

Password Hashing

If a person wants to gain unauthorised access to a system, the most useful bit of information they could obtain is the user's password. Unfortunately there are potentially many ways that this could happen.

A password may be known by (for example)...

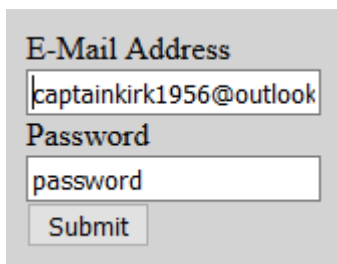
- Guessing – brute force attack
- Deduction – social engineering attack
- Stupidity – bad password choice of telling other people the password
- Shoulder surfing - watching a person type their password
- Intercepting the password between client and server
- Accessing the password by obtaining a copy of the database file
- Accessing the password by breaking in and stealing the entire server
- Accessing the password by bribing an administrator of the system

The Problem of Plain Text Passwords

Whilst some of the above points relate to educating the system users about password security others relate to storing passwords as plain text.

One really simple example of this is when the user types a password at login.

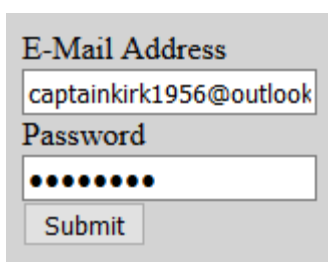
For example the following design would be problematic as anybody can see the password as it is typed.



A login form with a light gray background. It contains two text input fields and a submit button. The first field is labeled 'E-Mail Address' and contains the text 'captainkirk1956@outlook'. The second field is labeled 'Password' and contains the text 'password'. Below the password field is a button labeled 'Submit'.

To remove this problem it is typically the case that the password is masked in some way.

For example...



A login form with a light gray background, similar to the one above. It contains two text input fields and a submit button. The first field is labeled 'E-Mail Address' and contains the text 'captainkirk1956@outlook'. The second field is labeled 'Password' and contains ten black dots, indicating that the password is masked. Below the password field is a button labeled 'Submit'.

The problem also arises as the data is passed from the client browser to the server for processing (and back from server to client!) Using a tool such as Wireshark it is possible to intercept the data in transit obtaining something like the following...

```
c1qyP1f+zmN13HaNCmKPXwtS1MJv1pwqcQ4eK2p5Vt8UQ/14M1rAJ1tU9/GZzQKd%2B3n5H51o/1
IQkI15xZyfn1u8Xzwsvh5kPagHSUngdMgwQhAgrN8SxHOPE44gPXhzYNmv%2FwOMYQpFeB038pl
3PZm3b5AoMQr1VZxSEP4tcSCEXecVeHowjuLbQ3SCDNP25ngUP0zr7C%2BtGCiXzr%2BZAS0th
xtSearch=&txtEmail=captainkirk1956@outlook.com&txtPassword=password&btnLog:
' 63 Pm' -1Ü>E P[€ q YÃ·“ [D>Š ( (E (uY@ € [ Å" [Å" 63 Pm' -1Ü>™iP[€ q Y
cFo)/²:YÃ·- ♣:, [Q [Q@E [Q@üÿ [MÅ" ïÿÿúPy[1],H;NOTIFY * HTTP/1.1
239.255.255.250:1900
-CONTROL: max-age=60
ION: http://192.168.0.10:5200/Printer.xml
```

Using a secure connection via HTTPS / SSL it would be possible to encrypt the data in transit making it harder to read.

Finally once the password arrives at the data layer we also have a problem with plain text passwords.

If you look in the database for the DVDSwap Shop we can see that the password is being stored as plain text.

Users						
UserNo	FirstName	LastName	Email	UserPassword	Administrat	Add New Field
5	Fred	Smith	captainkirk1956@outlook.com	password	<input checked="" type="checkbox"/>	
*	(New)				<input type="checkbox"/>	

This renders the system vulnerable to the following attacks...

- Accessing the password by obtaining a copy of the database file
- Accessing the password by breaking in and stealing the entire server
- Accessing the password by bribing an administrator of the system

To get around this issue the password should never be stored as plain text on the server and must be encrypted in some way typically using a process called hashing.

