Salting and Stretching: Concepts and Implementations

By Logan Gore

This paper was written in January 2015 as a reflection of current best practices. When reading this paper, consideration should be given to its age and whether this information is still current.

Abstract

Are you an IT professional who could be held responsible for a company data breach? Many systems encrypt and hash data without effective salting and stretching schemes, and then suffer the consequences when they get hacked. Yours might, too. Salting and stretching is a simple yet effective means to squeeze the most security from a limited-entropy key or password, crippling the most common and successful cryptographic attacks. This paper includes a *Concepts* section that explains how salting and stretching relates to cryptographic solutions, and an *Implementations* section that provides *complete* C# examples of salting and stretching as applied to symmetric encryption and password hashing.

Scope of Cryptography

The scope of cryptography in this paper will extend to hashing, symmetric key encryption, some relevant attacks, and how salting and stretching relates cryptographic solutions. This is not a deep survey of cryptography or cryptanalysis, but readers are expected to understand basic aspects of symmetric encryption and hashing.

Scope of Coding

The *Concepts* section will not contain any code. In the *Implementations* section, code examples will be in C# because it's easy to read and widely used. .NET's *System.Security.Cryptography* namespace will be used to illustrate hashing, stretching, and encryption schemes. Readers should be experienced in the programming language they are developing in and have a firm understanding of secure memory management, string immutability issues, and cryptography libraries.

Part 1- Concepts

Storing Secret Data

Storing secret data commonly leads to one of two processes: symmetric encryption or hashing. Symmetric encryption is a two-way process that protects data you want to store and recover/decrypt at a later time. Examples of this type of data could be credit card numbers (CCNs), social security numbers (SSNs), personally identifiable information (PII), or anything else that warrants this level of protection.

Hashing is a one way process concerned with integrity –generating and comparing hashes (fingerprinting) rather than recovering the secret data. While hashing is used to check the integrity of many kinds of data, the context of hashing in this paper will be limited to the storage of password hashes.

Attacking Secret Data

Security in modern cryptographic systems really just boils down to time. Compromise is inevitable when thinking in terms of exhaustive search; what we really care about is time. How many years or decades would it take? At the end of the day, bit length, diffusion, salting, stretching, entropy, perfect forward secrecy (PFS), or pretty much any other fancy cryptographic term you hear, plays its part in the race against the clock.

Attacking Symmetric-Key Encryption

Most successful attacks against symmetric encryption systems attempt to find the private key. It's the most important piece to the puzzle. Common cryptographic attacks include:

- known-plaintext
- chosen-plaintext
- chosen-chiphertext
- ciphertext-only
- chosen-key
- related-key
- dictionary attacks
- information leakage and side-channel (outside the scope of this paper)
- and generic attacks like meet-in-the-middle, birthday, and exhaustive search

[Ferguson, Schneier, Kohno]

Once the private key has been compromised, all secret data encrypted with that key can be decrypted at will.

Attacking Password Hashes

Attacking password hashes is a little different. The most efficient attack against the password hash is a *rainbow table*, though generic attacks are still applicable. The concept itself basically combines dictionary and brute-force attacks. A rainbow table is a lookup of plaintext passwords and their corresponding hashes derived from a specific hash algorithm such as SHA256. Attackers use rainbow tables because they are very fast. Many rainbow tables are able to take a hash and find the corresponding password (if it is in the table) within a few seconds.

Most tables are computed around a set of specs that include max password length and complexity (numbers, symbols). Therefore, a rainbow table could be produced for SHA256 that includes all possible passwords under 8 characters. The password hash

"6XDEoHE+6457XtB1zc3xx0VKd4g4HdjhymFEsiEVRmk=" could be run against the table to find that the plaintext password is "32max!":

