14

Security Hacks

When most people launch a new website, they are optimistic and look forward to successfully meeting the goals they originally envisioned. Unfortunately, in today's world the hopes of individuals and companies can be instantly dashed by someone with malicious intent. The reality is that part of building a website is considering security. The question to be asked is not "if" but "when" your site will be attacked. Everyone must include security as an integral part of Web application requirements if they are to achieve success.

This chapter includes some techniques to use to harden your system a little better. A couple of the hacks address a particularly insidious attack called the *SQL injection attack* — including the first hack in the following section, "Avoiding SQL Injection." This chapter includes a related hack that enables you to parameterize a SQL `IN` expression, which is another way to avoid SQL injection. You'll also find a couple of hacks dealing with *canonicalization attacks* — those involving URIs that try to bypass normal address and filename checking. For those of you who have OS resources, such as files on a system guarded by Windows authentication, this chapter includes an impersonation hack you can use. Last, but not least, there is a hack for extending the ASP.NET Login control to validate a strong password. Certainly, there is much more to cover in the world of security, but it is hoped that the hacks in this chapter give you some important reusable code and techniques and stimulate your own thinking about how to make your applications more secure.

Avoiding SQL Injection

One of the more serious threats you'll encounter when exposing a Web application on the Internet is the SQL injection attack. A hacker launches this type of attack by adding extra information to an input field. Your code then interprets this extra input as part of the SQL that it sends to the database.

For example, Figure 14-1 shows an input form in which a hacker is attempting an attack by inserting the following string:

```
', ''); drop table MyTable;--
```

The hacker is hoping that the code does something insecure, such as concatenate the input with the SQL statement to build the SQL string on the fly. In many situations, this is exactly what is happening in the code, and this section provides some advice about how to avoid it.

> **With SQL injection, it isn't too hard for attackers to quickly extract data and then figure out your entire database schema. To see how they do it, visit your favorite search engine, starting with the keywords "SQL injection."**

Figure 14-1

In Figure 14-1, the example has two buttons, Bad Add Shipper and Good Add Shipper. The hacker hopes the code like that shown in Listing 14-1 executes when he or she submits the page. The Bad Add Shipper button will run the code in Listing 14-1.

Listing 14-1: The wrong way to build an ad hoc SQL query

```
    protected void btnBadAddShipper_Click(object sender, EventArgs e)
    {
        string connStr = "Server=(local);Database=Northwind;Integrated
Security=SSPI";

        // this is *bad* because the user can
        // enter anything they want
        string cmdStr =
            "insert into Shippers (CompanyName, Phone) values ('" +
            txtCompanyName.Text + "', '" + txtPhone.Text + "')";

        using (SqlConnection conn = new SqlConnection(connStr))
        using (SqlCommand cmd = new SqlCommand(cmdStr, conn))
        {
            conn.Open();
            cmd.ExecuteNonQuery();
        }
    }
```

17_597663 ch14.qxp 4/25/06 9:58 PM Page 301

The code in Listing 14-1 is bad is because it concatenates the input with the rest of the SQL statement. This causes the resulting SQL command to be sent to the database as follows:

```
insert into Shippers (CompanyName, Phone) values ('', ''); drop table MyTable;--',
'')
```

The first part of this statement adds a new row with blank values to the Shippers table. The second part, after the first semicolon (;), removes the MyTable table from the database. The rest of the statement, after the second semicolon, is commented out to prevent errors.

As you can see, attackers can do a lot of damage to your system or view information they aren't supposed to see. Even worse, although it may not be totally obvious by this example, through SQL injection an attacker could potentially take over your entire system.

To fix this problem, you need to use parameters, as shown in Listing 14-2. When the user clicks the Good Add Shipper button, shown in Figure 14-1, it runs the `btnGoodAddShipper` method, shown in Listing 14-2.

