Introduction to Hacking Thick Clients: Part 4 – The Assemblies

Introduction to Hacking Thick Clients is a series of blog posts that will outline many of the tools and methodologies used when performing thick client security assessments. In conjunction with these posts, NetSPI has released two vulnerable thick clients: BetaFast, a premier Betamax movie rental service, and Beta Bank, a premier finance application for the elite. Many examples in this series will be taken directly from these applications, which can be downloaded from the BetaFast GitHub repo. A brief overview is covered in a previous blog post.

Installments:

1. The GUI
2. The Network
3. The Filesystem and Registry
4. The Assemblies
5. The API
6. The Memory

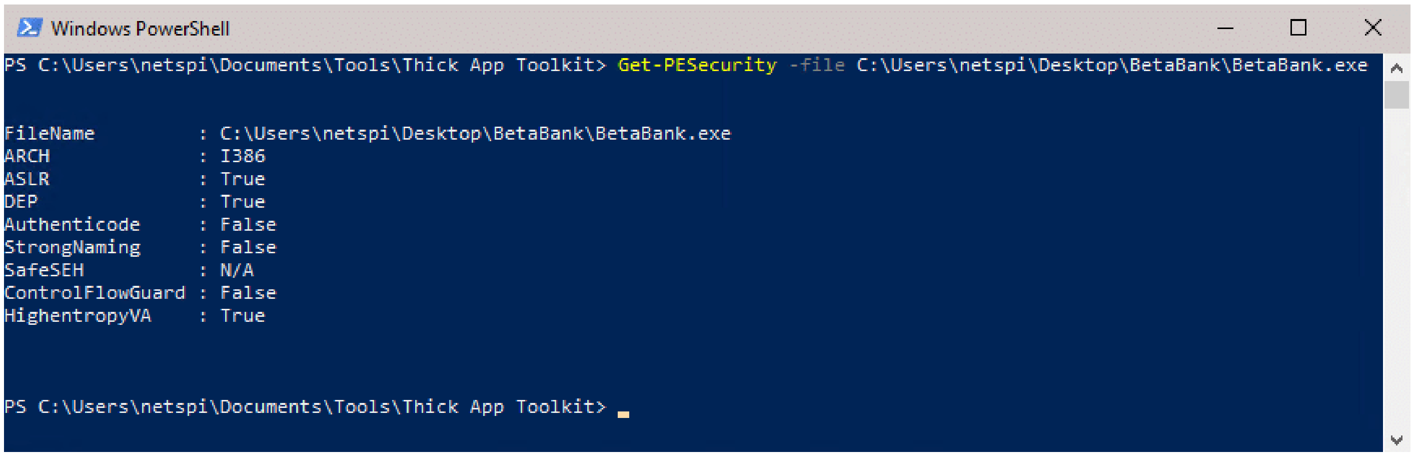
Assembly Controls

Libraries and executables can be compiled with some additional security measures to protect against code exploitation:

- Address Space Layout Randomization (ASLR) – An application's locations in memory are randomized at load time, preventing attacks such as return-to-libc that lead to code execution by overwriting specific addresses.
- SafeSEH – A list of safe exception handlers is stored within a binary, preventing an attacker from forcing the application to execute code during a call to a malicious exception.
- Data Execution Prevention (DEP) – Areas of memory can be marked as non-executable, preventing an attacker from storing code for a buffer overflow attack in these regions.
- Authenticode/Strong Naming – Assemblies can be protected by signing. If left unsigned, an attacker is able to modify and replace them with malicious content.
- Controlflowguard – An extension of ASLR and DEP that limits the addresses where code can execute from.
- HighentropyVA – A 64 bit application uses ASLR.

Thankfully, NetSPI's very own Eric Gruber has released a tool called PESecurity to check if a binary has been compiled with the above code execution preventions. Many of the thick client applications we test have installation directories filled to the brim with assemblies, and PESecurity is especially useful for checking a large number of files.

In the below example, PESecurity is used to check BetaBank.exe. It is compiled with ASLR and DEP. SafeSEH is only applicable for 32 bit assemblies. But the executable is unsigned.



```
PS C:\Users\netspi\Documents\Tools\Thick App Toolkit> Get-PESecurity -file C:\Users\netspi\Desktop\BetaBank\BetaBank.exe


FileName         : C:\Users\netspi\Desktop\BetaBank\BetaBank.exe
ARCH             : I386
ASLR             : True
DEP              : True
Authenticode     : False
StrongNaming     : False
SafeSEH          : N/A
ControlFlowGuard : False
HighentropyVA    : True



PS C:\Users\netspi\Documents\Tools\Thick App Toolkit>
```

Decompiling

Decompiling is one of my favorite parts of testing thick clients. As someone who has made far too many programming mistakes, it's cathartic to find those of other programmers. By their very nature, .Net assemblies can be read as source code using tools such as the following:

- dnSpy
- JustDecompile
- .NET Reflector

This is because .Net assemblies are managed code. When a .Net application is compiled, it's compiled to Intermediate Language code. Only at runtime is Intermediate Language code finally compiled to machine code by a runtime environment. .Net assemblies are so easy to "reverse" into source code because the Intermediate Language contains so much information such as types and names.

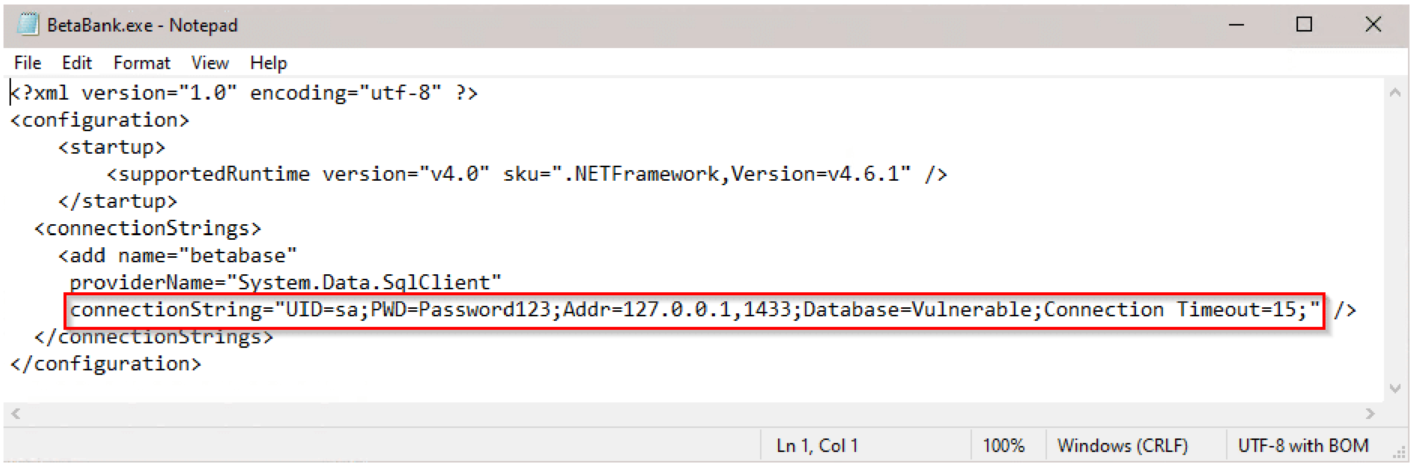
Unmanaged code, such as C or C++, is compiled down to a binary. It doesn't run through a runtime like C# does with Common Language Runtime – it's loaded directly into memory.

