PBKDF2

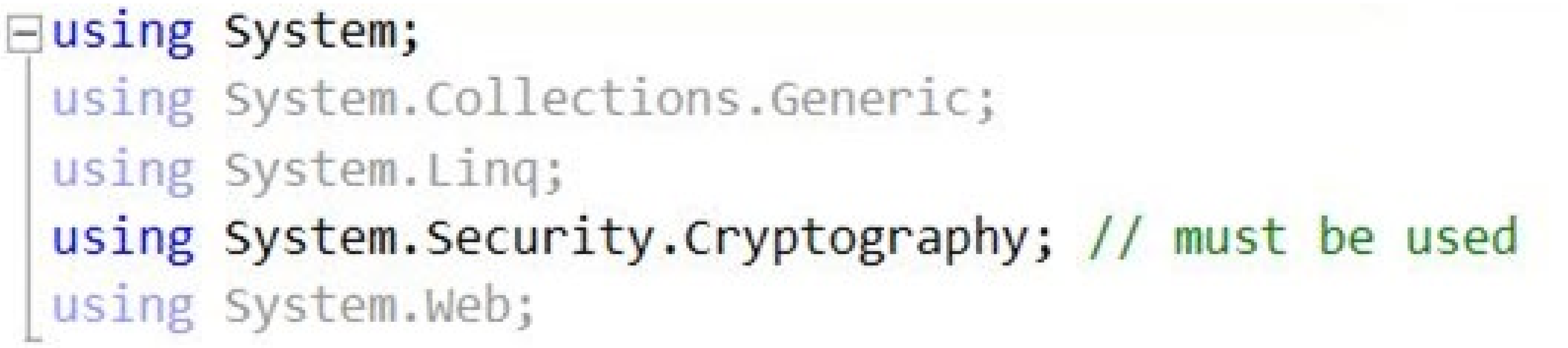
Password-Based Key Derivation Function 2

Password Based Key Derivation Function 2 (PBKDF2)

- This method of password hashing is **slow by design** (uses key stretching)
- Prevents against **brute force attacks** (using a GPU to guess different character combinations over and over until one works)
- MD5 is secure, but **designed to be fast** - susceptible to brute force attacks
- Large random "**salt**" values are created to make sure that each user's password is hashed uniquely (if 2 or more users have the same password, the computer won't be able to find the others because random characters have been added)
- With regular cryptographic hash functions (e.g. MD5, SHA256), an attacker can guess **billions** of passwords per second. With PBKDF2, the attacker can only make **thousands** of guesses per second (depending on the configuration).
- As always, security is most improved by users creating **strong** passwords and these processes are constantly **changing**!