Listing 14-2: The proper way to build an ad hoc SQL query

```
    protected void btnGoodAddShipper_Click(object sender, EventArgs e)
    {
        string connStr = "Server=(local);Database=Northwind;Integrated
Security=SSPI";

        // this is good because all input becomes a
        // parameter and not part of the SQL statement
        string cmdStr =
            "insert into Shippers (CompanyName, Phone) values (" +
            "@CompanyName, @Phone)";

        using (SqlConnection conn = new SqlConnection(connStr))
        using (SqlCommand cmd = new SqlCommand(cmdStr, conn))
        {
            // add parameters
            cmd.Parameters.AddWithValue("@CompanyName", txtCompanyName.Text);
            cmd.Parameters.AddWithValue("@Phone", txtPhone.Text);

            conn.Open();
            cmd.ExecuteNonQuery();
        }
    }
```

The code in Listing 14-2 is more secure because all input is treated as a parameter, rather than part of the SQL statement. Refer to the documentation on `SqlParameter`, or the corresponding type for whatever data provider you're using, for information on how to use parameters. An even more secure example would have instantiated a `SqlParameter` and explicitly set type, size, and other arguments to constrain the input.

ASP.NET v2.0 includes new datasource controls that enable you to add a bindable object to your Web form for populating data controls, such as GridView. These datasource controls accept parameters, meaning that they are also a safe way to get input from the user. For more information, see Chapter 8, "Extreme Data Binding."

In addition to increasing performance, stored procedures are more secure because they require passing input as parameters.

To re-create this scenario, add a test table to your database, such as the one shown in Listing 14-3, and add input as shown in Figure 14-1. Clicking the Bad Add Shipper button deletes the table. However, clicking the Good Add Shipper button processes the input properly.

Listing 14-3: Example table for demonstrating a SQL injection attack

```
create table MyTable
(
TempColumn      char(5)
)
```

Figure 14-2 shows the output in SQL Query Analyzer after running a couple of queries. Row 4 shows the results of running the insecure code from Listing 14-1 (clicking the Bad Add Shipper button). Row 5 shows the results of running the more secure code from Listing 14-2 (clicking the Good Add Shipper button). As you can see by looking at Row 5, processing input via parameters prevents interpretation of the input as part of the SQL statement and saves that input as column data.

```
select * from Shippers
```

	ShipperID	CompanyName	Phone
1	1	Speedy Express	(503) 555-9831
2	2	United Package	(503) 555-3199
3	3	Federal Shipping	(503) 555-9931
4	4		
5	5	','');drop table MyTable;--	1-800-555-GOOD

Figure 14-2

If you are coding ad hoc SQL statements, using parameters is one of the best techniques you can use to increase the security of your entire application.

Parameterizing an IN Expression

The last section discussed using parameters to secure your code with ad hoc queries. That works fine when you are doing a comparison in a `where` clause. However, it doesn't work at all if you want to pass a parameter to an `IN` expression. For example, the following does not work:

```
select EmployeeID, LastName from Employees
where EmployeeID in (@list)
```

One way to get around this limitation is to use a SQL function. An `IN` will accept a table. Therefore, you can call a function that parses each element in a comma-separated list, passed in the parameter, and build a table. The function's return value would be the table, which works fine with the `IN`. Listing 14-4 shows a function that accomplishes this.

