

CRYPTOGRAPHY IN .NET

SUCCINCTLY

BY **DIRK STRAUSS**

Cryptography in .NET Succinctly

By
Dirk Strauss

Foreword by Daniel Jebaraj



Copyright © 2017 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: John Elderkin

Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.

Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

About the Author	7
Chapter 1 Cryptography: A Brief History	8
Chapter 2 Cryptographic Signatures	10
Generating and verifying signatures	10
Why you shouldn't use String.GetHashCode()	18
Chapter 3 Hashing and Salting Passwords	19
Encrypting user details	21
Validating user details	26
How strong are unsalted passwords?	28
Chapter 4 Symmetric Encryption	30
The generated key	36
The initialization vector	36
Chapter 5 Asymmetric Encryption	37
Writing the code	38
Chapter 6 Cryptographic Key Containers	40
Writing the code	42
Exporting the public key	49
Importing a public key	53
Using Azure Key Vault	54
Chapter 7 Using SecureString	59
Conclusion	67

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge
As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Dirk Strauss is a software developer and Microsoft MVP from South Africa with more than 13 years of programming experience. He has extensive experience in SYSPRO Customization (an ERP system). C# and web development are his main focus. In 2016, he authored *The C# Programming Cookbook* and followed that up in 2017 with *The C# 7 and .NET Core Cookbook*. His previous titles for Syncfusion's *Succinctly* series include *C# Code Contracts Succinctly* and *Asynchronous Programming Succinctly*. Apart from authoring, he blogs at www.dirkstrauss.com whenever he gets a chance.

Chapter 1 Cryptography: A Brief History

The need to convey information in a secure manner has existed for thousands of years. For as long as people have needed to communicate, other people have wanted to undermine any efforts to secure that information. You might think that you don't have information important enough for anyone to want to steal, but that is where you are mistaken.

Apathy regarding information security is a problem for all of us. It permeates everyday life at home, at the workplace, and in every aspect of human interaction. Irresponsible ownership of data (no matter to whom that data belongs) is the cause of many leaked emails, data, and other damaging information. The rise of WikiLeaks and the Edward Snowden incident go to show that information is indeed power.

Whatever side of the fence you are on with regards to the Snowden leaks, a few things should be evident.

- Governments will and do actively surveil the public. This includes emails, phone calls, and Internet data.
- Government intelligence agencies do cooperate to share information with each other (British intelligence shared intercepted global emails, Facebook posts, calls, and Internet history with the NSA).
- The agencies responsible for collecting data have vast budgets and almost limitless resources.

If you think that you are very careful with your own data, unfortunately the same cannot be said of organizations that store your data and personal information. In 2016 alone, the list of notable institutions hacked is indeed very alarming. Here are a few of the more notable breaches:

- Feb. 8, 2016—University of Central Florida data hack.
- Feb. 9, 2016—U.S. Dept. of Justice data hack.
- March 3, 2016—700 Snapchat employees phished.
- March 10, 2016—Premier Healthcare data breach after a password-protected (but nonencrypted) laptop was stolen and data of 200,000 patients (including financial info) stolen.
- May 11, 2016—Wendy's data breach that leaked customer card data.
- May 17, 2016—LinkedIn data breach in 2012 resulted with the information (117 user email and password combinations) posted online.
- Aug. 12, 2016—Oracle's MICROS POS system data breach.
- Sept. 2, 2016—Dropbox revealed that 68 million usernames and passwords were breached in 2012 (considerably more than initially reported in 2012).
- Sept. 22, 2016—Yahoo announced that account information from 500 million users was stolen in 2014, making it the largest data breach in history at the time.
- Nov. 3, 2016—Cisco's Professional Careers website leaked the personal information of job seekers because of a faulty security setting.
- Nov. 13, 2016—AdultFriendFinder.com was hacked and had more than 400 million members' account information leaked online.
- Dec. 14, 2016—Yahoo announced that another data breach in 2013 compromised the personal information of one billion Yahoo accounts, making it the biggest data breach in history.

With the price of information at a premium and with a legion of black hats chomping at the bit to get their hands on that information, where does the buck stop? It stops with you, the software developer. The very fact that you are reading a document that discusses cryptography is a very positive indication that the security of user information is important to you.

Securing a user's personal information is not the job of someone else. It is your job. If you, as a developer, can decrypt the information stored in the database of the system you are working on, then so can anyone else. If a password is all there is standing between user data and unauthorized access, then the system has a serious vulnerability.

Here is a shortened (incomplete) history of cryptography. For the full list, refer to https://en.wikipedia.org/wiki/Timeline_of_cryptography.

BCE

- 36th century—Cuneiform, or wedge-shaped writing, is invented by the Sumerians.
- 600-500—Atbash substitution cipher used by Hebrew writers to encode words. Some found in the Bible, such as Sheshach, which means Babylon.
- 400—Herodotus uses steganography to send a message to Aristagoras.
- 100—Roman shift cipher known as Caesar's cipher is used. Named after Julius Caesar, who used it.

CE

- 801-873—Techniques for breaking monoalphabetic substitution ciphers are developed by mathematician Al-Kindi.
- 1355-1418—Ahmad al-Qalqashandi writes the Subh al-a 'sha, which included a section on cryptology.
- 1450-1520—The Voynich manuscript (named after Wilfrid Voynich, who purchased it in 1912) is a mysterious and as yet undeciphered document written in an unknown writing system.
- 1795—Thomas Jefferson invents the Jefferson disk cipher.
- 1854—Playfair cipher invented by Charles Wheatstone.
- 1919—Edward Hebern invents the first rotor machine.
- 1943—The cipher-breaking machine Heath Robinson is completed at Bletchley Park.
- 1974—The Feistel network block cipher design is developed by Horst Feistel.
- 1976—DES is approved as a standard and, in January 1977, is published as a FIPS standard for the US.
- 1977—The public-key cryptosystem RSA is invented.
- 1992—MD5 is first published.
- 1995—SHA-1 hash algorithm is published by the NSA.
- 1997—OpenPGP specification is released and a message encrypted with DES is cracked for the first time.
- 2001—Rijndael algorithm is selected as the U.S. Advanced Encryption Standard.
- 2004—MD5 is shown to be vulnerable to collision attacks.
- 2013—Speck and Simon, which are a family of lightweight block ciphers, is publicly released by the NSA.
- 2014—The website TrueCrypt.org is suddenly and mysteriously taken down by its developers without explanation. Later audit by Steve Gibson found the departure of TrueCrypt to be well planned and concluded that the application is still safe to use.
- 2017—Google successfully performs a collision attack against SHA-1.

Chapter 2 Cryptographic Signatures

Cryptographic signatures provide a method for verifying the integrity of data. By signing data with a digital signature, anyone can verify the signature that then proves that the data originated from you. It also verifies that the information has not been altered after you signed it.

Generating and verifying signatures

When generating signatures, you might be tempted to use the `String.GetHashCode` method. This, however, is not recommended at all. `String.GetHashCode` is actually intended for use in collections based on a hash table.

- Don't serialize hash codes for storing inside a database.
- Hash codes are not suitable for use as a key for object retrieval in a keyed collection.
- Unequal objects can have identical hash codes.
- To generate cryptographically strong hashes, instead use:
`System.Security.Cryptography.HashAlgorithm` or
`System.Security.Cryptography.KeyedHashAlgorithm`.

To see how this works, we will create a hash using SHA256. We will then create a container that will store the public and private asymmetric keys. We will then sign and return the signed hash to the calling application.

To verify the signed message, we will specify the container to use along with the hash of the original message and the digital signature. A method called `VerifySignedMessage` will verify the validity of the transmitted message.



Note: *SHA256 is a hash algorithm—not an encryption algorithm.*

Start by creating a new console application in Visual Studio. Before you start to write any code, add the following namespaces to your console application.

Code Listing 1: Adding a New Namespace

```
using static System.Console;
using System.Security.Cryptography;
```

The `System.Security.Cryptography` namespace will provide the cryptographic services we need for this application. The namespace provides services such as message authentication, encoding and decoding of data, random number generation, hashing, etc. The `System.Console` static namespace we brought into scope allows us to type less code. As of C# 6.0, static classes can be brought directly into scope without us fully qualifying the static namespace.

Next, you need to add a class-scope property that will contain the container name. The function of the container name will be explained in more detail later.

Code Listing 2: Adding Container Name

```
public static string ContainerName { get; private set; }
```

Next, add a method called **SignMessage** to the **Program** class. This method accepts the original message as a parameter and will return the hashed value of this message via an **out** parameter.

Code Listing 3: Adding SignMessage Method

```
private static byte[] SignMessage(string message, out byte[] hashValue)
{
}
}
```

Inside this method, create the using statement in Code Listing 4.

Code Listing 4: SHA256Managed

```
using (SHA256 sha = SHA256Managed.Create())
{
}
}
```

As mentioned earlier, SHA256 is a hash algorithm. As the class name suggests, the hash size for SHA256 is 256 bits. SHA256 is an abstract class with SHA256Managed being the only implementation of the abstract class.

```

namespace System.Security.Cryptography
{
    //
    // Summary:
    //     Computes the System.Security.Cryptography.SHA256 hash for the input data.
    [ComVisible(true)]
    public abstract class SHA256 : HashAlgorithm
    {
        //
        // Summary:
        //     Initializes a new instance of System.Security.Cryptography.SHA256.
        protected SHA256();

        //
        // Summary:
        //     Creates an instance of the default implementation of System.Security.Cryptography.SHA256.
        //
        // Returns:
        //     A new instance of System.Security.Cryptography.SHA256.
        //
        // Exceptions:
        //     T:System.Reflection.TargetInvocationException:
        //         The algorithm was used with Federal Information Processing Standards (FIPS) mode
        //         enabled, but is not FIPS compatible.
        public static SHA256 Create();

        //
        // Summary:
        //     Creates an instance of a specified implementation of System.Security.Cryptography.SHA256.
        //
        // Parameters:
        //     hashName:
        //         The name of the specific implementation of System.Security.Cryptography.SHA256
        //         to be used.
        //
        // Returns:
        //     A new instance of System.Security.Cryptography.SHA256 using the specified implementation.
        //
        // Exceptions:
        //     T:System.Reflection.TargetInvocationException:
        //         The algorithm described by the hashName parameter was used with Federal Information
        //         Processing Standards (FIPS) mode enabled, but is not FIPS compatible.
        public static SHA256 Create(string hashName);
    }
}

```

Figure 1: SHA256 Abstract Class

You will notice that the abstract class inherits from **HashAlgorithm** and exposes an overloaded **Create()** method. In fact, the inheritance chain looks like Figure 2.

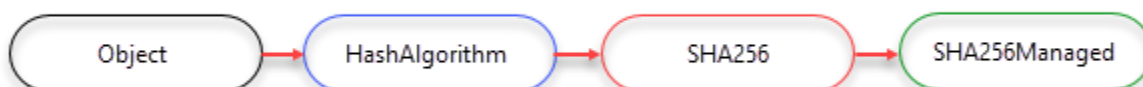


Figure 2: Inheritance

Swinging back to our code example, now inside the using statement, add the code from Code Listing 5.

Code Listing 5: SignMessage Method

```
hashValue = sha.ComputeHash(Encoding.UTF8.GetBytes(message));  
  
CspParameters csp = new CspParameters();  
  
csp.KeyContainerName = ContainerName;  
  
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(csp);  
  
var formatter = new RSAPKCS1SignatureFormatter(rsa);  
  
formatter.SetHashAlgorithm("SHA256");  
  
return formatter.CreateSignature(hashValue);
```

The hash value is created by calling the **ComputeHash()** method, which has now cryptographically hashed the message we passed to it. The **RSACryptoServiceProvider** and **DSACryptoServiceProvider** classes provide asymmetric encryption. In the previous code listing, a new public/private key pair was generated for us the moment we instantiated a new instance of the **RSACryptoServiceProvider**.

As a rule, asymmetric private keys must never be stored locally or in plain text on the local machine. If you must store a private key, you must use a **key container**. When we instantiate a new instance of the **CspParameters** class, we create such a key container by passing the name you want to use to the **KeyContainerName** field. This is where we make use of the **ContainerName** field created earlier.

We are now ready to create the signature using the **RSAPKCS1SignatureFormatter** and return that to the calling code. The message we passed to the **SignMessage()** method is now hashed and digitally signed. The full **SignMessage()** method should look like Code Listing 6.

Code Listing 6: Full SignMessage Method

```
private static byte[] SignMessage(string message, out byte[] hashValue)  
{  
    using (SHA256 sha = SHA256Managed.Create())  
    {  
        hashValue = sha.ComputeHash(Encoding.UTF8.GetBytes(message));  
        CspParameters csp = new CspParameters();  
        csp.KeyContainerName = ContainerName;  
        RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(csp);  
  
        var formatter = new RSAPKCS1SignatureFormatter(rsa);  
        formatter.SetHashAlgorithm("SHA256");  
    }  
}
```

```

        return formatter.CreateSignature(hashValue);
    }
}

```

The hashed message can be verified by the **VerifySignedMessage()** method. Note that the **RSAPKCS1SignatureDeformatter** object does the verification of the digital signature.

Code Listing 7: Verify Digital Signature

```

private static bool VerifySignedMessage(byte[] hash, byte[] signature)
{
    CspParameters csp = new CspParameters();
    csp.KeyContainerName = ContainerName;
    using (RSACryptoServiceProvider rsa = new
    RSACryptoServiceProvider(csp))
    {
        var deformatter = new RSAPKCS1SignatureDeformatter(rsa);
        deformatter.SetHashAlgorithm("SHA256");
        return deformatter.VerifySignature(hash, signature);
    }
}

```

In order to verify the digital signature, you will need:

- The public key from the person who signed the message.
- The actual digital signature generated.
- The hash algorithm that the person used to sign the message.
- The hash value of the signed message.

In this instance, we are the signer of the message. The public key we used to sign the data is safely stored inside the container we created.



Note: *We have not encrypted the message. We have only generated a hash in order to validate that the message hasn't been tampered with between the time it was sent and the time it was received.*

To implement this digital signature, let's give our container the name of **KeyContainer** and get the signature for a message we generate. The code will then verify the digital signature that will obviously pass the validation.

Code Listing 8: Implementation

```

public static string ContainerName { get; private set; }

static void Main(string[] args)
{

```

```

    ContainerName = "KeyContainer";
    string message = $"This secure message was generated on
{DateTime.Now}";
    byte[] signature = SignMessage(message, out byte[] hash);

    if (VerifySignedMessage(hash, signature))
        WriteLine($"Message '{message}' is valid");
    else
        WriteLine($"Message '{message}' is not valid");

    ReadLine();
}

```

Running the application will result in the output in Figure 3.

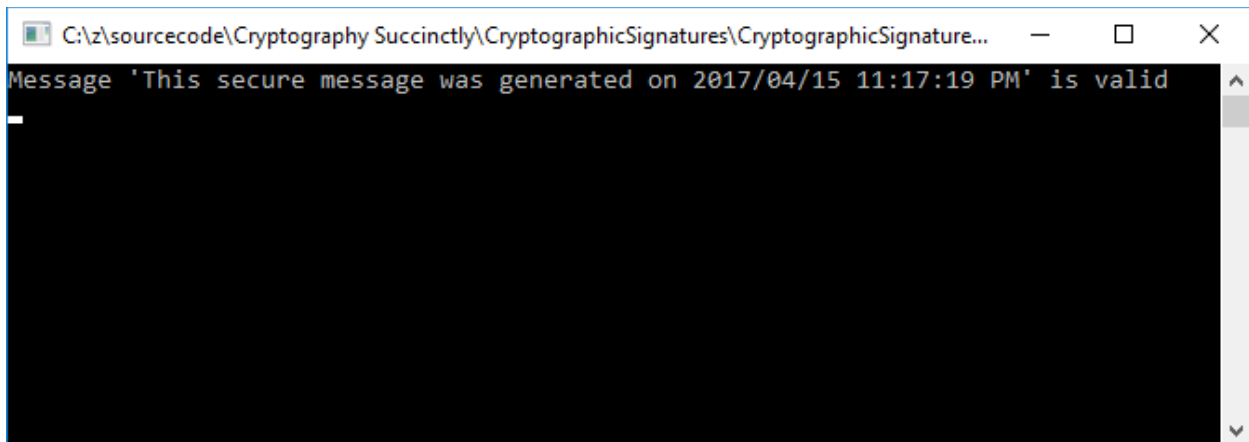


Figure 3: Output Result

The output from the previous code is obvious. So, let's have a look at a more practical example. In the following code, we will not go into detail regarding passing around the public key. We will simply use it in the created container.

The situation can be illustrated with the diagram in Figure 4.

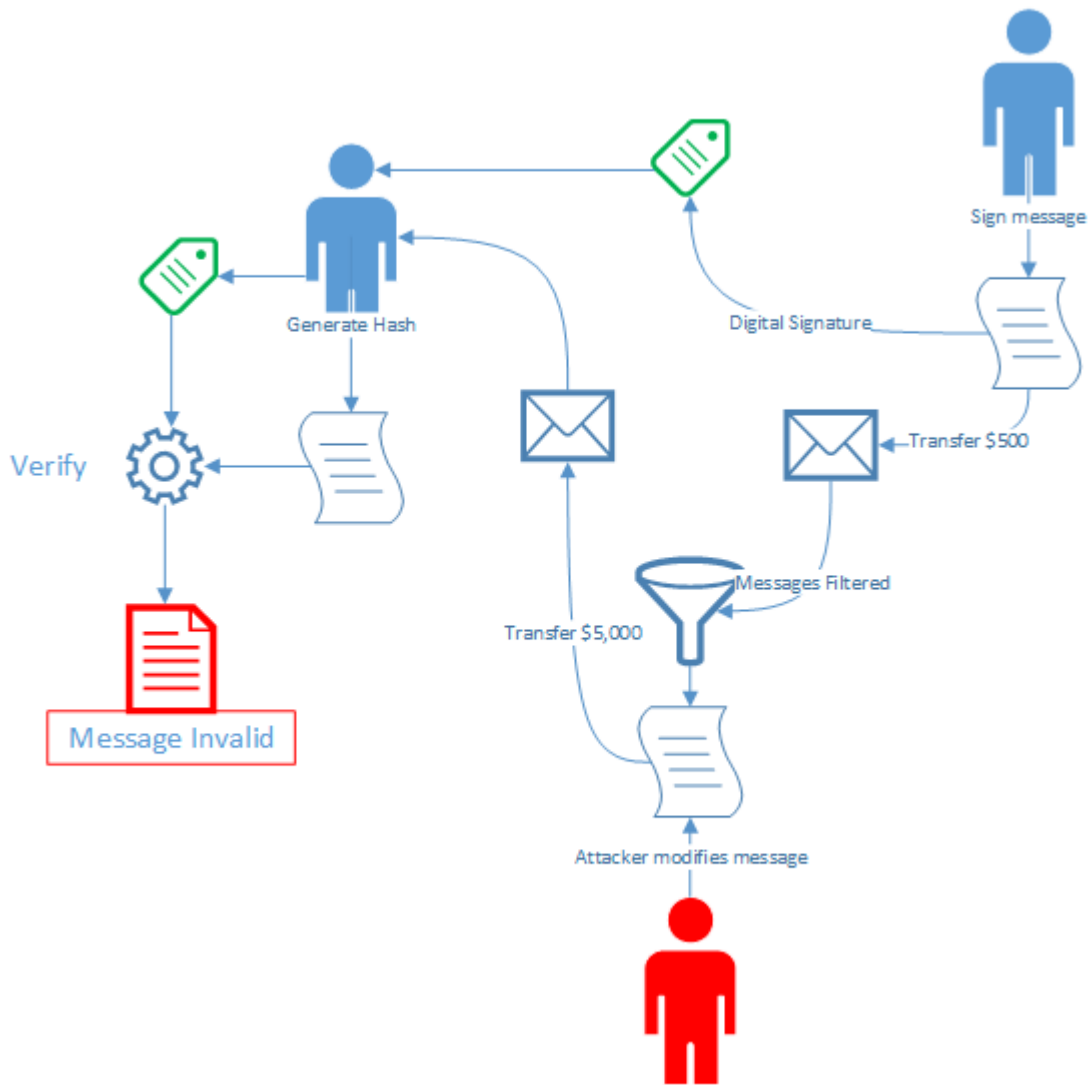


Figure 4: Signing a Message

Imagine that you are sending me a message to transfer \$500 into your bank account with the provided account number. The message clearly states the amount and the account number. Before you send me the message, however, you sign it and send the message to me along with the digital signature.