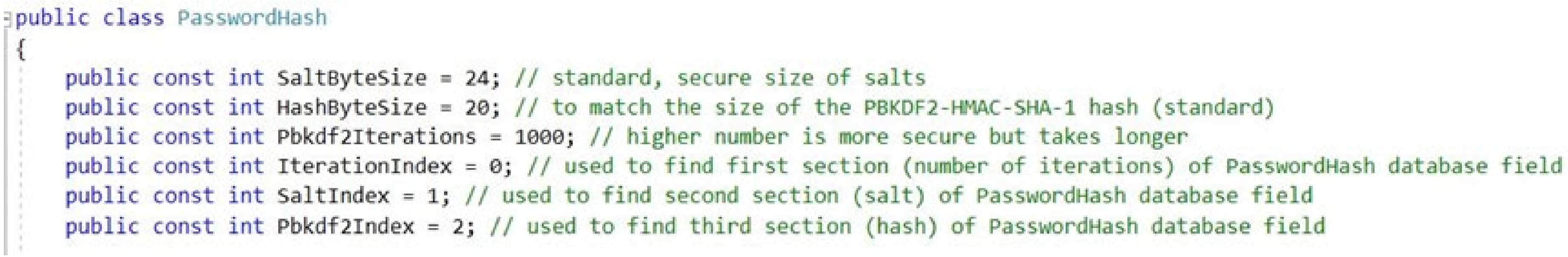
Set up

1. Using in class (PasswordHash.cs)



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography; // must be used
using System.Web;
```

2. Declare constants (PasswordHash.cs)

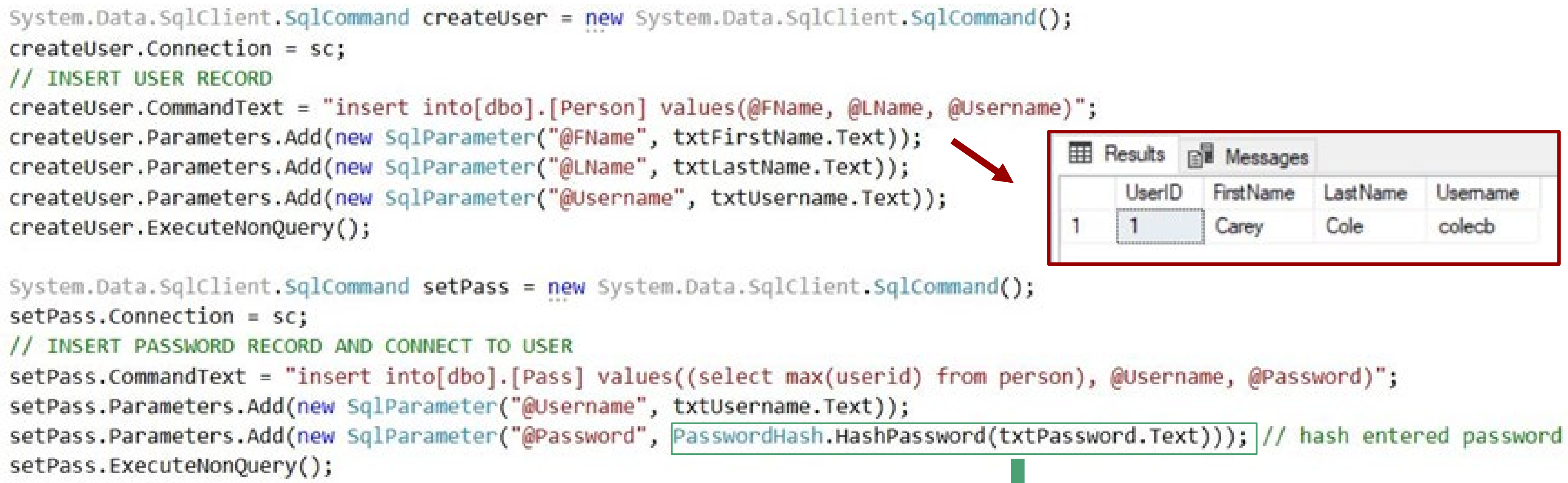


```
public class PasswordHash
{
    public const int SaltByteSize = 24; // standard, secure size of salts
    public const int HashByteSize = 20; // to match the size of the PBKDF2-HMAC-SHA-1 hash (standard)
    public const int Pbkdf2Iterations = 1000; // higher number is more secure but takes longer
    public const int IterationIndex = 0; // used to find first section (number of iterations) of PasswordHash database field
    public const int SaltIndex = 1; // used to find second section (salt) of PasswordHash database field
    public const int Pbkdf2Index = 2; // used to find third section (hash) of PasswordHash database field
```

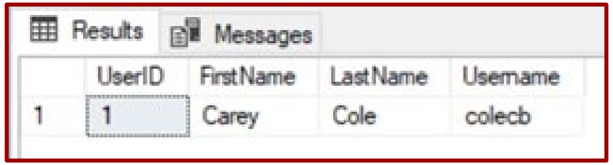
Create new user

Simple Example

1. Create new user (createUser.aspx.cs)

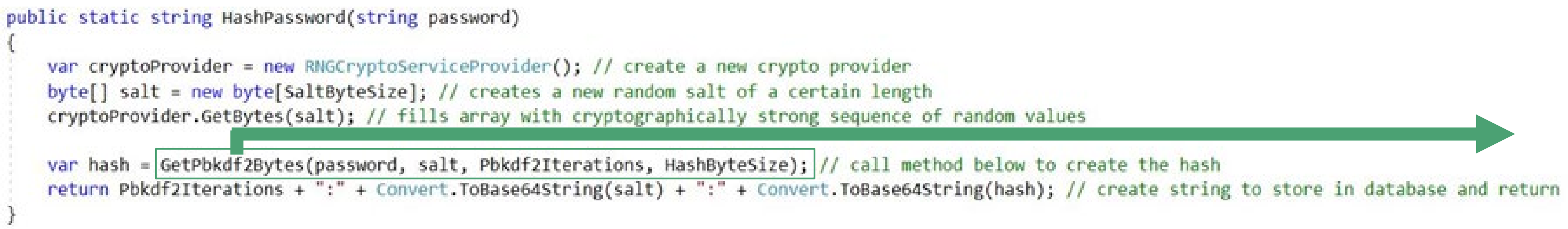


```
System.Data.SqlClient.SqlCommand createUser = new System.Data.SqlClient.SqlCommand();
createUser.Connection = sc;
// INSERT USER RECORD
createUser.CommandText = "insert into[dbo].[Person] values(@FName, @LName, @Username)";
createUser.Parameters.Add(new SqlParameter("@FName", txtFirstName.Text));
createUser.Parameters.Add(new SqlParameter("@LName", txtLastName.Text));
createUser.Parameters.Add(new SqlParameter("@Username", txtUsername.Text));
createUser.ExecuteNonQuery();

