Introduction to Hacking Thick Clients: Part 5 – The API

written by Austin Altmann | June 11, 2020

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:

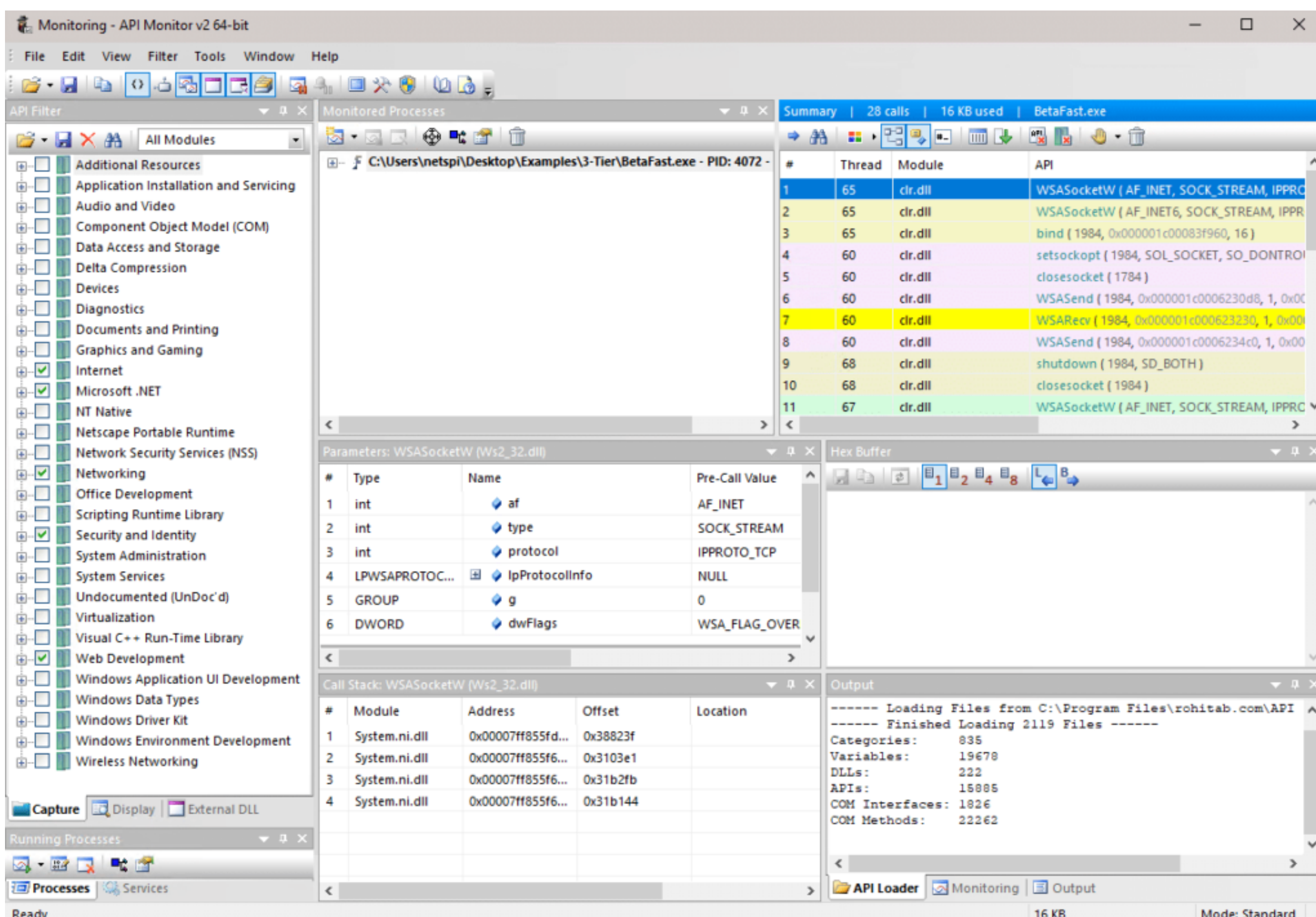
1. The GUI
2. The Network
3. The File System and Registry
4. The Assemblies
5. The API
6. The Memory

Information Gathering

When testing any application, it's important to have a good understanding of what's going on beneath the surface. Consider the largest desktop application you've ever seen – that's like half of the thick client applications I test. They're monuments to teamwork and the human spirit. I once had six hours of demos for a test, and the scope didn't even cover the entire application. Even if the code isn't obfuscated, there's no way I can read through and understand everything that's going on. Thankfully, people have made tools to help.

