Using MD5 to Encrypt Passwords in a Database

By Scott Mitchell

Introduction

There are, as you know, a plethora of Web sites on the Internet that allow for, if not require, some sort of user account. For example, on ASPMessageboard.com, in order to post a message one must create a user account, which includes information like username, email address, and password. To buy a book from Amazon.com you must create an account, which includes your name, an email address, a password, a shipping address, and so on. Note that each of these user accounts requires, among other things, a username and password pair, which are used to authenticate a user.

- continued -

If you've designed a site that required allowing user accounts you likely implemented the user account by creating a database table named something like `UserAccount`, with fields like `UserName` and `Password`. This table would contain a row for each user; when a user wished to log on, they'd submit their username and password, which would be used to query the `UserAccount` to see if there was a row whose `UserName` and `Password` fields matched the user-supplied username and password.

While this approach is pretty typical one of its downsides is that each user's password is stored in an unencrypted form in the database. That means that if someone compromises your database they have access to the passwords for every user. In this article we'll look at how to use MD5 to encrypt your passwords so that not even the database administrator can determine a user's password.

Is Your Data Safe?

Realize that the data in your database is not safe. Imagine that you host a database at a respectable Web hosting company that keeps the database patches up-to-date and has never had someone gain unauthorized access to the database before. While you may think that in such a situation your data is private from all others, understand that any database administrator who works at the Web hosting company will be able to view your data at will.

Before deciding to implement a password encryption scheme, as will be described in this article, one must ask themselves not how safe is their data, but how sensitive it is. That is, assume that others will see your information. If it is vitally important that this not incur, such a password encrypting scheme is ideal. For example, a financial institution should use encrypted passwords, so that there is not a risk of someone issuing an unauthorized transaction in your name. However, you may decide it not worth the hassle to implement password encryption for, say, a messageboard site, figuring that even if someone does obtain the password for every user, you can just reset everyone's password to some random value and have each person login with their new password and change it. In the former case much irreparable harm can be done; in the latter case, it may be an annoyance to deal with, but in the end does it really matter if your messageboard Web site incurs a security breach?

MD5 Encryption - A Brief Summary

There are two general classes of encryption: one-way encryption and two-way encryption. Two-way encryption is the most common form of encryption. It takes a plain-text input and encrypts it into some encrypted text. Then, at some later point in time, this encrypted text can be decrypted, which

results in the plain-text that was originally encrypted. Two-way encryption is useful for private communications. For example, imagine that you wanted to send an eCommerce Web site your credit card number to make a purchase. You wouldn't want to have your credit card numbers sent over the Internet in plain-text, because someone monitoring the Internet might see your credit card information whiz by. Rather, you'd want to send your credit card information as an encrypted message. When this encrypted message was received by the Web server, it could be decrypted, resulting in the actual credit card numbers.

One-way encryption, on the other hand, only allows for a plain-text input to be encrypted. That is, there is no way to decrypt the data. At first it may seem that such an encryption scheme is not needed - after all, why would you only want to be able to encrypt data and not decrypt it? A practical example of this is storing encrypted passwords on a database server, which is what this article is all about! That is, when a user creates a new account, he or she will supply their password. Rather than storing this password to the database as plain text, this password can be encrypted using a one-way encrypting algorithm and its encrypted form can be saved to the database. That way, if someone gains access to the database they will not see any of the passwords in plain-text.

MD5 encryption is an example of a one-way encryption algorithm; specifically, MD5 encryption maps a plain-text string of an arbitrary length to a small encrypted string of a fixed length. Two important properties of the MD5 algorithm are that two plain-text inputs cannot map to the same encrypted form, and that any given input always maps to the same encrypted value. The former property means that no two plain-text inputs will have the same encrypted value; the latter property means that if you wish to encrypt a particular plain-text input that it will always result in the same encrypted output.

