

Session sidejacking consists in sniffing network connection in order to steal the session cookie. Some sites uses TLS only to encrypt passwords and content of the messages exchanged but left in clear the cookie. A good prevention is to send an encrypted cookie in order to prevent its sniffing.

Another opportunity is to use a malware to hijack a session. Malware can alter the behavior of the browser by changing its settings and redirect to websites that were not meant to be visited. Malwares can also grab files/folders containing session ids.