Watch out for SQL injection:
SQL injection attacks are when an attacker uses a web form field or URL parameter to gain access to or manipulate your database. When you use standard Transact SQL it is easy to unknowingly insert rogue code into your query that could be used to change tables, get information and delete data. You can easily prevent this by always using parameterised queries, most web languages have this feature and it is easy to implement.

Consider this query:
"SELECT * FROM table WHERE column = '" + parameter + "';"
If an attacker changed the URL parameter to pass in ' or '1'='1 this will cause the query to look like this:

"SELECT * FROM table WHERE column = '' OR '1'='1';"
Since '1' is equal to '1' this will allow the attacker to add an additional query to the end of the SQL statement which will also be executed.

You could fix this query by explicitly parameterising it. For example, if you're using MySQLi in PHP this should become:

```
$stmt = $pdo->prepare('SELECT * FROM table WHERE column = :value');
$stmt->execute(array('value' => $parameter));
```

Protect against XSS attacks:
Cross-site scripting (XSS) attacks inject malicious JavaScript into your pages, which then runs in the browsers of your users, and can change page content, or steal information to send back to the attacker. For example, if you show comments on a page without validation, then an attacker might submit comments containing script tags and JavaScript, which could run in every other user's browser and steal their login cookie, allowing the attack to take control of the account of every user who viewed the comment. You need to ensure that users cannot inject active JavaScript content into your pages.

This is a particular concern in modern web applications, where pages are now built primarily from user content, and which in many cases generate HTML that's then also interpreted by front-end frameworks like Angular and Ember. These frameworks provide many XSS protections, but mixing server and client rendering creates new and more complicated attack avenues too: not only is injecting JavaScript into the HTML effective, but you can also inject content that will run code by inserting Angular directives, or using Ember helpers.

Beware of error messages:
Be careful with how much information you give away in your error messages. Provide only minimal errors to your users, to ensure they don't leak secrets present on your server (e.g. API keys or database passwords). Don't provide full exception details either, as these can make complex attacks like SQL injection far easier. Keep detailed errors in your server logs, and show users only the information they need.

--> Validate on both sides:
Validation should always be done both on the browser and server side. The browser can catch simple failures like mandatory fields that are empty and when you enter text into a numbers only field. These can however be bypassed, and you should make sure you check for these validation and deeper validation server side as failing to do so could lead to malicious code or scripting code being inserted into the database or could cause undesirable results in your website.

→Check your passwords:
Everyone knows they should use complex passwords, but that doesn't mean they always do. It is crucial to use strong passwords to your server and website admin area, but equally also important to insist on good password practices for your users to protect the security of their accounts.

As much as users may not like it, enforcing password requirements such as a minimum of around eight characters, including an uppercase letter and number will help to protect their information in the long run.

Passwords should always be stored as encrypted values, preferably using a one way hashing algorithm such as SHA. Using this method means when you are authenticating users you are only ever comparing encrypted values. For extra website security it is a good idea to salt the passwords, using a new salt per password.

In the event of someone hacking in and stealing your passwords, using hashed passwords could help damage limitation, as decrypting them is not possible. The best someone can do is a dictionary attack or brute force attack, essentially guessing every combination until it finds a match. When using salted passwords, the process of cracking a large number of passwords is even slower as every guess has to be hashed separately for every salt + password which is computationally very expensive.

Thankfully, many CMSes provide user management out of the box with a lot of these website security features built in, although some configuration or extra modules might be required to use salted passwords (pre Drupal 7) or to set the minimum password strength. If you are using .NET then it's worth using membership providers as they are very configurable, provide inbuilt website security and include readymade controls for login and password reset.

→Avoid file uploads
Allowing users to upload files to your website can be a big website security risk, even if it's simply to change their avatar. The risk is that any file uploaded, however innocent it may look, could contain a script that when executed on your server, completely opens up your website.

If you have a file upload form then you need to treat all files with great suspicion. If you are allowing users to upload images, you cannot rely on the file extension or the mime type to verify that the file is an image as these can easily be faked. Even opening the file and reading the header, or using functions to check the image size are not foolproof. Most images formats allow storing a comment section that could contain PHP code that could be executed by the server.

So what can you do to prevent this? Ultimately you want to stop users from being able to execute any file they upload. By default web servers won't attempt to execute files with image extensions, but don't rely solely on checking the file extension as a file with the name image.jpg.php has been known to get through.