Code Listing 9: Original Message Signed

```
// Sender
string secureMessage = $"Transfer $500 into account number 029192819283
on {DateTime.Now}";
byte[] digitalSignature = SignMessage(secureMessage);
```

The method used to sign the message is the same as the one we used previously. This time it only returns the digital signature.

Code Listing 10: Create Digital Signature

```
private static byte[] SignMessage(string message)
{
    using (SHA256 sha = SHA256Managed.Create())
    {
        byte[] hashValue =
        sha.ComputeHash(Encoding.UTF8.GetBytes(message));
        CspParameters csp = new CspParameters();
        csp.KeyContainerName = ContainerName;
        RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(csp);

        var formatter = new RSAPKCS1SignatureFormatter(rsa);
        formatter.SetHashAlgorithm("SHA256");
        return formatter.CreateSignature(hashValue);
    }
}
```

Unfortunately, somewhere between sending the message to me and my receipt of it, somebody has intercepted the message and changed the content. The amount is changed from \$500 to \$5,000, and the bank account is changed to the attacker's bank account number.

Code Listing 11: Message Intercepted

```
// Message intercepted
secureMessage = $"Transfer $5000 into account number 849351278435 on
{DateTime.Now}";
```

When the modified message reaches me, I use the **ComputeMessageHash()** extension method to compute the hash for the message. Next, I pass the hash I computed to the **VerifySignedMessage()** method along with your digital signature to check the authenticity of the message.

Code Listing 12: Compute Hash and Verify Message

```
// Recipient
byte[] messageHash = secureMessage.ComputeMessageHash();
if (VerifySignedMessage(messageHash, digitalSignature))
    WriteLine($"Message '{secureMessage}' is valid and can be trusted.");
else
    WriteLine($"The following message: '{secureMessage}' is not valid. DO NOT TRUST THIS MESSAGE!");
```

The **ComputeMessageHash()** method is not complex and simply returns the hash for the message received.

Code Listing 13: ComputeMessageHash Method

```
public static class ExtensionMethods
{
    public static byte[] ComputeMessageHash(this string value)
    {
        using (SHA256 sha = SHA256.Create())
        {
            return sha.ComputeHash(Encoding.UTF8.GetBytes(value));
        }
    }
}
```

If you run the console application now, you will receive a very different message.

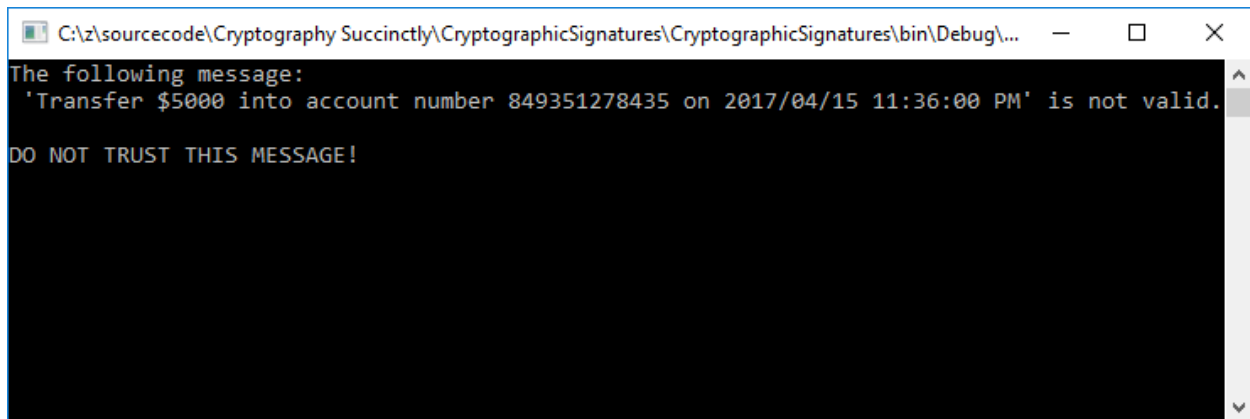


Figure 5: Received Message Invalid

You can see how crucial a role digital signatures can play in secure message transmission. Again, the use of digital signatures is not intended to obfuscate the message being sent but rather to verify the message's authenticity. If your intent is to send a message so that it can't be read by anyone intercepting it, you must encrypt the message.

We will have a look at doing that in a later chapter.

Why you shouldn't use `String.GetHashCode()`

As mentioned at the beginning of the chapter, `String.GetHashCode()` is not a suitable method for generating hashes. The value it returns will change, and it is platform dependent. This means that it will differ among the various versions of the .NET Framework. It will also differ between the 32-bit and 64-bit versions of the .NET Framework.

Lastly, the `GetHashCode()` method can return the same value for different strings. This makes it totally inefficient for the standard required to secure your data.

Chapter 3 Hashing and Salting Passwords

Developers must understand the encryption area of software development. It is even more essential that developers know how to implement it correctly. Unfortunately, too often, sensitive data is not handled or stored correctly. The fact is that, at some point in your life, your personal information will most likely be compromised because the database it was stored on was hacked and dumped.

The issue (for developers, anyway) is not the fact that the database could be compromised. The real issue is how secure the data is that you write **to** the database.

Before we look at how to encrypt data, let's quickly look at the mechanism we use to get the data into the database.



Note: *Symmetric encryption is used on streams of large data while asymmetric encryption is performed on small amounts of data.*

I recently had a difference of opinion with a member of a forum I contribute to. People there can pose questions (no, it's not Stack Overflow), then members of the site can answer those questions and provide support.

In this case, the question related to writing data to a database. A forum member suggested a solution that was in line with what the author of the question had initially written. The problem was that the SQL was inline and not parametrized. This is bad practice. End of story. There is no debate. The member of the site who answered the question did not assist the author of the question in implementing a best practice. His argument was that he assisted the author of the question with the problem at hand, which was that the insert statement wasn't working. While I can understand this, there is no excuse for assisting the author of the question without telling him that his implementation is bad practice. And, more worrying was the fact that the question's author accepted the solution, which was an implementation of the insert statement that we all know is bad practice.

You as a developer have a responsibility to keep the data as secure as possible for as long as possible, up until it is written to the database. In other words, do not do what we see in Code Listing 14.

Code Listing 14: Bad SQL Insert Logic

```
private void BadSQLExample(string name, string surname, string username,
string password, string email)
{
    StringBuilder sb = new StringBuilder();

    sb.Append(" INSERT INTO [dbo].[UserTable]");
    sb.Append("      ([Name]");
    sb.Append("      ,[Surname]");
    sb.Append("      ,[Username]");
```

```

sb.Append("        ,[Password]");
sb.Append("        ,[Email])");
sb.Append(" VALUES");
sb.Append($"        ('{name}');
sb.Append($"        ,'{surname}';
sb.Append($"        ,'{username}';
sb.Append($"        ,'{password}';
sb.Append($"        ,'{email}')");

// Perform data insert
}

```

Not only is this bad practice, but it will come back to bite you at some point in your career. Just ask the developers at a small development company who wrote an access control application for a client. When the client decided to have the code audited by an outside firm, the code failed miserably. The repercussions were disastrous for the development company involved.

Instead of the inline SQL, consider this coding approach below as better in terms of securing your code.

Code Listing 15: Good SQL Insert Logic

```

private void SaveUserData(string name, string surname, string username,
string password, string email)
{
    SqlCommand cmd = new SqlCommand();
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.CommandText = "spoc_InsertUserData";

    cmd.Parameters.Add("Name", SqlDbType.NVarChar, 50).Value = name;
    cmd.Parameters.Add("Surname", SqlDbType.NVarChar, 50).Value =
surname;
    cmd.Parameters.Add("Username", SqlDbType.NVarChar, 50).Value =
username;
    cmd.Parameters.Add("Password", SqlDbType.NVarChar, 50).Value =
password;
    cmd.Parameters.Add("Email", SqlDbType.NVarChar, 50).Value = email;

    ExecuteNonQuery(cmd);
}

```

Now that we have that out of the way, let's have a look at encrypting the data you write to the database. If the data you store in the database looks like Figure 6, you still have a problem.

Results		Messages				
	ID	Name	Surname	Username	Password	Email
1	1	Dirk	Strauss	dirkstrauss	6Xqi9zIFNeNXm9u#HcS0	dirk@email.com

Figure 6: User Table Data

The data is visible to anyone who cares to take a look. SQL administrators can see the information at will. Worst case scenario, if the database is ever hacked, your data is a free-for-all for anyone out there.

The problem is compounded by the use of:

- Weak passwords—developers need to ensure strong passwords in their systems.
- Single passwords across all sites—over this, you have no control.

Even though the password in the table above is quite strong, it is still visible to anyone who cares to look. Apart from using strong passwords, we need to encrypt the rest of the data. Let's have a look at how to accomplish this.

Encrypting user details

We want to take the users' personal data and store it in the database using a method called salting and hashing. We salt the data to be encrypted and, in doing so, make it more complex and more random. A common way to crack encrypted data is by using **lookup tables** or **rainbow tables**. By using a salt, we ensure that we add a truly random value to the password so that even if two users in the system have the same password, a truly random hash (encrypted value) is generated.



Note: Salt is random data that is added to the password before it is hashed.

In the town where I grew up, there was another Dirk Strauss. This bloke must have been a golf player, because I had people coming up to me (people I'd known for years) congratulating me for winning some golf tournament or other. I never met the guy, but consider the possibility of two users with the same password. If they use weak passwords, that possibility is all the more probable.

Table 1

User	Weak Password
Dirk Strauss (the real one)	d1Rk\$tr@u55
Dirk Strauss (the golfer)	d1Rk\$tr@u55

If both of us use the very weak password of **d1Rk\$tr@u55**, then it would not stand up very well against any type of attack. The possibility of coming across such a similar weak password is very likely, especially in a large organization or on a public website. In fact, I have recently seen passwords in a database (spot the error here? No encryption) entered as **ian** and **mark123** and, shockingly, simply **1**.



Tip: If you use this type of password, stop using it. Use an application such as LastPass to generate truly unique passwords while also managing those passwords for you across the Internet.

The encrypted password generated for me if salting is used during encryption:

- **User:** Dirk
- **Salt:** fxJis2foDGEEnmx6XTGNQNi3ECvxEDSYEsv/71ds17a0=
- **Encrypt:** fxJis2foDGEEnmx6XTGNQNi3ECvxEDSYEsv/71ds17a0+=d1Rk\$tr@u55
- **Hashed Value:** c/LUZdrhzkD8iOr6BLZORbawtYubcyggbsjleQceJ+4=

The encrypted password generated for the golfer if salting is used during encryption:

- **User:** Golfer
- **Salt:** icqlqXBkYQgerEfzQx/lm3/XdpsIHswR5CNiKeC35qs=
- **Encrypt:** icqlqXBkYQgerEfzQx/lm3/XdpsIHswR5CNiKeC35qs+=d1Rk\$tr@u55
- **Hashed Value:** FBxS85sYOPQX3hIcijiM8NHcnWWOibJamJeDrtCluYk=

As you can see, even though two different users use the same weak password, the encrypted passwords generated are totally different.



Tip: Always enforce strong passwords in your applications. Salting and hashing do not prevent passwords from being guessed.

Now that we have established that salting is essential, we must decide if we are going to concatenate the salt to the beginning or the end of the password. It does not matter if you add the salt at the beginning or the end, just ensure that whatever you decide (beginning or end), you do it consistently each time.

A few rules when working with salt:

- Never reuse the salt in each hash.
- Never hard code a value in the code to use as the salt.
- Don't use a short salt.
- Never use the username or any other entered data as the salt—it must be random.
- If the user ever changes their password (or any other secure info), generate a new salt.

A good practice is to generate a new salt for each value that you encrypt. You then store the salt with the encrypted data in each field of the database table. This means that you have a different random salt for every value, which makes the information you encrypt very secure.

In the examples below, the encrypted data is stored with a colon “:” separating the salt and encrypted value as **[encrypted value]:[salt value]**. You can see the old value for the username, as opposed to the newly encrypted and salted value for the username.

	Username
1	dirkstrauss
2	nsF+vEdplMV8djX+DPkMgd1OIKJJ/YXqzvN+bwYz0=:cPiZcsp5v7Or0tpmsP6xfNbVdc7+yEDUDFOMo0H0+3k=

Figure 7: Salted and Encrypted Username

If we look at the password field, you will notice that the salt used to encrypt the password is not the same salt used to encrypt the username. Compared to the old value stored for the password, the new encrypted and salted password is more secure.

	Password
1	6Xqi9zIFNeNXm9u#HcS0
2	+d9tmRI+JLGSrm/O6X1qR0kgs8l0syHc5Znkg1O/9bA=:AmpG518xf7sYYqiv4WuLGbwZf0A+1BoFAyWUC0PWMPU=

Figure 8: Salted and Encrypted Password

We therefore ensure that the data we encrypt is very secure. So how do we do this? Consider the example in Code Listing 16:

Code Listing 16: Encrypting Data

```
private static string EncryptData(string valueToEncrypt)
{
    string GenerateSalt()
    {
        RNGCryptoServiceProvider crypto = new RNGCryptoServiceProvider();
        byte[] salt = new byte[32];
        crypto.GetBytes(salt);

        return Convert.ToBase64String(salt);
    }

    string EncryptValue(string strvalue)
    {
        string saltValue = GenerateSalt();
        byte[] saltedPassword = Encoding.UTF8.GetBytes(saltValue +
strvalue);

        SHA256Managed hashstr = new SHA256Managed();
        byte[] hash = hashstr.ComputeHash(saltedPassword);
    }
}
```

```

        return $"{Convert.ToBase64String(hash)}:{saltValue}";
    }

    return EncryptValue(valueToEncrypt);
}

```

The method **EncryptData()** receives a value to encrypt (**valueToEncrypt**) and returns the encrypted value in the return statement.

Inside the **EncryptData()** method, there are two local functions called **GenerateSalt()** and **EncryptValue()**. The **GenerateSalt()** local function ensures that a unique salt value is created for every value you pass to the **EncryptValue()** local function. The encrypted value and its salt are returned to the calling code.



Note: *Local functions is a new feature of C# 7.*

To test the encryption code, simply create some hard coded values and pass them to the **EncryptData()** method. The string returned to the calling code will be the salted and encrypted value that you can store inside the database.

Code Listing 17: Testing Encryption

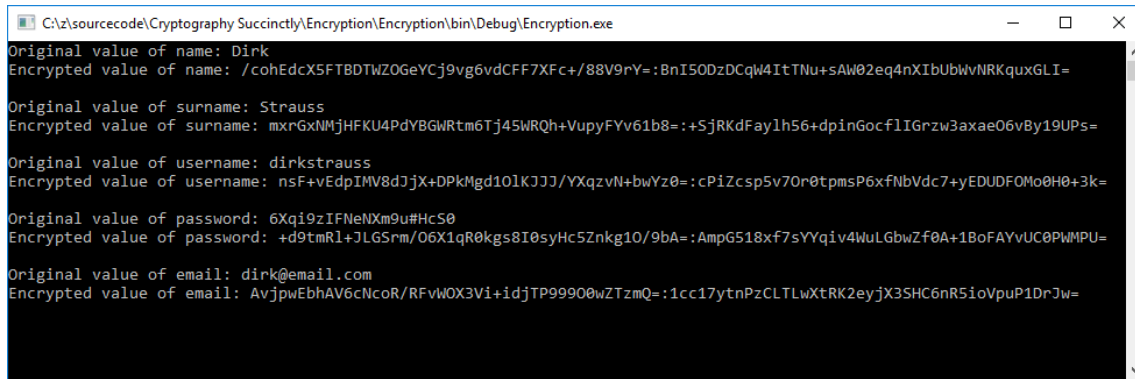
```

string name = "Dirk";
string surname = "Strauss";
string username = "dirkstrauss";
string password = "6Xqi9zIFNeNXm9u#HcS0";
string email = "dirk@email.com";

string encryptedName = EncryptData(name);
string encryptedSurname = EncryptData(surname);
string encryptedUsername = EncryptData(username);
string encryptedPassword = EncryptData(password);
string encryptedEmail = EncryptData(email);

```

Output these values to the console window and see the results, as in Figure 9.



```
CA:\sourcecode\Cryptography Succinctly\Encryption\bin\Debug\Encryption.exe
Original value of name: Dirk
Encrypted value of name: /cohEdcX5FTBDTWZ0GeYCj9vg6vdCFF7XFc+/88V9rY=:8nIS0DzDCqW4ItTNu+sAw02eq4nXIbUbWvNRKquxGLI=
Original value of surname: Strauss
Encrypted value of surname: mxrGxNMjHFKU4PdYBGWRtm6Tj45WRQh+VupyFYv61b8=:+SjRKdFaylh56+dpinGocf1IGrzw3axae06vBy19UPs=
Original value of username: dirkstrauss
Encrypted value of username: nsF+vEdpIMV8dJjX+DPkMgd10lKJJJ/YXqzvN+bwYz0=:cPiZcsp5v70r0tpmsP6xfNbVdc7+yEDUDF0Mo0H0+3k=
Original value of password: 6Xqi9zIFNeXm9u#HcS0
Encrypted value of password: +d9tmRl+JLGSrm/O6X1qR0kgs8I0syHc5Znkg10/9bA=:AmpG518xf7sYYqiv4WuLGbwZf0A+1BoFAYvUC0PWMPU=
Original value of email: dirk@email.com
Encrypted value of email: AvjpwEbhAV6cNcoR/RfvWOX3Vi+idjTP99900wZTzmQ=:1cc17ytnPzCLTLwXtRK2eyjX3SHC6nR5ioVpuP1DrJw=
```

Figure 9: Encrypted Results

Even if the database is compromised at some point, the data stored inside the database is still quite secure. Putting the code together using a stored procedure and parameters, we get the example in Code Listing 18.