Information Disclosures

Managed Code

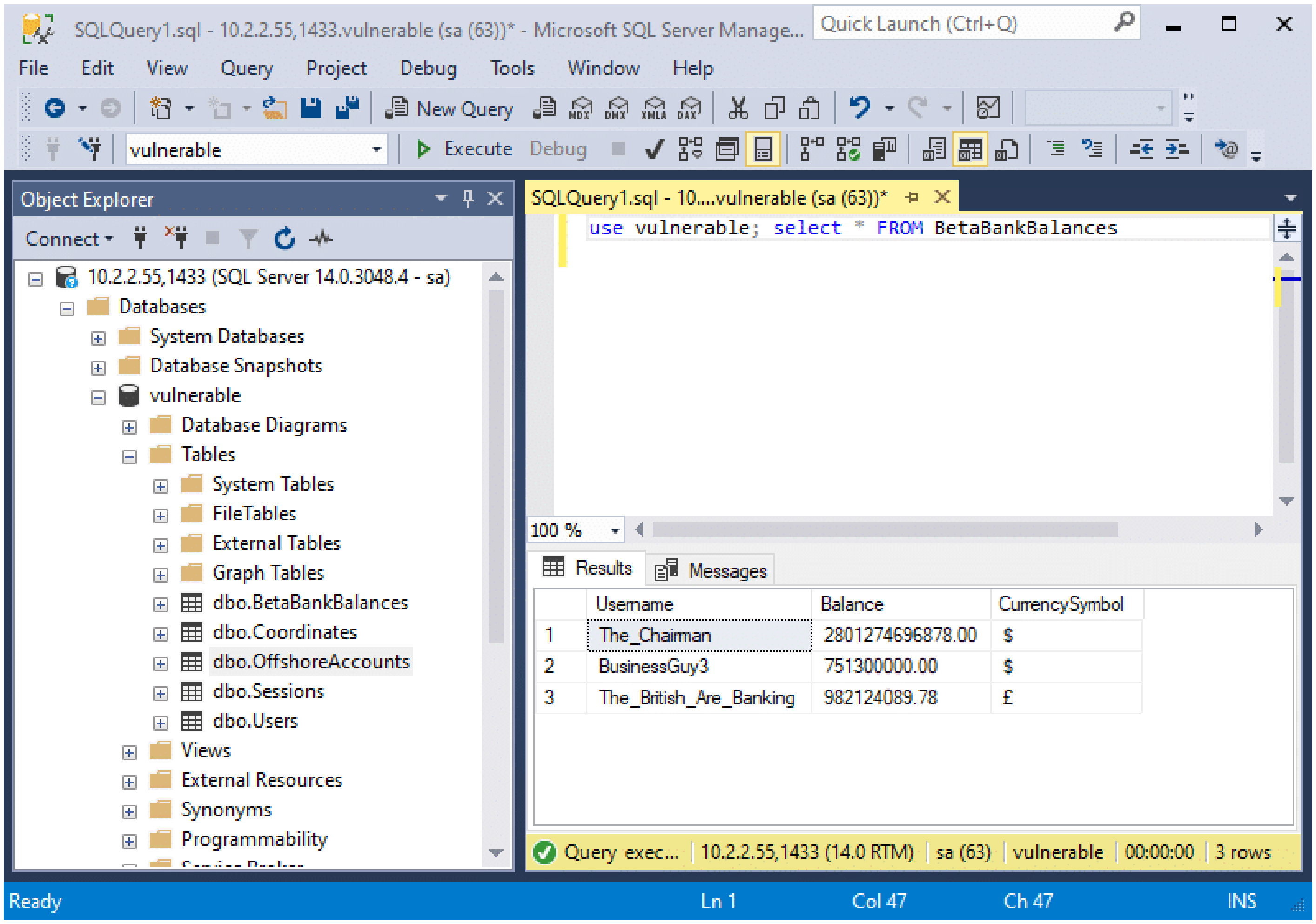
The following example will use the aforementioned BetaBank.exe, found in our BetaFast GitHub. It will also be using dnSpy as the decompiler of choice.

One of the first things I look for when testing a thick client application is hardcoded sensitive information such as credentials, encryption keys, and connection strings. Decompiling isn't even required to find sensitive information – configuration files are great places to look. When a .Net assembly is run, it may search a configuration file for global values such as a connection string, web endpoints, or passwords that are referenced in the executable. Procmon and the steps outlined in the previous entry are very useful for identifying these files. There's a connection string stored right in the Beta Bank's config file. This can be used to authenticate directly to the database using a tool such as SQL Server Management Studio.