Listing 14-4: A function that accepts a parameter and returns a table

```
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS OFF
GO
ALTER  FUNCTION dbo.ufStringToIntTable (@list varchar(8000))
      RETURNS @tbl TABLE (val int,seqnum int) AS
   BEGIN
      DECLARE @ix  int,
              @pos int,
              @seq int,
              @str varchar(8000),
              @num int

      SET @pos = 1
      SET @ix = 1
      SET @seq = 1

      WHILE @ix > 0
      BEGIN
         -- extract next parameter
         SET @ix = charindex(',', @list, @pos)
         IF @ix > 0
            SET @str = substring(@list, @pos, @ix - @pos)
         ELSE
            SET @str = substring(@list, @pos, len(@list))
         SET @str = ltrim(rtrim(@str))

         -- ensure valid number
         IF @str LIKE '%[0-9]%' AND
            (@str NOT LIKE '%[^0-9]%' OR
             @str LIKE '[-+]%' AND
              substring(@str, 2, len(@str)) NOT LIKE '[-+]%[^0-9]%')
         BEGIN
            -- convert and add number to table
            SET @num = convert(int, @str)
            INSERT @tbl (val,seqnum) VALUES(@num, @seq)
         END

         -- prepare for next parameter
         SET @pos = @ix + 1
         SET @seq = @Seq + 1
      END

      -- return table with all parameters
      RETURN
   END

GO
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO
```

The function in Listing 14-4 accepts a parameter and returns a table. The content of the input parameter is a comma-separated list of numbers. The function parses the values, converts each value to an integer, and adds each integer as a row into the table. This dynamically created table is then returned to the caller. Because an `IN` expression accepts a table, this works out well. Listing 14-5 shows how this function can be used to accept a parameter for an `IN` expression.

Listing 14-5: Using a function for an IN Expression

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="InParameter.aspx.cs"
Inherits="InParameter" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>IN Parameter</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <h1>View Employees</h1>
        <asp:Label ID="Label1" runat="server" Text="Employee IDs (comma separated):
"></asp:Label><br />
        <br />
        <asp:TextBox
            ID="txtEmployeeIDs" runat="server" Width="227px"></asp:TextBox><br />
        <br />
        <asp:Button ID="Button1" runat="server" Text="Update" /><br />
        <br />
        <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
DataKeyNames="EmployeeID"
            DataSourceID="SqlDataSource1">
            <Columns>
                <asp:BoundField DataField="EmployeeID" HeaderText="EmployeeID"
InsertVisible="False"
                    ReadOnly="True" SortExpression="EmployeeID" />
                <asp:BoundField DataField="LastName" HeaderText="LastName"
SortExpression="LastName" />
                <asp:BoundField DataField="FirstName" HeaderText="FirstName"
SortExpression="FirstName" />
            </Columns>
        </asp:GridView>
        <asp:SqlDataSource ID="SqlDataSource1" runat="server"
            ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
            SelectCommand="SELECT [EmployeeID], [LastName], [FirstName], [Photo]
FROM [Employees] WHERE ([EmployeeID] IN (select val from
ufStringToIntTable(@EmployeeID)))">
            <SelectParameters>
                <asp:ControlParameter ControlID="txtEmployeeIDs" DefaultValue="1"
Name="EmployeeID"
                    PropertyName="Text" Type="string" />
            </SelectParameters>
        </asp:SqlDataSource>
```

17_597663 ch14.qxp 4/25/06 9:58 PM Page 305


```
      </div>
      </form>
   </body>
   </html>
```

Listing 14-5 uses a SqlDataSource control to populate a GridView control. Notice the `IN` expression in the `SelectCommand` attribute. It contains `...WHERE ([EmployeeID] IN (select val from ufStringToIntTable(@EmployeeID)))`. `ufStringToIntTable` is the function from Listing 14-4. As shown in the `SelectParameters` element, the `@EmployeeID` parameter is populated from the `txtEmployeeIDs` TextBox control. You can test this by running the Web form in Listing 14-5 and adding a comma-separated string of numbers. The Northwind Employees table contains nine records, so you can select some, none, or all of them. Pressing the Update button causes a postback that repopulates the GridView control.

Protecting against Canonicalization Attacks

A canonicalization attack occurs when someone enters a filename requesting a file they aren't allowed to have or overwrites a file they shouldn't. Returning files that a user shouldn't have opens security holes because the file can contain sensitive information you don't want to expose. Allowing users to overwrite files causes a couple of problems. Perhaps they delete important information necessary for the operation of the site or the business. Another problem occurs when someone overwrites a file that is executable with a malicious file that can launch a virus.

The operating system tries to be user friendly and can resolve a filename, regardless of how you specify it. For example, the following four lines are equivalent:

