SECURING APACHE : XSS INJECTIONS - II

Types of XSS vulnerabilities

Most XSS vulnerabilities are classified into three types, based on how the attacker exploits the processing of the code they injected, by the Web application. These types are:

- ♣ Persistent or stored vulnerabilities
- ♣ Non-persistent or reflected vulnerabilities
- ♣ DOM-based or local vulnerabilities

Let’s look at each of these in turn.

Persistent or stored vulnerabilities

The persistent XSS vulnerability is the most powerful and effective of all. It exists when data that’s provided to a Web application by a user is first stored persistently on the server (in a database, file system, or other storage), and later displayed to users in a Web page without being properly sanitised.

The attack scenario on an online message board, given above, is a classic example of this. An attack based on a persistent vulnerability is visualised in Figure 3.

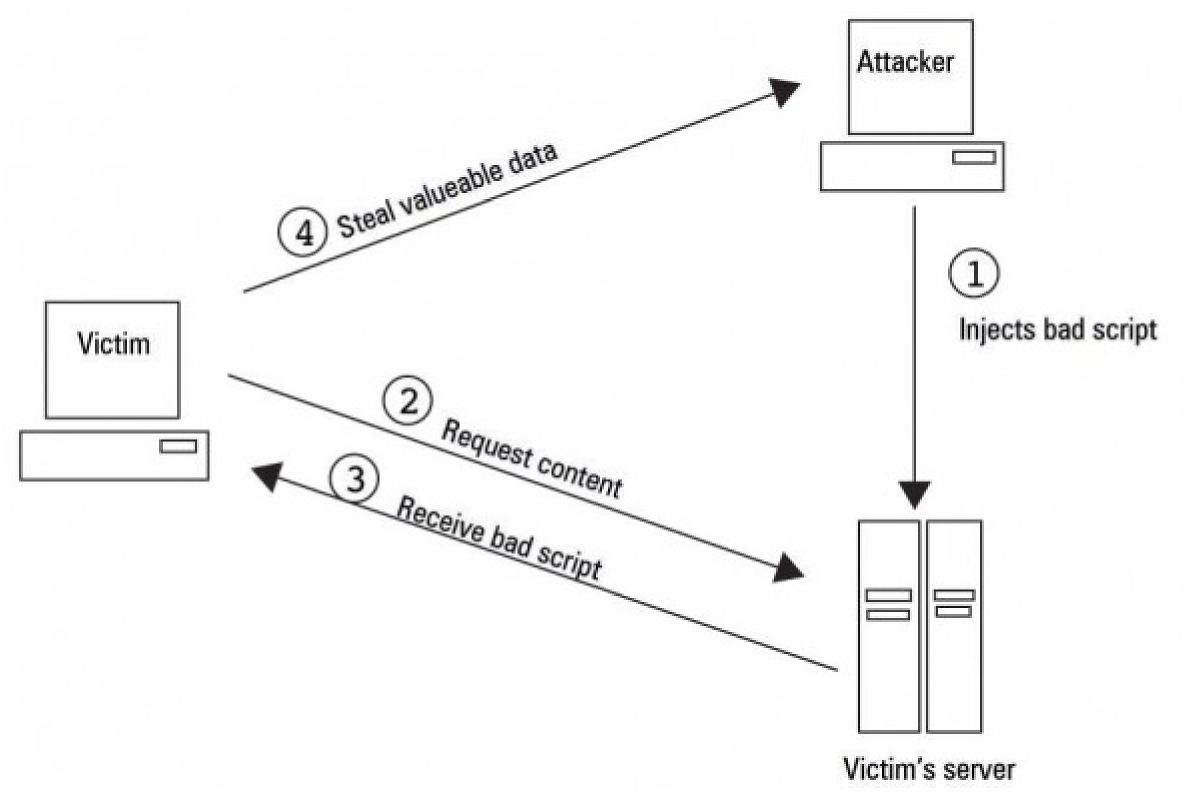

Figure 3: Visualisation of persistent XSS vulnerability

The procedure is that the attacker first injects a malicious script through an input Web form. This is then stored by the server in its database. When any user requests the page, the malicious script is rendered into it. Anyone who clicks the link (or merely views the message, in case of the IFRAME-based attack) becomes a victim, as the malicious script is executed in the victim's browser, passing the authentication cookies back to the attacker.

This type of vulnerability is very effective, since the attacker can target several users of the server — whoever clicks on the link or message.

Non-persistent or reflected vulnerabilities

The non-persistent XSS vulnerability is by far the most widely exploited. This type of XSS vulnerability is commonly triggered by server-side scripts that use non-sanitised user-supplied data when rendering the HTML document.

For example, an attacker finds an XSS vulnerability in a Web application, where the application's script displays the criteria used in the website query, as well as the results for the query. The usual URL in the browser might be `http://www.example.com/search.php?query=products`. Normally, this link would display products available from the website. Once the attackers find the vulnerability, in an effort to hijack the victim's credentials, they might post a modified link (which changes the known variables) to the victim: `http://www.example.com/search.php?query=<script>alert(document.cookie)</script>`

Clicking this link will cause the victims' browser to pop up an alert box showing their current set of cookies. This particular example is harmless; an attacker can do much more damage, including stealing passwords, resetting the victim's home page, or redirecting the victim to another website, by using modified JavaScript code.

A visualisation of an attack using a reflected vulnerability is shown in Figure 4.

