# WoW64 internals

WoW64 - aka Windows (32-bit) on Windows (64-bit) - is a subsystem that enables 32-bit Windows applications to run on 64-bit Windows. Most people today are familiar with WoW64 on Windows **x64**, where they can run **x86** applications. WoW64 has been with us since Windows XP, and x64 wasn't the only architecture where WoW64 has been available - it was available on **IA-64** architecture as well, where WoW64 has been responsible for emulating x86. Newly, WoW64 is also available on **ARM64**, enabling emulation of both **x86** and **ARM32** appllications.

MSDN offers brief article on WoW64 implementation details. We can find that WoW64 consists of (ignoring IA-64):

- Translation support DLLs:
  - `wow64.dll`: translation of `Nt*` system calls (`ntoskrnl.exe` / `ntdll.dll`)
  - `wow64win.dll`: translation of `NtGdi*`, `NtUser*` and other GUI-related system calls (`win32k.sys` / `win32u.dll`)
- Emulation support DLLs:
  - `wow64cpu.dll`: support for running x86 programs on x64
  - `wowarmhw.dll`: support for running ARM32 programs on ARM64
  - `xtajit.dll`: support for running x86 programs on ARM64

Besides `Nt*` system call translation, the `wow64.dll` provides the core emulation infrastructure.

If you have previous experience with reversing WoW64 on x64, you can notice that it shares plenty of common code with WoW64 subsystem on ARM64. Especially if you peeked into WoW64 of recent x64 Windows, you may have noticed that it actually contains strings such as `SysArm32` and that some functions check against `IMAGE_FILE_MACHINE_ARMNT (0x1C4)` machine type:

```
UNICODE_STRING *__fastcall Wow64SelectSystem32PathInternal(USHORT MachineType)
{
  switch ( MachineType )
  {
    case IMAGE_FILE_MACHINE_TARGET_HOST:
      return &hostDir;                          // "\\system32"
    case IMAGE_FILE_MACHINE_I386:
      return &x86Dir;                           // "\\SysWOW64"
    case IMAGE_FILE_MACHINE_ARMNT:
      return &armDir;                           // "\\SysArm32"
    case IMAGE_FILE_MACHINE_AMD64:
      return &amd64Dir;                         // "\\SysX8664"
    case IMAGE_FILE_MACHINE_ARM64:
      return &arm64Dir;                         // "\\SysArm64"
  }
  return 0i64;
}
```

Wow64SelectSystem32PathInternal found in wow64.dll on Windows x64

```
ULONG __fastcall Wow64ArchGetSP(USHORT MachineType, _CONTEXT_UNION *Context)
{
  if ( MachineType == IMAGE_FILE_MACHINE_I386 ) // 0x14c
    return Context->X86.Esp;
  if ( MachineType == IMAGE_FILE_MACHINE_ARMNT )// 0x1c4
    return Context->ARM32.Sp;
  return 0;
}
```

Wow64ArchGetSP found in wow64.dll on Windows x64

WoW on x64 systems cannot emulate ARM32 though - it just apparently shares common code. But `SysX8664` and `SysArm64` sound particularly interesting!

Those similarities can help anyone who is fluent in x86/x64, but not that much in ARM. Also, HexRays decompiler produce much better output for x86/x64 than for ARM32/ARM64.

Initially, my purpose with this blogpost was to get you familiar with how WoW64 works for ARM32 programs on ARM64. But because WoW64 itself changed a lot with Windows 10, and because WoW64 shares some similarities between x64 and ARM64, I decided to briefly get you through how WoW64 works in general.

Everything presented in this article is based on Windows 10 - insider preview, build 18247.

## Table of contents

## Terms

Througout this article I'll be using some terms I'd like to explain beforehand:

- `ntdll` or `ntdll.dll` - these will be always refering to the native `ntdll.dll` (x64 on Windows x64, ARM64 on Windows ARM64, ...), until said otherwise or until the context wouldn't indicate otherwise.
- `ntdll32` or `ntdll32.dll` - to make an easy distinction between native and WoW64 `ntdll.dll`, **any** WoW64 `ntdll.dll` will be refered with the `*32` suffix.
- `emu` or `emu.dll` - these will represent any of the emulation support DLLs (one of `wow64cpu.dll`, `wowarmhw.dll`, `xtajit.dll`)
- `module!FunctionName` - refers to a symbol `FunctionName` within the `module`. If you're familiar with WinDbg, you're already familiar with this notation.
- `CHPE` - "compiled-hybrid-PE", a new type of PE file, which looks as if it was x86 PE file, but has ARM64 code within them. `CHPE` will be tackled in more detail in the x86 on ARM64 section.
- The terms **emulation** and **binary-translation** refer to the WoW64 workings and they may be used interchangeably.

## Kernel

This section shows some points of interest in the `ntoskrnl.exe` regarding to the WoW64 initialization. If you're interested only in the user-mode part of the WoW64, you can skip this part to the Initialization of the WoW64 process.

### Kernel (initialization)

Initalization of WoW64 begins with the initialization of the kernel:

- `nt!KiSystemStartup`
- `nt!KiInitializeKernel`
- `nt!InitBootProcessor`
- `nt!PspInitPhase0`
- `nt!Phase1Initialization`
    - `nt!IoInitSystem`
        - `nt!IoInitSystemPreDrivers`
        - `nt!PsLocateSystemDlls`

`nt!PsLocateSystemDlls` routine takes a pointer named `nt!PspSystemDlls`, and then calls `nt!PspLocateSystemDll` in a loop. Let's figure out what's going on here:

```
PAGEDATA:00000001408EA218 ; PPS_SYSTEM_DLL_DATA PspSystemDlls[6]
PAGEDATA:00000001408EA218 PspSystemDlls   dq offset PspNativeSystemDllData
PAGEDATA:00000001408EA218                                       ; DATA XREF: PsMapSystemDlls+4D↑o
PAGEDATA:00000001408EA218                                       ; PspPrepareSystemDllInitBlock+96↑r ...
PAGEDATA:00000001408EA220                 dq offset PspWowX86SystemDllData ; SectionObject
PAGEDATA:00000001408EA228                 dq offset PspWowArm32SystemDllData
PAGEDATA:00000001408EA230                 dq 0
PAGEDATA:00000001408EA238                 dq 0
PAGEDATA:00000001408EA240 ; PPS_SYSTEM_DLL_DATA off_1408EA240
PAGEDATA:00000001408EA240 off_1408EA240   dq offset PspVsmEnclaveRuntimeDllData
```

PspSystemDlls (x64)

```
PAGEDATA:00000001408A4210 ; PPS_SYSTEM_DLL_DATA PspSystemDlls[6]
PAGEDATA:00000001408A4210 PspSystemDlls   DCQ PspNativeSystemDllData
PAGEDATA:00000001408A4210                                       ; DATA XREF: PspPrepareSystemDllInitBlock+88↑o
PAGEDATA:00000001408A4210                                       ; PspPrepareSystemDllInitBlock+8C↑r ...
PAGEDATA:00000001408A4218                 DCQ PspWowX86SystemDllData
PAGEDATA:00000001408A4220                 DCQ PspWowArm32SystemDllData ; SectionObject (runtime init)
PAGEDATA:00000001408A4228                 DCQ PspWowAmd64SystemDllData
PAGEDATA:00000001408A4230                 DCQ PspWowChpeX86SystemDllData
PAGEDATA:00000001408A4238                 DCQ 0
```

PspSystemDlls (ARM64)

`nt!PspSystemDlls` appears to be array of pointers to some structure, which holds some NTDLL-related data. The order of these NTDLLs corresponds with this `enum` (included in the **PDB**):

```
typedef enum _SYSTEM_DLL_TYPE
{
PsNativeSystemDll = 0,
PsWowX86SystemDll = 1,
PsWowArm32SystemDll = 2,
PsWowAmd64SystemDll = 3,
PsWowChpeX86SystemDll = 4,
PsVsmEnclaveRuntimeDll = 5,
PsSystemDllTotalTypes = 6,
} SYSTEM_DLL_TYPE;
```

view raw SYSTEM_DLL_TYPE.h hosted with ❤ by GitHub

Now, let's look how such structure looks like:

```
PAGEDATA:00000001408EA950 ; PPS_SYSTEM_DLL_DATA PspWowX86SystemDllData
PAGEDATA:00000001408EA950 PspWowX86SystemDllData dq 0             ; DATA XREF: PAGEDATA:00000001408EA220↑o
PAGEDATA:00000001408EA950                                         ; SectionObject
PAGEDATA:00000001408EA958                 dq 0                    ; PushLock
PAGEDATA:00000001408EA960                 dw 8                    ; Flags
PAGEDATA:00000001408EA962                 dw 14Ch                 ; MachineType
PAGEDATA:00000001408EA964                 dd 0
PAGEDATA:00000001408EA968                 dw 3Ch                  ; UNICODE_STRING.Length
PAGEDATA:00000001408EA96A                 dw 3Eh                  ; UNICODE_STRING.MaximumLength
PAGEDATA:00000001408EA96C                 dd 0                    ; --- padding ---
PAGEDATA:00000001408EA970                 dq offset aSystemrootSysw_0 ; "\\SystemRoot\\SysWOW64\\ntdll.dll"
PAGEDATA:00000001408EA978                 dq 0                    ; ImageBase (runtime init)
PAGEDATA:00000001408EA980                 dq offset aNtdll32Dll   ; "ntdll32.dll"
PAGEDATA:00000001408EA988                 dq 0                    ; --- unused ---
PAGEDATA:00000001408EA990                 dq 0                    ; SectionRelocationInformation (runtime init)
PAGEDATA:00000001408EA998                 dq 0                    ; --- unused ---
PAGEDATA:00000001408EA9A0 ; PPS_SYSTEM_DLL_DATA PspNativeSystemDllData
PAGEDATA:00000001408EA9A0 PspNativeSystemDllData dq 0             ; DATA XREF: PAGEDATA:PspSystemDlls↑o
PAGEDATA:00000001408EA9A8                 dq 0
PAGEDATA:00000001408EA9B0                 dw 13h                  ; Flags
PAGEDATA:00000001408EA9B2                 dw 0                    ; MachineType (native)
PAGEDATA:00000001408EA9B4                 dd 0
PAGEDATA:00000001408EA9B8                 dw 3Ch                  ; UNICODE_STRING
PAGEDATA:00000001408EA9BA                 dw 3Eh
PAGEDATA:00000001408EA9BC                 dd 0
PAGEDATA:00000001408EA9C0                 dq offset aSystemrootSyst_14 ; "\\SystemRoot\\System32\\ntdll.dll"
PAGEDATA:00000001408EA9C8                 dq 0
PAGEDATA:00000001408EA9D0                 dq offset aNtdllDll     ; "ntdll.dll"
PAGEDATA:00000001408EA9D8                 dq 0
PAGEDATA:00000001408EA9E0                 dq 0
PAGEDATA:00000001408EA9E8                 dq 0
PAGEDATA:00000001408EA9F0 ; PPS_SYSTEM_DLL_DATA PspVsmEnclaveRuntimeDllData
PAGEDATA:00000001408EA9F0 PspVsmEnclaveRuntimeDllData dq 0        ; DATA XREF: PAGEDATA:off_1408EA240↑o
PAGEDATA:00000001408EA9F8                 dq 0
PAGEDATA:00000001408EAA00                 dw 12h                  ; Flags
PAGEDATA:00000001408EAA02                 dw 0                    ; MachineType
PAGEDATA:00000001408EAA04                 dd 0
PAGEDATA:00000001408EAA08                 dw 40h                  ; UNICODE_STRING
PAGEDATA:00000001408EAA0A                 dw 42h
PAGEDATA:00000001408EAA0C                 dd 0
PAGEDATA:00000001408EAA10                 dq offset aSystemrootSyst_9 ; "\\SystemRoot\\System32\\vertdll.dll"
PAGEDATA:00000001408EAA18                 dq 0
PAGEDATA:00000001408EAA20                 dq offset aVertdllDll   ; "vertdll.dll"
PAGEDATA:00000001408EAA28                 dq 0
PAGEDATA:00000001408EAA30                 dq 0
PAGEDATA:00000001408EAA38                 dq 0
```

SystemDllData (x64)

```
PAGEDATA:00000001408A4950 PspWowArm32SystemDllData DCQ 0          ; DATA XREF: PAGEDATA:00000001408A4220↑o
PAGEDATA:00000001408A4950                                         ; SectionObject (runtime init)
PAGEDATA:00000001408A4958                 DCQ 0                   ; PushLock (runtime init)
PAGEDATA:00000001408A4960                 DCW 8                   ; Flags
PAGEDATA:00000001408A4962                 DCW 0x1C4               ; MachineType
PAGEDATA:00000001408A4964                 DCD 0
PAGEDATA:00000001408A4968                 DCW 0x3C                ; UNICODE_STRING.Length
PAGEDATA:00000001408A496A                 DCW 0x3E                ; UNICODE_STRING.MaximumLength
PAGEDATA:00000001408A496C                 DCD 0                   ; --- padding --
PAGEDATA:00000001408A4970                 DCQ aSystemrootSysa     ; "\\SystemRoot\\SysArm32\\ntdll.dll"
PAGEDATA:00000001408A4978                 DCQ 0                   ; ImageBase (runtime init)
PAGEDATA:00000001408A4980                 DCQ aNtdll32Dll         ; "ntdll32.dll"
PAGEDATA:00000001408A4988                 DCQ 0                   ; --- unused ---
PAGEDATA:00000001408A4990                 DCQ 0                   ; SectionRelocationInformation (runtime init)
PAGEDATA:00000001408A4998                 DCQ 0                   ; --- unused ---
PAGEDATA:00000001408A49A0 PspWowX86SystemDllData DCQ 0            ; DATA XREF: PAGEDATA:00000001408A4218↑o
PAGEDATA:00000001408A49A8                 DCQ 0
PAGEDATA:00000001408A49B0                 DCW 8
PAGEDATA:00000001408A49B2                 DCW 0x14C               ; MachineType
PAGEDATA:00000001408A49B4                 DCD 0
PAGEDATA:00000001408A49B8                 DCW 0x3C                ; UNICODE_STRING
PAGEDATA:00000001408A49BA                 DCW 0x3E
PAGEDATA:00000001408A49BC                 DCD 0
PAGEDATA:00000001408A49C0                 DCQ aSystemrootSysw_0   ; "\\SystemRoot\\SysWOW64\\ntdll.dll"
PAGEDATA:00000001408A49C8                 DCQ 0
PAGEDATA:00000001408A49D0                 DCQ aNtdll32Dll         ; "ntdll32.dll"
PAGEDATA:00000001408A49D8                 DCQ 0
PAGEDATA:00000001408A49E0                 DCQ 0
PAGEDATA:00000001408A49E8                 DCQ 0
PAGEDATA:00000001408A49F0 PspNativeSystemDllData DCQ 0            ; DATA XREF: PsShutdownSystem+1C4↑o
```

SystemDllData (ARM64)

The `nt!PspLocateSystemDll` function intializes fields of this structure. The layout of this structure isn't unfortunatelly in the **PDB**, but you can find a reconstructed version in the appendix.