The `MD5CyptoServiceProvider` class in the `System.Security.Cryptography` namespace of the .NET Framework provides a class for performing one-way, MD5 encryption. It is this class that we'll use to provide encrypted passwords in our database. Before we examine how to implement encrypted passwords, let's take a minute to investigate the functionality of the `MD5CyptoServiceProvider` class. The main method of this class is the `ComputeHash` method, which takes as input an array of bytes (the plain-text string to encrypt) and returns an array of bytes, which is the encrypted value. Commonly we'll want to encrypt a string, meaning that we must convert our string to an array of bytes in order to use the `ComputeHash` method. This conversion can be accomplished by using the `UTF8Encoding` encoding class, as shown in the following example:

```
'The string we wish to encrypt
Dim strPlainText as String = "Encrypt me!"

'The array of bytes that will contain the encrypted value of strPlainText
Dim hashedDataBytes as Byte()

'The encoder class used to convert strPlainText to an array of bytes
Dim encoder as New UTF8Encoding()

'Create an instance of the MD5CryptoServiceProvider class
Dim md5Hasher as New MD5CryptoServiceProvider()

'Call ComputeHash, passing in the plain-text string as an array of bytes
'The return value is the encrypted value, as an array of bytes
hashedDataBytes = md5Hasher.ComputeHash(encoder.GetBytes(strPlainText))
```

[View a Live Demo!]

Keep in mind that the `ComputeHash` method deals with arrays of bytes, not strings. Hence, to encrypt a plain-text string you must convert it to an array of bytes. This is accomplished by using the `UTF8Encoding` encoding class's `GetBytes` method (see the last line of the above code example).

The return result of the ComputeHash method is the encrypted data as an array of bytes. (For all practical purposes, the encrypted array has exactly 16 elements.)

Now that we've discussed the motivation behind using encrypted passwords and looked at the MD5 encryption algorithm, let's turn our attention to actually implementing encrypted passwords using MD5. We'll examine this, and more, in Part 2.

Using MD5 to Encrypt Passwords in a Database, Part 2

By Scott Mitchell

- Read Part 1

In Part 1 we looked briefly at the motivation behind using encrypted passwords and examined different classes of encryption algorithms, including MD5. In this part we'll examine how to implement encrypted passwords using MD5!

- continued -

Using MD5 To Store Encrypted Passwords

Earlier I mentioned that most sites that support user accounts do so by using a database table named something like UserAccount, with fields like UserName and Password. In the case where the password is saved as plain-text, both the UserName and Password fields are of type varchar. However, if we plan on using encrypted passwords we need to change the type of the Password field from a varchar to a binary type of length 16. This change is needed because the encrypted version of the user's password will be a 16-element array of bytes.

Now, whenever a user creates an account, we need to be certain to store the encrypted form of the password they selected into the database. The following ASP.NET code provides a sample Web page for creating a user account. It prompts the user for their username and password and stores these values into a UserAccount database table. However, instead of storing the password as-entered by the user, it first encrypts it using the MD5 code we examined earlier, and then saves to the database this encrypted version.

```
<%@ Import Namespace="System.Security.Cryptography" %>
<%@ Import Namespace="System.Text" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server" language="VB">
    Sub CreateAccount(sender as Object, e as EventArgs)
      '1. Create a connection
      Const strConnString as String = "connection string"
      Dim objConn as New SqlConnection(strConnString)

      '2. Create a command object for the query
      Dim strSQL as String = _
          "INSERT INTO UserAccount(Username,Password) " & _
          "VALUES(@Username, @Password)"
      Dim objCmd as New SqlCommand(strSQL, objConn)
```

