CHAPTER 5

Monitoring Native API Calls

Intercepting operating system calls is an all-time favorite of programmers everywhere. The motivations for this public interest are numerous: code profiling and optimization, reverse engineering, user activity logging, and the like. All of these share a common intention: to pass control to a special piece of code whenever an application calls a system service, making it possible to find out which service was called, what parameters it received, what results it returned, and how long it took to execute. Based on a technique originally proposed by Mark Russinovich and Bryce Cogswell (Russinovich and Cogswell 1997), this chapter presents a general framework for implanting hooks into arbitrary Native API functions. The approach used here is completely data-driven, so it can be easily extended and adapted to other Windows 2000/NT versions. The data gathered from the API calls of all processes in the system are written to a circular buffer that can be read by a client application via device I/O control. The protocol data are formatted as a simple line-oriented ANSI text stream that obeys strict formatting rules, making automated postprocessing by an application easy. To demonstrate the basic outline of such a client application, this chapter also presents a sample protocol data viewer running in a console window.

PATCHING THE SERVICE DESCRIPTOR TABLE

Whereas "primitive" operating systems such as DOS or Windows 3.xx offered little resistance to programmers who wanted to apply hooks to their Application Programming Interfaces (APIs), Win32 systems such as Windows 2000, Windows NT, and Windows 9x are much harder to handle, because they use clever protection mechanisms to separate unrelated pieces of code from each other. Setting a system-wide hook on a Win32 API is not a small task. Fortunately, we have Win32 wizards such as Matt Pietrek (Pietrek 1996e) and Jeffrey Richter (Richter 1997), who have put much work into showing us how it can be done, despite the fact that there's no

simple and elegant solution. In 1997, Russinovich and Cogswell presented a completely different approach to system-wide hooks for Windows NT, intercepting the system at a much lower level (Russinovich and Cogswell 1997). They proposed to inject the logging mechanism into the Native API dispatcher, just below the frontier between user-mode and kernel-mode, where Windows NT exposes a "bottleneck" that all user-mode threads must pass through to be serviced by the operating system kernel.

Service and Argument Tables

As discussed in Chapter 2, the doorway through which all Native API calls originating in user-mode must pass is the `INT 2Eh` interface that provides an i386 interrupt gate for the privilege level change. You might recall as well that all `INT 2Eh` calls are handled in kernel-mode by the internal function `KiSystemService()`, which uses the system's Service Descriptor Table (SDT) to look up the entry points of the Native API handlers. In Figure 5-1, the interrelations of the basic components of this dispatching mechanism are outlined. The formal definitions of the `SERVICE_DESCRIPTOR_TABLE` structure and its subtypes from Chapter 2 (Listing 2-1) are repeated in Listing 5-1.

`KiSystemService()` is called with two arguments, passed in by the `INT 2Eh` caller in the CPU registers `EAX` and `EDX`. `EAX` contains a zero-based index into an array of API handler function pointers, and `EDX` points to the caller's argument stack. `KiSystemService()` retrieves the base address of the function array by reading the value of the `ServiceTable` member of a public `ntoskrnl.exe` data structure named `KeServiceDescriptorTable`, shown on the left-hand side of Figure 5-1. Actually, `KeServiceDescriptorTable` points to an array of four service table parameter structures, but only the first one contains valid entries by default. `KiSystemService()` looks up the address of the function that should handle the API call by using `EAX` as an index into the internal `KiServiceTable` structure. Before calling the target function, `KiSystemService()` queries the `KiArgumentTable` structure in much the same way to find out how many bytes were passed in by the caller on the argument stack, and uses this value to copy the arguments to the current kernel-mode stack. After that, a simple assembly language `CALL` instruction is required to invoke the API handler. Everything is then set up as a normal `__stdcall` C function would expect.

Windows 2000 provides another service descriptor table parameter block named `KeServiceDescriptorTableShadow`. Whereas `KeServiceDescriptorTable` is publicly exported by `ntoskrnl.exe` so kernel-mode drivers can readily access it, `KeServiceDescriptorTableShadow` is not. On Windows 2000, `KeServiceDescriptorTableShadow` follows immediately after `KeServiceDescriptorTable`, but you should not count on that—this rule does not hold on Windows NT 4.0, and it is possible that it won't hold on future updates of Windows 2000. The difference between both parameter blocks is that in `KeServiceDescriptorTableShadow` the second slot

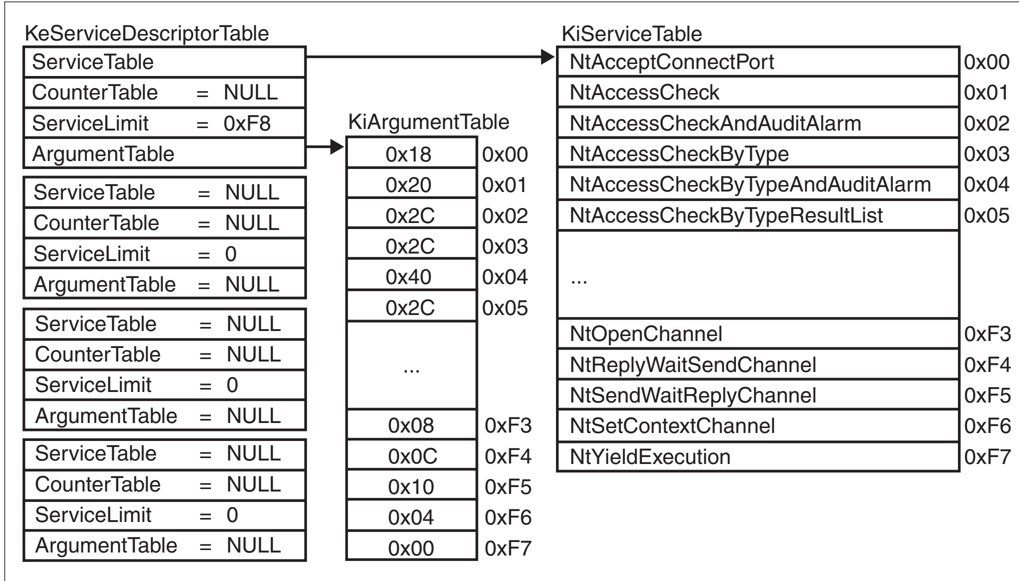

FIGURE 5-1. *Structure of the* `KeServiceDescriptorTable`

```
typedef NTSTATUS (NTAPI *NTPROC) ();
typedef NTPROC *PNTPROC;
#define NTPROC_ sizeof (NTPROC)

// -----------------------------------------------------------------

typedef struct _SYSTEM_SERVICE_TABLE
    {
    PNTPROC ServiceTable;           // array of entry points
    PDWORD  CounterTable;           // array of usage counters
    DWORD   ServiceLimit;           // number of table entries
    PBYTE   ArgumentTable;          // array of byte counts
    }
        SYSTEM_SERVICE_TABLE,
     * PSYSTEM_SERVICE_TABLE,
    **PPSYSTEM_SERVICE_TABLE;

// -----------------------------------------------------------------

typedef struct _SERVICE_DESCRIPTOR_TABLE
    {
    SYSTEM_SERVICE_TABLE ntoskrnl;  // ntoskrnl.exe (native api)
```

(continued)

```
SYSTEM_SERVICE_TABLE win32k;     // win32k.sys (gdi/user support)
SYSTEM_SERVICE_TABLE Table3;     // not used
SYSTEM_SERVICE_TABLE Table4;     // not used
}
    SERVICE_DESCRIPTOR_TABLE,
 * PSERVICE_DESCRIPTOR_TABLE,
**PPSERVICE_DESCRIPTOR_TABLE;
```

LISTING 5-1. *Definition of the* `SERVICE_DESCRIPTOR_TABLE` *Structure*

is used by the system, too. It contains references to the internal `w32pServiceTable` and `w32pArgumentTable` structures that are used by the Win32 kernel-mode component `win32k.sys` to dispatch its own API calls, as shown in Figure 5-2. `KiSystemService()` knows that it is handling a `win32k.sys` API call by examining bits #12 and 13 of the function index in register `EAX`. If both bits are zero, it is a Native API call handled by `ntoskrnl.exe`, so `KiSystemService()` uses the first SDT slot. If bit #12 is set and bit #13 is zero, `KiSystemService()` uses the second slot. The remaining two bit combinations are assigned to the last pair of slots, which are currently not used by the system. This means that the index numbers of Native API calls potentially range from `0x0000` to `0x0FFF`, and `win32k.sys` calls involve index numbers in the range `0x1000` to `0x1FFF`. Consequently, the ranges `0x2000` to `0x2FFF` and `0x3000` to `0x3FFF` are assigned to the reserved tables. On Windows 2000, the Native API service table contains 248 entries and the `win32k.sys` table contains 639 entries.

The ingenious idea of Russinovich and Cogswell was to hook API calls by simply putting a different handler into the `KiServiceTable` array. This handler would ultimately call the original handler inside `ntoskrnl.exe`, but it had the opportunity to take a peek at the input and output parameters of the called function. This approach is extremely powerful but also very simple. Because all user-mode threads have to pass through this needle's eye in order to get their Native API requests serviced, a simple exchange of function pointers installs a global hook that continues to work reliably even after new processes and threads have been started. There is no need for a notification mechanism that signals the addition or removal of processes and threads.

Unfortunately, the system service pointer tables are subject to nontrivial changes across Windows NT versions. Table 5-1 compares the `KiServiceTable` entries of Windows 2000 and Windows NT 4.0. It is obvious that not only has the number of handlers been increased from 211 to 248 but the new handlers haven't been appended to the end of the list. They were inserted somewhere in between! Thus, a service function index of, say, `0x20` refers to `NtCreateFile()` on Windows 2000 but is associated with `NtCreateProfile()` on Windows NT 4.0. Consequently,

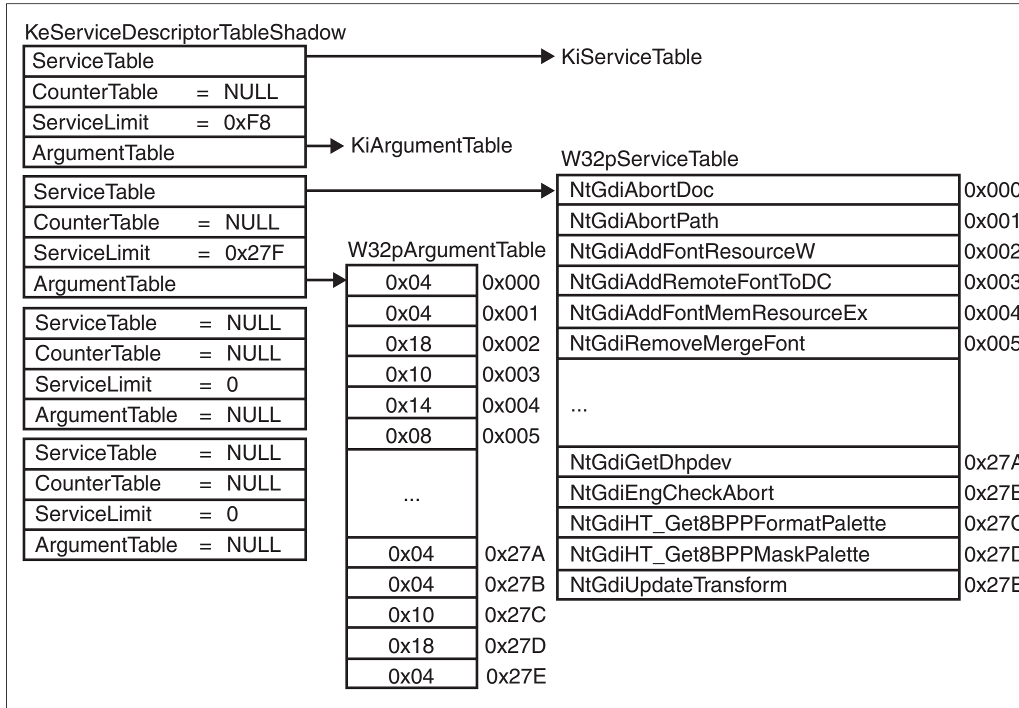

FIGURE 5-2. *Structure of* `KeServiceDescriptorTableShadow`

an API call monitor that installs a hook by manipulating the entries in the service function table must carefully check the Windows NT version it is running on. This can be done in several ways:

- One possibility is to check the public `NtBuildNumber` variable exported by `ntoskrnl.exe`, as Russinovich and Cogswell did in their original article (Russinovich and Cogswell 1997). Windows NT 4.0 exposes a build number of 1,381 for all service packs. The build number of Windows 2000 is currently 2,195. Hopefully, this number will remain as stable as it did in the previous Windows NT versions.
- Another possibility is to check the `NtMajorVersion` and `NtMinorVersion` members of the `SharedUserData` structure defined in the Windows 2000 header file `ntddk.h`. All Windows NT 4.0 service packs set `SharedUserData->NtMajorVersion` to four and `SharedUserData->NtMinorVersion` to zero. Windows 2000 currently indicates a Windows NT version of 5.0.

- The code presented in this chapter uses yet another alternative—it tests whether the `ServiceLimit` member of the SDT entry matches its expectations, which is 211 (`0xD3`) for Windows NT 4.0 and 248 (`0xF8`) for Windows 2000.