Code Listing 18: Writing to the Database

```
private void SaveUserData(string name, string surname, string username,
string password, string email)
{
    SqlCommand cmd = new SqlCommand();
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.CommandText = "spoc_InsertUserData";

    cmd.Parameters.Add("Name", SqlDbType.NVarChar, -1).Value =
EncryptData(name);
    cmd.Parameters.Add("Surname", SqlDbType.NVarChar, -1).Value =
EncryptData(surname);
    cmd.Parameters.Add("Username", SqlDbType.NVarChar, -1).Value =
EncryptData(username);
    cmd.Parameters.Add("Password", SqlDbType.NVarChar, -1).Value =
EncryptData(password);
    cmd.Parameters.Add("Email", SqlDbType.NVarChar, -1).Value =
EncryptData(email);

    ExecuteNonQuery(cmd); // Implementation omitted
}
```

You will notice that I have used **nvarchar(max)** in the database fields, therefore the **-1** is used as the size of the field.

Code Listing 19: Parameter

```
cmd.Parameters.Add("Password", SqlDbType.NVarChar, -1).Value =
EncryptData(password);
```

Validating user details

Now that we have ensured that the data stored in the database is secure, what do we do when we need to validate that the username or password typed in by the user is correct? A rule of thumb is that you should never encrypt the data stored in the database. Instead, you encrypt the data entered at a login page and compare that to the encrypted data stored in the database. For example, if the two values match each other, the user has entered the correct password.

In the `EncryptData()` method, we returned the encrypted data as `($"{Convert.ToBase64String(hash)}:{saltValue}");` where the `hash` is the encrypted data and the `saltValue` is the salt we used during encryption. We separate the encrypted data and the salt by using a `:` character. We need to keep this in mind when we validate the user input. Consider the `ValidateEncryptedData()` method.

Code Listing 20: Validate User Input

```
private static bool ValidateEncryptedData(string valueToValidate, string
valueFromDatabase)
{
    string[] arrValues = valueFromDatabase.Split(':');
    string encryptedDbValue = arrValues[0];
    string salt = arrValues[1];

    byte[] saltedValue = Encoding.UTF8.GetBytes(salt + valueToValidate);

    SHA256Managed hashstr = new SHA256Managed();
    byte[] hash = hashstr.ComputeHash(saltedValue);

    string enteredValueToValidate = Convert.ToBase64String(hash);

    return encryptedDbValue.Equals(enteredValueToValidate);
}
```

The encrypted string is split using the `:` character. Because we stored the values as `[encrypted value]:[salt value]` it means that `arrValues[0]` will contain the encrypted value and `arrValues[1]` will contain the salt.

Also remember that when we encrypted the data, we prefixed the value to encrypt with the salt value as follows: `saltValue + valueToEncrypt`. This means that we need to be consistent when validating the entered data.

Inside the `ValidateEncryptedData()` method, we then must salt the value to validate in the same way.

Code Listing 21: Order of Salt and Value

```
byte[] saltedValue = Encoding.UTF8.GetBytes(salt + valueToValidate);
```

We follow the same essential process as when we were encrypting the data. The value the user entered is salted with the same salt that the original value was salted with and then encrypted. We then do a string comparison of the encrypted value read from the database and the encrypted value the user entered.

If two encrypted values match, the data was entered correctly and you can validate the user. Testing the code can be done as in Code Listing 22.

Code Listing 22: Validating User Password

```
string name = "Dirk";
string surname = "Strauss";
string username = "dirkstrauss";
string password = "6Xqi9zIFNeNXm9u#HcS0";
string email = "dirk@email.com";

string encryptedName = EncryptData(name);
string encryptedSurname = EncryptData(surname);
string encryptedUsername = EncryptData(username);
string encryptedPassword = EncryptData(password);
string encryptedEmail = EncryptData(email);

if (ValidateEncryptedData(password, encryptedPassword))
    WriteLine($"The {nameof(password)} entered is correct");
else
    WriteLine($"The {nameof(password)} entered is not correct");
```

Place a breakpoint just before the entered password must be validated. Hover over the **password** variable and pin it.



The screenshot shows a code editor with a C# file. A breakpoint is set on the `password` variable in the `password` assignment line. A tooltip is visible over the `password` variable, showing its value: `"6Xqi9zIFNeNXm9u#HcS0"`. The `if` statement for validating the password is highlighted with a yellow background.

```
0 references
static void Main(string[] args)
{
    string name = "Dirk";
    string surname = "Strauss";
    string username = "dirkstrauss";
    string password = "6Xqi9zIFNeNXm9u#HcS0";
    string email = password; // password = "6Xqi9zIFNeNXm9u#HcS0"

    string encryptedName = EncryptData(name);
    string encryptedSurname = EncryptData(surname);
    string encryptedUsername = EncryptData(username);
    string encryptedPassword = EncryptData(password);
    string encryptedEmail = EncryptData(email);

    if (ValidateEncryptedData(password, encryptedPassword))
        WriteLine($"The {nameof(password)} entered is correct");
    else
        WriteLine($"The {nameof(password)} entered is not correct");
}
```

Figure 10: Testing Validation

Change the **password** variable value that you pinned to anything else.

```
0 references
static void Main(string[] args)
{
    string name = "Dirk";
    string surname = "Strauss";
    string username = "dirkstrauss";
    string password = "6Xqi9zIFNeNXm9u#HcS0";
    string email = password; // "wrongPassw0rD"

    string encryptedName = EncryptData(name);
    string encryptedSurname = EncryptData(surname);
    string encryptedUsername = EncryptData(username);
    string encryptedPassword = EncryptData(password);
    string encryptedEmail = EncryptData(email);

    if (ValidateEncryptedData(password, encryptedPassword))
        WriteLine($"The {nameof(password)} entered is correct");
    else
        WriteLine($"The {nameof(password)} entered is not correct");
}
```

Figure 11: Changing Password to Validate

Continue the debugging and you will see that the encrypted value of the **password** variable we changed did not match the encrypted value of the **encryptedPassword** variable. Validation therefore fails.

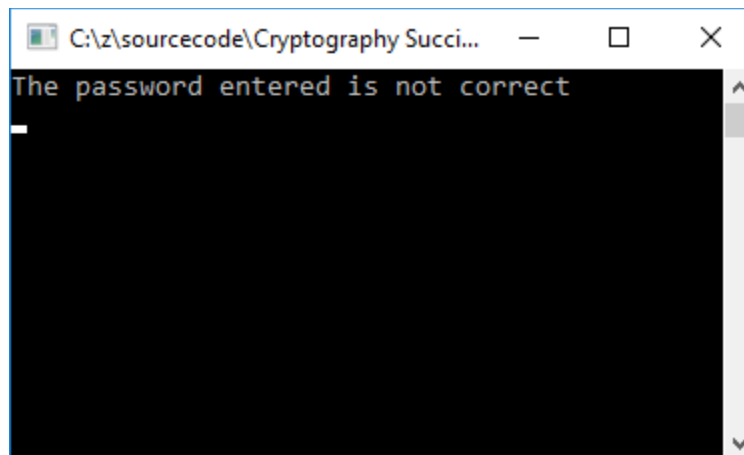


Figure 12: Password Not Validated

Encrypting user data using a salt and hash is essential if you are storing user login details.

How strong are unsalted passwords?

Just have a look at crackstation.net. You can test up to 20 nonsalted hashes using their free password hash cracker.

The crackstation.net folks created the lookup tables by scraping every word in the Wikipedia databases and adding as many password lists as they could find. The lookup tables then store a mapping between the hash of a password and the actual password.

The result? At the time of this writing, they had a 190 GB, 15 billion entry lookup table for MD5 and SHA1 hashes. For other hashes, they have a 19 GB, 1.5 billion entry lookup table.

Salt your data before encryption.

Chapter 4 Symmetric Encryption

Symmetric encryption is performed on streams and is useful when encrypting large amounts of data. A special stream class called **CryptoStream** is used to encrypt the data that is read into the stream.



Note: *Symmetric encryption uses the same key for encrypting and decrypting data.*

Streams that derive from the **Stream** class can be used with the **CryptoStream** class. The derived classes of the **Stream** class are:

- Microsoft.JScript.COMCharStream
- System.Data.OracleClient.OracleBFile
- System.Data.OracleClient.OracleLob
- System.Data.SqlTypes.SqlFileStream
- System.IO.BufferedStream
- System.IO.FileStream
- System.IO.MemoryStream
- System.IO.UnmanagedMemoryStream
- System.IO.Compression.DeflateStream
- System.IO.Compression.GZipStream
- System.IO.Pipes.PipeStream
- System.Net.Security.AuthenticatedStream
- System.Net.Sockets.NetworkStream
- System.Printing.PrintQueueStream
- System.Security.Cryptography.CryptoStream

To illustrate the use of symmetric encryption, we will read a large text file, encrypt the contents, and write it to an encrypted file.

Code Listing 23: Symmetric Encryption

```
public static void SymmetricEncryptionExample()
{
    string path = @"c:\temp\war_and_peace.txt";
    FileInfo fi = new FileInfo(path);
    string directory = fi.DirectoryName;

    string textToEncrypt = "";
    byte[] encryptedText;

    using (StreamReader sr = new StreamReader(path))
    {
        textToEncrypt = sr.ReadToEnd();
    }
    using (RijndaelManaged crypto = new RijndaelManaged())
```

```

    {
        crypto.GenerateKey();
        crypto.GenerateIV();
        ICryptoTransform encrypt = crypto.CreateEncryptor(crypto.Key,
crypto.IV);

        using (MemoryStream ms = new MemoryStream())
        {
            using (CryptoStream cryptStream = new CryptoStream(ms,
encrypt, CryptoStreamMode.Write))
            {
                using (StreamWriter writer = new
StreamWriter(cryptStream))
                {
                    writer.Write(textToEncrypt);
                }
                encryptedText = ms.ToArray();
            }

            using (FileStream fs = new FileStream(Path.Combine(directory,
"war_and_peace_encrypted.txt"), FileMode.Create, FileAccess.Write))
            {
                fs.Write(encryptedText, 0, encryptedText.Length);
            }
        }
    }
}

```

In the first section of the **SymmetricEncryptionExample()** method, we're not doing anything spectacular. We are simply opening a large text file and reading its contents into a string. We then create a **RijndaelManaged** object and generate a random symmetric algorithm key and a random initialization vector for the algorithm. We then create a **Rijndael** encryptor object for the key and initialization vector. Then we encrypt the text using a **CryptoStream** object.

Lastly, we write the encrypted text to a new file.

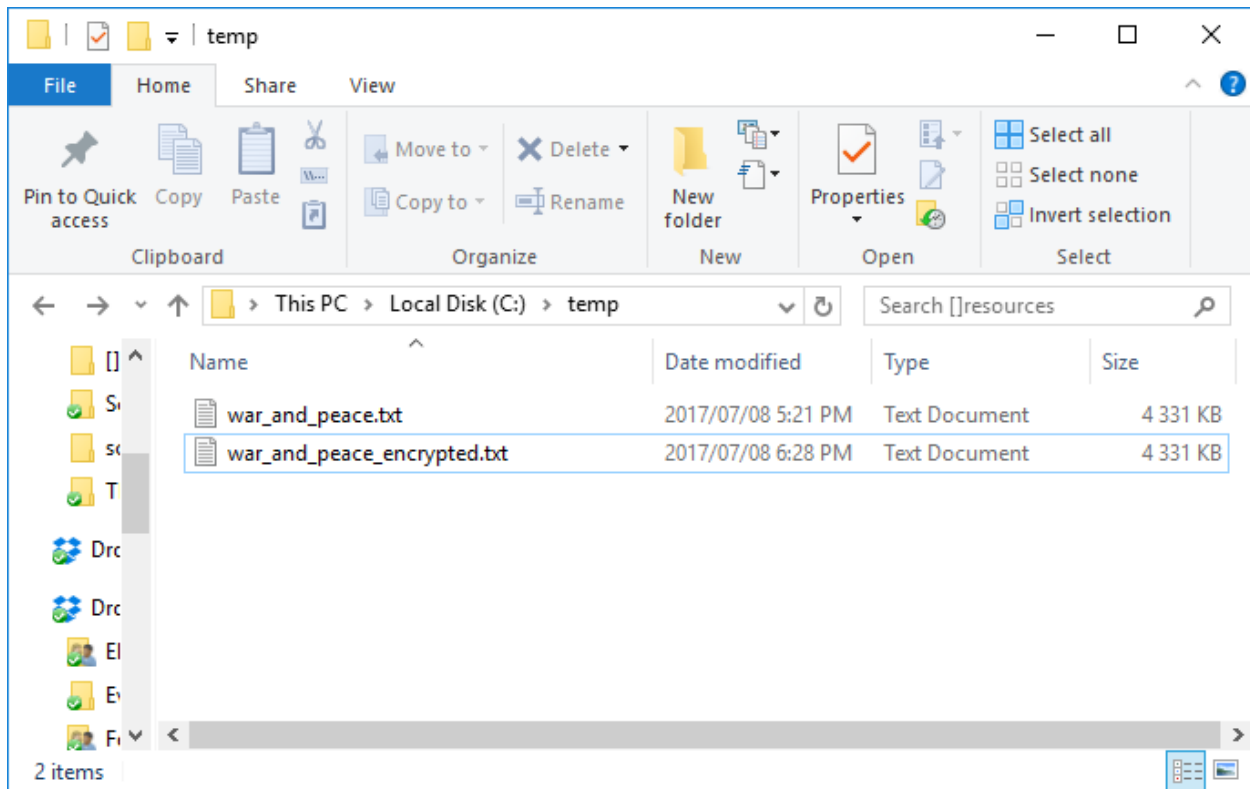


Figure 13: Encrypted File

If we have a look at the contents of the file, we can see that it has most definitely been encrypted.

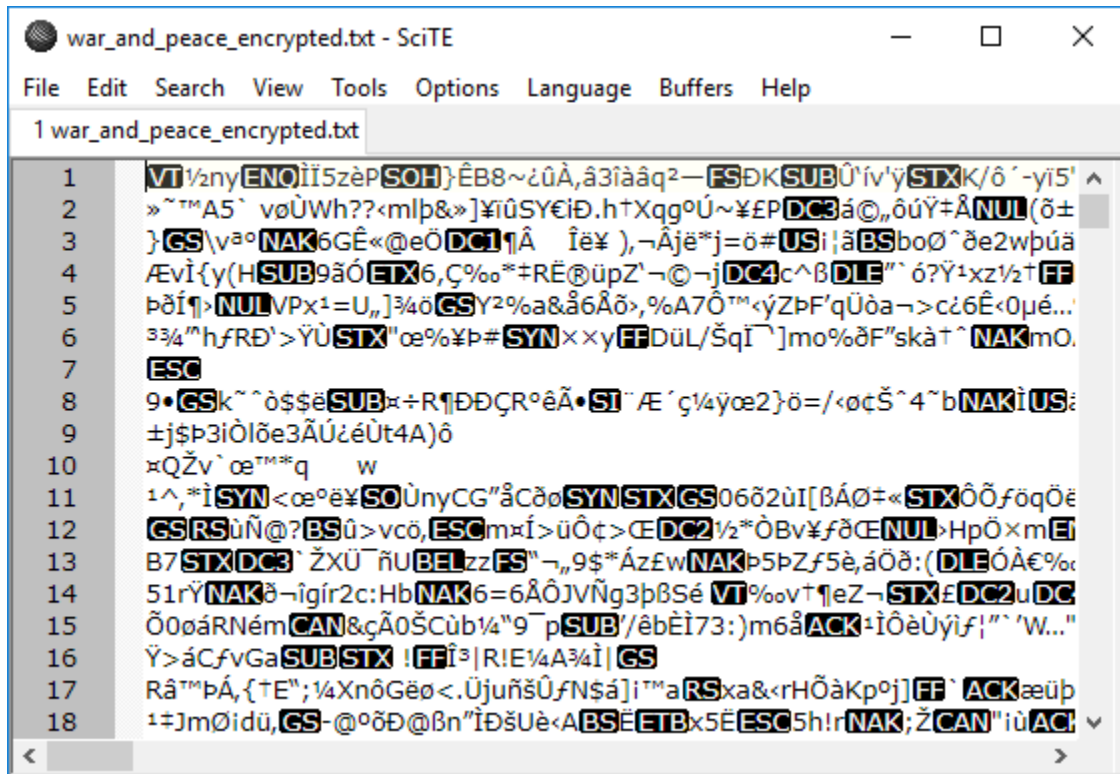


Figure 14: Encrypted File Contents

Decrypting the file back into plain text is also easy enough to do. We pass the method the encrypted text, the random symmetric algorithm key, and the random initialization vector for the algorithm. You will notice that this time the encrypted text is passed to the **MemoryStream**.

Code Listing 24: Decrypt the File

```
public static void DecryptFile(byte[] encryptedText, byte[] key, byte[]
initVector, string directory)
{
    string decryptedText;

    using (RijndaelManaged crypto = new RijndaelManaged())
    {
        crypto.Key = key;
        crypto.IV = initVector;

        ICryptoTransform decrypt = crypto.CreateDecryptor(crypto.Key,
crypto.IV);

        using (MemoryStream ms = new MemoryStream(encryptedText))
        {
            using (CryptoStream decryptStream = new CryptoStream(ms,
decrypt, CryptoStreamMode.Read))
            {
```

```
        using (StreamReader reader = new
StreamReader(decryptStream))
        {
            decryptedText = reader.ReadToEnd();
        }
        File.WriteAllText(Path.Combine(directory,
"war_and_peace_decrypted.txt"), decryptedText);
    }
}
```

We now also call the **CreateDecryptor** method on the **RijndaelManaged** object. The **CryptoStream** then decrypts the text and writes that to a file.

