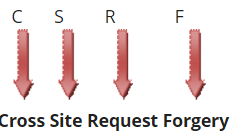
**What is CSRF?**



Cross Site Request forgery can be defined as, a forgery request, i.e a fraud or fake request, which comes on an authenticated site from a cross site and is treated as an authenticated request. This type of attack has been called a CSRF. For avoiding this situation, Microsoft provides us built-in functionality which we use in our application for security purposes, so no one can hack our site or invade some critical information.

**Live - Scenario**

Suppose, you are logged into your bank account and are going to transfer some money to your friend. A hacker knows that you are logged in and also knows the URL of the money transfer submission. Suddenly, you get an email and check it.  You see an image and by mistake, you click on that. Then, after a minute or so, you get another message that some amount has been deducted from your account. Actually, that image had been sent by the hacker and behind that image a URL has been submitted on your click.

Web applications are exposed to several security threats such as cross-site scripting attacks and cross-site request forgery. In an attempt to assist developers protect their web applications from these attacks ASP.NET provides various techniques. To that end this article shows how an ASP.NET MVC web site can be secured from Cross Site Request Forgery (CSRF).

Before we go into any details let's quickly discuss what CSRF is. Suppose that you have an ASP.NET MVC web site with two views - Index.aspx and SaveData.aspx. The Index.aspx is shown below:

<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<dynamic>" %>

<!DOCTYPE html>

<html>

<head runat="server">

<title>Index</title>

</head>

<body>

<div>

<form action="/home/savedata" method="post">

<input type="submit" value="Submit from App1" />

</form>

</div>

</body>

</html>

As you can see the Index.aspx consists of a <form> that gets submitted to an action method SaveData() when the Submit button is clicked. The SaveData() method does some internal processing and is supposed to save some data in the database. The skeleton of SaveData() is shown below:

public ViewResult SaveData()

{

//do some processing and data saving here

return View();

}

In most of the case the <form> shown above will be posting to the SaveData() action method. However, consider a case where some altogether different application starts posting data to SaveData() action method. For example, consider another view Index.aspx from a different ASP.NET MVC application:

<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<dynamic>" %>

<!DOCTYPE html>

<html>

<head runat="server">

<meta name="viewport" content="width=device-width" />

<title>Index</title>

</head>

<body>

<div>

**<form action="http://localhost:49172/home/savedata" method="post">**

**<input type="submit" value="Submit from App2" />**

**</form>**

</div>

</body>

</html>

*(Note that this is just a simplified example intended to give you an idea as to how CSRF works. In a more realistic case a user who has been authenticated in a web application is tricked by a malicious website to request a resource belonging to the authenticated website without his knowledge.)*

Have a look at the marked in bold letters. Here some another application is posting data to your controller action method. Obviously you may want to avoid such situations because then any malicious website can send data to your web application and can even participate in financial transactions. It must be remembered that CSRF is based on the basic way in which any web site works. There are various ways to prevent CSRF including:

* Having a Captcha system for your forms
* Checking domain referrer in your code
* Unique user specific identifier / token

Luckily, ASP.NET MVC includes a readymade technique that you can use in your applications. What you can do is:

* Use AntiForgeryToken() HTML helper in the view.
* Decorate the controller action method with [ValidateAntiForgeryToken] attribute.

To perform the first step open the Index.aspx and change the <form> section as shown below:

<form action="/home/savedata" method="post">

**<%= Html.AntiForgeryToken() %>**

<input type="submit" value="Submit from App1" />

</form>

Here, you added Html.AntiForgeryTken() call in the <form>. When you do this, ASP.NET MVC emits a cookie and a form field with an anti-forgery token (an encrypted token).

To perform the second step, go to the SaveData() action method and decorate it with [ValidateAntiForgeryToken] attribute as shown below:

[HttpPost]

**[ValidateAntiForgeryToken]**

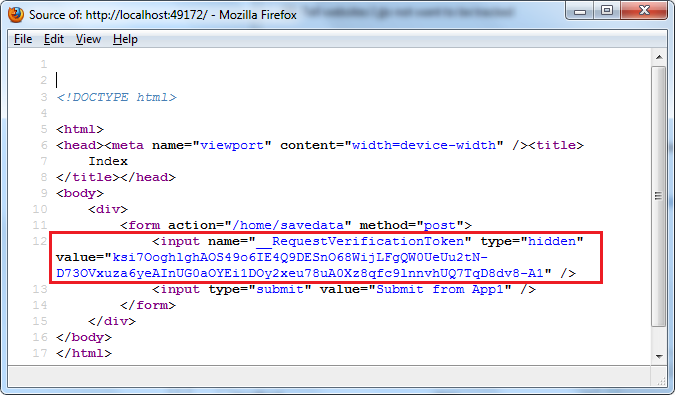
public ViewResult SaveData()

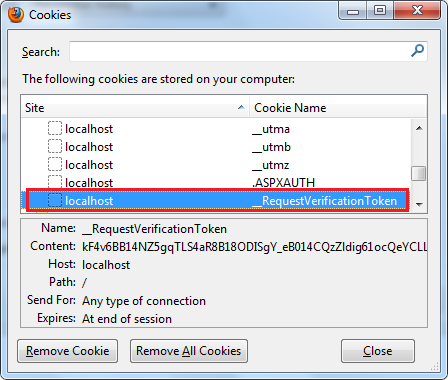
{

return View();

}

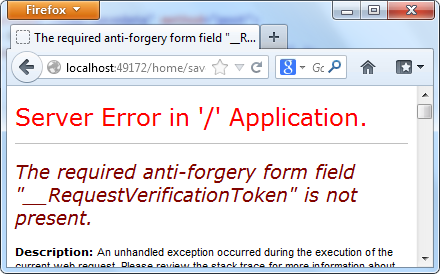
 The following figures show the hidden form field and the cookie being emitted by the Html.AntiForgeryToken().





As you can see the <form> now has a hidden field with name \_RequestVerificationToken. There is also a cookie with the same name. Once the [ValidateAntiForgeryToken] attribute is set the controller will check that the incoming request has the request verification cookie and the hidden request verification form field.

When another web application tries to post data to your form now, the other application won't know the values of  request verification cookie and the hidden request verification form field and as a result an exception will be thrown as shown below:



It must be remembered that this technique requires a cookie enabled browser for obvious reasons. Additionally this technique works with POST requests only. If you are using jQuery $.ajax() to make Ajax calls to the controller action methods that are marked with [ValidateAntiForgeryToken] attribute it can create problems because $.ajax() won't pass the hidden form field and the cookie automatically as the classic form submission technique does.