Now let's get back to the `nt!Phase1Initialization` - there's more:

- `...`
- `nt!Phase1Initialization`
  - `nt!Phase1InitializationIoReady`
    - `nt!PspInitPhase2`
    - `nt!PspInitializeSystemDlls`

`nt!PspInitializeSystemDlls` routine takes a pointer named `nt!NtdllExportInformation`. Let's look at it:

```
INIT:000000014097A6D0 NtdllExportInformation dq offset NtdllExports
INIT:000000014097A6D8 _              dq 19                   ; DATA XREF: PspInitializeSystemDlls+1A↑o
INIT:000000014097A6E0                dq offset NtdllWowX86Exports
INIT:000000014097A6E8                dq 8
INIT:000000014097A6F0                dq offset NtdllWowArm32Exports
INIT:000000014097A6F8                dq 8, 6 dup(0)
```

NtdllExportInformation (x64)

```
INIT:000000014092C7C0 NtdllExportInformation DCQ NtdllExports ; DATA XREF: PspInitializeSystemDlls+18↑o
INIT:000000014092C7C0                                         ; PspInitializeSystemDlls+1C↑o
INIT:000000014092C7C8                DCQ 14
INIT:000000014092C7D0                DCQ NtdllWowX86Exports
INIT:000000014092C7D8                DCQ 8
INIT:000000014092C7E0                DCQ NtdllWowArm32Exports
INIT:000000014092C7E8                DCQ 8
INIT:000000014092C7F0                DCQ NtdllWowAmd64Exports
INIT:000000014092C7F8                DCQ 8
INIT:000000014092C800                DCQ NtdllWowChpeX86Exports
INIT:000000014092C808                DCQ 8, 0, 0
```

NtdllExportInformation (ARM64)

It looks like it's some sort of array, again, ordered by the `enum _SYSTEM_DLL_TYPE`. Let's examine `NtdllExports`:

```
INIT:000000014097AA30 NtdllExports    dq offset aLdrinitializet
INIT:000000014097AA30                                         ; DATA XREF: INIT:NtdllExportInformation↑o
INIT:000000014097AA30                                         ; "LdrInitializeThunk"
INIT:000000014097AA38                dq offset PspLoaderInitRoutine
INIT:000000014097AA40                dq offset aRtluserthreads ; "RtlUserThreadStart"
INIT:000000014097AA48                dq offset PspUserThreadStart
INIT:000000014097AA50                dq offset aKiuserexceptio ; "KiUserExceptionDispatcher"
INIT:000000014097AA58                dq offset KeUserExceptionDispatcher
INIT:000000014097AA60                dq offset aKiuserapcdispa ; "KiUserApcDispatcher"
INIT:000000014097AA68                dq offset KeUserApcDispatcher
INIT:000000014097AA70                dq offset aKiusercallback ; "KiUserCallbackDispatcher"
INIT:000000014097AA78                dq offset KeUserCallbackDispatcher
INIT:000000014097AA80                dq offset aKiraiseuserexc ; "KiRaiseUserExceptionDispatcher"
INIT:000000014097AA88                dq offset KeRaiseUserExceptionDispatcher
INIT:000000014097AA90                dq offset aRtlpexecuteums ; "RtlpExecuteUmsThread"
INIT:000000014097AA98                dq offset KeExecuteUmsThread
INIT:000000014097AAA0                dq offset aRtlpumsthready ; "RtlpUmsThreadYield"
INIT:000000014097AAA8                dq offset KeUmsThreadYield
INIT:000000014097AAB0                dq offset aRtlpumsexecute ; "RtlpUmsExecuteYieldThreadEnd"
INIT:000000014097AAB8                dq offset KeUmsExecuteYieldThreadEnd
INIT:000000014097AAC0                dq offset aExpinterlocked_2 ; "ExpInterlockedPopEntrySListEnd"
INIT:000000014097AAC8                dq offset KeUserPopEntrySListEnd
INIT:000000014097AAD0                dq offset aExpinterlocked_4 ; "ExpInterlockedPopEntrySListFault"
INIT:000000014097AAD8                dq offset KeUserPopEntrySListFault
INIT:000000014097AAE0                dq offset aExpinterlocked_1 ; "ExpInterlockedPopEntrySListResume"
INIT:000000014097AAE8                dq offset KeUserPopEntrySListResume
INIT:000000014097AAF0                dq offset aLdrsystemdllin ; "LdrSystemDllInitBlock"
INIT:000000014097AAF8                dq offset PspSystemDllInitBlock
INIT:000000014097AB00                dq offset aRtlpfreezetime ; "RtlpFreezeTimeBias"
INIT:000000014097AB08                dq offset PspFreezeTimeBiasAddress
INIT:000000014097AB10                dq offset aKiuserinverted ; "KiUserInvertedFunctionTable"
INIT:000000014097AB18                dq offset KeUserInvertedFunctionTable
INIT:000000014097AB20                dq offset aWerreportexcep ; "WerReportExceptionWorker"
INIT:000000014097AB28                dq offset DbgkWerReportExceptionWorker
INIT:000000014097AB30                dq offset aRtlcallenclave ; "RtlCallEnclaveReturn"
INIT:000000014097AB38                dq offset PspCallEnclaveReturn
INIT:000000014097AB40                dq offset aRtlenclavecall ; "RtlEnclaveCallDispatch"
INIT:000000014097AB48                dq offset PspEnclaveDispatch
INIT:000000014097AB50                dq offset aRtlenclavecall_0 ; "RtlEnclaveCallDispatchReturn"
INIT:000000014097AB58                dq offset PspEnclaveDispatchReturn
```

NtdllExportInformation (x64)

Nothing unexpected - just tuples of **function name** and **function pointer**. Did you notice the difference in the number after the `NtdllExports` field? On x64 there is **19** meanwhile on ARM64 there is **14**. This number represents number of items in `NtdllExports` - and indeed, there is slightly different set of them:

| x64 | ARM64 |
| --- | --- |
| (0) `LdrInitializeThunk` | (0) `LdrInitializeThunk` |
| (1) `RtlUserThreadStart` | (1) `RtlUserThreadStart` |

| x64 | ARM64 |
|---|---|
| (2) `KiUserExceptionDispatcher` | (2) `KiUserExceptionDispatcher` |
| (3) `KiUserApcDispatcher` | (3) `KiUserApcDispatcher` |
| (4) `KiUserCallbackDispatcher` | (4) `KiUserCallbackDispatcher` |
| - | (5) `KiUserCallbackDispatcherReturn` |
| (5) `KiRaiseUserExceptionDispatcher` | (6) `KiRaiseUserExceptionDispatcher` |
| (6) `RtlpExecuteUmsThread` | - |
| (7) `RtlpUmsThreadYield` | - |
| (8) `RtlpUmsExecuteYieldThreadEnd` | - |
| (9) `ExpInterlockedPopEntrySListEnd` | (7) `ExpInterlockedPopEntrySListEnd` |
| (10) `ExpInterlockedPopEntrySListFault` | (8) `ExpInterlockedPopEntrySListFault` |
| (11) `ExpInterlockedPopEntrySListResume` | (9) `ExpInterlockedPopEntrySListResume` |
| (12) `LdrSystemDllInitBlock` | (10) `LdrSystemDllInitBlock` |
| (13) `RtlpFreezeTimeBias` | (11) `RtlpFreezeTimeBias` |
| (14) `KiUserInvertedFunctionTable` | (12) `KiUserInvertedFunctionTable` |
| (15) `WerReportExceptionWorker` | (13) `WerReportExceptionWorker` |
| (16) `RtlCallEnclaveReturn` | - |
| (17) `RtlEnclaveCallDispatch` | - |
| (18) `RtlEnclaveCallDispatchReturn` | - |

We can see that ARM64 is missing `Ums` (User-Mode Scheduling) and `Enclave` functions. Also, we can see that ARM64 has one extra function: `KiUserCallbackDispatcherReturn`.

On the other hand, all `NtdllWow*Exports` contain the same set of function names:

```
INIT:000000014092C870                   DCQ 0x22000000020, 0
INIT:000000014092C8B0 NtdllWowX86Exports DCQ aLdrinitializet  ; DATA XREF: INIT:000000014092C7D0↑o
INIT:000000014092C8B0                                         ; "LdrInitializeThunk"
INIT:000000014092C8B8                   DCQ PsWowX86SharedInformation
INIT:000000014092C8C0                   DCQ aKiuserexceptio     ; "KiUserExceptionDispatcher"
INIT:000000014092C8C8                   DCQ qword_1408A6478
INIT:000000014092C8D0                   DCQ aKiuserapcdispa     ; "KiUserApcDispatcher"
INIT:000000014092C8D8                   DCQ qword_1408A6480
INIT:000000014092C8E0                   DCQ aKiusercallback     ; "KiUserCallbackDispatcher"
INIT:000000014092C8E8                   DCQ qword_1408A6488
INIT:000000014092C8F0                   DCQ aRtluserthreads     ; "RtlUserThreadStart"
INIT:000000014092C8F8                   DCQ qword_1408A6490
INIT:000000014092C900                   DCQ aRtlpqueryproce     ; "RtlpQueryProcessDebugInformationRemote"
INIT:000000014092C908                   DCQ qword_1408A6498
INIT:000000014092C910                   DCQ aLdrsystemdllin     ; "LdrSystemDllInitBlock"
INIT:000000014092C918                   DCQ qword_1408A64A8
INIT:000000014092C920                   DCQ aRtlpfreezetime     ; "RtlpFreezeTimeBias"
INIT:000000014092C928                   DCQ qword_1408A64B0
INIT:000000014092C930 NtdllWowChpeX86Exports DCQ aLdrinitializet
INIT:000000014092C930                                         ; DATA XREF: INIT:000000014092C800↑o
INIT:000000014092C930                                         ; "LdrInitializeThunk"
INIT:000000014092C938                   DCQ PsWowChpeX86SharedInformation
INIT:000000014092C940                   DCQ aKiuserexceptio     ; "KiUserExceptionDispatcher"
INIT:000000014092C948                   DCQ qword_1408A64F8
INIT:000000014092C950                   DCQ aKiuserapcdispa     ; "KiUserApcDispatcher"
INIT:000000014092C958                   DCQ qword_1408A6500
INIT:000000014092C960                   DCQ aKiusercallback     ; "KiUserCallbackDispatcher"
INIT:000000014092C968                   DCQ qword_1408A6508
INIT:000000014092C970                   DCQ aRtluserthreads     ; "RtlUserThreadStart"
INIT:000000014092C978                   DCQ qword_1408A6510
INIT:000000014092C980                   DCQ aRtlpqueryproce     ; "RtlpQueryProcessDebugInformationRemote"
INIT:000000014092C988                   DCQ qword_1408A6518
INIT:000000014092C990                   DCQ aLdrsystemdllin     ; "LdrSystemDllInitBlock"
INIT:000000014092C998                   DCQ qword_1408A6528
INIT:000000014092C9A0                   DCQ aRtlpfreezetime     ; "RtlpFreezeTimeBias"
INIT:000000014092C9A8                   DCQ qword_1408A6530
```

NtdllWowExports (ARM64)

Notice names of second fields of these “structures”: `PsWowX86SharedInformation`, `PsWowChpeX86SharedInformation`, … If we look at the address of those fields, we can see that they’re part of another array:

```
PAGEDATA:000000001408A646E                % 1
PAGEDATA:000000001408A646F                % 1
PAGEDATA:000000001408A6470 PsWowX86SharedInformation % 8          ; DATA XREF: PspWow64GetSharedInformation:loc_1404A5AD0↑o
PAGEDATA:000000001408A6470                                        ; PspWow64GetSharedInformation+34↑o ...
PAGEDATA:000000001408A6478 qword_1408A6478 % 8                    ; DATA XREF: INIT:000000014092C8C8↓o
PAGEDATA:000000001408A6480 qword_1408A6480 % 8                    ; DATA XREF: INIT:000000014092C8D8↓o
PAGEDATA:000000001408A6488 qword_1408A6488 % 8                    ; DATA XREF: INIT:000000014092C8E8↓o
PAGEDATA:000000001408A6490 qword_1408A6490 % 8                    ; DATA XREF: INIT:000000014092C8F8↓o
PAGEDATA:000000001408A6498 qword_1408A6498 % 8                    ; DATA XREF: INIT:000000014092C908↓o
PAGEDATA:000000001408A64A0                % 8
PAGEDATA:000000001408A64A8 qword_1408A64A8 % 8                    ; DATA XREF: INIT:000000014092C918↓o
PAGEDATA:000000001408A64B0 qword_1408A64B0 % 8                    ; DATA XREF: INIT:000000014092C928↓o
PAGEDATA:000000001408A64B8                % 1
```

PsWowX86SharedInformation (ARM64)

Those addresses are actually **targets** of the pointers in the `NtdllWow*Exports` structure. Also, those functions combined with `PsWow*SharedInformation` might give you hint that they're related to this `enum` (included in the **PDB**):

```
typedef enum _WOW64_SHARED_INFORMATION
{
SharedNtdll32LdrInitializeThunk = 0,
SharedNtdll32KiUserExceptionDispatcher = 1,
SharedNtdll32KiUserApcDispatcher = 2,
SharedNtdll32KiUserCallbackDispatcher = 3,
SharedNtdll32RtlUserThreadStart = 4,
SharedNtdll32pQueryProcessDebugInformationRemote = 5,
SharedNtdll32BaseAddress = 6,
SharedNtdll32LdrSystemDllInitBlock = 7,
SharedNtdll32RtlpFreezeTimeBias = 8,
Wow64SharedPageEntriesCount = 9,
} WOW64_SHARED_INFORMATION;
```

view raw WOW64_SHARED_INFORMATION.h hosted with ❤ by GitHub

Notice how the order of the `SharedNtdll32BaseAddress` corellates with the empty field in the previous screenshot (highlighted). The set of WoW64 NTDLL functions is same on both x64 and ARM64.

(The C representation of this data can be found in the appendix.)

Now we can tell what the `nt!PspInitializeSystemDlls` function does - it gets **image base** of each NTDLL (`nt!PsQuerySystemDllInfo`), resolves all `Ntdll*Exports` for them (`nt!RtlFindExportedRoutineByName`). Also, only for all WoW64 NTDLLs (`if ((SYSTEM_DLL_TYPE)SystemDllType > PsNativeSystemDll)`) it assigns the **image base** to the `SharedNtdll32BaseAddress` field of the `PsWow*SharedInformation` array (`nt!PspWow64GetSharedInformation`).

### Kernel (create process)

Let's talk briefly about process creation. As you probably already know, the native `ntdll.dll` is mapped as a first DLL into each created process. This applies for all architectures - **x86**, **x64** and also for **ARM64**. The WoW64 processes aren't exception to this rule - the WoW64 processes share the same initialization code path as native processes.

- `nt!NtCreateUserProcess`
- `nt!PspAllocateProcess`
  - `nt!PspSetupUserProcessAddressSpace`
    - `nt!PspPrepareSystemDllInitBlock`
    - `nt!PspWow64SetupUserProcessAddressSpace`
- `nt!PspAllocateThread`
  - `nt!PspWow64InitThread`
  - `nt!KeInitThread // Entry-point: nt!PspUserThreadStartup`
- `nt!PspUserThreadStartup`
- `nt!PspInitializeThunkContext`
  - `nt!KiDispatchException`

If you ever wondered how is the first user-mode instruction of the newly created process executed, now you know the answer - a "synthetic" user-mode exception is dispatched, with `ExceptionRecord.ExceptionAddress = &PspLoaderInitRoutine`, where `PspLoaderInitRoutine` points to the `ntdll!LdrInitializeThunk`. This is the first function that is executed in every process - including WoW64 processes.

## Initialization of the WoW64 process

The fun part begins!

> **NOTE:** Initialization of the `wow64.dll` is same on both x64 and ARM64. Eventual differences will be mentioned.

- `ntdll!LdrInitializeThunk`
- `ntdll!LdrpInitialize`
- `ntdll!_LdrpInitialize`

- `ntdll!LdrpInitializeProcess`
- `ntdll!LdrpLoadWow64`

The `ntdll!LdrpLoadWow64` function is called when the `ntdll!UseWOW64` global variable is `TRUE`, which is set when `NtCurrentTeb()->WowTebOffset != NULL`.

It constructs the full path to the `wow64.dll`, loads it, and then resolves following functions:

- `Wow64LdrpInitialize`
- `Wow64PrepareForException`
- `Wow64ApcRoutine`
- `Wow64PrepareForDebuggerAttach`
- `Wow64SuspendLocalThread`

> **NOTE:** The resolution of these pointers is wrapped between pair of `ntdll!LdrProtectMrdata` calls, responsible for protecting (1) and unprotecting (0) the `.mrdata` section - in which these pointers reside. `MRDATA` (Mutable Read Only Data) are part of the CFG (Control-Flow Guard) functionality. You can look at Alex's slides for more information.

When these functions are successfully located, the `ntdll.dll` finally transfers control to the `wow64.dll` by calling `wow64!Wow64LdrpInitialize`. Let's go through the sequence of calls that eventually bring us to the entry-point of the "emulated" application.

- `wow64!Wow64LdrpInitialize`
  - `wow64!Wow64InfoPtr = (NtCurrentPeb32() + 1)`
  - `NtCurrentTeb()->TlsSlots[/* 10 */ WOW64_TLS_WOW64INFO] = wow64!Wow64InfoPtr`
  - `ntdll!RtlWow64GetCpuAreaInfo`
  - `wow64!ProcessInit`
  - `wow64!CpuNotifyMapViewOfSection // Process image`
  - `wow64!Wow64DetectMachineTypeInternal`
  - `wow64!Wow64SelectSystem32PathInternal`
  - `wow64!CpuNotifyMapViewOfSection // 32-bit NTDLL image`
  - `wow64!ThreadInit`
  - `wow64!ThunkStartupContext64TO32`
  - `wow64!Wow64SetupInitialCall`
  - `wow64!RunCpuSimulation`
    - `emu!BTCpuSimulate`

`Wow64InfoPtr` is the first initialized variable in the `wow64.dll`. It contains data shared between 32-bit and 64-bit execution mode and its structure is not documented, although you can find this structure partialy restored in the appendix.