SHA256 Hash Value	Plaintext password
NlFaMi795BShmRBI2kS8ZWI8jhwx+MMMZSru4FQowjc=	Cat123
6XDEoHE+6457XtB1zc3xx0VKd4g4HdjhymFEsiEVRmk=	32max!
A6xnQhbz4Vx2HuGl4lXwZ5U2I8iziLRFnhP5eNfIRvQ=	1234
XohImNooBHFR0OVvjcYpJ3NgPQ1qq73WKhHvch0VQtg=	password
RJ9+mIT4XzBhheEgtrNcwluHEJ1AqDKpU2gmPgCwAmU=	seahawks

Brute-forcing is also an option and not a bad one when the attacker has a good set of tools and a little knowledge of the system. Don't assume that brute-forcing always takes a long time. Most password crackers can obtain low-entropy passwords (most of those in use today) within minutes. Some of these tools are worth learning to test the security of your system:

- JohnTheRipper
- L0phtcrack
- HashCat (oclHashCat)
- THC Hydra
- Medusa
  [McClure, Scambry, Kurtz]

Ultimately, you shouldn't care that your systems can be broken; you should concern yourself with how long this will take with current tools.

Security Level

The best way to quantify the risk an attack poses is to compare the time it takes to succeed against that of an exhaustive search attack (one that tries all possible values for a target, like the key). Therefore, your goal shouldn't be to stop attacks or build an unbreakable cryptographic solution. This just isn't feasible. The goal should really be to enforce a strong bit level of security that attackers can't avoid. In today's systems you should be aiming for a 128-bit or greater security level. This means that any attack will require 2^{128} steps. Additionally, you should be using 256-bit keys and 256-bit hashing algorithms due to birthday and meet in the middle attacks [Ferguson, Schneier, Kohno].

A well designed system truly providing a 128-bit security level will deter most attackers. By the time a persistent attacker is successful, the secret data protected by the encryption will hopefully be useless. Thus, never count on your encryption to keep something secret forever and understand that *Moore's Law* will always be working against you.

There is No Replacement for High Entropy

If password or key entropy is low enough, attacks harnessing the efficiency of rainbow tables, dictionaries, and key-guessing software, *can* circumvent the time factor that strong bit security provides. The best way to protect against this is to make sure you are using high-entropy passwords. Probably the

best tool for generating strong passwords is a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG). Most security libraries have one. Otherwise, you can just have fun on the keyboard: *HxGe%RY9#4jN@(dkUIp7389.*

Can't the attackers still brute-force your system? Yes! This is the only option you want them to have. 2^{128}? Be my guest.

What is Salting and Stretching?

In the simplest terms, a salt is a piece (bits) of data. Most salts are randomly generated sequences of data that are a certain length, such as 16 bytes (128 bits). Salts are added to secret data so when it's passed through a cryptographic function, the derivative is a product of the secret data and the salt –not just the secret data. This helps thwart attacks like rainbow tables that depend on the predictability of certain plaintext values (such as common passwords or dictionary words) producing certain hashes. Typically, salts do not need to be kept secret. Their purpose is just to make the process of attacking data derived from a cryptographic function take longer.

Stretching is the process of taking a key or password and hashing it –sometimes hundreds or even thousands of times– with a cryptographically secure hash function. These functions can be used with or without a salt as input, but for our purposes we will always assume a salt is being used. This helps increase entropy and can add a significant amount of extra work (time) to derive a key from a password and a salt. Depending on the solution, the hash produced from this process can be used as the fingerprint of the secret data if storing a password hash, or it can be used to encrypt and decrypt data in a symmetric algorithm like AES. The salt is usually stored in plaintext form with encrypted or hashed data to be used later in the decryption or validation process.

How does Salting and Stretching Increase Security?

As we've discussed earlier, time is the biggest factor in cryptography. The most important thing that salting and stretching does is increase the time it takes to compromise sensitive data protected by a cryptographic function.

Mitigating Attacks on Password Hashes:

Salting hashed passwords prevents attackers from using an existing rainbow table on your hashes. If each password is hashed using a cryptographically random salt (which is should be), the attacker must compute a rainbow table for every salt. Cracking your hashed passwords just went from a quick automated process, to a labor intensive and exponentially longer endeavor. This could actually mean the difference between a one second rainbow table search, and years of computing. Salting will also prevent identical user passwords from rendering the same hashes, which is good.

