



ASP.NET Forms Authentication – Best Practices for Software Developers

By Rudolph Araujo, Foundstone Professional Services

August 2005

Background

ASP.NET does an excellent job of providing out of the box support for multiple forms of authentication using the classes in System.Web.Security namespace. In v1.1 of the framework, there exists support for forms-based, Microsoft Passport based and Integrated Windows (or NTLM) based authentication. These are intended to provide developers with easy access to an intuitive API which they can use to add authentications features to their own applications without having to reinvent it from scratch. As can be seen in listing 1 below, the code to leverage for instance forms-based authentication is fairly short and easily understood.

```
void btnLogin_Click(Object Source, EventArgs e)
{
    // Pull credentials from form fields and try to authenticate the user.
    if (FormsAuthentication.Authenticate(txtName.Text, txtPassword.Text))
    {
        // Redirect the client back to the originally requested resource and
        // create a new persistent cookie that identifies the user.
        FormsAuthentication.RedirectFromLoginPage(txtName.Text, true);
    }
}
```

Listing 1

The Authenticate function above validates the credentials submitted by the end user against the user data store – for instance the web.config’s authentication section to determine if the user should be logged in or not. If the credentials are indeed valid the RedirectFromLoginPage function sets an authentication ticket in the form of a cookie¹ before redirecting the user to whatever page they were trying to login to. This mechanism supports storing the credentials either in the clear or hashed using MD5 or SHA1.

Perhaps more often, web applications might use their own custom data store such as an LDAP based directory or SQL / XML database. In this case code like the fragment below could be used:

```
private bool Authenticate(string username, string password)
{
    // This method authenticates the user for the application.
    // In this demonstration application it always returns
    // true.

    return true;
}
```

¹ Cookieless authentication tickets can also be used wherein the ticket is embedded within the URL. These however are even more difficult to secure effectively and hence will not be discussed within this article.

```
void btnLogin_Click(Object Source, EventArgs e)
{
    string username = txtUserName.Text;
    string password = txtPassword.Text;
    bool isPersistent = chkBoxPersist.Checked;

    if(Authenticated(username,password))
    {
        //This could be used to store information such as what role
        // the logged on user belongs to
        string userData = "ApplicationSpecific data for this user.";

        FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(
            1,
            username,
            System.DateTime.Now,
            System.DateTime.Now.AddMinutes(30),
            isPersistent,
            userData,
            FormsAuthentication.FormsCookiePath);
        // Encrypt the ticket.
        string encTicket = FormsAuthentication.Encrypt(ticket);

        // Create the cookie.
        Response.Cookies.Add(new HttpCookie(FormsAuthentication.FormsCookieName,
encTicket));

        // Redirect back to original URL.
        Response.Redirect(FormsAuthentication.GetRedirectUrl(username, isPersistent));
    }
}
```

Listing 2

Alternatively, it is possible to avoid using the FormsAuthenticationTicket entirely by using the wrappers in the FormsAuthentication class such as SetAuthCookie and RedirectFromLoginPage.

The code in Listing 2 explicitly creates the authentication ticket and then cryptographically protects it to prevent tampering attacks. The types of cryptographic protection as well as other parameters are configured via the web.config file. Listing 3 describes all the settings as well as describes the values that those can hold.

```
<forms name="name"
  loginUrl="url"
  protection="All/None/Encryption/Validation"
  timeout="30"
  path="/"
  requireSSL="true/false"
  slidingExpiration="true/false">
  <credentials passwordFormat="format"/>
</forms>
```

Listing 3

A few of the attributes above merit further explanation. Firstly the *protection* attribute is used to cryptographically protect the authentication ticket. By setting this attribute to 'All' provides both confidentiality and integrity. The *requireSSL* attribute determines if the ticket is passed back and forth as a secure HTTP cookie i.e. this informs the browser not to transmit the cookie when the request is being made over vanilla HTTP as opposed to over HTTPS. Finally, *slidingExpiration* if set to true, resets an active authentication cookie's time to expiration upon each request during a single session. Thus, the authentication timeout specified in the *timeout* attribute is ignored in this case so long as the session is active. The default values for these and other security sensitive attributes in the different versions of the .NET framework as well as recommended values are provided later in this article.