```
type c:\log.txt

type \log.txt

type \..\log.txt

type c:\log.txt;;;
```

It is difficult to test for every case. Figure 14-3 shows a fictitious example of input that renames a file. Notice that the New Name field is set to `C:\\SomeData.xml`, which should never be allowed. Sure enough, it is possible to write code that is not secure enough to prevent this. Listing 14-6 shows what happens when you click the Bad Rename button.

Figure 14-3

Listing 14-6: The wrong way to handle filename input

```
protected void btnBadRename_Click(object sender, EventArgs e)
{
    // bad file handling - open to attack
    string appPath = Request.PhysicalApplicationPath;

    string oldPath = Path.Combine(appPath, txtOldName.Text);
    string newPath = Path.Combine(appPath, txtNewName.Text);

    File.Move(oldPath, newPath);
}
```

The problem with Listing 14-6 is that it grabs the input filename with no processing at all. The `File` class has no knowledge that it is an ASP.NET environment and you get no protection at all. Therefore, it does exactly as told and you'll have to hope that other security mechanisms, such as ACLs, help you out.

To work with filenames, most people use the `Request.MapPath` or `Server.MapPath` calls. Besides being a convenient way to get a full path to a file, the `MapPath` methods also help protect against canonicalization attacks. Listing 14-7 shows the secure way to work with filename input.

Listing 14-7: The proper way to handle filename input

```
protected void btnGoodRename_Click(object sender, EventArgs e)
{
    // good file handling - Server.MapPath
    // keeps files in application directory
    string oldPath = Server.MapPath(txtOldName.Text);
    string newPath = Server.MapPath(txtNewName.Text);

    File.Move(oldPath, newPath);
}
```

17_597663 ch14.qxp 4/25/06 9:58 PM Page 307

I know that most people use the MapPath methods all the time. However, some people just like to be different. Also, if you are calling a reusable library that handles files, it may not have security in mind. You should test the library with bad input to determine whether it is secure. If not, you can write your own routine or wrap the call in your own type that does proper validation on the input. Using Request.MapPath and Server.MapPath makes your ASP.NET application more resistant to canonicalization attacks.

If you're wrapping a third-party library to validate input with your own class, you won't have direct access to the Server property. However, you can still get to the MapPath method by calling HttpContext.Current.Server.MapPath();.

Using the New File Upload Control

ASP.NET 2.0 includes a new File Upload Web Server control. It works like the HTML File Upload control does except it is object-oriented with Web control properties and you no longer have to manually set the enctype attribute on the form element. What is more significant in terms of security is that you now have a FileName property on the ASP.NET File Upload control. The ASP.NET File Upload control still has the PostedFile property, but you don't need to use it for obtaining the filename anymore. Because the FileName property returns only the filename, and not the full path, there is less opportunity for mishandling the file and opening any security holes:

```
// the only way to get a file name from an HTML control
string htmlFilePath = fupHtmlUpload.PostedFile.FileName;

// still supported in ASP.NET Web control
string aspNetFilePath = fupAspNetUpload.PostedFile.FileName;

// new FileName property in ASP.NET Web control
string filePath = fupAspNetUpload.FileName;