Password Hashing

A step in the right direction is to encrypt the passwords however the problem here is that there is a possibility that an encrypted string could be decrypted. A better approach not just with passwords but other sensitive information e.g. credit card details is to create a hash value.

.NET Cryptography

.NET provides a name-space that handles (amongst other things) hashing of data using System.Security.Cryptography.

Hashing is different to encrypting because you cannot calculate the original value of the string from the hash value.

For example

password

might become...

5E884898DA28047151D0E56F8DC6292773603D0D6AABBDD62A11EF721D1542D8

Rather than storing passwords in plain text or encrypted form we would store the hash value.

When the user enters their password at a later date we take the original password string, recalculate the hash value and then compare this with the one in the database.

Adding Salt

So far with hashing our passwords there is still the problem of what happens if a brute force approach is applied to the data?

Once a hacker has access to the database file they simply apply the hashing algorithm to their own dictionary and eventually deduce the password by comparing results.

(This is not as easy as it sounds since the hacker will have to guess at which hashing algorithm you used to hash your data. Having said that if they have access to the server they probably also have access to your code!)

One way to make it harder to do this is to add salt to the hash.

Salt consists of extra data added to the password such that we are not just hashing the password. For example we could store the password + email address.

This also helps to reduce the problem of two users having the same password.

If the passwords for John and Fred without salt look like this...

John IKSv2XITzgf7LFJNFuHDkf9f4WQPZPLnEIY=

Fred IKSv2XITzgf7LFJNFuHDkf9f4WQPZPLnEIY=

We may assume that these two accounts share the same password giving us a clue as to how the data has been hashed.

Adding salt would change the hash values like so...

John 354rlrk8Jv7729qVOrOp0IXUv7RAsdV

Fred 9Wo0irC6+ylay0CJsLVtWBfbJBSn03j4gzhG

Even though they have the same password, with added salt this is now no longer obvious.

OK – Let's do this...

The first step of setting this up will be to create a new ASPX page called signup.aspx with an appropriate HTML form, something like this...

```

<%@ Page Language="C#" %>

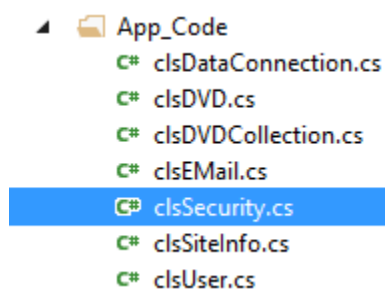
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Sign up</title>
</head>
<body>
    <h1>Sign Up</h1>
    <form method="post" action="SignUpProcessor.aspx">
        E-Mail Address
        <br />
        <input type="text" name="txtEMail" />
        <br />
        Password
        <br />
        <input type="password" name="txtPassword" />
        <br />
        Confirm Password
        <br />
        <input type="password" name="txtPasswordConfirm" />
        <input type="submit" value="Submit" />
    </form>
</body>
</html>

```

That gives us part of the presentation layer. The next step is to create the middle layer code.

You will need to create a new class called clsSecurity in the App_Code folder...



Once the class has been created you should have something like this...

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

/// <summary>
/// Summary description for clsSecurity
/// </summary>
public class clsSecurity
{
    public clsSecurity()
    {
        //
        // TODO: Add constructor logic here
        //
    }
}

```

Installing .NET Framework – Cryptography

Now that we have the class we need to instruct it to import the library for Cryptography.

One of the nice things about .NET is that it offers a huge number of off the shelf libraries providing us with extra functionality so we don't have to write it.

To add the Cryptography features add an additional “using” instruction to the top of the class like so...

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Security.Cryptography;

```

Now that we have the library available in the class we need to create two functions.

clsSecurity
+SignUp -GetHashString

(The + and – symbols indicate the SignUp is a public function whilst GetHashString is private.)

Make sure that you create the functions in the right place in relation to the brackets...