`RtlWow64GetCpuAreaInfo` is an internal `ntdll.dll` function which is called a lot during emulation. It is mainly used for fetching the machine type and architecture-specific CPU context (the `CONTEXT` structure) of the emulated process. This information is fetched into an undocumented structure, which we'll be calling `WOW64_CPU_AREA_INFO`. Pointer to this structure is then given to the `ProcessInit` function.

`Wow64DetectMachineTypeInternal` determines the machine type of the executed process and returns it. `Wow64SelectSystem32PathInternal` selects the "emulated" `System32` directory based on that machine type, e.g. `SysWOW64` for x86 processes or `SysArm32` for ARM32 processes.

You can also notice calls to `CpuNotifyMapViewOfSection` function. As the name suggests, it is also called on each "emulated" call of `NtMapViewOfSection`. This function:

- Checks if the mapped image is executable
- Checks if following conditions are true:
  - `NtHeaders->OptionalHeader.MajorSubsystemVersion == USER_SHARED_DATA.NtMajorVersion`
  - `NtHeaders->OptionalHeader.MinorSubsystemVersion == USER_SHARED_DATA.NtMinorVersion`

If these checks pass, `CpupResolveReverseImports` function is called. This function checks if the mapped image exports the `Wow64Transition` symbol and if so, it assigns there a **32-bit pointer value** returned by `emu!BTCpuGetBopCode`.

The `Wow64Transition` is mostly known to be exported by `SysWOW64\ntdll.dll`, but there are actually multiple of Windows' WoW DLLs which exports this symbol. You might be already familiar with the term "Heaven's Gate" - this is where the `Wow64Transition` will point to on Windows x64 - a simple far jump instruction which switches into long-mode (64-bit) enabled code segment. On ARM64, the `Wow64Transition` points to a "nop" function.

> **NOTE:** Because there are no checks on the `ImageName`, the `Wow64Transition` symbol is resolved for all executable images that passes the checks mentioned earlier. If you're wondering whether `Wow64Transition` would be resolved for your custom executable or DLL - it indeed would!

The initialization then continues with thread-specific initialization by calling `ThreadInit`. This is followed by pair of calls `ThunkStartupContext64TO32(CpuArea.MachineType, CpuArea.Context, NativeContext)` and `Wow64SetupInitialCall(&CpuArea)` - these functions perform the necessary setup of the architecture-specific WoW64 `CONTEXT` structure to prepare start of the execution in the emulated environment. This is done in the exact same way as if `ntoskrnl.exe` would actually executed the emulated application - i.e.:

- setting the instruction pointer to the address of `ntdll32!LdrInitializeThunk`
- setting the stack pointer below the WoW64 `CONTEXT` structure
- setting the 1st parameter to point to that `CONTEXT` structure
- setting the 2nd parameter to point to the base address of the `ntdll32`

Finally, the `RunCpuSimulation` function is called. This function just calls `BTCpuSimulate` from the binary-translator DLL, which contains the actual emulation loop that never returns.

#### `wow64!ProcessInit`

- `wow64!Wow64ProtectMrdata // 0`
- `wow64!Wow64pLoadLogDll`
    - `ntdll!LdrLoadDll // "%SystemRoot%\system32\wow64log.dll"`

`wow64.dll` has also it's own `.mrdata` section and `ProcessInit` begins with unprotecting it. It then tries to load the `wow64log.dll` from the constructed system directory. Note that this DLL is never present in any released Windows installation (it's probably used internally by Microsoft for debugging of the WoW64 subsystem). Therefore, load of this DLL will normally fail. This isn't problem, though, because no critical functionality of the WoW64 subsystem depends on it. If the load would actually succeed, the `wow64.dll` would try to find following exported functions there:

- `Wow64LogInitialize`
- `Wow64LogSystemService`
- `Wow64LogMessageArgList`
- `Wow64LogTerminate`

If any of these functions wouldn't be exported, the DLL would be immediately unloaded.

If we'd drop custom `wow64log.dll` (which would export functions mentioned above) into the `%SystemRoot%\System32` directory, it would actually get loaded into every WoW64 process. This way we could drop a custom logging DLL, or even inject every WoW64 process with native DLL!

For more details, you can check my injdrv project which implements injection of native DLLs into WoW64 processes, or check this post by Walied Assar.

Then, certain important values are fetched from the `LdrSystemDllInitBlock` array. These contains base address of the `ntdll32.dll`, pointer to functions like `ntdll32!KiUserExceptionDispatcher`, `ntdll32!KiUserApcDispatcher`, ..., control flow guard information and others.

Finally, the `Wow64pInitializeFilePathRedirection` is called, which - as the name suggests - initializes WoW64 path redirection. The path redirection is completely implemented in the `wow64.dll` and the mechanism is basically based on string replacement. The path redirection can be disabled and enabled by calling `kernel32!Wow64DisableWow64FsRedirection` & `kernel32!Wow64RevertWow64FsRedirection` function pairs. Both of these functions internally call `ntdll32!RtlWow64EnableFsRedirectionEx`, which directly operates on `NtCurrentTeb()->TlsSlots[/* 8 */ WOW64_TLS_FILESYSREDIR]` field.

**wow64!ServiceTables**

Next, a `ServiceTables` array is initialized. You might be already familiar with the `KSERVICE_TABLE_DESCRIPTOR` from the `ntoskrnl.exe`, which contains - among other things - a pointer to an array of system functions callable from the user-mode. `ntoskrnl.exe` contains 2 of these tables: one for `ntoskrnl.exe` itself and one for the `win32k.sys`, aka the Windows (GUI) subsystem. `wow64.dll` has 4 of them!

The `WOW64_SERVICE_TABLE_DESCRIPTOR` has the exact same structure as the `KSERVICE_TABLE_DESCRIPTOR`, except that it is extended:

```
typedef struct _WOW64_ERROR_CASE {
ULONG Case;
NTSTATUS TransformedStatus;
} WOW64_ERROR_CASE, *PWOW64_ERROR_CASE;
typedef struct _WOW64_SERVICE_TABLE_DESCRIPTOR {
KSERVICE_TABLE_DESCRIPTOR Descriptor;
WOW64_ERROR_CASE ErrorCaseDefault;
PWOW64_ERROR_CASE ErrorCase;
} WOW64_SERVICE_TABLE_DESCRIPTOR, *PWOW64_SERVICE_TABLE_DESCRIPTOR;
```

view raw 2_WOW64_SERVICE_TABLE_DESCRIPTOR.h hosted with ❤ by GitHub

(More detailed definition of this structure is in the appendix.)

`ServiceTables` array is populated as follows:

- `ServiceTables[/* 0 */ WOW64_NTDLL_SERVICE_INDEX] = sdwhnt32`
- `ServiceTables[/* 1 */ WOW64_WIN32U_SERVICE_INDEX] = wow64win!sdwhwin32`
- `ServiceTables[/* 2 */ WOW64_KERNEL32_SERVICE_INDEX = wow64win!sdwhcon`
- `ServiceTables[/* 3 */ WOW64_USER32_SERVICE_INDEX] = sdwhbase`

> **NOTE:** `wow64.dll` directly depends (by import table) on two DLLs: the native `ntdll.dll` and `wow64win.dll`. This means that `wow64win.dll` is loaded even into "non-Windows-subsystem" processes, that wouldn't normally load `user32.dll`.
>
> These two symbols mentioned above are the only symbols that `wow64.dll` requires `wow64win.dll` to export.

Let's have a look at `sdwhnt32` service table:

```
0000001800362A0 ; WOW64_SERVICE_TABLE_DESCRIPTOR sdwhbase
0000001800362A0 sdwhbase        WOW64_SERVICE_TABLE_DESCRIPTOR <offset sdwhbaseJumpTable, 0, 1000h, \
0000001800362A0                                                ; DATA XREF: ProcessInit+12E↑r
0000001800362A0                                                        offset sdwhbaseNumber, <0>, \
0000001800362A0                                                        offset sdwhbaseErrorCase>
0000001800362D0 ; WOW64_SERVICE_TABLE_DESCRIPTOR sdwhnt32
0000001800362D0 sdwhnt32        WOW64_SERVICE_TABLE_DESCRIPTOR <offset sdwhnt32JumpTable, 0, 1000h, \
0000001800362D0                                                ; DATA XREF: ProcessInit+C4↑r
0000001800362D0                                                        offset sdwhnt32Number, <0>, 0>
0000000180036300 ; BT_FUNC_ITEM off_180036300[33]
0000000180036300 off_180036300   dq offset aBtcpuprocessin
0000000180036300                                               ; DATA XREF: CpuLoadBinaryTranslator+97↑o
```

sdwhnt32 (x64)

```
0000001800366618                dq 0
000000180036620 sdwhnt32JumpTable dq offset whNtAccessCheck
000000180036620                                        ; DATA XREF: .rdata:sdwhnt32↑o
000000180036628                 dq offset whNtWorkerFactoryWorkerReady
000000180036630                 dq offset whNtAcceptConnectPort
000000180036638                 dq offset whNtMapUserPhysicalPagesScatter
000000180036640                 dq offset whNtWaitForSingleObject
000000180036648                 dq offset whNtCallbackReturn
000000180036650                 dq offset whNtReadFile
000000180036658                 dq offset whNtDeviceIoControlFile
000000180036660                 dq offset whNtWriteFile
000000180036668                 dq offset whNtRemoveIoCompletion
000000180036670                 dq offset whNtReleaseSemaphore
000000180036678                 dq offset whNtReplyWaitReceivePort
000000180036680                 dq offset whNtReplyPort
000000180036688                 dq offset whNtSetInformationThread
000000180036690                 dq offset whNtSetEvent
000000180036698                 dq offset whNtClose
0000001800366A0                 dq offset whNtQueryObject
0000001800366A8                 dq offset whNtQueryInformationFile
0000001800366B0                 dq offset whNtOpenKey
0000001800366B8                 dq offset whNtEnumerateValueKey
0000001800366C0                 dq offset whNtFindAtom
0000001800366C8                 dq offset whNtQueryDefaultLocale
```

sdwhnt32JumpTable (x64)

```
00000018003A2BA                 align 20h
00000018003A2C0 sdwhnt32Number  db 20h                  ; DATA XREF: .rdata:sdwhnt32↑o
00000018003A2C1                 db    4
00000018003A2C2                 db  18h
00000018003A2C3                 db  0Ch
00000018003A2C4                 db  0Ch
00000018003A2C5                 db  0Ch
00000018003A2C6                 db  24h ; $
00000018003A2C7                 db  28h ; (
00000018003A2C8                 db  24h ; $
00000018003A2C9                 db  14h
00000018003A2CA                 db  0Ch
00000018003A2CB                 db  10h
00000018003A2CC                 db    8
00000018003A2CD                 db  10h
00000018003A2CE                 db    8
00000018003A2CF                 db    4
00000018003A2D0                 db  14h
00000018003A2D1                 db  14h
00000018003A2D2                 db  0Ch
00000018003A2D3                 db  18h
00000018003A2D4                 db  0Ch
00000018003A2D5                 db    8
00000018003A2D6                 db  14h
```

sdwhnt32Number (x64)

There is nothing surprising for those who already dealt with service tables in `ntoskrnl.exe`. `sdwhnt32JumpTable` contains array of the system call functions, which are traditionaly prefixed. WoW64 “system calls” are prefixed with `wh*`, which honestly I don’t have any idea what it stands for - although it might be the case as with `Zw*` prefix - it stands for nothing and is simply used as an unique distinguisher.

The job of these `wh*` functions is to correctly convert any arguments and return values from the 32-bit version to the native, 64-bit version. Keep in mind that that it not only includes conversion of integers and pointers, but also content of the structures. Interesting note might be that each of the `wh*` functions has only one argument, which is pointer to an array of 32-bit values. This array contains the parameters passed to the 32-bit system call.

As you could notice, in those 4 service tables there are “system calls” that are not present in the `ntoskrnl.exe`. Also, I mentioned earlier that the `Wow64Transition` is resolved in multiple DLLs. Currently, these DLLs export this symbol:

- `ntdll.dll`
- `win32u.dll`
- `kernel32.dll` and `kernelbase.dll`
- `user32.dll`

The `ntdll.dll` and `win32u.dll` are obvious and they represent the same thing as their native counterparts. The service tables used by `kernel32.dll` and `user32.dll` contain functions for transformation of particular `csrss.exe` calls into their 64-bit version.

It’s also worth noting that at the end of the `ntdll.dll` system table, there are several functions with `NtWow64*` calls, such as `NtWow64ReadVirtualMemory64`, `NtWow64WriteVirtualMemory64` and others. These are special functions which are provided only to WoW64 processes.

One of these special functions is also `NtWow64CallFunction64`. It has it's own small dispatch table and callers can select which function should be called based on its index:

```
0000001800375A0 Wow64FunctionDispatch64 dq offset Wow64CallFunction64Nop
0000001800375A0                                         ; DATA XREF: whNtWow64CallFunction64+1A↑o
0000001800375A8                 dq offset Wow64CallFunctionQueryProcessDebugInfo
0000001800375B0                 dq offset Wow64CallFunctionTurboThunkControl
0000001800375B8                 dq offset Wow64CallFunctionCfgDispatchControl
0000001800375C0                 dq offset Wow64CallFunctionOptimizeChpeImportThunks
0000001800375C8                 dq 0
```

Wow64FunctionDispatch64 (x64)

**NOTE:** I'll be talking about one of these functions - namely `Wow64CallFunctionTurboThunkControl` - later in the Disabling Turbo thunks section.

#### wow64!Wow64SystemServiceEx

This function is similar to the kernel's `nt!KiSystemCall64` - it does the dispatching of the system call. This function is exported by the `wow64.dll` and imported by the emulation DLLs. `Wow64SystemServiceEx` accepts 2 arguments:

- The system call number
- Pointer to an array of 32-bit arguments passed to the system call (mentioned earlier)

The system call number isn't just an index, but also contains index of a system table which needs to be selected (this is also true for `ntoskrnl.exe`):

```
typedef struct _WOW64_SYSTEM_SERVICE
{
USHORT SystemCallNumber : 12;
USHORT ServiceTableIndex : 4;
} WOW64_SYSTEM_SERVICE, *PWOW64_SYSTEM_SERVICE;
```

view raw 2_WOW64_SYSTEM_SERVICE_1.h hosted with ❤ by GitHub

This function then selects `ServiceTables[ServiceTableIndex]` and calls the appropriate `wh*` function based on the `SystemCallNumber`.

```
52  whService = (__int64 (__fastcall *)(WOW64_LOG_ARGUMENTS))ServiceTables[ServiceTable].Base[ServiceNumber];
53  LogService.ServiceTable = (SystemService >> 12) & 3;
54  LogService.ServiceNumber = SystemService & 0xFFF;
55  Teb->LastErrorValue = Teb32->LastErrorValue;
56  if ( pfnWow64LogSystemService )
57  {
58    LogService.Arguments = WowArguments;
59    LogService.PostCall = 0;
60    Wow64LogSystemServiceWrapper(&LogService);
61
62    ServiceStatus = whService(WowArguments);
63
64    LogService.PostCall = 1;
65    LogService.Status = ServiceStatus;
66    Wow64LogSystemServiceWrapper(&LogService);
67  }
68  else if ( whService == whNtCallbackReturn )
69  {
70    ServiceStatus = whNtCallbackReturn(WowArguments);
71  }
72  else if ( whService == whNtQueryVirtualMemory )
73  {
74    ServiceStatus = whNtQueryVirtualMemory(WowArguments);
75  }
76  else if ( whService == whNtOpenKeyEx )
77  {
78    ServiceStatus = whNtOpenKeyEx(WowArguments);
79  }
80  else if ( whService == whNtQueryValueKey )
81  {
82    ServiceStatus = whNtQueryValueKey(WowArguments);
83  }
84  else if ( whService == whNtProtectVirtualMemory )
85  {
86    ServiceStatus = whNtProtectVirtualMemory(WowArguments);
87  }
88  else
89  {
90    ServiceStatus = whService(WowArguments);
91  }
92  Teb32->LastErrorValue = Teb->LastErrorValue;
```

Wow64SystemServiceEx (x64)

**NOTE:** In case the `wow64log.dll` has been successfully loaded, the `Wow64SystemServiceEx` function calls `Wow64LogSystemServiceWrapper` (wrapper around

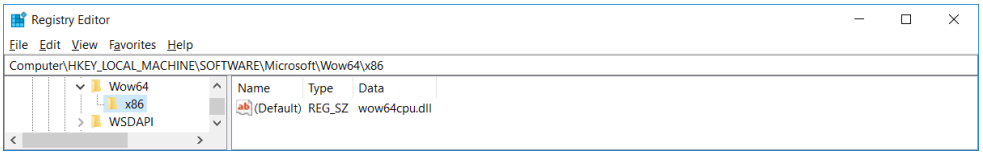
`wow64log!Wow64LogSystemService` function): once before the actual system call and one immediately after. This can be used for instrumentation of each WoW64 system call! The structure passed to `Wow64LogSystemService` contains every important information about the system call - it's table index, system call number, the argument list and on the second call, even the resulting `NTSTATUS`! You can find layout of this structure in the appendix (`WOW64_LOG_SERVICE`).