Some options are to rename the file on upload to ensure the correct file extension, or to change the file permissions, for example, chmod 0666 so it can't be executed. If using *nix, you could create a .htaccess file (see below) that will only allow access to set files preventing the double extension attack mentioned earlier.

deny from all

```
<Files ~ "^\w+\.(gif|jpe?g|png)$">
order deny,allow
allow from all
</Files>
```

Ultimately, the recommended solution is to prevent direct access to uploaded files altogether. This way, any files uploaded to your website are stored in a folder outside of the webroot or in the database as a blob. If your files are not directly accessible you will need to create a script to fetch the files from the private folder (or an HTTP handler in .NET) and deliver them to the browser. Image tags support an src attribute that is not a direct URL to an image, so your src attribute can point to your file delivery script providing you set the correct content type in the HTTP header. For example:

```
<img src="/imageDelivery.php?id=1234" />

<?php
    // imageDelivery.php

    // Fetch image filename from database based on $_GET["id"]
    ...

    // Deliver image to browser
     Header('Content-Type: image/gif');
    readfile('images/'.$fileName);

?>
```

Most hosting providers deal with the server configuration for you, but if you are hosting your website on your own server then there are few things you will want to check.

Ensure you have a firewall setup, and are blocking all non essential ports. If possible setting up a DMZ (Demilitarised Zone) only allowing access to port 80 and 443 from the outside world. Although this might not be possible if you don't have access to your server from an internal network as you would need to open up ports to allow uploading files and to remotely log in to your server over SSH or RDP.

If you are allowing files to be uploaded from the Internet only use secure transport methods to your server such as SFTP or SSH.

If possible have your database running on a different server to that of your web server. Doing this means the database server cannot be accessed directly from the outside world, only your web server can access it, minimising the risk of your data being exposed.

Finally, don't forget about restricting physical access to your server.

Use HTTPS

HTTPS is a protocol used to provide security over the Internet. HTTPS guarantees that users are talking to the server they expect, and that nobody else can intercept or change the content they're seeing in transit.

If you have anything that your users might want private, it's highly advisable to use only HTTPS to deliver it. That of course means credit card and login pages (and the URLs they submit to) but typically far more of your site too. A login form will often set a cookie for example, which is sent with every other request to your site that a logged-in user makes, and is used to authenticate those

requests. An attacker stealing this would be able to perfectly imitate a user and take over their login session. To defeat these kind of attacks, you almost always want to use HTTPS for your entire site.

That's no longer as tricky or expensive as it once was. Let's Encrypt provides totally free and automated certificates, which you'll need to enable HTTPS, and there are existing community tools available for a wide range of common platforms and frameworks to automatically set this up for you.

Notably Google have announced that they will boost you up in the search rankings if you use HTTPS, giving this an SEO benefit too. Insecure HTTP is on its way out, and now's the time to upgrade.

Already using HTTPS everywhere? Go further and look at setting up HTTP Strict Transport Security (HSTS), an easy header you can add to your server responses to disallow insecure HTTP for your entire domain.

Get website security tools:
Once you think you have done all you can then it's time to test your website security. The most effective way of doing this is via the use of some website security tools, often referred to as penetration testing or pen testing for short.

There are many commercial and free products to assist you with this. They work on a similar basis to scripts hackers in that they test all know exploits and attempt to compromise your site using some of the previous mentioned methods such as SQL Injection.

Some free tools that are worth looking at:
Netsparker (Free community edition and trial version available). Good for testing SQL injection and XSS
OpenVAS Claims to be the most advanced open source security scanner. Good for testing known vulnerabilities, currently scans over 25,000. But it can be difficult to setup and requires a OpenVAS server to be installed which only runs on *nix. OpenVAS is fork of a Nessus before it became a closed-source commercial product.
SecurityHeaders.io (free online check). A tool to quickly report which security headers mentioned above (such as CSP and HSTS) a domain has enabled and correctly configured.
Xenotix XSS Exploit Framework A tool from OWASP (Open Web Application Security Project) that includes a huge selection of XSS attack examples, which you can run to quickly confirm whether your site's inputs are vulnerable in Chrome, Firefox and IE.
The results from automated tests can be daunting, as they present a wealth of potential issues. The important thing is to focus on the critical issues first. Each issue reported normally comes with a good explanation of the potential vulnerability. You will probably find that some of the medium/low issues aren't a concern for your site.

There are some further steps you can take to manually try to compromise your site by altering POST/GET values. A debugging proxy can assist you here as it allows you to intercept the values of an HTTP request between your browser and the server. A popular freeware application called Fiddler is a good starting point.

So what should you be trying to alter on the request? If you have pages which should only be visible to a logged in user then try changing URL parameters such as user id, or cookie values in an attempt to view details of another user. Another area worth testing are forms, changing the POST values to attempt to submit code to perform XSS or uploading a server side script.