Mitigating Attacks on Symmetric Key Systems:

Salting and stretching will increase the bit security level that your private key is offering regardless if best practices like AES256-CBC with cryptographically secure IV have been implemented. Attackers

attempting to compromise the private key will be caught between the *Scylla* of an increased effective key size from the work required to perform the stretching function, and the *Charybdis* of an exhaustive search. The attackers can use whichever cryptographic attacks they want. But there's a caveat. If the attackers fail to use the salting and stretching function when attempting to compromise the key, a success will only reveal the *derived* key, capable of decrypting only that particular ciphertext.

Increased Effective Key Size:

To understand how salting and stretching can increase effective key size, we will draw up a theoretical stretching function *H* that derives a key *K,* where we input a salt *s,* a password *p*, and specify the number of iterations, *r*:

$$K=H(s,p,r)$$

The increase in key size is realized where the attacker must perform *r* computations in the stretching function. If $r=2^{20}$, the attacker must perform an additional 2^{20} computations for each password that is tried. Therefore, trying 2^{100} passwords would take 2^{120} hash computations, making the effective key size 20 bits longer [Ferguson, Schneier, Kohno].

Compensating for Moore's Law:

Moore's Law says that number of transistors on integrated circuits double approximately every two years (though towards the end of 2013 this was said to have slowed to every three years). This means that over time, data derived from cryptographic functions will be attacked with increasing speed/resources, leading to a reduced mean-time-to-compromise (MTTC). To compensate for this, developers can increase the computations in their stretching functions to offset the gains in computing speed.

Basic Salting and Stretching Procedures

There is no "one way" to implement salting and stretching. But generally, the steps don't change much.

How is Salting and Stretching Applied to a Symmetric Encryption System?

1) Generate salt: Generate a cryptographically-secure random salt at least as long as your algorithm's key size (as you have it implemented). Therefore, salts should be *at least* 128 bits for regular AES, and 256 bits for AES256.
2) Stretch: Run your private key and your salt through a key stretching function to create the *derived key*. Most of these functions allow you to specify the number of hashing iterations that are performed. This number should make the function take 200-1000ms to execute.
3) Encrypt data, attach salt, and store: Encryption will be performed (producing the ciphertext) using the derived key from step 2. The salt will be attached to the ciphertext so it can be used in the decryption process. The ciphertext (with the salt attached) can now be stored or transmitted.

4) Disassemble, stretch, and decrypt: The ciphertext must be decrypted using the key that encrypted the data. Removing the salt from the ciphertext and repeating the function described in step 2 –exactly as it was performed in the encryption phase– will generate the derived key that will decrypt the ciphertext.

How is Salting and Stretching Applied to Password Hashes?

1) Generate salt: Generate a cryptographically-secure random salt at least as long as your algorithm's bit length. Therefore, salts should be at least 256 bits for SHA256.
2) Stretch: Run the password and the salt through a key stretching function to create the derived key. Most of these functions allow you to specify the number of hashing iterations that are performed. This number should make the function take 200-1000ms to execute.
3) Attach salt and store: Prepend or append the salt to the derived key and store the data.
4) Disassemble, stretch, and verify: When the plaintext password must be verified, retrieve the stored hash and remove its salt. The salt and plaintext password are stretched in the same stretching function used in step 2 and the result is compared to the stored hash.

Part 2- Implementations in C#/.NET

We will use examples that consume and return string data due to the overwhelming demand for "string encryption" and "string password hashing" examples in the industry, and moreover, because of the overwhelming amount of examples being shared that implement bad practices like fixed IV's, ECB mode, and in more than one case, the accidental storage of the private key with the ciphertext.

Appendix A and B contain complete working code for symmetric encryption and password hashing classes that implement general best practices and salting and stretching.

Most security libraries will give developers more than enough tools to build robust and secure solutions. We will use the well-known *System.Security.Cryptography* namespace in .NET.