System.Data.SqlClient.SqlCommand setPass = new System.Data.SqlClient.SqlCommand();
setPass.Connection = sc;
// INSERT PASSWORD RECORD AND CONNECT TO USER
setPass.CommandText = "insert into[dbo].[Pass] values((select max(userid) from person), @Username, @Password)";
setPass.Parameters.Add(new SqlParameter("@Username", txtUsername.Text));
setPass.Parameters.Add(new SqlParameter("@Password", PasswordHash.HashPassword(txtPassword.Text))); // hash entered password
setPass.ExecuteNonQuery();
```



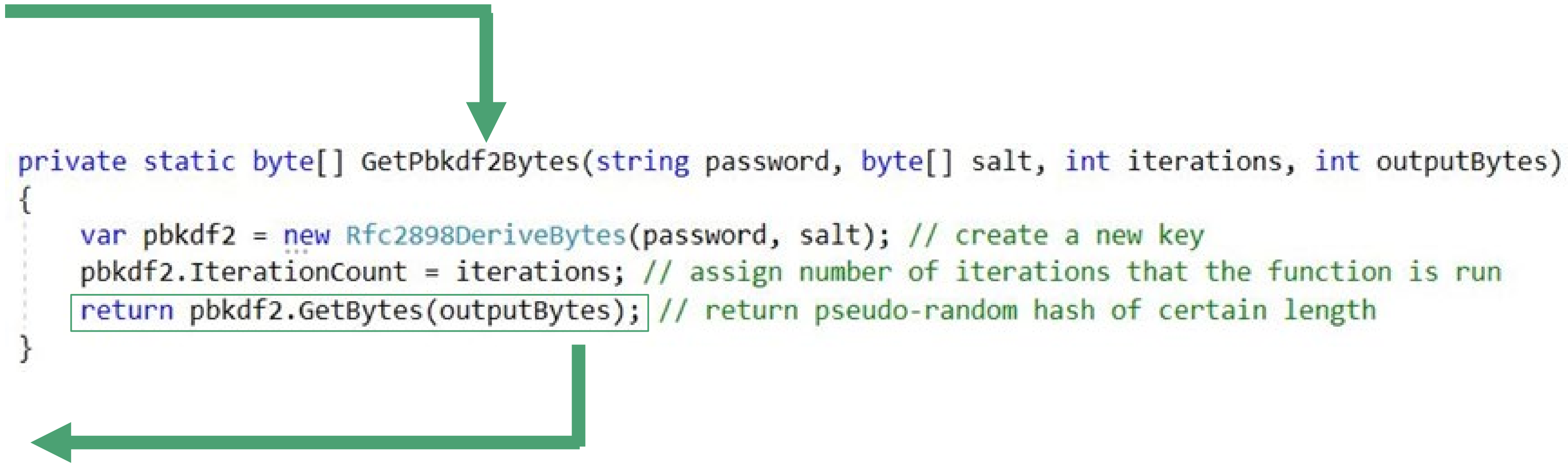
	UserID	FirstName	LastName	Username
1	1	Carey	Cole	colecb

2. Hash password (PasswordHash.cs)



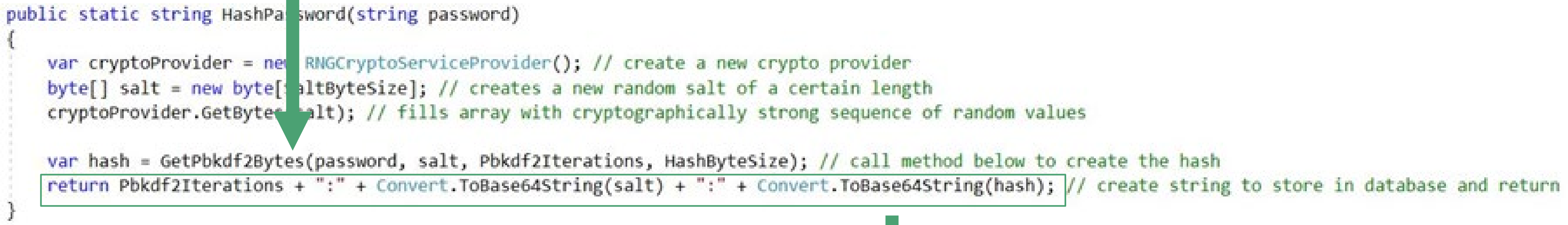
```
public static string HashPassword(string password)
{
    var cryptoProvider = new RNGCryptoServiceProvider(); // create a new crypto provider
    byte[] salt = new byte[SaltByteSize]; // creates a new random salt of a certain length
    cryptoProvider.GetBytes(salt); // fills array with cryptographically strong sequence of random values