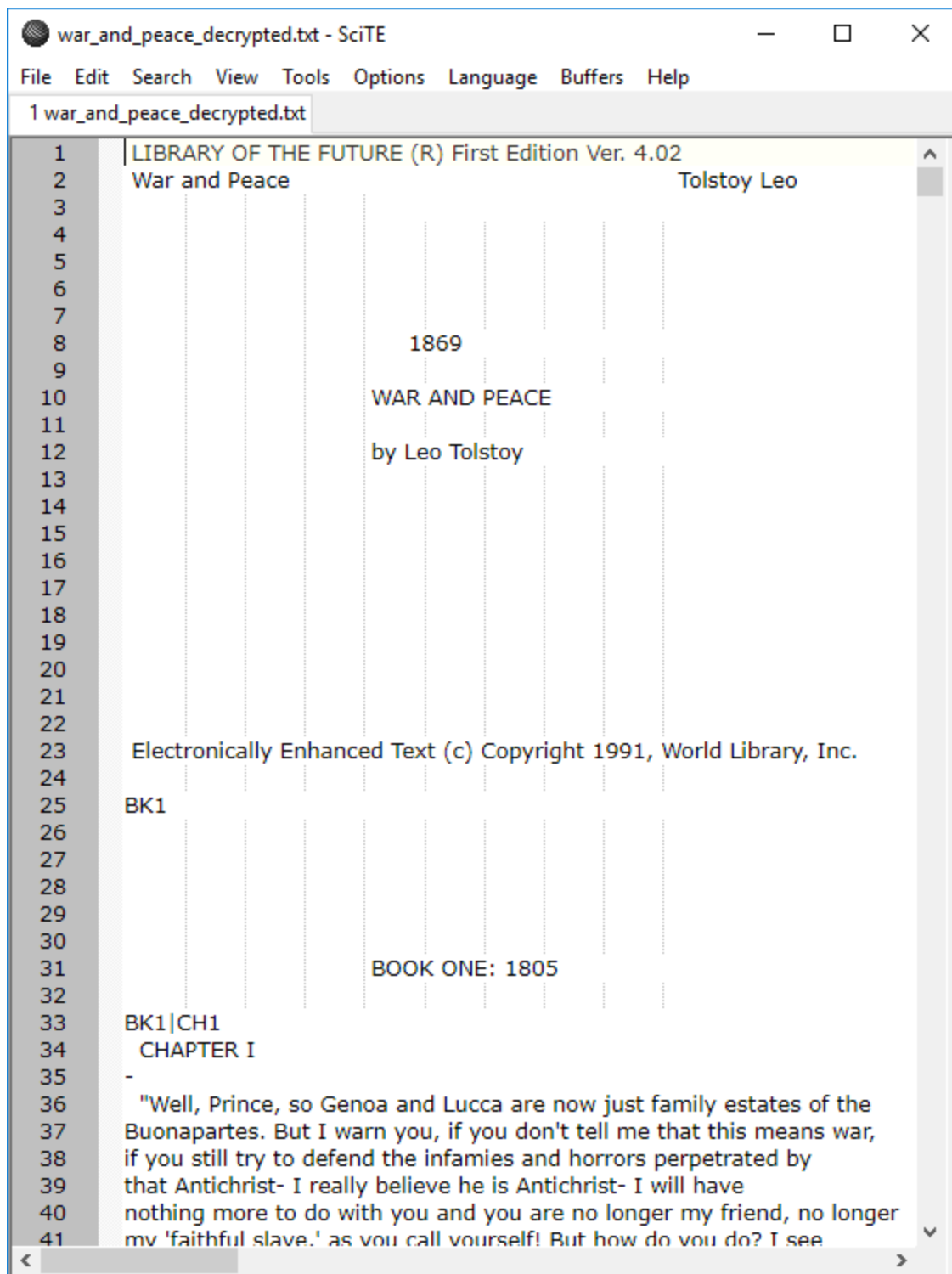


Figure 15: Decrypted File

As we can see, when the file is opened, it contains the decrypted text.

The generated key

When we called the `crypto.GenerateKey()` method on the `RijndaelManaged` object, we created a random byte array to store the encryption key. This key is used for both encryption and decryption of data.

The initialization vector

We also created an initialization vector by calling the `crypto.GenerateIV()` method on the `RijndaelManaged` object. It prevents data repetition and makes it difficult for hacking attempts to find patterns. A new random initialization vector is generated whenever the `GenerateIV()` method is called.

Chapter 5 Asymmetric Encryption

While symmetric encryption is performed on streams and is good for encrypting large amounts of data, asymmetric encryption is performed on small amounts of data. Imagine for a minute that you want to send me a secret message. How would you do that?

The following graphic summarizes the process of asymmetric encryption (also known as public key cryptography).

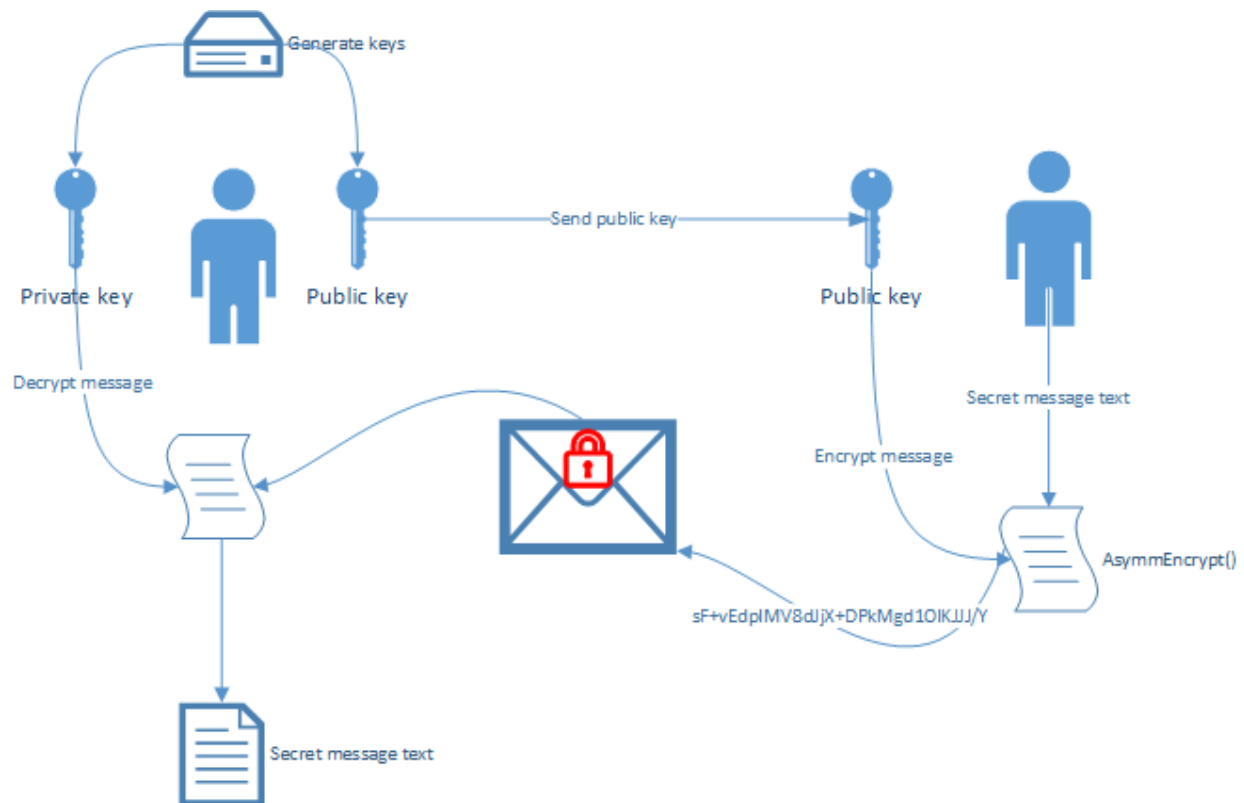


Figure 16: Asymmetric Encryption Summarized

You will see that the process starts on the left, where public and private keys are generated. You will also notice that the private key is never shared. Only the public key is shared between both parties.



Note: Asymmetric encryption uses mathematically linked public and private keys.

Let's see how to implement asymmetric encryption in code.

Writing the code

First, I generate a public and a private key that I store either in memory or in a cryptographic key container.



Note: The public key is the Modulus and Exponent.

Generating the public key is done as seen in Code Listing 25.

Code Listing 25: Generate Public and Private Keys

```
RSACryptoServiceProvider rsaCrypto = new RSACryptoServiceProvider();
RSAParameters RSAKeyInfo = rsaCrypto.ExportParameters(false);
byte[] publicMod = RSAKeyInfo.Modulus;
byte[] publicExp = RSAKeyInfo.Exponent;
```

Next, I will send you my public key. Using my public key, you encrypt the secret message and send the encrypted message back to me.

Code Listing 26: Encrypting the Message

```
byte[] toEncrypt = Encoding.ASCII.GetBytes("Secret message text to
encrypt");
byte[] encryptedData = AsymmEncrypt(toEncrypt, publicMod, publicExp);
```

The `AsymmEncrypt()` method encrypts the data you want to send me. Note that this code will be running on your computer, in the application you encrypt the message with. It is included here (Code Listing 27) in the same application for illustration purposes.

Code Listing 27: Asymmetric Encryption Method

```
public static byte[] AsymmEncrypt(byte[] dataToEncrypt, byte[] mod,
byte[] exp)
{
    RSACryptoServiceProvider crypto = new RSACryptoServiceProvider();
    RSAParameters RSAKeyInfo = new RSAParameters();
    RSAKeyInfo.Modulus = mod;
    RSAKeyInfo.Exponent = exp;

    crypto.ImportParameters(RSAKeyInfo);

    byte[] encryptedData;

    //Encrypt the data
    encryptedData = crypto.Encrypt(dataToEncrypt, false);

    return encryptedData;
```

```
}
```

After I receive the encrypted message from you, I decrypt it using the private key that corresponds to the public keys I sent you. I can only decrypt the message if I use the private key that corresponds to the public key you used to encrypt the secret message. If not, the decryption will fail.

Code Listing 28: Decrypt Secret Message

```
byte[] decrypted = rsaCrypto.Decrypt(encryptedData, false);  
string secretMessage = Encoding.Default.GetString(decrypted);
```

If we look at the output of the console application, we can see that the message was successfully decrypted.

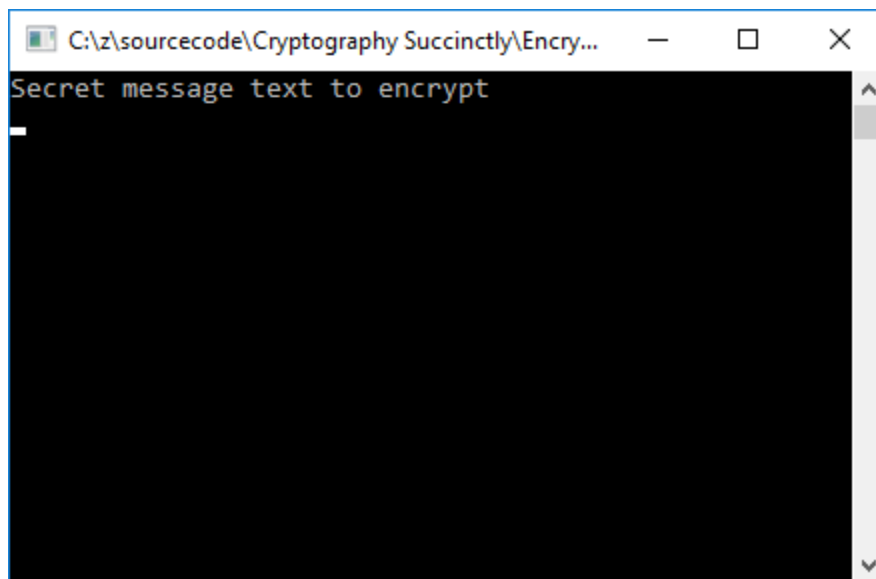


Figure 17: Decrypted Message

The message was encrypted and securely sent to me, where it was securely decrypted.

Chapter 6 Cryptographic Key Containers

We have seen how to generate a public and private key when using asymmetric encryption. While it is okay to share the public key (that's what it is there for, after all), the private key needs to be kept safe. The private key should never be stored in plain text on the hard drive. You must always use a key container if you want to store private keys.



Note: *For this code example, you must add `System.ValueTuple` from NuGet to your solution.*

In this chapter, we will take a look at the following:

- Creating a key container
- Encrypting a message
- Decrypting a message
- Deleting a key container
- Exporting a public key
- Importing a public key

At this point, I will wager a guess that some of you are wondering where exactly key containers are stored? Windows has a cryptographic key store, and it is simply located in a folder on your hard drive. On my Windows 10 machine, this path is **C:\ProgramData\Microsoft\Crypto** and inside that folder, there are various other folders for each key type. In this example, we will be looking at the **RSA\MachineKeys** subfolders.

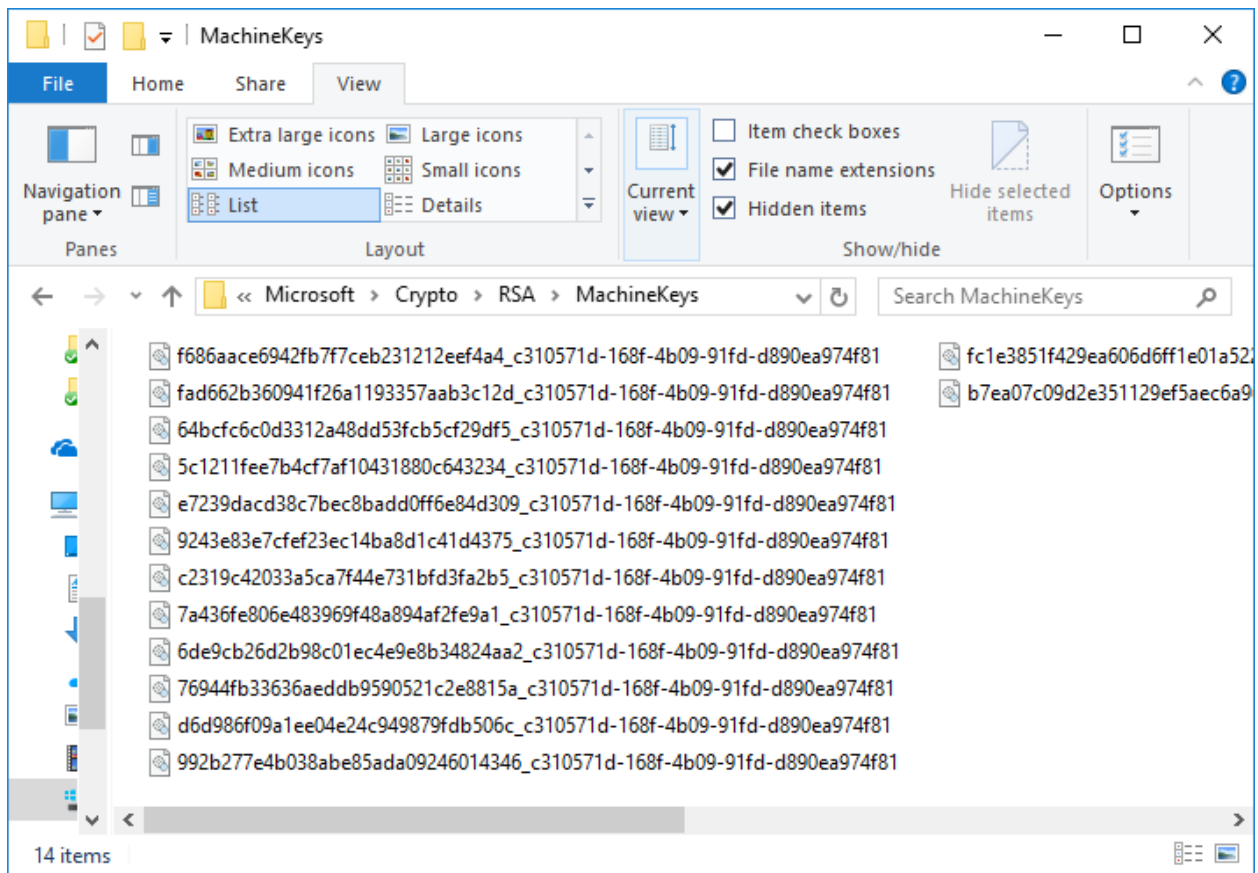


Figure 18: Windows 10 Cryptographic Key Store

If you navigate to this folder, you will see that there are various machine keys already created.

The other location where we can save key containers is the user profile. On my Windows 10 machine, this path is **C:\Users\<USER>\AppData\Roaming\Microsoft\Crypto** and inside that folder, there is an **RSA** subfolder with another subfolder with a GUID for a name.

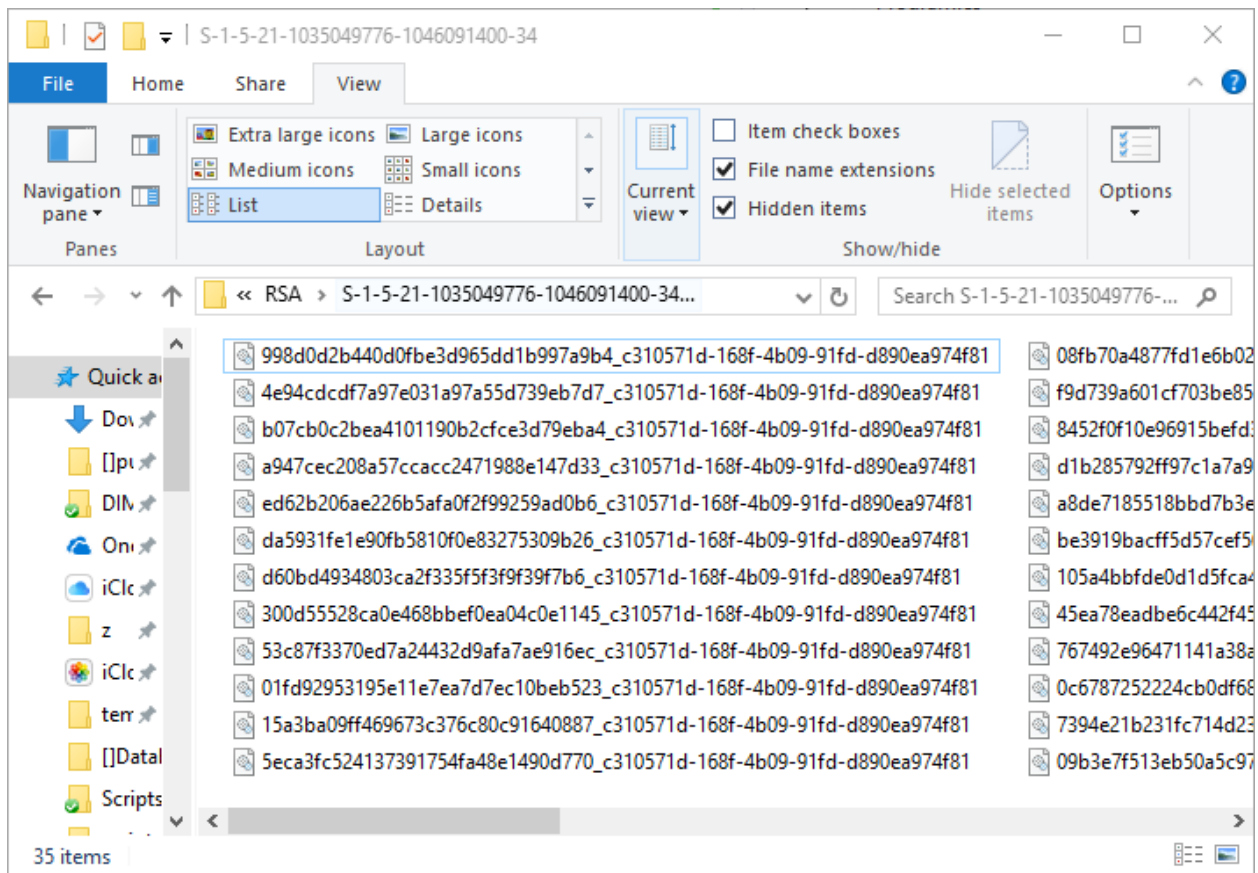


Figure 19: User Profile Key Store

It is here, in one of these two key stores, that we will be storing our private keys. Let's have a look at the key store under the user profile first.

