StraceNT – System Call Tracer for Windows NT

(**Written by**: **Pankaj Garg**)

0 Objective

This document discusses various API spying/hooking techniques for Windows and delves into details of IAT patching technique. It then describes the implementation of StraceNT for Windows. It also gives information about how stack is managed on x86 and also briefly discusses a minimalist debugger implementation.

1 Introduction

strace is a utility on Linux which can be used to trace all system calls made by a target process. It comes quite handy at times for debugging problems like deadlock or tracing the flow of a program. StraceNT is an attempt to provide a similar utility for Windows. Many debugging tools like Numega's Bounds checker, Identify software's AppSight are also available on windows which provide system call tracing to aid debugging but none is as easy to use Strace and most of them require debug build of application with debug symbols (.pdb files). StraceNT for Windows is implemented to be a lightweight and easy to use utility with limited functionality and it is *not* meant to replace commercial tools like Bounds checker. StraceNT is distributed free of cost for non-commercial as well as commercial use.

2 Requirements

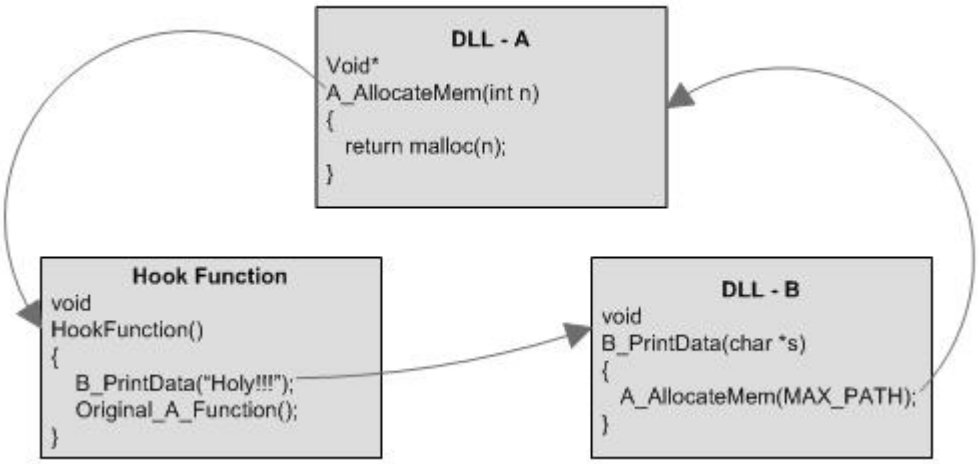
Below are a set of features that are required in StraceNT for it to be usable:

- StraceNT should be very efficient and it should not slow down the target application a lot.
- Its code should be maintainable and extendable so that spying on new DLLs is easy.
- It should be able to spy on any DLL.
- It should be able to log information in the same format as strace on Linux i.e. "*api_name(param1, param2....) = return_value*". This requirement is only to make Windows version look more like Linux version.

3 Windows System Services

Windows provides its system services through NTOSKRNL.EXE. A user mode interface for that is wrapped in NTDLL.DLL. This DLL requests system services from NTOSKRNL using instruction ***int 2E***, which causes a switch from user mode to kernel mode and then NTOSKRNL executes the requested system service. However, interface of NTDLL is not public and user mode applications use other DLLs such as KERNEL32.DLL to access the system services. Due to this reason there would be little use if we trace API calls to NTDLL because normally people don't use them directly. Hence I decided to hook API calls to other DLLs such as KERNEL32, USER32 etc. The interface of these DLLs is public and well documented. People mostly use these DLLs in their application instead of calling undocumented NTDLL functions. One more reason to avoid hooking NTDLL APIs

is that they sometime cause recursive hooking. Let's say we have two DLLs A and B, B uses functions from A and our hook function uses functions from B. Now if we hook functions of A, as soon as our hook function is called, it will call functions from B and if that function from B calls a function from A, then our hook function will be called again which will again call function from B and so on till the target process gets a stack overflow exception and crash. A diagram to depict such situation is shown below:



In our hook function implementation we rely upon kernel32.dll to do certain processing. Kernel32.dll imports functions from ntdll.dll, so if we hook ntdll.dll, we run into chances of circular hooking quite often.