Make sure you understand the immutability of string data in C#. If your classes are handling sensitive string data, it will be stored in memory until the garbage collector can free it. You should take this into consideration if your app is running in a virtualized or shared environment and make sure that you have secure memory solutions and active measures to prevent VMEscape.

Generating a Random Salt

Under best practices, salts should be random and –for best security– at least the bit length of your hashing function or encryption key. Most security libraries have a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG). In .NET we will use *RNGCryptoServiceProvider.GetBytes* to produce our salts. The *GetBytes* method fills an array of bytes with a cryptographically strong sequence

of random values. Below we will create a method that will generate random salts with a byte size that is specified in the method parameter.

```
private byte[] GetSalt(int byteSize)
{
    byte[] salt = new byte[byteSize];

    using (RNGCryptoServiceProvider RNG = new RNGCryptoServiceProvider())
    {
        RNG.GetBytes(salt);
    }

    return salt;
}
```

Creating a 256-bit (32 byte) salt could be done like so:

```
byte[] salt = GetSalt(32);
```

Stretching Functions

For our purposes, a *stretching function* is a function that takes a key/password and a salt as parameters, performs the salting and stretching using a cryptographically secure hash function, and returns a *derived* key. Developers are left with the choice of writing their own stretching function or using one of the standardized implementations available in many security libraries. It's always recommended to use the library approach because it's more productive and usually more secure.

Standardized Functions: PBKDF2 in a C#/.NET Example

The Public Key Cryptography Standard #5 (PKCS5) outlines key derivation functions that are used to salt and stretch passwords; namely Password Based Key Derivation Function 1 and 2 (PBKDF1 and PBKDF2). Both are used to produce derived keys, PBKDF2 however, uses pseudo-random number generation. Security libraries will often have implementations of PBKDF1 and PBKDF2. The *System.Security.Cryptography* namespace in .NET does just this with the *PasswordDeriveBytes* class (PBKDF1) and *Rfc2898DeriveBytes* (PBKDF2).

Below we will implement PBKDF2 via *Rfc2898DeriveBytes*. This will derive a key from a salt, a private key, and a specified number of stretching iterations.

```
private static byte[] GetDerivedKey(byte[] key, byte[] salt, int iterations)
{
    Rfc2898DeriveBytes RFC = new Rfc2898DeriveBytes(key, salt, iterations);
    return RFC.GetBytes(32); //32 is the key size in bytes that will be created
}
```

Writing Your Own

There is no "one way" to write a stretching function. Here is an example of a simple one using *SHA256*:

```
private byte[] GetKeyFromPasswordAndSalt(byte[] key, byte[] salt, int iterations)
{
     using (var SHA = new SHA256Managed())
     {
         byte[] keyPlusSalt = key.Concat(salt).ToArray();

         byte[] derivedKey = SHA.ComputeHash(keyPlusSalt);

         for (int i = 0; i < iterations; i++)
         {
             derivedKey = SHA.ComputeHash(derivedKey);
         }

         return derivedKey;
     }

}
```

Implementing Salting and Stretching in a Symmetric Key System

This example will include the code to implement AES256 encryption.