TABLE 5-1. *Windows 2000 and NT 4.0 Service Table Comparison*

WINDOWS 2000	INDEX	WINDOWS NT 4.0
NtAcceptConnectPort	0x00	NtAcceptConnectPort
NtAccessCheck	0x01	NtAccessCheck
NtAccessCheckAndAuditAlarm	0x02	NtAccessCheckAndAuditAlarm
NtAccessCheckByType	0x03	NtAddAtom
NtAccessCheckByTypeAndAuditAlarm	0x04	NtAdjustGroupsToken
NtAccessCheckByTypeResultList	0x05	NtAdjustPrivilegesToken
NtAccessCheckByTypeResultListAndAuditAlarm	0x06	NtAlertResumeThread
NtAccessCheckByTypeResultListAndAuditAlarmByHandle	0x07	NtAlertThread
NtAddAtom	0x08	NtAllocateLocallyUniqueId
NtAdjustGroupsToken	0x09	NtAllocateUuids
NtAdjustPrivilegesToken	0x0A	NtAllocateVirtualMemory
NtAlertResumeThread	0x0B	NtCallbackReturn
NtAlertThread	0x0C	NtCancelIoFile
NtAllocateLocallyUniqueId	0x0D	NtCancelTimer
NtAllocateUserPhysicalPages	0x0E	NtClearEvent
NtAllocateUuids	0x0F	NtClose
NtAllocateVirtualMemory	0x10	NtCloseObjectAuditAlarm
NtAreMappedFilesTheSame	0x11	NtCompleteConnectPort
NtAssignProcessToJobObject	0x12	NtConnectPort
NtCallbackReturn	0x13	NtContinue
NtCancelIoFile	0x14	NtCreateDirectoryObject
NtCancelTimer	0x15	NtCreateEvent
NtCancelDeviceWakeupRequest	0x16	NtCreateEventPair
NtClearEvent	0x17	NtCreateFile
NtClose	0x18	NtCreateIoCompletion
NtCloseObjectAuditAlarm	0x19	NtCreateKey
NtCompleteConnectPort	0x1A	NtCreateMailslotFile
NtConnectPort	0x1B	NtCreateMutant
NtContinue	0x1C	NtCreateNamedPipeFile
NtCreateDirectoryObject	0x1D	NtCreatePagingFile
NtCreateEvent	0x1E	NtCreatePort
NtCreateEventPair	0x1F	NtCreateProcess
NtCreateFile	0x20	NtCreateProfile
NtCreateIoCompletion	0x21	NtCreateSection

TABLE 5-1. *(continued)*

WINDOWS 2000	INDEX	WINDOWS NT 4.0
NtCreateJobObject	0x22	NtCreateSemaphore
NtCreateKey	0x23	NtCreateSymbolicLinkObject
NtCreateMailslotFile	0x24	NtCreateThread
NtCreateMutant	0x25	NtCreateTimer
NtCreateNamedPipeFile	0x26	NtCreateToken
NtCreatePagingFile	0x27	NtDelayExecution
NtCreatePort	0x28	NtDeleteAtom
NtCreateProcess	0x29	NtDeleteFile
NtCreateProfile	0x2A	NtDeleteKey
NtCreateSection	0x2B	NtDeleteObjectAuditAlarm
NtCreateSemaphore	0x2C	NtDeleteValueKey
NtCreateSymbolicLinkObject	0x2D	NtDeviceIoControlFile
NtCreateThread	0x2E	NtDisplayString
NtCreateTimer	0x2F	NtDuplicateObject
NtCreateToken	0x30	NtDuplicateToken
NtCreateWaitablePort	0x31	NtEnumerateKey
NtDelayExecution	0x32	NtEnumerateValueKey
NtDeleteAtom	0x33	NtExtendSection
NtDeleteFile	0x34	NtFindAtom
NtDeleteKey	0x35	NtFlushBuffersFile
NtDeleteObjectAuditAlarm	0x36	NtFlushInstructionCache
NtDeleteValueKey	0x37	NtFlushKey
NtDeviceIoControlFile	0x38	NtFlushVirtualMemory
NtDisplayString	0x39	NtFlushWriteBuffer
NtDuplicateObject	0x3A	NtFreeVirtualMemory
NtDuplicateToken	0x3B	NtFsControlFile
NtEnumerateKey	0x3C	NtGetContextThread
NtEnumerateValueKey	0x3D	NtGetPlugPlayEvent
NtExtendSection	0x3E	NtGetTickCount
NtFilterToken	0x3F	NtImpersonateClientOfPort
NtFindAtom	0x40	NtImpersonateThread
NtFlushBuffersFile	0x41	NtInitializeRegistry
NtFlushInstructionCache	0x42	NtListenPort
NtFlushKey	0x43	NtLoadDriver
NtFlushVirtualMemory	0x44	NtLoadKey
NtFlushWriteBuffer	0x45	NtLoadKey2
NtFreeUserPhysicalPages	0x46	NtLockFile
NtFreeVirtualMemory	0x47	NtLockVirtualMemory
NtFsControlFile	0x48	NtMakeTemporaryObject

(continued)

TABLE 5-1. *(continued)*

WINDOWS 2000	INDEX	WINDOWS NT 4.0
NtGetContextThread	0x49	NtMapViewOfSection
NtGetDevicePowerState	0x4A	NtNotifyChangeDirectoryFile
NtGetPlugPlayEvent	0x4B	NtNotifyChangeKey
NtGetTickCount	0x4C	NtOpenDirectoryObject
NtGetWriteWatch	0x4D	NtOpenEvent
NtImpersonateAnonymousToken	0x4E	NtOpenEventPair
NtImpersonateClientOfPort	0x4F	NtOpenFile
NtImpersonateThread	0x50	NtOpenIoCompletion
NtInitializeRegistry	0x51	NtOpenKey
NtInitiatePowerAction	0x52	NtOpenMutant
NtIsSystemResumeAutomatic	0x53	NtOpenObjectAuditAlarm
NtListenPort	0x54	NtOpenProcess
NtLoadDriver	0x55	NtOpenProcessToken
NtLoadKey	0x56	NtOpenSection
NtLoadKey2	0x57	NtOpenSemaphore
NtLockFile	0x58	NtOpenSymbolicLinkObject
NtLockVirtualMemory	0x59	NtOpenThread
NtMakeTemporaryObject	0x5A	NtOpenThreadToken
NtMapUserPhysicalPages	0x5B	NtOpenTimer
NtMapUserPhysicalPagesScatter	0x5C	NtPlugPlayControl
NtMapViewOfSection	0x5D	NtPrivilegeCheck
NtNotifyChangeDirectoryFile	0x5E	NtPrivilegedServiceAuditAlarm
NtNotifyChangeKey	0x5F	NtPrivilegeObjectAuditAlarm
NtNotifyChangeMultipleKeys	0x60	NtProtectVirtualMemory
NtOpenDirectoryObject	0x61	NtPulseEvent
NtOpenEvent	0x62	NtQueryInformationAtom
NtOpenEventPair	0x63	NtQueryAttributesFile
NtOpenFile	0x64	NtQueryDefaultLocale
NtOpenIoCompletion	0x65	NtQueryDirectoryFile
NtOpenJobObject	0x66	NtQueryDirectoryObject
NtOpenKey	0x67	NtQueryEaFile
NtOpenMutant	0x68	NtQueryEvent
NtOpenObjectAuditAlarm	0x69	NtQueryFullAttributesFile
NtOpenProcess	0x6A	NtQueryInformationFile
NtOpenProcessToken	0x6B	NtQueryIoCompletion
NtOpenSection	0x6C	NtQueryInformationPort
NtOpenSemaphore	0x6D	NtQueryInformationProcess
NtOpenSymbolicLinkObject	0x6E	NtQueryInformationThread

TABLE 5-1. *(continued)*

WINDOWS 2000	INDEX	WINDOWS NT 4.0
NtOpenThread	0x6F	NtQueryInformationToken
NtOpenThreadToken	0x70	NtQueryIntervalProfile
NtOpenTimer	0x71	NtQueryKey
NtPlugPlayControl	0x72	NtQueryMultipleValueKey
NtPowerInformation	0x73	NtQueryMutant
NtPrivilegeCheck	0x74	NtQueryObject
NtPrivilegedServiceAuditAlarm	0x75	NtQueryOleDirectoryFile
NtPrivilegeObjectAuditAlarm	0x76	NtQueryPerformanceCounter
NtProtectVirtualMemory	0x77	NtQuerySection
NtPulseEvent	0x78	NtQuerySecurityObject
NtQueryInformationAtom	0x79	NtQuerySemaphore
NtQueryAttributesFile	0x7A	NtQuerySymbolicLinkObject
NtQueryDefaultLocale	0x7B	NtQuerySystemEnvironmentValue
NtQueryDefaultUILanguage	0x7C	NtQuerySystemInformation
NtQueryDirectoryFile	0x7D	NtQuerySystemTime
NtQueryDirectoryObject	0x7E	NtQueryTimer
NtQueryEaFile	0x7F	NtQueryTimerResolution
NtQueryEvent	0x80	NtQueryValueKey
NtQueryFullAttributesFile	0x81	NtQueryVirtualMemory
NtQueryInformationFile	0x82	NtQueryVolumeInformationFile
NtQueryInformationJobObject	0x83	NtQueueApcThread
NtQueryIoCompletion	0x84	NtRaiseException
NtQueryInformationPort	0x85	NtRaiseHardError
NtQueryInformationProcess	0x86	NtReadFile
NtQueryInformationThread	0x87	NtReadFileScatter
NtQueryInformationToken	0x88	NtReadRequestData
NtQueryInstallUILanguage	0x89	NtReadVirtualMemory
NtQueryIntervalProfile	0x8A	NtRegisterThreadTerminatePort
NtQueryKey	0x8B	NtReleaseMutant
NtQueryMultipleValueKey	0x8C	NtReleaseSemaphore
NtQueryMutant	0x8D	NtRemoveIoCompletion
NtQueryObject	0x8E	NtReplaceKey
NtQueryOpenSubKeys	0x8F	NtReplyPort
NtQueryPerformanceCounter	0x90	NtReplyWaitReceivePort
NtQueryQuotaInformationFile	0x91	NtReplyWaitReplyPort
NtQuerySection	0x92	NtRequestPort
NtQuerySecurityObject	0x93	NtRequestWaitReplyPort
NtQuerySemaphore	0x94	NtResetEvent

(continued)

TABLE 5-1. *(continued)*

WINDOWS 2000	INDEX	WINDOWS NT 4.0
NtQuerySymbolicLinkObject	0x95	NtRestoreKey
NtQuerySystemEnvironmentValue	0x96	NtResumeThread
NtQuerySystemInformation	0x97	NtSaveKey
NtQuerySystemTime	0x98	NtSetIoCompletion
NtQueryTimer	0x99	NtSetContextThread
NtQueryTimerResolution	0x9A	NtSetDefaultHardErrorPort
NtQueryValueKey	0x9B	NtSetDefaultLocale
NtQueryVirtualMemory	0x9C	NtSetEaFile
NtQueryVolumeInformationFile	0x9D	NtSetEvent
NtQueueApcThread	0x9E	NtSetHighEventPair
NtRaiseException	0x9F	NtSetHighWaitLowEventPair
NtRaiseHardError	0xA0	NtSetHighWaitLowThread
NtReadFile	0xA1	NtSetInformationFile
NtReadFileScatter	0xA2	NtSetInformationKey
NtReadRequestData	0xA3	NtSetInformationObject
NtReadVirtualMemory	0xA4	NtSetInformationProcess
NtRegisterThreadTerminatePort	0xA5	NtSetInformationThread
NtReleaseMutant	0xA6	NtSetInformationToken
NtReleaseSemaphore	0xA7	NtSetIntervalProfile
NtRemoveIoCompletion	0xA8	NtSetLdtEntries
NtReplaceKey	0xA9	NtSetLowEventPair
NtReplyPort	0xAA	NtSetLowWaitHighEventPair
NtReplyWaitReceivePort	0xAB	NtSetLowWaitHighThread
NtReplyWaitReceivePortEx	0xAC	NtSetSecurityObject
NtReplyWaitReplyPort	0xAD	NtSetSystemEnvironmentValue
NtRequestDeviceWakeup	0xAE	NtSetSystemInformation
NtRequestPort	0xAF	NtSetSystemPowerState
NtRequestWaitReplyPort	0xB0	NtSetSystemTime
NtRequestWakeupLatency	0xB1	NtSetTimer
NtResetEvent	0xB2	NtSetTimerResolution
NtResetWriteWatch	0xB3	NtSetValueKey
NtRestoreKey	0xB4	NtSetVolumeInformationFile
NtResumeThread	0xB5	NtShutdownSystem
NtSaveKey	0xB6	NtSignalAndWaitForSingleObject
NtSaveMergedKeys	0xB7	NtStartProfile
NtSecureConnectPort	0xB8	NtStopProfile
NtSetIoCompletion	0xB9	NtSuspendThread
NtSetContextThread	0xBA	NtSystemDebugControl
NtSetDefaultHardErrorPort	0xBB	NtTerminateProcess

TABLE 5-1. *(continued)*

WINDOWS 2000	INDEX	WINDOWS NT 4.0
NtSetDefaultLocale	0xBC	NtTerminateThread
NtSetDefaultUILanguage	0xBD	NtTestAlert
NtSetEaFile	0xBE	NtUnloadDriver
NtSetEvent	0xBF	NtUnloadKey
NtSetHighEventPair	0xC0	NtUnlockFile
NtSetHighWaitLowEventPair	0xC1	NtUnlockVirtualMemory
NtSetInformationFile	0xC2	NtUnmapViewOfSection
NtSetInformationJobObject	0xC3	NtVdmControl
NtSetInformationKey	0xC4	NtWaitForMultipleObjects
NtSetInformationObject	0xC5	NtWaitForSingleObject
NtSetInformationProcess	0xC6	NtWaitHighEventPair
NtSetInformationThread	0xC7	NtWaitLowEventPair
NtSetInformationToken	0xC8	NtWriteFile
NtSetIntervalProfile	0xC9	NtWriteFileGather
NtSetLdtEntries	0xCA	NtWriteRequestData
NtSetLowEventPair	0xCB	NtWriteVirtualMemory
NtSetLowWaitHighEventPair	0xCC	NtCreateChannel
NtSetQuotaInformationFile	0xCD	NtListenChannel
NtSetSecurityObject	0xCE	NtOpenChannel
NtSetSystemEnvironmentValue	0xCF	NtReplyWaitSendChannel
NtSetSystemInformation	0xD0	NtSendWaitReplyChannel
NtSetSystemPowerState	0xD1	NtSetContextChannel
NtSetSystemTime	0xD2	NtYieldExecution
NtSetThreadExecutionState	0xD3	N/A
NtSetTimer	0xD4	N/A
NtSetTimerResolution	0xD5	N/A
NtSetUuidSeed	0xD6	N/A
NtSetValueKey	0xD7	N/A
NtSetVolumeInformationFile	0xD8	N/A
NtShutdownSystem	0xD9	N/A
NtSignalAndWaitForSingleObject	0xDA	N/A
NtStartProfile	0xDB	N/A
NtStopProfile	0xDC	N/A
NtSuspendThread	0xDD	N/A
NtSystemDebugControl	0xDE	N/A
NtTerminateJobObject	0xDF	N/A
NtTerminateProcess	0xE0	N/A
NtTerminateThread	0xE1	N/A
NtTestAlert	0xE2	N/A