Finally, as have been mentioned, the `WOW64_SERVICE_TABLE_DESCRIPTOR` structure differs from `KSERVICE_TABLE_DESCRIPTOR` in that it contains `ErrorCase` table. The code mentioned above is actually wrapped in a **SEH** `__try`/`__except` block. If `whService` raise an exception, the `__except` block calls `Wow64HandleSystemServiceError` function. The function looks if the corresponding service table which raised the exception has non-`NULL` `ErrorCase` and if it does, it selects the appropriate `WOW64_ERROR_CASE` for the system call. If the `ErrorCase` is `NULL`, the values from `ErrorCaseDefault` are used. The `NTSTATUS` of the exception is then transformed according to an algorithm which can be found in the appendix.

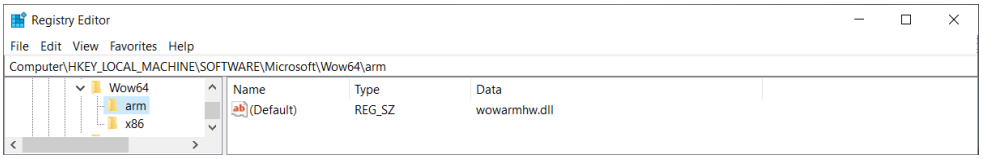
### `wow64!ProcessInit` (cont.)

- `...`
- `wow64!CpuLoadBinaryTranslator // MachineType`
  - `wow64!CpuGetBinaryTranslatorPath // MachineType`
    - `ntdll!NtOpenKey // "\Registry\Machine\Software\Microsoft\Wow64\"`
    - `ntdll!NtQueryValueKey // "arm" / "x86"`
    - `ntdll!RtlGetNtSystemRoot // "arm" / "x86"`
    - `ntdll!RtlUnicodeStringPrintf // "%ws\system32\%ws"`

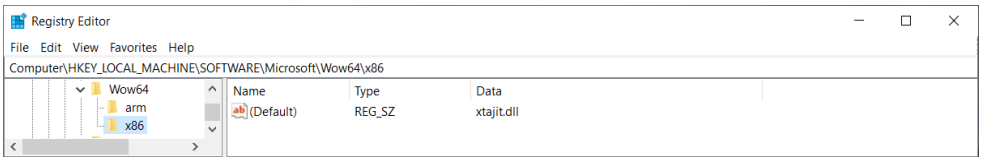
As you've probably guessed, this function constructs path to the **binary-translator DLL**, which is - on x64 - known as `wow64cpu.dll`. This DLL will be responsible for the actual low-level emulation.



\Registry\Machine\Software\Microsoft\Wow64\x86 (x64)



\Registry\Machine\Software\Microsoft\Wow64\arm (ARM64)



\Registry\Machine\Software\Microsoft\Wow64\x86 (ARM64)

We can see that there is no `wow64cpu.dll` on ARM64. Instead, there is `xtajit.dll` used for x86 emulation and `wowarmhw.dll` used for ARM32 emulation.

> **NOTE:** The `CpuGetBinaryTranslatorPath` function is same on both x64 and ARM64 except for one peculiar difference: on Windows x64, if the `\Registry\Machine\Software\Microsoft\Wow64\x86` key cannot be opened (is missing/was deleted), the function **contains a fallback** to load `wow64cpu.dll`. On Windows ARM64, though, it **doesn't have such fallback** and if the registry key is missing, the function fails and the **WoW64 process is terminated**.

`wow64.dll` then loads one of the selected DLL and tries to find there following exported functions:

| | |
|---|---|
| `BTCpuProcessInit` **(!)** | `BTCpuProcessTerm` |
| `BTCpuThreadInit` | `BTCpuThreadTerm` |
| `BTCpuSimulate` **(!)** | `BTCpuResetFloatingPoint` |
| `BTCpuResetToConsistentState` | `BTCpuNotifyDllLoad` |

```
BTCpuNotifyDllUnload                  BTCpuPrepareForDebuggerAttach
BTCpuNotifyBeforeFork                 BTCpuNotifyAfterFork
BTCpuNotifyAffinityChange             BTCpuSuspendLocalThread
BTCpuIsProcessorFeaturePresent        BTCpuGetBopCode (!)
BTCpuGetContext                       BTCpuSetContext
BTCpuTurboThunkControl                BTCpuNotifyMemoryAlloc
BTCpuNotifyMemoryFree                 BTCpuNotifyMemoryProtect
BTCpuFlushInstructionCache2           BTCpuNotifyMapViewOfSection
BTCpuNotifyUnmapViewOfSection         BTCpuUpdateProcessorInformation
BTCpuNotifyReadFile                   BTCpuCfgDispatchControl
BTCpuUseChpeFile                      BTCpuOptimizeChpeImportThunks
BTCpuNotifyProcessExecuteFlagsChange BTCpuProcessDebugEvent
BTCpuFlushInstructionCacheHeavy
```

Interestingly, not all functions need to be found - only those marked with the "**(!)**", the rest is optional. As a next step, the resolved `BTCpuProcessInit` function is called, which performs binary-translator-specific process initialization.

At the end of the `ProcessInit` function, `wow64!Wow64ProtectMrdata(1)` is called, making `.mrdata` non-writable again.

#### `wow64!ThreadInit`

- `wow64!ThreadInit`
  - `wow64!CpuThreadInit`
    - `NtCurrentTeb32()->WOW32Reserved = BTCpuGetBopCode()`
    - `emu!BTCpuThreadInit`

`ThreadInit` does some little thread-specific initialization, such as:

- Copying `CurrentLocale` and `IdealProcessor` values from 64-bit `TEB` into 32-bit `TEB`.
- For non-`WOW64_CPUFLAGS_SOFTWARE` emulators, it calls `CpuThreadInit`, which:
  - Performs `NtCurrentTeb32()->WOW32Reserved = BTCpuGetBopCode()`.
  - Calls `emu!BTCpuThreadInit()`.
- For `WOW64_CPUFLAGS_SOFTWARE` emulators, it creates an event, which added into `AlertByThreadIdEventHashTable` and set to `NtCurrentTeb()->TlsSlots[18]`. This event is used for special emulation of `NtAlertThreadByThreadId` and `NtWaitForAlertByThreadId`.

  **NOTE:** The `WOW64_CPUFLAGS_MSFT64 (1)` or the `WOW64_CPUFLAGS_SOFTWARE (2)` flag is stored in the `NtCurrentTeb()->TlsSlots[/* 10 */ WOW64_TLS_WOW64INFO]`, in the `WOW64INFO.CpuFlags` field. One of these flags is always set in the emulator's `BTCpuProcessInit` function (mentioned in the section above):

  - `wow64cpu.dll` sets `WOW64_CPUFLAGS_MSFT64 (1)`
  - `wowarmhw.dll` sets `WOW64_CPUFLAGS_MSFT64 (1)`
  - `xtajit.dll` sets `WOW64_CPUFLAGS_SOFTWARE (2)`

## x86 on x64

### Entering 32-bit mode

- `...`
- `wow64!RunCpuSimulation`
  - `wow64cpu!BTCpuSimulate`
    - `wow64cpu!RunSimulatedCode`

`RunSimulatedCode` runs in a loop and performs transitions into 32-bit mode either via:

- `jmp fword ptr[reg]` - a "far jump" that not only changes instruction pointer (`RIP`), but also the code segment register (`CS`). This segment usually being set to `0x23`, while 64-bit code segment is `0x33`
- synthetic "machine frame" and `iret` - called on every "state reset"

  **NOTE:** Explanation of segmentation and "why does it work just by changing a segment register" is beyond scope of this article. If you'd like to know more about "long mode" and segmentation, you can start here.

Far jump is used most of the time for the transition, mainly because it's faster. `iret` on the other hand is more powerful, as it can change `CS`, `SS`, `EFLAGS`, `RSP` and `RIP` all at once. The "state reset" occurs when `WOW64_CPURESERVED.Flags` has `WOW64_CPURESERVED_FLAG_RESET_STATE (1)` bit set. This happens during exception (see `wow64!Wow64PrepareForException` and `wow64cpu!BTCpuResetToConsistentState`). Also, this flag is cleared on every emulation loop (using `btr` - bit-test-and-reset).

```
.text:000000006B10162B                push    rbp
.text:000000006B10162C                sub     rsp, 68h
.text:000000006B101630                mov     r12, gs:30h
.text:000000006B101639                lea     r15, TurboThunkDispatch
.text:000000006B101640                mov     r13, [r12+(_TEB.TlsSlots+8)]
.text:000000006B101648                add     r13, 80h
.text:000000006B10164F
.text:000000006B10164F TagReturnFromSystemService:            ; CODE XREF: RunSimulatedCode+167↓j
.text:000000006B10164F                btr     dword ptr [r13+(WOW64_CPURESERVED.Flags-80h)], 0 ; WOW64_CPURESERVED_FLAG_RESET_STATE
.text:000000006B101655                jb      short TagDispatchUsingIret ; If set, use iret
.text:000000006B101657                mov     edi, [r13+(WOW64_CPURESERVED.Context._Edi-80h)] ; Restore non-volatile registers
.text:000000006B10165B                mov     esi, [r13+(WOW64_CPURESERVED.Context._Esi-80h)]
.text:000000006B10165F                mov     ebx, [r13+(WOW64_CPURESERVED.Context._Ebx-80h)]
.text:000000006B101663                mov     ebp, [r13+(WOW64_CPURESERVED.Context._Ebp-80h)]
.text:000000006B101667                mov     eax, [r13+(WOW64_CPURESERVED.Context._Eax-80h)] ; Restore eax
.text:000000006B10166B                mov     r14, rsp
.text:000000006B10166E                mov     dword ptr [rsp+0A8h+MachineFrame._Rip+4], 23h ; SegCs = 0x23
.text:000000006B101676                mov     r8d, 2Bh
.text:000000006B10167C                mov     ss, r8d         ; SegSs = 0x2B
.text:000000006B10167F                mov     r9d, [r13+(WOW64_CPURESERVED.Context._Eip-80h)]
.text:000000006B101683                mov     dword ptr [rsp+0A8h+MachineFrame._Rip], r9d
.text:000000006B101687                mov     esp, [r13+(WOW64_CPURESERVED.Context._Esp-80h)]
.text:000000006B10168B                jmp     fword ptr [r14] ; Jump to 32-bit mode
.text:000000006B10168E ; ---------------------------------------------------------------------------
```

Start of the RunSimulatedCode (x64)

You can see the simplest form of switching into the 32-bit mode. Also, at the beginning you can see that `TurboThunkDispatch` address is moved into the `r15` register. This register stays untouched during the whole `RunSimulatedCode` function.

### Leaving 32-bit mode

The switch back to the 64-bit mode is very similar - it also uses far jumps. The usual situation when code wants to switch back to the 64-bit mode is upon system call:

```
text:4B2F0420 ; __stdcall ZwMapViewOfSection(x, x, x, x, x, x, x, x, x, x)
text:4B2F0420                 public _ZwMapViewOfSection@40
text:4B2F0420 _ZwMapViewOfSection@40 proc near        ; CODE XREF: LdrpMapResourceFile(x,x,x,x,x,x)+135↑p
text:4B2F0420                                         ; LdrpMinimalMapModule(x,x)+8E↑p ...
text:4B2F0420                 mov     eax, 28h        ; NtMapViewOfSection
text:4B2F0425                 mov     edx, offset _Wow64SystemServiceCall@0 ; Wow64SystemServiceCall()
text:4B2F042A                 call    edx ; Wow64SystemServiceCall() ; Wow64SystemServiceCall()
text:4B2F042C                 retn    28h
text:4B2F042C _ZwMapViewOfSection@40 endp
```

NtMapViewOfSection (x64)

The `Wow64SystemServiceCall` is just a simple jump to the `Wow64Transition`:

```
text:4B305060
text:4B305060 ; _DWORD __stdcall Wow64SystemServiceCall()
text:4B305060 _Wow64SystemServiceCall@0 proc near     ; CODE XREF: NtAccessCheck(x,x,x,x,x,x,x,x)+A↑p
text:4B305060                                         ; NtWorkerFactoryWorkerReady(x)+A↑p ...
text:4B305060                 jmp     ds:_Wow64Transition
text:4B305060 _Wow64SystemServiceCall@0 endp
```

Wow64SystemServiceCall (x64)

If you remember, the `Wow64Transition` value is resolved by the `wow64cpu!BTCpuGetBopCode` function:

```
.text:000000006B1011B0
.text:000000006B1011B0                public BTCpuGetBopCode
.text:000000006B1011B0 BTCpuGetBopCode proc near              ; DATA XREF: .rdata:000000006B1031DF↓o
.text:000000006B1011B0                                        ; .rdata:off_6B103738↓o
.text:000000006B1011B0                cmp     cs:CpupSystemCallFast, 0
.text:000000006B1011B7                jz      short TagNotSystemCallFast
.text:000000006B1011B9                lea     rax, KiFastSystemCall
.text:000000006B1011C0                retn
.text:000000006B1011C1 ; ---------------------------------------------------------------------------
.text:000000006B1011C1
.text:000000006B1011C1 TagNotSystemCallFast:                  ; CODE XREF: BTCpuGetBopCode+7↑j
.text:000000006B1011C1                lea     rax, KiFastSystemCall2
.text:000000006B1011C8                retn
.text:000000006B1011C8 BTCpuGetBopCode endp
.text:000000006B1011C8
```

BTCpuGetBopCode - wow64cpu.dll (x64)

It selects either `KiFastSystemCall` or `KiFastSystemCall2` based on the `CpupSystemCallFast` value.

The `KiFastSystemCall` looks like this (used when `CpupSystemCallFast != 0`):

- `[x86] jmp 33h:$+9` (jumps to the instruction below)

- `[x64] jmp qword ptr [r15+offset]` (which points to `CpupReturnFromSimulatedCode`)

The `KiFastSystemCall2` looks like this (used when `CpupSystemCallFast == 0`):

- `[x86] push 0x33`
- `[x86] push eax`
- `[x86] call $+5`
- `[x86] pop eax`
- `[x86] add eax, 12`
- `[x86] xchg eax, dword ptr [esp]`
- `[x86] jmp fword ptr [esp]` (jumps to the instruction below)
- `[x64] add rsp, 8`
- `[x64] jmp wow64cpu!CpupReturnFromSimulatedCode`

Clearly, the `KiFastSystemCall` is faster, so why it's not used used every time?

It turns out, `CpupSystemCallFast` is set to 1 in the `wow64cpu!BTCpuProcessInit` function if the process is not executed with the `ProhibitDynamicCode` mitigation policy and if `NtProtectVirtualMemory(&KiFastSystemCall, PAGE_READ_EXECUTE)` succeeds.

This is because `KiFastSystemCall` is in a non-executable read-only section (`W64SVC`) while `KiFastSystemCall2` is in read-executable section (`WOW64SVC`).

But the actual reason why is `KiFastSystemCall` in non-executable section by default and needs to be set as executable manually is, honestly, unknown to me. My guess would be that it has something to do with relocations, because the address in the `jmp 33h:$+9` instruction must be somehow resolved by the loader. But maybe I'm wrong. Let me know if you know the answer!