First, we'll write our AES256 encryption and decryption methods. They are private, and as such, will not be exposed. These methods will not deal with any salting or stretching; their only responsibility is encrypting and decrypting data.

```
private static byte[] EncryptWithKey(byte[] data, byte[] key)
{
    byte[] rawData = data;
    byte[] encryptedData = null;

    using (var Aes = new AesManaged())
    {
        Aes.KeySize = 256;
        Aes.Key = key;
        Aes.Mode = CipherMode.CBC;
        Aes.Padding = PaddingMode.PKCS7;
        Aes.GenerateIV();

        using (ICryptoTransform encryptor = Aes.CreateEncryptor())

        using (MemoryStream memStrm = new MemoryStream())
        {
            var crptStrm = new CryptoStream(memStrm, encryptor, CryptoStreamMode.Write);

            memStrm.Write(Aes.IV, 0, Aes.IV.Length);

            crptStrm.Write(rawData, 0, rawData.Length);

            crptStrm.FlushFinalBlock();

            crptStrm.Close();

            encryptedData = memStrm.ToArray();
        }
    }
```

```
     return encryptedData;
}

private static byte[] DecryptWithKey(byte[] data, byte[] key)
{
    byte[] rawData = null;

    using (var Aes = new AesManaged())
    {
        int nBytes = 16;

        byte[] iv = new byte[nBytes];

        for (int i = 0; i < iv.Length; i++)
            iv[i] = data[i];

        Aes.Padding = PaddingMode.PKCS7;
        Aes.Mode = CipherMode.CBC;
        Aes.KeySize = 256;
        Aes.Key = key;
        Aes.IV = iv;

        using (ICryptoTransform decryptor = Aes.CreateDecryptor())

        using (MemoryStream memStrm = new MemoryStream())
        {
            var crptStrm = new CryptoStream(memStrm, decryptor, CryptoStreamMode.Write);

            crptStrm.Write(data, nBytes, data.Length - nBytes);

            crptStrm.FlushFinalBlock();

            crptStrm.Close();

            rawData = memStrm.ToArray();
        }
     }

    return rawData;
}
```

Next, we'll need a method to generate our salt. For simplicity, we'll just copy the method we used last section:

```
private static byte[] GetSalt(int byteSize)
{
    byte[] salt = new byte[byteSize];

    using (RNGCryptoServiceProvider RNG = new RNGCryptoServiceProvider())
    {
        RNG.GetBytes(salt);
    }

    return salt;
}
```

Our private stretching method will implement PBKDF2 via the *Rfc2898DeriveBytes* class. Just above the method we created a public variable called *StretchIterations* and set it at 40000. You can adjust it as you see fit. In Visual Studio 2013, on our VMWare Windows 8, it took 450ms to execute the stretching function @40000 iterations with 256-bit salts that derives a 32-byte key. This timeframe meets our target range of 200-1000ms. Remember to test yours in the environment where it will be running (or a decent simulation/lab).

```
public static int StretchIterations = 40000;

private static byte[] GetKeyFromPasswordAndSalt(byte[] key, byte[] salt, int iterations)
{
    using (Rfc2898DeriveBytes RFC = new Rfc2898DeriveBytes(key, salt, iterations))
    {
        key = RFC.GetBytes(32);
    }
    return key;
}
```

Our publicly exposed encryption and decryption methods will bring everything together. We will follow the procedures outlined previously in the *Concepts* section.

The *EncryptData* method will generate a random salt by calling the *GetSalt* method, input the salt and the private key into the stretching method to produce a derived key, and use the derived key to encrypt the data. The salt will be concatenated with the ciphertext and the resulting byte array is returned. You may also notice that we attached the value of the *StretchIterations* variable to the returned ciphertext; this ensures that if the stretch iterations are increased over time that we can still replicate derived keys that were produced using the previous stretch iteration number.

```
public static string EncryptData(string data, string key)
{
     if (data == null) throw new NullReferenceException(data);
     if (key == null) throw new NullReferenceException(key);

     byte[] salt = GetSalt();
     byte[] derivedKey = GetKeyFromPasswordAndSalt(Encoding.UTF8.GetBytes(key), salt,
StretchIterations);
     byte[] encryptedData = EncryptWithKey(Encoding.UTF8.GetBytes(data), derivedKey);


       return(StretchIterations.ToString()+":"+Convert.ToBase64String(salt)+":"+Convert.ToBase
64String(encryptedData));
}
```

The *DecryptData* method will remove the stretch iterations and salt from the encrypted data (the same salt used to encrypt the data), put the salt, private key, and stretch iterations in the stretching function to produce the derived key, and decrypt using the derived key.

```
public static string DecryptData(string encyptedData, string key)
{
```

```
    if (encyptedData == null) throw new NullReferenceException(encyptedData);

    string[] parts = encyptedData.Split(':');

    if (parts == null || parts[0] == null || parts[1] == null || parts[2] == null)
        throw new NullReferenceException("Null data found when parsing encrypted data.");

    int stretchIterations = Convert.ToInt32(parts[0]);
    byte[] salt = Convert.FromBase64String(parts[1]);
    byte[] ciphertext = Convert.FromBase64String(parts[2]);

    byte[] derivedKey = GetKeyFromPasswordAndSalt(Encoding.UTF8.GetBytes(key), salt,
stretchIterations);

    byte[] plaintext = DecryptWithKey(ciphertext, derivedKey);

    return Encoding.UTF8.GetString(plaintext);
}
```