// use it like this
string fileName = Path.Combine(Server.MapPath("."), filePath);
fupAspNetUpload.SaveAs(fileName);
```

This discussion assumes that you have weighed the benefits of allowing file uploads and have determined that it is a requirement. Remember that allowing file uploads is another vector that attackers can use to cause Denial-of-Service attacks on your site. You still need to be careful about file permissions you give the ASP .NET user. For example, if you are saving to a directory with a configuration file, the user could upload a file named web.config and overwrite yours. To stop this, put a deny write on the web.config ACL for the ASP.NET user (or the NETWORK SECURITY user on Windows Server 2003). For a more thorough security review, examine the identity that a user is operating to ensure secure settings.

Using Dynamic Impersonation Safely

Impersonation refers to the capability of code to run with the identity of a specific user. Typically, this is the logged on user, but it can also be a designated user (see the userName and password attributes of the identity element in web.config). You would use impersonation so that a user can access a

Windows operating system resource with specific permissions. This could be a file on the file system. This is one way to access that resource that would be impossible to access otherwise. Alternatively, you could expand the permissions on the resource, but that may not be good for security.

The `identity` element in `web.config` enables you to perform impersonation. When it is turned on, the application runs with the credentials of the currently logged on user. This setting applies to all files in the same directory as the `web.config` file. Because you don't want every logged on user to have access to protected resources, it is common to put the page that accesses the resource in a separate directory and add a `web.config` to that directory with impersonation turned on. Then configure security so only a certain user or role can access anything in that directory. After accessing a page in the subdirectory where impersonation is enabled, the user will run with impersonation and be able to access the resource.

When setting the identity element in `web.config`, a security hazard could exist whereby we can turn on impersonation for all files within the scope of that `web.config` file. This is convenient, as are many other things in ASP.NET. However, if you are following the security principle of least privilege, it may not be the most secure solution. In fact, it may open a security hole either now or in the future when more pages are added or when maintaining the site. Because the logged on user is impersonating during the entire time they have access to any pages in the same directory, they also have access to everything else to which the impersonated user has access. The most secure solution is to allow access to a resource for only the briefest amount of time possible and only when necessary.

I'm not advocating that you not use impersonation through `web.config` because if you need it you should use it. Conversely, if you want to restrict access to resources to only those who need it and only when they need it, then *dynamic impersonation* could be a good choice for you. Listing 14-8 shows how to do this.

Listing 14-8: Dynamic impersonation in code

```
protected void btnViewGrades_Click(object sender, EventArgs e)
{

    Response.Write(WindowsIdentity.GetCurrent().Name + "<br>");

    // impersonate current user
    WindowsImpersonationContext ctx =
        ((WindowsIdentity)User.Identity).Impersonate();

    Response.Write(WindowsIdentity.GetCurrent().Name + "<br>");

    try
    {
        DataSet ds = new DataSet();

        // throws exception when user doesn't have permission
        ds.ReadXml(Server.MapPath("Grades.xml"));

        GridView1.DataSource = ds.Tables[0];
        GridView1.DataBind();
    }
    catch (UnauthorizedAccessException)
    {
        lblResult.Text = "Not Authorized!";
```

17_597663 ch14.qxp 4/25/06 9:58 PM Page 309


```
            }

            // turn impersonation off
            ctx.Undo();