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Security.Cryptography;

/// <summary>
/// Summary description for clsSecurity
/// </summary>
public class clsSecurity
{
    public clsSecurity()
    {
        //
        // TODO: Add constructor logic here
        //
    }

    //create SignUp here

    //create GetHashString here
}

```

SignUp

SignUp looks like this...

```

public string SignUp(string EMail, string Password, string PasswordConfirm)
    //public method allowing the user to sign up for an account
{
    //var to store any errors
    string ErrorMsg="";
    //if the two passwords match
    if (Password == PasswordConfirm)
    {
        //get the hash of the plain text password
        string HashPassword = GetHashString(Password);
        //add the record to the database
        clsDataConnection DB = new clsDataConnection("select * from Users");
        DB.NewRecord["EMail"] = EMail;
        DB.NewRecord["UserPassword"] = HashPassword;
        DB.AddNewRecord();
        DB.SaveChanges();
    }
    //if the passwords do not match
    else
    {
        //generate an error message
        ErrorMsg = "The passwords do not match.";
    }
    //return the error message (if there is one)
    return ErrorMsg;
}

```

So what is going on?

The function accepts three parameters, Email, Password and PasswordConfirmation.

Email contains the new user's email address, password their password and password confirmation allows us to check if they have typed their password correctly.

The variable ErrorMsg allows us to record any errors that may come up and return them to flag problems.

The next thing the function does is check to make sure that password and password confirmation are the same. If not an error is returned, if everything is OK then the function continues.

```

    //get the hash of the plain text password
    string HashPassword = GetHashString(Password);

```

This line of code makes use of the function GetHashString (We will add that in a moment).

The function accepts a single parameter, in this case Password, it returns the hash string of the original plain text, in this case

5E884898DA28047151D0E56F8DC6292773603D0D6AABBDD62A11EF721D1542D8

```

clsDataConnection DB = new clsDataConnection("select * from Users");
DB.NewRecord["EMail"] = EMail;
DB.NewRecord["UserPassword"] = HashPassword;
DB.AddNewRecord();
DB.SaveChanges();

```

This code makes use of this program's `SqlConnection` to add the new record to the users table. It is worth pointing out that this data connection works in a different manner to the one that you will be used to. We shall explore the problems with this design when we look at SQL injection attacks.

GetHashString

`GetHashString` looks like this...

```
private string GetHashString(string SomeText)
{
    if (SomeText != "") //if there is text to process
    {
        //create an instance of the hash generator
        SHA256Managed HashGenerator = new SHA256Managed();
        //var to store the final hash
        string HashString;
        //array to store the bytes of the original text
        byte[] TextBytes;
        //array to store the bytes of the new hash
        byte[] HashBytes;
        //convert the text in the string to an array of bytes
        TextBytes = System.Text.Encoding.UTF8.GetBytes(SomeText);
        //generate the hash based on the array of bytes
        HashBytes = HashGenerator.ComputeHash(TextBytes);
        //generate the hash string replacing blank characters with -
        HashString = BitConverter.ToString(HashBytes).Replace("-", "");
        return HashString;
    }
    else //if there is nothing to process
    {
        //return a blank string
        return "";
    }
}
```

The first thing this code does is ask the question is there any data to process...

```
if (SomeText != "") //if there is text to process
{
```

If this is the case then it skips all processing and returns a blank string...

```
else //if there is nothing to process
{
    //return a blank string
    return "";
}
```

If however there is data to process then it gets on with the job...

It creates an instance of our hash generator from the `Cryptography .NET` library...

```
//create an instance of the hash generator
SHA256Managed HashGenerator = new SHA256Managed();
```


SHA-256 is one of a number of algorithms designed by the National Security Administration (NSA)

<https://en.wikipedia.org/wiki/SHA-2>

The function converts the string data into a series of bytes...

```
//convert the text in the string to an array of bytes
TextBytes = System.Text.Encoding.UTF8.GetBytes(SomeText);
```

It then uses the bytes to calculate the same bytes but this time as a hash...