(continued)

TABLE 5-1. *(continued)*

WINDOWS 2000	INDEX	WINDOWS NT 4.0
NtUnloadDriver	0xE3	N/A
NtUnloadKey	0xE4	N/A
NtUnlockFile	0xE5	N/A
NtUnlockVirtualMemory	0xE6	N/A
NtUnmapViewOfSection	0xE7	N/A
NtVdmControl	0xE8	N/A
NtWaitForMultipleObjects	0xE9	N/A
NtWaitForSingleObject	0xEA	N/A
NtWaitHighEventPair	0xEB	N/A
NtWaitLowEventPair	0xEC	N/A
NtWriteFile	0xED	N/A
NtWriteFileGather	0xEE	N/A
NtWriteRequestData	0xEF	N/A
NtWriteVirtualMemory	0xF0	N/A
NtCreateChannel	0xF1	N/A
NtListenChannel	0xF2	N/A
NtOpenChannel	0xF3	N/A
NtReplyWaitSendChannel	0xF4	N/A
NtSendWaitReplyChannel	0xF5	N/A
NtSetContextChannel	0xF6	N/A
NtYieldExecution	0xF7	N/A

The most important step taken by Russinovich and Cogswell was to write a kernel-mode device driver that installs and maintains the Native API hooks, because user-mode modules do not have the appropriate privileges to modify the system at this low system level. Like the spy driver in Chapter 4, this is a somewhat unusual driver, because it does not perform the usual I/O request processing. It just exposes a simple Device I/O Control (IOCTL) interface to give user-mode code access to the data it collects. The main task of this driver is to manipulate the `KiServiceTable` and intercept and log selected calls to the Windows 2000 Native API. Although this method is simple and elegant, it is also somewhat alarming. Its simplicity reminds me of the old DOS days when hooking a system service was as simple as modifying a pointer in the processor's interrupt vector table. Anyone who knows how to write a basic Windows 2000 kernel-mode driver can hook any NT system service without much effort.

Russinovich and Cogswell used their technique to develop a very useful Windows NT registry monitor. While adapting their code for other spying tasks, I quickly became annoyed by the requirement of writing an individual hook function for each API function on which I wanted to spy. To avoid having to write extensive stereotypic code, I

wanted to find a way to force all API functions I was interested in through a single hook function. This turned out to be a task that took considerable time and showed me all possible variants of Blue Screens. However, this resulted in a general-purpose solution that enabled me to vary the set of hooked API functions with minimum effort.

ASSEMBLY LANGUAGE TO THE RESCUE

The main obstacle to a general-purpose solution was the typical parameter passing mechanism of the C language. As you may know, C usually passes function arguments on the CPU stack before calling the function's entry point. Depending on the number of arguments a function requires, the size of the argument stack varies considerably. The 248 Native API functions of Windows 2000 involve argument stack sizes between zero and 68 bytes. Given the diligent type checking of C, this makes writing a unique hook function a tough job. Microsoft Visual C/C++ comes with a versatile integrated assembly language (ASM) compiler that is capable of processing moderately complex code. Ironically, the advantage of ASM in this situation is exactly what is commonly regarded as one of its biggest drawbacks: ASM doesn't provide a strict type checking mechanism. As long as the number of bits is OK, you can store almost anything in any register and you can call any address without concern for what is currently on the stack. Although this is a dangerous feature in application programming, it comes in quite handy here: In ASM, it is easy to call a common entry point with different arguments on the stack, and this feature will be exploited in the API hook dispatcher introduced in a moment.

The Microsoft Visual C/C++ inline assembler is invoked by putting ASM code into delimited blocks tagged by the keyword `__asm`. It lacks the macro definition and evaluation capabilities of Microsoft's big Macro Assembler (MASM), but this doesn't severely restrict its usefulness. The best feature of the inline assembler is that it has access to all C variables and type definitions, so it is quite easy to mix C and ASM code. However, when ASM code is included in a C function, some important basic conventions of the C compiler must be obeyed to avoid interference with the compiled C code:

- The caller of a C function assumes that the CPU registers `EBP`, `EBX`, `ESI`, and `EDI` are preserved.
- If the ASM code is mixed with C code in a single function, be careful to preserve all intermediate values the C code might hold in registers. It is always a good idea to save and restore all registers used inside an `__asm` clause.
- 8-bit function results (`CHAR`, `BYTE`, etc.) are returned in register `AL`.
- 16-bit function results (`SHORT`, `WORD`, etc.) are returned in register `AX`.

- 32-bit function results (INT, LONG, DWORD, pointers, etc.) are returned in register EAX.
- 64-bit function results (__int64, LONGLONG, DWORDLONG, etc.) are returned in register pair EDX:EAX. Register EAX contains bits #0 to 31, and EDX holds bits #32 to 63.
- Functions with a fixed number of arguments usually pass arguments according to the __stdcall convention. From the caller's perspective, this means that the arguments must be pushed onto the stack in reverse order before the call, and the callee is responsible for removing them from the stack before returning. From the perspective of the called function, this means that the stack pointer ESP points to the caller's return address, followed by the arguments in their original order. The original order is retained because the stack grows downward, from high linear addresses to lower ones. Therefore, the argument pushed last by the caller (i.e., argument #1) appears as the first argument in the array pointed to by ESP.
- Some API functions with fixed arguments, most notably the C Runtime Library functions exported by ntdll.dll and ntoskrnl.exe, traditionally employ the __cdecl calling convention, which involves the same argument ordering as __stdcall, but forces the caller to clean up the argument stack.
- Functions with a variable number of arguments are always of the __cdecl type, because only the caller knows exactly how many arguments were passed to the callee. Therefore, the responsibility of removing the arguments from the stack is left to the caller.
- Functions declared with the __fastcall modifier expect the first two arguments in the CPU registers ECX and EDX. If more arguments are required, they are passed in on the stack in reverse order, and the callee cleans up the stack, as in the __stdcall scheme.
- Many C compilers build a stack frame for the function arguments immediately after entering the function, using the CPU's base pointer register EBP. This code, shown in Listing 5-2, is frequently referred to as a function's "prologue" and "epilogue." Some compilers use the more elegant i386 ENTER and LEAVE operations that integrate this EBP/ESP shuffling into single instructions (cf. Intel 1999b). After the prologue has

```
SomeFunction:
```

```
; this is the function's prologue
push    ebp                   ; save current value of ebp
mov     ebp, esp              ; set stack frame base address
sub     esp, SizeOfLocalStorage ; create local storage area
...
; this is the function's epilogue
mov     esp, ebp              ; destroy local storage area
pop     ebp                   ; restore value of ebp
ret
```

LISTING 5-2. *Stack frame, prologue, and epilogue*

been executed, the stack appears as shown in Figure 5-3. The value of the `EBP` register is the unique point of reference that splits the function's parameter stack into (1) the local storage area containing all local variables defined within the scope of the function and (2) the caller's argument stack, including the `EBP` backup slot and the return address. Note that the latest versions of Microsoft Visual C/C++ don't use stack frames by default. Instead, the code accesses the values on the stack through register `ESP`, specifying the offset of the variable relative to the current top of the stack. Code of this kind is extremely difficult to read, because each `PUSH` and `POP` instruction affects the `ESP` value and, consequently, all parameter offsets. Because `EBP` isn't required in this scenario, it is used as an additional general-purpose register.

- Be extremely careful when accessing C variables. One of the most frequent inline ASM bugs is that you are loading the address of a variable to a register instead of its value, and vice versa. In case of potential ambiguity, use the `ptr` and `offset` address operators. For example, the instruction `mov eax, dword ptr SomeVariable` loads the `DWORD`-type value of `SomeVariable` to register `EAX`, whereas `mov eax, offset SomeVariable` loads its linear address (i.e., a pointer to its value) to `EAX`.

THE HOOK DISPATCHER

The code that follows is extremely difficult. It took many hours to write, and produced an incredible number of Blue Screens in the process. My original approach involved a separate module, written in native ASM language and assembled with Microsoft's MASM. However, this design created problems on the linker level, so I changed to inline ASM inserted into the main C module. Instead of creating another kernel-mode driver, I decided to integrate the hook code into the spy device

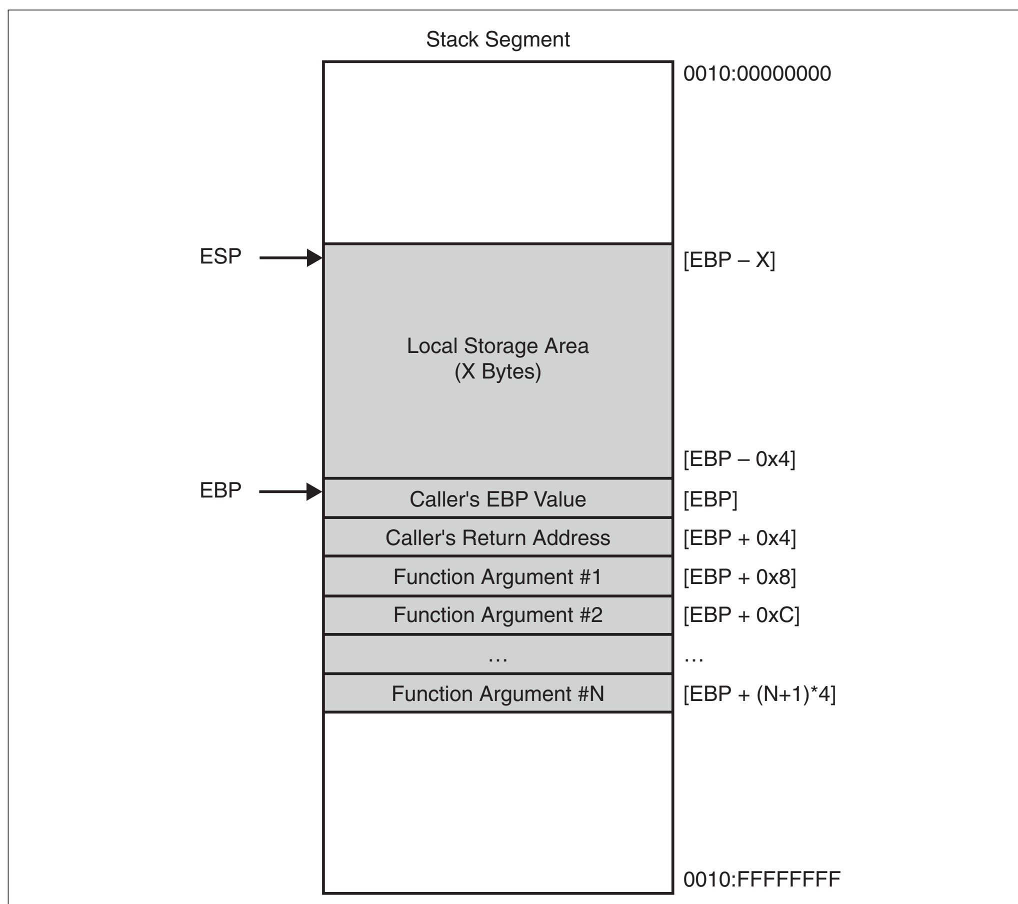

FIGURE 5-3. *Typical Layout of a Stack Frame*

introduced in Chapter 4. Remember the `SPY_IO_HOOK_*` IOCTL functions listed at the bottom of Table 4-2? Now is the time to take a closer look at them. The next section of sample code is taken from the source files `w2k_spy.c` and `w2k_spy.h`, found on the CD accompanying this book, in the `\src\w2k_spy` directory.