API Monitor

One tool in our arsenal is [API Monitor](). This application monitors all of the API calls in a process and displays their parameters. In this example, the Summary is a list of API calls made during the authentication to BetaFast. Note the API Filter on the left. When I had all of them checked and authenticated to BetaFast, the Summary almost immediately had over 500,000 API calls logged. And that takes up some serious resources.



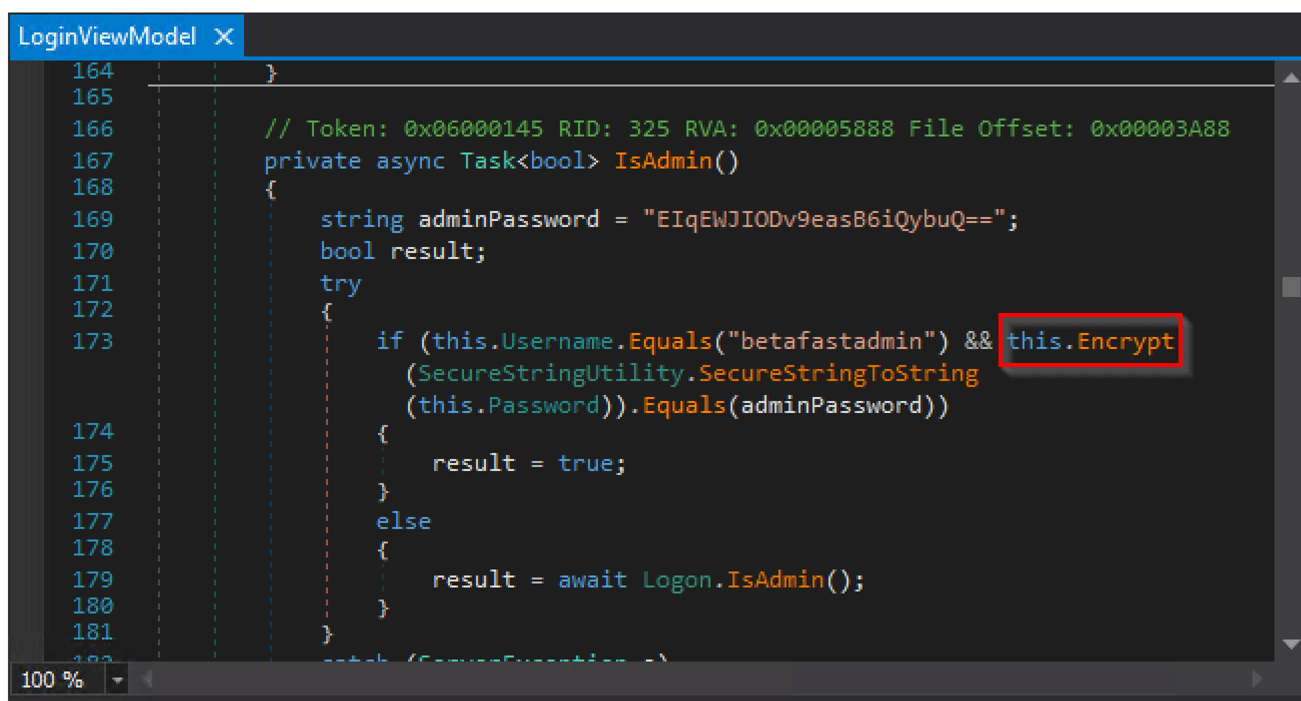
Monitoring BetaFast with API Monitor didn't highlight any particular vulnerabilities from my experience, and almost all of the main functionality is contained to a single assembly, so API monitor won't find the most sensitive data being sent between libraries. But it's a valuable tool for displaying information on which system resources are being used, which files are being referenced or created, network calls, security calls, etc. Moving beyond information gathering and into attacks, it is possible to intercept and modify data before

encryption, or to gather cleartext passwords and connection strings as they're passed between libraries.