    var hash = GetPbkdf2Bytes(password, salt, Pbkdf2Iterations, HashByteSize); // call method below to create the hash
    return Pbkdf2Iterations + ":" + Convert.ToBase64String(salt) + ":" + Convert.ToBase64String(hash); // create string to store in database and return
}
```

3. Create hash (PasswordHash.cs)



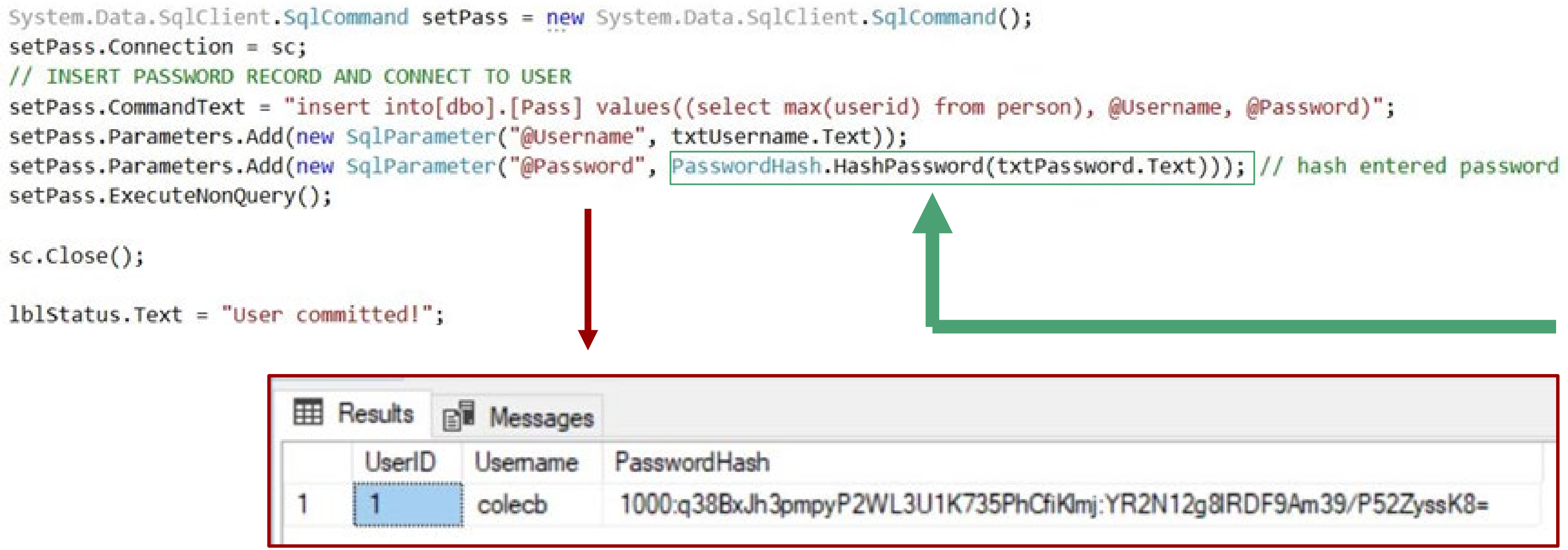
```
private static byte[] GetPbkdf2Bytes(string password, byte[] salt, int iterations, int outputBytes)
{
    var pbkdf2 = new Rfc2898DeriveBytes(password, salt); // create a new key
    pbkdf2.IterationCount = iterations; // assign number of iterations that the function is run
    return pbkdf2.GetBytes(outputBytes); // return pseudo-random hash of certain length
}
```

4. Finish hashing password (PasswordHash.cs)



```
public static string HashPassword(string password)
{
    var cryptoProvider = new RNGCryptoServiceProvider(); // create a new crypto provider
    byte[] salt = new byte[saltByteSize]; // creates a new random salt of a certain length
    cryptoProvider.GetBytes(salt); // fills array with cryptographically strong sequence of random values

    var hash = GetPbkdf2Bytes(password, salt, Pbkdf2Iterations, HashByteSize); // call method below to create the hash
    return Pbkdf2Iterations + ":" + Convert.ToBase64String(salt) + ":" + Convert.ToBase64String(hash); // create string to store in database and return
}
```

5. Finish creating new user (createUser.aspx.cs)



```
System.Data.SqlClient.SqlCommand setPass = new System.Data.SqlClient.SqlCommand();
setPass.Connection = sc;
// INSERT PASSWORD RECORD AND CONNECT TO USER
setPass.CommandText = "insert into[dbo].[Pass] values((select max(userid) from person), @Username, @Password)";
setPass.Parameters.Add(new SqlParameter("@Username", txtUsername.Text));
setPass.Parameters.Add(new SqlParameter("@Password", PasswordHash.HashPassword(txtPassword.Text))); // hash entered password
setPass.ExecuteNonQuery();

sc.Close();