In Listing 5-3, the core parts of the Native API hook mechanism are shown. The listing starts with a couple of constant and structure definitions referenced by the code and is followed by the definition of the array `aSpyHooks[]`. Following this array is a macro that evaluates to three important lines of inline assembly language that will be investigated in a moment. The last part of Listing 5-3 is made up of the function `SpyHookInitializeEx()`. On first sight, it is difficult to grasp what this function is supposed to do. This function is a combination of two functions:

1. The "outer" part of `SpyHookInitializeEx()` consists of C code that simply populates the `aSpyHooks[]` array with pointers to the spy device's hook functions and their associated protocol format strings. This function is split in two sections. The first ends inside the first `__asm` clause at the `jmp SpyHook9` instruction. It is obvious that the second section must start at an ASM label named `SpyHook9`, which can be found near the end of the second `__asm` block.
2. The "inner" part of `SpyHookInitializeEx()` comprises everything between the two C sections of the code. It starts with an extensive repetition of `SpyHook` macro invocations and is followed by a large and complex ASM code section. As you may have guessed, this code is the common hook handler mentioned earlier.

```
#define SPY_CALLS           0x00000100 // max api call nesting level
#define SDT_SYMBOLS_NT4     0xD3
#define SDT_SYMBOLS_NT5     0xF8
#define SDT_SYMBOLS_MAX     SDT_SYMBOLS_NT5

// -----------------------------------------------------------------

typedef struct _SPY_HOOK_ENTRY
    {
    NTPROC Handler;
    PBYTE  pbFormat;
    }
    SPY_HOOK_ENTRY, *PSPY_HOOK_ENTRY, **PPSPY_HOOK_ENTRY;

#define SPY_HOOK_ENTRY_ sizeof (SPY_HOOK_ENTRY)

// -----------------------------------------------------------------

typedef struct _SPY_CALL
    {
    BOOL            fInUse;               // set if used entry
    HANDLE          hThread;              // id of calling thread
    PSPY_HOOK_ENTRY pshe;                 // associated hook entry
    PVOID           pCaller;              // caller's return address
    DWORD           dParameters;          // number of parameters
    DWORD           adParameters [1+256]; // result and parameters
    }
    SPY_CALL, *PSPY_CALL, **PPSPY_CALL;

#define SPY_CALL_ sizeof (SPY_CALL)

// -----------------------------------------------------------------
```

(continued)

```
SPY_HOOK_ENTRY aSpyHooks [SDT_SYMBOLS_MAX];

// -----------------------------------------------------------------
// The SpyHook macro defines a hook entry point in inline assembly
// language. The common entry point SpyHook2 is entered by a call
// instruction, allowing the hook to be identified by its return
// address on the stack. The call is executed through a register to
// remove any degrees of freedom from the encoding of the call.

#define SpyHook                              \
        __asm   push    eax                  \
        __asm   mov     eax, offset SpyHook2 \
        __asm   call    eax

// -----------------------------------------------------------------
// The SpyHookInitializeEx() function initializes the aSpyHooks[]
// array with the hook entry points and format strings. It also
// hosts the hook entry points and the hook dispatcher.

void SpyHookInitializeEx (PPBYTE ppbSymbols,
                          PPBYTE ppbFormats)
    {
    DWORD dHooks1, dHooks2, i, j, n;

    __asm
        {
        jmp     SpyHook9
        ALIGN   8
SpyHook1:       ; start of hook entry point section
        }

// the number of entry points defined in this section
// must be equal to SDT_SYMBOLS_MAX (i.e. 0xF8)

SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //08
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //10
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //18
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //20
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //28
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //30
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //38
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //40
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //48
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //50
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //58
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //60
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //68
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //70
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //78
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //80
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //88
```

```
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //90
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //98
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //A0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //A8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //B0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //B8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //C0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //C8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //D0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //D8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //E0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //E8
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //F0
SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook SpyHook //F8

    __asm
        {
SpyHook2:       ; end of hook entry point section
        pop     eax                     ; get stub return address
        pushfd
        push    ebx
        push    ecx
        push    edx
        push    ebp
        push    esi
        push    edi
        sub     eax, offset SpyHook1    ; compute entry point index
        mov     ecx, SDT_SYMBOLS_MAX
        mul     ecx
        mov     ecx, offset SpyHook2
        sub     ecx, offset SpyHook1
        div     ecx
        dec     eax
        mov     ecx, gfSpyHookPause     ; test pause flag
        add     ecx, -1
        sbb     ecx, ecx
        not     ecx
        lea     edx, [aSpyHooks + eax * SIZE SPY_HOOK_ENTRY]
        test    ecx, [edx.pbFormat]     ; format string == NULL?
        jz      SpyHook5
        push    eax
        push    edx
        call    PsGetCurrentThreadId    ; get thread id
        mov     ebx, eax
        pop     edx
        pop     eax
        cmp     ebx, ghSpyHookThread    ; ignore hook installer
        jz      SpyHook5
        mov     edi, gpDeviceContext
        lea     edi, [edi.SpyCalls]     ; get call context array
        mov     esi, SPY_CALLS          ; get number of entries
```

(continued)

```
SpyHook3:
        mov     ecx, 1                  ; set in-use flag
        xchg    ecx, [edi.fInUse]
        jecxz   SpyHook4                ; unused entry found
        add     edi, SIZE SPY_CALL      ; try next entry
        dec     esi
        jnz     SpyHook3
        mov     edi, gpDeviceContext
        inc     [edi.dMisses]           ; count misses
        jmp     SpyHook5                ; array overflow
SpyHook4:
        mov     esi, gpDeviceContext
        inc     [esi.dLevel]            ; set nesting level
        mov     [edi.hThread], ebx      ; save thread id
        mov     [edi.pshe], edx         ; save PSPY_HOOK_ENTRY
        mov     ecx, offset SpyHook6    ; set new return address
        xchg    ecx, [esp+20h]
        mov     [edi.pCaller], ecx      ; save old return address
        mov     ecx, KeServiceDescriptorTable
        mov     ecx, [ecx].ntoskrnl.ArgumentTable
        movzx   ecx, byte ptr [ecx+eax] ; get argument stack size
        shr     ecx, 2
        inc     ecx                     ; add 1 for result slot
        mov     [edi.dParameters], ecx  ; save number of parameters
        lea     edi, [edi.adParameters]
        xor     eax, eax                ; initialize result slot
        stosd
        dec     ecx
        jz      SpyHook5                ; no arguments
        lea     esi, [esp+24h]          ; save argument stack
        rep     movsd
SpyHook5:
        mov     eax, [edx.Handler]      ; get original handler
        pop     edi
        pop     esi
        pop     ebp
        pop     edx
        pop     ecx
        pop     ebx
        popfd
        xchg    eax, [esp]              ; restore eax and...
        ret                             ; ...jump to handler
SpyHook6:
        push    eax
        pushfd
        push    ebx
        push    ecx
        push    edx
        push    ebp
        push    esi
        push    edi
        push    eax
```

```
        call    PsGetCurrentThreadId    ; get thread id
        mov     ebx, eax
        pop     eax
        mov     edi, gpDeviceContext
        lea     edi, [edi.SpyCalls]     ; get call context array
        mov     esi, SPY_CALLS          ; get number of entries
SpyHook7:
        cmp     ebx, [edi.hThread]      ; find matching thread id
        jz      SpyHook8
        add     edi, SIZE SPY_CALL      ; try next entry
        dec     esi
        jnz     SpyHook7
        push    ebx                     ; entry not found ?!?
        call    KeBugCheck
SpyHook8:
        push    edi                     ; save SPY_CALL pointer
        mov     [edi.adParameters], eax ; store NTSTATUS
        push    edi
        call    SpyHookProtocol
        pop     edi                     ; restore SPY_CALL pointer
        mov     eax, [edi.pCaller]
        mov     [edi.hThread], 0        ; clear thread id
        mov     esi, gpDeviceContext
        dec     [esi.dLevel]            ; reset nesting level
        dec     [edi.fInUse]            ; clear in-use flag
        pop     edi
        pop     esi
        pop     ebp
        pop     edx
        pop     ecx
        pop     ebx
        popfd
        xchg    eax, [esp]              ; restore eax and...
        ret                             ; ...return to caller
SpyHook9:
        mov     dHooks1, offset SpyHook1
        mov     dHooks2, offset SpyHook2
        }
    n = (dHooks2 - dHooks1) / SDT_SYMBOLS_MAX;

    for (i = j = 0; i < SDT_SYMBOLS_MAX; i++, dHooks1 += n)
        {
        if ((ppbSymbols     != NULL) && (ppbFormats != NULL) &&
            (ppbSymbols [j] != NULL))
            {
            aSpyHooks [i].Handler  = (NTPROC) dHooks1;
            aSpyHooks [i].pbFormat =
                SpySearchFormat (ppbSymbols [j++], ppbFormats);
            }
        else
            {
```

(continued)

```
            aSpyHooks [i].Handler  = NULL;
            aSpyHooks [i].pbFormat = NULL;
            }
        }
    return;
    }
```

LISTING 5-3. *Implementation of the Hook Dispatcher*

So what is the `SpyHook` macro all about? Inside `SpyHookInitializeEx()`, this macro is repeated exactly 248 (`0xF8`) times, which matches the number of Windows 2000 Native API functions. At the top of Listing 5-3, this number is assigned to the constant `SDT_SYMBOLS_MAX`, which is the maximum of `SDT_SYMBOLS_NT4` and `SDT_SYMBOLS_NT5`. Yes, that's right—I am going to support Windows NT 4.0 as well! Back to the `SpyHook` macro: This sequence of invocations produces the ASM code shown in Listing 5-4. Each `SpyHook` entry produces three lines of code:

1. First, the current contents of the `EAX` register are saved on the stack.
2. Next, the linear address of the label `SpyHook2` is stored in `EAX`.
3. Finally, a `CALL` to the address in `EAX` is performed.

You might wonder what will happen when this `CALL` returns. Would the next group of `SpyHook` code lines be invoked? No—this `CALL` is not supposed to return, because the return address of this call is removed immediately from the stack after reaching the destination label `SpyHook2`, as the `POP EAX` instruction at the end of Listing 5-4 proves. This apparently senseless code is a trick of the old ASM programming days that has fallen into disuse in today's world of high-level object-oriented application development. This trick was applied by ASM gurus when they had to build an array of homogenous entry points to be dispatched to individual functions. Using almost identical code for all entry points guarantees equal spacing, so the index of the entry point used by a client could easily be calculated from the return address of the `CALL` instruction, the base address and total size of the array, and the number of entries, using a simple rule of three.

```
SpyHook1:
        push    eax
        mov     eax, offset SpyHook2
        call    eax
        push    eax
        mov     eax, offset SpyHook2
        call    eax
; 244 boring repetitions omitted
        push    eax
```

```
        mov     eax, offset SpyHook2
        call    eax
        push    eax
        mov     eax, offset SpyHook2
        call    eax
SpyHook2:
        pop     eax
```

LISTING 5-4. *Expansion of the SpyHook Macro Invocations*

For example, the return address of the first `CALL EAX` instruction in Listing 5-4 is the address of the second entry point. Generally, the return address of the `N`-th `CALL EAX` is equal to the address of entry `N+1`, except for the last one, which, of course, would return to `SpyHook2`. Thus, the zero-based array index of all entry points can be computed by the general formula in Figure 5-4. The underlying rule of three is as follows: `SDT_SYMBOLS_MAX` entry points fit into the memory block `SpyHook2–SpyHook1`. How many entry points fit into `ReturnAddress–SpyHook1`? Because the result of this computation is a number between one and `SDT_SYMBOLS_MAX`, it must be decremented by one to get the zero-based index.

The implementation of the formula in Figure 5-4 can be found in Listing 5-3, right after the ASM label `SpyHook2`. It is also included in the lower left corner of Figure 5-5, which presents the basic mechanics of the hook dispatch mechanism. Note that the i386 `MUL` instruction yields a 64-bit result in registers `EDX:EAX`, while the `DIV` instruction expects a 64-bit dividend in `EDX:EAX`, so there is no danger of an integer overflow. In the upper left corner, the `KiServiceTable` is depicted, which will be patched with the addresses of the entry points generated by the `SpyHook` macro. The middle section shows again the expanded macro code from Listing 5-4. The linear addresses of the entry points are shown on the right-hand side. By pure coincidence, the size of each entry point is 8 bytes, so the address is computed by multiplying the `KiServiceTable` index of each function by 8 and adding it to the address of `SpyHook1`.

Actually, I was just kidding—it's *not* pure coincidence that each entry is 8 bytes long. In reality, I spent a considerable amount of time figuring out the ideal implementation of the hook entries. Although not strictly necessary, aligning code on 32-bit boundaries is never a bad idea, because it speeds up performance. Of course, the performance gain is marginal here. You may wonder why I perform an indirect `CALL` to label `SpyHook2` through register `EAX`—wouldn't a simple middle-of-the-road `CALL SpyHook2` instruction have been much more efficient? Right! However, the problem with the i386 call (and jump) instructions is that they can be implemented in several ways that have the same effect but yield different instruction sizes. Just consult Intel's *Instruction Set Reference* of the Pentium CPU family (Intel 1999b).

$$\text{Index} = \frac{(\text{ReturnAddress} - \text{SpyHook1}) * \text{SDT_SYMBOLS_MAX}}{\text{SpyHook2} - \text{SpyHook1}} - 1$$

FIGURE 5-4. *Identifying Hook Entry Points by their Return Addresses*

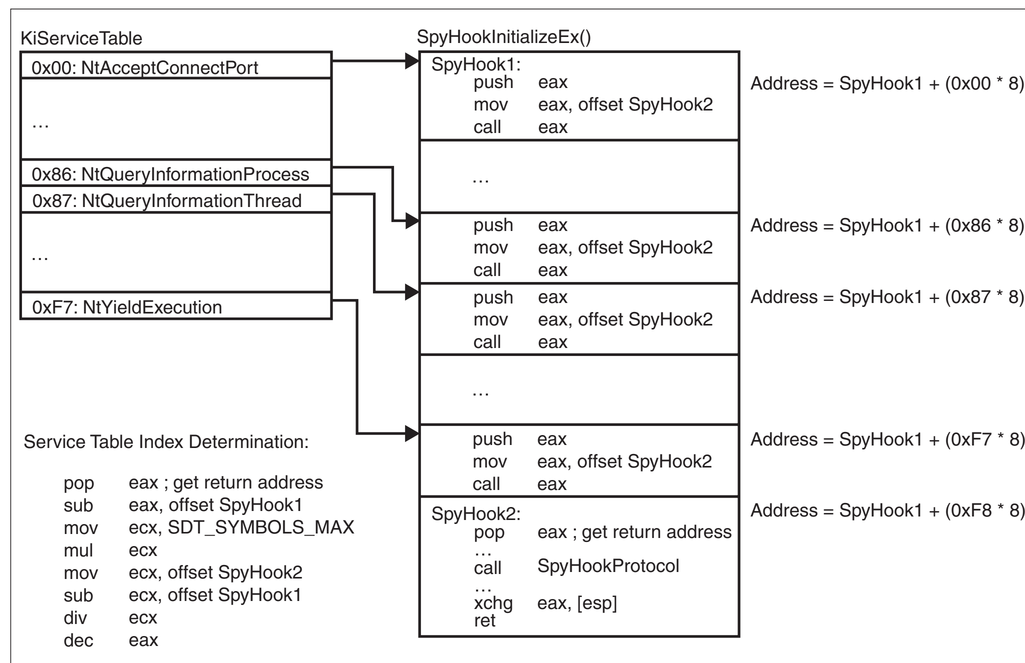

FIGURE 5-5. *Functional Principle of the Hook Dispatcher*

Because the choice of variant used is up to the compiler/assembler, there would be no guarantee that all entry points would end up in the same encoding. On the other hand, a `MOV EAX` with a constant 32-bit operand is always encoded in the same way, and so is the `CALL EAX` instruction.

