Cross-site scripting

Use a custom tag library to encode dynamic content

Paul Lee

September 01, 2002

Cross-site scripting is a potentially dangerous security exposure that should be considered when designing a secure Web-based application. In this article, Paul describes the nature of the exposure, how it works, and has an overview of some recommended remediation strategies.

Most Web sites today add dynamic content to a Web page making the experience for the user more enjoyable. Dynamic content is content generated by some server process, which when delivered can behave and display differently to the user depending upon their settings and needs. Dynamic Web sites have a threat that static Web sites don't, called "cross-site scripting," also known as "XSS."

"A Web page contains both text and HTML markup that is generated by the server and interpreted by the client browser. Web sites that generate only static pages are able to have full control over how the browser user interprets these pages. Web sites that generate dynamic pages do not have complete control over how their outputs are interpreted by the client. The heart of the issue is that if untrusted content can be introduced into a dynamic page, neither the Web sites nor the client has enough information to recognize that this has happened and take protective actions," according to CERT Coordination Center, a federally funded research and development center to study Internet security vulnerabilities and provide incident response.

Cross-site scripting is gaining popularity among attackers as an easy exposure to find in Web sites. Every month cross-site scripting attacks are found in commercial sites and advisories are published explaining the threat. Left unattended, your Web site's ability to operate securely, as well as your company's reputation, may become victim of the attacks.

This article is written to raise the awareness of this emerging threat and to present a solution implementation for Web applications to avoid this kind of attack.

The threats of cross-site scripting

Cross-site scripting poses server application risks that include, but are not limited to, the following:
• Users can unknowingly execute malicious scripts when viewing dynamically generated pages based on content provided by an attacker.
• An attacker can take over the user session before the user's session cookie expires.
• An attacker can connect users to a malicious server of the attacker's choice.
• An attacker who can convince a user to access a URL supplied by the attacker could cause script or HTML of the attacker's choice to be executed in the user's browser. Using this technique, an attacker can take actions with the privileges of the user who accessed the URL, such as issuing queries on the underlying SQL databases and viewing the results and to exploit the known faulty implementations on the target system.

Launching an attack

After an application on a Web site is known to be vulnerable to cross-site scripting, an attacker can formulate an attack. The technique most often used by attackers is to inject JavaScript, VBScript, ActiveX, HTML, or Flash for execution on a victim's system with the victim's privileges. Once an attack is activated, everything from account hijacking, changing of user settings, cookie theft and poisoning, or false advertising is possible.

Sample attack scenarios

The following scenario diagrams illustrate some of the more relevant attacks. We will not, however, be able to list all variants of the vulnerability. To learn more about the documented attacks and how to protect yourself as a vendor or as a user, see the Related topics section.

Scripting via a malicious link

In this scenario, the attacker sends a specially crafted e-mail message to a victim containing malicious link scripting such as one shown below:

```
<A HREF=http://legitimateSite.com/registration.cgi?clientprofile=<SCRIPT>malicious
code</SCRIPT>>Click here</A>
```

When an unsuspecting user clicks on this link, the URL is sent to legitimateSite.com including the malicious code. If the legitimate server sends a page back to the user including the value of clientprofile, the malicious code will be executed on the client Web browser as shown in Figure 1.
Stealing users' cookies

If any part of the Web site uses cookies, then it may be possible to steal them from its users. In this scenario, the attacker files a page with malicious script to the part of the site that is vulnerable. When the page is displayed, the malicious script runs, collects the users' cookies, and sends a request to the attacker's Web site with the cookies gathered. Using this technique, the attacker can gain sensitive data such as passwords, credit card numbers, and any arbitrary information the user inputs as shown in Figure 2.

Figure 2. Cookie theft and account hijacking
Sending an unauthorized request

In this scenario, the user unknowingly executes scripts written by an attacker when they follow a malicious link in a mail message. Because the malicious scripts are executed in a context that appears to have originated from the legitimate server, the attacker has full access to the document retrieved and may send data contained in the page back to their site. If the embedded script code has additional interactions capability with the legitimate server without alerting the victim, the attacker could develop and exploit that posted data to a different page on the legitimate Web server as shown in Figure 3.