Writing the code

Start off by creating the class-scope variables at the top of your code file, as seen in Code Listing 29.

Code Listing 29: Constants

```
const string KEY_STORE_NAME = "MyKeyStore";
public static CspParameters csp = new CspParameters();
public static RSACryptoServiceProvider rsa;
```

Next, create a method called **CreateAsmKeys()** and note the flag that tells the code to create the key container in the machine key store. The code is quite easy to understand. It simply creates a container with the name **MyKeyStore** and then instructs the **RSACryptoServiceProvider** object to persist the keys in the cryptographic service provider.

Note that I am also displaying the container name as well as the unique key container name created when it is saved in the key store.

Code Listing 30: Creating the Key Container

```
private static void CreateAsmKeys(out string containerName, bool
useMachineKeyStore)
{
    csp.KeyContainerName = KEY_STORE_NAME;
    if (useMachineKeyStore)
        csp.Flags = CspProviderFlags.UseMachineKeyStore;

    rsa = new RSACryptoServiceProvider(csp);
    rsa.PersistKeyInCsp = true;

    CspKeyContainerInfo info = new CspKeyContainerInfo(csp);
    WriteLine($"The key container name: {info.KeyContainerName}");
    containerName = info.KeyContainerName;

    WriteLine($"Unique key container name:
{info.UniqueKeyContainerName}");
}
```

The code to encrypt a message is also something we have seen before. We tell the cryptographic service provider (**CspParameters** object) what the container name is (I didn't use the constant here, as I explicitly wanted to give it a container name) and encrypt the message using the **RSACryptoServiceProvider**. You will notice that it is here that I return a tuple.

Code Listing 31: Encrypting a Message

```
public static (byte[] encrBytes, string encrString) AsymmEncrypt(string
message, string keyContainerName)
{
    CspParameters cspParams = new CspParameters()
    {
        KeyContainerName = keyContainerName
    };
    RSACryptoServiceProvider rsa = new
    RSACryptoServiceProvider(cspParams);
    byte[] encryptedAsBytes =
    rsa.Encrypt(Encoding.UTF8.GetBytes(message), true);
    string encryptedAsBase64 = Convert.ToBase64String(encryptedAsBytes);

    return (encryptedAsBytes, encryptedAsBase64);
}
```

The same logic is used in reverse. We tell the cryptographic service provider (**CspParameters** object) what the container name is and then use the **RSACryptoServiceProvider** to decrypt the message.

Code Listing 32: Decrypting a Message

```
public static (byte[] decrBytes, string decrString) DecryptWithCsp(byte[]
encrBytes, string containerName)
{
    CspParameters cspParams = new CspParameters()
    {
        KeyContainerName = containerName
    };
    RSACryptoServiceProvider rsa = new
    RSACryptoServiceProvider(cspParams);
    byte[] decryptBytes = rsa.Decrypt(encrBytes, true);
    string secretMessage = Encoding.Default.GetString(decryptBytes);

    return (decryptBytes, secretMessage);
}
```

To test the code, add the following logic to your console application's **static void Main** and run the console application. Note that the **blnUseMachineKeyStore** flag is set to **false**.

Code Listing 33: Code to Test

```
bool blnUseMachineKeyStore = false;
CreateAsmKeys(out string containerName, blnUseMachineKeyStore);

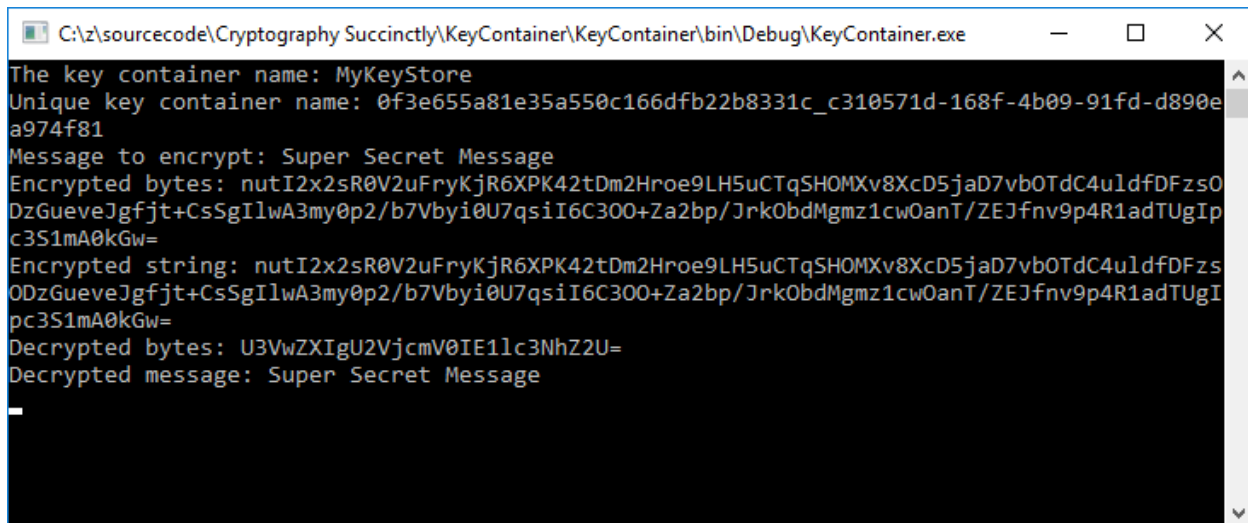
string superSecretText = "Super Secret Message";
WriteLine($"Message to encrypt: {superSecretText}");

var (encrBytes, encrString) = AsymmEncrypt(superSecretText,
containerName);
WriteLine($"Encrypted bytes: {Convert.ToBase64String(encrBytes)}");
WriteLine($"Encrypted string: {encrString}");

var (decrBytes, decrString) = DecryptWithCsp(encrBytes, containerName);
WriteLine($"Decrypted bytes: {Convert.ToBase64String(decrBytes)}");
WriteLine($"Decrypted message: {decrString}");
```

We can see a few things from the output generated. You can see that we used a key container name of **MyKeyStore**, but that the unique container name is quite different.

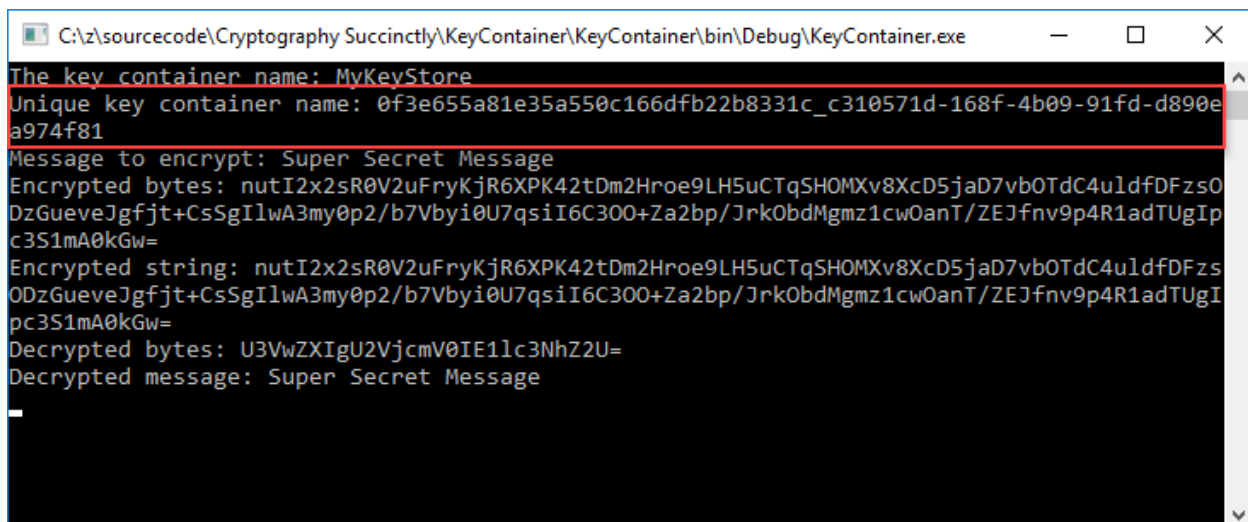
The message to encrypt was passed to the encryption method that returned the encrypted byte array and string. We then immediately decrypted the message using the key container created earlier. This resulted in the decrypted byte array and message.



```
C:\z\sourcecode\Cryptography Succinctly\KeyContainer\KeyContainer\bin\Debug\KeyContainer.exe
The key container name: MyKeyStore
Unique key container name: 0f3e655a81e35a550c166dfb22b8331c_c310571d-168f-4b09-91fd-d890ea974f81
Message to encrypt: Super Secret Message
Encrypted bytes: nutI2x2sR0V2uFryKjR6XPK42tDm2Hroe9LH5uCTqSHOMXv8XcD5jaD7vb0TdC4uldfDFzs0DzGueveJgfjt+CsSgIlwA3my0p2/b7Vbyi0U7qsiI6C300+Za2bp/JrkObdMgmz1cw0anT/ZEJfnnv9p4R1adTUGIp c3S1mA0kGw=
Encrypted string: nutI2x2sR0V2uFryKjR6XPK42tDm2Hroe9LH5uCTqSHOMXv8XcD5jaD7vb0TdC4uldfDFzs0DzGueveJgfjt+CsSgIlwA3my0p2/b7Vbyi0U7qsiI6C300+Za2bp/JrkObdMgmz1cw0anT/ZEJfnnv9p4R1adTUGI pc3S1mA0kGw=
Decrypted bytes: U3VwZXIgU2VjcmV0IE1lc3NhZ2U=
Decrypted message: Super Secret Message
```

Figure 20: Saved Keystore Output

Now let's focus for a minute on the unique key generated for the key container.



```
C:\z\sourcecode\Cryptography Succinctly\KeyContainer\KeyContainer\bin\Debug\KeyContainer.exe
The key container name: MyKeyStore
Unique key container name: 0f3e655a81e35a550c166dfb22b8331c_c310571d-168f-4b09-91fd-d890ea974f81
Message to encrypt: Super Secret Message
Encrypted bytes: nutI2x2sR0V2uFryKjR6XPK42tDm2Hroe9LH5uCTqSHOMXv8XcD5jaD7vb0TdC4uldfDFzs0DzGueveJgfjt+CsSgIlwA3my0p2/b7Vbyi0U7qsiI6C300+Za2bp/JrkObdMgmz1cw0anT/ZEJfnnv9p4R1adTUGIp c3S1mA0kGw=
Encrypted string: nutI2x2sR0V2uFryKjR6XPK42tDm2Hroe9LH5uCTqSHOMXv8XcD5jaD7vb0TdC4uldfDFzs0DzGueveJgfjt+CsSgIlwA3my0p2/b7Vbyi0U7qsiI6C300+Za2bp/JrkObdMgmz1cw0anT/ZEJfnnv9p4R1adTUGI pc3S1mA0kGw=
Decrypted bytes: U3VwZXIgU2VjcmV0IE1lc3NhZ2U=
Decrypted message: Super Secret Message
```

Figure 21: Unique Key Container Name

You will notice that we have a new file created in the local user profile key store. The file created matches the name of the unique key container name in the output.

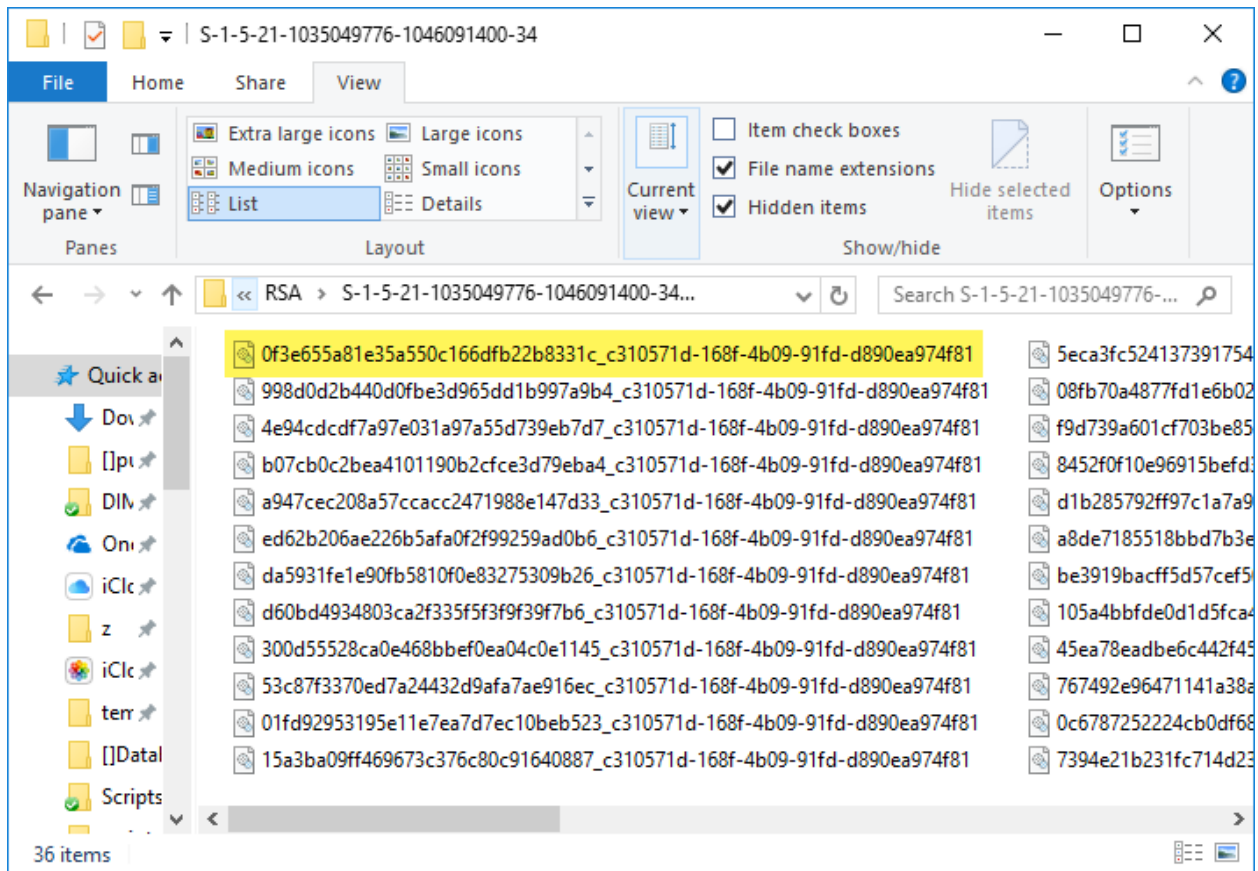


Figure 22: User Profile Key Container

Opening the file, you will see that the contents are encrypted.

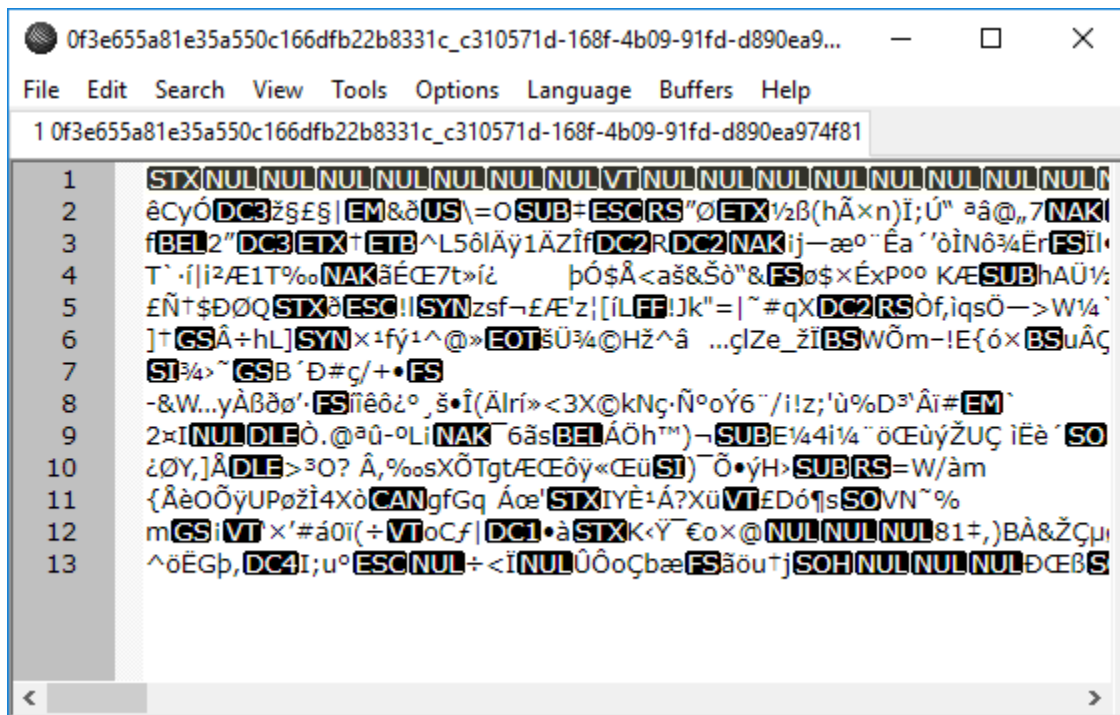


Figure 23: Encrypted Key Container File

Change the flag and let the code create the key store in the Windows key store. Run the application again.

Code Listing 34: Toggle Use Machine Key Store On

```
bool bInUseMachineKeyStore = true;
```

You will notice that the file is now created in the Windows key store.