4 Hooking techniques

There are many different ways for API spying in windows, each having its own advantages and disadvantages. We will analyze each of these different API spying techniques and decide which technique will be best suited for an application like StraceNT. We will keep the focus on General purpose API spying rather than specific techniques like Winsock hooking or Browser Helper objects which are easy to implement for specific tasks. Details on various Hooking methods can be found here[1].

4.1 Clone DLL

This is one of the easiest methods of API spying. Suppose we want to spy all calls made by a target process to PSAPI. We can create our own version of PSAPI that exports exactly the same functions as exported by PSAPI. If we copy this DLL into the target process's directory, then instead of original psapi.dll, our version of psapi.dll will be loaded by the process when it starts. All PSAPI calls will now get sent to our DLL where we can then do the required processing and call the original psapi.dll function.

Advantages

- Easy to implement

- Gives complete control over the API including its parameters and return value

Disadvantages

- Need to write stubs for all the exported functions of the DLL, even if we don't want to spy them.
- Difficult to maintain if the DLL's export function changes over time.

4.2 Import Address Table (IAT) patching

This technique is used by Windows loader to load a program in memory for execution. To understand how this process works we need to dig a little bit (actually a tiny bit) deeper into windows executable file format and how windows loader executes a program.

Windows stores its executables in a special format called Portable Executable or PE format. Every PE file contains a special section called *Import Address Table* (IAT) to store information about the imported functions from various DLLs. Calls made to these functions are in the form of 6 byte indirect call instruction e.g. if *notepad.exe* uses a function *IsSystemDirectory* imported from a DLL then the IAT of *notepad.exe* will contain an entry to another table which lists all the the functions imported from that DLL. One of the entry in this table will contain address of *IsSystemDirectory*. All the calls made by *notepad.exe* to *IsSystemDirectory* will be in the form of "call dword ptr [_imp_IsSystemDirectory]" where _imp_IsSystemDirectory represents the IAT entry which will contains the actual address of IsSystemDirectory during the execution of *notepad.exe*.

To execute a program (PE file), windows loader allocates memory in Virtual Address space and maps that PE file in the allocated memory. Each PE file has a preferred base address, where loader tries to load it. If the preferred address is not available, then the loader loads it at the available address and performs relocation[2] on the in-memory copy of the executable. Once relocation process is done, loader walks through the Import Address Table and loads each DLL one by one. The process of loading each DLL is exactly same as loading the executable. Once all the DLLs are loaded, loader walks through the IAT of each loaded module (exe and dll) and performs an address fix-up to point to the actual in-memory address of the imported function.

To hook *IsSystemDirectory* in *notepad.exe*, all we need to do is replace the IAT entry which represents *IsSystemDirectory* with address of the hook function. This will cause all calls made by *notepad.exe* to *IsSystemDirectory* to be routed to the hook function. The hook function then can chose to pass control to original function or take some other action. More information on this topic can be found in Matt Pietrek's arcticle[3].

Advantages

- Need to only implement stubs for APIs that needs to be patched
- IAT patching is standard mechanism in windows and also used by Windows program loader.

Disadvantages

- Have to deal with issues on how to execute our hook functions in the context of the target process.
- Implementation is relatively complex.

4.3 Minimalist debugger

In this technique the spying application can act as a debugger and insert x86 "int 3" as the first instruction for all the APIs that we wish to spy on. As soon as this instruction is hit, a STATUS_BREAKPOINT exception will occur and in our spying application, we can do required processing like logging API information, generating call stack etc (much like a full fledged debugger).

Advantages

- Relatively easier to implement than IAT patching.
- It is more maintainable because the STATUS_BREAKPOINT exception handler in the hooking application can handle all the APIs.
- We can easily improve it to take advantage if the application is a debug build application and record much more information than just function arguments and return value.