BetaBank.exe - Notepad

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6.1" />
    </startup>
  <connectionStrings>
    <add name="betabase"
     providerName="System.Data.SqlClient"
     connectionString="UID=sa;PWD=Password123;Addr=127.0.0.1,1433;Database=Vulnerable;Connection Timeout=15;" />
  </connectionStrings>
</configuration>
```

If you're wondering about the 10.2.2.55 address, that's the VM I'm running Docker in.

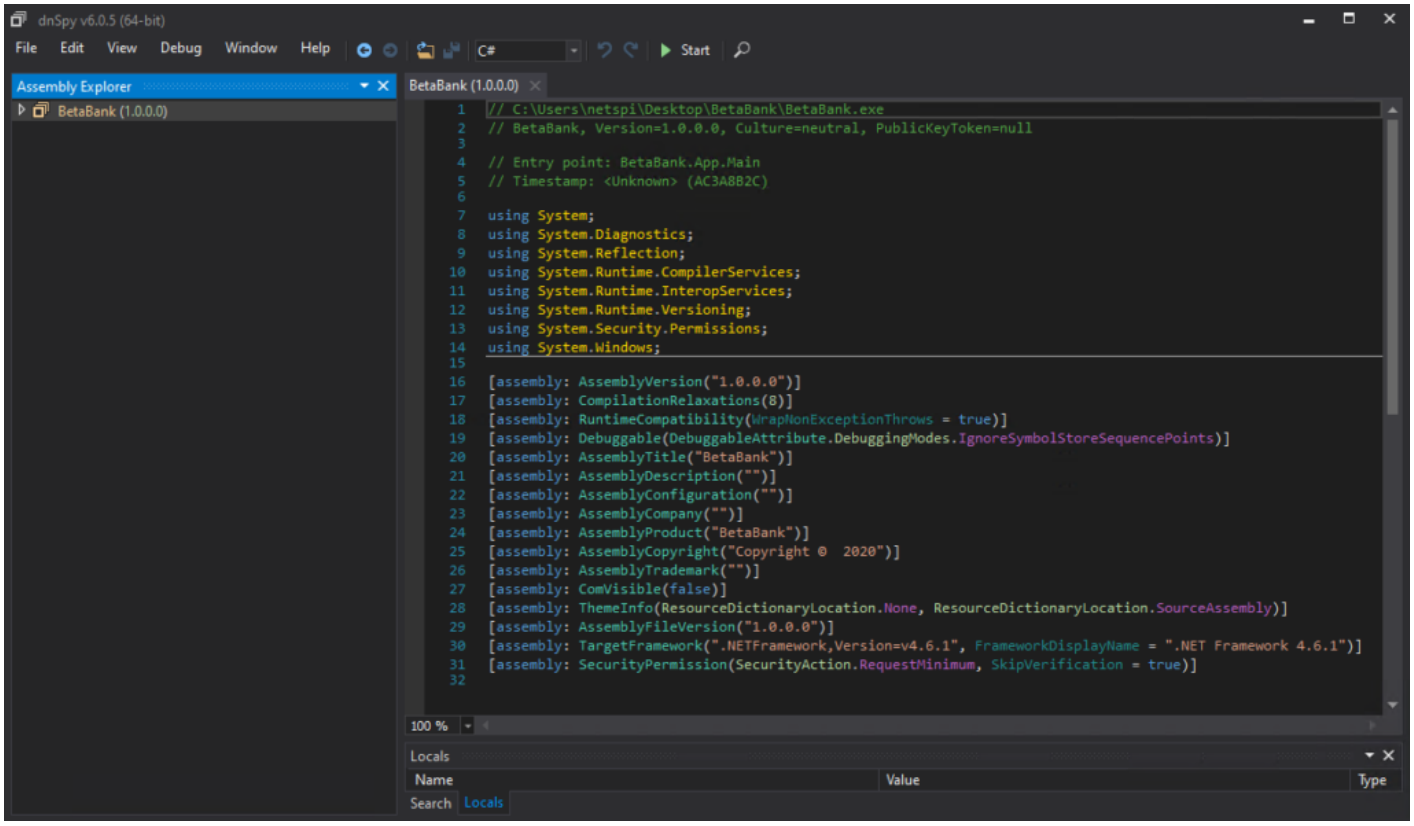

SQLQuery1.sql - 10.2.2.55,1433.vulnerable (sa (63))* - Microsoft SQL Server Manage...

```
use vulnerable; select * FROM BetaBankBalances
```

	Username	Balance	CurrencySymbol
1	The_Chairman	2801274696878.00	$
2	BusinessGuy3	751300000.00	$
3	The_British_Are_Banking	982124089.78	£

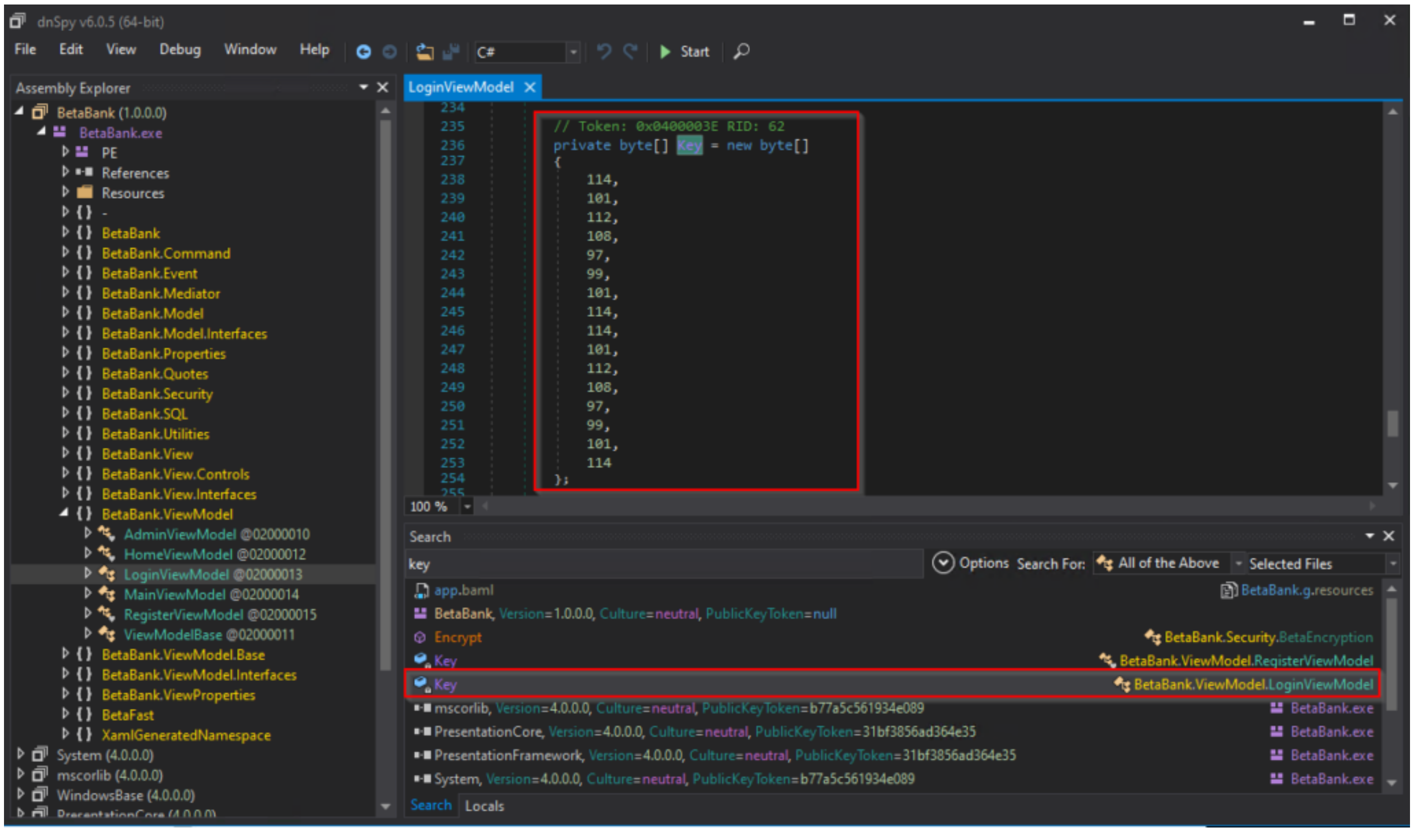
Query exec... | 10.2.2.55,1433 (14.0 RTM) | sa (63) | vulnerable | 00:00:00 | 3 rows

But for information disclosures in the source code, first decompile any of the application's binaries. BetaBank.exe is loaded into dnSpy.