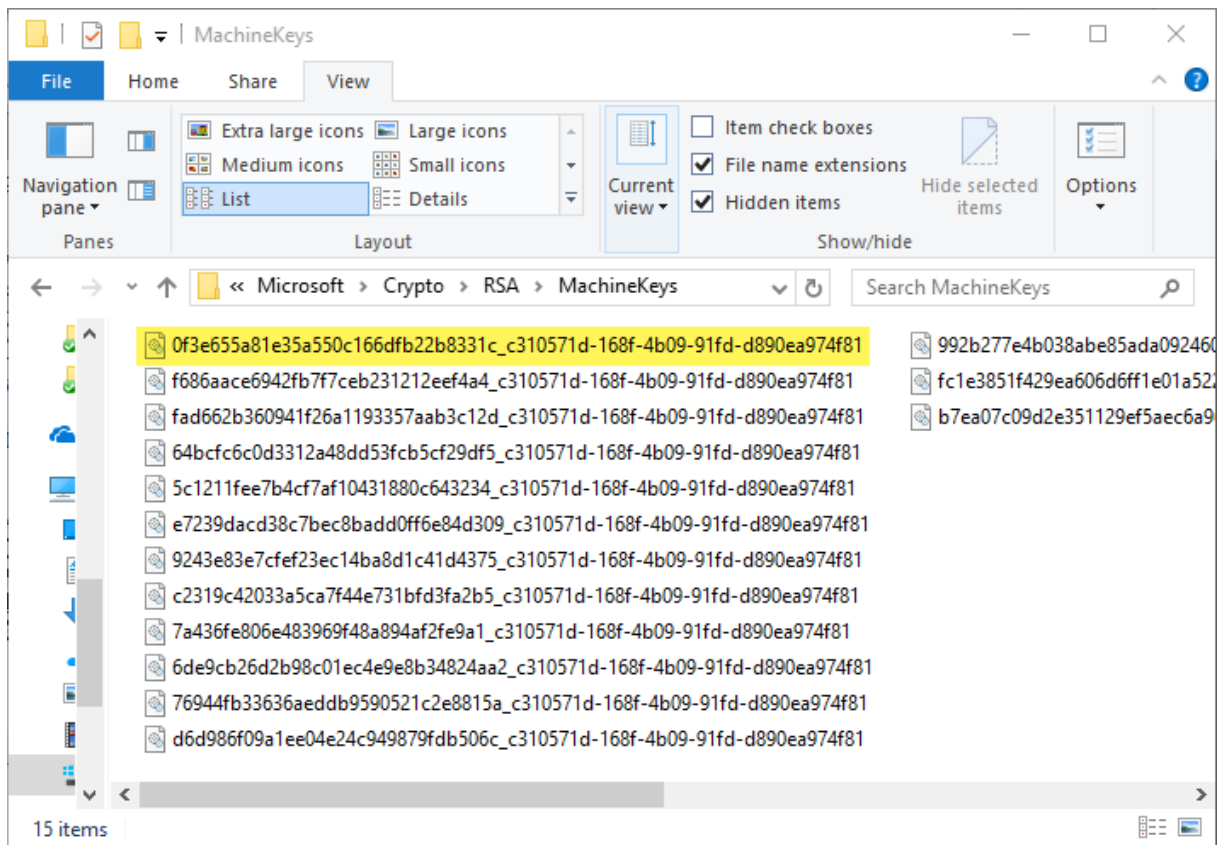


Figure 24: Windows Key Store

If we want to clear the key container, we only need to do the following, seen in Code Listing 35.

Code Listing 35: Delete the Key Container

```
private static void ClearContainer(string containerName, bool
useMachineKeyStore)
{
    csp = new CspParameters() { KeyContainerName = containerName };
    if (useMachineKeyStore)
        csp.Flags = CspProviderFlags.UseMachineKeyStore;
    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(csp) {
PersistKeyInCsp = false };
    rsa.Clear();
}
```

Keeping the **blnUseMachineKeyStore = true** will remove the key container from the Windows key store location.

Code Listing 36: Toggle Use Machine Key Store On

```
bool blnUseMachineKeyStore = true;
ClearContainer(KEY_STORE_NAME, blnUseMachineKeyStore);
```

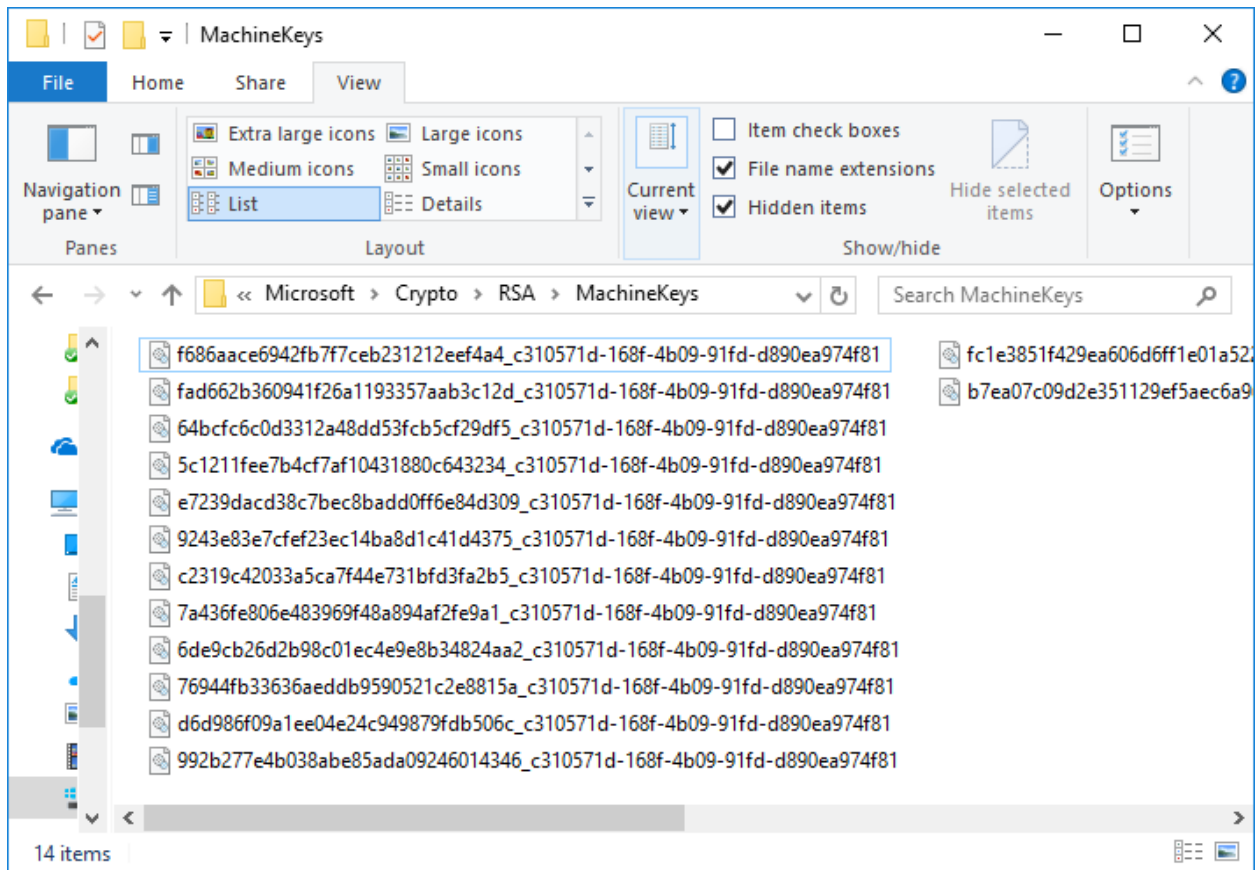



Figure 25: Container Removed from Windows Key Store

To remove the container from the user profile key store, just set the `blnUseMachineKeyStore` flag to `false` and run the code.

Code Listing 37: Toggle Use Machine Key Store Off

```
bool blnUseMachineKeyStore = false;
ClearContainer(KEY_STORE_NAME, blnUseMachineKeyStore);
```

Exporting the public key

Exporting the public key is very easy. We'll do this in order to give our public key to the person wanting to send us a message so that they can encrypt their message.

Add two constants to your code for the export folder and file name.

Code Listing 38: Public Key Export Location

```
const string EXPORT_FOLDER = @"C:\public_key\";
const string PUBLIC_KEY_FILE = @"rsaPublicKey.txt";
```

We then simply check if the directory exists and, if not, create it. We create a file in the export location, then we write the **RSACryptoServiceProvider** object xml to the file. This is done via the **ToXmlString(false)** method in which **false** tells the code not to include the private keys.

Code Listing 39: Export Public Keys Method

```
private static void ExportPublicKey()
{
    if (!(Directory.Exists(EXPORT_FOLDER)))
        Directory.CreateDirectory(EXPORT_FOLDER);

    using (StreamWriter writer = new
    StreamWriter(Path.Combine(EXPORT_FOLDER, PUBLIC_KEY_FILE)))
    {
        writer.Write(rsa.ToXmlString(false));
    }
}
```

Run the code in Code Listing 40 to test the export. First, create a container, then export the public keys.

Code Listing 40: Code to Test Export

```
CreateAsmKeys(out string containerName, blnUseMachineKeyStore);
ExportPublicKey();
```

You will see that the file is created in the location you provided.

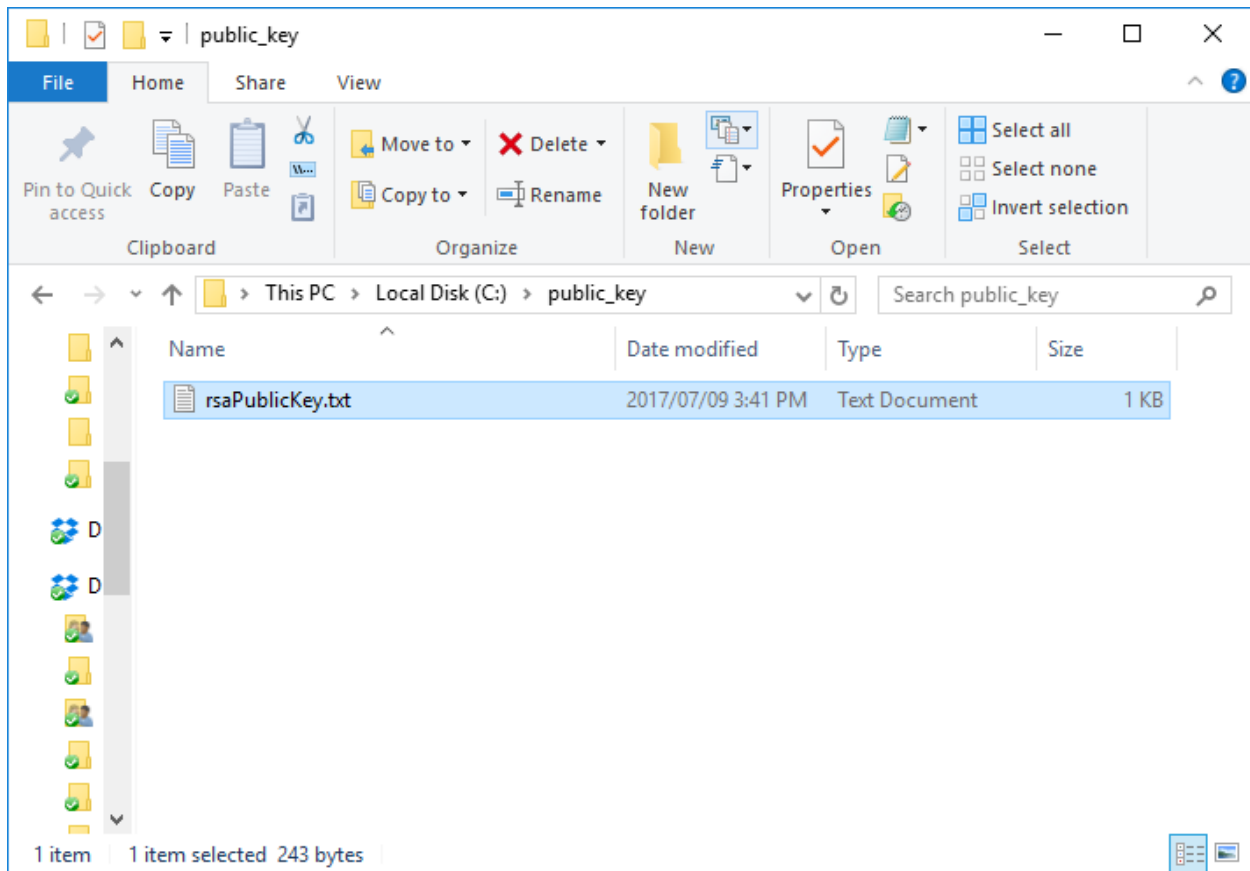


Figure 26: Exported Public Keys

Opening the file, you will notice that the contents are not encrypted at all. This is because the person you are sending the public key to needs to import it on their side and encrypt their message with it.

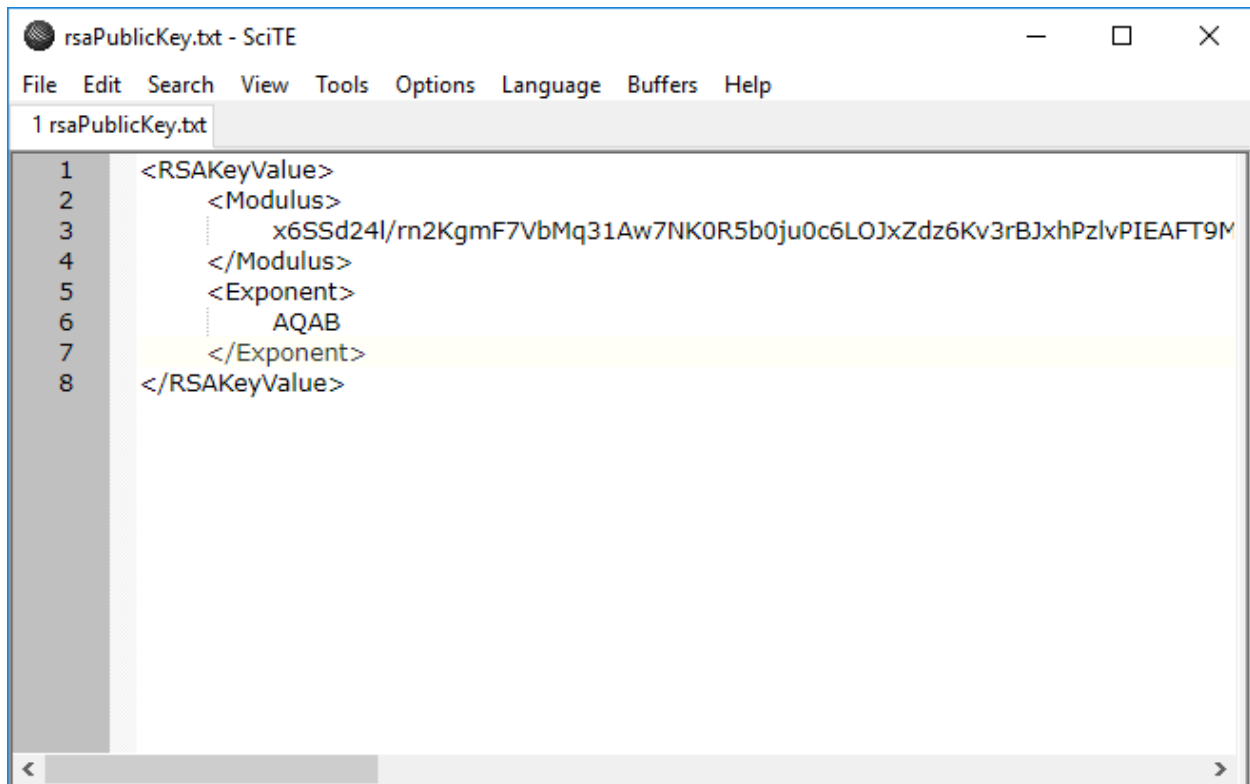


Figure 27: Exported Public Keys

If we change the code to include the private parameters, you will see a different file generated.

Code Listing 41: Include Private Parameters

```
writer.Write(rsa.ToXmlString(true));
```

Opening the unencrypted file will display in plain text the public and private key information. This is generally not something you will want to do.

```

1  <RSAKeyValue>
2    <Modulus>
3      uLe7EC+4W67pEczFpSPWUOgpiM3wOHbvdoAIVmy+gOFmYXHZMzhDaS0mBhPt
4    </Modulus>
5    <Exponent>
6      AQAB
7    </Exponent>
8    <P>
9      7ULU6SiA3kH2d9wNqBgZCoU10IUFc2m1zsmu20ECHXtepukhdwhVginKcFDshwbi
10   </P>
11   <Q>
12     x06HFWTQlaXOdSD6H2kqnZGBx2UvIAFyUw0q5n690CfyyniS53AtECnJn4A5j4XEC
13   </Q>
14   <DP>
15     xzKYLRVbdL8u3hz7vkhq+gczU98cD3UONopv6xceXoBIzJvpA9YUbI8HrtrSat9mF+
16   </DP>
17   <DQ>
18     X41qfftDdTt06/rPb9e/sqNcEPiWAuNeqCqb8r24yt8cK6364DKLqEAs/P9AmWqMnj)
19   </DQ>
20   <InverseQ>
21     V4TSOe8JIGLArOEqH3E+wHRwVLpe/uJIGcKlOvrXv1sM8LzIIEq7OxD8AYMCeDq7I
22   </InverseQ>
23   <D>
24     HfcUPgZ3wkvIHG6bG3SKVlqrLjm2P/qvUwPnD9oZN2AZUzJ8WS5F5UCBPBgpou4
25   </D>
26 </RSAKeyValue>

```

Figure 28: Exported Private and Public Keys

Importing a public key

The person you will be sending the public key to will need to import the public key into their application. If you are exchanging encrypted messages with each other, you will probably also need a method to import their public key if you ever want to send them a message.

The method to import logic simply imports the text read from the public key file sent to you into the **CspParameters** object and persists it.

Code Listing 42: Importing a Public Key

```

private static void ImportPublicKey()
{
    FileInfo fi = new FileInfo(Path.Combine(IMPORT_FOLDER,
PUBLIC_KEY_FILE));

```

```

    if (fi.Exists)
    {
        using (StreamReader reader = new
StreamReader(Path.Combine(IMPORT_FOLDER, PUBLIC_KEY_FILE)))
        {
            csp.KeyContainerName = KEY_STORE_NAME;
            rsa = new RSACryptoServiceProvider(csp);
            string publicKeyText = reader.ReadToEnd();
            rsa.FromXmlString(publicKeyText);
            rsa.PersistKeyInCsp = true;
        }
    }
}

```

Using Azure Key Vault

Azure Key Vault allows developers to safeguard cryptographic keys and secrets for use in applications and services. Key Vault allows you to encrypt keys and secrets by using keys protected by hardware security models. Key Vault also gives you control over the keys that access and encrypt your data. Creating keys for development is quick, and the migration to production keys is easy and seamless. Keys can also easily be revoked as required.

Consider, for example, the keys that need to be used for signing and encryption but need to be kept external to your application because the application's users will be geographically distributed. Azure Key Vault provides a great solution here because the keys stored in the vault are invoked by a URI when needed.