Disadvantages

- Windows exception handling is too slow which makes it non-scalable
- Catching return value of an API would be difficult

5 Implementation

The requirements for StraceNT and our analysis of API spying techniques above shows IAT patching as the most suited solution for StraceNT. It is one of the fastest solutions and it does not require writing stubs for all the functions of the DLL even if we want to spy only few of them. Debugger method will not be used due to slowness of exception handling even though it is easy to implement and more maintainable than IAT patching. We will now discuss the implementation of StraceNT by disseminating various pieces:

5.1 Import Address Table (IAT) Patching

5.1.1 Processes and modules

Before we go further into the details of IAT patching, it will be helpful to agree upon few terms here:

- **Process** – is a running instance of an executable on Windows.
- **Module** – is the active instance of a PE binary inside a process. To understand this better lets take example of a hypothetical windows executable called “hypo.exe”. This executable (hypo.exe) uses functions

imported from kernel32.dll. To execute this executable, window loads hypo.exe in memory and then load kernel32.dll because hypo.exe is dependent upon kernel32.dll. Thus we have two modules for the process hypo.exe which are “hypo.exe” and “kernel32.dll”.

5.1.2 Traversing IAT of a module

Each module inside a process is loaded at a distinct address. We can obtain the list and address of each loaded modules inside a process using ***PSAPI*** functions such as EnumProcessModules (*documented in MSDN)*. Once we obtain the address of a module inside a process, we can easily find the IAT of that module. The sample code below shows how to find the IAT of a module using its loaded base address:

```
void
PatchModule(
      LPVOID moduleBaseAddress)
{
PIMAGE_DOS_HEADER pIDH = (PIMAGE_DOS_HEADER)moduleBaseAd dress;

PIMAGE_NT_HEADERS pINTH =
      (PIMAGE_NT_HEADERS)
            (moduleBaseAddress + pIDH->e_lfanew);

DWORD dwImportTableOffset =
      pINTH->OptionalHeader.DataDirectory
            [IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress;

PIMAGE_IMPORT_DESCRIPTOR pIID =
      (PIMAGE_IMPORT_DESCRIPTOR)
                  (moduleBaseAddress + dwImportTableOffset);

//
// At this point pIID points to an array of Import Address Table, each
// entry of the array is specific to a particular DLL. The code below
// shows how we can traverse each array and fin d out which functions
// are imported from which DLL.
//
while (pIID->FirstThunk != 0 &&
       pIID->OriginalFirstThunk != 0)
{
      //
      // DLL name from which functions are imported
      //
      LPSTR pszModuleName =
            (LPSTR)(moduleBaseAddress + pIID->Name);

      //
      // First thunk points to IMAGE_THUNK_DATA
      //
      PIMAGE_THUNK_DATA pITDA = (PIMAGE_THUNK_DATA)
            (moduleBaseAddress + (DWORD)pIID->FirstThunk);

      //
      // OriginalFirstThunk points to IMAGE_IMPORT_BY_NAME array but
      // due to its identical structure to IMAGE_THUNK_ DATA, we can use
```

```
	// IMAGE_THUNK_DATA object to dereference it.
	//
	PIMAGE_THUNK_DATA pIINA = (PIMAGE_THUNK_DATA)
		(moduleBaseAddress + (DWORD)pIID->OriginalFirstThunk);

	// While Function address is not NULL
	while (pITDA->u1.Ordinal != 0)
	{
		// Ordinal and Function are a part of union u1
		// pFunction gives the address of function which
		// this IAT entry correspond too.
		PVOID pFunction = (PVOID)pITDA->u1.Function;



		//
		// If the function is Imported by name
		//
		if (!IMAGE_SNAP_BY_ORDINAL(pIINA->u1.Ordinal))
		{
			PIMAGE_IMPORT_BY_NAME pIIN = (PIMAGE_IMPORT_BY_NAME)
				(moduleBaseAddress + pIINA->u1.AddressOfData);

			// pIIN->Name - points to name of the function

			HookApi(pITDA, (LPSTR)pIIN->Name);
		}

		pITDA++;
		pIINA++;
	}

	pIID++;
}
}
```

Extensive details of the code above are beyond the scope of our current discussion but more information on this topic can be found in PE tutorial on MSDN[4].