### Turbo thunks

I hope you didn't forget about the `TurboThunkDispatch` address hanging in the `r15` register. This value is used as a jump-table:

```
.rdata:000000006B103600 TurboThunkDispatch dq offset TurboDispatchJumpAddressEnd ; DATA XREF: ...
.rdata:000000006B103600                                         ; Index = 0
.rdata:000000006B103608 off_6B103608    dq offset Thunk0Arg     ; DATA XREF: ...
.rdata:000000006B103610 off_6B103610    dq offset Thunk0ArgReloadState ; DATA XREF: ...
.rdata:000000006B103618 off_6B103618    dq offset Thunk1ArgSp   ; DATA XREF: ...
.rdata:000000006B103620 off_6B103620    dq offset Thunk1ArgNSp  ; DATA XREF: ...
.rdata:000000006B103628 off_6B103628    dq offset Thunk2ArgNSpNSp ; DATA XREF: ...
.rdata:000000006B103630 off_6B103630    dq offset Thunk2ArgNSpNSpReloadState ; DATA XREF: ...
.rdata:000000006B103638 off_6B103638    dq offset Thunk2ArgSpNSp ; DATA XREF: ...
.rdata:000000006B103640 off_6B103640    dq offset Thunk2ArgSpSp ; DATA XREF: ...
.rdata:000000006B103648 off_6B103648    dq offset Thunk2ArgNSpSp ; DATA XREF: ...
.rdata:000000006B103650 off_6B103650    dq offset Thunk3ArgNSpNSpNSp ; DATA XREF: ...
.rdata:000000006B103658 off_6B103658    dq offset Thunk3ArgSpSpSp ; DATA XREF: ...
.rdata:000000006B103660 off_6B103660    dq offset Thunk3ArgSpNSpNSp ; DATA XREF: ...
.rdata:000000006B103668 off_6B103668    dq offset Thunk3ArgSpNSpNSpReloadState ; DATA XREF: ...
.rdata:000000006B103670 off_6B103670    dq offset Thunk3ArgSpSpNSp ; DATA XREF: ...
.rdata:000000006B103678 off_6B103678    dq offset Thunk3ArgNSpSpNSp ; DATA XREF: ...
.rdata:000000006B103680 off_6B103680    dq offset Thunk3ArgSpNSpSp ; DATA XREF: ...
.rdata:000000006B103688 off_6B103688    dq offset Thunk4ArgNSpNSpNSpNSp ; DATA XREF: ...
.rdata:000000006B103690 off_6B103690    dq offset Thunk4ArgSpSpNSpNSp ; DATA XREF: ...
.rdata:000000006B103698 off_6B103698    dq offset Thunk4ArgSpSpNSpNSpReloadState ; DATA XREF: ...
.rdata:000000006B1036A0 off_6B1036A0    dq offset Thunk4ArgSpNSpNSpNSp ; DATA XREF: ...
.rdata:000000006B1036A8 off_6B1036A8    dq offset Thunk4ArgSpNSpNSpNSpReloadState ; DATA XREF: ...
.rdata:000000006B1036B0 off_6B1036B0    dq offset Thunk4ArgNSpSpNSpNSp ; DATA XREF: ...
.rdata:000000006B1036B8 off_6B1036B8    dq offset Thunk4ArgSpSpSpNSp ; DATA XREF: ...
.rdata:000000006B1036C0 off_6B1036C0    dq offset QuerySystemTime ; DATA XREF: ...
.rdata:000000006B1036C8 off_6B1036C8    dq offset GetCurrentProcessorNumber ; DATA XREF: ...
.rdata:000000006B1036D0 off_6B1036D0    dq offset ReadWriteFile ; DATA XREF: ...
.rdata:000000006B1036D8 off_6B1036D8    dq offset DeviceIoctlFile ; DATA XREF: ...
.rdata:000000006B1036E0 off_6B1036E0    dq offset RemoveIoCompletion ; DATA XREF: ...
.rdata:000000006B1036E8 off_6B1036E8    dq offset WaitForMultipleObjects ; DATA XREF: ...
.rdata:000000006B1036F0 off_6B1036F0    dq offset WaitForMultipleObjects32 ; DATA XREF: ...
.rdata:000000006B1036F8 off_6B1036F8    dq offset CpupReturnFromSimulatedCode ; DATA XREF: ...
.rdata:000000006B103700                 dq offset ThunkNone     ; Index: 32
.rdata:000000006B103708                 align 10h
.rdata:000000006B103710 ;
```

TurboThunkDispatch (x64)

There are 32 items in the jump-table.

```
.text:000000006B101742
.text:000000006B101742 CpupReturnFromSimulatedCode:           ; CODE XREF: KiFastSystemCall2+18↓j
.text:000000006B101742                                        ; DATA XREF: BTCpuResetToConsistentState+8A↓o ...
.text:000000006B101742                 xchg    rsp, r14
.text:000000006B101745                 mov     r8d, [r14]
.text:000000006B101748                 add     r14, 4
.text:000000006B10174C                 mov     [r13+(_WOW64_CONTEXT._Eip-7Ch)], r8d
.text:000000006B101750                 mov     [r13+(_WOW64_CONTEXT._Esp-7Ch)], r14d
.text:000000006B101754                 lea     r11, [r14+4]
.text:000000006B101758                 mov     [r13+(_WOW64_CONTEXT._Edi-7Ch)], edi
.text:000000006B10175C                 mov     [r13+(_WOW64_CONTEXT._Esi-7Ch)], esi
.text:000000006B101760                 mov     [r13+(_WOW64_CONTEXT._Ebx-7Ch)], ebx
.text:000000006B101764                 mov     [r13+(_WOW64_CONTEXT._Ebp-7Ch)], ebp
.text:000000006B101768                 pushfq
.text:000000006B101769                 pop     r8
.text:000000006B10176B                 mov     [r13+(_WOW64_CONTEXT.EFlags-7Ch)], r8d
.text:000000006B10176F ; Exported entry   9. TurboDispatchJumpAddressStart
.text:000000006B10176F
.text:000000006B10176F                 public TurboDispatchJumpAddressStart
.text:000000006B10176F TurboDispatchJumpAddressStart:         ; DATA XREF: .rdata:off_6B103738↓o
.text:000000006B10176F                 mov     ecx, eax
.text:000000006B101771                 shr     ecx, 16
.text:000000006B101774                 jmp     qword ptr [r15+rcx*8]
.text:000000006B101778 ; ---------------------------------------------------------------------------
.text:000000006B101778 ; Exported entry   8. TurboDispatchJumpAddressEnd
.text:000000006B101778
.text:000000006B101778                 public TurboDispatchJumpAddressEnd
.text:000000006B101778 TurboDispatchJumpAddressEnd:           ; CODE XREF: RunSimulatedCode+26B↓j
.text:000000006B101778                                        ; RunSimulatedCode+31B↓j
.text:000000006B101778                                        ; DATA XREF: ...
.text:000000006B101778                 mov     ecx, eax
.text:000000006B10177A                 mov     rdx, r11
.text:000000006B10177D                 call    cs:__imp_Wow64SystemServiceEx
.text:000000006B101783                 mov     [r13+_CPU_RESERVED_EX._Eax], eax
.text:000000006B101787                 jmp     TagReturnFromSystemService ; WOW64_CPURESERVED_FLAG_RESET_STATE
```

TurboDispatchJumpAddressStart (x64)

`CpupReturnFromSimulatedCode` is the first code that is always executed in the 64-bit mode when 32-bit to 64-bit transition occurs. Let's recapitulate the code:

- Stack is swapped,
- Non-volatile registers are saved
- `eax` - which contains the encoded service table index and system call number - is moved into the `ecx`
- it's high-word is acquired via `ecx >> 16`.
- the result is used as an index into the `TurboThunkDispatch` jump-table

You might be confused now, because few sections above we've defined the service number like this:

```
typedef struct _WOW64_SYSTEM_SERVICE
{
USHORT SystemCallNumber : 12;
USHORT ServiceTableIndex : 4;
} WOW64_SYSTEM_SERVICE, *PWOW64_SYSTEM_SERVICE;
```

view raw 2_WOW64_SYSTEM_SERVICE_1.h hosted with ❤ by GitHub

…therefore, after right-shifting this value by 16 bits we should get always 0, right?

It turns out, on x64, the `WOW64_SYSTEM_SERVICE` might be defined like this:

```
typedef struct _WOW64_SYSTEM_SERVICE
{
ULONG SystemCallNumber : 12;
ULONG ServiceTableIndex : 4;
ULONG TurboThunkNumber : 5; // Can hold values 0 - 31
ULONG AlwaysZero : 11;
} WOW64_SYSTEM_SERVICE, *PWOW64_SYSTEM_SERVICE;
```

view raw 2_WOW64_SYSTEM_SERVICE_2.h hosted with ❤ by GitHub

Let's examine few WoW64 system calls:

```
text:4B2F0420 ; __stdcall ZwMapViewOfSection(x, x, x, x, x, x, x, x, x, x)
text:4B2F0420                 public _ZwMapViewOfSection@40
text:4B2F0420 _ZwMapViewOfSection@40 proc near        ; CODE XREF: LdrpMapResourceFile(x,x,x,x,x,x)+135↑p
text:4B2F0420                                         ; LdrpMinimalMapModule(x,x)+8E↑p ...
text:4B2F0420                 mov     eax, 28h        ; NtMapViewOfSection
text:4B2F0425                 mov     edx, offset _Wow64SystemServiceCall@0 ; Wow64SystemServiceCall()
text:4B2F042A                 call    edx ; Wow64SystemServiceCall() ; Wow64SystemServiceCall()
text:4B2F042C                 retn    28h
text:4B2F042C _ZwMapViewOfSection@40 endp
```

NtMapViewOfSection (x64)

```
text:4B2F01C0
text:4B2F01C0 ; __stdcall ZwWaitForSingleObject(x, x, x)
text:4B2F01C0                 public _ZwWaitForSingleObject@12
text:4B2F01C0 _ZwWaitForSingleObject@12 proc near     ; CODE XREF: LdrpDrainWorkQueue(x)+14A↑p
text:4B2F01C0                                         ; EtwpLogger(x)+5A↑p ...
text:4B2F01C0                 mov     eax, 0D0004h    ; NtWaitForSingleObject
text:4B2F01C5                 mov     edx, offset _Wow64SystemServiceCall@0 ; Wow64SystemServiceCall()
text:4B2F01CA                 call    edx ; Wow64SystemServiceCall() ; Wow64SystemServiceCall()
text:4B2F01CC                 retn    0Ch
text:4B2F01CC _ZwWaitForSingleObject@12 endp
text:4B2F01CC
```

NtWaitForSingleObject (x64)

```
text:4B2F01F0
text:4B2F01F0 ; __stdcall ZwDeviceIoControlFile(x, x, x, x, x, x, x, x, x, x)
text:4B2F01F0                 public _ZwDeviceIoControlFile@40
text:4B2F01F0 _ZwDeviceIoControlFile@40 proc near     ; CODE XREF: GetProcessIptTrace(x,x,x)+9B↓p
text:4B2F01F0                                         ; GetProcessIptTraceSize(x,x)+95↓p ...
text:4B2F01F0                 mov     eax, 1B0007h    ; NtDeviceIoControlFile
text:4B2F01F5                 mov     edx, offset _Wow64SystemServiceCall@0 ; Wow64SystemServiceCall()
text:4B2F01FA                 call    edx ; Wow64SystemServiceCall() ; Wow64SystemServiceCall()
text:4B2F01FC                 retn    28h
text:4B2F01FC _ZwDeviceIoControlFile@40 endp
text:4B2F01FC
```

NtDeviceIoControlFile (x64)

Based on our new definition of `WOW64_SYSTEM_SERVICE`, we can conclude that:

- `NtMapViewOfSection` uses turbo thunk with index 0 (`TurboDispatchJumpAddressEnd`)
- `NtWaitForSingleObject` uses turbo thunk with index 13 (`Thunk3ArgSpNSpNSpReloadState`)
- `NtDeviceIoControlFile` uses turbo thunk with index 27 (`DeviceIoctlFile`)

Let's finally explain "turbo thunks" in proper way.

Turbo thunks are an optimalization of WoW64 subsystem - specifically on Windows x64 - that enables for particular system calls to never leave the `wow64cpu.dll` - the conversion of parameters and return value, and the `syscall` instruction itself is fully performed there. The set of functions that use these turbo thunks reveals, that they are usually very simple in terms of parameter conversion - they receive numerical values or handles.

The notation of `Thunk*` labels is as follows:

- The number specifies how many arguments the function receives
- `Sp` converts parameter with sign-extension
- `NSp` converts parameter without sign-extension
- `ReloadState` will return to the 32-bit mode using `iret` instead of far jump, if `WOW64_CPURESERVED_FLAG_RESET_STATE` is set
- `QuerySystemTime`, `ReadWriteFile`, `DeviceIoctlFile`, ... are special cases

Let's take the `NtWaitForSingleObject` and its turbo thunk `Thunk3ArgSpNSpNSpReloadState` as an example:

- it receives 3 parameters
- 1st parameter is sign-extended
- 2nd parameter isn't sign-extended
- 3rd parameter isn't sign-extended
- it can switch to 32-bit mode using `iret` if `WOW64_CPURESERVED_FLAG_RESET_STATE` is set

When we cross-check this information with its function prototype, it makes sense:

```
NTSTATUS
NTAPI
NtWaitForSingleObject(
_In_ HANDLE Handle,
_In_ BOOLEAN Alertable,
_In_ PLARGE_INTEGER Timeout
);
```

view raw 2_NtWaitForSingleObject.h hosted with ❤ by GitHub

The sign-extension of `HANDLE` makes sense, because if we pass there an `INVALID_HANDLE_VALUE`, which happens to be `0xFFFFFFFF (-1)` on 32-bits, we don't want to convert this value to `0x00000000FFFFFFFF`, but `0xFFFFFFFFFFFFFFFF`.

On the other hand, if the `TurboThunkNumber` is 0, the call will end up in the `TurboDispatchJumpAddressEnd` which in turn calls `wow64!Wow64SystemServiceEx`. You can consider this case as the "slow path".

### Disabling Turbo thunks

On Windows x64, the Turbo thunk optimization can be actually disabled!

In one of the previous sections I've been talking about `ntdll32!NtWow64CallFunction64` and `wow64!Wow64CallFunctionTurboThunkControl` functions. As with any other `NtWow64*` function, `NtWow64CallFunction64` is only available in the WoW64 `ntdll.dll`. This function can be called with an index to WoW64 function in the `Wow64FunctionDispatch64` table (mentioned earlier).

The function prototype might look like this:

```
typedef enum _WOW64_FUNCTION {
Wow64Function64Nop,
Wow64FunctionQueryProcessDebugInfo,
Wow64FunctionTurboThunkControl,
Wow64FunctionCfgDispatchControl,
Wow64FunctionOptimizeChpeImportThunks,
} WOW64_FUNCTION;
NTSYSCALLAPI
NTSTATUS
NTAPI
NtWow64CallFunction64(
_In_ WOW64_FUNCTION Wow64Function,
_In_ ULONG Flags,
_In_ ULONG InputBufferLength,
_In_reads_bytes_opt_(InputBufferLength) PVOID InputBuffer,
_In_ ULONG OutputBufferLength,
_Out_writes_bytes_opt_(OutputBufferLength) PVOID OutputBuffer,
_Out_opt_ PULONG ReturnLength
);
```

view raw 2_NtWow64CallFunction64.h hosted with ❤ by GitHub

> **NOTE:** This function prototype has been reconstructed with the help of the `wow64!Wow64CallFunction64Nop` function code, which just logs the parameters.

We can see that `wow64!Wow64CallFunctionTurboThunkControl` can be called with an index of 2. This function performs some sanity checks and then passes calls `wow64cpu!BTCpuTurboThunkControl(*(ULONG*)InputBuffer)`.

`wow64cpu!BTCpuTurboThunkControl` then checks the input parameter.

- If it's 0, it patches every target of the jump table to point to `TurboDispatchJumpAddressEnd` (remember, this is the target that is called when `WOW64_SYSTEM_SERVICE.TurboThunkNumber` is 0).
- If it's non-0, it returns `STATUS_NOT_SUPPORTED`.

This means 2 things:

- Calling `wow64cpu!BTCpuTurboThunkControl(0)` disables the Turbo thunks, and every system call ends up taking the "slow path".
- It is not possible to enable them back.

With all this in mind, we can achieve disabling Turbo thunks by this call:

```
#define WOW64_TURBO_THUNK_DISABLE 0
#define WOW64_TURBO_THUNK_ENABLE 1 // STATUS_NOT_SUPPORTED :(
ThunkInput = WOW64_TURBO_THUNK_DISABLE;
Status = NtWow64CallFunction64(Wow64FunctionTurboThunkControl,
0,
sizeof(ThunkInput),
&ThunkInput,
0,
NULL,
NULL);
```

view raw 2_NtWow64CallFunction64_Wow64FunctionTurboThunkControl.h hosted with ❤ by GitHub

What it might be good for? I can think of 3 possible use-cases:

- If we deploy custom `wow64log.dll`, disabling Turbo thunks guarantees that we will see **every WoW64 system call** in our `wow64log!Wow64LogSystemService` callback. We wouldn't see such calls if the Turbo thunks were enabled, because they would take the "fast path" inside of the `wow64cpu.dll` where the `syscall` would be executed.
- If we decide to hook `Nt*` functions in the **native `ntdll.dll`**, disabling Turbo thunks guarantees that for each `Nt*` function called in the `ntdll32.dll`, the correspondint `Nt*` function will be called in the native `ntdll.dll`. (This is basically the same point as the previous one.)