Azure keeps the keys safe using industry-standard algorithms and Hardware Security Modules (HSMs). Your keys are therefore well protected without requiring you to write your own code. Best of all, they are easy to use from your applications.

There is a lot more to creating a Key Vault than illustrated here, but I want to give you an idea of what we're doing.

Creating a Key Vault is done within a resource group. You can also create a new resource group when creating the Key Vault.

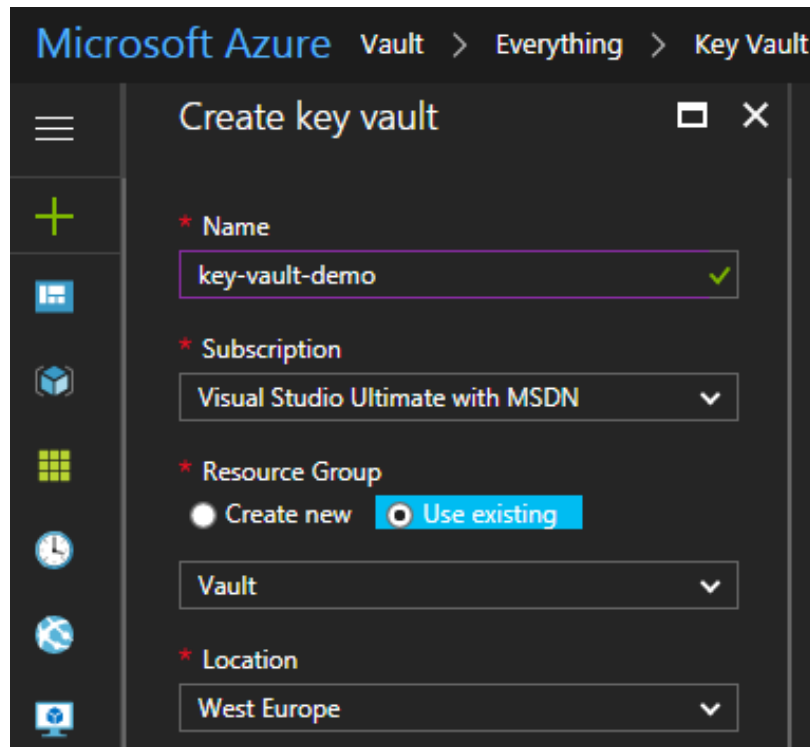


Figure 29: Create an Azure Key Vault

Once you have created your Key Vault, you will click **Secrets** under the Key Vault settings to add a new secret.

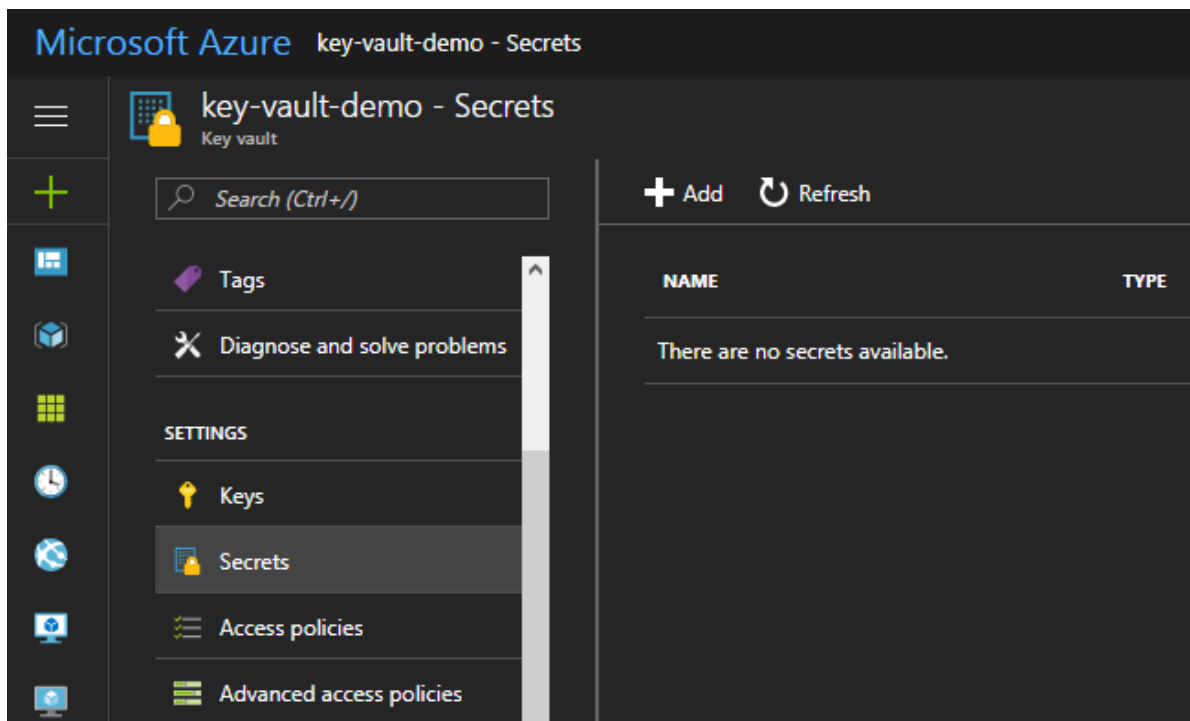
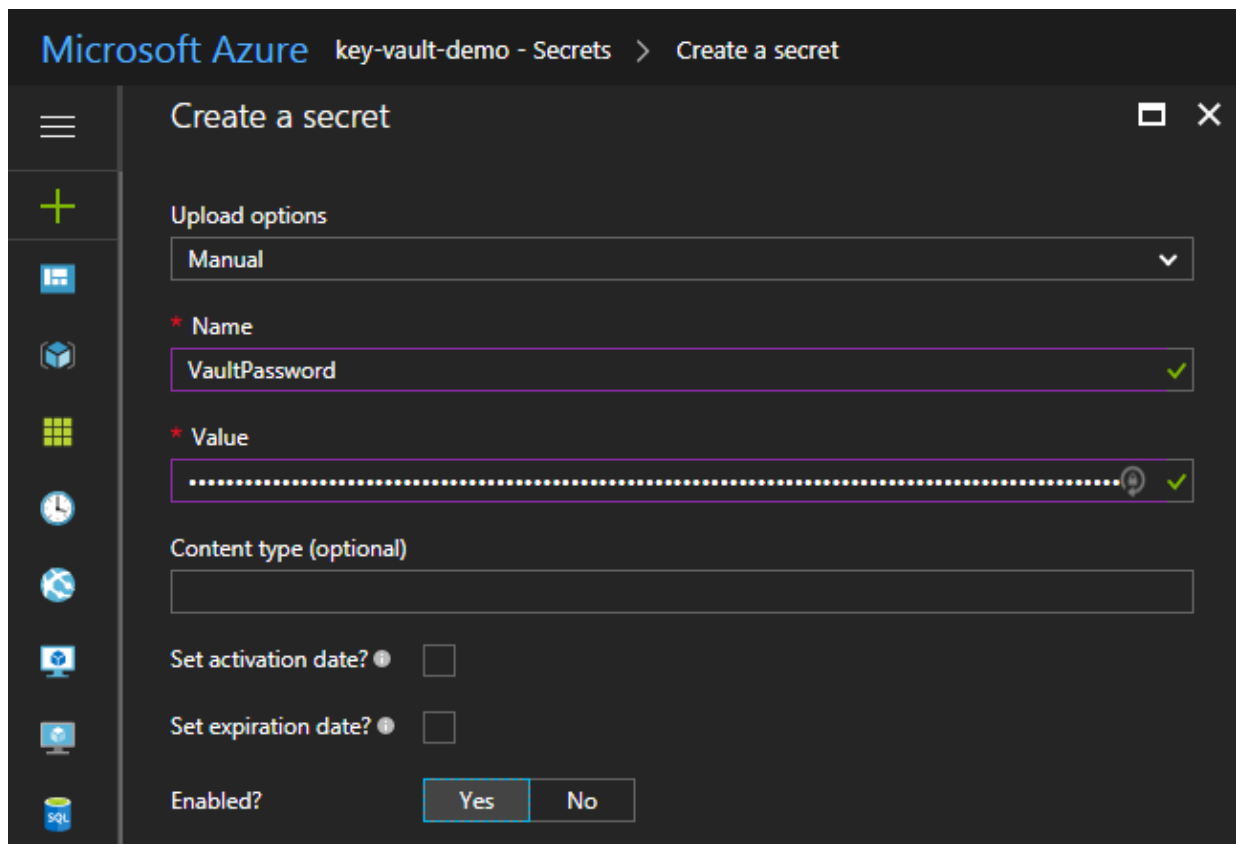


Figure 30: Store Secrets

Next, you select **Manual** as an upload option and give the secret a name and a value.



The screenshot shows the 'Create a secret' window in the Microsoft Azure portal. The breadcrumb path is 'key-vault-demo - Secrets > Create a secret'. The 'Upload options' dropdown menu is set to 'Manual'. The 'Name' field is labeled with a red asterisk and contains the text 'VaultPassword' with a green checkmark. The 'Value' field is also labeled with a red asterisk and contains a 100-character alphanumeric string with a green checkmark. Below these fields is the 'Content type (optional)' text box, which is currently empty. At the bottom, there are two checkboxes: 'Set activation date?' and 'Set expiration date?', both of which are unchecked. Finally, there is an 'Enabled?' section with 'Yes' and 'No' buttons; the 'Yes' button is highlighted.

Figure 31: Manually Create a Secret

The value of the secret is a 100-character key. This is the value we want to retrieve from the Key Vault and use in our application.

HuyvFwwc7v&G!sh2F3xJG!89N1YRNS%6QTrmtP034C7CoAbc^RKRWBbv80dSb\$xq!txEq2rDjp0jknBPjB%9%iuE2q1ZZEIge1c

We will now simply call the Key Vault from a console application. You will need to add the following NuGet packages to the application:

- Microsoft.Azure.KeyVault
- Microsoft.IdentityModel.Clients.ActiveDirectory

After you have added these, add the following code using statements to your application.

Code Listing 43: Additional Code

```
using Microsoft.Azure.KeyVault;  
using Microsoft.IdentityModel.Clients.ActiveDirectory;
```


The code we add to the console application is minimal. We are authenticating our application using the Active Directory application ID, application key, and the URL for the secret API key provided to us in Azure.

Code Listing 44: Additional Code

```
static void Main(string[] args)
{
    MainAsync(args).Wait();
}

public static async Task MainAsync(params string[] args)
{
    var applicationID = "[YOUR_APPLICATION_ID]";
    var vaultMasterKey = "[YOUR_APPLICATION_KEY]";
    var secretIdentifier = "[SECRET_API_KEY_URL]";

    var cloudVault = new KeyVaultClient(async (string authority, string
resource, string scope) => {
        var authContext = new AuthenticationContext(authority);
        var credential = new ClientCredential(applicationID,
vaultMasterKey);
        var token = await authContext.AcquireTokenAsync(resource,
credential);

        return token.AccessToken;
    });

    var apiKey =
cloudVault.GetSecretAsync(secretIdentifier).Result.Value;
}
```

When the console application is run, you will see that the secret key is returned to the application in the `cloudVault.GetSecretAsync(secretIdentifier).Result.Value` call.

Chapter 7 Using SecureString

There is another and often overlooked part of the .NET Framework that provides an additional bit of security to variables that contain sensitive information as it moves throughout your code. **SecureString** is a .NET class that provides a more secure alternative to the **String** class because it maintains a single copy in memory.



*Please note that **SecureString** is part of the **System.Security** namespace.*

It is a fact that **string** is quite an insecure way to keep sensitive information such as passwords, usernames, and credit card numbers. This is because **string** is not pinned in RAM, which means:

- It can be moved or copied by the Garbage Collector.
- It can leave multiple traces in RAM and multiple copies can exist in memory.
- RAM can be swapped to disk.
- The value of a string variable is stored in plain text.
- You can't easily zero it out.
- Because strings are immutable, the value of the string variable in memory is never modified. An extra copy of the string variable is created in memory with the changed value.

To see the value of a string variable in memory, do the following: write the code in Code Listing 45 to a console application and add a class-scope variable to the console application.

Code Listing 45: Global Variable

```
static int loop = 1;
```

In the **static void Main**, create a variable called **stringPassword** and assign it a value. Also place a breakpoint on the call to **PlainStringExample()** method.

Code Listing 46: Create Password Variable

```
string stringPassword = "This is a password";  
PlainStringExample(stringPassword);
```

Inside the method, assign a different value to the **stringPassword** variable. Place another breakpoint where the method exits.

Code Listing 47: Change Password Variable

```
private static void PlainStringExample(string stringPassword)  
{  
    stringPassword = stringPassword + $"{loop}";  
}
```

Now run the application and wait until it hits the first breakpoint.

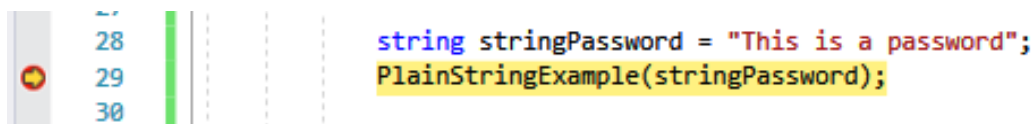


Figure 33: Code Breakpoint 1

With the code execution pauses at the first breakpoint, open the **Memory 1** window by going to **Debug, Windows, Memory**.

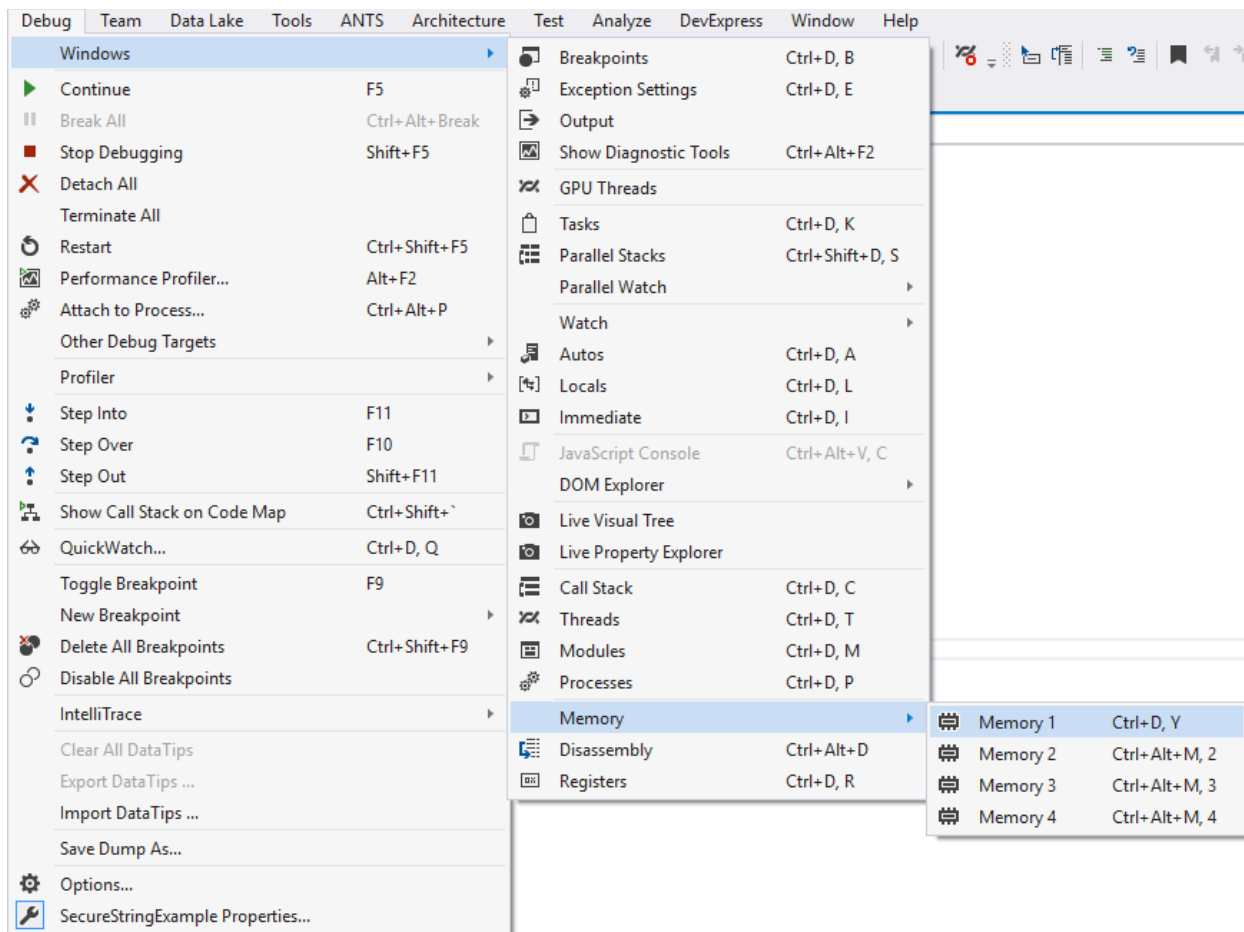


Figure 34: Open Memory Window

In the **Memory 1** window, change the value in the **Address** text box to the name of the variable **stringPassword** and click Enter. You will immediately see the memory window jump to the memory location of the **stringPassword** variable.



Note: The memory address locations will be different on your machine.

The address is **0x02CA24D0** and the plain text password is displayed to the right.