Other points in Listing 5-3 should be clarified. Let's start with the final C code section starting after the label `SpyHook9`. The ASM code at this label has preset the C variables `dHook1` and `dHook2` with the linear addresses of the labels `SpyHook1` and `SpyHook2`. Next, the variable `n` is set to the size of each hook entry point by dividing

the size of the entry point array by the number of entries. Of course, this will yield eight. The remaining part of Listing 5-3 is a loop that initializes all entries of the global `aSpyHooks[]` array. This array consists of `SPY_HOOK_ENTRY` structures defined in the top half of Listing 5-3, and each entry is associated with a Native API function. To understand how their `Handler` and `pbFormat` members are set up, it is necessary to know more about the arguments `ppbSymbols` and `ppbFormats` passed to `SpyHookInitializeEx()`. Listing 5-5 shows the wrapper function `SpyHookInitialize()` that calls `SpyHookInitializeEx()` with arguments appropriate for the operating system (OS) version currently running. As noted earlier, the code doesn't test the OS version or the build number directly, but rather compares the `ServiceLimit` member of the SDT entry assigned to `ntoskrnl.exe` with the constants `SDT_SYMBOLS_NT4` and `SDT_SYMBOLS_NT5`. If none of them matches, the spy device will initialize all `aSpyHooks[]` entries with `NULL` pointers, effectively disabling the entire Native API hook mechanism.

```
BOOL SpyHookInitialize (void)
    {
    BOOL fOk = TRUE;

    switch (KeServiceDescriptorTable->ntoskrnl.ServiceLimit)
        {
        case SDT_SYMBOLS_NT4:
            {
            SpyHookInitializeEx (apbSdtSymbolsNT4, apbSdtFormats);
            break;
            }
        case SDT_SYMBOLS_NT5:
            {
            SpyHookInitializeEx (apbSdtSymbolsNT5, apbSdtFormats);
            break;
            }
        default:
            {
            SpyHookInitializeEx (NULL, NULL);
            fOk = FALSE;
            break;
            }
        }
    return fOk;
    }
```

LISTING 5-5. SpyHookInitialize() *Chooses the Symbol Table Matching the OS Version*

The global arrays apbSdtSymbolsNT4[] and apbSdtSymbolsNT5[] passed into SpyHookInitializeEx() as first argument ppbSymbols are simply string tables that contain all Windows NT 4.0 and Windows 2000 Native API function names, sorted by their KiServiceTable index, and terminated by a NULL pointer. The string array apbSdtFormats[] is shown in Listing 5-6. This format string list is one of the most important parts of the hook mechanism because it determines which Native API calls are logged and the appearance of each log entry. Obviously, the structure of these strings is borrowed from the printf() function of the C Runtime Library but specifically tailored to the most frequently used argument types of the Native API. Table 5-2 is a complete list of format IDs recognized by the API logger.

```
PBYTE apbSdtFormats [] =
    {
    "%s=NtCancelIoFile(%!,%i)",
    "%s=NtClose(%-)",
    "%s=NtCreateFile(%+,%n,%o,%i,%l,%n,%n,%n,%n,%p,%n)",
    "%s=NtCreateKey(%+,%n,%o,%n,%u,%n,%d)",
    "%s=NtDeleteFile(%o)",
    "%s=NtDeleteKey(%-)",
    "%s=NtDeleteValueKey(%!,%u)",
    "%s=NtDeviceIoControlFile(%!,%p,%p,%p,%i,%n,%p,%n,%p,%n)",
    "%s=NtEnumerateKey(%!,%n,%n,%p,%n,%d)",
    "%s=NtEnumerateValueKey(%!,%n,%n,%p,%n,%d)",
    "%s=NtFlushBuffersFile(%!,%i)",
    "%s=NtFlushKey(%!)",
    "%s=NtFsControlFile(%!,%p,%p,%p,%i,%n,%p,%n,%p,%n)",
    "%s=NtLoadKey(%o,%o)",
    "%s=NtLoadKey2(%o,%o,%n)",
    "%s=NtNotifyChangeKey(%!,%p,%p,%p,%i,%n,%b,%p,%n,%b)",
"%s=NtNotifyChangeMultipleKeys(%!,%n,%o,%p,%p,%p,%i,%n,%b,%p,%n,%b)",
    "%s=NtOpenFile(%+,%n,%o,%i,%n,%n)",
    "%s=NtOpenKey(%+,%n,%o)",
    "%s=NtOpenProcess(%+,%n,%o,%c)",
    "%s=NtOpenThread(%+,%n,%o,%c)",
    "%s=NtQueryDirectoryFile(%!,%p,%p,%p,%i,%p,%n,%n,%b,%u,%b)",
    "%s=NtQueryInformationFile(%!,%i,%p,%n,%n)",
    "%s=NtQueryInformationProcess(%!,%n,%p,%n,%d)",
    "%s=NtQueryInformationThread(%!,%n,%p,%n,%d)",
    "%s=NtQueryKey(%!,%n,%p,%n,%d)",
    "%s=NtQueryMultipleValueKey(%!,%p,%n,%p,%d,%d)",
    "%s=NtQueryOpenSubKeys(%o,%d)",
    "%s=NtQuerySystemInformation(%n,%p,%n,%d)",
    "%s=NtQuerySystemTime(%l)",
    "%s=NtQueryValueKey(%!,%u,%n,%p,%n,%d)",
    "%s=NtQueryVolumeInformationFile(%!,%i,%p,%n,%n)",
    "%s=NtReadFile(%!,%p,%p,%p,%i,%p,%n,%l,%d)",
    "%s=NtReplaceKey(%o,%!,%o)",
    "%s=NtSetInformationKey(%!,%n,%p,%n)",
```

```
    "%s=NtSetInformationFile(%!,%i,%p,%n,%n)",
    "%s=NtSetInformationProcess(%!,%n,%p,%n)",
    "%s=NtSetInformationThread(%!,%n,%p,%n)",
    "%s=NtSetSystemInformation(%n,%p,%n)",
    "%s=NtSetSystemTime(%l,%l)",
    "%s=NtSetValueKey(%!,%u,%n,%n,%p,%n)",
    "%s=NtSetVolumeInformationFile(%!,%i,%p,%n,%n)",
    "%s=NtUnloadKey(%o)",
    "%s=NtWriteFile(%!,%p,%p,%p,%i,%p,%n,%l,%d)",
    NULL
    };
```

LISTING 5-6. *Format Strings Used by the Native API Logger*

It's important to note that each format string must contain the function correctly spelled. `SpyHookInitializeEx()` walks though the list of Native API symbols it receives via its `ppbSymbols` argument and attempts to find a format string in the `ppbFormats` list that contains a matching function name. The comparison is performed by the helper function `SpySearchFormat()`, invoked in the `if` clause at the end of Listing 5-3. Because many string search operations must be performed for all `aSpyHooks[]` entries to be set up, I am using a highly optimized search engine based on the ingenious "Shift/And Search Algorithm." If you want to learn more about its implementation, please check out the `SpySearch*()` function group in the source file `\src\w2k_spy\w2k_spy.c` on the companion CD. As soon as `SpyHookInitializeEx()` exits the loop, all `Handler` members in the `aSpyHooks[]` array point to the appropriate hook entry points, and the `pbFormat` members provide the matching format string, if any. With Windows NT 4.0, both members of the entries in the index range `0xD3` to `0xF8` are set to `NULL`, because they are undefined for this version.

TABLE 5-2. *Recognized Format Control IDs*

ID	NAME	DESCRIPTION
%+	Handle (register)	Logs a handle and object name and adds them to the handle table
%!	Handle (retrieve)	Logs a handle and retrieves its object name from the handle table
%-	Handle (unregister)	Logs a handle and object name and removes them from the handle table
%a	ANSI string	Logs a string of 8-bit ANSI characters
%b	BOOLEAN	Logs an 8-bit BOOLEAN value
%c	CLIENT_ID *	Logs the members of a CLIENT_ID structure

(continued)

TABLE 5-2. *(continued)*

ID	NAME	DESCRIPTION
%d	DWORD *	Logs the value of the addressed DWORD
%i	IO_STATUS_BLOCK *	Logs the members of an IO_STATUS_BLOCK structure
%l	LARGE_INTEGER *	Logs the value of a LARGE_INTEGER structure
%n	Number (DWORD)	Logs the value of an unsigned 32-bit number
%o	OBJECT_ATTRIBUTES *	Logs the *ObjectName* of an object
%p	Pointer	Logs the target address of a pointer
%s	Status (NTSTATUS)	Logs a NT status code
%u	UNICODE_STRING *	Logs the *Buffer* member of an UNICODE_STRING structure
%w	Wide character string	Logs a string of 16-bit Unicode characters
%%	Percent escape	Logs a single '%' character

The most notable property of this hook mechanism design is that it is completely data driven. The hook dispatcher can be adapted to a new Windows 2000 release by simply adding a new API symbol table. Moreover, the logging of additional API functions can be enabled at any time by adding new format strings to the `apbSdtFormats[]` array. There is no need to write any additional code—the actions of the API spy are completely determined by a set of character strings! However, care must be taken while defining format strings. Never forget that `w2k_spy.sys` runs as a kernel-mode driver. On this system level, errors are not handled very gracefully. Giving an invalid argument to a Win32 API is not a problem—you will get an error window, and the application will be terminated. In kernel-mode, the tiniest access violation will cause a Blue Screen. So be careful—an improper or missing format control ID at the right place can easily tear down your system. Even a simple character string *sometimes* can be deadly!

The only thing left to discuss is the large ASM block inside `SpyHookInitializeEx()`, enclosed by the ASM labels `SpyHook2` and `SpyHook9`. One interesting property of this code is that it is never executed when `SpyHookInitializeEx()` is called. On entry, the function code simply jumps across this entire section and resumes execution at the label `SpyHook9`, shortly before the C section containing the `aSpyHooks[]` array initialization starts. This code can only be entered via the `Handler` members of this array. Later, I will show how these entry points are linked to the SDT.

One of my foremost aims in designing this code was to make it absolutely nonintrusive. Intercepting operating system calls is dangerous because you never know whether the called code relies on some unknown properties of the calling context. Theoretically, it should suffice to obey the `__stdcall` convention, but it is possible

that problems may occur. I have chosen to put the original Native API function handler into almost exactly the same environment it would find if no hooks were present. This means that the function should run on the original argument stack and see all CPU registers as they are passed in by the caller. Of course, a minimal degree of intrusion must be accepted—otherwise, no monitoring would be possible. Here, the most significant intervention is the manipulation of the return address on the stack. If you flip back to Figure 5-3, you see that the caller's return address is on top of the argument stack on entry of the function. The hook dispatcher inside `SpyHookInitializeEx()` grabs this address and puts its own `SpyHook6` label address there. Thus, the original Native API function handler will branch to this location after terminating, enabling the hook dispatcher to inspect its arguments and returned values.

Before calling the original handler, the dispatcher sets up a `SPY_CALL` control block (see top section of Listing 5-3) containing parameters it needs later. Some of them are required for proper API call logging, whereas others provide information about the caller so the dispatcher can return control to it after writing the log entry, just as if nothing had happened. The spy device maintains an array of `SPY_CALL` structures in its global `DEVICE_CONTEXT` block, accessible via the global variable `gpDeviceContext` in `w2k_spy.c`. The hook dispatcher searches for a free `SPY_CALL` slot by examining their `fInUse` members. It uses the CPU's `XCHG` instruction to load and set this member in a single operation. This is very important because this code runs in a multithreaded environment, where read/write accesses to global data must be protected against race conditions. If a free slot is available, the dispatcher stores the caller's thread ID obtained from `PsGetCurrentThreadId()`, the address of the `SPY_HOOK_ ENTRY` associated with the current API function, the return address of the caller, and the entire argument stack. The number of argument bytes to be copied is taken from the `KiArgumentTable` array stored in the system's SDT. If all `SPY_CALL` entries are in use, the original API function handler is invoked without logging it.

The necessity of a `SPY_CALL` array comes again from the multithreading nature of Windows 2000. It happens quite frequently that a Native API function is suspended, and another thread gains control, invoking another Native API function during its time slice. This means that the spy device's hook dispatcher can be reentered at any time and at any execution point. If the hook dispatcher would have a single global `SPY_CALL` storage area, it would be overwritten by the running thread before the waiting thread has finished using it. This situation is an ideal candidate for a Blue Screen. To gain a better sense of the nesting level typically occurring within the Native API, I have added the `dLevel` and `dMisses` members to the spy's `DEVICE_CONTEXT` structure. Whenever the hook dispatcher is reentered (i.e., whenever a new `SPY_CALL` slot is occupied) `dLevel` is increased by one. If the maximum nesting level is exceeded (i.e., if no more `SPY_CALL` structures are available), `dMisses` is increased, indicating that a log entry is missing. My observations have shown that in practical

situations, nesting levels of up to four are easily observable. It is possible that the Native API is reentered even more frequently in heavy-load situations, so I set the upper limit generously to 256.

Before invoking the original API handler, the hook dispatcher restores all CPU registers including the `EFLAGS`, and branches to the function's entry point. This is done immediately before the `SpyHook6` label in Listing 5-3. At this time, `SpyHook6` is on top of the stack, followed by the caller's arguments. As soon as the API handler exits, control is passed back to the hook dispatcher at the `SpyHook6` label. The code executed from there on is also designed to be as nonintrusive as possible. This time, the main objective is to allow the caller to see the call context almost exactly as it was set up by the original API function handler. The main problem of the dispatcher is now to find the `SPY_CALL` entry where it has stored the information about the current API call. The only reliable cue it has is the caller's thread ID, which has been saved to the `hThread` member of the `SPY_CALL` structure. Therefore, the dispatcher loops through the entire `SPY_CALL` array trying to find a matching thread ID. Note that the code is not concerned about the value of the `fInUse` flag; this is not necessary because all unused entries have `hThread` set to zero, which is the thread ID of the system idle thread. The loop should *always* terminate before the end of the array is reached. Otherwise, the dispatcher cannot return control to the caller, which is fatal. In this case, the code has few options, so it runs into a `KeBugCheck()` that results in a controlled system shutdown. This situation should never occur, but if it does, something terrible must have happened to the system, so the shutdown is probably the best solution.

