**Cross-Site Request Forgery** (**CSRF**) is a type of attack that occurs when a malicious web site, email, blog, instant message, or program causes a user's web browser to perform an unwanted action on a trusted site for which the user is currently authenticated.

**What is the AntiForgeryToken and why do I need it? - ASP.NET MVC 101 series**

|  |  |
| --- | --- |
|  |  |

[*This article is from our ASP.NET MVC 101 Tutorial Series*](http://www.devcurry.com/p/aspnet-mvc-101-tutorials-series.html)  
  
When we get started with projects, often our core focus is Domain and Business logic. We often slack up on the nitty gritty of securing our web input and sometimes it continues till a disaster strikes and at that point, it becomes a major PR as well as legal headache.  
  
Well today, we will look at a type of security breach in a web application that is called *Cross Site Request Forgery or CSRF hack*. CSRF is the lesser known cousin of XSS. We will look at XSS in a few days’ time.

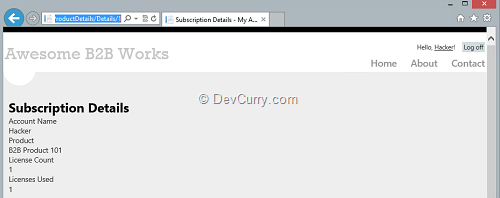
**What is Cross Site Request Forgery?**

Cross Site Request forgery is a type of a hack where the hacker exploits the trust of a website on the user. In other words, the site trusts the user (because they have authenticated themselves) and accepts data that turns out to be malicious.  
  
You can read more about CSRF on the [OWASP](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)) site.

**How does it work?**

A CSRF attack depends on the fact that the site trusts the user’s input. From here on the hacker attempts to get authenticated users to click on links that submit data without the user actually realizing. For example, say you are logged on to your bank that has the ability to transfer money from one account to another. The hacker somehow reverse engineers this form and sets up a duplicate form that submits transfer requests to their own account. (This is an overly simplistic scenario because most banks require you to register and ‘Transfer Account’ as a separate step). To see a sample of this you can watch Scott Hanselman and Phil Haack’s HaHa show from Mix 2010.  
  
Today, I’ll demonstrate it in a simplified licensing scenario. Let’s suppose we have a cloud hosted B2B solution that allows your users to purchase a license count ‘n’ for *n x*$*YY,YYYY*/-. Now obviously this is a costly license and say one of your customers is a bad guy. They initially bought 1 license but now their needs have increased and they need 5 more licenses. This is going to set them back by 5x$YY,YYYY/- which is a lot of money in these hard economic times. Instead, they employ a BlackHat (hacker) for $Y,YYY/-, give them access to the current account, and ask them to devise a hack to do a surreptitious modification of license count. Let’s see how this could be possible.

**Hacking your Web App**

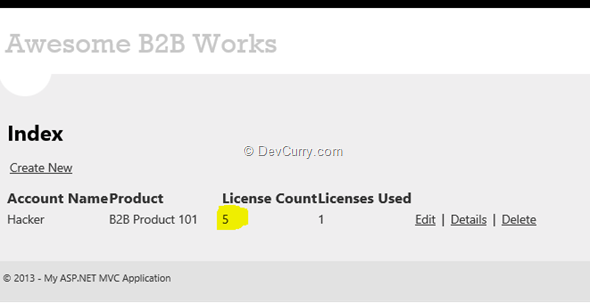
Let’s say you have a standard ASP.NET MVC site that uses Forms Authentication. Your users have a user name/password they log in, do their stuff and log out. While logged in, they can see their subscription via an Account Details page that looks as follows  
  
  
  
**Guessing Game Begins**  
  
Hacking involves a lot of things amongst which guessing the functionality plays a big role. Now the first thing the hacker will note is that the ProductDetails/Details/1 in the URL. They will guess that it’s probably an ASP.NET MVC site that has a ProductDetails/Edit/1 and ProductDetails/Create pages for their corresponding actions in the ProductDetails controller.   
  
So the next thing they will try is alter the URL to go to the Edit Page. This will not work and they will get re-directed to the login page. Why? That’s because we were careful enough to Authorize only users in the **Admins** group access to the Edit action method. We did this using the Authorize Attribute as follows.

[Authorize(Roles = "Admins")]   
public ActionResult Edit(int id = 0)   
{   
    ProductDetails productdetails = db.ProductDetails.Find(id);   
    if (productdetails == null)   
    {   
        return HttpNotFound();   
    }   
    return View(productdetails);   
}   
  
The Authorize attribute makes sure that only users in the **Admins** group can access the Edit action method. At the time of developmen,t this seemed like an adequate measure.   
  
To allow non admin users access to the Details method, we have allowed two groups ‘Admins’ and ‘Users’, by decorating the action method as follows.  
  