5.1.3 Patching an IAT entry

The next thing we need to do is to patch the IAT table entries with our hook function address so that all patched function calls are redirected to our hook function instead. To patch an IAT entry, we will first make the memory location of IAT entry writable and then simply replace the Function address stored inside that IAT entry with our hook function address. The code below shows how to do this:

```
void
PatchIATEntry(
	PIMAGE_THUNK_DATA pITDA,
	LPVOID            myHookFunction)
{

DWORD dwOldProtect;
```

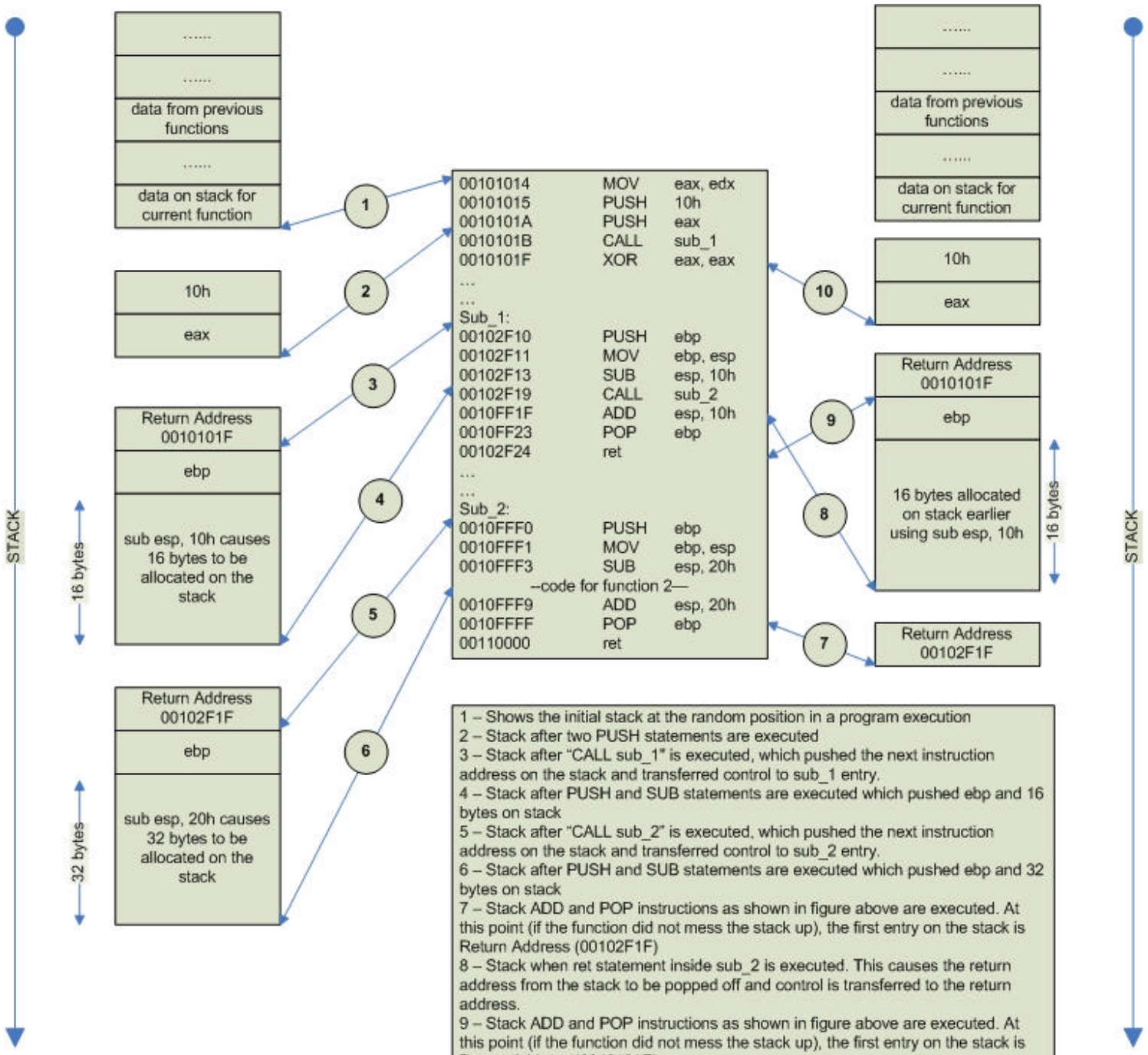
```
DWORD dwTemp;

// Make the page writable
if (VirtualProtect(
            &pITDA->u1.Function,
            sizeof(DWORD),
            PAGE_READWRITE,
            &dwOldProtect))
{
      // Point the IAT entry to our hook function
      pITDA->u1.Function = (DWORD)myHookFunction;

      VirtualProtect(
            &pITDA->u1.Function,
            sizeof(DWORD),
            dwOldProtect,
            &dwTemp);
}
}
```