```
// C:\Users\netspi\Desktop\BetaBank\BetaBank.exe
// BetaBank, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null

// Entry point: BetaBank.App.Main
// Timestamp: <Unknown> (AC3A8B2C)

using System;
using System.Diagnostics;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Runtime.Versioning;
using System.Security.Permissions;
using System.Windows;

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints)]
[assembly: AssemblyTitle("BetaBank")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("BetaBank")]
[assembly: AssemblyCopyright("Copyright ©  2020")]
[assembly: AssemblyTrademark("")]
[assembly: ComVisible(false)]
[assembly: ThemeInfo(ResourceDictionaryLocation.None, ResourceDictionaryLocation.SourceAssembly)]
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: TargetFramework(".NETFramework,Version=v4.6.1", FrameworkDisplayName = ".NET Framework 4.6.1")]
[assembly: SecurityPermission(SecurityAction.RequestMinimum, SkipVerification = true)]
```

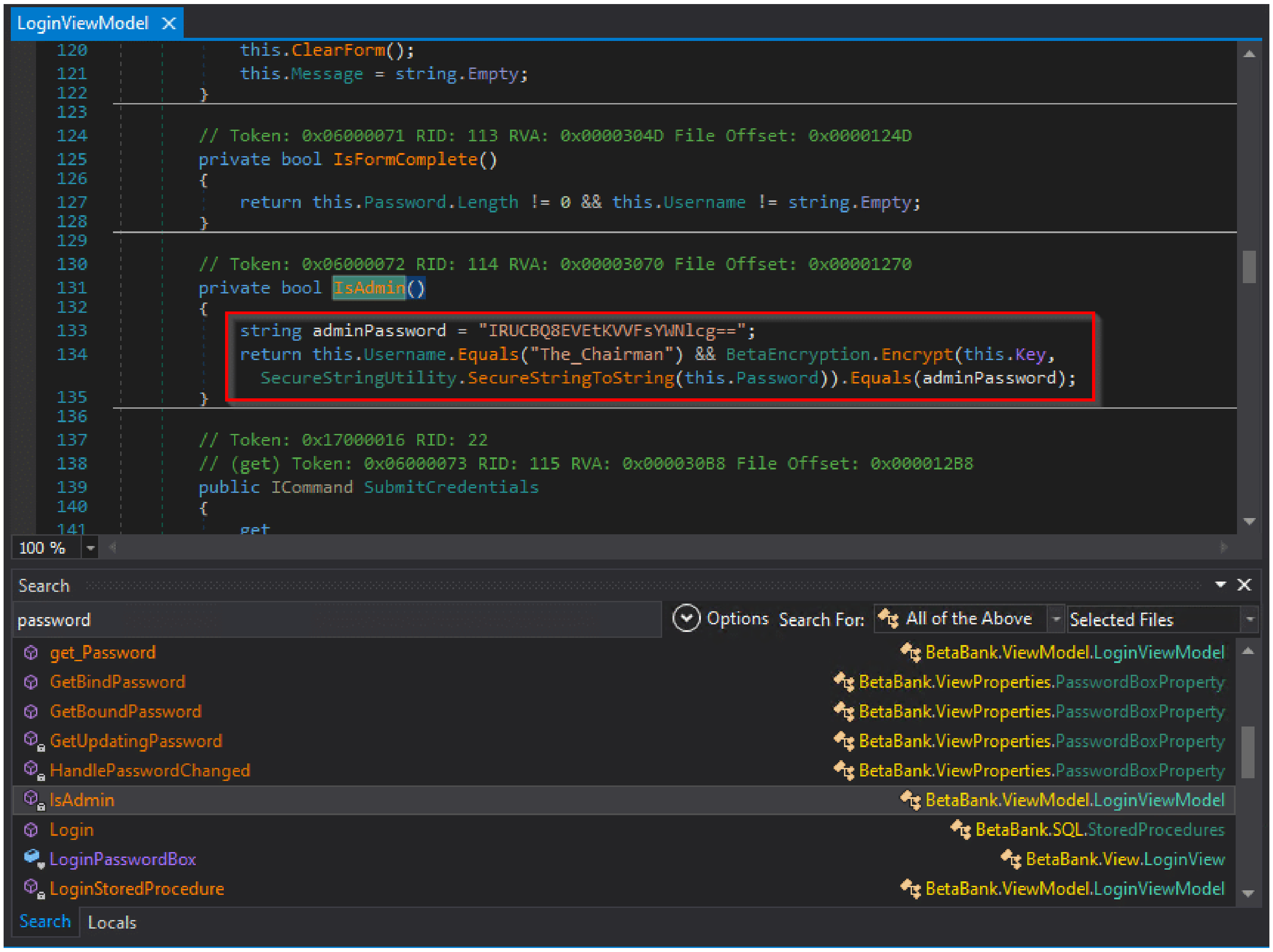
dnSpy has a solid search feature. I type in terms like “key,” “IV,” “connection,” “password,” “encrypt,” and “decrypt” to look for how the application handles encryption, authentication, and database connections. The application also has a filter in its search function. Below, I limited the search to only the Selected Files, but a search can also be limited to specific libraries and object types. Looks like there’s a hardcoded key in BetaBank.ViewModel.RegisterViewModel and BetaBank.ViewModel.LoginViewModel.



```
// Token: 0x0400003E RID: 62
private byte[] Key = new byte[]
{
    114,
    101,
    112,
    108,
    97,
    99,
    101,
    114,
    114,
    101,
    112,
    108,
    97,
    99,
    101,
    114
};
```