While the timeout mechanism can be used to log the user out, developers do have the option to explicitly log the user out, for instance in response to the user clicking an logout button or link. This is achieved using the `SignOut` method on the `FormsAuthentication` class as shown below:

```
// The LogOffBtn_Click event handler uses the SignOut method
// to remove the authentication ticket from the users computer.
void LogOffBtn_Click(Object sender, EventArgs e)
{
    FormsAuthentication.SignOut();
    Status.Text = "You have been successfully logged out from the application.";
}
```

Listing 4

The Problem

While the `FormsAuthentication` class represents an excellent interface for developers providing commonly used functionality as part of the framework, it has what we believe to be a major design flaw. Like most cookie based authentication and session management mechanisms, this mechanism is susceptible to cookie replay and hence attacks such as session hijacking and stealing. The problem in this case is exacerbated due to the way the authentication ticket is created and maintained. When such a ticket is created it is only maintained in the cookie. No record or status about this ticket is maintained on the server side including information about expiry after a timeout. All such information is encoded into the ticket. Hence, so long as the user's ticket is valid i.e. the timeout has not expired the ticket can be stolen and misused. Even restarting Internet Information Services or for that matter the computer can not invalidate the ticket. This issue discovered by Foundstone and reported to Microsoft, has been acknowledged and published by Microsoft in KB Article 900111.

To compound matters calling the `SignOut` method simply clears the cookie from the client side – as can be seen in the code for the `SignOut` method shown in Listing 5 and obtained using .NET Reflector - but in no manner or means

invalidates it. Thus the developer can do absolutely nothing to effectively sign a user out and prevent their session from being misused after they have in fact “logged out”. This therefore represents a problem not only for the developer and team that created the application but also for the end user. Unfortunately the only effective way to effectively sign the user out seems to somehow cause the timeout to expire. This is explained in the recommendations section later in the article.

```
public static void SignOut()
{
    FormsAuthentication.Initialize();
    HttpContext context1 = HttpContext.Current;
    HttpCookie cookie1 = new HttpCookie(FormsAuthentication.FormsCookieName, "");
    cookie1.Path = FormsAuthentication._FormsCookiePath;
    cookie1.Expires = new DateTime(0x7cf, 10, 12);
    cookie1.Secure = FormsAuthentication._RequireSSL;
    context1.Response.Cookies.RemoveCookie(FormsAuthentication.FormsCookieName);
    context1.Response.Cookies.Add(cookie1);
}
```

Listing 5

Exploitation Scenarios

This bug can manifest itself and will be exploited in 3 basic classes of scenarios described in the Table 1.

| | |
|-----------------------------|--|
| Cross Site Scripting Vector | If an application using the forms authentication mechanism is vulnerable to cross-site scripting (XSS), a malicious attacker can use an injected script to steal the authentication ticket of a legitimate and currently logged in user, send that over to his own servers out on the Internet and then use those tickets to impersonate the user until the timeout expires. This is especially significant when the application is vulnerable to a persistent or stored XSS attack. |
| Network Sniffing Vector | If the vulnerable application does not use SSL for transport security, the cookie can simply be stolen while being transmitted over the network in the clear. Once the attacker has the ticket, they can as described above replay the ticket and URL as described above and steal the legitimate user’s session. |
| Shared Computer Scenario | If the vulnerable application allows creation of a persistent cookie, then this can be misused by other users of the same machine. The attacker can search the browser and/or cookie cache on the shared computer for the persisted authentication ticket and use it until the timeout expires. |

Table 1

Proof of Concept Application

To reproduce this problem as a proof of concept follow the steps outlined in Table 2 below to first build the application. Table 3 has information on running the application.

| |
|--|
| 1. Create a new ASP.NET web application in Visual Studio.NET 2003 |
| 2. Add the following to the web.config file to setup forms authentication: <pre> <authentication mode="Forms"> <forms name=".ASPXUSERDEMO" loginUrl="login.aspx" protection="All" timeout="60" /> </authentication> </pre> |
| 3. Rename the default WebForm1.aspx to default.aspx. This page represents the protected resource in this sample application – the page the user needs to login to view. |
| 4. Add a page called Login.aspx as shown below: <pre> <% @ Import Namespace="System.Web.Security " %> <HTML> <script language="C#" runat="server"> void Login_Click(Object sender, EventArgs E) { if (((UserEmail.Value == "joe") && (UserPass.Value == "joe")) ((UserEmail.Value == "admin") && (UserPass.Value == "admin"))) { FormsAuthentication.RedirectFromLoginPage(UserEmail.Value, PersistCookie.Checked); } else { Msg.Text = "Invalid Credentials: Please try again"; } } </script> <body> <form runat="server" ID="Form1"> <h3>Login Page</h3> <table> <tr> <td>Email:</td> <td><input id="UserEmail" type="text" runat="server" NAME="UserEmail"></td> <td><ASP:RequiredFieldValidator ControlToValidate="UserEmail" Display="Static" ErrorMessage="*" runat="server" ID="Requiredfieldvalidator1" /></td> </tr> <tr> <td>Password:</td> <td><input id="UserPass" type="password" runat="server" NAME="UserPass"></td> <td><ASP:RequiredFieldValidator ControlToValidate="UserPass" Display="Static" ErrorMessage="*" runat="server" ID="Requiredfieldvalidator2" /></td> </tr> <tr> <td>Persistent Cookie:</td> <td><ASP:CheckBox id="PersistCookie" runat="server" /> <td></td> </tr> </table> <asp:button text="Login" OnClick="Login_Click" runat="server" ID="Button1" /> </p> </pre> |

```
runat="server" />
    <asp:Label id="Msg" ForeColor="red" Font-Name="Verdana" Font-Size="10"
        />
    </form>
    </P>
</body>
</HTML>
```