```
//generate the hash based on the array of bytes
HashBytes = HashGenerator.ComputeHash(TextBytes);
```

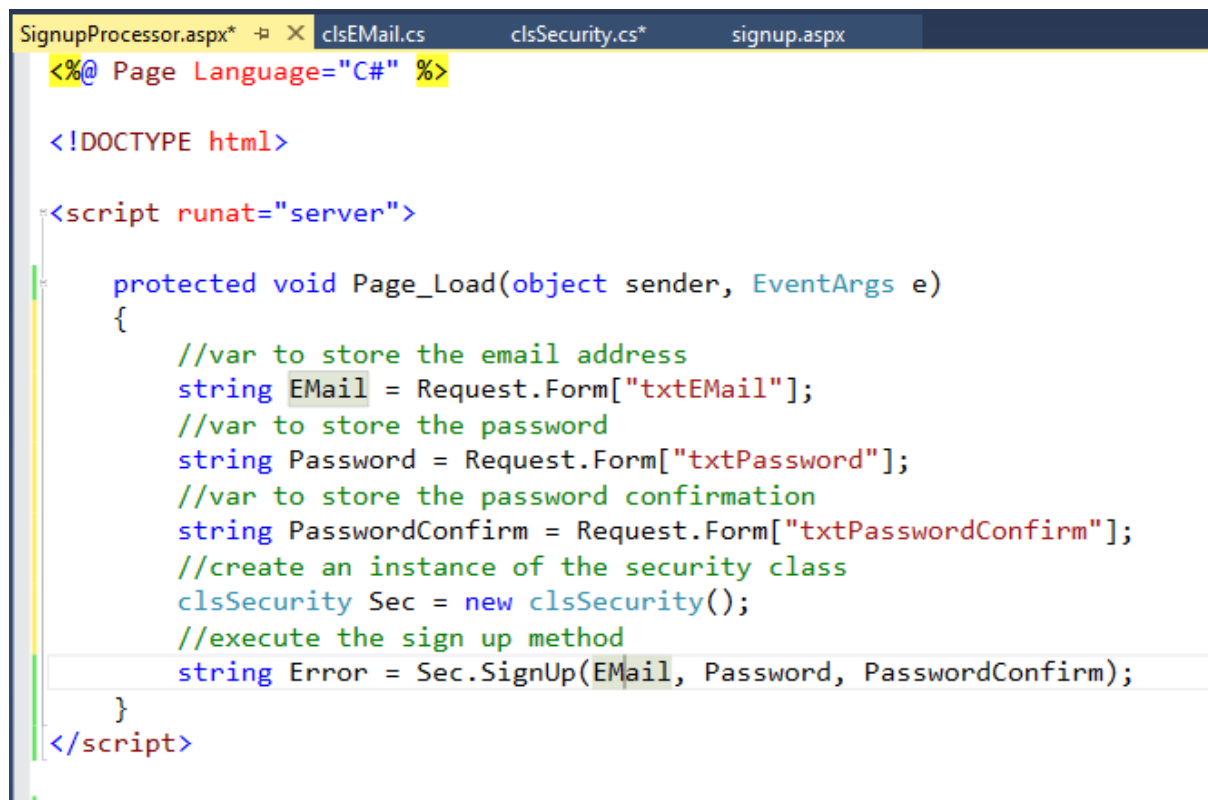
Finally it converts the hashed bytes to a string...

```
//generate the hash string replacing blank characters with -
HashString = BitConverter.ToString(HashBytes).Replace("-", "");
return HashString;
```

It is worth noting that you don't necessarily need to understand how this all works but know that you have some code now should you ever need to generate a hash value!

Completing the Presentation Layer

To link the presentation layer to the middle layer you will need to create a form processor, in this case called SignupProcessor.aspx which looks something like this...