```
key
app.baml                                                                  BetaBank.g.resources
BetaBank, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
Encrypt                                                   BetaBank.Security.BetaEncryption
Key                                                  BetaBank.ViewModel.RegisterViewModel
Key                                                     BetaBank.ViewModel.LoginViewModel
mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089          BetaBank.exe
PresentationCore, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35  BetaBank.exe
PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35  BetaBank.exe
System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089            BetaBank.exe
```

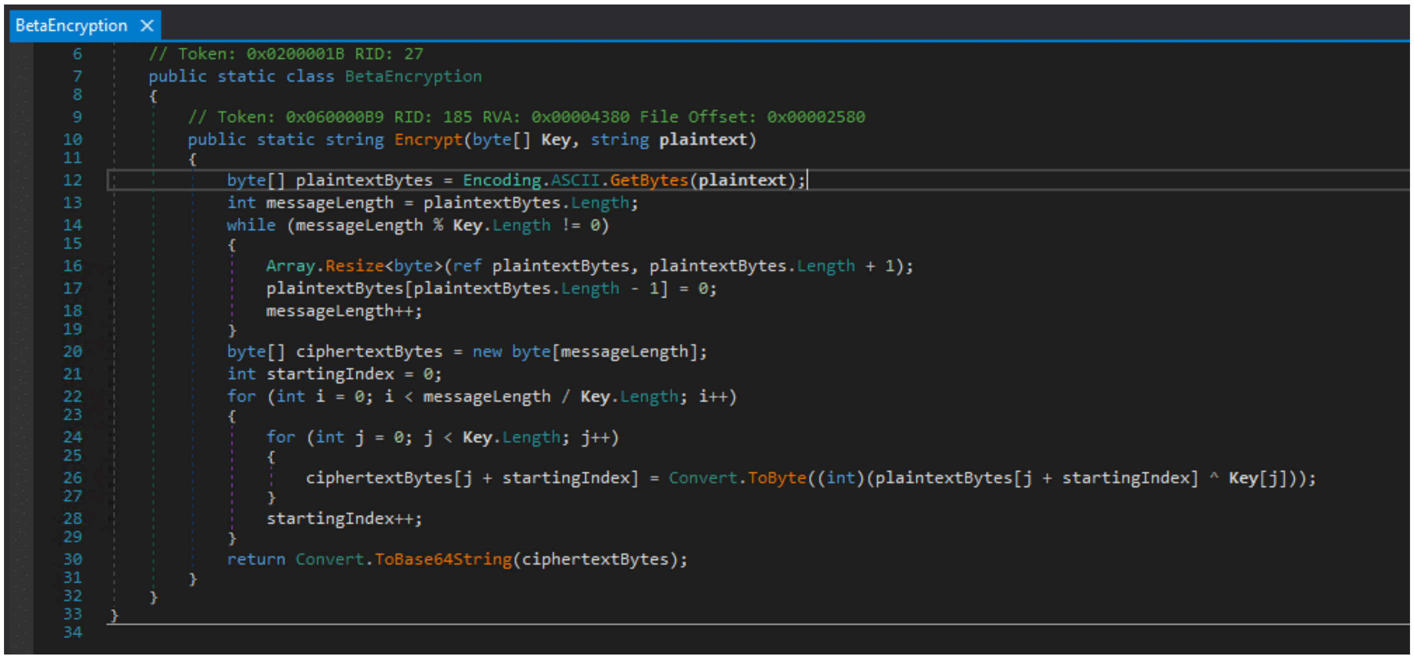
And searching for “password” shows a hardcoded encrypted password. Apparently the developer(s) implemented an authorization check on the client side. The username “The_Chairman” is directly compared to the value in the Username textbox, and the encrypted password is directly compared to the encrypted value of the value in the Password textbox.



```
            this.ClearForm();
            this.Message = string.Empty;
        }

        // Token: 0x06000071 RID: 113 RVA: 0x0000304D File Offset: 0x0000124D
        private bool IsFormComplete()
        {
            return this.Password.Length != 0 && this.Username != string.Empty;
        }

        // Token: 0x06000072 RID: 114 RVA: 0x00003070 File Offset: 0x00001270
        private bool IsAdmin()
        {
            string adminPassword = "IRUCBQ8EVEtKVVFsYWNlcg==";
            return this.Username.Equals("The_Chairman") && BetaEncryption.Encrypt(this.Key,
              SecureStringUtility.SecureStringToString(this.Password)).Equals(adminPassword);
        }

        // Token: 0x17000016 RID: 22
        // (get) Token: 0x06000073 RID: 115 RVA: 0x000030B8 File Offset: 0x000012B8
        public ICommand SubmitCredentials
        {
            get
```