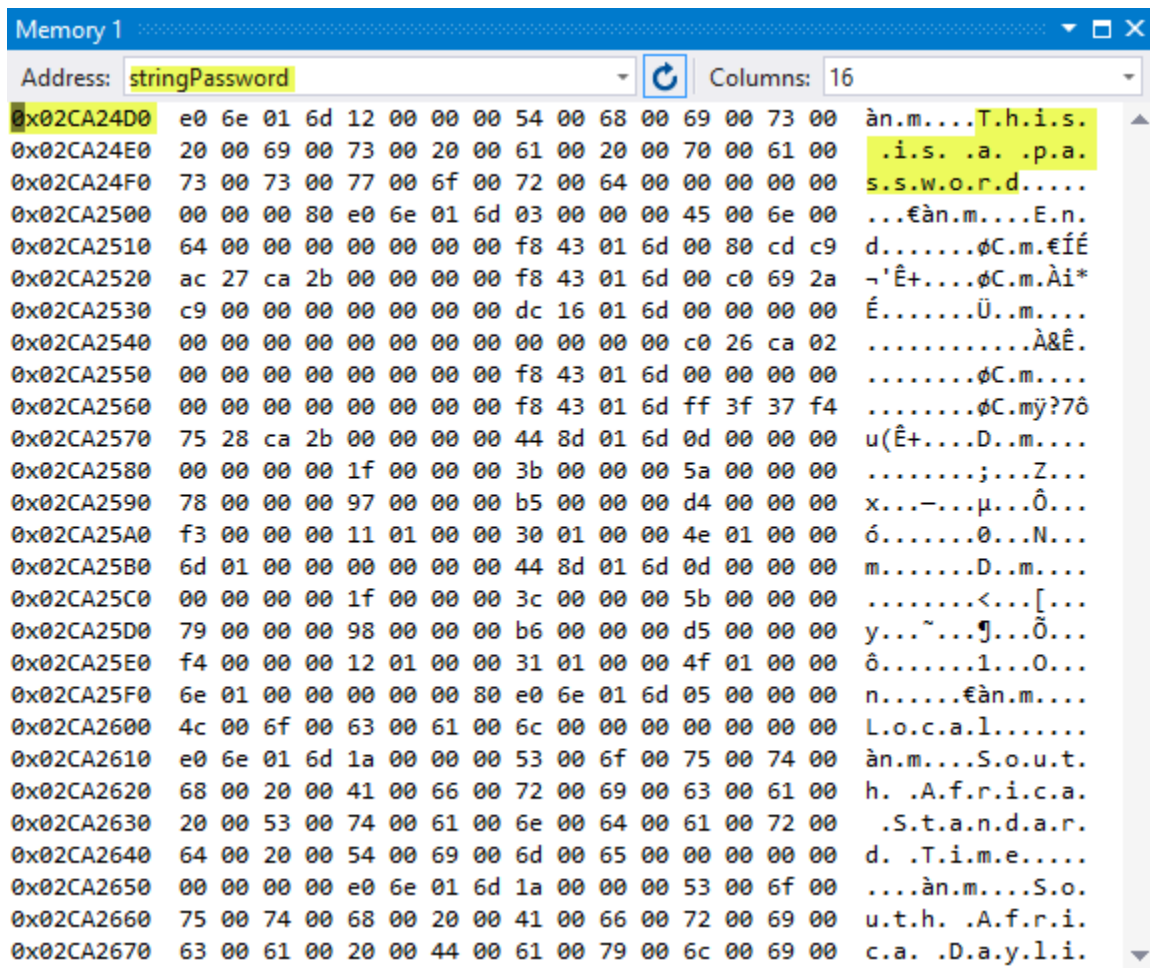


Figure 35: stringPassword Variable Memory Location 1

Continue the code execution and wait for the second breakpoint to be reached.

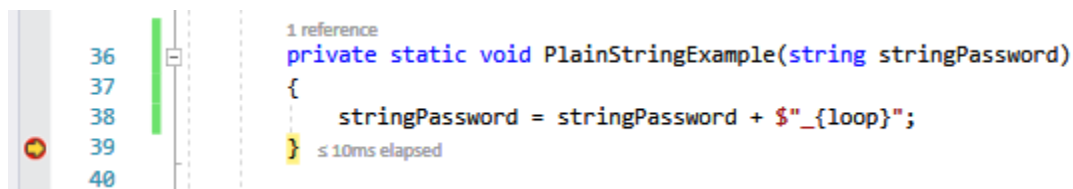


Figure 36: Code Breakpoint 2

You will immediately see the memory address for **stringPassword** change to a different address. In this example, the memory address has changed to **0x02CA65F8** with the updated text that is still visible in plain text.

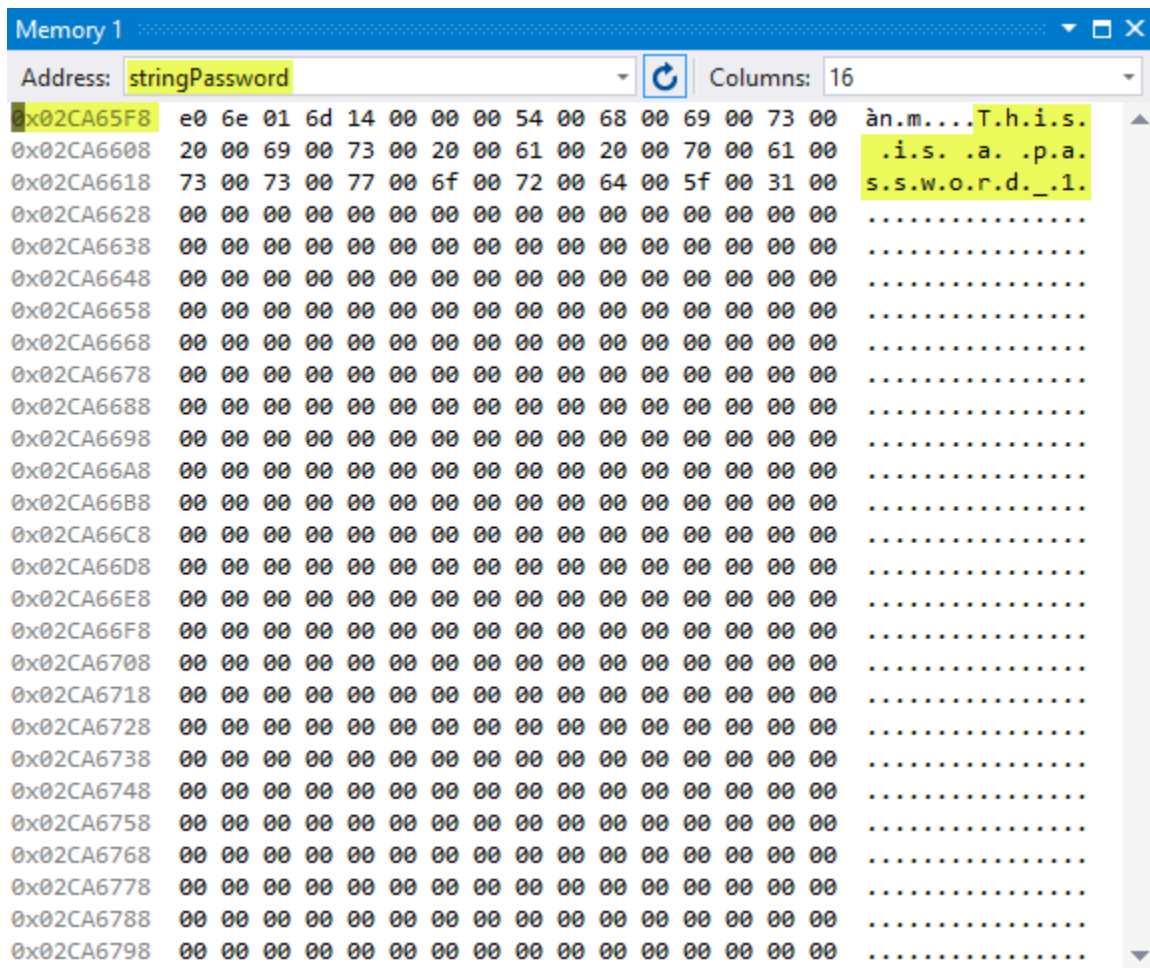


Figure 37: stringPassword Variable Memory Location 2

The benefits of using **SecureString** include:

- A single copy in memory.
- The value is not visible as plain text in memory.
- Implementation of **IDisposable** and a **Clear()** method.
- Changes to the value can be made by calling the **MakeReadOnly()** method.

To see how this works, let's look at a practical code example using **SecureString**. Start by adding the following using statements to the console application.

Code Listing 48: Using Statements

```
using static System.Console;
using System.Runtime.InteropServices;

using System.Security;
```

Next, add a class called **ExtensionMethods()**.

Code Listing 49: Extension Methods Class

```
public static class ExtensionMethods
{
}
}
```

Add an extension method called **Secure()** that will act on a string variable and return a **SecureString** value for the entered value. Notice how the **SecureString** value is created by adding the individual **char** values of the **string** to the **SecureString** object. This makes it easy to use a **SecureString** object with the individual key characters typed in on a keyboard.

Code Listing 50: Convert to SecureString

```
public static SecureString Secure(this string value)
{
    SecureString securedStringValue = new SecureString();

    if (!(string.IsNullOrEmpty(value)))
    {
        foreach (char c in value.ToCharArray())
        {
            securedStringValue.AppendChar(c);
        }
    }
    else
        return null;

    return securedStringValue;
}
```

Now we add a second extension method that will convert the **SecureString** value back to a **string**. It does this by allocating a read-only pointer, then it copies the **SecureString** value to the pointer. Finally, the method converts it to a managed **string** variable. After this, it frees the pointer.

Code Listing 51: Convert to Unsecure String

```
public static string Unsecure(this SecureString value)
{
    // Read-only field representing a pointer initialized to zero.
    var unsecuredString = IntPtr.Zero;

    try
    {
        // Copies the value of SecureString to unmanaged memory.
        unsecuredString =
        Marshal.SecureStringToGlobalAllocUnicode(value);
    }
}
```

```

        return Marshal.PtrToStringUni(unsecuredString);
    }
    finally
    {
        // Frees unmanaged string pointer.
        Marshal.ZeroFreeGlobalAllocUnicode(unsecuredString);
    }
}

```

To the console application **static void Main**, change the code to look like Code Listing 52. Put a breakpoint on **securePassword.Clear()** and another one on **securePassword.Dispose()**.

Code Listing 52: Calling Code

```

string stringPassword = "This is a password";
PlainStringExample(stringPassword);

SecureString securePassword = stringPassword.Secure();
string unsecuredPassword = securePassword.Unsecure();
securePassword.Clear();
securePassword.Dispose();

```

Run your application and have a look at the **securePassword** variable.

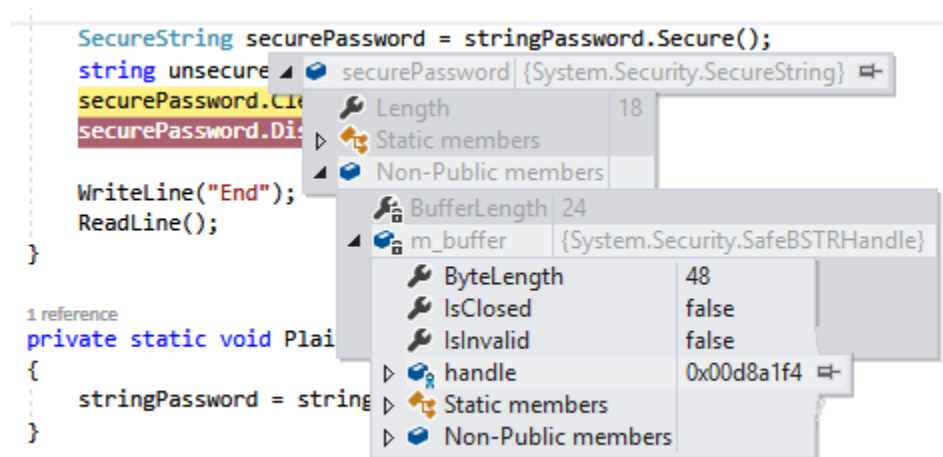


Figure 38: SecureString Handle

When you drill down far enough, you will see the handle for the **securePassword** variable. In this example, it is **0x00D8A1F4**.

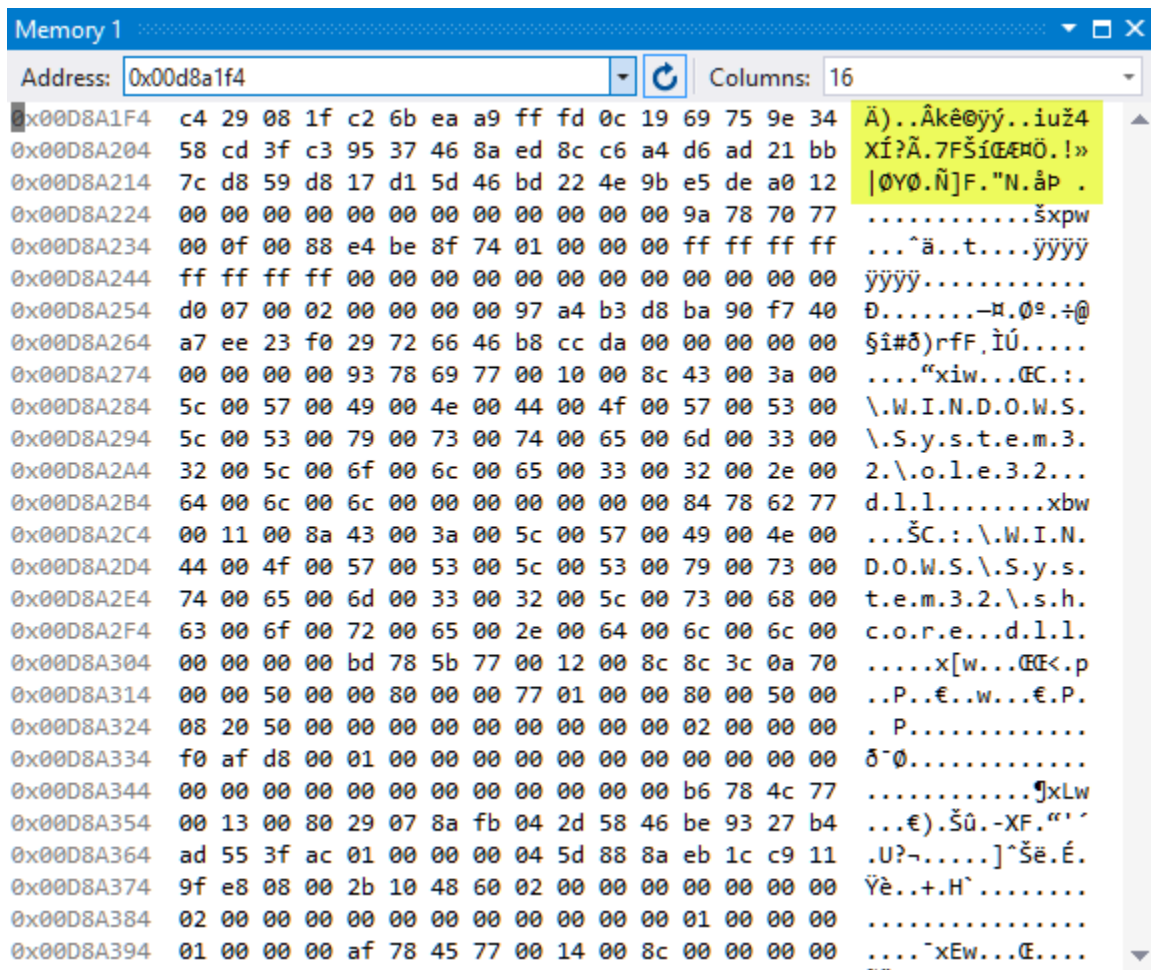


Figure 39: SecureString Encrypted Value

If you open the memory window, you will notice that the value of the **SecureString** variable in memory is encrypted.

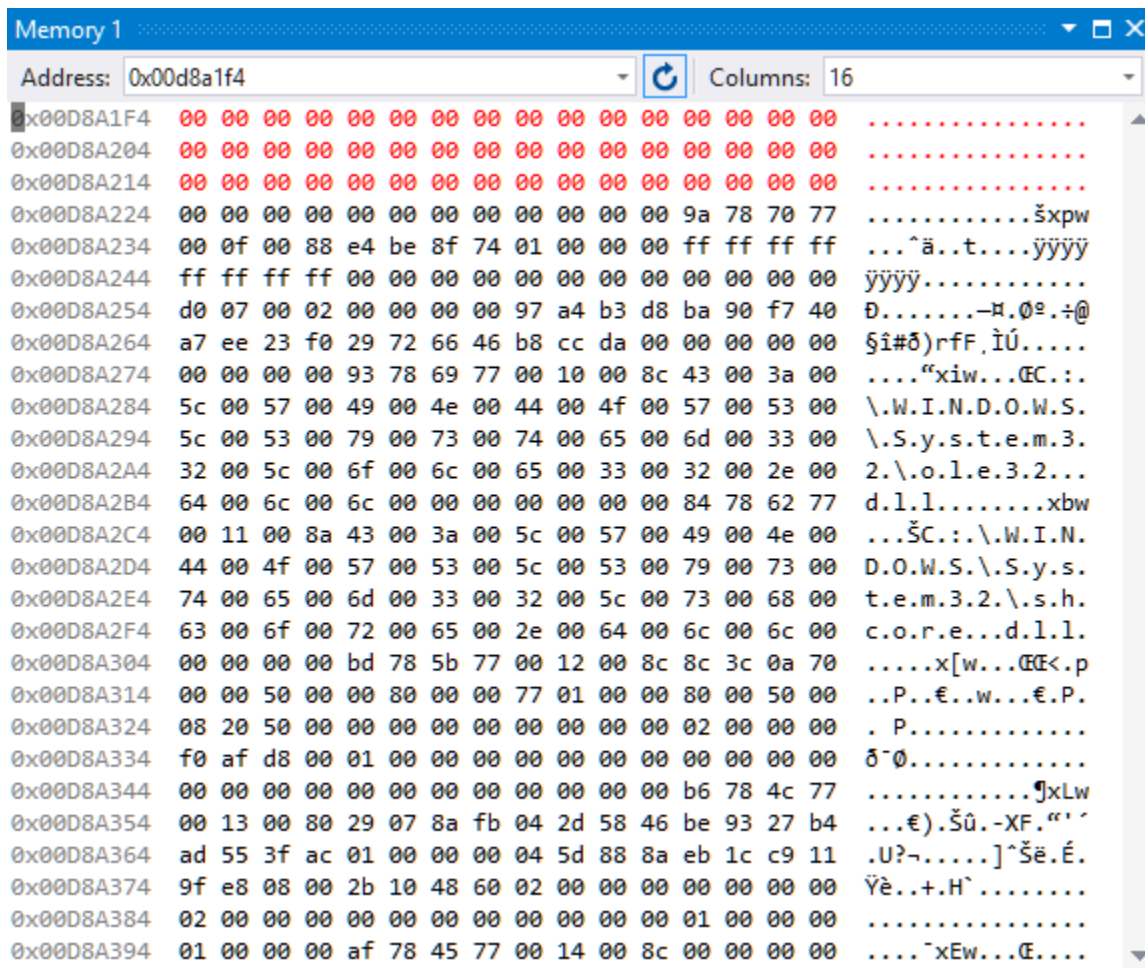


Figure 40: SecureString Variable Cleared

If you advance the breakpoint, you will see that the value of the **SecureString** object is cleared from memory.

Note that the **SecureString** object is not a replacement for keeping passwords stored inside a database. It is intended to keep the value of sensitive information obscured from prying eyes while in memory. There is a school of thought that says the benefit of using **SecureString** is so minimal that it does not warrant being implemented at all. If an attacker really wanted to, they could intercept the value of a string as it is being set to a **SecureString** object (on each keystroke—for example, by a key logger).

Personally, I feel that it is better to reduce the amount of attack vectors in an application and, in doing so, minimize the attack surface (sum of attack vectors). Combining this with solid cryptographic principles in the way you store and work with your user's data will help build more secure software.

Conclusion

Cryptography is something developers need to consider when developing world-class applications. There is always a place for it in any app. Thank you for reading this book. I hope that the examples illustrated in it will enable you to take the concept of securing your applications further.