```
<%@ Page Language="C#" %>

<!DOCTYPE html>

<script runat="server">

    protected void Page_Load(object sender, EventArgs e)
    {
        //var to store the email address
        string Email = Request.Form["txtEmail"];
        //var to store the password
        string Password = Request.Form["txtPassword"];
        //var to store the password confirmation
        string PasswordConfirm = Request.Form["txtPasswordConfirm"];
        //create an instance of the security class
        clsSecurity Sec = new clsSecurity();
        //execute the sign up method
        string Error = Sec.SignUp(Email, Password, PasswordConfirm);
    }
</script>
```

At this point place a break point in the form processor and using F10 / F11 make sure that the hash is being added to the database correctly.

If all is going to plan you should see something like the following added to the table Users...

Users					
UserNo	FirstName	LastName	EEmail	UserPassword	Admin
7	Fred	Smith	captainkirk1956@outlook.com	password	[
			mjdean@dmu.ac.uk	5E884898DA28047151D0E56F8DC6292773603D0D6AABDD62A11EF721D1542D8	[
*	(New)				[

In both cases the password is “password”, but one has the hash applied, the other does not.

Creating the Login Functionality

We now have a rudimentary sign up page; the next thing to do is create the code to make the login page work.

To do this we will modify clsSecurity to include the following function...

```
public Boolean Login(string EMail, string Password)
{
    //convert the plain text password to a hash code
    Password = GetHashString(Password);
    //find the record matching the users email address and password
    clsDataConnection UserAccount = new clsDataConnection("select * from Users where EMail='" + EMail + "' and UserPassword='" + Password + "'");
    //If there is only one record found then return true
    if (UserAccount.Count >= 1)
    {
        return true;
    }
    else //otherwise return false
    {
        return false;
    }
}
```

This code takes the password and then generates the hash value overwriting the plain text password.

```
//convert the plain text password to a hash code
Password = GetHashString(Password);
```

Remember also to place the code in the right place with reference to the brackets...

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Security.Cryptography;

/// <summary>
/// Summary description for clsSecurity
/// </summary>
public class clsSecurity
{
    public clsSecurity()
    {
        //
        // TODO: Add constructor logic here
        //
    }

    //create SignUp here

    //create GetHashString here
}

```

This hashed password combined with the user's email address is used to query the database to see if there is a record that matches this email password combination. There should only be a single record found.

```

//find the record matching the users email address and password
clsDataConnection UserAccount = new clsDataConnection("select * from Users where
EMail='" + EMail + "' and UserPassword='" + Password + "'");

```

The return value of the function is set depending on if a user with these credentials is found or not...

```

//If there is only one record found then return true
if (UserAccount.Count >= 1)
{
    return true;
}
else //otherwise return false
{
    return false;
}

```

The class now contains the following functions...

clsSecurity
+SignUp

-GetHashString
+Login

To add this functionality we will need to modify the form processor DefaultProcessor.aspx, like so...

```
<%@ Page Language="C#" %>
```

```
<!DOCTYPE html>
```

```
<script runat="server">
    protected void Page_Load(object sender, EventArgs e)
    {
        //create an instance of the security class
        clsSecurity Sec = new clsSecurity();
        //declare a variable to store the password from the form
        string Password;
        //var for email address
        string EMail;
        //request the contents of the text boxes from the form
        Password = Request.Form["txtPassword"];
        EMail= Request.Form["txtEmail"];
        if (Sec.Login(EMail, Password) == true)
        {
            Response.Write("Login successful");
        }
        else
        {
            Response.Write("Login failed");
        }
    }
</script>
```

Question?

What happens if you try to log in as “captainkirk1956@outlook.com” using the password “password”?

Even though you type the correct password at the interface, the login operation should fail.

The problem is that the password for that user is stored in the data base as plain text.

If you copy the password for the user you created above and replace the plain text password with the hash, both users will now have the same password...

Users						
UserNo	FirstName	LastName	EEmail	UserPassword	Administrat	Add New Field
5	Fred	Smith	captainkirk1956@outlook.com	5E884898DA28047151D0E56F8DC6292773603D0D6AAB8DD62A11EF721D1542D8	<input checked="" type="checkbox"/>	
7			mjdean@dmu.ac.uk	5E884898DA28047151D0E56F8DC6292773603D0D6AAB8DD62A11EF721D1542D8	<input type="checkbox"/>	
*	(New)				<input type="checkbox"/>	

This will fix the login problem, but it now creates a problem that even with the hash we can see that these two users have the same password!