If the matching `SPY_CALL` slot can be found, the hook dispatcher has almost finished its job. The last major action is the invocation of the logging function `SpyHookProtocol()`, passing in a pointer to the `SPY_CALL` structure. Everything the logger needs is stored there. After `SpyHookProtocol()` returns, the dispatcher frees its `SPY_CALL` slot, restores all CPU registers, and returns to the caller.

THE API HOOK PROTOCOL

A good API spy should look at the arguments *after* the original function has been called, because the function might return additional data in buffers passed in by reference. Therefore, the main logging function `SpyHookProtocol()` is called at the end of the hook handler, just before the API function returns to the caller. Before discussing secrets of its implementation, examine the following two sample protocols for a foretaste of what's to come. Figure 5-6 is a snapshot of the logged file operations performed in the context of the console command **dir c:**.

Please compare the log entries listed in Figure 5-6 with the protocol format strings contained in Listing 5-6. In Example 5-1, the format strings of `NtOpenFile()` and `NtClose()` are contrasted to the first and fourth protocol lines in Figure 5-6, respectively. The similarities are striking; for each format control ID preceded by a percent character (cf. Table 5-2), an associated parameter value entry is generated in the protocol. However, the protocol obviously contains some additional information that is not part of the format strings. I'll reveal the reason for this discrepancy in a moment.

The general format of a protocol entry is shown in Example 5-2. Each entry consists of a fixed number of fields with intermittent separators. The separators allow the entries to be easily parsed by a program. The fields are constructed on the basis of the following set of simple rules:

- All numeric quantities are stated in hexadecimal notation without leading zeros and without the usual leading "0x."
- Function arguments are separated by commas.
- String arguments are enclosed in double quotes.
- If a pointer argument is `NULL,` its value is omitted.
- The values of structure members are separated by dots.

```
18:s0=NtOpenFile(+46C.18,n100001,o"\??\C:\",i0.1,n3,n4021)1BFEE5AE05B6710,278,2
19:s0=NtQueryInformationFile(!46C.18="\??\C:\",i0.6,p12E21C,n210,n9)1BFEE5AE05B6710,278,2
1A:s0=NtQueryVolumeInformationFile(!46C.18="\??\C:\",i0.12,p1321C8,n21C,n5)1BFEE5AE05B6710,278,2
1B:s0=NtClose(-46C.18="\??\C:\")1BFEE5AE05B6710,278,1
1C:s0=NtOpenFile(+46C.18,n100001,o"\??\C:\",i0.1,n3,n4021)1BFEE5AE05B6710,278,2
1D:s0=NtQueryInformationFile(!46C.18="\??\C:\",i0.6,p12E664,n210,n9)1BFEE5AE05B6710,278,2
1E:s0=NtQueryVolumeInformationFile(!46C.18="\??\C:\",i0.26,p1321C8,n220,n1)1BFEE5AE05B6710,278,2
1F:s0=NtClose(-46C.18="\??\C:\")1BFEE5AE05B6710,278,1
20:s0=NtOpenFile(+46C.18,n100001,o"\??\c:\",i0.1,n3,n4021)1BFEE5AE05FFCA0,278,2
21:s0=NtQueryDirectoryFile(!46C.18="\??\c:\",p,p,p,i0.68,p12E994,n268,n3,bTRUE,u"*",bFALSE)1BFEE5AE05FFCA0,278,2
22:s0=NtQueryDirectoryFile(!46C.18="\??\c:\",p,p,p,i0.9FE,p139128,n1000,n3,bFALSE,u,bFALSE)1BFEE5AE05FFCA0,278,2
23:s80000006=NtQueryDirectoryFile(!46C.18="\??\c:\",p,p,p,i80000006.0,p139128,n1000,n3,bFALSE,u,bFALSE)1BFEE5AE0661960,278,2
24:s0=NtClose(-46C.18="\??\c:\")1BFEE5AE0661960,278,1
25:s0=NtOpenFile(+46C.18,n100001,o"\??\c:\",i0.1,n3,n800021)1BFEE5AE0661960,278,2
26:s0=NtQueryVolumeInformationFile(!46C.18="\??\c:\",i0.20,p12ED10,n20,n7)1BFEE5AE0661960,278,2
27:s0=NtClose(-46C.18="\??\c:\")1BFEE5AE0661960,278,1
```

FIGURE 5-6. *Sample Protocol of the Command* dir c:\

```
"%s=NtOpenFile(%+,%n,%o,%i,%n,%n)"
18:s0=NtOpenFile(+46C.18,n100001,o"\??\C:\",i0.1,n3,n4021)1BFEE5AE05B6710,278,2

"%s=NtClose(%-)"
1B:s0=NtClose(-46C.18="\??\C:\")1BFEE5AE05B6710,278,1
```

EXAMPLE 5-1. *Comparing Format Strings to Protocol Entries*

```
<#>:<status>=<function>(<arguments>)<time>,<thread>,<handles>
```

EXAMPLE 5-2. *General Protocol Entry Format*

- Object names associated with a handle are appended to the handle's value with a separating "=" character.
- The date/time stamp is specified in 1/10 microsecond since 01-01-1601—the basic system time format of Windows 2000.
- The thread ID indicates the unique numeric identifier of the thread that called the API function.
- The handle count states the number of handles currently registered in the spy device's handle list. This list allows the protocol function to look up the object names associated with handles.

Figure 5-7 is another API spy protocol resulting from the command **type c:\boot.ini** issued in a console window. The following is the semantic interpretation of some selected log entries:

- In line `0x31`, `NtCreateFile()` is called to open the file `\??\c:\boot.ini`. `(o"\??\c:\boot.ini")` The function returned an `NTSTATUS` code of zero `(s0)`, that is, `STATUS_SUCCESS`, and allocated a new file handle with value `0x18`, owned by process `0x46C (+46C.18)`. Consequently, the handle count rises from one to two.
- In line `0x36`, the **type** command reads in the first 512 bytes `(n200)` from file `\??\c:\boot.ini` to a buffer at the linear address `0x0012F5B4` `(p12F5B4)`, passing the handle obtained from `NtCreateFile()` `(!46C.18="\??\c:\boot.ini")` to `NtReadFile()`. The system successfully returns 512 bytes `(i0.200)`.

```
2D:s0=NtOpenFile(+46C.18,n100001,o"\??\c:\",i0.1,n3,n4021)1BFEE5B075EE890,278,2
2E:s0=NtQueryDirectoryFile(!46C.18="\??\c:\",p,p,p,i0.6E,p12F4DC,n268,n3,bTRUE,u"boot.ini",bFALSE)1BFEE5B075EE890,278,2
2F:s80000006=NtQueryDirectoryFile(!46C.18="\??\c:\",p,p,p,i80000006.0,p1389F0,n1000,n3,bFALSE,u,bFALSE)1BFEE5B07606FC0,278,2
30:s0=NtClose(-46C.18="\??\c:\")1BFEE5B07606FC0,278,1
31:s0=NtCreaterFile(+46C.18,n80100080,o"\??\c:\boot.ini",i0.1,l,n80,n3,n1n60,p,n0)1BFEE5B07606FC0,278,2
32:s0=NtQueryVolumeInformationFile(!46C.18="\??\c:\boot.ini",i0.8,p12E728,n8,n4)1BFEE5B07606FC0,278,2
33:s0=NtQueryVolumeInformationFile(!46C.18="\??\c:\boot.ini",i0.8,p12E778,n8,n4)1BFEE5B07606FC0,278,2
34:s0=NtQueryInformationFile(!46C.18="\??\c:\boot.ini",i0.18,p12E758,n18,n5)1BFEE5B07606FC0,278,2
35:s0=NtSetInformationFile(!46C.18="\??\c:\boot.ini",i0.0,p12E780,n8,nE)1BFEE5B07606FC0,278,2
36:s0=NtReadFile(!46C.18="\??\c:\boot.ini",p,p,p,i0.200,p12F5B4,n200,l,d)1BFEE5B07606FC0,278,2
37:s0=NtQueryInformationFile(!46C.18="\??\c:\boot.ini",i0.8,p12E780,n8,nE)1BREE5B07650550,278,2
38:s0=NtSetInformationFile(!46C.18="\??\c:\boot.ini",i0.0,p12E780,n8,nE)1BFEE5B07650550,278,2
39:s0=NtReadFile(!46C.18="\??\c:\boot.ini",p,p,p,i0.4B,p12F5B4,n200,l,d)1BFEE5B07650550,278,2
3A:s0=NtQueryInformationFile(!46C.18="\??\c:\boot.ini",i0.8,p12E780,n8,nE)1BFEE5B07650550,278,2
3B:s0=NtSetInformationFile(!46C.18="\??\c:\boot.ini",i0.0,p12E780,n8,nE)1BFEE5B07650550,278,2
3C:s0=NtClose(-46C.18="\??\c:\boot.ini")1BFEE5B07650550,278,1
```

FIGURE 5-7. *Sample Protocol of the Command* `type c:\boot.ini`

- In line `0x39`, another file block of 512 bytes is ordered `(n200)`. This time, however, the end of the file is reached, so `NtReadFile()` returns 75 bytes only `(i0.4B)`. Obviously, the size of my `boot.ini` file is 512 + 75 = 587 bytes, which is correct.
- In line `0x3C`, the file handle to `\??\c:\boot.ini` is successfully released by `NtClose() (-46C.18="\??\c:\boot.ini")`, so the handle count drops from two to one.

By now, you should have an idea of how the API spy protocol is structured, which will help you grasp the details of the protocol generation mechanism, to be discussed next. As already noted, the main API call logging function is called `SpyHookProtocol()`. This function, shown in Listing 5-7, uses the data in the `SPY_CALL` structure it receives from the hook dispatcher to write a protocol record for each API function call to a circular buffer. A spy device client can read this protocol via IOCTL calls. Each record is a text line terminated by a single line-feed character (`'\n'` in C notation). Access to the protocol buffer is serialized by means of the kernel mutex `KMUTEX kmProtocol`, located in the global `DEVICE_CONTEXT` structure of the spy device. The functions `SpyHookWait()` and `SpyHookRelease()` in Listing 5-7 acquire and release this mutex object. All accesses to the protocol buffer must be preceded by `SpyHookWait()` and followed `SpyHookRelease()`, as demonstrated by the `SpyHookProtocol()` function.

```
NTSTATUS SpyHookWait (void)
    {
    return MUTEX_WAIT (gpDeviceContext->kmProtocol);
    }

// -----------------------------------------------------------------

LONG SpyHookRelease (void)
    {
    return MUTEX_RELEASE (gpDeviceContext->kmProtocol);
    }

// -----------------------------------------------------------------
// <#>:<status>=<function>(<arguments>)<time>,<thread>,<handles>

void SpyHookProtocol (PSPY_CALL psc)
    {
    LARGE_INTEGER liTime;
    PSPY_PROTOCOL psp = &gpDeviceContext->SpyProtocol;

    KeQuerySystemTime (&liTime);

    SpyHookWait ();

    if (SpyWriteFilter (psp, psc->pshe->pbFormat,
                             psc->adParameters,
                             psc->dParameters))
        {
        SpyWriteNumber (psp, 0, ++(psp->sh.dCalls));   // <#>:
        SpyWriteChar   (psp, 0, ':');
                                                    // <status>=
        SpyWriteFormat (psp, psc->pshe->pbFormat, //  <function>
                             psc->adParameters);  //   (<arguments>)

        SpyWriteLarge  (psp, 0, &liTime);              // <time>,
        SpyWriteChar   (psp, 0, ',');

        SpyWriteNumber (psp, 0, (DWORD) psc->hThread); // <thread>,
        SpyWriteChar   (psp, 0, ',');

        SpyWriteNumber (psp, 0, psp->sh.dHandles);     // <handles>
        SpyWriteChar   (psp, 0, '\n');
        }
    SpyHookRelease ();
    return;
    }
```

LISTING 5-7. *The main hook protocol function* SpyHookProtocol()

If you compare the main body of `SpyHookProtocol()` in Listing 5-7 with the general protocol entry layout in Example 5-2, it is obvious which statement generates which entry field. It also becomes clear why the protocol strings in Listing 5-6 don't account for the entire entry data—some function-independent data are added by `SpyHookProtocol()` without the help of the format string. It's the `SpyWriteFormat()` call at the heart of `SpyHookProtocol()` that generates the `<status>=<function>(<arguments>)` part, based on the format string associated with the currently logged API function. Consult the source files `w2k_spy.c` and `w2k_spy.h` in directory `\src\w2k_spy` of the accompanying sample CD for more information about the implementation of the various `SpyWrite*()` functions inside the spy device driver.

Note that this code is somewhat critical. This code was written in 1997 for Windows NT 4.0, and it worked like a charm then. After porting the program to Windows 2000, occasional Blue Screens occurred when the hooks remained installed for a longer time interval. Worse yet, some special operations reliably produced an instant Blue Screen, for example, navigating to "My Computer" in the **File \ Open** dialog of my favorite text editor. Analyzing numerous crash dumps, I found that the crashes were the result of invalid non-`NULL` pointers passed to some API functions. As soon as the spy device attempted to follow one of these pointers in order to log the data it referenced, the system crashed. Typical candidates were pointers to `IO_STATUS_BLOCK` structures, and invalid string pointers inside `UNICODE_STRING` and `OBJECT_ATTRIBUTES` structures. I also found some `UNICODE_STRING`s with `Buffer` members that were not zero-terminated. Therefore, I emphasize again that you should not assume that all `UNICODE_STRING`s are zero-terminated. In case of doubt, the `Length` member always tells the number of valid bytes you can expect at the `Buffer` address.

