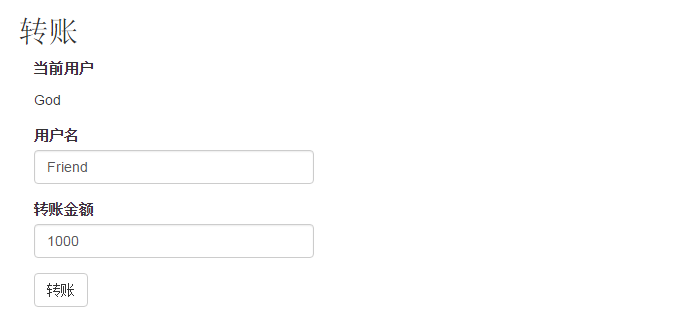
[Web安全相关（二）：跨站请求伪造（CSRF/XSRF）](http://www.cnblogs.com/Erik_Xu/p/5481441.html)

**简介**

　　CSRF（Cross-site request forgery跨站请求伪造，也被称为“One Click Attack”或者Session Riding，通常缩写为CSRF或者XSRF，是一种对网站的恶意利用。尽管听起来像跨站脚本（XSS），但它与XSS非常不同，并且攻击方式几乎相左。XSS利用站点内的信任用户，而CSRF则通过伪装来自受信任用户的请求来利用受信任的网站。与XSS攻击相比，CSRF攻击往往不大流行（因此对其进行防范的资源也相当稀少）和难以防范，所以被认为比XSS更具危险性。

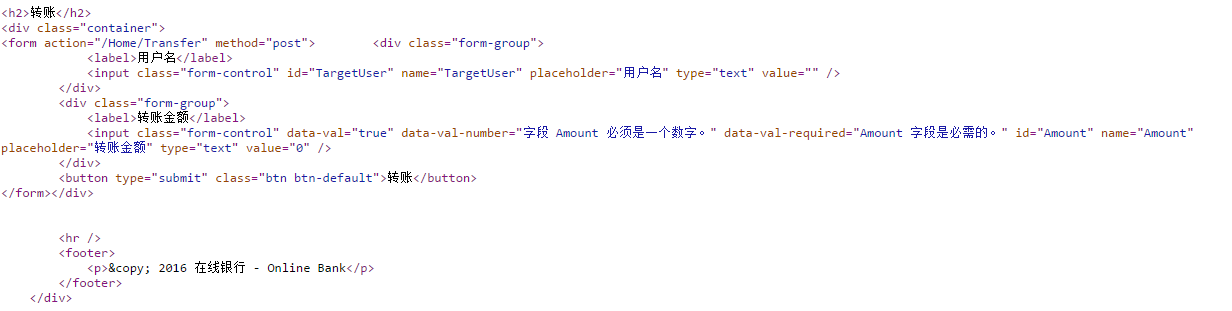
**场景**

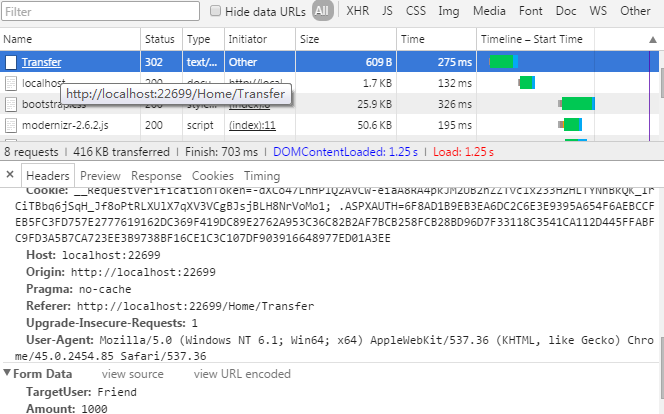
      某程序员大神God在某在线银行Online Bank给他的朋友Friend转账。