5.1.4 Implementing Hook function

We discussed how to find and patch an IAT entry and redirect all patched functions to our hook function. Now we will discuss the design of our generic hook function. Recall from the "Requirements" section that to make StraceNT useful and maintainable, we need to write single (or generic) hook function for all the functions. Seems trivial? Think again! The biggest problem with implementing a generic hook functions is that if we don't know how many arguments the original function expect, then how can our hook function pass control to the original function (without messing the stack up)? The other problem has to do with different calling conventions[5]. How to handle different function with different calling conventions like __fastcall, __stdcall or __cdecl? We devised a unique solution to solve this problem but before we delve into the details of that, we need to understand how stack is managed in x86 architecture.

5.1.4.1 Stack management on x86

The diagram below shows the execution of a piece of code and the state of stack as the program is executing. To make things easier to understand, I drew the stack on both side of the code. When we are calling functions, data is getting pushed on the stack and when we are returning from functions, data is getting popped off the stack. Also note that on x86 architecture, stack grows from top to bottom so as we push something on the stack, new items appear at the bottom.



1 – Shows the initial stack at the random position in a program execution
2 – Stack after two PUSH statements are executed
3 – Stack after "CALL sub_1" is executed, which pushed the next instruction address on the stack and transferred control to sub_1 entry.
4 – Stack after PUSH and SUB statements are executed which pushed ebp and 16 bytes on stack
5 – Stack after "CALL sub_2" is executed, which pushed the next instruction address on the stack and transferred control to sub_2 entry.
6 – Stack after PUSH and SUB statements are executed which pushed ebp and 32 bytes on stack
7 – Stack ADD and POP instructions as shown in figure above are executed. At this point (if the function did not mess the stack up), the first entry on the stack is Return Address (00102F1F)
8 – Stack when ret statement inside sub_2 is executed. This causes the return address from the stack to be popped off and control is transferred to the return address.
9 – Stack ADD and POP instructions as shown in figure above are executed. At this point (if the function did not mess the stack up), the first entry on the stack is Return Address (0010101F)
10 – Stack when ret statement inside sub_1 is executed. This causes the return address from the stack to be popped off and control is transferred to the return address.

5.1.4.2 Hook Function Design

At this point we have enough background knowledge to design our hook function. I decided to implement the hook function as a naked[6] function because of few advantages discussed further in this article. The IAT entry of the DLL to patch is replaced with the address of in-memory executable instructions as shown below:

```
void
HookApi(
      PIMAGE_THUNK_DATA pITDA,
      LPSTR             inApiName)
{
//
// pITDA -  is the Image thunk data entry for the IAT entry
// that we want to patch. The entry is located as shown
// in section 5.1.2 i.e. Traversing the IAT Table
//

//
// inApiName - Contains the name of the function we are
// going to hook
//

//
// Allocate memory in the process for executable instructions
// ioPatchCode is the address with which the IAT entry will be
// replaced.
//
unsigned char *ioPatchCode =
                  VirtualAlloc(NULL,
                               14,
                               MEM_COMMIT,
                               PAGE_EXECUTE_READWRITE);

if (ioPatchCode)
{
      //
      // Store the API name and its original function address
      // in a global array.
      //
      // Implementation of InsertNewAPIName is left to  reader as an
      // exercise.
      //
      DWORD apiIndex = InsertNewAPIName(pITDA->u1.Function, inApiName);

      //
      // The code below is in 6-byte indirect CALL instruction
      // format. Once this code is executed, apiIndex is
      // pushed on the stack as return address and control is
      // transferred to HookFunctionEntry
      //
      ioPatchCode[0]                      = 0xFF;
      ioPatchCode[1]                      = 0x15;
      memcpy(&ioPatchCode[2], (DWORD)&ioPatchCode[10]);
      memcpy(&ioPatchCode[6], (DWORD)apiIndex);
```

```
        memcpy(&ioPatchCode[10], (DWORD)&HookFunctionEntry);

        //
        // The function below is implemented in section 5.1.3
        // i.e. Patching an IAT entry
        //
        PatchIATEntry(pITDA, ioPatchCode);
}
}
```