To remedy this problem, I have added pointer validation to all logging functions that have to follow client pointers. To this end, I use the `SpyMemoryTestAddress()` function discussed in Chapter 4 that checks out whether a linear address points to a valid page-table entry (PTE). See Listings 4-22 and 4-24 for details. Another alternative possibility would have been the addition of Structured Exception Handling (SEH) clauses `(__try` / `__except)`.

HANDLING HANDLES

It is important to note that `SpyHookProtocol()` logs an API function call only if the `SpyWriteFilter()` function in its `if` clause condition returns `TRUE`. This is a trick that helps to suppress garbage in the hook protocol. For example, moving the mouse across the screen triggers a distracting series of `NtReadFile()` calls. Another source of garbage has an interesting equivalent in physics: If you are measuring a physical effect in an experimental situation, the act of measurement itself interferes with the measured effect and leads to distortion of the results. This also can happen in API logging. Note that the `NtDeviceIoControlFile()` function is also included in the format string

array in Listing 5-6. However, a client of the spy device uses device I/O control calls to read the API hook protocol. This means that the client will find its own `NtDeviceIoControlFile()` calls in the protocol data. Depending on the frequency of the IOCTL transactions, the desired data might easily get lost in self-made noise. The spy device works around this problem by remembering the ID of the thread that installed the API hooks to be able to ignore all API calls originating from this thread.

`SpyWriteFilter()` eliminates garbage by ignoring all API calls involving handles if the call that generated the handle has not been logged. If the spy device observes that a handle is closed or otherwise returned to the system, any subsequent functions using this handle value are discarded as well. Effectively, this trick suppresses all API calls that involve long-term handles created by the system or other processes before the start of the API hook protocol. Of course, filtering can be enabled or disabled on behalf of the client by means of IOCTL. You can easily test the usefulness of the filter mechanism with the sample client application introduced later in this chapter. You will be surprised how great this simple "noise filter" works!

In Listing 5-6, the functions that generate handles are `NtCreateFile()`, `NtCreateKey()`, `NtOpenFile()`, `NtOpenKey()`, `NtOpenProcess()`, and `NtOpenThread()`. All of these functions contain a `%+` control token in their format strings, which is identified as "Handle (register)" in Table 5-2. Functions that close or invalidate handles are `NtClose()` and `NtDeleteKey()`. Both include a `%-` token in their format strings, labeled "Handle (unregister)" in Table 5-2. Other functions that simply use a handle without creating or releasing it feature a `%!` format control ID. Basically, a handle is a number that uniquely identifies an object in the context of a process. Physically, it provides an index into a handle table that contains the properties of the associated object. When a new handle is issued by an API function, the client usually has to pass in an `OBJECT_ATTRIBUTES` structure that contains, among other things, the name of the object it wishes to access. Later, this name is no longer required because the system can look up all object properties it needs using the object handle and the handle table. This is unfortunate for the user of an API spy because it necessitates wading through countless protocol entries containing meaningless numbers instead of symbolic names. Therefore, my spy device registers all object names together with the respective handle values and the IDs of the owning processes, updating this list whenever a new handle appears. When one of the registered handle/process pairs reappears later, the API logger retrieves the original symbolic name from the list and adds it to the protocol.

A handle remains registered until it is explicitly closed by an API function or reappears in an API call that generates a new handle. With Windows 2000, I frequently observed that the same handle value is returned several times by the system, although the protocol doesn't contain any call that has closed this handle before. I don't remember having seen this with Windows NT 4.0. A registered handle that reappears with different object attributes has obviously been closed somehow, so it

must be unregistered. Otherwise, the handle directory of the spy device eventually would run into an overflow situation.

The `SpyWriteFilter()` function called by `SpyHookProtocol()` in Listing 5-7 is an essential part of this handle tracking mechanism. Every call to any of the hooked API functions has to pass through it. The implementation is shown in Listing 5-8.

```
BOOL SpyWriteFilter (PSPY_PROTOCOL psp,
                     PBYTE         pbFormat,
                     PVOID         pParameters,
                     DWORD         dParameters)
    {
    PHANDLE            phObject = NULL;
    HANDLE             hObject  = NULL;
    POBJECT_ATTRIBUTES poa      = NULL;
    PDWORD             pdNext;
    DWORD              i, j;

    pdNext = pParameters;
    i = j  = 0;

    while (pbFormat [i])
        {
        while (pbFormat [i] && (pbFormat [i] != '%')) i++;

        if (pbFormat [i] && pbFormat [++i])
            {
            j++;

            switch (pbFormat [i++])
                {
                case 'b':
                case 'a':
                case 'w':
                case 'u':
                case 'n':
                case 'l':
                case 's':
                case 'i':
                case 'c':
                case 'd':
                case 'p':
                    {
                    break;
                    }
                case 'o':
                    {
                    if (poa == NULL)
                        {
                        poa = (POBJECT_ATTRIBUTES) *pdNext;
```

(continued)

```
                    }
                break;
                }
            case '+':
                {
                if (phObject == NULL)
                    {
                    phObject = (PHANDLE) *pdNext;
                    }
                break;
                }
            case '!':
            case '-':
                {
                if (hObject == NULL)
                    {
                    hObject = (HANDLE) *pdNext;
                    }
                break;
                }
            default:
                {
                j—;
                break;
                }
            }
        pdNext++;
        }
    }
return // number of arguments ok
       (j == dParameters)
       &&
        // no handles involved
       (((phObject == NULL) && (hObject == NULL))
        ||
        // new handle, successfully registered
        ((phObject != NULL) &&
         SpyHandleRegister (psp, PsGetCurrentProcessId (),
                            *phObject, OBJECT_NAME (poa)))
        ||
        // registered handle
        SpyHandleSlot (psp, PsGetCurrentProcessId (), hObject)
        ||
        // filter disabled
        (!gfSpyHookFilter));
}
```

LISTING 5-8. `SpyWriteFilter()` *Excludes Undesired API Calls from the Protocol*

Basically, `SpyWriteFilter()` scans a protocol format string for occurrences of `%o` (object attributes), `%+` (new handle), `%!` (open handle), and `%-` (closed handle) and takes special actions for certain combinations, as follows:

- If no handles are involved, the API call is always logged. This concerns all API functions with format strings that don't contain the format control IDs `%+`, `%!`, and `%-`.
- If `%+` is included in the format string, indicating that this function allocates a new handle, this handle is registered and associated with the name of the first `%o` item in the format string using the helper function `SpyHandleRegister()`. If no such item exists, the handle is registered with an empty string. If the registration succeeds, the call is logged.
- If `%!` or `%-` occur in the format string, the called function uses or closes an open handle. In this case, `SpyWriteFilter()` tests whether this handle is registered by querying its slot number via `SpyHandleSlot()`. If this function succeeds, the API call is logged.
- In all other cases, the call is logged only if the filter mechanism is disabled, as indicated by the global Boolean variable `gfSpyHookFilter`.

The handle directory is part of the `SPY_PROTOCOL` structure, included in the global `DEVICE_CONTEXT` structure of the spy device `w2k_spy.sys` and defined in Listing 5-9, along with its `SPY_HEADER` substructure. Following the structure definitions is the source code of the four handle management functions `SpyHandleSlot()`, `SpyHandleName()`, `SpyHandleUnregister()`, and `SpyHandleRegister()`. A handle is registered by appending its value to the current end of the `ahObjects[]` array. At the same time, the ID of the owning process is added to the `ahProcesses[]` array, the object name is copied to the `awNames[]` buffer, and the start offset of the name is stored in the `adNames[]` array. When a handle is unregistered, these actions are undone, shifting left all subsequent array members to ensure that none of the arrays contains "holes." The constant definitions at the top of Listing 5-9 define the dimensions of the handle directory: It can take up to 4,096 handles, the name data limit is set to 1,048,576 Unicode characters (2 MB), and the protocol buffer size amounts to 1 MB.

```
#define SPY_HANDLES         0x00001000 // max number of handles
#define SPY_NAME_BUFFER     0x00100000 // object name buffer size
#define SPY_DATA_BUFFER     0x00100000 // protocol data buffer size

// -----------------------------------------------------------------

typedef struct _SPY_HEADER
    {
    LARGE_INTEGER liStart;  // start time
    DWORD         dRead;    // read data index
    DWORD         dWrite;   // write data index
    DWORD         dCalls;   // api usage count
    DWORD         dHandles; // handle count
    DWORD         dName;    // object name index
    }
    SPY_HEADER, *PSPY_HEADER, **PPSPY_HEADER;

#define SPY_HEADER_ sizeof (SPY_HEADER)

// -----------------------------------------------------------------

typedef struct _SPY_PROTOCOL
    {
    SPY_HEADER    sh;                            // protocol header
    HANDLE        ahProcesses [SPY_HANDLES];     // process id array
    HANDLE        ahObjects   [SPY_HANDLES];     // handle array
    DWORD         adNames     [SPY_HANDLES];     // name offsets
    WORD          awNames     [SPY_NAME_BUFFER]; // name strings
    BYTE          abData      [SPY_DATA_BUFFER]; // protocol data
    }
    SPY_PROTOCOL, *PSPY_PROTOCOL, **PPSPY_PROTOCOL;

#define SPY_PROTOCOL_ sizeof (SPY_PROTOCOL)

// -----------------------------------------------------------------

DWORD SpyHandleSlot (PSPY_PROTOCOL psp,
                     HANDLE        hProcess,
                     HANDLE        hObject)
    {
    DWORD dSlot = 0;

    if (hObject != NULL)
        {
        while ((dSlot < psp->sh.dHandles)
               &&
               ((psp->ahProcesses [dSlot] != hProcess) ||
                (psp->ahObjects   [dSlot] != hObject ))) dSlot++;
```

```
        dSlot = (dSlot < psp->sh.dHandles ? dSlot+1 : 0);
        }
    return dSlot;
    }

// -----------------------------------------------------------------

DWORD SpyHandleName (PSPY_PROTOCOL psp,
                     HANDLE        hProcess,
                     HANDLE        hObject,
                     PWORD         pwName,
                     DWORD         dName)
    {
    WORD  w;
    DWORD i;
    DWORD dSlot = SpyHandleSlot (psp, hProcess, hObject);

    if ((pwName != NULL) && dName)
        {
        i = 0;

        if (dSlot)
            {
            while ((i+1 < dName) &&
                   (w = psp->awNames [psp->adNames [dSlot-1] + i]))
                {
                pwName [i++] = w;
                }
            }
        pwName [i] = 0;
        }
    return dSlot;
    }

// -----------------------------------------------------------------

DWORD SpyHandleUnregister (PSPY_PROTOCOL psp,
                           HANDLE        hProcess,
                           HANDLE        hObject,
                           PWORD         pwName,
                           DWORD         dName)
    {
    DWORD i, j;
    DWORD dSlot = SpyHandleName (psp, hProcess, hObject,
                                 pwName, dName);
    if (dSlot)
        {
        if (dSlot == psp->sh.dHandles)
            {
            // remove last name entry
```

(continued)

```
            psp->sh.dName = psp->adNames [dSlot-1];
            }
        else
            {
            i = psp->adNames [dSlot-1];
            j = psp->adNames [dSlot  ];

            // shift left all remaining name entries
            while (j < psp->sh.dName)
                {
                psp->awNames [i++] = psp->awNames [j++];
                }
            j -= (psp->sh.dName = i);

            // shift left all remaining handles and name offsets
            for (i = dSlot; i < psp->sh.dHandles; i++)
                {
                psp->ahProcesses [i-1] = psp->ahProcesses [i];
                psp->ahObjects   [i-1] = psp->ahObjects   [i];
                psp->adNames     [i-1] = psp->adNames     [i] - j;
                }
            }
        psp->sh.dHandles–;
        }
    return dSlot;
    }

// -----------------------------------------------------------------

DWORD SpyHandleRegister (PSPY_PROTOCOL   psp,
                         HANDLE          hProcess,
                         HANDLE          hObject,
                         PUNICODE_STRING puName)
    {
    PWORD pwName;
    DWORD dName;
    DWORD i;
    DWORD dSlot = 0;

    if (hObject != NULL)
        {
        // unregister old handle with same value
        SpyHandleUnregister (psp, hProcess, hObject, NULL, 0);

        if (psp->sh.dHandles == SPY_HANDLES)
            {
```

```
            // unregister oldest handle if overflow
            SpyHandleUnregister (psp, psp->ahProcesses [0],
                                 psp->ahObjects [0], NULL, 0);
            }
        pwName = ((puName != NULL) && SpyMemoryTestAddress (puName)
                  ? puName->Buffer
                  : NULL);

        dName  = ((pwName != NULL) && SpyMemoryTestAddress (pwName)
                  ? puName->Length / WORD_
                  : 0);

        if (dName + 1 <= SPY_NAME_BUFFER - psp->sh.dName)
            {
            // append object to end of list
            psp->ahProcesses [psp->sh.dHandles] = hProcess;
            psp->ahObjects   [psp->sh.dHandles] = hObject;
            psp->adNames     [psp->sh.dHandles] = psp->sh.dName;

            for (i = 0; i < dName; i++)
                {
                psp->awNames [psp->sh.dName++] = pwName [i];
                }
            psp->awNames [psp->sh.dName++] = 0;

            psp->sh.dHandles++;
            dSlot = psp->sh.dHandles;
            }
        }
    return dSlot;
    }
```

LISTING 5-9. *Handle Management Structures and Functions*

CONTROLLING THE API HOOKS IN USER-MODE

A spy device client running in user-mode can control the Native API hook mechanism and the protocol it generates by means of a set of IOCTL functions. This set of functions with names of type `SPY_IO_HOOK_*` was mentioned in Chapter 4, where the memory spying functions of `w2k_spy.sys` were discussed (see Listing 4-7 and Table 4-2).