```
password

get_Password                BetaBank.ViewModel.LoginViewModel
GetBindPassword             BetaBank.ViewProperties.PasswordBoxProperty
GetBoundPassword            BetaBank.ViewProperties.PasswordBoxProperty
GetUpdatingPassword         BetaBank.ViewProperties.PasswordBoxProperty
HandlePasswordChanged       BetaBank.ViewProperties.PasswordBoxProperty
IsAdmin                     BetaBank.ViewModel.LoginViewModel
Login                       BetaBank.SQL.StoredProcedures
LoginPasswordBox            BetaBank.View.LoginView
LoginStoredProcedure        BetaBank.ViewModel.LoginViewModel
```

The BetaEncryption class can be decompiled, showing some very custom encryption logic.



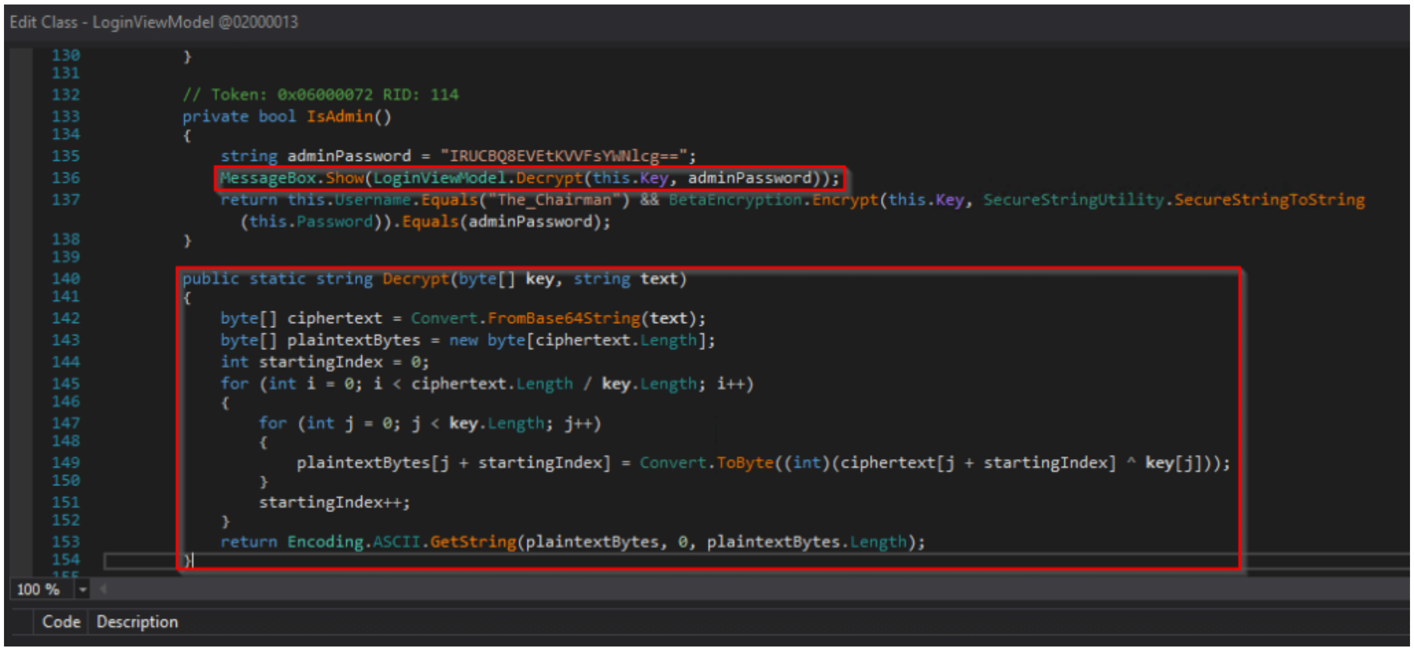
```
    // Token: 0x0200001B RID: 27
    public static class BetaEncryption
    {
        // Token: 0x060000B9 RID: 185 RVA: 0x00004380 File Offset: 0x00002580
        public static string Encrypt(byte[] Key, string plaintext)
        {
            byte[] plaintextBytes = Encoding.ASCII.GetBytes(plaintext);
            int messageLength = plaintextBytes.Length;
            while (messageLength % Key.Length != 0)
            {
                Array.Resize<byte>(ref plaintextBytes, plaintextBytes.Length + 1);
                plaintextBytes[plaintextBytes.Length - 1] = 0;
                messageLength++;
            }
            byte[] ciphertextBytes = new byte[messageLength];
            int startingIndex = 0;
            for (int i = 0; i < messageLength / Key.Length; i++)
            {
                for (int j = 0; j < Key.Length; j++)
                {
                    ciphertextBytes[j + startingIndex] = Convert.ToByte((int)(plaintextBytes[j + startingIndex] ^ Key[j]));
                }
                startingIndex++;
            }
            return Convert.ToBase64String(ciphertextBytes);
        }
    }
}
```

Unmanaged Code

As I mentioned at the beginning of this section, it’s not possible to have such a clean look at the source code when testing unmanaged code. Instead, we once again look to the CTO of Microsoft Azure for help. Strings is a tool in the Sysinternals suite that will output a list of strings saved in an executable. When analyzing memory dumps or unmanaged code binaries, I will retrieve the list of all the strings (most of them will probably look like nonsense), and then search for key terms like the ones above in a text editor of choice.

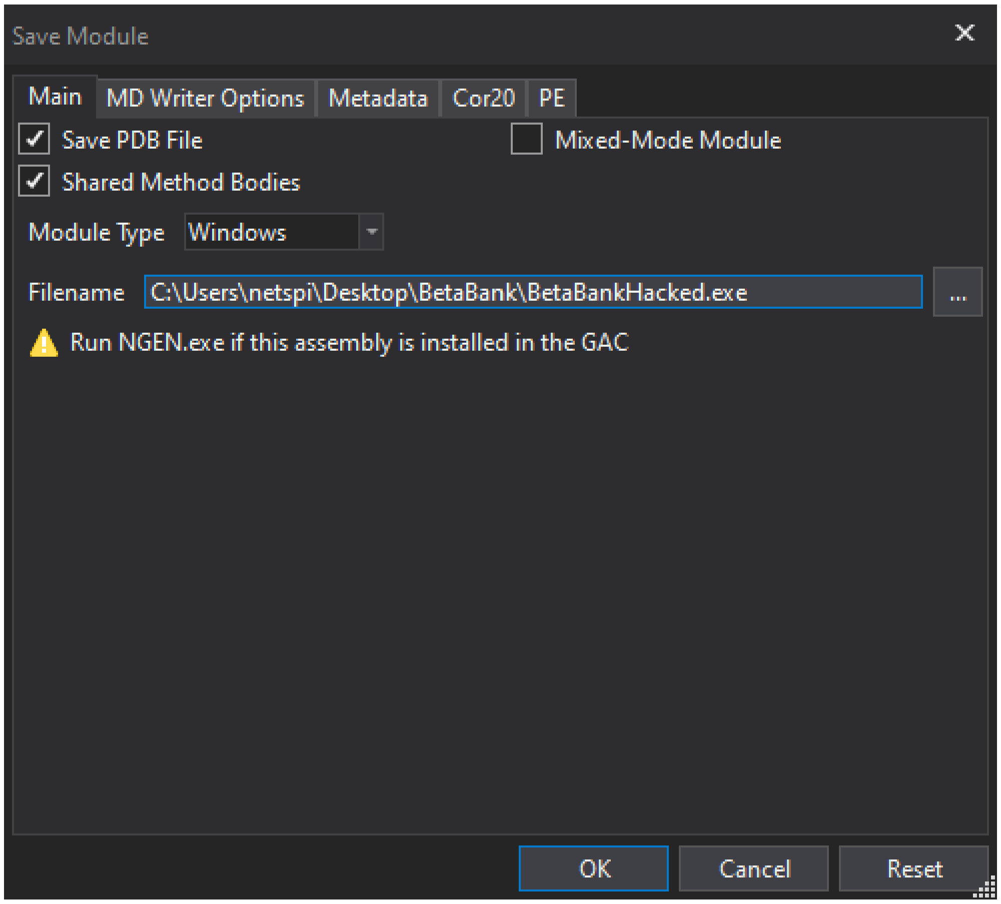
Modifying Assemblies

Using dnSpy, a class can actually be modified, and the binary can be recompiled. Below, I’ve reversed the Encrypt function, and I use a MessageBox to show the decrypted adminPassword when a user successfully authenticates to the application.



```
        }

        // Token: 0x06000072 RID: 114
        private bool IsAdmin()
        {
            string adminPassword = "IRUCBQ8EVEtKVVFsYWNlcg==";
            MessageBox.Show(LoginViewModel.Decrypt(this.Key, adminPassword));
            return this.Username.Equals("The_Chairman") && BetaEncryption.Encrypt(this.Key, SecureStringUtility.SecureStringToString
              (this.Password)).Equals(adminPassword);
        }

