# Mixing x86 with x64 code

ReWolf

Few months ago I was doing some small research about possibility of running native **x64** code in **32-bits** processes under the **WoW64** layer. I was also checking it the other way round: run native **x86** code inside **64-bit**s processes. Both things are possible and as far as I googled some people used it already:

- http://vx.netlux.org/lib/vrg02.html
- http://www.corsix.org/content/dll-injection-and-wow64
- http://int0h.wordpress.com/2009/12/24/the-power-of-wow64/
- http://int0h.wordpress.com/2011/02/22/anti-anti-debugging-via-wow64/

Unfortunately I wasn't aware of any of above results when I was doing my research, so I'll just present my independent insights ;)

> **UPDATE:**
> All mentioned tricks (with necessary bugfixes and Windows 10 support) are currently part of the WoW64Ext library, that can be found on the github: https://github.com/rwfpl/rewolf-wow64ext

## x86 <-> x64 Transition

The easiest method to check how x86 <-> x64 transition is made is to look at any syscall in the 32-bits version of **ntdll.dll** from **x64** version of windows:

**32-bits ntdll from Win7 x86**

```
mov     eax, X

mov     edx, 7FFE0300h
call    dword ptr [edx]
        ;ntdll.KiFastSystemCall

retn    Z
```

**32-bits ntdll from Win7 x64**

```
mov     eax, X
mov     ecx, Y
lea     edx, [esp+4]
call    dword ptr fs:[0C0h]
        ;wow64cpu!X86SwitchTo64BitMode
add     esp, 4
ret     Z
```

As you may see, on the 64-bits systems there is a call to **fs:[0xC0]** (wow64cpu!X86SwitchTo64BitMode) instead of a standard call to ntdll.KiFastSystemCall. wow64cpu!X86SwitchTo64BitMode is implemented as a simple far jump into the 64-bits segment:

```
wow64cpu!X86SwitchTo64BitMode:
748c2320 jmp     0033:748C271E   ;wow64cpu!CpupReturnFromSimulatedCode
```

That's all magic behind switching **x64** and **x86** modes on 64-bits versions of Windows. Moreover it also works on non-**WoW64** processes (standard native 64-bits applications), so 32-bits code can be run inside

64-bits applications. Summing things up, for every process (**x86** & **x64**) running on 64-bits Windows there are allocated two code segments:

- **cs = 0x23 -> x86 mode**
- **cs = 0x33 -> x64 mode**

## Running x64 code inside 32-bits process

At first I've prepared few macros that will be used to mark beginning and end of the 64-bits code:

```
#define EM(a) __asm __emit (a)

#define X64_Start_with_CS(_cs) \
{ \
        EM(0x6A) EM(_cs)                        /*  push   _cs                  */
\
        EM(0xE8) EM(0) EM(0) EM(0) EM(0)        /*  call   $+5                  */
\
        EM(0x83) EM(4) EM(0x24) EM(5)           /*  add    dword [esp], 5       */
\
        EM(0xCB)                                /*  retf                        */
\
}

#define X64_End_with_CS(_cs) \
{ \
        EM(0xE8) EM(0) EM(0) EM(0) EM(0)        /*  call   $+5                  */
\
        EM(0xC7) EM(0x44) EM(0x24) EM(4)        /*                              */
\
        EM(_cs) EM(0) EM(0) EM(0)               /*  mov    dword [rsp + 4], _cs */
\
        EM(0x83) EM(4) EM(0x24) EM(0xD)         /*  add    dword [rsp], 0xD     */
\
        EM(0xCB)                                /*  retf                        */
\
}

#define X64_Start() X64_Start_with_CS(0x33)
#define X64_End() X64_End_with_CS(0x23)
```

**CPU** is switched into **x64** mode immediately after execution of the **X64_Start()** macro, and back to **x86** mode right after the **X64_End()** macro. Above macros are position independent thanks to the far return

opcode.

It would be also useful to have ability to call **x64** versions of **APIs**. I've tried to load **x64** version of **kernel32.dll** but it is not trivial task and I've failed, so I need to stick only with the **Native API**. The main problem with 64-bits version of **kernel32.dll** is that there is already loaded **x86** version of this library and **x64 kernel32.dll** have some additional checks that prevents proper loading. I believe that it is possible to achieve this goal through some nasty hooks that will intercept kernel32!BaseDllInitialize, but it is very complicated task. When I started this research, I was working on **Windows Vista** and I was able to load (with some hacks) 64-bits versions of **kernel32** and **user32** libraries but they were not fully functional, meanwhile I've switched to **Windows 7** and method that was used on **Vista** isn't working anymore.