lblStatus.Text = "User committed!";
```

Results Messages

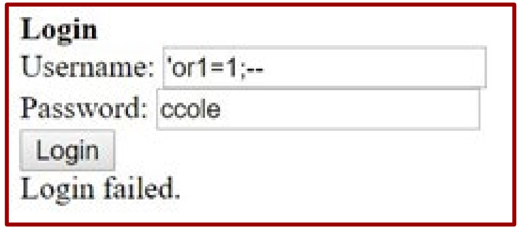
	UserID	Username	PasswordHash
1	1	colecb	1000:q38BxJh3pmpyP2WL3U1K735PhCfiKmj:YR2N12g8IRDF9Am39/P52ZyssK8=

Login

Simple Example

1. Login (userLogin.aspx.cs)

Prevents
SQL injection

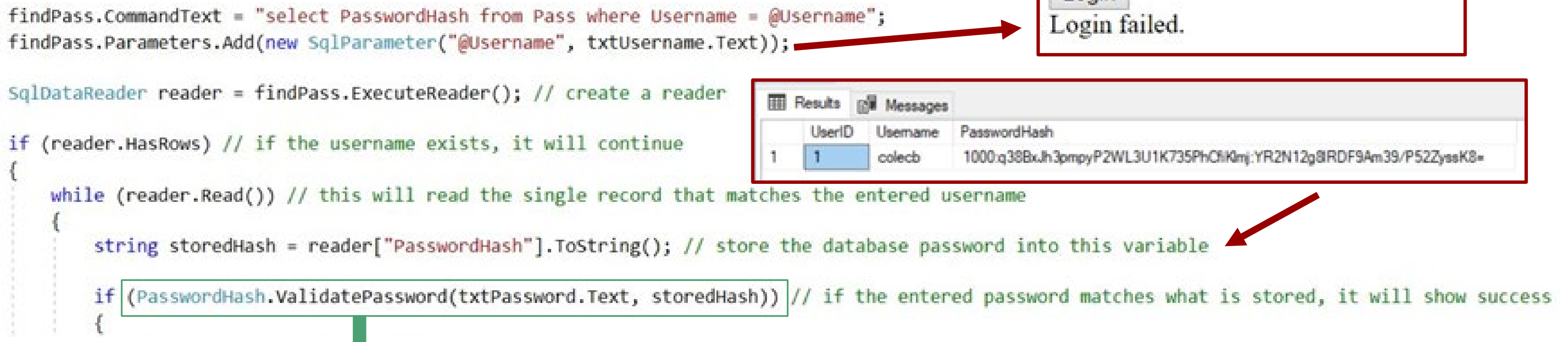


Login

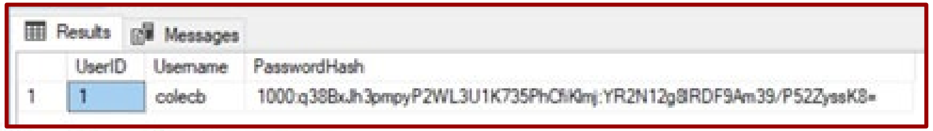
Username:	'or1=1;--
Password:	ccole

Login

Login failed.

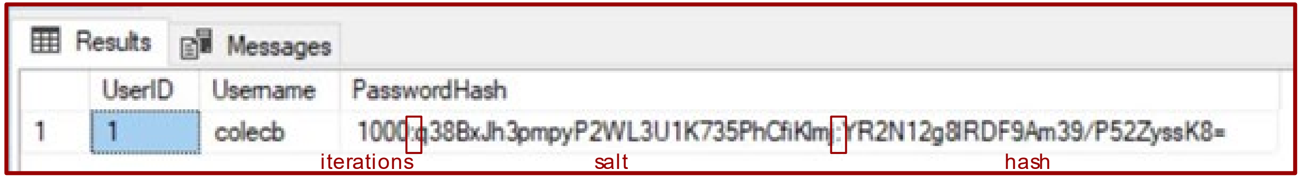


```
findPass.CommandText = "select PasswordHash from Pass where Username = @Username";
findPass.Parameters.Add(new SqlParameter("@Username", txtUsername.Text));

SqlDataReader reader = findPass.ExecuteReader(); // create a reader

if (reader.HasRows) // if the username exists, it will continue
{
    while (reader.Read()) // this will read the single record that matches the entered username
    {
        string storedHash = reader["PasswordHash"].ToString(); // store the database password into this variable