  > **NOTE:** Keep in mind that this only applies on system calls, i.e. on `Nt*` or `Zw*` functions. Other functions are not called from the 32-bit `ntdll.dll` to the 64-bit `ntdll.dll`. For example, if we hooked `RtlDecompressBuffer` in the native `ntdll.dll` of the WoW64 process, it wouldn't be called on `ntdll32!RtlDecompressBuffer` call. This is because the full implementaion of the `Rtl*` functions is already in the `ntdll32.dll`.
- We can "harmlessly" patch high-word moved to the `eax` in every WoW64 system call stub to 0. For example we could see in `NtWaitForSingleObject` there is `mov eax, 0D0004h`. If we patched appropriate 2 bytes in that instruction so that the instruction would become `mov eax, 4h`, the system call would still work.

  This approach can be used as an anti-hooking technique - if there's a jump at the start of the function, the patch will break it. If there's not a jump, we just disable the Turbo thunk for this function.

## x86 on ARM64

Emulation of x86 applications on ARM64 is handled by an actual binary translation. Instead of `wow64cpu.dll`, the `xtajit.dll` (probably shortcut for "x86 to ARM64 JIT") is used for its emulation. As with other emulation DLLs, this DLL is native (ARM64).

The x86 emulation on Windows ARM64 consists also of other "XTA" components:

- `xtac.exe` - XTA Compiler
- `XtaCache.exe` - XTA Cache Service

Execution of x86 programs on ARM64 appears to go way behind just emulation. It is also capable of caching already binary-translated code, so that next execution of the same application should be faster. This cache is located in the `Windows\XtaCache` directory which contains files in format `FILENAME.EXT.HASH1.HASH2.mp.N.jc`. These files are then mapped to the user-mode address space of the application. If you're asking whether you can find an actual ARM64 code in these files - indeed, you can.

Unfortunatelly, Microsoft doesn't provide symbols to any of these `xta*` DLLs or executables. But if you're feeling adventurous, you can find some interesting artifacts, like this array of structures inside of the `xtajit.dll`, which contains name of the function and its pointer. There are thousands of items in this array:

BT functions (before) (ARM64)

With a simple Python script, we can mass-rename all functions referenced in this array:

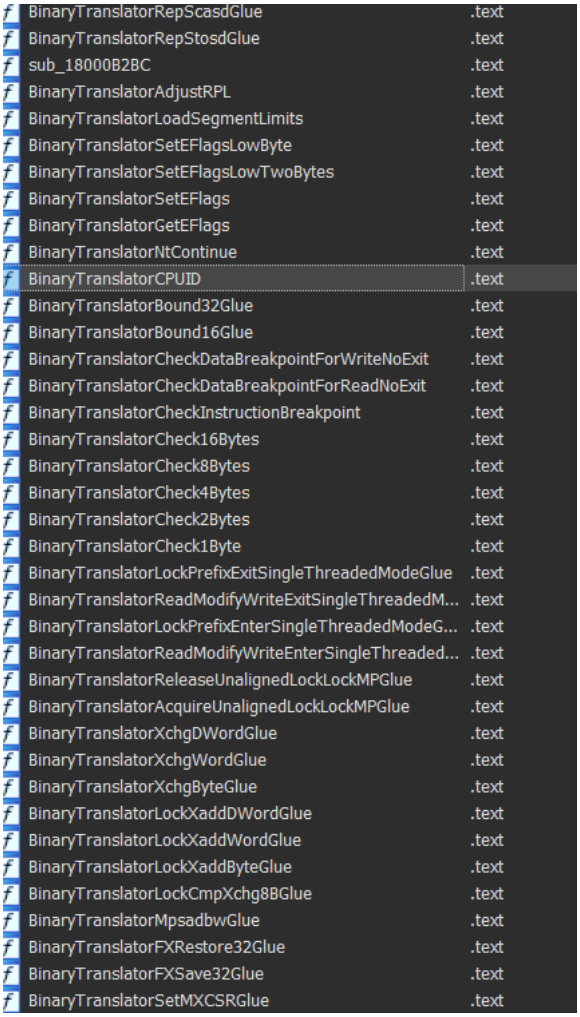
```
begin = 0x01800A8C20
end = 0x01800B7B4F
struct_size = 24
ea = begin
while ea < end:
ea += struct_size
name = idc.GetString(idc.Qword(ea))
idc.MakeName(idc.Qword(ea+8), name)
```

view raw 2_IDA_BT_rename.py hosted with ❤ by GitHub

I'd like to thank *Milan Boháček* for providing me this script.

```
.rdata:00000001800A8C1F                 DCB 0xFF ; ˙
.rdata:00000001800A8C20                 DCQ aBinarytranslat_0   ; "BinaryTranslatorCheckSumBlock"
.rdata:00000001800A8C28                 DCQ loc_18000AB0C
.rdata:00000001800A8C30                 DCB 0x58 ; X
.rdata:00000001800A8C31                 DCB    0
.rdata:00000001800A8C32                 DCB    0
.rdata:00000001800A8C33                 DCB    0
.rdata:00000001800A8C34 byte_1800A8C34  DCB 1                   ; DATA XREF: sub_180045420+2C↑o
.rdata:00000001800A8C34                                         ; sub_180045420+4C↑o ...
.rdata:00000001800A8C35                 ALIGN 8
.rdata:00000001800A8C38                 DCQ aBinarytranslat_1   ; "BinaryTranslatorCheckSumBlockEnd"
.rdata:00000001800A8C40                 DCQ BinaryTranslatorCheck1Byte
.rdata:00000001800A8C48                 DCB 0x60 ; `
.rdata:00000001800A8C49                 DCB    0
.rdata:00000001800A8C4A                 DCB    0
.rdata:00000001800A8C4B                 DCB    0
.rdata:00000001800A8C4C                 DCB    0
.rdata:00000001800A8C4D                 DCB    0
.rdata:00000001800A8C4E                 DCB    0
.rdata:00000001800A8C4F                 DCB    0
.rdata:00000001800A8C50                 DCQ aBinarytranslat_2   ; "BinaryTranslatorCheck1Byte"
.rdata:00000001800A8C58                 DCQ BinaryTranslatorCheck1Byte
.rdata:00000001800A8C60                 DCB 0x68 ; h
.rdata:00000001800A8C61                 DCB    0
.rdata:00000001800A8C62                 DCB    0
.rdata:00000001800A8C63                 DCB    0
.rdata:00000001800A8C64                 DCB    1
.rdata:00000001800A8C65                 DCB    0
.rdata:00000001800A8C66                 DCB    0
.rdata:00000001800A8C67                 DCB    0
.rdata:00000001800A8C68                 DCQ aBinarytranslat_3   ; "BinaryTranslatorCheck1ByteEnd"
.rdata:00000001800A8C70                 DCQ BinaryTranslatorCheck2Bytes
.rdata:00000001800A8C78                 DCB 0x70 ; p
.rdata:00000001800A8C79                 DCB    0
.rdata:00000001800A8C7A                 DCB    0
.rdata:00000001800A8C7B                 DCB    0
.rdata:00000001800A8C7C                 DCB    0
.rdata:00000001800A8C7D                 DCB    0
.rdata:00000001800A8C7E                 DCB    0
.rdata:00000001800A8C7F                 DCB    0
.rdata:00000001800A8C80                 DCQ aBinarytranslat_4   ; "BinaryTranslatorCheck2Bytes"
.rdata:00000001800A8C88                 DCQ BinaryTranslatorCheck2Bytes
.rdata:00000001800A8C90                 DCB 0x78 ; x
.rdata:00000001800A8C91                 DCB    0
.rdata:00000001800A8C92                 DCB    0
.rdata:00000001800A8C93                 DCB    0
```

BT functions (after) (ARM64)

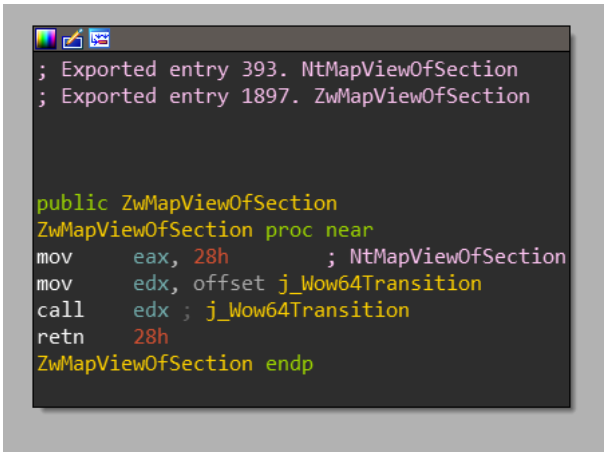


BT translated function list (ARM64)

### `Windows\SyCHPE32` & `Windows\SysWOW64`

One thing you can observe on ARM64 is that it contains two folders used for x86 emulation. The difference between them is that `SyCHPE32` contains small subset of DLLs that are frequently used by applications, while contents of the `SysWOW64` folder is quite identical with the content of this folder on Windows x64.

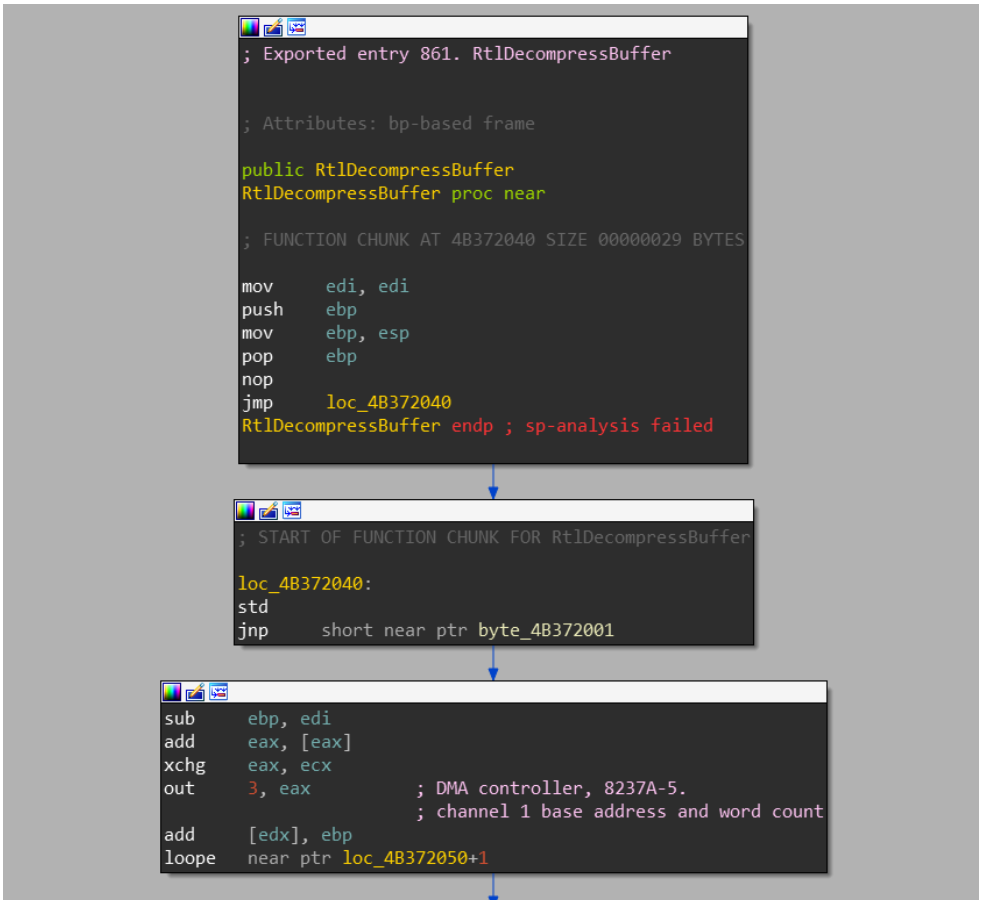
The `CHPE` DLLs are not pure-x86 DLLs and not even pure-ARM64 DLLs. They are “compiled-hybrid-PE”s. What does it mean? Let’s see:



NtMapViewOfSection (CHPE) (ARM64)

After opening `SyCHPE32\ntdll.dll`, IDA will first tell us - unsurprisingly - that it cannot download PDB for this DLL. After looking at randomly chosen `Nt*` function, we can see that it doesn’t differ from what we would see in the

`SysWOW64\ntdll.dll`. Let's look at some non-`Nt*` function:



RtlDecompressBuffer (CHPE) (ARM64)

We can see it contains regular x86 function prologue, immediately followed by x86 function epilogue and then jump somewhere, where it looks like that there's just garbage. That "garbage" is actually ARM64 code of that function.

My guess is that the reason for this prologue is probably compatibility with applications that check whether some particular functions are hooked or not - by checking if the first bytes of the function contain real x86 prologue.

> **NOTE:** Again, if you're feeling adventurous, you can patch `FileHeader.Machine` field in the PE header to `IMAGE_FILE_MACHINE_ARM64 (0xAA64)` and open this file in IDA. You will see a whole lot of correctly resolved ARM64 functions. Again, I'd like to thank to *Milan Boháček* for this tip.

If your question is "how are these images generated?", I would answer that I don't know, but my bet would be on some internal version of Microsoft's C++ compiler toolchain. This idea appears to be supported by various occurences of the `CHPE` keyword in the ChakraCore codebase.

## ARM32 on ARM64

The loop inside of the `wowarmhw!BTCpuSimulate` is fairly simple compared to `wow64cpu.dll` loop:

```
DECLSPEC_NORETURN
VOID
BTCpuSimulate(
VOID
)
{
NTSTATUS Status;
PCONTEXT Context;
//
// Gets WoW64 CONTEXT structure (ARM32) using
// the RtlWow64GetCurrentCpuArea() function.
//
```

```
 Status = CpupGetArmContext(&Context, NULL);
 if (!NT_SUCCESS(Status))
 {
 RtlRaiseStatus(Status);
 //
 // UNREACHABLE
 //
 return;
 }
 for (;;)
 {
 //
 // Switch to ARM32 mode and run the emulation.
 //
 NtCurrentTeb()->TlsSlots[/* 2 */ WOW64_TLS_INCPUSIMULATION] = TRUE;
 CpupSwitchTo32Bit(Context);
 NtCurrentTeb()->TlsSlots[/* 2 */ WOW64_TLS_INCPUSIMULATION] = FALSE;
 //
 // When we get here, it means ARM32 code performed a system call.
 // Advance instruction pointer to skip the "UND 0F8h" instruction.
 //
 Context->Pc += 2;
 //
 // Set LSB (least significat bit) if ARM32 is executing in
 // Thumb mode.
 //
 if (Context->Cpsr & 0x20) {
 Context->Pc |= 1;
 }
 //
 // Let wow64.dll emulate the system call. R12 has the system call
 // number, Sp points to the stack which contains the system call
 // arguments.
 //
 Context->R0 = Wow64SystemServiceEx(Context->R12, Context->Sp);
 }
 }
```

view raw 2_BTCpuSimulate_ARM64.h hosted with ❤ by GitHub

`CpupSwitchTo32Bit` does nothing else than saving the whole `CONTEXT`, performing `SVC 0xFFFF` instruction and then restoring the `CONTEXT`.

### `nt!KiEnter32BitMode` / `SVC 0xFFFF`

I won't be explaining here how system call dispatching works in the `ntoskrnl.exe` - Bruce Dang already did an excellent job doing it. This section is a follow up on his article, though.