Salting and Stretching Stored Hashes

In this example we will hash our data using the *SHA256Managed* class in .NET and stretch the computed SHA hash in *Rfc2898DeriveBytes*. We will use the salting and stretching classes from the last example, but here we'll incorporate them into the publicly exposed methods rather than abstracting them into their own private methods.

The first method we have is *HashPassword*. This method takes a password as the sole parameter and is in charge of generating the salt, hashing and stretching the password, and returning the stretch iterations and salt concatenated with the stretched password. Notice that we have a public variable called *StretchIterations* that is defaulted to 40000, but will allow the developer to adjust the stretch iterations as they see fit.

```
public static int StretchIterations = 40000;

public static string HashPassword(string password)
{
    byte[] hash = new byte[32];
    byte[] salt = new byte[32];

    using (RNGCryptoServiceProvider RNG = new RNGCryptoServiceProvider())
    {
        RNG.GetBytes(salt);
    }

    using (var SHA = new SHA256Managed())
    {
        byte[] shaBytes = SHA.ComputeHash(Encoding.UTF8.GetBytes(password));

        using (Rfc2898DeriveBytes RFC = new Rfc2898DeriveBytes(shaBytes, salt,
StretchIterations))
        {
            hash = RFC.GetBytes(32);
        }
    }
```

```
    return (StretchIterations.ToString() + ":" + Convert.ToBase64String(salt) + ":" +
Convert.ToBase64String(hash));
 }
```

The next method is *VerifyHash*. It's used to verify that the plaintext password that is entered will render an identical hash to the existing/stored password hash. If the same plaintext is entered that was used to derive the stored hash, the method will return true. Internally, the method will remove the salt and stretch iterations from the stored hash, pass the plaintext password, salt, and stretch iterations into the same stretching function that was used in the *HashPassword* method (*Rfc2898DeriveBytes*), and compare the stored hash to the computed hash.

```
public static bool VerifyHash(string stretchedPassword, string plaintextPassword)
{
     if (stretchedPassword == null) return false;

     string[] parts = stretchedPassword.Split(':');

     if (parts == null || parts[0] == null || parts[1] == null || parts[2] ==
null||plaintextPassword==null) return false;

     byte[] hash = new byte[32];

     byte[] salt = Convert.FromBase64String(parts[1]);

     int iterations = Convert.ToInt32(parts[0]);

     using (var SHA = new SHA256Managed())
     {
         byte[] shaBytes = SHA.ComputeHash(Encoding.UTF8.GetBytes(plaintextPassword));

         using (Rfc2898DeriveBytes RFC = new Rfc2898DeriveBytes(shaBytes, salt, iterations))
         {
             hash = RFC.GetBytes(32);
         }
     }

     return (parts[2] == Convert.ToBase64String(hash));
}
```

Salt Storage Alternatives: Defense in Depth

Throughout this paper we have discussed storing salts with the ciphertext. This is by far the most common method of storing the salt; it's also the simplest. There are other circumstances, however, that warrant the salts to be separated from the ciphertext, and stored in a secured database or file.

Security, maintainability, and scalability are often cited as the driving factors for storing salts separately. Systems can realize many benefits from an abstracted salt store:

- Granular access control and increased security posture from requiring strong permissions to access salts
- Increased simplicity and atomicity as a result of storing only the salt data
- Cleaner and more logical methods to provide redundancy and backup solutions

Some solutions have used input data as the salt and handled the stretching on the fly, opting not to store the salt. This is fine if the salt can be correctly reproduced at a later date by the user or application. However, this often opens the door to the use of low-entropy (insecure) salts.