5. Add a button called Signout to the default.aspx page and create the following event handler:

```
private void Signout_Click(object sender, System.EventArgs e)
{
    if(User.Identity.Name == "")
    {
        Response.Redirect("login.aspx");
    }
    else
    {
        FormsAuthentication.SignOut();
        Response.Redirect("login.aspx");
    }
}
```

6. Add a sensitive operation to the default.aspx page. In our sample we choose to read a file and set the value of a control using the text read from that file. This is illustrated below. If you do use this code ensure that the ASPNET/Network Service account has permissions on the file secret.txt. We also add a link to another "admin" page – UpdateTitle.aspx.

```
private void Page_Load(object sender, System.EventArgs e)
{
    if(User.Identity.Name == "")
    {
        btnSignIn.Text = "Signin";
    }

    if(User.Identity.Name != "admin")
    {
        hyperLinkUpdate.Visible = false;
    }

    string title = "Default";
    try
    {
        StreamReader sr = new StreamReader(Request.MapPath("secret.txt"));
        title = sr.ReadLine();
        sr.Close();
    }
    catch
    {
        title = "Exception";
    }

    Msg.Text = title;
}
```

7. Add a new page UpdateTitle.aspx and add a button called btnTitle and a textbox txtTitle. Add the following event handlers to complete the code for the proof of concept application.

```
private void Page_Load(object sender, System.EventArgs e)
```

```
{
  if(User.Identity.Name != "admin")
  {
    Response.Redirect("login.aspx");
  }
}

private void btnTitle_Click(object sender, System.EventArgs e)
{
  StreamWriter sw = new StreamWriter(Request.MapPath("secret.txt"));
  sw.WriteLine(txtTitle.Text);

  sw.Close();

  Response.Redirect("default.aspx");
}
```

Table 2

| |
|---|
| 1. Download a HTTP proxy such as Fiddler ² or Paros ³ . Configure and start the proxy. |
| 2. Browse to <a href="http://localhost/<vdir_name>/default.aspx">http://localhost/<vdir_name>/default.aspx |
| 3. Login using “admin”/”admin” |
| 4. Click the hyperlink to access the “administrative resource”. |
| 5. Add some text to the textbox and click the button to update the title. This will refresh the home page with the new title. |
| 6. Next click on the Signout button which in turn calls FormsAuthentication.Signout |
| 7. Go into Fiddler and find the second request to UpdateTitle.aspx. This request should have a form POST in the request body. |
| 8. Drag and drop this request into the “Request Builder” in Fiddler. |
| 9. Edit the value of the txtTitle textbox and then click execute in Fiddler when ready. |
| 10. Refresh the homepage and you should see the new title from Step 9 above. That title was saved in “secret.txt” in Step 9 and is then restored from the “secret.txt” file. This request succeeds even though the administrator is logged out. |

Table 3

Recommendations

For the most part developers can only partially mitigate the risks described above. The following are widely regarded as best practices:

1. Enable strong cryptographic protection on the forms authentication ticket. This should include both encryption and integrity support. Use SHA1 for HMAC generation and AES for encryption.
2. Ensure transport security is used to protect the authentication tickets while in transit. This should be done by enabling SSL on the web server and enforcing the same with the requireSSL attribute.

² <http://www.fiddlertool.com/fiddler/>

³ <http://www.parosproxy.org>

3. Minimize the lifetime of the ticket as far as possible. Set the timeout attribute to a small value and disable sliding expiration to ensure a fixed expiration period.