`SVC` instruction is sort-of equivalent of `SYSCALL` instruction on ARM64 - it basically enters the kernel mode. But there is a small difference between `SYSCALL` and `SVC`: while on Windows x64 the system call number is moved into the `eax` register, on ARM64 the system call number can be encoded directly into the `SVC` instruction.

```
LDP             W4, W5, [X19,#0x14]
LDP             W6, W7, [X19,#0x1C]
LDP             W8, W9, [X19,#0x24]
LDP             W10, W11, [X19,#0x2C]
LDP             W12, W13, [X19,#0x34]
LDP             W14, W15, [X19,#0x3C]
LDP             W16, W17, [X19,#0x44]
LDP             Q0, Q1, [X19,#0x50]
LDP             Q2, Q3, [X19,#0x70]
LDP             Q4, Q5, [X19,#0x90]
LDP             Q6, Q7, [X19,#0xB0]
LDP             Q8, Q9, [X19,#0xD0]
LDP             Q10, Q11, [X19,#0xF0]
LDP             Q12, Q13, [X19,#0x110]
LDP             Q14, Q15, [X19,#0x130]
SVC             0xFFFF
LDR             X19, [SP,#0x80+CapturedContext] ; X19 = _CONTEXT*
STP   Supervisor Call
STP
STP   Causes an exception to be taken to EL1.
STP   On executing an SVC instruction, the PE records the exception as a
STP   Supervisor Call exception in ESR_ELx, using the EC value 0x15, and
STP   the value of the immediate argument.
STP
STP   SVC #<imm>
STP             W16, W17, [X19,#0x44]
STP             Q0, Q1, [X19,#0x50]
STP             Q2, Q3, [X19,#0x70]
STP             Q4, Q5, [X19,#0x90]
STP             Q6, Q7, [X19,#0xB0]
STP             Q8, Q9, [X19,#0xD0]
STP             Q10, Q11, [X19,#0xF0]
```

SVC 0xFFFF (ARM64)

Let's peek for a moment into the kernel to see how is this `SVC` instruction handled:

- `nt!KiUserExceptionHandler`
  - `nt!KiEnter32BitMode`

```
.text:0000000140003380 ; ---------------------------------------------------------------------------
.text:0000000140003384                 ALIGN 0x80
.text:0000000140003400 ; START OF FUNCTION CHUNK FOR FrontendsmcKiUserExceptionHandler
.text:0000000140003400
.text:0000000140003400 KiUserExceptionHandler                  ; CODE XREF: FrontendsmcKiUserExceptionHandler+34↓j
.text:0000000140003400                                         ; FrontendhvcKiUserExceptionHandler+34↓j
.text:0000000140003400                 SUB             SP, SP, #0x370
.text:0000000140003404                 STP             X18, X30, [SP,#0x370+var_240]
.text:0000000140003408                 MRS             X18, ESR_EL1
.text:000000014000340C                 LSR             X30, X18, #0x1A ; Extract the EC bits
.text:0000000140003410                 CMP             X30, #0x15 ; Check for SVC exception
.text:0000000140003414                 B.NE            TagNonSvcException
.text:0000000140003418                 SXTH            X18, W18 ; if SVC == 0xFFFF
.text:000000014000341C                 CMN             X18, #1 ; WORD(-1)
.text:0000000140003420                 B.EQ            KiEnter32BitMode
.text:0000000140003424                 B               KiSystemServiceException
.text:0000000140003428 ; ---------------------------------------------------------------------------
.text:0000000140003428
.text:0000000140003428 TagNonSvcException                      ; CODE XREF: FrontendsmcKiUserExceptionHandler-7EC↑j
.text:0000000140003428                                         ; KiEnter32BitMode+8↓j
.text:0000000140003428                 MRS             X18, TPIDR_EL1
```

KiUserExceptionHandler (ARM64)

```
.text:0000000140004CE4
.text:0000000140004CE4
.text:0000000140004CE4 KiEnter32BitMode                    ; CODE XREF: FrontendsmcKiUserExceptionHandler-7E0↑j
.text:0000000140004CE4
.text:0000000140004CE4 arg_130         =  0x130
.text:0000000140004CE4
.text:0000000140004CE4                 ADRP            X19, #KiIs32BitEl0Supported@PAGE
.text:0000000140004CE8                 LDRB            W19, [X19,#KiIs32BitEl0Supported@PAGEOFF]
.text:0000000140004CEC                 CBZ             W19, TagNonSvcException
.text:0000000140004CF0                 LDR             X18, [SP,#arg_130]
.text:0000000140004CF4                 ADD             SP, SP, #0x370
.text:0000000140004CF8                 MRS             X30, ELR_EL1
.text:0000000140004CFC                 AND             W19, W19, #0xFFFFFFFE
.text:0000000140004D00                 MSR             ELR_EL1, X15
.text:0000000140004D04                 MOV             X19, #0xFC20
.text:0000000140004D08                 MOVK            X19, #0xFE00,LSL#16
.text:0000000140004D0C                 MRS             X20, SPSR_EL1
.text:0000000140004D10                 AND             X16, X16, X19
.text:0000000140004D14                 AND             X20, X20, #0b111000000
.text:0000000140004D18                 ORR             X16, X16, #0b10000
.text:0000000140004D1C                 ORR             X16, X16, X20
.text:0000000140004D20                 MSR             SPSR_EL1, X16
.text:0000000140004D24                 MOV             X19, #0x7C00000
.text:0000000140004D28                 MOV             X20, #0x9F
.text:0000000140004D2C                 MOVK            X20, #0xF800,LSL#16
.text:0000000140004D30                 AND             X19, X17, X19
.text:0000000140004D34                 AND             X20, X17, X20
.text:0000000140004D38                 MSR             FPCR, X19
.text:0000000140004D3C                 MSR             FPSR, X20
.text:0000000140004D40                 ERET
.text:0000000140004D40 ; End of function KiEnter32BitMode
.text:0000000140004D40
```

KiEnter32BitMode (ARM64)

We can see that:

- `MRS X30, ELR_EL1` - current interrupt-return address (stored in `ELR_EL1` system register) will be moved to the register `X30` (link register - `LR`).
- `MSR ELR_EL1, X15` - the interrupt-return address will be replaced by value in the register `X15` (**which is aliased to the instruction pointer register** - `PC` - in the 32-bit mode).
- `ORR X16, X16, #0b10000` - bit [4] is being set in `X16` which is later moved to the `SPSR_EL1` register. Setting this bit **switches the execution mode to 32-bits**.

Simply said, in the `X15` register, there is an address that will be executed once we leave the kernel-mode and enter the user-mode - which happens with the `ERET` instruction at the end.

### `nt!KiExit32BitMode` / `UND #0xF8`

Alright, we're in the 32-bit ARM mode now, how exactly do we leave? Windows solves this transition via `UND` instruction - which is similar to the `UD2` instruction on the **Intel** CPUs. If you're not familiar with it, you just need to know that it is instruction that basically guarantees that it'll throw "invalid instruction" exception which can OS kernel handle. It is defined-"undefined instruction". Again there is the same difference between the `UND` and `UD2` instruction in that the ARM can have any 1-byte immediate value encoded directly in the instruction.

Let's look at the `NtMapViewOfSection` system call in the `SysArm32\ntdll.dll`:

```
.text:4B2A26F0 ; =============== S U B R O U T I N E =======================================
.text:4B2A26F0
.text:4B2A26F0 ; Attributes: noreturn
.text:4B2A26F0
.text:4B2A26F0                 EXPORT ZwMapViewOfSection
.text:4B2A26F0 ZwMapViewOfSection                      ; CODE XREF: LdrpMinimalMapModule+7E↓p
.text:4B2A26F0                                         ; LdrpFindLoadedDllByMappingFile+8A↓p ...
.text:4B2A26F0                 PUSH            {R0-R3}
.text:4B2A26F2                 MOV.W           R12, #0x28
.text:4B2A26F6                 UND             #0xF8   ; nt!KiExit32BitMode
.text:4B2A26F8 ; ---------------------------------------------------------------------------
.text:4B2A26F8                 ADD             SP, SP, #0x10
.text:4B2A26FA                 BX              LR
.text:4B2A26FA ; End of function ZwMapViewOfSection
.text:4B2A26FA
```

NtMapViewOfSection (ARM64)

Let's peek into the kernel again:

- `nt!KiUser32ExceptionHandler`
  - `nt!KiFetchOpcodeAndEmulate`
    - `nt!KiExit32BitMode`

```
.text:0000000140004E00 ; ---------------------------------------------------------------------------
.text:0000000140004E00
.text:0000000140004E00 loc_140004E00                           ; CODE XREF: KiFetchOpcodeAndEmulate+2C↑j
.text:0000000140004E00                 MOV             W19, #0xDEF8 ; UND #0xF8
.text:0000000140004E04                 CMP             W17, W19
.text:0000000140004E08                 B.EQ            KiExit32BitMode
.text:0000000140004E0C                 UBFX            W19, W17, #0xC, #4
.text:0000000140004E10                 CMP             W19, #0xC
.text:0000000140004E14                 B.EQ            KiEmulateLoadStoreMultiple16
.text:0000000140004E18                 MOV             X30, X24
.text:0000000140004E1C                 B               loc_140003620
.text:0000000140004E20 ; ---------------------------------------------------------------------------
.text:0000000140004E20
```

KiEnter32BitMode (ARM64)

```
.text:0000000140004D44
.text:0000000140004D44 ; Attributes: noreturn
.text:0000000140004D44
.text:0000000140004D44 KiExit32BitMode                         ; CODE XREF: KiFetchOpcodeAndEmulate+50↓j
.text:0000000140004D44                 MRS             X15, ELR_EL1
.text:0000000140004D48                 MRS             X16, SPSR_EL1
.text:0000000140004D4C                 AND             X19, X16, #0xFFFFFFFFFFFFFFC0
.text:0000000140004D50                 MSR             SPSR_EL1, X19
.text:0000000140004D54                 MRS             X17, FPCR
.text:0000000140004D58                 MRS             X20, FPSR
.text:0000000140004D5C                 ORR             X17, X17, X20
.text:0000000140004D60                 MSR             ELR_EL1, X24
.text:0000000140004D64                 MOV             X30, X24
.text:0000000140004D68                 MOV             X19, #0
.text:0000000140004D6C                 MOV             X20, #0
.text:0000000140004D70                 MOV             X21, #0
.text:0000000140004D74                 MOV             X22, #0
.text:0000000140004D78                 MOV             X23, #0
.text:0000000140004D7C                 MOV             X24, #0
.text:0000000140004D80                 MOV             X25, #0
.text:0000000140004D84                 MOV             X26, #0
.text:0000000140004D88                 MOV             X27, #0
.text:0000000140004D8C                 MOV             X28, #0
.text:0000000140004D90                 MOV             X29, #0
.text:0000000140004D94                 ERET
.text:0000000140004D94 ; End of function KiExit32BitMode
.text:0000000140004D94
```

KiEnter32BitMode (ARM64)

Keep in mind that meanwhile the 32-bit code is running, it cannot modify the value of the previously stored `X30` register - it is not visible in 32-bit mode. It stays there the whole time. Upon `UND #0xF8` execution, following happens:

- the `KiFetchOpcodeAndEmulate` function moves value of `X30` into `X24` register (not shown on the screenshot).
- `AND X19, X16, #0xFFFFFFFFFFFFFFC0` - bit [4] (among others) is being cleared in the `X19` register, which is later moved to the `SPSR_EL1` register. Clearing this bit **switches the execution mode back to 64-bits**.
- `KiExit32BitMode` then moves the value of `X24` register into the `ELR_EL1` register. That means when this function finishes its execution, the `ERET` brings us back to the 64bit code, right after the `SVC 0xFFFF` instruction.

  **NOTE:** It can be noticed that Windows uses `UND` instruction for several purposes. Common example might also be `UND #0xFE` which is used as a breakpoint instruction (equivalent of `__debugbreak()` / `int3`)

As you could spot, 3 kernel transitions are required for emulation of the system call (`SVC 0xFFFF`, system call itself, `UND 0xF8`). This is because on ARM there doesn't exist a way how to switch between 32-bit and 64-bit mode only in user-mode.

If you're looking for "ARM Heaven's Gate" - this is it. Put whatever function address you like into the `X15` register and execute `SVC 0xFFFF`. Next instruction will be executed in the 32-bit ARM mode, starting with that address. When you feel you'd like to come back into 64-bit mode, simply execute `UND #0xF8` and your execution will continue with the next instruction after the `SVC 0xFFFF`.