Issuing Calls

Welcome back to the high level. It's much nicer up here. There's a tool called .net SmokeTest that can be used to load and run individual method calls. No need to load the program and search for where a method call takes place, attach the application to a debugger, or write a single line of code. If a test requires writing a method, .net SmokeTest is an easy way to execute it rather than finding the right spot for it to be called in the application. But it's especially useful for testing various data inputs in prewritten methods.

Take this encrypt method . . . please! Using a decompiler as outlined in the previous blog post, I found a hardcoded encrypted password in BetaFast.exe. Specifically, in the LoginViewModel class. Upon login, the password from the login form is encrypted and compared to this stored value.



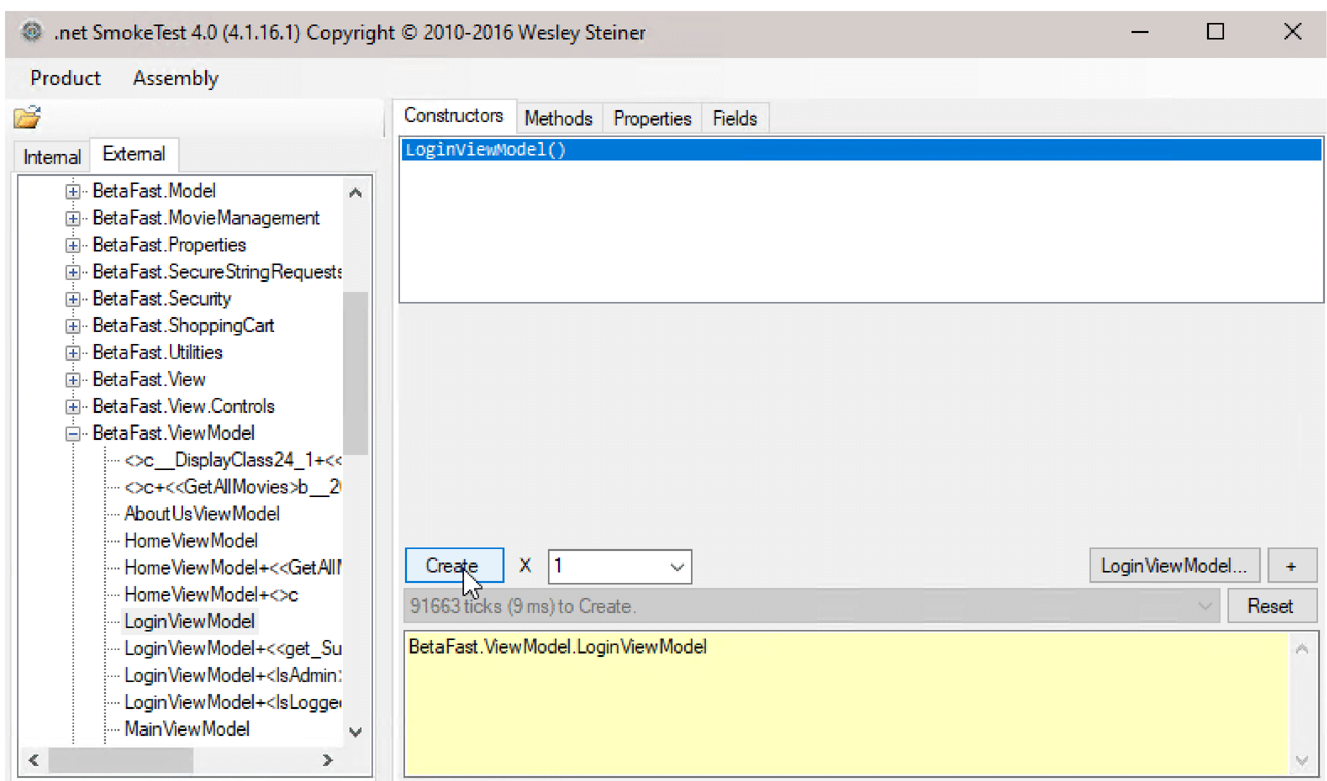
The method is written below.

```
LoginViewModel X
204         }
205
206         // Token: 0x06000146 RID: 326 RVA: 0x000058D0 File Offset: 0x00003AD0
207         private string Encrypt(string plaintext)
208         {
209             byte[] ciphertextBytes;
210             using (Aes aes = Aes.Create())
211             {
212                 aes.Key = this.Key;
213                 aes.IV = this.IV;
214                 ICryptoTransform encryptor = aes.CreateEncryptor(aes.Key,
                      aes.IV);
215                 using (MemoryStream msEncrypt = new MemoryStream())
216                 {
217                     using (CryptoStream csEncrypt = new CryptoStream
                        (msEncrypt, encryptor, CryptoStreamMode.Write))
218                     {
219                         using (StreamWriter swEncrypt = new StreamWriter
                        (csEncrypt))
220                         {
221                             swEncrypt.Write(plaintext);
222                         }
223                         ciphertextBytes = msEncrypt.ToArray();
224                     }
225                 }
226             }
227             return Convert.ToBase64String(ciphertextBytes);
228         }
229
100 %
```