　　转账后，出于好奇，大神God查看了网站的源文件，以及捕获到转账的请求。





　　大神God发现，这个网站没有做防止CSRF的措施，而且他自己也有一个有一定访问量的网站，于是，他计划在自己的网站上内嵌一个隐藏的Iframe伪造请求（每10s发送一次），来等待鱼儿Fish上钩，给自己转账。

　　网站源码：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />

<title></title>

</head>

<body>

<div>

我是一个内容丰富的网站，你不会关闭我！

</div>

<iframe name="frame" src="invalid.html" sandbox="allow-same-origin allow-scripts allow-forms" style="display: none; width: 800px; height: 1000px;"> </iframe>

<script type="text/javascript">

setTimeout("self.location.reload();", 10000);

</script>

</body>

</html>

[复制代码](javascript:void(0);)

　   伪造请求源码：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

<html>

<head>

<title></title>

</head>

<body>

<form id="theForm" action="http://localhost:22699/Home/Transfer" method="post">

<input class="form-control" id="TargetUser" name="TargetUser" placeholder="用户名" type="text" value="God" />

<input class="form-control" id="Amount" name="Amount" placeholder="转账金额" type="text" value="100" />

</form>

<script type="text/javascript">

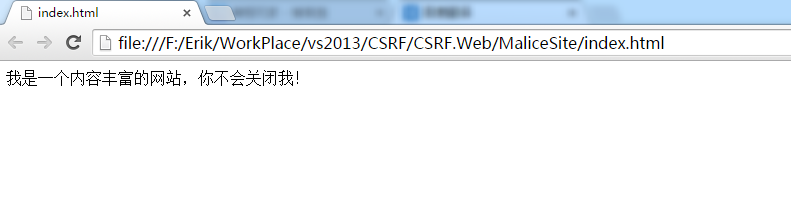
document.getElementById('theForm').submit();

</script>

</body>

</html>

[复制代码](javascript:void(0);)