We are not advocating one method of storage over the other, but programmers developing cryptographic solutions need to be aware of the pros and cons of such designs.

Designing for Maintainability and Future Threats

Wise developers will increase stretching iterations to maintain an effective key size and compensate for the gains in computing power. What would have happened with our example code if we hadn't attached the number of stretch iterations to the ciphertext or hash? Changing the stretching iterations would have prevented us from decrypting or verifying data that used a different number of stretching iterations. Issues like this are very common and need to be considered in the design of your system. Architectures that balance security, maintainability, and scalability, are critical in cryptographic solutions. This is best achieved by using good programming practices to design solutions that can implement changing security and business requirements.

Conclusion

Big data breaches are headline news. The vast majority could be prevented by attention to detail and layered security. What isn't headline news are the millions of attacks underway right now and the millions of successful breaches that haven't yet been discovered or reported.

The cost benefit analysis of salting and stretching –in terms of developer hours and CPU usage vs. the cost of a single data breach– for *any* company is a no-brainer. If you're reading this and think that your performance demands are too high to salt and stretch your keys and passwords, you probably serve such a large user base that a data breach would be catastrophic. So think again.

Salting and stretching is an effective and inexpensive method to mitigate many of today's, and more importantly, tomorrow's, cryptographic attacks.

References

[McClure, Scambry, Kurtz]

McClure, Scambry, Kurtz. *Hacking Exposed, 7th Edition*. McGraw Hill. 2012.

[Schneier, Ferguson, Kohno]

Schneier, Ferguson, Kohno. *Cryptography Engineering: Design Principles and Practical Applications.* Wiley Publishing, Inc. 2010.

About the Author

Logan Gore is a software engineer who has published several Open Source projects for C#/.NET and papers on current industry topics such as enterprise architecture, cryptography, and Big Data. He holds professional certifications in enterprise security, software and database development, cloud technologies, networking, and project management. Logan is currently involved in cloud, game, and mobile development.