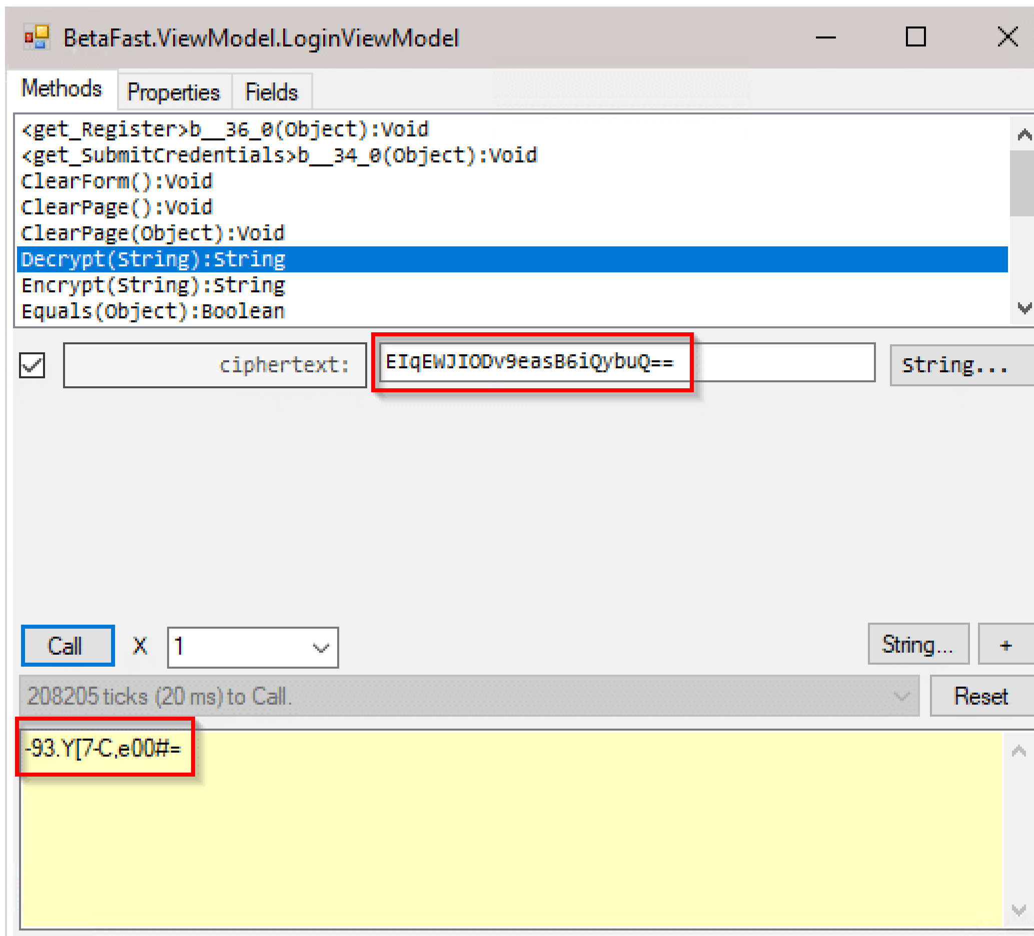
Even further below is a Decrypt method, but it’s never called in the application. Since the IV and Key are hardcoded in the application, it just needs a ciphertext input.

```
LoginViewModel ×
229
230        // Token: 0x06000147 RID: 327 RVA: 0x000059A0 File Offset: 0x00003BA0
231        private string Decrypt(string ciphertext)
232        {
233            string plaintext = string.Empty;
234            byte[] ciphertextBytes = Convert.FromBase64String(ciphertext);
235            using (Aes aes = Aes.Create())
236            {
237                aes.Key = this.Key;
238                aes.IV = this.IV;
239                ICryptoTransform decryptor = aes.CreateDecryptor(aes.Key,
                     aes.IV);
240                using (MemoryStream msDecrypt = new MemoryStream
                     (ciphertextBytes))
241                {
242                    using (CryptoStream csDecrypt = new CryptoStream
                       (msDecrypt, decryptor, CryptoStreamMode.Read))
243                    {
244                        using (StreamReader srDecrypt = new StreamReader
                       (csDecrypt))
245                        {
246                            plaintext = srDecrypt.ReadToEnd();
247                        }
248                    }
249                }
250            }
251            return plaintext;
252        }
100 %
```