	UserID	UserName	Password
1	1	Scott	0x3858F6223OAC3C915F300C664312C63F
2	2	John	0xAE2D699ACA20886F6BED96A0425C6168
3	4	Frank	0x059BF68F71C80FCE55214B411DD2280C
4	5	Ezra	0xFF8B689ED9E3D6E3A22BB0357384C770
5	6	Jisun	0x2B898208A66F66447F49FD0532C01AEB

```
        '3. Create parameters
        Dim paramUsername as SqlParameter
        paramUsername = New SqlParameter("@Username", SqlDbType.VarChar, 25)
        paramUsername.Value = txtUsername.Text
        objCmd.Parameters.Add(paramUsername)


        'Encrypt the password
        Dim md5Hasher as New MD5CryptoServiceProvider()

        Dim hashedBytes as Byte()
        Dim encoder as New UTF8Encoding()

        hashedBytes = md5Hasher.ComputeHash(encoder.GetBytes(txtPwd.Text))

        Dim paramPwd as SqlParameter
        paramPwd = New SqlParameter("@Password", SqlDbType.Binary, 16)
        paramPwd.Value = hashedBytes
        objCmd.Parameters.Add(paramPwd)


        'Insert the records into the database
        objConn.Open()
        objCmd.ExecuteNonQuery()
        objConn.Close()

        'Redirect user to confirmation page...
    End Sub
</script>

<form runat="server">
  <h1>Create an Account</h1>
  Username: <asp:TextBox runat="server" id="txtUsername" />
  <br />Password:
    <asp:TextBox runat="server" id="txtPwd" TextMode="Password" />
  <p><asp:Button runat="server" Text="Create Account"
       OnClick="CreateAccount" /></p>
</form>
```

Note that the above code sample imports a number of namespaces. These namespaces are imported to save typing (for example, if the `System.Security.Cryptography` namespace were not imported, when referring to the `MD5CryptoServiceProvider` class, the code would have to appear as: `System.Security.Cryptography.MD5CryptoServiceProvider`). The code provides the user with two TextBoxes, one for the username and one for the password. Once the user has supplied these values and clicked the "Create Account" button, the `CreateAccount` event handler will be executed, and the database table `UserAccounts` will have a new row added representing the new user.

The screenshot to the right shows the values in the `UserAccounts` table after some users have been created. Note that the password contains a 16-element binary array, representing the encrypted password. Clearly if someone were to be able to examine the `UserAccounts` table they could not deduce the plain-text password of any of the users.

Using MD5 To Authenticate a User

Since we are storing the passwords in encrypted form, and since, by the nature of a one-way encryption algorithm it is impossible to retrace from the encrypted form to the plain-text form, you may be wondering how in the world we'll authenticate a user. That is, when a user wants to login and supplies her username and password, how will we know if she provided the correct password?

Recall one of the properties of MD5 - that for any plain-text input the encrypted version of the input will be the same, always. That is, if we use MD5 to generate an encrypted form of the plain-text string "my password", the encrypted version of this will be the same today and forever more. Therefore, to authenticate a user we can simply take the password they provide, encrypt it using MD5, and then see if that encrypted form exists in the database (along with their username). The following code performs this check, logging in a user:

```
<%@ Import Namespace="System.Security.Cryptography" %>
<%@ Import Namespace="System.Text" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<script runat="server" language="VB">
    Sub Login(sender as Object, e as EventArgs)
      '1. Create a connection
      Const strConnString as String = "connection string"
      Dim objConn as New SqlConnection(strConnString)

      '2. Create a command object for the query
      Dim strSQL as String = "SELECT COUNT(*) FROM UserAccount " & _
                          "WHERE Username=@Username AND Password=@Password"
      Dim objCmd as New SqlCommand(strSQL, objConn)

      '3. Create parameters
      Dim paramUsername as SqlParameter
      paramUsername = New SqlParameter("@Username", SqlDbType.VarChar, 25)
      paramUsername.Value = txtUsername.Text
      objCmd.Parameters.Add(paramUsername)


      'Encrypt the password
      Dim md5Hasher as New MD5CryptoServiceProvider()

      Dim hashedDataBytes as Byte()
      Dim encoder as New UTF8Encoding()

      hashedDataBytes =
md5Hasher.ComputeHash(encoder.GetBytes(txtPwd.Text))

      Dim paramPwd as SqlParameter
      paramPwd = New SqlParameter("@Password", SqlDbType.Binary, 16)
      paramPwd.Value = hashedDataBytes
      objCmd.Parameters.Add(paramPwd)


      'Insert the records into the database
      objConn.Open()
      Dim iResults as Integer = objCmd.ExecuteScalar()
      objConn.Close()

      If iResults = 1 then
```

```
        'The user was found in the DB
      Else
        'The user was not found in the DB
      End If
    End Sub
</script>

<form runat="server">
  <h1>Login</h1>
  Username: <asp:TextBox runat="server" id="txtUsername" />
  <br />Password:
    <asp:TextBox runat="server" id="txtPwd" TextMode="Password" />
  <p><asp:Button runat="server" Text="Login" OnClick="Login" />
</form>
```

