



WHITE PAPER

Common web application security problems and how to identify them

Part 4 Cross site scripting

By

Lee J Lawson
Lead Penetration Tester, **dns**.

Introduction

This is **part 4** of a series of papers designed to raise the level of security knowledge regarding common security flaws in web applications. All papers revolve around the top 10 list of web application security issues as defined by the Open Web Application Security Project (OWASP). The last issue described the vulnerability known as **Broken authentication and session management**. This paper concentrates on **Cross site scripting**, the identification and resolution of this flaw.

Here is a list of the most common security flaws at the time of writing taken directly from the OWASP site.

1. **Unvalidated input**
2. **Broken access control**
3. **Broken authentication and session management**
4. **Cross site scripting**
5. **Buffer overflows**
6. **Injection flaws**
7. **Improper error handling**
8. **Insecure storage**
9. **Application denial of service**
10. **Insecure configuration management**

It should be noted that this paper is not intended to be used as a replacement to professional penetration testing. Rather it is aimed toward raising the level of security knowledge in the community. Professional penetration testing requires years of experience to be able to apply worthwhile interpretation to results from testing tools and gain the skills required to perform manual assurance testing.

Cross site scripting

The web application can be used as a mechanism to transport an attack to an end user's browser. A successful attack can disclose the end user's session token, attack the local machine, or spoof content to fool the user.

this security flaw has become one of the most common attack vectors

The Problem

This security flaw has become one of the most common attack vectors. In fact at the time of writing, the as yet unpublished 2007 OWASP top 10 list places it at the top. It is similar in essence to the vulnerability described in Part 1 of this series, that of unvalidated input. However Cross-Site Scripting, commonly referred to as XSS, tends to have a different end target in mind.

XSS occurs when malicious executable code is injected into a web application which is then echoed back to the browser without any validation or encoding. This allows an attacker to input HTML or scripting code and have it execute on the browser. The code execution was sand-boxed to the web browser and therefore could only obtain information from the browser, until BlackHat 2006! We will cover that later.

The reason that XSS is called Cross-Site Scripting is because any scripting code injected into an application web page is executed on the web browser that is being used to view that page, no matter where they are in the world.

To explain XSS fully, let's use an example.

Consider a web application that provides a discussion forum for their authenticated users. A user has a valid username and password combination which they use to authenticate and be presented with a custom homepage. After clicking on the link for the forum, the application checks the user's cookie for their permissions and presents the forum page where they can read, post and answer other users' questions.

When a question is written, the text is presented back to the screen and parsed as code, this means that `
` would be interpreted as a HTML tag, the break command instead of the plain text of `
`.

Any normal user would use this system to ask and answer questions about whatever subject the forum is about. However, a malicious user may craft questions in such a way that scripting code is executed when another user either views their question or clicks on a link.

Let's say that an attacker posts this question:

Hi all, you can get great info on XSS [here](#).

Upon closer inspection, the [here](#) link actually has some JavaScript embedded within it.

```
<a href="http://www.dns.co.uk">
<script>alert(document.cookie)</script> here </a>
```

once the attacker has the cookie of the unsuspecting user, they apply it to their own browser and use it to authenticate against the web application

This would simply execute an alert box with the user's cookie details within it when it is clicked, but there are very simple ways of taking the users cookie information and sending it to a waiting server ready to be read and used by an attacker. Once the attacker has the cookie of the unsuspecting user, they apply it to their own browser and use it to authenticate against the web application.

So there are actually two victims to this attack vector, a secondary victim - the web application and the primary victim – the user. The web application is being used as a means to spread the attack whereas the user is the final victim.

There could also be a tertiary (third) victim. Let's say that an online search engine presents unchecked search criteria text back to the browsers screen. A hyperlink could be crafted that executes JavaScript code by clicking it. This hyperlink could then be emailed or posted on forums where no XSS flaw exists. These tertiary victims have little to fear directly, however they are being used to spread the attack further.

So far we have only been able to retrieve a user's application cookie, but that is a critical failure if that cookie is being used to authenticate users against an online banking web application. Consider the implications then!

As I mentioned earlier, Black Hat 2006 was used as a platform by SPI Dynamics to showcase a new attack method using XSS attacks. Previously, XSS was 'sandboxed' to the web browser and could only retrieve information from that browser. Now XSS has broken out of the browser and has entered the network.

there are very complex attacks utilising XSS scripting which can port scan the internal network of the primary victim, the user

There are very complex attacks utilising XSS scripting which can port scan the internal network of the primary victim, the user. This information can then be transported back to the attacker and be used to formulate an attack plan. Again we have a secondary (application) and primary (user) victims, but this time the real threat is against the primary victim as it is their network which is being probed. The web applications owners should fear this new trend also. How long before a user successfully sues a web application owner for having a XSS flaw which lead to that users network being scanned? I think this will happen sooner than some expect!

Identification

Identifying the problem requires careful analysis of your web applications; all user inputs must be checked. All methods of passing text to the application must be checked and all pages returned from the application must be analysed in case the script was executed silently.

If you think that you have found a method of passing text to your application and see that text presented back to the screen, say through an error message, search page or forum, then try entering this script text:

```
<script>alert(document.cookie)</script>
```

If an alert box pops up on your screen, then you are vulnerable. Be aware that pop up blockers can prevent this code from executing. This is not a security measure as you cannot be sure that all users will use pop up blockers!

Countermeasures

There are many ways that XSS flaws can be eradicated from your applications, but the most effective method is to filter and check ALL user input. Strip out dangerous characters such as <>/ etc.

The resolution methods are very similar to prevented **Unvalidated Input** attack vectors as described in Part 1 of this series. It is always better to filter input and allow only known safe characters such as a-z, A-Z and 0-9 rather than trying to make a filter which will attempt to prevent all possible attacks and hope it is good enough to prevent all future attack methods.

Another method used on many discussion forums is to use 'pseudo' code. This means that you use your own code tags to for instance, make text bold or italic. By using your own code command tags, standard XSS code will not execute at all or at the very least not as expected by the attacker.

automated tools combined
with manual verification
testing should be used

The problem you will face is identifying all possible input points to the application. There could be hundreds if not thousands for a bigger application. This is where automated tools combined with manual verification testing should be used. Ensure that whoever is performing these checks has extensive experience in web based coding techniques and penetration testing as a whole.

The next part of this series concentrates on **Buffer overflows** and how they are used to exploit systems.