        if (PasswordHash.ValidatePassword(txtPassword.Text, storedHash)) // if the entered password matches what is stored, it will show success
        {
```

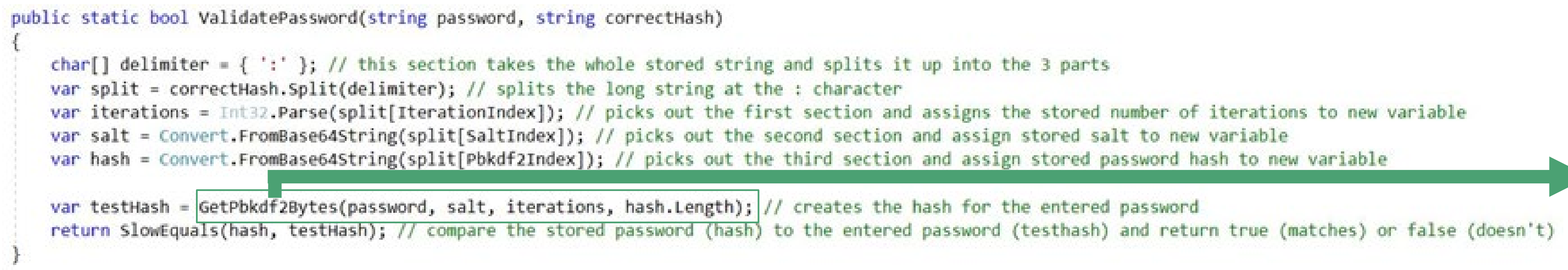


	UserID	Username	PasswordHash
1	1	colecb	1000:q38BxJh3pmpyP2WL3U1K735PhCfiKlmj:YR2N12g8RDF9Am39/P52ZyssK8=

2. Validate password (HashPassword.cs)

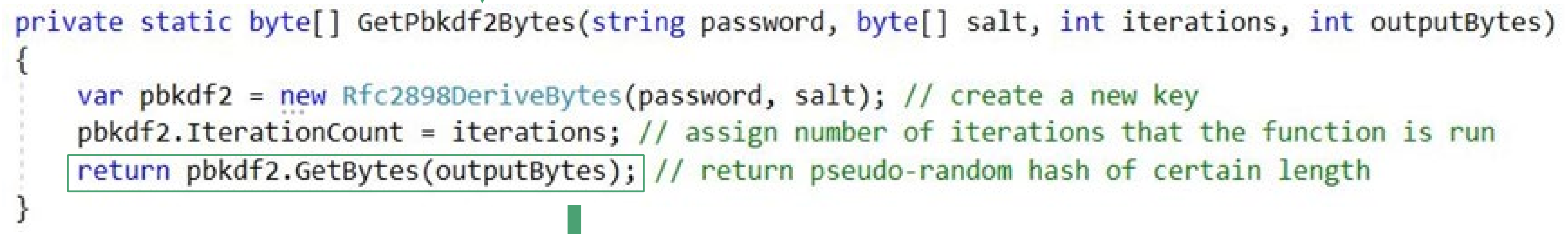


	UserID	Username	PasswordHash
1	1	colecb	1000:q38BxJh3pmpyP2WL3U1K735PhCfiKm:YR2N12g8IRDF9Am39/P52ZyssK8=



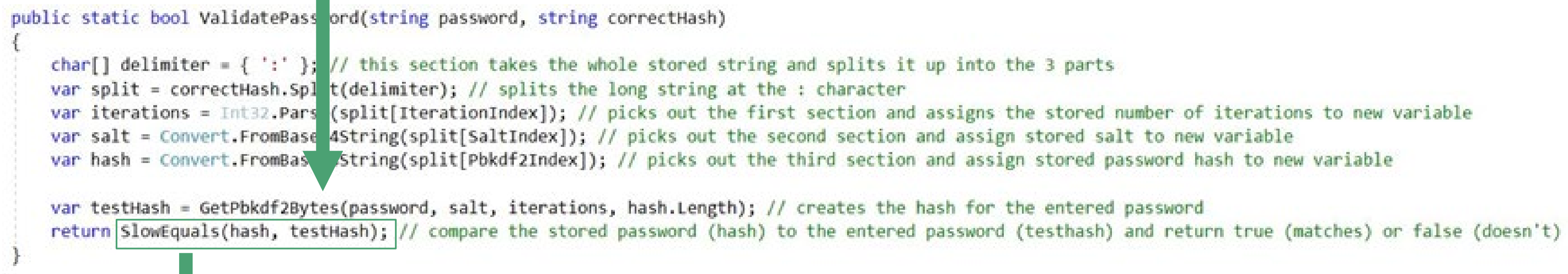
```
public static bool ValidatePassword(string password, string correctHash)
{
    char[] delimiter = { ':' }; // this section takes the whole stored string and splits it up into the 3 parts
    var split = correctHash.Split(delimiter); // splits the long string at the : character
    var iterations = Int32.Parse(split[IterationIndex]); // picks out the first section and assigns the stored number of iterations to new variable
    var salt = Convert.FromBase64String(split[SaltIndex]); // picks out the second section and assign stored salt to new variable
    var hash = Convert.FromBase64String(split[Pbkdf2Index]); // picks out the third section and assign stored password hash to new variable