Limitations of Storing Encrypted Passwords in the Database
Before you decide whether or not to employ encrypted passwords in your next project, there are a few limitations to be aware of. First, realize that since the passwords are encrypted, there is no way to determine what a user's password is! While this is exactly what we were after by encrypting passwords in the first place, it means that you cannot provide users with a "Click here to have your password emailed to you" feature. Rather, if the user forgets his password he'll have to have his password reset to some random password, and then be emailed that new, random password. Essentially we cannot email the user his forgotten password because there's no way to determine what, exactly, his password is!

Also, converting from a plain-text password system to an encrypted system is possible, but can be a bit difficult. Essentially, you need to create a new table with the `Password` field being of type `binary` and of length 16. Next, you have to use an ASP.NET Web page or a .NET program to read the contents of the existing user database, and for each record, add it to the new table making sure to encrypt the user's password using MD5. Be careful not to delete the old user's account information until you're certain that you copied over their information correctly!

Important Security Note!

The hashing technique discussed in this article is susceptible to dictionary attacks. A much more secure approach is to *salt* the hash in some manner. For a thorough discussion on what salting is and why it's an important precaution, be sure to read: Could you Pass the Salt? Improving the Security in Encrypting Passwords using MD5.

Conclusion
In this article we looked at how to use MD5 to provide encrypted passwords in a database. The .NET Framework contains an `MD5CryptoServiceProvider` class that provides MD5 encryption functionality. Recall that MD5 is a one-way encrypting algorithm, meaning that it can be used to encrypt a plain-text string to an encrypted form, but not back the other way around. Encrypted passwords make sense for applications where one can do much damage by discovering a user's password, but due to their limitations, may not be suitable for less sensitive applications.

Happy Programming!

- By Scott Mitchell

Could you Pass the Salt? Improving the Security in Encrypting Passwords using MD5

By Scott Mitchell and Thomas Tomiczek

Introduction

A couple weeks ago Scott Mitchell wrote an article titled Using MD5 to Encrypt Passwords in a Database. In his article, Scott examined how to use the built-in ASP.NET `MD5CryptoServiceProvider` class to use MD5 hashing. To recap, MD5 is known as a one-way encryption algorithm. It is presented a plain-text string and then computes an encrypted version of that string. Given the encrypted version, it is computationally infeasible to determine the plain-text version. (If you haven't yet read Scott's article, I would encourage you to closely read the article before continuing here.)

- continued -

Specifically, in his article, Scott encourages the hashing of the user's password, and storing that encrypted value in the database. When a user wishes to login, then, they will enter their plain-text password, which will then be hashed and compared to the value of their hashed password in the database. The idea behind this is that if a insidious hacker does somehow manage to get access to the database, they won't be able to determine any of the user's passwords since they are stored in encrypted form.

The Perils of Scott's Approach

Unfortunately there is a security risk with Scott's approach. To see why, assume that a hacker has managed to break into the user database of a Web site with 10 million user accounts, and that the passwords in this database are protected by the same, simple MD5 hash of the password field as Scott proposed. Now, this hacker is interested in obtaining a valid password for any arbitrary user (not necessarily a specific user or specific set of users). The hacker can create a dictionary of common password entries, which he can then compute their hashed values using the MD5 algorithm. Next, the hacker can start going through the dictionary of common passwords in their hashed form, checking to see if any of the 10 million user accounts have a matching encrypted password field.