        public static string Decrypt(byte[] key, string text)
        {
            byte[] ciphertext = Convert.FromBase64String(text);
            byte[] plaintextBytes = new byte[ciphertext.Length];
            int startingIndex = 0;
            for (int i = 0; i < ciphertext.Length / key.Length; i++)
            {
                for (int j = 0; j < key.Length; j++)
                {
                    plaintextBytes[j + startingIndex] = Convert.ToByte((int)(ciphertext[j + startingIndex] ^ key[j]));
                }
                startingIndex++;
            }
            return Encoding.ASCII.GetString(plaintextBytes, 0, plaintextBytes.Length);
        }
```

Saving the module as a new executable . . .


Save Module

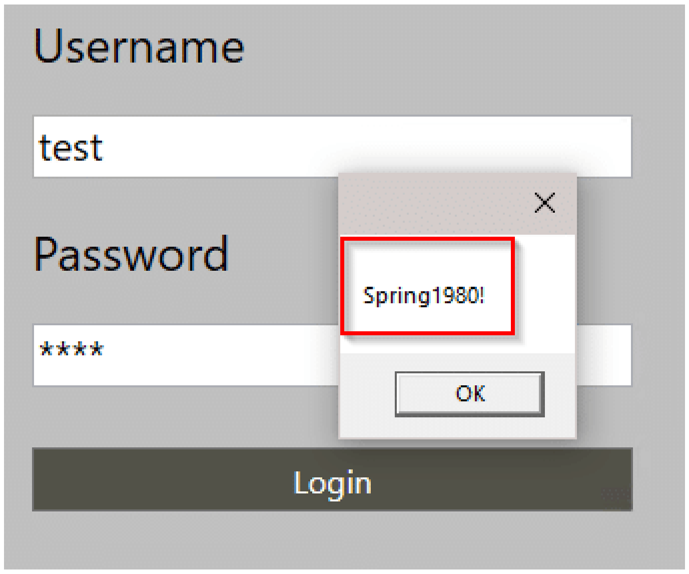
[✓] Save PDB File
[ ] Mixed-Mode Module
[✓] Shared Method Bodies

Module Type: Windows

Filename: C:\Users\netspi\Desktop\BetaBank\BetaBankHacked.exe

Run NGEN.exe if this assembly is installed in the GAC

Finally, when I authenticate as a user, the MessageBox prints The_Chairman's password.

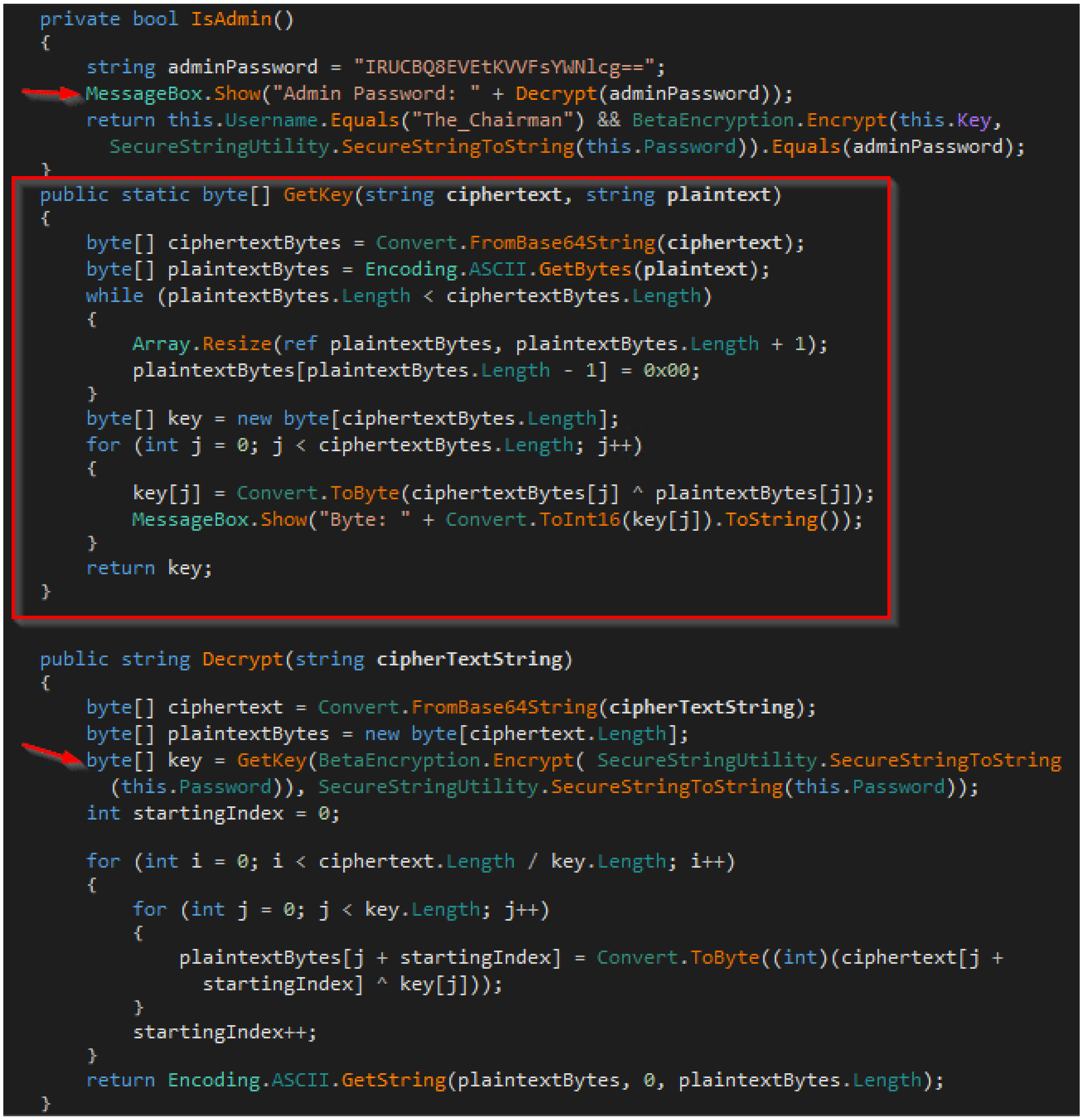

Username: test

Password: ****

Spring1980!

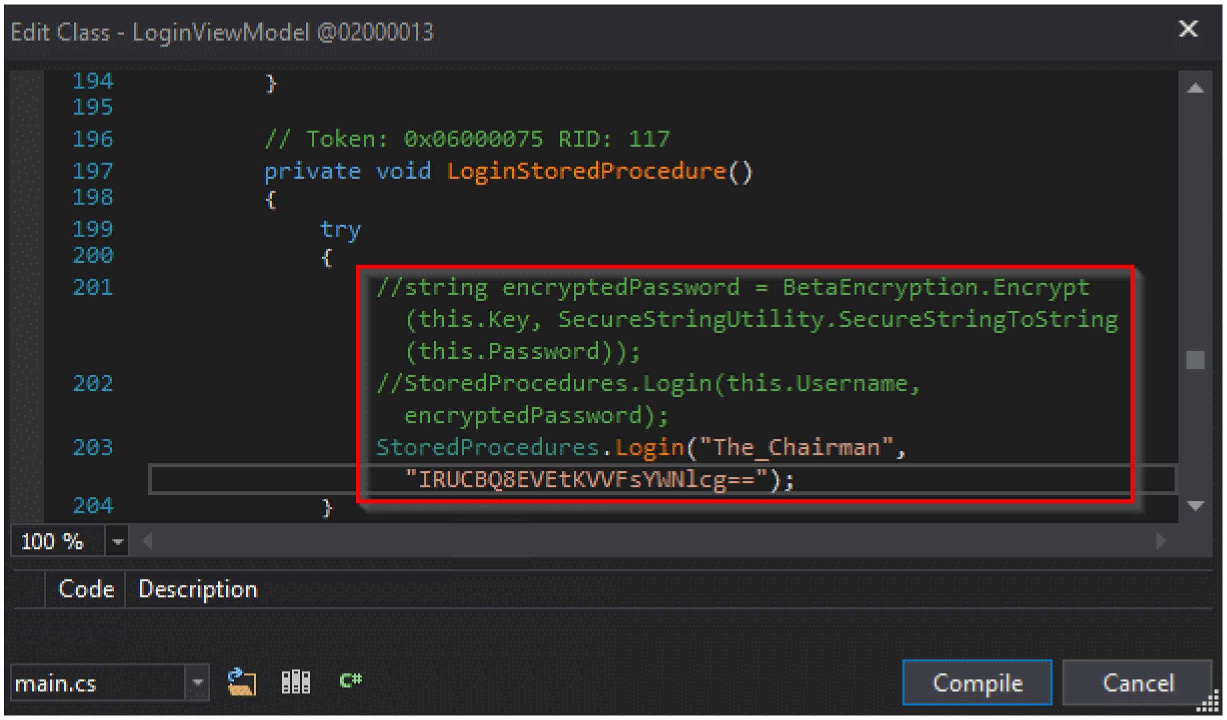
A MessageBox is very useful to add to the application when trying to gather information. It's like when I use a print statement instead of a debugger.

A Beta Bank developer may decide to patch this vulnerability by removing the hardcoded encryption key. However, Beta Bank's custom encryption can still be reverse engineered to gather the key for decryption. Below, I've added a function that performs an XOR on the plaintext password and the encrypted password, thus exposing the encryption key. In the Decrypt function, I retrieve the key by passing a known plaintext password (my own) and its encrypted value.



