# Security Vulnerabilities

## SQL Injection

An attack on your **database** through injection of malicious SQL statements via URL or web form. If the attacker knows or guesses the name of your database or tables, they could for example inject ‘; DROP table users’ into a field that they can see is going to be used to make a database query.

## XSS (Cross Site Scripting)

Where you have a web page that re-displays input that came from outside your system, an attacker could insert a malicious script which is then executed client-side when the page is rendered.

A common target is the comments section of a blog.

Things a malicious client-side script might do:

* deface website
* hijack session cookies
* redirect user to unwanted/malicious sites

## Session Hijacking

Session data is persistent on the server side and accessed with a session ID stored in a cookie on the client side. So if a malicious person gets the session ID and the session has not been expired they get access to the session data – bad news if the session contains authentication data.

XSS is the usual way that session IDs are stolen.

Ways to guard against session hijacking:

* change the session id as often as possible (antidote to ‘session fixation’
* use specific php.ini setting (PHP > 5.2) that prevents javascript from accessing the session id
* store session ID somewhere inaccessible (where the default maybe global /tmp dir)

## Cross Site Request Forgery (CSRF)

A logged in user is tricked into unknowingly submitting a malicious HTTP request.

The malicious HTTP request isn’t on the site the user is logged in to, but a site that the user visits whilst logged into that site. Submitting the request could be done by tricking the user into clicking on something, but it can also be something automatic e.g. a spoof image link (the malicious link is the src of the image, and the image is sized 0x0 – it is not seen but the request is sent) – this would work for a malicious GET – or a hidden form that is automatically submitted using javascript – this is what you’d need to do to submit a malicious POST request.

A major difference between this and XSS is that the perpetrator of CSRF cannot receive the contents of the HTTP response, but only benefit from its effects. So in an online banking scenario, they couldn’t get access to someone’s credentials, but they could initiate an erroneous transaction.

(Can this be in a different tab, or does it require that the user follows a link away from the logged-in site?)

CSRF attacks are defeated with use of CSRF tokens. The application generates and stores a token, in any form used by the application this token is sent via hidden form field, when a request is received nothing happens unless it includes a token that matches the token stored by the application.

## Accidentally revealing source code

If something breaks server-side, you might inadvertently serve a .php file as text rather than it being sent to the PHP processor. Particularly bad news if this is a config file. If you are using a commonly used third party application or framework, it doesn’t have to be a config file as such that is revealed but just enough code to identify the app/framework you are using – then an attacker will know where to find the config files.

The key to defeating this vulnerability is to use a correct directory structure. For instance the structure of MVC frameworks ensures nothing outside the public folder can be revealed.

Could also include under this heading incorrect error reporting, i.e. in development you will usually have error reporting switched on, but you need to be sure that it isn’t switched on in your remote site.

## Remote file inclusion (RFI)

## Directory traversal

## Remote code execution (RCE)

How is this different from RFI?