Function *HookFunctionEntry* is implemented as a naked function to force the compiler to *not* generate function prolog and epilog. The usual function prolog and epilog, generated by the compiler, makes it hard for a function to manipulate the stack but with naked function we can do so. The actual work is done by *HookFunctionProcessing* and *HookFunctionEntry* is used so that we can manipulate the stack in whatever way we want.

```
__declspec (naked)
void
HookFunctionEntry()
{
//
// We only need to preserve Callee saved registers
// which are ebx, esi and edi, So i can safely modify
// eax, ecx and edx here.
//
__asm
{
        push  ebx
        pushf
        pushf

        mov         ebx, esp
        add         ebx, 8
        push  edx
        push  ecx
        push  ebx
        call  HookFunctionProcessing

        popf
        popf
        pop         ebx

        ; Pop off as many bytes of stack now
        ; as popped off by original API
        add         esp, edx

        ret
}
}
```

HookFunctionEntry first saves the required registers on the stack and then move the value of stack pointer ESP to EBX and subtract 8 from EBX to compensate for the registers we saved on stack. This value of EBX will be used to access the original stack for the patched function. ECX and EDX are passed

as arguments so that this hook function can hook both C++ and __fastcall functions. After pushing ECX and EDX on stack, it calls *HookFunctionProcessing* which does actual hook processing. On return, *HookFunctionProcessing* sets the number of bytes original function popped off from the stack in EDX. We restore our saved registers and then pop off the number of bytes specified in EDX off the stack to make sure that we removed exactly as many items off the stack as original API would have without any patching. This proper stack manipulation is *most important* to make this hooking work without causing a crash. Once stack is fixed, we simply return and calling process has no idea that the function it called was hooked.

HookFunctionProcessing does the actual processing like logging API information, calling original API, then recording its return value. The code below shows how it is done:

```
typedef PVOID  (_stdcall *PFNORIGINAL)(void);

void
__stdcall
HookFunctionProcessing(
      DWORD **ppStackPos,
      DWORD inECX,
      DWORD inEDX)
{
//
// Initially this stack location contains apiIndex
// that is pushed by the compiler, when the instruction
// call is executed by the compiler for this function
//
DWORD apiIndex                = **ppStackPos;

//
// ppStackPos also points to the return address
// which we will replace with original API that we patched.
//
DWORD *pReturnAddress   = (DWORD *)ppStackPos;

//
// Original return address is the address where the original
// API would have returned.
//
DWORD *pOriginalRetAddr = (DWORD *)(ppStackPos + 1);

//
// pFirstParam points to first argument for original API
//
DWORD *pFirstParam            = (DWORD *)(ppStackPos + 2);
LPVOID valueReturn            = NULL;
DWORD errorCode               = 0;
char szStr[1024];
LPCSTR str                    = NULL;

//
```

```
// GetOrigFuncAddr - shall retrive original function address
// that we patched for the apiIndex from the global array.
//
// Its implementation is left for reader as an exercise.
//
PFNORIGINAL pOrgFunc    = (PFNORIGINAL)GetOrigFuncAddr(apiIndex);

//
// used to manage stack
//
DWORD dwESP;
DWORD dwNewESP;
DWORD dwESPDiff;


//
// Log API Parameters Information
//
// GetAPIName - Shall retrive the API name from global array
// Its implementation is left for reader as an exercise.
//
str = GetAPIName(apiIndex);
sprintf(    szStr,
            "$[T%d] %s(%x, %x, %x, %x, ...) ",
            GetCurrentThreadId(),
            str,
            *pFirstParam,
            *(pFirstParam+1),
            *(pFirstParam+2),
            *(pFirstParam+3));

//
// Print API information in a debugger
//
OutputDebugStringA(szStr);

//
// Note:
// What we do here is kind of tricky. We make space for 100 bytes
// i.e. 25 paramters on stack. After that we copy the original
// 100 bytes from the original API stack to this location and call
// the original API. After the original API return, we see the
// difference in esp to determine, how many bytes it popped off
// because we need to pop off that many bytes once we return from
// ihiPatchProlog which was our detour function for original API.
//
// Warning!!!
// If a function takes more than 25 parameters, we are screwed.
//
__asm
{
    pushad
    mov     dwESP,      esp
    sub     esp,        100
    mov     dwNewESP,   esp
}
```

```
memcpy((PVOID)dwNewESP, (PVOID)pFirstParam, 100);

//
// for C++ functions we need to restore ecx because it contains
// this pointer.
// for __fastcall functions we need to restore ecx and edx because
// they contain first and second argument respectively
//
__asm
{
      mov         ecx,        inECX
      mov         edx,        inEDX
}

valueReturn = (*pOrgFunc)();

//
// At this point, esp is messed up because
// original function might have removed only
// partial number of parameters. We need to find
// our how many did it remove
//
__asm
{
      mov         dwNewESP,   esp
      mov         esp,        dwESP
      popad
}

//
// We always need to fix the error code because
// many times its reset due to the API calls that
// we make. So save error code here.
//
errorCode = GetLastError();

//
// Log API Return value Information
//
sprintf(    szStr,
            "$= %x\n",
            valueReturn);

OutputDebugStringA(szStr);

//
// This is the size that we need to pop when we return
// because this is the stack difference, when we called
// the original API. This means that we will pop off
// same number of bytes as done by original API, once we
// return from HookFunctionEntry
//
dwESPDiff = dwNewESP - (dwESP - 100);

//
// The address where HookFunctionEntry will return to
// should point to the original return address of original API
```

```
// so that once HookFunctionEntry returns, normal code execution
// can continue
//
*(pReturnAddress + 1 + (dwESPDiff / 4)) = *pOriginalRetAddr;

//
// Add 4 to remove extra return address (apiIndex) stored by call to
// HookFunctionEntry
//
dwESPDiff += 4;

//
// Restore error code here
//
SetLastError(errorCode);

// Set the registers for use in HookFunctionEntry
__asm
{
      mov         eax,  valueReturn
      mov         edx,  dwESPDiff
}

return;
}
```