    var testHash = GetPbkdf2Bytes(password, salt, iterations, hash.Length); // creates the hash for the entered password
    return SlowEquals(hash, testHash); // compare the stored password (hash) to the entered password (testhash) and return true (matches) or false (doesn't)
}
```

3. Create hash (HashPassword.cs)



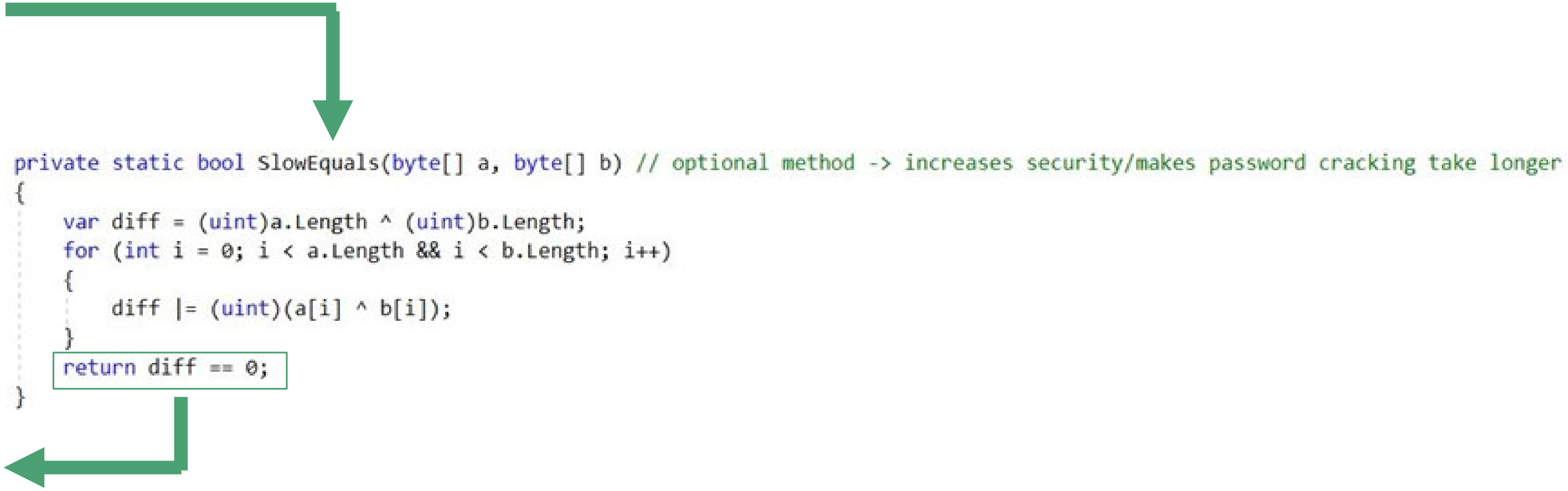
```
private static byte[] GetPbkdf2Bytes(string password, byte[] salt, int iterations, int outputBytes)
{
    var pbkdf2 = new Rfc2898DeriveBytes(password, salt); // create a new key
    pbkdf2.IterationCount = iterations; // assign number of iterations that the function is run
    return pbkdf2.GetBytes(outputBytes); // return pseudo-random hash of certain length
}
```

4. Continue validating password (HashPassword.cs)



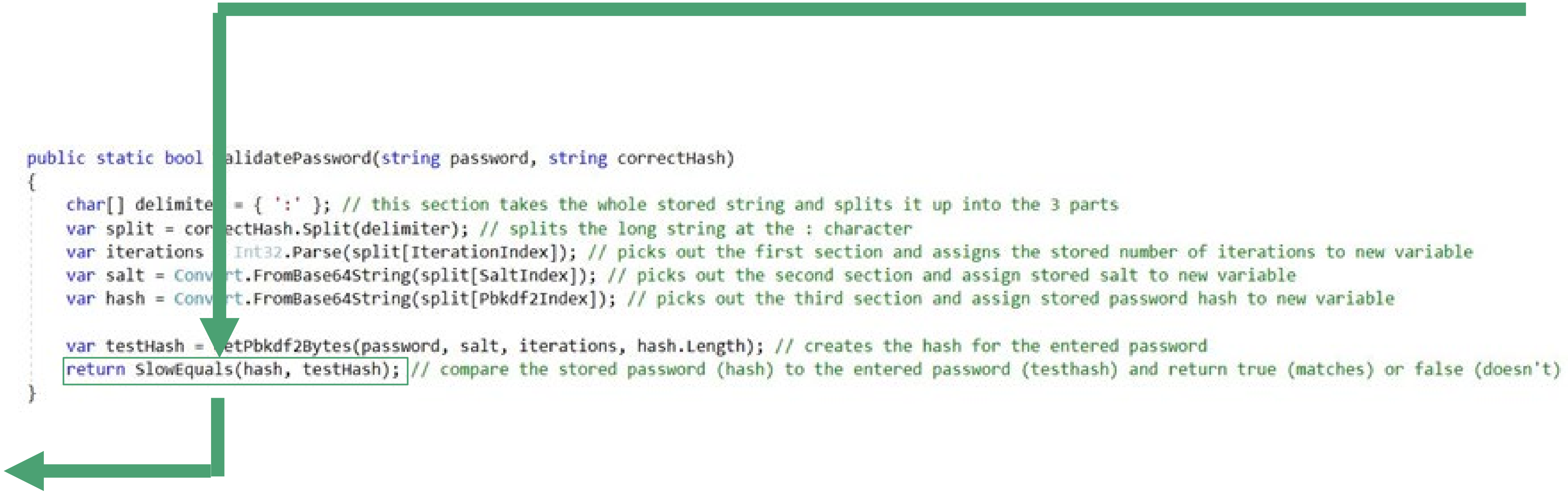
```
public static bool ValidatePassword(string password, string correctHash)
{
    char[] delimiter = { ':' }; // this section takes the whole stored string and splits it up into the 3 parts
    var split = correctHash.Split(delimiter); // splits the long string at the : character
    var iterations = Int32.Parse(split[IterationIndex]); // picks out the first section and assigns the stored number of iterations to new variable
    var salt = Convert.FromBase64String(split[SaltIndex]); // picks out the second section and assign stored salt to new variable
    var hash = Convert.FromBase64String(split[Pbkdf2Index]); // picks out the third section and assign stored password hash to new variable