```
private bool IsAdmin()
{
    string adminPassword = "IRUCBQ8EVEtKVVFsYWNlcg==";
    MessageBox.Show("Admin Password: " + Decrypt(adminPassword));
    return this.Username.Equals("The_Chairman") && BetaEncryption.Encrypt(this.Key,
      SecureStringUtility.SecureStringToString(this.Password)).Equals(adminPassword);
}
public static byte[] GetKey(string ciphertext, string plaintext)
{
    byte[] ciphertextBytes = Convert.FromBase64String(ciphertext);
    byte[] plaintextBytes = Encoding.ASCII.GetBytes(plaintext);
    while (plaintextBytes.Length < ciphertextBytes.Length)
    {
        Array.Resize(ref plaintextBytes, plaintextBytes.Length + 1);
        plaintextBytes[plaintextBytes.Length - 1] = 0x00;
    }
    byte[] key = new byte[ciphertextBytes.Length];
    for (int j = 0; j < ciphertextBytes.Length; j++)
    {
        key[j] = Convert.ToByte(ciphertextBytes[j] ^ plaintextBytes[j]);
        MessageBox.Show("Byte: " + Convert.ToInt16(key[j]).ToString());
    }
    return key;
}

public string Decrypt(string cipherTextString)
{
    byte[] ciphertext = Convert.FromBase64String(cipherTextString);
    byte[] plaintextBytes = new byte[ciphertext.Length];
    byte[] key = GetKey(BetaEncryption.Encrypt( SecureStringUtility.SecureStringToString
      (this.Password)), SecureStringUtility.SecureStringToString(this.Password));
    int startingIndex = 0;

    for (int i = 0; i < ciphertext.Length / key.Length; i++)
    {
        for (int j = 0; j < key.Length; j++)
        {
            plaintextBytes[j + startingIndex] = Convert.ToByte((int)(ciphertext[j +
              startingIndex] ^ key[j]));
        }
        startingIndex++;
    }
    return Encoding.ASCII.GetString(plaintextBytes, 0, plaintextBytes.Length);
}
```

But that was actually too much work. If we don’t need The_Chairman’s password for later, the username and encrypted password can just be placed directly into the Login function. Boolean logic can also be modified so that anything verified server-side, such as IsAdmin(), can be modified to always return true.



Edit Class - LoginViewModel @02000013

```
        }

        // Token: 0x06000075 RID: 117
        private void LoginStoredProcedure()
        {
            try
            {
                //string encryptedPassword = BetaEncryption.Encrypt
                  (this.Key, SecureStringUtility.SecureStringToString
                  (this.Password));
                //StoredProcedures.Login(this.Username,
                  encryptedPassword);
                StoredProcedures.Login("The_Chairman",
                  "IRUCBQ8EVEtKVVFsYWNlcg==");
            }
```

Obfuscation

Source code isn't often this legible. Sometimes, code is obfuscated. And while there's no security through obscurity, it sure does make the job of a security consultant a little more difficult.

Below is the BetaEncryption class from earlier after I applied some custom obfuscation by mashing my keyboard to rename the class, methods, and parameters.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

public static class jinjnhglkjd
{
    public static string dsfggd(byte[] erttr, string dgfhfgi)
    {
        byte[] plaintextBytes = Encoding.ASCII.GetBytes(dgfhfgi);

        byte[] ciphertextBytes;

        int messageLength = plaintextBytes.Length;

        while (messageLength % erttr.Length != 0)
        {
            Array.Resize(ref plaintextBytes, plaintextBytes.Length + 1);
            plaintextBytes[plaintextBytes.Length - 1] = 0x00;
            messageLength += 1;
        }

        ciphertextBytes = new byte[messageLength];
        int startingIndex = 0;
        for (int i = 0; i < (messageLength / erttr.Length); i++)
        {
            for (int j = 0; j < erttr.Length; j++)
            {
                ciphertextBytes[j + startingIndex] =
Convert.ToByte(plaintextBytes[j + startingIndex] ^ erttr[j]);
            }
            startingIndex++;
        }
        return Convert.ToBase64String(ciphertextBytes);
    }
}
```

All of the functionality is still there. Some obfuscation techniques may go even further to make code illegible. But what this does is makes it more difficult to search through and reverse engineer the source code. The Assembly Explorer of dnSpy would be full of nonsense class names, and searching for specific terms such as “Encrypt” would not yield any results.

There are various tools for deobfuscation such as de4dot. Below is an example of what a deobfuscator may interpret my keyboard mashing obfuscation as. The classes at least have some sense of legibility, and methods are named some variation of *method* rather than *dsfggd*.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

public static class class_1
{
    public static string method_1(byte[] byte_1, string string_1)
    {
        byte[] plaintextBytes = Encoding.ASCII.GetBytes(string_1);

        byte[] ciphertextBytes;

        int messageLength = plaintextBytes.Length;

        while (messageLength % byte_1.Length != 0)
        {
            Array.Resize(ref plaintextBytes, plaintextBytes.Length + 1);
            plaintextBytes[plaintextBytes.Length - 1] = 0x00;
            messageLength += 1;
        }

        ciphertextBytes = new byte[messageLength];
        int startingIndex = 0;
        for (int i = 0; i < (messageLength / byte_1.Length); i++)
        {
            for (int j = 0; j < byte_1.Length; j++)
            {
                ciphertextBytes[j + startingIndex] =
Convert.ToByte(plaintextBytes[j + startingIndex] ^ byte_1[j]);
            }
            startingIndex++;
        }
        return Convert.ToBase64String(ciphertextBytes);
    }
}
```

Obfuscation doesn’t secure code. But it makes browsing the source code all around more of a headache than reading someone else’s large codebase typically is.