Mixing x86 with x64 code

July 4, 2011 / ReWolf posted in papers, programming, reverse engineering, source code,x64 / No Comments

在几个月之前，我在做一些小研究是关于在 WoW64 下的 32 位进程中运行原生 x64 代码的可能性。第二个想法是在 64 位进程内部运行原生 x86 代码。这两种想法都是可行的，并且我 google 到了一些已经被使用的方法：

- 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/

太不幸了，在我做研究的时候并没有注意到上面的任何一个成果，所以我提出的仅是我自己的见解。

x86 <-> x64 过度

最简单的检测 x86 <-> x64 之间如何过度的方法是在 x64 版本下的 ntdll.dll 中查看 32 位的 syscall。

32-bits ntdll from Win7 x86	32-bits ntdll from Win7 x64

mov eax, X
 	
mov edx, 7FFE0300h
call dword ptr [edx]
;ntdll.KiFastSystemCall
 	
retn Z
	mov eax, X
	mov ecx, Y
	lea edx, [esp+4]
	call dword ptr fs:[0C0h]
	;wow64cpu!X86SwitchTo64BitMode
	add esp, 4
	ret Z

和你想的一样，在 64 位系统上 **fs:[0xC0]**(wow64cpu!X86SwitchTo64BitMode) 代替了 ntdll.KiFastSystemCall。wow64cpu!X86SwitchTo64BitMode 被实现用来做为一个简单的 far jump 进入到 64 位段：

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

这就是在 64 位 Windows 上切换 x64 和 x86 模式的戏法。另外，它还可以工作在非 WoW64 进程（标准的原生 64 位应用程序）上，因此 32 位代码能被运行于 64 位应用程序中。由此可以总结，对于所有（x86 和 x64）运行于 64 位 Windows 的进程都分配了两个代码段：

- **cs = 0×23 -> x86 模式**
- **cs = 0×33 -> x64 模式**

在 32 位进程里运行 64 位代码

首先我准备了一些宏用来标记 64 位代码的开始和结束：

```
#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)
```

在执行到 **X64_Start()** 宏的时候，**CPU** 会被立即切换到 x64 模式，**X64_End()** 执行之后会返回到

X86 模式。由于 far return opcode 的作用，上面的宏是位置无关的。

它也有调用 64 位版本 API 的能力。我尝试过加载 64 位版本的 **kernel32.dll**，但是这并不容易并且我失败了，所以我只有坚持使用 **Native API**。使用 64 位版本的 **kernel32.dll** 的主要问题是已经加载了 **x86** 版本的库，并且 **x64 kernel32.dll** 有一些额外检查来保证正确加载。我相信使用一些讨厌的钩子拦截 kernel32!BaseDllInitialize 以达到这个目的是有可能的，但这样非常复杂。我开始这个研究的时候是在 **Windows Vista** 上工作的，并且我已经能加载 64 位的 **kernel32** 和 **user32** 库，但是它们没有完全发挥作用，同时我已经切换平台到 **Windows 7** 上了，在 **Vista** 上使用的方法再也无法工作了。

让我们回到主题，要使用 **Native APIs** 就得定位到内存中的 64 位 ntdll.dll。为了完成这个任务，我分析了**_PEB_LDR_DATA** 结构体中的 InLoadOrderModuleList 成员。64 位的 **_PEB** 可以从 64 位的 **_TEB** 获取，获取 64 位的**_TEB** 与 x86 平台类似（在 x64 上我需要使用 **gs** 段而不是 **fs**）：

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

它还可以更简洁，由于 wow64cpu!CpuSimulate（负责切换 CPU 到 x86 模式的函数）传送 **gs:[0x30]** 值给 **r12** 寄存器，所以我的 **getTEB64()**看起来是这样的：

```
// 为了糊弄 M$ 内联汇编器， 我使用 2 个 DWORD 代替 DWORD64
// 使用 DWORD64 会生成错误的 'pop word ptr[]' 并且将破坏栈
union reg64
{
	DWORD dw[2];
	DWORD64 v;
};

// 宏简化了 64 位寄存器的入栈
#define X64_Push(r) EM(0x48 | ((r) >> 3)) EM(0x50 | ((r) & 7))

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

	X64_Start();
	// 在 WoW64 进程中 R12 应该总是包含指向 TEB64 的指针
	X64_Push(_R12);
	// 下面的 pop 会把 QWORD 从栈中弹出，我们现在在 x64 模式下
	__asm pop reg.dw[0]
	X64_End();

	// 在 WoW64 进程中高 32 位应该总是 0
	if (reg.dw[1] != 0)
		return 0;

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

WOW64 命名空间定义在“os_structs.h” 文件中，它会和其余的示例被附带在文章的末尾。

负责定位 64 位 ntdll.dll 的函数被定义如下：

```
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->InLoadOrderModuleLi
st.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;
}
```

为了完全支持 x64 原生 API 调用，我还需要一些等价的 GetProcAddress，它们可以轻易地被 ntdll!LdrGetProcedureAddress 交换。下面的代码用来获取 LdrGetProcedureAddress 的地址：

```
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);
	// 不搜索了，不需要为了一个函数使用 binsearch
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:

来一个锦上添花，我要提供辅助函数从 **x86 C/C++** 代码直接调用 **x64 Native APIs**。

```
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);

	// 转换为 QWORD 以便以内联汇编中更易使用
	DWORD64 _argC = argC;
```

```
	DWORD64 _func = func;

	DWORD back_esp = 0;

	__asm
	{
		;// 保存原始的 esp 到 back_esp 变量中
		mov    back_esp, esp

		;// 对齐 esp 到 8, 一些 syscalls 没有对齐栈
		;// 可能返回错误!
		and    esp, 0xFFFFFFF8

		X64_Start();

		;// 填充前 4 个参数
		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);

		;// 其余的参数放在栈上
		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:

                ;// 开辟栈空间用来溢出寄存器
                sub    esp, 0x20

                call   _func

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

                pop    edi

                ;// 设置返回值
                X64_Push(_RAX);
                pop    _rax.dw[0]

                X64_End();

                mov    esp, back_esp
        }
        return _rax.v;
}
```

函数有一点长，不过加了注释并且整个思路非常简单。第一个参数是我要调用的 x64 函数的地址，第二个参数是指定的函数所需要的参数个数。其余的参数取决所依赖的函数，它们都应该被转换为 **DWORD64**。**X64Call**() 用法的小例子：

```
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;
}
```

在 64 位进程中运行 x86 代码

这和之前的情况非常类似，只是有一点点不方便的地方。由于 64 位版本的 **MS C/C++** 编译器不支持内联汇编，所有的东西都要在一个单独的 .asm 文件中完成。下面定义的 **X86_Start** 和 **X86_End** 用于 **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
```

结束语

文章中所提到的代码： http://rewolf.pl/stuff/x86tox64.zip