    var testHash = GetPbkdf2Bytes(password, salt, iterations, hash.Length); // creates the hash for the entered password
    return SlowEquals(hash, testHash); // compare the stored password (hash) to the entered password (testhash) and return true (matches) or false (doesn't)
}
```

5. Slow equals (HashPassword.cs)



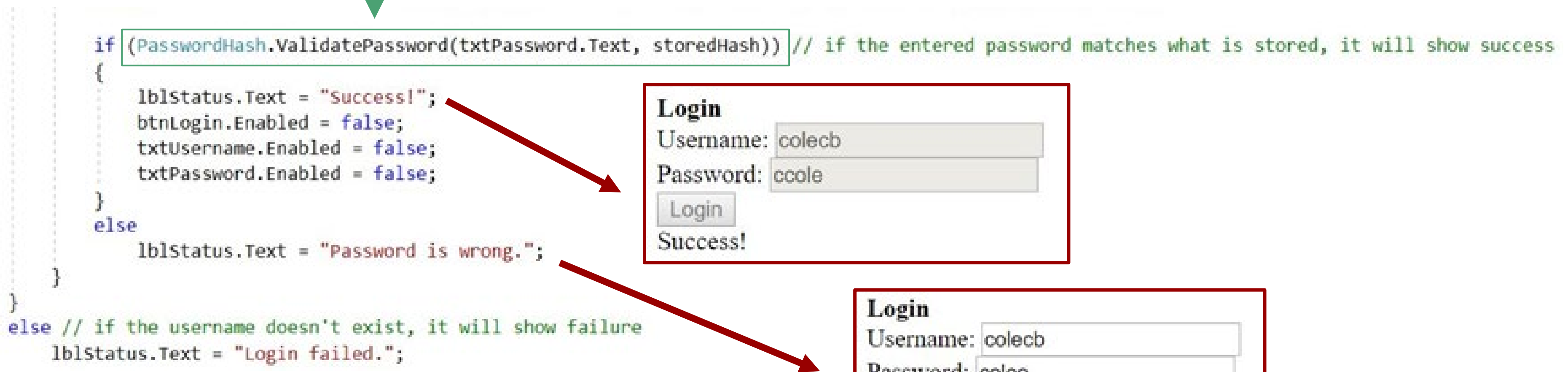
```
private static bool SlowEquals(byte[] a, byte[] b) // optional method -> increases security/makes password cracking take longer
{
    var diff = (uint)a.Length ^ (uint)b.Length;
    for (int i = 0; i < a.Length && i < b.Length; i++)
    {
        diff |= (uint)(a[i] ^ b[i]);
    }
    return diff == 0;
}
```

6. Finish validating password (HashPassword.cs)

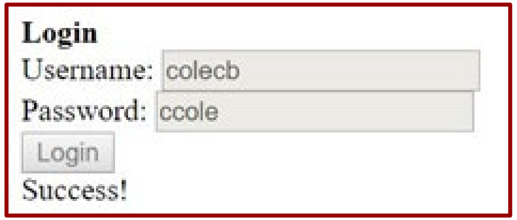


```
public static bool  alidatePassword(string password, string correctHash)
{
    char[] delimite  = { ':' }; // this section takes the whole stored string and splits it up into the 3 parts
    var split = cor ectHash.Split(delimiter); // splits the long string at the : character
    var iterations   Int32.Parse(split[IterationIndex]); // picks out the first section and assigns the stored number of iterations to new variable
    var salt = Conv rt.FromBase64String(split[SaltIndex]); // picks out the second section and assign stored salt to new variable
    var hash = Conv rt.FromBase64String(split[Pbkdf2Index]); // picks out the third section and assign stored password hash to new variable

    var testHash =  etPbkdf2Bytes(password, salt, iterations, hash.Length); // creates the hash for the entered password
    return SlowEquals(hash, testHash); // compare the stored password (hash) to the entered password (testhash) and return true (matches) or false (doesn't)
}
```

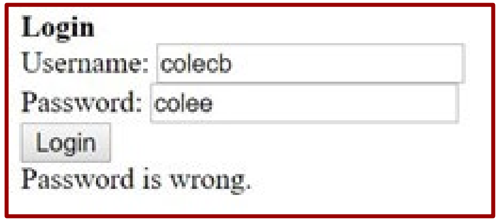
7. Finish logging in (userLogin.aspx.cs)



```
        if (PasswordHash.ValidatePassword(txtPassword.Text, storedHash)) // if the entered password matches what is stored, it will show success
        {
            lblStatus.Text = "Success!";
            btnLogin.Enabled = false;
            txtUsername.Enabled = false;
            txtPassword.Enabled = false;
        }
        else
            lblStatus.Text = "Password is wrong.";
    }
}
else // if the username doesn't exist, it will show failure
    lblStatus.Text = "Login failed.";
```



Login
Username: colecb
Password: ccole
Login
Success!



Login
Username: colecb
Password: colee
Login
Password is wrong.

Resources

- https://www.youtube.com/watch?v=cczlpiiu42M&t=317s
- https://www.youtube.com/watch?v=425_1-eFel4
- https://learningnetwork.cisco.com/thread/129462
- https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet
- https://www.lynda.com/IT-Infrastructure-tutorials/Key-stretching/645055/720283-4.html
- https://cmatskas.com/-net-password-hashing-using-pbkdf2/