　　鱼儿Fish打开了大神God的网站，在上面浏览丰富多彩的内容。此时伪造请求的结果是这样的（为了演示效果，去掉了隐藏）：



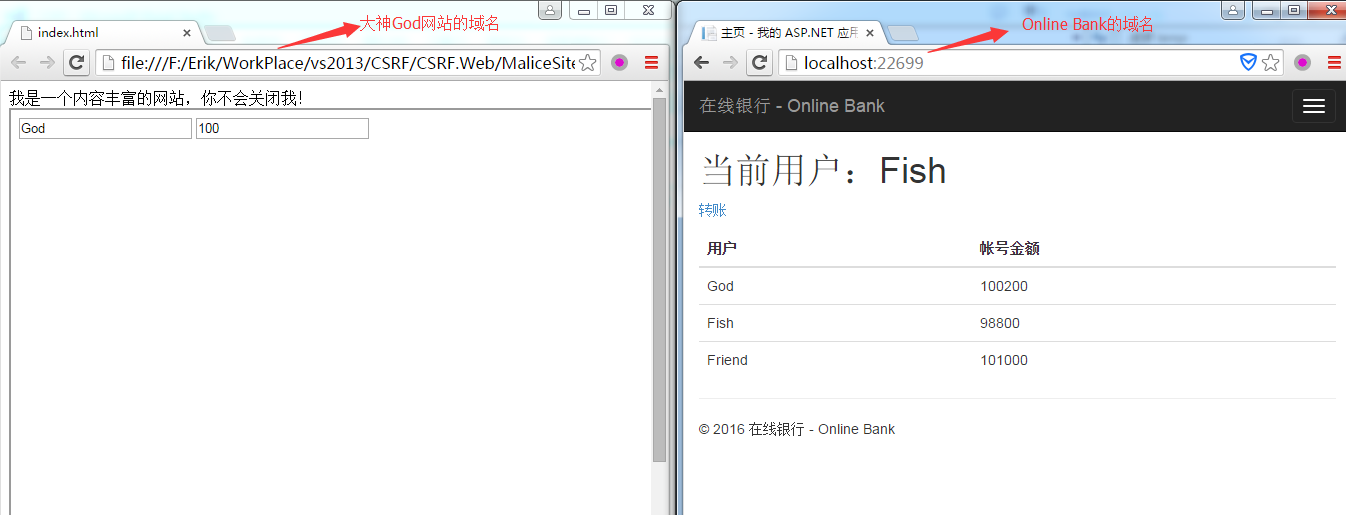
　　因为鱼儿Fish没有登陆，所以，伪造请求一直无法执行，一直跳转回登录页面。

　　然后鱼儿Fish想起了要登录在线银行Online Bank查询内容，于是他登录了Online Bank。

　　此时伪造请求的结果是这样的（为了演示效果，去掉了隐藏）：



　　鱼儿Fish每10秒会给大神God转账100元。

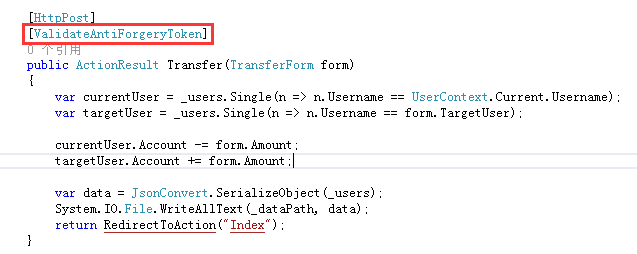


**防止CSRF**

　　CSRF能成功是因为同一个浏览器会共享Cookies，也就是说，通过权限认证和验证是无法防止CSRF的。那么应该怎样防止CSRF呢？其实防止CSRF的方法很简单，只要确保请求是自己的站点发出的就可以了。那怎么确保请求是发自于自己的站点呢？ASP.NET以Token的形式来判断请求。