loganlgore@gmail.com

Appendix 1: Symmetric Encryption C#

```
        public class AesEncrypt
        {
            public static int StretchIterations = 40000;

            private static byte[] EncryptWithKey(byte[] data, byte[] key)
            {
                byte[] rawData = data;
                byte[] encryptedData = null;

                using (var Aes = new AesManaged())
                {
                    Aes.KeySize = 256;
                    Aes.Key = key;
                    Aes.Mode = CipherMode.CBC;
                    Aes.Padding = PaddingMode.PKCS7;
                    Aes.GenerateIV();

                    using (ICryptoTransform encryptor = Aes.CreateEncryptor())

                    using (MemoryStream memStrm = new MemoryStream())
                    {
                        var crptStrm = new CryptoStream(memStrm, encryptor,
CryptoStreamMode.Write);

                        memStrm.Write(Aes.IV, 0, Aes.IV.Length);

                        crptStrm.Write(rawData, 0, rawData.Length);

                        crptStrm.FlushFinalBlock();

                        crptStrm.Close();

                        encryptedData = memStrm.ToArray();
                    }
                }

                return encryptedData;
            }

            private static byte[] DecryptWithKey(byte[] data, byte[] key)
            {
                byte[] rawData = null;

                using (var Aes = new AesManaged())
                {
                    int nBytes = 16;

                    byte[] iv = new byte[nBytes];

                    for (int i = 0; i < iv.Length; i++)
                        iv[i] = data[i];

                    Aes.Padding = PaddingMode.PKCS7;
                    Aes.Mode = CipherMode.CBC;
                    Aes.KeySize = 256;
                    Aes.Key = key;
                    Aes.IV = iv;
```

```
                    using (ICryptoTransform decryptor = Aes.CreateDecryptor())

                    using (MemoryStream memStrm = new MemoryStream())
                    {
                        var crptStrm = new CryptoStream(memStrm, decryptor,
CryptoStreamMode.Write);

                        crptStrm.Write(data, nBytes, data.Length - nBytes);

                        crptStrm.FlushFinalBlock();

                        crptStrm.Close();

                        rawData = memStrm.ToArray();
                    }
                }

                return rawData;
            }

            private static byte[] GetSalt(int byteSize)
            {
                byte[] salt = new byte[byteSize];

                using (RNGCryptoServiceProvider RNG = new RNGCryptoServiceProvider())
                {
                    RNG.GetBytes(salt);
                }

                return salt;
            }

            private static byte[] GetKeyFromPasswordAndSalt(byte[] key, byte[] salt, int
iterations)
            {
                using (Rfc2898DeriveBytes RFC = new Rfc2898DeriveBytes(key, salt, iterations))
                {
                    key = RFC.GetBytes(32);
                }
                return key;
            }

            public static string EncryptData(string data, string key)
            {
                if (data == null) throw new NullReferenceException(data);
                if (key == null) throw new NullReferenceException(key);

                byte[] salt = GetSalt(32);
                byte[] derivedKey = GetKeyFromPasswordAndSalt(Encoding.UTF8.GetBytes(key),
salt, StretchIterations);
                byte[] encryptedData = EncryptWithKey(Encoding.UTF8.GetBytes(data),
derivedKey);

                return (StretchIterations.ToString() + ":" + Convert.ToBase64String(salt) +
":" + Convert.ToBase64String(encryptedData));
            }

            public static string DecryptData(string encyptedData, string key)
            {
                if (encyptedData == null) throw new NullReferenceException(encyptedData);
```

```
                string[] parts = encyptedData.Split(':');

                if (parts == null || parts[0] == null || parts[1] == null || parts[2] == null)
                    throw new NullReferenceException("Null data found when parsing encrypted
data.");

                int stretchIterations = Convert.ToInt32(parts[0]);
                byte[] salt = Convert.FromBase64String(parts[1]);
                byte[] ciphertext = Convert.FromBase64String(parts[2]);

                byte[] derivedKey = GetKeyFromPasswordAndSalt(Encoding.UTF8.GetBytes(key),
salt, stretchIterations);

                byte[] plaintext = DecryptWithKey(ciphertext, derivedKey);

                return Encoding.UTF8.GetString(plaintext);
            }


        }
```

Appendix B: Password Hashing C#

```
public class Hashing
        {
            public static int StretchIterations = 40000;

            public static string HashPassword(string password)
            {
                byte[] hash = new byte[32];
                byte[] salt = new byte[32];

                using (RNGCryptoServiceProvider RNG = new RNGCryptoServiceProvider())
                {
                    RNG.GetBytes(salt);
                }

                using (var SHA = new SHA256Managed())
                {
                    byte[] shaBytes = SHA.ComputeHash(Encoding.UTF8.GetBytes(password));

                    using (Rfc2898DeriveBytes RFC = new Rfc2898DeriveBytes(shaBytes, salt,
StretchIterations))
                    {
                        hash = RFC.GetBytes(32);
                    }
                }

                return (StretchIterations.ToString() + ":" + Convert.ToBase64String(salt) +
":" + Convert.ToBase64String(hash));
            }

            public static bool VerifyHash(string stretchedPassword, string plaintextPassword)
            {
                if (stretchedPassword == null) return false;
```

```
                string[] parts = stretchedPassword.Split(':');

                if (parts == null || parts[0] == null || parts[1] == null || parts[2] == null
|| plaintextPassword == null) return false;

                byte[] hash = new byte[32];

                byte[] salt = Convert.FromBase64String(parts[1]);

                int iterations = Convert.ToInt32(parts[0]);

                using (var SHA = new SHA256Managed())
                {
                    byte[] shaBytes =
SHA.ComputeHash(Encoding.UTF8.GetBytes(plaintextPassword));

                    using (Rfc2898DeriveBytes RFC = new Rfc2898DeriveBytes(shaBytes, salt,
iterations))
                    {
                        hash = RFC.GetBytes(32);
                    }
                }

                return (parts[2] == Convert.ToBase64String(hash));
            }
        }
    }
```