4. Protect authentication cookies from cross-site scripting enabled cookie stealing. This is done by both performing effective data validation and using the HttpOnly mechanism to protect the forms authentication cookie from client side attacks⁴.

5. Always use specific cookie names and paths to prevent their unnecessary transmission.

6. Possibly use an HTTP module to provide a server side backing store for authentication ticket that can be checked for expiry. This module can maintain a data structure that keeps track of all outstanding tickets, which user's they belong to and what their current status is i.e. if the user has logged out or the timeout expired etc.

Recommended Forms Authentication Settings

| Setting | ASP.NET 1.0 Default Value | ASP.NET 1.1 Default Value | ASP.NET 2.0 Default Value | Recommended Value | Justification |
|-------------------------|---------------------------|---------------------------|---------------------------|-----------------------------|---|
| CookieMode | Not supported | Not supported | UseCookies | UseCookies | Cookieless sessions have greater risk since they use the query string to pass the authentication ticket between the client and the server. Consequently all requests must use SSL and must not be cached. |
| EnableCrossAppRedirects | Not supported | Not supported | False | False | Prevent compromise in shared hosting type scenarios. Also ensure the IsolateApps flag is set to true for the machine key auto-generation mechanism. |
| FormsCookiePath | / | / | / | Protected Virtual Directory | Set the path to as specific a path as possible to prevent the cookie from being transmitted unnecessarily. Always partition your site into secure and non-secure folders that don't use HTTPS. |
| LoginUrl | N/A | N/A | N/A | Protected Virtual Directory | Ensure the URL for the login page is SSL protected. |

⁴ In ASP.NET 2.0 the forms authentication cookie if created through the framework is HttpOnly by default.

| | | | | | |
|-------------------|-------|-------|-------|--|--|
| RequireSSL | False | False | False | True | Prevent the forms authentication ticket from being transmitted in the clear. |
| SlidingExpiration | True | True | True | False | Set an absolute session expiry time ensuring that the time period is as small as possible. |
| Protection | All | All | All | All | Turns on both encryption and integrity checking on the forms authentication ticket. In addition ensure that the encryption and validation keys are auto-generated and IsolateApps is set to true. In a web farm scenario this is not possible and the keys must be hardcoded in the web.config. Ensure that the section of the web.config is then encrypted. |
| Timeout | 30 | 30 | 30 | 10-20 minutes or as small as possible. | Minimize the attack surface by decreasing the amount of time the ticket is vulnerable. |

Table 4

Conclusions

The biggest concern with the current implementation is that it provides no way for the developer (the user of the API) or the end-user to do the right thing and implement the correct security pattern, thus effectively significantly increasing the attack surface.

The analogy here is if you put your car key into the keyhole; closed the door and turned the key you would expect the door to be locked. If someone could now unlock your car by simply knowing what your key looked like you would say it were broken and would not rely on the security of the door.

While the risks associated with this issue cannot be completely eliminated, developers should mitigate risks by following the guidelines provided in this whitepaper.

References

1. KB 900111: A forms authentication cookie can be used to authenticate to a forms authentication ASP.NET application after the FormsAuthentication.SignOut method has been called

<http://support.microsoft.com/?kbid=900111>

2. How To: Protect Forms Authentication in ASP.NET 2.0

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/tmwa.asp>



About Foundstone Professional Services

Foundstone Professional Services, a division of McAfee, offers a unique combination of services and education to help organizations continuously and measurably protect the most important assets from the most critical threats. Through a strategic approach to security, Foundstone identifies, recommends, and implements the right balance of technology, people, and process to manage digital risk and leverage security investments more effectively.

Foundstone's Secure Software Security Initiative (S3i™) services help organizations design and engineer secure software. By building in security throughout the Software Development Lifecycle, organizations can significantly reduce their risk of malicious attacks and minimize costly remediation efforts. Services include:

- Source Code Audits
- Software Design and Architecture Reviews
- Threat Modeling
- Web Application Penetration Testing
- Software Security Metrics and Measurement

For more information about Foundstone S3i services, go to www.foundstone.com/s3i.

Foundstone S3i training is designed to teach programmers and application developers how to build secure software and to write secure code. Classes include:

- [Writing Secure Code – ASP.NET \(C#\)](#)
- [Building Secure Software](#)
- [Writing Secure Code – Java \(J2EE\)](#)
- [Ultimate Web Hacking](#)

For the latest course schedule, go to www.foundstone.com/education.