　　我们需要在我们的页面生成一个Token，发请求的时候把Token带上。处理请求的时候需要验证Cookies+Token。



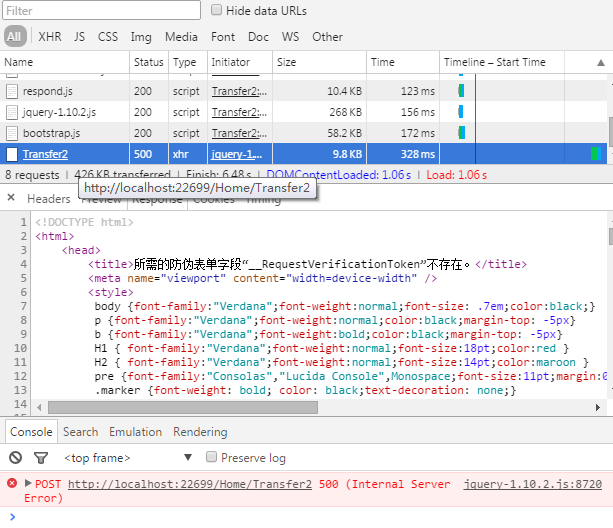


　　此时伪造请求的结果是这样的（为了演示效果，去掉了隐藏）：



**$.ajax**

　　如果我的请求不是通过Form提交，而是通过Ajax来提交，会怎样呢？结果是验证不通过。

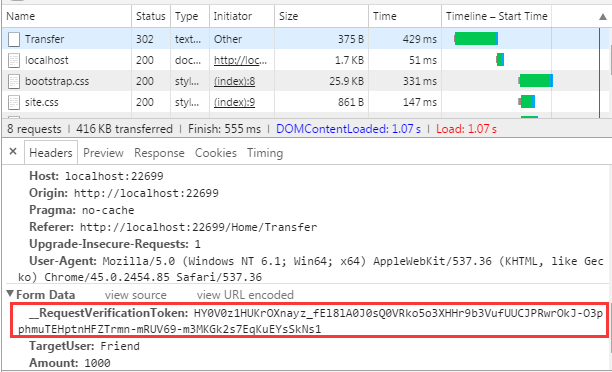


　　为什么会这样子？我们回头看看加了@Html.AntiForgeryToken()后页面和请求的变化。

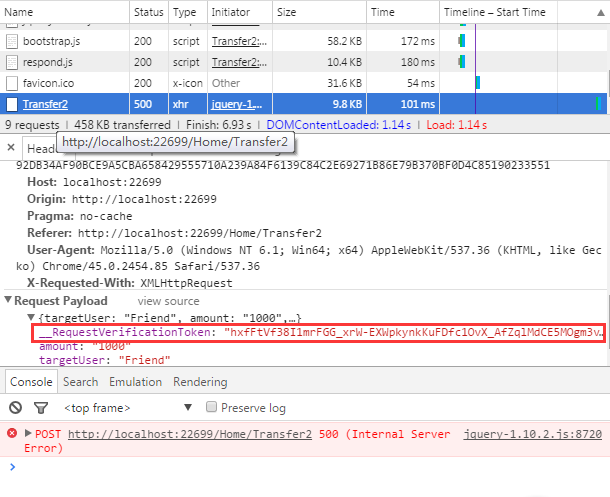
　　1. 页面多了一个隐藏域，name为\_\_RequestVerificationToken。



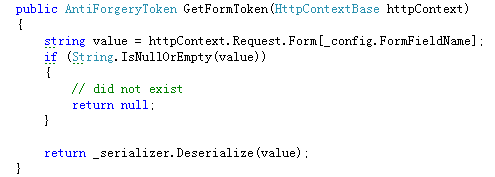
　　2. 请求中也多了一个字段\_\_RequestVerificationToken。



　　原来要加这么个字段，我也加一个不就可以了！



　　啊！为什么还是不行...逼我放大招，研究源码去！



　　噢！原来token要从Form里面取。但是ajax中，Form里面并没有东西。那token怎么办呢？我把token放到碗里，不对，是放到header里。

　　js代码：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

$(function () {

var token = $('@Html.AntiForgeryToken()').val();

$('#btnSubmit').click(function () {

var targetUser = $('#TargetUser').val();

var amount = $('#Amount').val();

var data = { 'targetUser': targetUser, 'amount': amount };

return $.ajax({

url: '@Url.Action("Transfer2", "Home")',

type: 'POST',

data: JSON.stringify(data),

contentType: 'application/json',

dataType: 'json',

traditional: 'true',

beforeSend: function (xhr) {

xhr.setRequestHeader('\_\_RequestVerificationToken', token);

},

success:function() {

window.location = '@Url.Action("Index", "Home")';

}

});

});

});

[复制代码](javascript:void(0);)

　　在服务端，参考ValidateAntiForgeryTokenAttribute，编写一个AjaxValidateAntiForgeryTokenAttribute：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]

public class AjaxValidateAntiForgeryTokenAttribute : FilterAttribute, IAuthorizationFilter

{

public void OnAuthorization(AuthorizationContext filterContext)

{

if (filterContext == null)

{

throw new ArgumentNullException("filterContext");

}

var request = filterContext.HttpContext.Request;

var antiForgeryCookie = request.Cookies[AntiForgeryConfig.CookieName];

var cookieValue = antiForgeryCookie != null ? antiForgeryCookie.Value : null;

var formToken = request.Headers["\_\_RequestVerificationToken"];

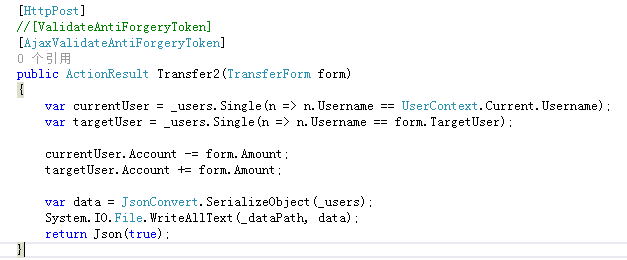
AntiForgery.Validate(cookieValue, formToken);

}

}

[复制代码](javascript:void(0);)