Figure 3. Sending an unauthorized request

Avoiding an attack

As stated above, cross-site scripting is achieved when an attacker is able to cause a legitimate Web server to send a page to a victim user's Web browser that contains a malicious script of the attacker's choosing. The attacker then has the malicious script run with the privileges of a legitimate script originating from the legitimate Web server.

Now that we know the basis of an attack, what can we do to protect ourselves from this vulnerability?

Web site developers can protect their sites from being abused in conjunction with these attacks by ensuring that dynamically generated pages do not contain undesired tags.

From the Web user's perspective, two options exist to reduce the risk of being attacked through this vulnerability. The first -- disabling scripting languages in the Web browser as well as the HTML-enabled e-mail client -- provides the most protection but has the side effect of disabling functionality. The second -- only following links from the main Web site for viewing -- will significantly reduce a user's exposure while still maintaining functionality.
However, none of the solutions that Web users can take are complete solutions. In the end, it is up to Web page developers to modify their pages to eliminate these types of problems. This can be accomplished by properly filtering and validating the input received and properly encoding or filtering the output returned to the user.

**Filtering**

The basis of this approach is never trust user input and always filter metacharacters ("special" characters) that are defined in the HTML specification. Each input field, including link parameters will be validated for script tags. When found and dependent on the context, the input will be rejected and thus prevent the malicious HTML from being presented to the user.

Adding to the complexity is that many Web browsers try to correct common errors in HTML. As a result, they sometimes treat characters as special when, according to the specification, they are not. Therefore, it is important to note that individual situations may warrant including additional characters in the list of special characters. Web developers must examine their applications and determine which characters can affect their web applications.

Filtering on the input side is less effective because dynamic content can be entered into a Web site database via methods other than HTTP. In this case, the Web server may never see the data as part of the data input process and the data elements still remain tainted. Alternatively, it is recommended that filtering be done as part of the data output process, just before it is rendered as part of the dynamic page. Done correctly, this approach ensures that all dynamic content is filtered.

**Encoding**

Cross-site scripting attacks can be avoided when a Web server adequately ensures that generated pages are properly encoded to prevent unintended execution of scripts.

Each character in the ISO-8859-1 specification can be encoded using its numeric entry value. Server side encoding is a process where all dynamic content will go through an encoding function where scripting tags will be replaced with codes in the chosen character set.

Generally speaking, encoding is recommended because it does not require you to make a decision about what characters could legitimately be entered and need to be passed through. Unfortunately, encoding all untrusted data can be resource intensive and may have a performance impact on some Web servers.

**Which strategy is right for me?**

CGI-based Web applications or applications that favor field edit check at the browser will likely adapt to the filtering strategy by extending the existing field edit check to cover the cross-site scripting vulnerability. Note that although browser side field edit check saves a few runs back to the server, it only works for the honest user and requires thorough code walkthroughs to guarantee that all input fields are checked in order to meet the remediation recommendation. Web applications with server side validation designed-in, however, can have a choice to adapt to either or both strategies.
For the filtering strategy to work properly, Web developers need to ensure that the list of metacharacters for filtering is up-to-date according to the needs of their applications. The encoding strategy, on the other hand, does not have the above-described maintenance effort, and it also has less impact on the existing application code as well as on application functionality. For these reasons, the encoding strategy appears to be a favorite choice of implementation. A sample encoding implementation is described next.

The 1-2-3 of a sample encoding

A simple, yet effective, way for a Web server to ensure that the generated pages are properly encoded is to pass each character in the dynamic content through an encoding function where the scripting tags in the dynamic content are replaced with codes in the chosen character set. This task is perfect for a custom tag library.

Custom tag library basics

A custom tag library is comprised of one or more Java language classes (called tag handlers) and an XML tag library description file (TLD), which dictates the new tag names and valid attributes for those tags. Tag handlers and TLDs determine how the tags, their attributes, and their bodies will be interpreted and processed at request time from inside a JSP page. A custom tag library provides an architecture that is more flexible than a Java bean at encapsulating a complex operation.