Let's back to the topic, to use **Native APIs** I need to locate **x64** version of **ntdll.dll** in memory. To accomplish this task I'm parsing InLoadOrderModuleList from **_PEB_LDR_DATA** structure. 64-bits **_PEB** can be obtained from 64-bits **_TEB**, and obtaining 64-bits **_TEB** is similar to **x86** platform (on **x64** I need to use **gs** segment instead of **fs**) :

```
        mov   eax, gs:[0x30]
```

It can be even simpler, because wow64cpu!CpuSimulate (function responsible for switching **CPU** to **x86** mode) moves **gs:[0x30]** value into **r12** register, so my version of **getTEB64()** looks like this:

```
//to fool M$ inline asm compiler I'm using 2 DWORDs instead of DWORD64
//use of DWORD64 will generate wrong 'pop word ptr[]' and it will break stack
union reg64
{
        DWORD dw[2];
        DWORD64 v;
};

//macro that simplifies pushing x64 registers
#define X64_Push(r) EM(0x48 | ((r) >> 3)) EM(0x50 | ((r) & 7))

WOW64::TEB64* getTEB64()
{
        reg64 reg;
        reg.v = 0;

        X64_Start();
        //R12 register should always contain pointer to TEB64 in WoW64 processes
        X64_Push(_R12);
        //below pop will pop QWORD from stack, as we're in x64 mode now
        __asm pop reg.dw[0]
        X64_End();
```

```
        //upper 32 bits should be always 0 in WoW64 processes
        if (reg.dw[1] != 0)
                return 0;

        return (WOW64::TEB64*)reg.dw[0];
}
```

**WOW64** namespace is defined in “os_structs.h” file that will be appended at the end of this post with the rest of sample sources.

Function responsible for locating 64-bits **ntdll.dll** will be defined as follows:

```
DWORD getNTDLL64()
{
        static DWORD ntdll64 = 0;
        if (ntdll64 != 0)
                return ntdll64;

        WOW64::TEB64* teb64 = getTEB64();
        WOW64::PEB64* peb64 = teb64->ProcessEnvironmentBlock;
        WOW64::PEB_LDR_DATA64* ldr = peb64->Ldr;

        printf("TEB: %08X\n", (DWORD)teb64);
        printf("PEB: %08X\n", (DWORD)peb64);
        printf("LDR: %08X\n", (DWORD)ldr);

        printf("Loaded modules:\n");
        WOW64::LDR_DATA_TABLE_ENTRY64* head = \
                (WOW64::LDR_DATA_TABLE_ENTRY64*)ldr-
>InLoadOrderModuleList.Flink;
        do
        {
                printf("  %ws\n", head->BaseDllName.Buffer);
                if (memcmp(head->BaseDllName.Buffer, L"ntdll.dll",
                           head->BaseDllName.Length) == 0)
                {
                        ntdll64 = (DWORD)head->DllBase;
                }
                head = (WOW64::LDR_DATA_TABLE_ENTRY64*)head-
>InLoadOrderLinks.Flink;
        }
```

```
        while (head != (WOW64::LDR_DATA_TABLE_ENTRY64*)&ldr-
>InLoadOrderModuleList);
        printf("NTDLL x64: %08X\n", ntdll64);
        return ntdll64;
}
```

To fully support **x64 Native API** calling I'll also need some equivalent of GetProcAddress, which can be easily exchanged by ntdll!LdrGetProcedureAddress. Below code is responsible for obtaining address of LdrGetProcedureAddress:

```
DWORD getLdrGetProcedureAddress()
{
        BYTE* modBase = (BYTE*)getNTDLL64();
        IMAGE_NT_HEADERS64* inh = \
                (IMAGE_NT_HEADERS64*)(modBase + ((IMAGE_DOS_HEADER*)modBase)-
>e_lfanew);
        IMAGE_DATA_DIRECTORY& idd = \
                inh->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT];
        if (idd.VirtualAddress == 0)
                return 0;

        IMAGE_EXPORT_DIRECTORY* ied = \
                (IMAGE_EXPORT_DIRECTORY*)(modBase + idd.VirtualAddress);

        DWORD* rvaTable = (DWORD*)(modBase + ied->AddressOfFunctions);
        WORD* ordTable = (WORD*)(modBase + ied->AddressOfNameOrdinals);
        DWORD* nameTable = (DWORD*)(modBase + ied->AddressOfNames);
        //lazy search, there is no need to use binsearch for just one function
        for (DWORD i = 0; i < ied->NumberOfFunctions; i++)
        {
                if (strcmp((char*)modBase + nameTable[i],
"LdrGetProcedureAddress"))
                        continue;
                else
                        return (DWORD)(modBase + rvaTable[ordTable[i]]);
        }
        return 0;
}
```