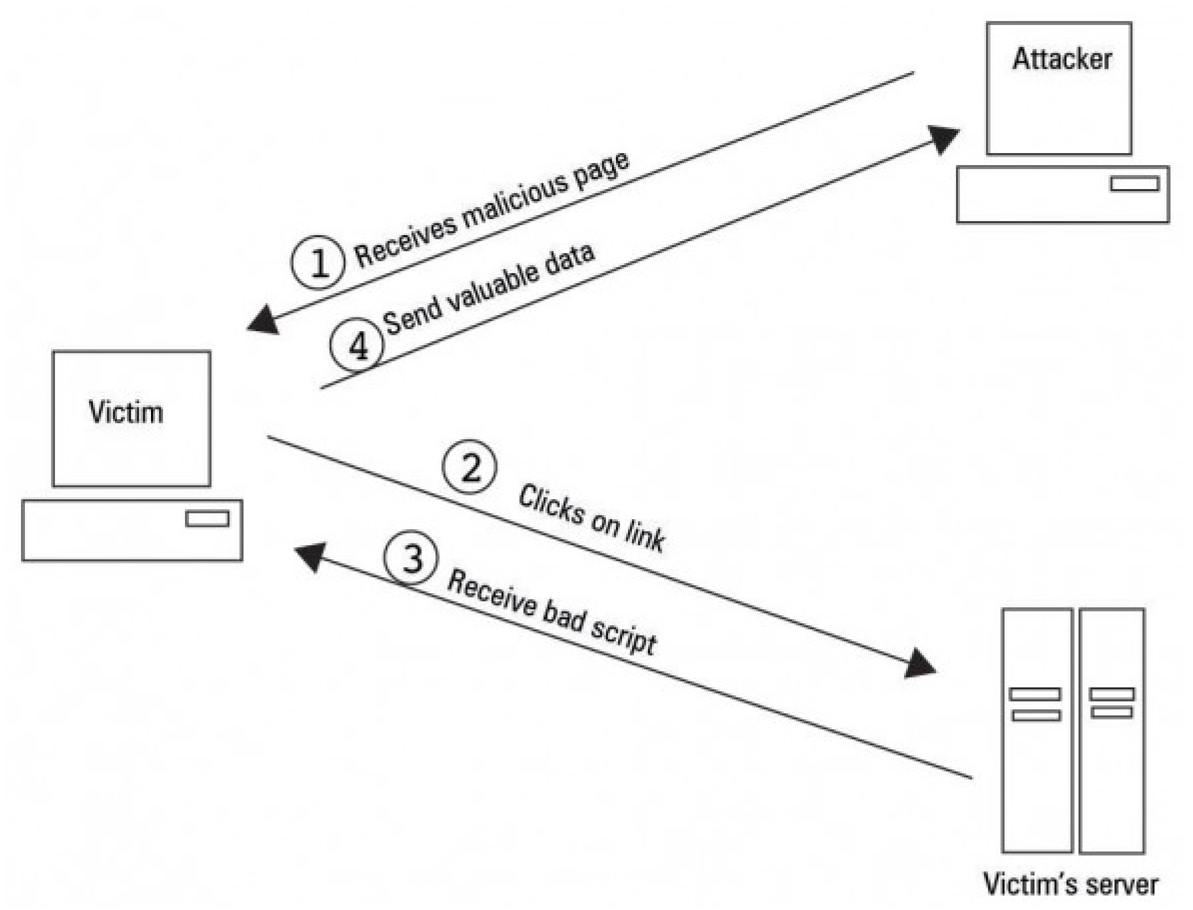

Figure 4: Stepwise attack using reflected vulnerability

Now, embedding such bulky scripts might draw the victim's attention, so attackers simply convert these into hexadecimal format using one of the many converters available, such as`http://code.cside.com/3rdpage/us/url/converter.html`. Moreover, if the malicious script is quite big, then URL-shortening services like Tiny URL are used to create a short URL that maps to the long one.

DOM-based or local vulnerabilities

DOM-based XSS vulnerabilities exist within the sites' HTML (as a static script) and can be exploited non-persistently. A brief example of a DOM-based XSS vulnerability would be a static script embedded in a page, which, when executed, uses a DOM function like `document.write` to display the results of a POST variable.

The only real difference in the DOM-based vulnerability is that the server doesn't send back the results; instead, the DOM parses the code locally, and the malicious script is executed with the same privilege as the browser on the victim's machine. Consider a scenario where a vulnerable site has the following content (named, for example, `http://www.example.com/welcome.html`):

```
<HTML>
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome to our site
…
</HTML>
```

Normally, the code in this page would welcome the user, if invoked with the following URL:`http://www.example.com/welcome.html?name=Joe`

However, a little tampering with this URL results in displaying the users' cookies in their browser, if they click the URL hyperlink: `http://www.example.com/welcome.html?name=<script>alert(document.cookie)</script>`

What happens is that to open this URL, the victim's browser sends an HTTP request to`www.example.com`. It receives the above (static) HTML page. The victim's browser then starts parsing this HTML into DOM. In this case, the code references `document.URL`, and so, a part of this string is embedded at parsing time in the HTML. It is then immediately parsed, and the malicious JavaScript code passed through the URL is executed in the context of the same page, resulting in an XSS attack.

You might realise here that the payload did arrive at the server (in the query part of the HTTP request), and so it could be detected just like any other XSS attack — but attackers even take care of that with something like the following: `http://www.example.com/welcome.html#name=<script>alert(document.cookie)<script>`

Notice the hash sign (`#`) used here; it tells the browser that everything beyond it is a fragment, and not part of the query. IE (6.0) and Mozilla do not send the fragment to the server, and for these browsers, the server would only see `http://www.example.com/welcome.html`, with the payload remaining hidden.

Source : http://www.opensourceforu.com/2010/09/securing-apache-part-2-xss-injections/