5.2 Executing code inside another process

You can notice from above that the patching code and hook function should be in target process's address space for them to work. So how do we insert this code in target process's address space and make the target process execute this code? A well known solution to this problem is to implement the code in a DLL and to load the DLL in target process's address space by using documented Win32 API CreateRemoteThread. CreateRemoteThread as the name suggest is used to create a thread in another process. A thread function has same prototype as LoadLibrary, so we can call CreateRemoteThread with LoadLibrary's address and it will cause LoadLibrary call to be executed in target process's context. This way LoadLibrary can load our patching DLL in target process. Once the DLL is loaded, its DllMain will get called with parameter DLL_PROCESS_ATTACH and at that point the DLL can patch all the required IAT entries of target process. We already showed you the code to patch IAT entries; the code to load the DLL is shown below:

```
bool
WINAPI
InjectDll(
      HANDLE            hProcess,
      const wchar_t*    inDllPath)
{
      HMODULE hModule = GetModuleHandleA("kernel32.dll");

      if (hModule == NULL)
      {
            goto funcExit;
      }
```

```
LPVOID loadLibraryW = GetProcAddress(
                                hModule,
                                "LoadLibraryW");

if (loadLibraryW == NULL)
{
      goto funcExit;
}

LPVOID            pInjectionData;
wchar_t           szDllPath[MAX_PATH];
SIZE_T            notUsed;
bool              funcResult = false;

//
// Allocate the memory inside target process
// for DLL full path
//
pInjectionData = VirtualAllocEx(
                                hProcess,
                                NULL,
                                sizeof(szDllPath),
                                MEM_COMMIT,
                                PAGE_READWRITE);

if (pInjectionData == NULL)
{
      goto funcExit;
}

wcscpy(szDllPath, inDllPath);

WriteProcessMemory(
                        hProcess,
                        pInjectionData,
                        szDllPath,
                        sizeof(szDllPath),
                        &notUsed);

DWORD threadId = 0;

//
// The code below will cause following to get executed
// in target process
//
// (*loadLibraryW)(pInjectionData);
//
// This will load our patching DLL in target process
//
HANDLE hThread = CreateRemoteThread(
                        hProcess,
                        0,
                        0,
                        (LPTHREAD_START_ROUTINE)loadLibraryW,
                        pInjectionData,
                        0,
```

```
                                    &threadId);

        if (hThread)
        {
                // Set the return status
                funcResult = true;
        }

funcExit:
        return funcResult;
}
```