## Appendix

```
//////////////////////////////////////////////////////////////////////////////
// General definitions.
//////////////////////////////////////////////////////////////////////////////

//
// Context flags.
// winnt.h (Windows SDK)
//
```

```
#define CONTEXT_i386 0x00010000L
#define CONTEXT_AMD64 0x00100000L
#define CONTEXT_ARM 0x00200000L
#define CONTEXT_ARM64 0x00400000L
//
// Machine type.
// winnt.h (Windows SDK)
//
#define IMAGE_FILE_MACHINE_TARGET_HOST 0x0001 // Useful for indicating we want to interact with
the host and not a WoW guest.
#define IMAGE_FILE_MACHINE_I386 0x014c // Intel 386.
#define IMAGE_FILE_MACHINE_ARMNT 0x01c4 // ARM Thumb-2 Little-Endian
#define IMAGE_FILE_MACHINE_ARM64 0xAA64 // ARM64 Little-Endian
#define IMAGE_FILE_MACHINE_CHPE_X86 0x3A64 // Hybrid PE (defined in ntimage.h (WDK))
////////////////////////////////////////////////////////////////////////////
// ntoskrnl.exe
////////////////////////////////////////////////////////////////////////////
typedef struct _PS_NTDLL_EXPORT_ITEM {
PCSTR RoutineName;
PVOID RoutineAddress;
} PS_NTDLL_EXPORT_ITEM, *PPS_NTDLL_EXPORT_ITEM;
PS_NTDLL_EXPORT_ITEM NtdllExports[] = {
//
// 19 exports on x64
// 14 exports on ARM64
//
};
PVOID PsWowX86SharedInformation[Wow64SharedPageEntriesCount];
PS_NTDLL_EXPORT_ITEM NtdllWowX86Exports[] = {
{ "LdrInitializeThunk",
&PsWowX86SharedInformation[SharedNtdll32LdrInitializeThunk] },
{ "KiUserExceptionDispatcher",
&PsWowX86SharedInformation[SharedNtdll32KiUserExceptionDispatcher] },
{ "KiUserApcDispatcher",
&PsWowX86SharedInformation[SharedNtdll32KiUserApcDispatcher] },
{ "KiUserCallbackDispatcher",
&PsWowX86SharedInformation[SharedNtdll32KiUserCallbackDispatcher] },
{ "RtlUserThreadStart",
&PsWowX86SharedInformation[SharedNtdll32RtlUserThreadStart] },
{ "RtlpQueryProcessDebugInformationRemote",
&PsWowX86SharedInformation[SharedNtdll32pQueryProcessDebugInformationRemote] },
{ "LdrSystemDllInitBlock",
&PsWowX86SharedInformation[SharedNtdll32LdrSystemDllInitBlock] },
{ "RtlpFreezeTimeBias",
&PsWowX86SharedInformation[SharedNtdll32RtlpFreezeTimeBias] },
};
#ifdef _M_ARM64
PVOID PsWowArm32SharedInformation[Wow64SharedPageEntriesCount];
PS_NTDLL_EXPORT_ITEM NtdllWowArm32Exports[] = {
//
// ...
//
};
PVOID PsWowAmd64SharedInformation[Wow64SharedPageEntriesCount];
PS_NTDLL_EXPORT_ITEM NtdllWowAmd64Exports[] = {
//
// ...
//
};
PVOID PsWowChpeX86SharedInformation[Wow64SharedPageEntriesCount];
PS_NTDLL_EXPORT_ITEM NtdllWowChpeX86Exports[] = {
//
```

```
// ...
//
};
#endif // _M_ARM64
//
// ...
//
typedef struct _PS_NTDLL_EXPORT_INFORMATION {
PPS_NTDLL_EXPORT_ITEM NtdllExports;
SIZE_T Count;
} PS_NTDLL_EXPORT_INFORMATION, *PPS_NTDLL_EXPORT_INFORMATION;
//
// RTL_NUMBER_OF(NtdllExportInformation)
// == 6
// == (SYSTEM_DLL_TYPE)PsSystemDllTotalTypes
//
PS_NTDLL_EXPORT_INFORMATION NtdllExportInformation[PsSystemDllTotalTypes] = {
{ NtdllExports, RTL_NUMBER_OF(NtdllExports) },
{ NtdllWowX86Exports, RTL_NUMBER_OF(NtdllWowX86Exports) },
#ifdef _M_ARM64
{ NtdllWowArm32Exports, RTL_NUMBER_OF(NtdllWowArm32Exports) },
{ NtdllWowAmd64Exports, RTL_NUMBER_OF(NtdllWowAmd64Exports) },
{ NtdllWowChpeX86Exports, RTL_NUMBER_OF(NtdllWowChpeX86Exports) },
#endif // _M_ARM64
//
// { NULL, 0 } for the rest...
//
};
typedef struct _PS_SYSTEM_DLL_INFO {
//
// Flags.
// Initialized statically.
//
USHORT Flags;
//
// Machine type of this WoW64 NTDLL.
// Initialized statically.
// Examples:
// - IMAGE_FILE_MACHINE_I386
// - IMAGE_FILE_MACHINE_ARMNT
//
USHORT MachineType;
//
// Unused, always 0.
//
ULONG Reserved1;
//
// Path to the WoW64 NTDLL.
// Initialized statically.
// Examples:
// - "\\SystemRoot\\SysWOW64\\ntdll.dll"
// - "\\SystemRoot\\SysArm32\\ntdll.dll"
//
UNICODE_STRING Ntdll32Path;
//
// Image base of the DLL.
// Initialized at runtime by PspMapSystemDll.
// Equivalent of:
// RtlImageNtHeader(BaseAddress)->
// OptionalHeader.ImageBase;
//
```

```
PVOID ImageBase;
//
// Contains DLL name (such as "ntdll.dll" or
// "ntdll32.dll") before runtime initialization.
// Initialized at runtime by MmMapViewOfSectionEx,
// called from PspMapSystemDll.
//
union {
PVOID BaseAddress;
PWCHAR DllName;
};
//
// Unused, always 0.
//
PVOID Reserved2;
//
// Section relocation information.
//
PVOID SectionRelocationInformation;
//
// Unused, always 0.
//
PVOID Reserved3;
} PS_SYSTEM_DLL_INFO, *PPS_SYSTEM_DLL_INFO;
typedef struct _PS_SYSTEM_DLL {
//
// _SECTION* object of the DLL.
// Initialized at runtime by PspLocateSystemDll.
//
union {
EX_FAST_REF SectionObjectFastRef;
PVOID SectionObject;
};
//
// Push lock.
//
EX_PUSH_LOCK PushLock;
//
// System DLL information.
// This part is returned by PsQuerySystemDllInfo.
//
PS_SYSTEM_DLL_INFO SystemDllInfo;
} PS_SYSTEM_DLL, *PPS_SYSTEM_DLL;
////////////////////////////////////////////////////////////////////////////////
// ntdll.dll
////////////////////////////////////////////////////////////////////////////////
ULONG
RtlpArchContextFlagFromMachine(
_In_ USHORT MachineType
)
/*++
Routine description:
This routine translates architecture-specific CONTEXT
flag to the machine type.
Arguments:
MachineType - One of IMAGE_FILE_MACHINE_* values.
Return Value:
Context flag.
Note:
```

```
RtlpArchContextFlagFromMachine can be found only in
ntoskrnl.exe symbols, but from ntdll.dll disassembly
it is obvious that this function is present there
as well (probably __forceinline'd, or used as a macro).
--*/
{
switch (MachineType)
{
case IMAGE_FILE_MACHINE_I386:
return CONTEXT_i386;
case IMAGE_FILE_MACHINE_AMD64:
return CONTEXT_AMD64;
case IMAGE_FILE_MACHINE_ARMNT:
return CONTEXT_ARM;
case IMAGE_FILE_MACHINE_ARM64:
return CONTEXT_ARM64;
default:
return 0;
}
}
ULONG
RtlpGetLegacyContextLength(
_In_ ULONG ArchContextFlag,
_Out_opt_ PULONG SizeOfContext,
_Out_opt_ PULONG AlignOfContext
)
/*++
Routine description:
This routine determines size and alignment of the architecture-
-specific CONTEXT structure.
Arguments:
ArchContextFlag - Architecture-specific CONTEXT flag.
SizeOfContext - Receives sizeof(CONTEXT).
AlignOfContext - Receives __alignof(CONTEXT).
Return Value:
Alignment of the CONTEXT structure.
Note:
You can find corresponding DECLSPEC_ALIGN specifiers
for each CONTEXT structure in the winnt.h (Windows SDK).
By WOW64_CONTEXT_* here is meant an original CONTEXT
structure for the specific architecture (as CONTEXT
structures for other architectures are not available,
because it is selected during compile-time).
--*/
{
ULONG SizeOf = 0;
ULONG AlignOf = 0;
switch (ArchContextFlag)
{
case CONTEXT_i386:
SizeOf = sizeof(WOW64_CONTEXT_i386);
AlignOf = __alignof(WOW64_CONTEXT_i386); // 4
break;
case CONTEXT_AMD64:
SizeOf = sizeof(WOW64_CONTEXT_AMD64);
AlignOf = __alignof(WOW64_CONTEXT_AMD64); // 16
break;
case CONTEXT_ARM:
SizeOf = sizeof(WOW64_CONTEXT_ARM);
AlignOf = __alignof(WOW64_CONTEXT_ARM); // 8
```

```
break;
case CONTEXT_ARM64:
SizeOf = sizeof(WOW64_CONTEXT_ARM64);
AlignOf = __alignof(WOW64_CONTEXT_ARM64); // 16
break;
}
if (SizeOfContext) {
*SizeOfContext = SizeOf;
}
if (AlignOfContext) {
*AlignOfContext = AlignOf;
}
return AlignOf;
}
PULONG
RtlpGetContextFlagsLocation(
_In_ PCONTEXT_UNION Context,
_In_ ULONG ArchContextFlag
)
/*++
Routine description:
This routine returns pointer to the the "ContextFlags"
member of the CONTEXT structure.
Arguments:
Context - Architecture-specific CONTEXT structure.
ArchContextFlag - Architecture-specific CONTEXT flag.
Return Value:
Pointer to the the "ContextFlags" member.
--*/
{
//
// ContextFlags is always the first member of the
// CONTEXT struct - except for AMD64.
//
switch (ArchContextFlag)
{
case CONTEXT_i386:
return &Context->X86.ContextFlags; // Context + 0x00
case CONTEXT_AMD64:
return &Context->X64.ContextFlags; // Context + 0x30
case CONTEXT_ARM:
return &Context->ARM.ContextFlags; // Context + 0x00
case CONTEXT_ARM64:
return &Context->ARM64.ContextFlags; // Context + 0x00
default:
//
// Assume first member (Context + 0x00).
//
return (PULONG)Context;
}
}
//
// Architecture-specific WoW64 structure,
// holding the machine type and context
// structure.
//
#define WOW64_CPURESERVED_FLAG_RESET_STATE 1
typedef struct _WOW64_CPURESERVED {
USHORT Flags;
USHORT MachineType;
```

```
//
// CONTEXT has different alignment for
// each architecture and its location
// is determined at runtime (see
// RtlWow64GetCpuAreaInfo below).
//
// CONTEXT Context;
// CONTEXT_EX ContextEx;
//
} WOW64_CPURESERVED, *PWOW64_CPURESERVED;
typedef struct _WOW64_CPU_AREA_INFO {
PCONTEXT_UNION Context;
PCONTEXT_EX ContextEx;
PVOID ContextFlagsLocation;
PWOW64_CPURESERVED CpuReserved;
ULONG ContextFlag;
USHORT MachineType;
} WOW64_CPU_AREA_INFO, *PWOW64_CPU_AREA_INFO;
NTSTATUS
RtlWow64GetCpuAreaInfo(
_In_ PWOW64_CPURESERVED CpuReserved,
_In_ ULONG Reserved,
_Out_ PWOW64_CPU_AREA_INFO CpuAreaInfo
)
/*++
Routine description:
This routine returns architecture- and WoW64-specific
information based on the CPU-reserved region. It is
used mainly for fetching MachineType and the pointer
to the architecture-specific CONTEXT structure (which
is part of the WOW64_CPURESERVED structure). Because
the CONTEXT structure has different size and alignment
for each architecture, the pointer must be obtained
dynamically.
Arguments:
CpuReserved - WoW64 CPU-reserved region, usually located
at NtCurrentTeb()->TlsSlots[/* 1 */ WOW64_TLS_CPURESERVED]
Reserved - Unused. All callers set this argument to 0.
CpuAreaInfo - Receives the CPU-area information.
Return Value:
STATUS_SUCCESS - on success
STATUS_INVALID_PARAMETER - if CpuReserved contains invalid MachineType
--*/
{
ULONG ContextFlag;
ULONG SizeOfContext;
ULONG AlignOfContext;
//
// In the ntdll.dll, this call is probably inlined, because
// RtlpArchContextFlagFromMachine symbol is not present there.
//
ContextFlag = RtlpArchContextFlagFromMachine(CpuReserved->MachineType);
if (!ContextFlag) {
return STATUS_INVALID_PARAMETER;
}
RtlpGetLegacyContextLength(ContextFlag, &SizeOfContext, &AlignOfContext);
//
// CpuAreaInfo->Context = &CpuReserved->Context;
// CpuAreaInfo->ContextEx = &CpuReserved->ContextEx;
//
CpuAreaInfo->Context = ALIGN_UP_POINTER_BY(
```

```
(PUCHAR)CpuArea + sizeof(WOW64_CPU_AREA),
AlignOfContext
);
CpuAreaInfo->ContextEx = ALIGN_UP_POINTER_BY(
(PUCHAR)Context + SizeOfContext + sizeof(CONTEXT_EX),
sizeof(PVOID)
);
CpuAreaInfo->ContextFlagsLocation = ContextFlagsLocation;
CpuAreaInfo->CpuArea = CpuArea;
CpuAreaInfo->ContextFlag = ContextFlag;
CpuAreaInfo->MachineType = CpuReserved->MachineType;
return STATUS_SUCCESS;
}
////////////////////////////////////////////////////////////////////////
// wow64.dll
////////////////////////////////////////////////////////////////////////
//
// WOW64INFO, based on:
// wow64t.h (WRK: https://github.com/mic101/windows/blob/master/WRK-v1.2/public/internal/base/inc/wow64t.h#L269)
//
#define WOW64_CPUFLAGS_MSFT64 0x00000001
#define WOW64_CPUFLAGS_SOFTWARE 0x00000002
typedef struct _WOW64INFO {
ULONG NativeSystemPageSize;
ULONG CpuFlags;
ULONG Wow64ExecuteFlags;
ULONG Unknown1;
USHORT NativeMachineType;
USHORT EmulatedMachineType;
} WOW64INFO, *PWOW64INFO;
//
// Thread Local Storage (TLS) support. TLS slots are statically allocated.
// wow64tls.h (WRK: https://github.com/mic101/windows/blob/master/WRK-v1.2/public/internal/base/inc/wow64tls.h#L23)
// Note: Not all fields probably matches their names on Windows 10.
//
#define WOW64_TLS_STACKPTR64 0 // contains 64-bit stack ptr when simulating 32-bit code
#define WOW64_TLS_CPURESERVED 1 // per-thread data for the CPU simulator
#define WOW64_TLS_INCPUSIMULATION 2 // Set when inside the CPU
#define WOW64_TLS_TEMPLIST 3 // List of memory allocated in thunk call.
#define WOW64_TLS_EXCEPTIONADDR 4 // 32-bit exception address (used during exception unwinds)
#define WOW64_TLS_USERCALLBACKDATA 5 // Used by win32k callbacks
#define WOW64_TLS_EXTENDED_FLOAT 6 // Used in ia64 to pass in floating point
#define WOW64_TLS_APCLIST 7 // List of outstanding usermode APCs
#define WOW64_TLS_FILESYSREDIR 8 // Used to enable/disable the filesystem redirector
#define WOW64_TLS_LASTWOWCALL 9 // Pointer to the last wow call struct (Used when wowhistory is enabled)
#define WOW64_TLS_WOW64INFO 10 // Wow64Info address (structure shared between 32-bit and 64-bit code inside Wow64).
#define WOW64_TLS_INITIAL_TEB32 11 // A pointer to the 32-bit initial TEB
#define WOW64_TLS_PERFDATA 12 // A pointer to temporary timestamps used in perf measurement
#define WOW64_TLS_DEBUGGER_COMM 13 // Communicate with 32bit debugger for event notification
#define WOW64_TLS_INVALID_STARTUP_CONTEXT 14 // Used by IA64 to indicate an invalid startup context. After startup, it stores a pointer to the context.
#define WOW64_TLS_SLIST_FAULT 15 // Used to retry RtlpInterlockedPopEntrySList faults
#define WOW64_TLS_UNWIND_NATIVE_STACK 16 // Forces an unwind of the native 64-bit stack after an APC
#define WOW64_TLS_APC_WRAPPER 17 // Holds the Wow64 APC jacket routine
#define WOW64_TLS_IN_SUSPEND_THREAD 18 // Indicates the current thread is in the middle of NtSuspendThread. Used by software CPUs.
#define WOW64_TLS_MAX_NUMBER 19 // Maximum number of TLS slot entries to allocate
typedef struct _WOW64_ERROR_CASE {
```

```
ULONG Case;
NTSTATUS TransformedStatus;
} WOW64_ERROR_CASE, *PWOW64_ERROR_CASE;
typedef struct _WOW64_SERVICE_TABLE_DESCRIPTOR {
//
// struct _KSERVICE_TABLE_DESCRIPTOR {
//
// //
// // Pointer to a system call table (array of function pointers).
// //
//
// PULONG_PTR Base;
//
// //
// // Pointer to a system call count table.
// // This field has been set only on checked (debug) builds,
// // where the Count (with the corresponding system call index)
// // has been incremented with each system call.
// // On non-checked builds it is set to NULL.
// //
//
// PULONG Count;
//
// //
// // Maximum number of items in the system call table.
// // In ntoskrnl.exe it corresponds with the actual number
// // of system calls. In wow64.dll it is set to 4096.
// //
//
// ULONG Limit;
//
// //
// // Pointer to a system call argument table.
// // The elements in this table actually contain how many
// // bytes on the stack are assigned to the function parameters
// // for a particular system call.
// // On 32-bit systems, if you divide this number by 4, you'll
// // get the the number of arguments that the system call expects.
// //
//
// PUCHAR Number;
// };
//
KSERVICE_TABLE_DESCRIPTOR Descriptor;
//
// Extended fields of the WoW64 servie table:
// Wow64HandleSystemServiceError
//
WOW64_ERROR_CASE ErrorCaseDefault;
PWOW64_ERROR_CASE ErrorCase;
} WOW64_SERVICE_TABLE_DESCRIPTOR, *PWOW64_SERVICE_TABLE_DESCRIPTOR;
#define WOW64_NTDLL_SERVICE_INDEX 0
#define WOW64_WIN32U_SERVICE_INDEX 1
#define WOW64_KERNEL32_SERVICE_INDEX 2
#define WOW64_USER32_SERVICE_INDEX 3
#define WOW64_SERVICE_TABLE_MAX 4
WOW64_SERVICE_TABLE_DESCRIPTOR ServiceTables[WOW64_SERVICE_TABLE_MAX];
typedef struct _WOW64_LOG_SERVICE
{
PVOID Reserved;
PULONG Arguments;
ULONG ServiceTable;
```

```
ULONG ServiceNumber;
NTSTATUS Status;
BOOLEAN PostCall;
} WOW64_LOG_SERVICE, *PWOW64_LOG_SERVICE;
NTSTATUS
Wow64HandleSystemServiceError(
_In_ NTSTATUS ExceptionStatus,
_In_ PWOW64_LOG_SERVICE LogService
)
/*++
Routine description:
This routine transforms exception from native system
call to WoW64-compatible NTSTATUS.
Arguments:
ExceptionStatus - NTSTATUS raised from executing system call.
LogService - Information about the WoW64 system call.
Return Value:
Transformed NTSTATUS.
--*/
{
PWOW64_SERVICE_TABLE_DESCRIPTOR ServiceTable;
PWOW64_ERROR_CASE ErrorCaseTable;
ULONG ErrorCase;
NTSTATUS TransformedStatus;
ErrorCaseTable = ServiceTables[LogService->ServiceTable].ErrorCase;
if (!ErrorCaseTable)
{
ErrorCaseTable = &ServiceTables[LogService->ServiceTable].ErrorCaseDefault;
}
ErrorCase = ErrorCaseTable[LogService->ServiceNumber].ErrorCase;
TransformedStatus = ErrorCaseTable[LogService->ServiceNumber].TransformedStatus;
switch (ErrorCase)
{
case 0:
return ExceptionStatus;
case 1:
NtCurrentTeb()->LastErrorValue = RtlNtStatusToDosError(ExceptionStatus);
return ExceptionStatus;
case 2:
return TransformedStatus;
case 3:
NtCurrentTeb()->LastErrorValue = RtlNtStatusToDosError(ExceptionStatus);
return TransformedStatus;
default:
return STATUS_INVALID_PARAMETER;
}
}
```

view raw 2_appendix.h hosted with ❤ by GitHub

## References


How does one retrieve the 32-bit context of a Wow64 program from a 64-bit process on Windows Server 2003 x64?
http://www.nynaeve.net/?p=191

Mixing x86 with x64 code
http://blog.rewolf.pl/blog/?p=102

Windows 10 on ARM
https://channel9.msdn.com/Events/Build/2017/P4171

Knockin' on Heaven's Gate – Dynamic Processor Mode Switching
http://rce.co/knockin-on-heavens-gate-dynamic-processor-mode-switching/

Closing “Heaven’s Gate”
http://www.alex-ionescu.com/?p=300

- Previous Post
- Next Post