It makes sense that the next step would be calling Decrypt on the hardcoded encrypted password. First, start up .net SmokeTest and create an instance of the LoginViewModel class.



Open the LoginViewModel class and enter the stored password as the ciphertext parameter in the Decrypt method. The plaintext password is then displayed below.



There's a NetSPI blog post from years back that highlights a Powershell alternative. The following code can be run in to call specific methods.

```
# Load Assembly
$Assembly = [System.Reflection.Assembly]::LoadFile("C:\Users\netspi\Desktop\Examples\3-Tier\BetaFast.exe")

# Create binding
$BindingFlags= [Reflection.BindingFlags] "NonPublic,Instance"

# Target the class
$Instance = new-object "BetaFast.ViewModel.LoginViewModel"

# Call the methods
$Instance.GetType().GetMethod("Encrypt",$BindingFlags).Invoke(
```

```
$Instance,"Spring1980!")
$Instance.GetType().GetMethod("Decrypt",$BindingFlags).Invoke(
$Instance,"PE/jSP7tOGDLoZLXRvPtlA==")
```

Spring1980! is encrypted, and PE/jSP7tOGDLoZLXRvPtlA== is decrypted.

```
PS C:\Users\netspi> # Load Assembly
$Assembly = [System.Reflection.Assembly]::LoadFile("C:\Users\netspi\Desktop\Examples\3-Tier\BetaFast.exe")

# Create binding
$BindingFlags= [Reflection.BindingFlags] "NonPublic,Instance"

# Target the class
$Instance = new-object "BetaFast.ViewModel.LoginViewModel"

# Call the methods
$Instance.GetType().GetMethod("Encrypt",$BindingFlags).Invoke($Instance,"Spring1980!")
$Instance.GetType().GetMethod("Decrypt",$BindingFlags).Invoke($Instance,"PE/jSP7tOGDLoZLXRvPtlA==")

PE/jSP7tOGDLoZLXRvPtlA==
Spring1980!
```