As a cherry on top I'll present helper function that will enable me to call **x64 Native APIs** directly from the **x86 C/C++** code:

```
DWORD64 X64Call(DWORD func, int argC, ...)
{
        va_list args;
        va_start(args, argC);
        DWORD64 _rcx = (argC > 0) ? argC--, va_arg(args, DWORD64) : 0;
        DWORD64 _rdx = (argC > 0) ? argC--, va_arg(args, DWORD64) : 0;
        DWORD64 _r8 = (argC > 0) ? argC--, va_arg(args, DWORD64) : 0;
        DWORD64 _r9 = (argC > 0) ? argC--, va_arg(args, DWORD64) : 0;
        reg64 _rax;
        _rax.v = 0;

        DWORD64 restArgs = (DWORD64)&va_arg(args, DWORD64);

        //conversion to QWORD for easier use in inline assembly
        DWORD64 _argC = argC;
        DWORD64 _func = func;

        DWORD back_esp = 0;

        __asm
        {
                ;//keep original esp in back_esp variable
                mov    back_esp, esp

                ;//align esp to 8, without aligned stack some syscalls
                ;//may return errors !
                and    esp, 0xFFFFFFF8

                X64_Start();

                ;//fill first four arguments
                push   _rcx
                X64_Pop(_RCX);
                push   _rdx
                X64_Pop(_RDX);
                push   _r8
                X64_Pop(_R8);
                push   _r9
                X64_Pop(_R9);

                push   edi
```

```
        push   restArgs
        X64_Pop(_RDI);

        push   _argC
        X64_Pop(_RAX);

        ;//put rest of arguments on the stack
        test   eax, eax
        jz     _ls_e
        lea    edi, dword ptr [edi + 8*eax - 8]

        _ls:
        test   eax, eax
        jz     _ls_e
        push   dword ptr [edi]
        sub    edi, 8
        sub    eax, 1
        jmp    _ls
        _ls_e:

        ;//create stack space for spilling registers
        sub    esp, 0x20

        call   _func

        ;//cleanup stack
        push   _argC
        X64_Pop(_RCX);
        lea    esp, dword ptr [esp + 8*ecx + 0x20]

        pop    edi

        ;//set return value
        X64_Push(_RAX);
        pop    _rax.dw[0]

        X64_End();

        mov    esp, back_esp
}
```

```
        return _rax.v;
}
```

Function is a bit long, but there are comments and the whole idea is pretty simple. The first argument is address of **x64** function that I want to call, second argument is number of arguments that specific function takes. Rest of the arguments depends on the function that is supposed to be called, all of them should be casted to **DWORD64**. Small example of **X64Call**() usage:

```
DWORD64 GetProcAddress64(DWORD module, char* funcName)
{
        static DWORD _LdrGetProcedureAddress = 0;
        if (_LdrGetProcedureAddress == 0)
        {
                _LdrGetProcedureAddress = getLdrGetProcedureAddress();
                printf("LdrGetProcedureAddress: %08X\n",
_LdrGetProcedureAddress);
                if (_LdrGetProcedureAddress == 0)
                        return 0;
        }

        WOW64::ANSI_STRING64 fName = { 0 };
        fName.Buffer = funcName;
        fName.Length = strlen(funcName);
        fName.MaximumLength = fName.Length + 1;
        DWORD64 funcRet = 0;
        X64Call(_LdrGetProcedureAddress, 4,
                (DWORD64)module, (DWORD64)&fName,
                (DWORD64)0, (DWORD64)&funcRet);

        printf("%s: %08X\n", funcName, (DWORD)funcRet);
        return funcRet;
}
```

## Running x86 code inside 64-bits process

It is very similar to the previous case with just one small inconvenience. Because **64-bits** version of **MS C/C++** compiler doesn't support inline assembly, all tricks should be done in a separate .asm file. Below there are definitions of **X86_Start** and **X86_End** macros for **MASM64**:

```
X86_Start MACRO
        LOCAL  xx, rt
        call   $+5
        xx     equ $
```

```
        mov     dword ptr [rsp + 4], 23h
        add     dword ptr [rsp], rt - xx
        retf
        rt:
ENDM

X86_End MACRO
        db 6Ah, 33h                     ; push  33h
        db 0E8h, 0, 0, 0, 0             ; call  $+5
        db 83h, 4, 24h, 5               ; add   dword ptr [esp], 5
        db 0CBh                         ; retf
ENDM
```

## Ending notes

Link to source code used in the article: http://rewolf.pl/stuff/x86tox64.zip

> **UPDATE:**
> All mentioned tricks (with necessary bugfixes and Windows 10 support) are currently part of the WoW64Ext library, that can be found on the github: https://github.com/rwfpl/rewolf-wow64ext