　　然后调用时把ValidateAntiForgeryToken替换成AjaxValidateAntiForgeryToken。



　　大功告成，好有成就感！

**全局处理**

　　如果所有的操作请求都要加一个ValidateAntiForgeryToken或者AjaxValidateAntiForgeryToken，不是挺麻烦吗？可以在某个地方统一处理吗？答案是阔仪的。

　　ValidateAntiForgeryTokenAttribute继承IAuthorizationFilter，那就在AuthorizeAttribute里做统一处理吧。

　　ExtendedAuthorizeAttribute：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

public class ExtendedAuthorizeAttribute : AuthorizeAttribute

{

public override void OnAuthorization(AuthorizationContext filterContext)

{

PreventCsrf(filterContext);

base.OnAuthorization(filterContext);

GenerateUserContext(filterContext);

}

/// <summary>

/// http://www.asp.net/mvc/overview/security/xsrfcsrf-prevention-in-aspnet-mvc-and-web-pages

/// </summary>

private static void PreventCsrf(AuthorizationContext filterContext)

{

var request = filterContext.HttpContext.Request;

if (request.HttpMethod.ToUpper() != "POST")

{

return;

}

var allowAnonymous = HasAttribute(filterContext, typeof(AllowAnonymousAttribute));

if (allowAnonymous)

{

return;

}

var bypass = HasAttribute(filterContext, typeof(BypassCsrfValidationAttribute));

if (bypass)

{

return;

}

if (filterContext.HttpContext.Request.IsAjaxRequest())

{

var antiForgeryCookie = request.Cookies[AntiForgeryConfig.CookieName];

var cookieValue = antiForgeryCookie != null ? antiForgeryCookie.Value : null;

var formToken = request.Headers["\_\_RequestVerificationToken"];

AntiForgery.Validate(cookieValue, formToken);

}

else

{

AntiForgery.Validate();

}

}

private static bool HasAttribute(AuthorizationContext filterContext, Type attributeType)

{

return filterContext.ActionDescriptor.IsDefined(attributeType, true) ||

filterContext.ActionDescriptor.ControllerDescriptor.IsDefined(attributeType, true);

}

private static void GenerateUserContext(AuthorizationContext filterContext)

{

var formsIdentity = filterContext.HttpContext.User.Identity as FormsIdentity;

if (formsIdentity == null || string.IsNullOrWhiteSpace(formsIdentity.Name))

{

UserContext.Current = null;

return;

}

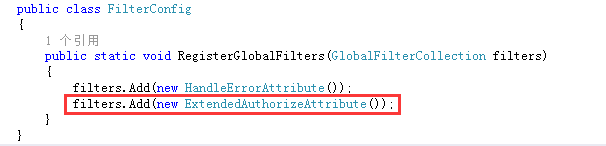
UserContext.Current = new WebUserContext(formsIdentity.Name);

}

}

[复制代码](javascript:void(0);)

　　然后在FilterConfig注册一下。



　　FAQ：

　　1. BypassCsrfValidationAttribute是什么鬼？不是有个AllowAnonymousAttribute吗？

　　如果有些操作你不需要做CSRF的处理，比如附件上传，你可以在对应的Controller或Action上添加BypassCsrfValidationAttribute。

　　AllowAnonymousAttribute不仅会绕过CSRF的处理，还会绕过认证和验证。BypassCsrfValidationAttribute绕过CSRF但不绕过认证和验证，

也就是BypassCsrfValidationAttribute作用于那些登录或授权后的Action。

　　2. 为什么只处理POST请求？

　　我开发的时候有一个原则，查询都用GET，操作用POST，而对于查询的请求没有必要做CSRF的处理。大家可以按自己的需要去安排！

　　3. 我做了全局处理，然后还在Controller或Action上加了ValidateAntiForgeryToken或者AjaxValidateAntiForgeryToken，会冲突吗？

　　不会冲突，只是验证会做两次。

**源码下载**

　　为了方便使用，我没有使用任何数据库，而是用了一个文件来存储数据。代码下载后可以直接运行，无需配置。

　　下载地址：<https://github.com/ErikXu/CSRF>