[Authorize(Roles="Admins,Users")]   
public ActionResult Details(int id = 0)   
{   
    ProductDetails productdetails = db.ProductDetails.Find(id);   
    if (productdetails == null)   
    {   
        return HttpNotFound();   
    }   
    return View(productdetails);   
}   
  
So because of the Authorize attributes, the hacker was thwarted in their first attempt.  
  
**Setting up the CSRF Hack**  
  
When the hacker tries to navigate to /ProductDetails/Edit, we redirect them to the login page. This gives them the hint that the page/action may exist. Based on that assumption, they setup a fake form with an increased number of licenses as follows  
  
<form name="csrfhackForm" method="post"   
    action="http://localhost:63577/ProductDetails/Edit/1">   
    <input type="hidden" name="Id" value="1" />   
    <input type="hidden" name="LicenseCount" value="5" />   
    <input type="hidden" name="LicenseUsedCount" value="1" />   
    <input type="hidden" name="AccountUserName" value="Hacker" />   
    <input type="hidden" name="ProductName" value="B2B Product 101" />   
</form>   
<script type="text/javascript">   
    document.csrfhackForm.submit();   
</script>   
  
The above form is malicious in the following ways

1. It submits to a URL that’s normally only available to Administrators.
2. Next it submits form data as soon as you load it
3. Redirects to Index page (because that’s the action on the Edit controller’s successful post method).
4. Posts data without leaving any trace of the hacker.

The Hacker now loads this form in a public website and a url is ready that will auto increment the License count.

**Authorized User – The missing Ingredient**

As we mentioned earlier, a CSRF hack depends on the website’s trust on the user. Thus the hack is all set up, but it has one thing missing. The hacker can’t do anything unless an actual Admin clicks on the link. Using various social engineering techniques, hacker somehow gets the link to the Admin, say via an email.  
  
**Last line of defense and the breach**  
  
Let’s say the admin receives the malicious link in an email and clicks on the link, the final line of defense is the login page. The admin will be redirected to the login page and after logging in, because the form submit got thwarted, it will just present the Admin the Edit page. This will confuse the Admin but the hacker’s intent is still not accomplished.  
  
However things become easier if the Admin clicked on the ‘Remember me’ checkbox to keep an authentication cookie alive or was logged in to the site in another browser. In that case, clicking on the link will silently increase the license without the admin realizing it. And if the system logs changes, it will always log the administrator as the person who made the change. ATTACK SUCCESSFUL!!!  
  


**Thwarting the Hacker**

Thankfully the solution to this hack is not far away or difficult to implement. This is based on the fact that the hacker created a third party form not hosted on the original site. Utilizing this information we can thwart the hacker. The easy way to do this is to use the ValidateAnitForgery token attribute in the ProductDetails post action method as follows  
  
[HttpPost]   
[Authorize(Roles = "Admins")]   
[ValidateAntiForgeryToken()]   
public ActionResult Edit(ProductDetails productdetails)   
{   
    if (ModelState.IsValid)   
    {   
        db.Entry(productdetails).State = EntityState.Modified;   
        db.SaveChanges();   
        return RedirectToAction("Index");   
    }   
    return View(productdetails);   
}   
  
To generate the AntiForgeryToken and the Cookie on the client side, we declare it as follows in the HTML form in the Edit.cshtml  
  
@using (Html.BeginForm()) {   
    @Html.ValidationSummary(true)   
    @Html.AntiForgeryToken()  
    <fieldset>   
        <legend>ProductDetails</legend>   
…   
  
This ensures that a form being posted to the server was actually generated by the same server. Thus fake forms that do not have the AntiForgeryToken from the correct server, gets rejected.  
  
  
  
As you can see, after adding the AntiForgeryToken, if you click on the malicious link, you get the above error instead.

**Conclusion - One down, many more to go**

Well with that we covered one attack vector for a CSRF hack attack. There are other vectors like if you get JSON data on HTTP GET requests, that data can potentially be tampered with, and another variation of the above attack where hacker guesses fields in POST and tries to alter data. We will cover some of these in future posts.  
  
Moral of the story is (and you will hear this multiple times in future) – You cannot trust ANY user input. Be suspicious of every type of data coming in to your server.

**Testing this for yourself**

The associated code sample of this article has two sites, WhatIsValidateAntiForgeryToken and DoingCsrfHack. To set it up, you need to register once and create an Administrator account. Then in the Database (that will be created automatically if you have a SQL Express installed at the default location), add a Role called ‘Admins’ in the webpages\_Roles table. In the webpages\_UsersInRoles table, put the ID of the Role and the Id of the User (from the UserProfile table) so that the user belongs to the ‘Admins’ group.  
  
Now set the WhatIsValidateAntiForgeryToken project as the default project and run the application. Log in as Administrator. Go to ProductDetails/Create and create a new Product Detail entry.  
In a new browser tab, navigate to the DoingCsrfHack site (get the Url from IIS Express in the system tray). Bingo, hack deployed.  
  
Now uncomment the AntiForgeryToken attribute and check again.