Our "custom" fitted tag library

What better name is there for our custom tag library besides naming it XSS? Tag libraries are software components that are plugged into a servlet container. The servlet container creates tag handlers, initializes them and calls the doStartTag(), doEndTag() and release() methods, in that order.

Through these interactions, our XSS custom tag library will be able to apply the "custom" action of encoding the dynamic data found on a JSP page. Implementing custom tags is straightforward and the steps are as follows:

- Create a tag library descriptor (.tld) describing the tags.
- Add a taglib directive to JSP files that use the tags.
- Implement a tag handler that extends TagSupport and overrides the doStartTag() or the doEndTag() methods.

TLD (tag library descriptor)

A tag library descriptor is an XML file whose elements describe a particular tag library. The tld file for our XSS custom tag library, is shown in Listing 1. The tag element defines the encode action, including an attribute, property. The tagclass element defines the tag handler class EncodeTag.
Listing 1. The xss.tld file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<doctype taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <tag>
    <name>encode</name>
    <tagclass>dw.demo.xss.EncodeTag</tagclass>
    <bodycontent>empty</bodycontent>
    <attribute>
      <name>property</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>
```

The taglib directive

The taglib directive identifies the tag library descriptor and defines the tag prefix that associates subsequent tags with the library. A sample taglib directive, which appears in the JSP using the xss custom tag library, is shown below:

```
<%@ taglib uri="/WEB-INF/tlds/xss.tld" prefix="xss" %>
```

Coding the tag handler

A tag handler is an object in the Web container that helps evaluate actions when a JSP page executes. The EncodeTag class is the tag handler for the encode action. Its `doStartTag` method, which encodes the dynamic content to the ISO-8859-1 character set, is shown in Listing 2.

Listing 2. Encoding the dynamic content

```java
public int doStartTag() throws JspException {
    StringBuffer sbuf = new StringBuffer();
    char[] chars = property.toCharArray();
    for (int i = 0; i < chars.length; i++) {
        sbuf.append("&#" + (int) chars[i]);
    }
    try{
        pageContext.getOut().print(sbuf.toString());
    } catch (IOException ex) {
        throw new JspException(ex.getMessage());
    }
    return SKIP_BODY;
}
```

Deployment

The xss custom tag library, which is part of a Web application, is packaged as additional files into the Web application’s WAR file as follows:
Putting it to work

The following scenario illustrates how the custom tag library would be used. Suppose that a hypothetical Web site for receiving articles included a page for reviewing articles that you have subscribed-to. The dynamic content, article items intended for you, is prepared inside a JSP file using the `<%= expression %>` syntax.

Let us assume an attacker succeeded in filling a page containing malicious script to the Web site for the subscribed members. The effect of this successful attack, which when executed on the user browser will cause a popup window to be displayed, is demonstrated in Figure 4.

**Figure 4. Before encoding**

```jsp
<jsp:useBean id="Bulletins" class="dw.demo.xss.Bulletins" scope="session" />
<demo:iterate collection="<%= Bulletins.getItems() %>">
   <jsp:useBean id="item" scope="page" class="java.lang.String"/>
   Items: `<%= item %>` <br>
</demo:iterate>
```

In the next scenario, the hypothetical Web site ensures that the generated pages are properly encoded by using the `xss` custom tag library and is able to protect itself from the attack. The untrusted data is preserved for visual appearance in the browser as shown in Figure 5.
Summary

In this article, we discussed how attackers use cross-site scripting as a technique to launch attacks against Web sites. We have also demonstrated that the majority of the attacks can be eliminated when a Web site uses a simple custom tag library to properly encode the dynamic content. Use the xss custom tag library as-is or, better yet, change it to fit your Web application needs and become protected from this emerging threat.
Related topics

- The CERT Coordination Center is a federally funded research and development center operated by Carnegie Mellon University. They study Internet security vulnerabilities and provide security incidents and publish responses.
- Read this article from the CERT organization on Malicious HTML Tags Embedded in Client Web Requests.

© Copyright IBM Corporation 2002
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)