Since only the password field is hashed, the hacker can simultaneously check each of the 10 million user's passwords at once! For example, imagine that the word "foobar" was a common password, and that the hashed MD5 version of "foobar" was "#KH##NM@MM@M". The hacker could then run a SQL query like:

```
SELECT username
FROM UserTable
WHERE password = '#KH##NM@MM@M'
```

If *any* user has the password "foobar" the hacker would see their username. With this information Hence, by only hashing on the password field a hacker can attack 10 million encrypted passwords at once.

Take it With a Grain of Salt

This weakness of using hashes is well known, but can be easily solved by "salting" the password fields before hasing them. This is done by adding a second piece of information to the hash that is non-changing and unique for every user – like, for example, the username, or a user ID. Personally, when

developing a user database, I normally use a GUID as primary key for the table since it is random and big enough to be used as "salt input" into the hash.

Hopefully an example will clear up any confusion. In Scott's article from a few weeks back, when a user is creating a new user account their chosen password is hashed and the hashed version is stored in the database. In VB.NET, this code might look like:

```
Dim md5Hasher as New MD5CryptoServiceProvider()

Dim hashedBytes as Byte()
Dim encoder as New UTF8Encoding()

hashedBytes = md5Hasher.ComputeHash(encoder.GetBytes(txtPwd.Text))
```

Where `txtPwd.Text` is the value of the user's plain-text password and `hashedBytes` is the encrypted byte array. (This encrypted byte array would then be stored in the user's password field in the database.) My approach would include adding a salt to the value before computing its hashed value. For example, if we used the username, the last line of code in the above example would become:

```
hashedBytes = md5Hasher.ComputeHash(encoder.GetBytes(txtPwd.Text &
txtUsername.Text))
```

Where `txtUsername.Text` corresponds to the user's username. Say that this user, "user1" chose the popular password "foobar". Also imagine that "user2" choose the password "foobar". In Scott's original article both users would have the same value in the encrypted password field. However, by salting the value before hashing, "user1"'s password field will contain the hashed version of "foobaruser1", while "user2"'s password field will contain the hashed version of "foobaruser2" - these are different and will result in nonsimilar hashed values.

Naturally the code needed to authenticate a user would need to change. For example, when a user logs on they would supply their username and password. In Scott's original article, the code used hashed the user's entered password value and then examined if the hashed value matched up to the password field with the user's specified username. With salting, however, you would need to combine the user's username and password, then compute the hash value, and then determine if the particular user's password field matched up to this computed value.

Even if the hacker is privy to how we salt the hash values, he can only attack one user's password at a time. That is, say that he wants to check all users to see if any of them has the password "foobar". Rather than being able to issue one SQL query, he must issue a unique SQL query for each of the 10 million users.

Conclusion

In this article we examined Scott's earlier article titled Using MD5 to Encrypt Passwords in a Database. Specifically, we examined a major security issue inherent in Scott's approach, which was present due to the fact that Scott was not salting the password values before storing their hashed value. As we discussed, failure to salt a value prior to hashing allows for a hacker who has compromised the system to efficiently attack *all* user accounts. That is, the hacker can quickly determine if any user is using a particular password. For sites with many, many users, there is a good chance that there will be at least one user with a common password.

However, the problem becomes much more difficult for the hacker if we salt the password value with some known, non-changing value that is unique to every user, such as a username or user ID. Salting improves security because it denies the hacker the ability to attack all user accounts in one go.

Therefore, if you plan on encrypting your passwords in your database using a hashing encryption algorithm like MD5, be certain to salt the passwords prior to computing their hashed value!

- By Scott Mitchell and Thomas Tomiczek

About the Authors:
Thomas Tomiczek is Microsoft MVP for C# and Director of Technology of THONA Consulting Ltd., a UK based software developer with its main development offices in Berlin, Germany. His main area of expertise is the development of high performance distributed applications and database based systems, and he has spent a good part of the last year leading the development of a business object framework for the .NET environment that allows developers to work with automatically persisted business objects, instead of writing SQL.

For information about Scott Mitchell, please visit http://www.4guysfromrolla.com/ScottMitchell.shtml