            Response.Write(WindowsIdentity.GetCurrent().Name + "<br>");
        }
```

Calling the `Impersonate` method of the current user's `WindowsIdentity` makes the program run with the permissions of the current user. The impersonation context is closed when calling `Undo` on the `WindowsImpersonationContext` object that was returned by the call to `Impersonate`. Therefore, all code between the call to `Impersonate` and the call to `Undo` in Listing 14-8 runs with the permissions of the caller. All code outside these bounds runs as the NETWORK SERVICE account (on Windows Server 2003) or the ASPNET account (all other OSs).

Listing 14-8 puts access to a file named `Grades.xml` within the block of code where we are impersonating. To demonstrate how this works, set ACLs to deny access to this file to everyone but a single person. Then log on with an account that doesn't have access to demonstrate that an `UnauthorizedAccess Exception` will be raised when you run the program. You can also prove that it works by logging on as the person who does have access and then running the program. Dynamic impersonation allows you to follow the principle of least privilege by limiting impersonation to a single block of code, only when necessary.

Validating a Strong Password in Login Controls

When using both the Create User Wizard and the Change Password controls, users are allowed to enter any password they want. This opens new security holes because too many users enter passwords that are common words. These passwords are easy for the user to remember, but they are also easy for a hacker to figure out. Because the Login controls don't check for weak password vulnerabilities, your site is at risk.

The hack in this section adds validation capabilities to passwords. We're simply going to add a `RegularExpressionValidator` control. To get started, perform the following actions:

1. Add a Create User Wizard to a Web form.
2. Drag a Regular Expression Validator to the right side of the same table cell as the Password TextBox.
3. Change the `RegularExpressionValidator ErrorMessage` property to `"Must have at least 1 number, 1 special character, and more than 6 characters."`
4. Change the `RegularExpressionValidator Text` property to `*`.
5. Change the `RegularExpressionValidator ControlToValidate` property to `Password`.
6. Change the `RegularExpressionValidator ValidationExpression` property to "`(?=^.{6,}$)(?=.*\d)(?=.*\W+)(?![.\n]).*$"`.

As its name suggests, a RegularExpressionValidator control uses what is called a *regular expression* to perform its validation. Regular expressions are a pattern matching language, which at first glance look cryptic and terse. However, once you know how to use them, you are very likely to find them fast and powerful. A good regular expressions site on the Web is `regexlib.com`.

The HTML for the cell where the `Password` is located should now look like this:

```
<td>
    <asp:TextBox ID="Password" runat="server"
        TextMode="Password"></asp:TextBox>
    <asp:RequiredFieldValidator ID="PasswordRequired" runat="server"
        ControlToValidate="Password"
        ErrorMessage="Password is required."
        ToolTip="Password is required."
        ValidationGroup="CreateUserWizard1">*</asp:RequiredFieldValidator>
    <asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server"
        ControlToValidate="Password"
        ErrorMessage="Must have at least 1 number, 1 special character, and more
than 6 characters."
        ValidationExpression=
"(?=^.{6,}$)(?=.*\d)(?=.*\W+)(?![.\n]).*$">*</asp:RegularExpressionValidator>
</td>
```

You can cut and paste the highlighted `RegularExpressionValidator` element from the preceding HTML into the Password cell of your own templated Create User Wizard. While you're at it, you can add a RegularExpressionValidator control to the Create User Wizard e-mail address, too. The RegularExpressionValidator `ValidationExpression` property already has a pop-up dialog in which a regular expression for e-mail is available for selection from a list of other regular expressions.

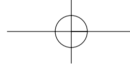
The same technique works for the Change Password control. After adding the Change Password control to your page, select Create Template, and use the same RegularExpressionValidator described above. The only difference will be that you should set the `ControlToValidate` property to `NewPassword`.

Wrapping Up

One of the best ways to protect a site against malicious users is to guard your code against SQL injection attacks. The technique is to simply parameterize your code. Using a SQL function allows you to pass a parameter to an `IN` expression.

Most code is safe from canonicalization attacks if you are using `Server.MapPath` or `Request.MapPath` API calls. It is when you use some of the file manipulation commands that you open potential security holes. The new File Upload Web control contains a `FileName` property that returns the name of the file without its full path. This could be made more secure by reducing the chance that the path could somehow be handled improperly, potentially opening a security hole.

17_597663 ch14.qxp 4/25/06 9:58 PM Page 311



The dynamic impersonation technique highlights the principle of least privilege by showing you how to limit resource exposure only in specific places in your code and only when it is needed. The alternative is using the `identity` attribute of the `web.config` file, which opens security holes by allowing everyone who has access to any page in a directory to have the same permissions associated with the user they are impersonating. The new ASP.NET Login controls make it easy to work with security, but they can be enhanced for even better security. You can add a `RequiredFieldValidator` component to their templates to further refine the validation you need to apply. For example, you can define a regular expression to ensure that users enter strong passwords in the Create User Wizard and Change Password controls.