Adding Salt

To fix this we need to add salt to the hash string that is generated.

One easy way of doing this is to concatenate the email address with the password to create a more unique original string.

In clsSecurity modify the code for SignUp like so...

```
public string SignUp(string EMail, string Password, string PasswordConfirm)
//public method allowing the user to sign up for an account
{
    //var to store any errors
    string ErrorMsg = "";
    //if the two passwords match
    if (Password == PasswordConfirm)
    {
        //get the hash of the plain text password
        string HashPassword = GetHashString(Password + EMail);
        //add the record to the database
        clsDataConnection DB = new clsDataConnection("select * from Users");
        DB.NewRecord["EMail"] = EMail;
        DB.NewRecord["UserPassword"] = HashPassword;
        DB.AddNewRecord();
        DB.SaveChanges();
    }
    //if the passwords do not match
    else
    {
        //generate an error message
        ErrorMsg = "The passwords do not match.";
    }
    //return the error message (if there is one)
    return ErrorMsg;
}
```

Also we need to make sure the salt is added to the Login function like so...

```

public Boolean Login(string EMail, string Password)
{
    //convert the plain text password to a hash code
    Password = GetHashString(Password + EMail);
    //find the record matching the users email address and password
    clsDataConnection UserAccount = new clsDataConnection("select * from Users where EMail="
    //If there is only one record found then return true
    if (UserAccount.Count >= 1)
    {
        return true;
    }
    else //otherwise return false
    {
        return false;
    }
}

```

To test this you will need to create another couple of new user accounts via the sign up process (make sure you remember the passwords)...

UserNo	FirstName	LastName	EEmail	UserPassword	Administrat	Add New Field
6	Fred	Smith	captainkirk1956@outlook.com	5E884898DA28047151D0E56F8DC6292773603D0D6AA8BD062A11EF721D1542D8	<input checked="" type="checkbox"/>	
7			mjdean@dmu.ac.uk	5E884898DA28047151D0E56F8DC6292773603D0D6AA8BD062A11EF721D1542D8	<input type="checkbox"/>	
8			somebod@bod.co.uk	C0C1664CC490D0FD9CE0BA81035B9199248AD16426301F03A70A4DECCDD5990	<input type="checkbox"/>	
9			fred@domain.com	92F45AC854246D5A272B8FDB620B48F32230D3FF509299A807BC62C7396BAFB7	<input type="checkbox"/>	
*	(New)				<input type="checkbox"/>	

In these examples all of the users have the same password of “password”.

The first two have no salt added, so it is obvious they are the same, the second have salt added so it isn’t possible to know this.

The problem is that having added the salt, the passwords on the first two users are not going to allow the user to login! You will need to fix this by deleting and re-creating the accounts.

Things to try...

Using the command Response.Redirect, modify the form processors DefaultProcessor.aspx and SignupProcessor.aspx such that they re-direct to an appropriate page.

To give you a clue SignupProcessor.aspx might do the following upon successful sign up...

```
protected void Page_Load(object sender, EventArgs e)
{
    //var to store the email address
    string EMail = Request.Form["txtEMail"];
    //var to store the password
    string Password = Request.Form["txtPassword"];
    //var to store the password confirmation
    string PasswordConfirm = Request.Form["txtPasswordConfirm"];
    //create an instance of the security class
    clsSecurity Sec = new clsSecurity();
    //execute the sign up method
    string Error = Sec.SignUp(EMail, Password, PasswordConfirm);
    //if there were no errors
    if (Error == "")
    {
        //redirect to sign up success
        Response.Redirect("SignUpSuccess.html");
    }
}
```

This would redirect to an HTML 5 page that looks like this...

You have been sucessfully signed up to the system

[Click here to go back to the main page](#)

Also have a think about what other methods might the security class include?

So far we have...

clsSecurity
+SignUp -GetHashString +Login

What other features do security systems off the user? Modify the class design diagram to indicate what would be appropriate. Clue – what if the user wants to change their password, or they have forgotten it?