5.3 Logging interface

So far we have seen how to patch IAT entries, implementation of our hook function and how to load our patching DLL in target process's address space. The only remaining part of the problem is, how to log the information that our hook function is sending. There can be numerous ways to do this. We can use shared memory to transfer the information, or we can use any IPC mechanism to do the same. We chose a simpler approach, which is to call OutputDebugString from hook function to log information. StraceNT can now act as a debugger and attach to the target process and print the API logging information coming from hook function. This approach is easy to implement and has another advantage that if a real debugger is attached to target process, then the entire API logging information will be displayed in that too. There is a very good sample for writing a minimum debugger in MSDN but for completeness sake, I am listing the code for my implementation below:

```
void
AttachDebugger(
        DWORD processId)
{
HANDLE ghProcess;
int threadCount = 0;
bool processInfected = false;

if (!DebugActiveProcess(processId))
{
        goto funcExit;
}

DebugSetProcessKillOnExit(FALSE);

DEBUG_EVENT debugEvent;
DWORD       dwContinueStatus = DBG_CONTINUE;

bool keepAlive = true;

while(keepAlive)
{
        WaitForDebugEvent(&debugEvent, INFINITE);
        dwContinueStatus = DBG_CONTINUE;

        switch (debugEvent.dwDebugEventCode)
```

```
{
	case EXCEPTION_DEBUG_EVENT:
	{
	switch
	(debugEvent.u.Exception.ExceptionRecord.ExceptionCode)
		{
			case EXCEPTION_BREAKPOINT:
			{
				break;
			}
			default:
			{
				//
				// If this was a second chance exception,
				// it will cause the process to terminate
				//
				dwContinueStatus =
						DBG_EXCEPTION_NOT_HANDLED;
				break;
			}
		}

		break;
	}
	case CREATE_PROCESS_DEBUG_EVENT:
	{
		if (ghProcess == NULL)
		{
			ghProcess =
				debugEvent.u.CreateProcessInfo.hProcess;
		}
		break;
	}
	case EXIT_PROCESS_DEBUG_EVENT:
	{
		keepAlive = false;
		break;
	}
	case OUTPUT_DEBUG_STRING_EVENT:
	{
		DWORD cbRead = 0;

		ReadProcessMemory(
			ghProcess,
			debugEvent.u.DebugString.lpDebugStringData,
			gDbgString,
			debugEvent.u.DebugString.nDebugStringLength,
			&cbRead);

		if (debugEvent.u.DebugString.fUnicode)
		{
			wprintf(L"%ws", gDbgString);
		}
		else
		{
			printf("%s", gDbgString);
		}
```

```
                    break;
                }
            }

            ContinueDebugEvent(debugEvent.dwProcessId,
                               debugEvent.dwThreadId,
                               dwContinueStatus);
}

funcExit:
            return;
}
```

6 Recap

In summary, you need to do following to implement a full blown solution with API hooking techniques:

- Implement a DLL which can patch its process's IAT entries and which also implements a hook function for the patched IAT entries (or functions).
- Implement a module to inject this DLL into address space of target process.
- Implement a process which should first inject the DLL into target process and then attach to the target process as a debugger to log the API information. You may skip attaching to the target process as a debugger and instead use some other IPC mechanism to log the API information.

7 Conclusion

IAT Patching technique for API hooking though not new; is still a lot of fun to play with. It is a fun thing to learn and once you develop a solution using this technique, you will sure find that you have a better understanding of many concepts in windows. There has been many implementations using this technique, but none to my knowledge provide as flexible API hooking as provided by StraceNT. Its ability to hook any API with any calling convention makes it unique and powerful to trace APIs from any DLL.

8 Reference

1. http://www.internals.com/articles/apispy/apispy.htm
2. http://www.iecc.com/linker/linker07.html
3. Learn System Level Win32 Coding Techniques by Writing an API Spy Program by Matt Pietrek published in December 1994 issue of MSJ.
4. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndebug/html/msdn_peeringpe.asp
5. http://weblogs.asp.net/oldnewthing/archive/2004/01/08/48616.aspx
6. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccelng/htm/msmod_25.asp