Limiting The Exposure of Plain Text Passwords in C#

written by Austin Altmann | October 24, 2019

One vulnerability that we frequently look for when testing thick client applications is plain text passwords that are exposed in memory. Microsoft provides the SecureString to help protect passwords in memory, but what it does not provide is a perfect solution to actually using the SecureString when sending web requests. And as you'll see below, there is none, but there are some safer ways to operate.

NetworkCredential Might Be Good Enough

If you're handling passwords in forms other than the login, NetworkCredential won't be the best tool for the job. But if you want to use an authentication scheme like Basic or Digest, you might be satisfied implementing a CredentialCache and NetworkCredential as seen here. After all, there's a NetworkCredential constructor with a SecureString password argument. However, the password will be exposed in memory as a string when preparing to send a WebRequest with the credentials. Each authentication scheme makes a call to the NetworkCredential's InternalGetPassword function which will return the password in plain text.

```
internal string InternalGetPassword() {
                        string     decryptedString     =
UnsafeNclNativeMethods.SecureStringHelper.CreateString(m_passw
ord);
    return decryptedString;
}
```

The String Problem

The entire point of using a SecureString is to limit the exposure of the plain text password in memory (see the blog

title). Strings are immutable and therefore cannot be cleared on command. Their values will exist in memory until the garbage collector rolls around. If strings are used to handle the plain text password, it may exist in more addresses and for longer than anticipated.

The String Solution

Don't use strings. Use a character or byte array – arrays are mutable!

The Code

Below is some of my code to log into the application. The basic idea of it comes from some Microsoft examples as well as this StackOverflow post. SecureStringLogin accepts the username, password (as a SecureString), and the login function itself which accepts the username as a string and the password as a byte array. GCHandle.Alloc is called to pin the byte array of the future plain text password, ensuring that it stays fixed in one memory location.

```
public static async Task SecureStringLogin(string username,
SecureString password, Func<string, byte[], Task> Login)
{
    byte[] plaintext = null;
       GCHandle gcHandle = GCHandle.Alloc(plaintext,
GCHandleType.Pinned);

    IntPtr unmanagedPtr = IntPtr.Zero;

    try
    {
         SecureStringToByteArray(unmanagedPtr, password, ref
plaintext);
        await Login(username, plaintext);
    }
    finally
    {
         ClearPasswordFromMemory(unmanagedPtr, ref plaintext,
```

```
password, gcHandle);
    }
}
```

And to convert the SecureString to a byte array, we'll need the SecureString password itself, a reference to the byte array, and to set that IntPtr to the location of the SecureString's contents.

```
private static void SecureStringToByteArray(IntPtr ptr, SecureString ciphertext, ref byte[] plaintext)
{
    ptr= Marshal.SecureStringToBSTR(ciphertext);
    plaintext = new byte[ciphertext.Length];
    unsafe
    {
        // Copy without null bytes
        byte* bstr = (byte*)ptr;
        for (int i = 0; i < plaintext.Length; i++)
        {
            plaintext[i] = *bstr ++;
            *bstr = *bstr++;
        }
    }
}
```

Once the login is completed, the byte array will need to be cleared and unpinned.

```
private static void ClearPasswordFromMemory(IntPtr ptr, ref byte[] plaintext, SecureString password, GCHandle gcHandle)
{
    password.Dispose();
    Array.Clear(plaintext, 0, plaintext.Length);
    gcHandle.Free();
    if (ptr != IntPtr.Zero)
        Marshal.ZeroFreeBSTR(ptr);
}
```

Finally, whether you're using streams or an HttpClient for your requests, you'll just need to craft the body as a byte array and remember to clear and dispose of everything as

usual. And there you go!

Closing Thoughts

The system that the application is running on could be compromised, and I believe the application should therefore be designed as securely as possible independent of where it is running. But this code is only cutting down on the time the plain text password is in memory and providing control over when it's cleared rather than depending on the garbage collector. If the password's value is going to be used for anything on another machine, it will need to be exposed as plain text, and there's no way around it.
Microsoft says a SecureString is a *little* better than using a string. But if you're set on using a PasswordBox in some sort of form, those are your two choices.

Source Code

- Basic
- Digest
- Kerberos
- Negotiate
- NTLM
- NetworkCredential