The relevant part of Table 4-2 is repeated below in Table 5-3. Listing 5-10 is an excerpt from Listing 4-7, demonstrating how the hook management functions are dispatched. Each of these functions is reviewed in the subsequent subsections.

TABLE 5-3. *IOCTL Hook Management Functions Supported by the Spy Device*

FUNCTION NAME	ID	IOCTL CODE	DESCRIPTION
SPY_IO_HOOK_INFO	11	0x8000602C	Returns info about Native API hooks
SPY_IO_HOOK_INSTALL	12	0x8000E030	Installs Native API hooks
SPY_IO_HOOK_REMOVE	13	0x8000E034	Removes Native API hooks
SPY_IO_HOOK_PAUSE	14	0x8000E038	Pauses/resumes the hook protocol
SPY_IO_HOOK_FILTER	15	0x8000E03C	Enables/disables the hook protocol filter
SPY_IO_HOOK_RESET	16	0x8000E040	Clears the hook protocol
SPY_IO_HOOK_READ	17	0x80006044	Reads data from the hook protocol
SPY_IO_HOOK_WRITE	18	0x8000E048	Writes data to the hook protocol

```
NTSTATUS SpyDispatcher (PDEVICE_CONTEXT pDeviceContext,
                        DWORD           dCode,
                        PVOID           pInput,
                        DWORD           dInput,
                        PVOID           pOutput,
                        DWORD           dOutput,
                        PDWORD          pdInfo)
    {
    SPY_MEMORY_BLOCK smb;
    SPY_PAGE_ENTRY   spe;
    SPY_CALL_INPUT   sci;
    PHYSICAL_ADDRESS pa;
    DWORD            dValue, dCount;
    BOOL             fReset, fPause, fFilter, fLine;
    PVOID            pAddress;
    PBYTE            pbName;
    HANDLE           hObject;
    NTSTATUS         ns = STATUS_INVALID_PARAMETER;

    MUTEX_WAIT (pDeviceContext->kmDispatch);

    *pdInfo = 0;

    switch (dCode)
        {
// =====================================================
// unrelated IOCTL functions omitted (cf. Listing 4-7)
// =====================================================
        case SPY_IO_HOOK_INFO:
            {
            ns = SpyOutputHookInfo (pOutput, dOutput, pdInfo);
            break;
            }
```

```
        case SPY_IO_HOOK_INSTALL:
            {
            if (((ns = SpyInputBool (&fReset,
                                     pInput, dInput))
                 == STATUS_SUCCESS)
                &&
                ((ns = SpyHookInstall (fReset, &dCount))
                 == STATUS_SUCCESS))
                {
                ns = SpyOutputDword (dCount,
                                     pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_REMOVE:
            {
            if (((ns = SpyInputBool (&fReset,
                                     pInput, dInput))
                 == STATUS_SUCCESS)
                &&
                ((ns = SpyHookRemove (fReset, &dCount))
                 == STATUS_SUCCESS))
                {
                ns = SpyOutputDword (dCount,
                                     pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_PAUSE:
            {
            if ((ns = SpyInputBool (&fPause,
                                    pInput, dInput))
                == STATUS_SUCCESS)
                {
                fPause = SpyHookPause (fPause);

                ns = SpyOutputBool (fPause,
                                    pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_FILTER:
            {
            if ((ns = SpyInputBool (&fFilter,
                                    pInput, dInput))
                == STATUS_SUCCESS)
                {
                fFilter = SpyHookFilter (fFilter);

                ns = SpyOutputBool (fFilter,
                                    pOutput, dOutput, pdInfo);
                }
```

(continued)

```
            break;
            }
        case SPY_IO_HOOK_RESET:
            {
            SpyHookReset ();
            ns = STATUS_SUCCESS;
            break;
            }
        case SPY_IO_HOOK_READ:
            {
            if ((ns = SpyInputBool (&fLine,
                                    pInput, dInput))
                == STATUS_SUCCESS)
                {
                ns = SpyOutputHookRead (fLine,
                                        pOutput, dOutput, pdInfo);
                }
            break;
            }
        case SPY_IO_HOOK_WRITE:
            {
            SpyHookWrite (pInput, dInput);
            ns = STATUS_SUCCESS;
            break;
            }
// ====================================================
// unrelated IOCTL functions omitted (cf. Listing 4-7)
// ====================================================
        }
    MUTEX_RELEASE (pDeviceContext->kmDispatch);
    return ns;
    }
```

LISTING 5-10. *Excerpt from the Spy Driver's Hook Command Dispatcher*

THE IOCTL FUNCTION SPY_IO_HOOK_INFO

The IOCTL Function `SPY_IO_HOOK_INFO` function fills a `SPY_HOOK_INFO` structure with information about the current state of the hook mechanism, as well as the system's SDT. This structure (Listing 5-11) contains or references various other structures introduced earlier:

- The `SERVICE_DESCRIPTOR_TABLE` is defined in Listing 5-1.
- `SPY_CALL` and `SPY_HOOK_ENTRY` are defined in Listing 5-2.
- `SPY_HEADER` and `SPY_PROTOCOL` are defined in Listing 5-9.

```
typedef struct _SPY_HOOK_INFO
    {
    SPY_HEADER                sh;
    PSPY_CALL                 psc;
    PSPY_PROTOCOL             psp;
    PSERVICE_DESCRIPTOR_TABLE psdt;
    SERVICE_DESCRIPTOR_TABLE  sdt;
    DWORD                     ServiceLimit;
    NTPROC                    ServiceTable  [SDT_SYMBOLS_MAX];
    BYTE                      ArgumentTable [SDT_SYMBOLS_MAX];
    SPY_HOOK_ENTRY            SpyHooks      [SDT_SYMBOLS_MAX];
    }
    SPY_HOOK_INFO, *PSPY_HOOK_INFO, **PPSPY_HOOK_INFO;

#define SPY_HOOK_INFO_ sizeof (SPY_HOOK_INFO)
```

LISTING 5-11. *Definition of the* `SPY_HOOK_INFO` *structure*

Be careful when evaluating the members of this structure. Some of them are pointers into kernel-mode memory that is not accessible from user-mode. However, you can use the spy device's `SPY_IO_MEMORY_DATA` function to examine the contents of these memory blocks.

THE IOCTL FUNCTION SPY_IO_HOOK_INSTALL

The IOCTL `SPY_IO_HOOK_INSTALL` function patches the service table of `ntoskrnl.exe` inside the system's SDT with the hook entry points stored in the global `aSpyHooks[]` array. This array is prepared by `SpyHookInitialize()` (Listing 5-5) and `SpyHookInitializeEx()` (Listing 5-3) during driver initialization. Each `aSpyHooks[]` entry comprises a hook entry point and a corresponding format string address, if available. The `SpyDispatcher()` calls the `SpyHookInstall()` helper function shown in Listing 5-12 to install the hooks. `SpyHookInstall()` in turn uses `SpyHookExchange()`, also included in Listing 5-12, to perform this task.

```
DWORD SpyHookExchange (void)
    {
    PNTPROC ServiceTable;
    BOOL    fPause;
    DWORD   i;
    DWORD   n = 0;

    fPause       = SpyHookPause (TRUE);
    ServiceTable = KeServiceDescriptorTable->ntoskrnl.ServiceTable;

    for (i = 0; i < SDT_SYMBOLS_MAX; i++)
```

(continued)

```
        {
        if (aSpyHooks [i].pbFormat != NULL)
            {
            aSpyHooks [i].Handler = (NTPROC)
                InterlockedExchange ((PLONG) ServiceTable+i,
                                     ( LONG) aSpyHooks [i].Handler);
            n++;
            }
        }
    gfSpyHookState = !gfSpyHookState;
    SpyHookPause (fPause);
    return n;
    }

// ---------------------------------------------------------------

NTSTATUS SpyHookInstall (BOOL   fReset,
                         PDWORD pdCount)
    {
    DWORD    n  = 0;
    NTSTATUS ns = STATUS_INVALID_DEVICE_STATE;

    if (!gfSpyHookState)
        {
        ghSpyHookThread = PsGetCurrentThreadId ();

        n = SpyHookExchange ();
        if (fReset) SpyHookReset ();

        ns = STATUS_SUCCESS;
        }
    *pdCount = n;
    return ns;
    }
```

LISTING 5-12. *Patching the System's API Service Table*

`SpyHookExchange()` is used both in the installation and removal of hooks, because it simply swaps the entries in the system's API service table and the `aSpyHooks[]` array. Therefore, calling this function twice restores the service table and the array to their original states. `SpyHookExchange()` loops through the `aSpyHooks[]` array and searches for entries that contain a format string pointer. The presence of such a string indicates that the function should be monitored. In this case, the API function pointer in the service table and the `Handler` member of the `aSpyHooks[]` entry are exchanged using the `ntoskrnl.exe` function `InterlockedExchange()`, which guarantees that no other thread can interfere

in this operation. The protocol mechanism is temporarily paused until the entire service table is patched. `SpyHookInstall()` is merely a wrapper around `SpyHookExchange()` that performs some additional actions:

- The service table is not touched if the global `gfSpyHookState` flag indicates that the hooks are already installed.
- The thread ID of the caller is written to the global variable `ghSpyHookThread`. The hook dispatcher inside `SpyHookInitializeEx()` uses this information to suppress all API calls originating from this thread. Otherwise, the hook protocol would be interrupted with irrelevant and distracting material as a result of the interaction of the spy device and its user-mode client.
- On request of the client, the protocol is reset. This means that all buffer contents are discarded and the handle directory is reinitialized.

The `SPY_IO_HOOK_INSTALL` function receives a Boolean input parameter from the caller. If `TRUE`, the protocol is reset after the hooks have been installed. This is the most frequently used option. Passing in `FALSE` continues a protocol eventually left over from a previous hook session. The return value of the function tells you how many service table entries were patched. On Windows 2000, `SPY_IO_HOOK_INSTALL` reports a value of 44, which is the number of entries in the format string array `apbSdtFormats[]` in Listing 5-6. On Windows NT 4.0, only 42 hooks are installed, because the API functions `NtNotifyChangeMultipleKeys()` and `NtQueryOpenSubKeys()` are not supported by this operating system version.

THE IOCTL FUNCTION SPY_IO_HOOK_REMOVE

The IOCTL `SPY_IO_HOOK_REMOVE` function is similar to `SPY_IO_HOOK_INSTALL`, because it basically reverses the actions of the latter. The IOCTL input and output arguments are identical. However, the `SpyHookRemove()` helper function called inside the `SpyDispatcher()` deviates in some important respects from `SpyHookInstall()`, as a comparison of Listing 5-12 and 5-13 reveals:

- The call is ignored if the global `gfSpyHookState` flag indicates that no hooks are currently installed.
- After the service table has been restored to its original state, the thread ID of the client that installed the hooks is cleared by setting the global variable `ghSpyHookThread` to zero.

- The most important extra feature is the `do/while` loop in the middle of Listing 5-13. In this loop, `SpyHookRemove()` tests whether other threads are currently serviced by the hook dispatcher by testing the `fInUse` members of all `SPY_CALL` structures inside the global `DEVICE_CONTEXT` structure. This is necessary because a client might attempt to unload the spy driver immediately after uninstalling the hooks. If this happens while some other processes' API calls are still within the hook dispatcher, the system throws an exception, followed by a Blue Screen. These in-use tests are performed in 100-msec intervals to give the other threads time to exit the spy device.

```
NTSTATUS SpyHookRemove (BOOL   fReset,
                        PDWORD pdCount)
    {
    LARGE_INTEGER liDelay;
    BOOL          fInUse;
    DWORD         i;
    DWORD         n  = 0;
    NTSTATUS      ns = STATUS_INVALID_DEVICE_STATE;

    if (gfSpyHookState)
        {
        n = SpyHookExchange ();
        if (fReset) SpyHookReset ();

        do  {
            for (i = 0; i < SPY_CALLS; i++)
                {
                if (fInUse = gpDeviceContext->SpyCalls [i].fInUse)
                    break;
                }
            liDelay.QuadPart = -1000000;
            KeDelayExecutionThread (KernelMode, FALSE, &liDelay);
            }
        while (fInUse);

        ghSpyHookThread = 0;

        ns = STATUS_SUCCESS;
        }
    *pdCount = n;
    return ns;
    }
```

LISTING 5-13. *Restoring the System's API Service Table*

Note that a final 100-msec delay is added even if all fInUse flags are clear. This precaution is required because a tiny security hole exists inside the hook dispatcher, just between the instruction where the fInUse flag of the current SPY_CALL entry is reset and the RET instruction where the dispatcher returns control to the caller (cf. Listing 5-2 between the ASM labels SpyHook8 and SpyHook9). If all fInUse flags are FALSE, there is a small probability that some threads have been suspended just before the RET instruction could be executed. Delaying the hook removal for another 100-msec interval should allow all threads time to leave this critical code sequence.

THE IOCTL FUNCTION SPY_IO_HOOK_PAUSE

The IOCTL SPY_IO_HOOK_PAUSE function, shown in Listing 5-14, allows a client to temporarily disable and reenable the hook protocol function. Essentially, it sets the global variable gfSpyHookPause to the Boolean value supplied by the client and returns its previous value, using the ntoskrnl.exe API function Interlocked Exchange(). By default, the protocol is enabled; that is, gfSpyHookPause is FALSE.

It is important to note that SPY_IO_HOOK_PAUSE works totally independent of SPY_IO_HOOK_INSTALL and SPY_IO_HOOK_REMOVE. If the protocol is paused while hooks are installed, the hooks remain in effect, but the hook dispatcher lets all API calls pass through without interference. You can also disable the protocol before installing the hooks, if you don't want the protocol to start automatically after SPY_IO_HOOK_INSTALL has patched the API service table. Note that the protocol is automatically reset when the protocol is resumed.

THE IOCTL FUNCTION SPY_IO_HOOK_FILTER

The IOCTL function SPY_IO_HOOK_FILTER manipulates a global flag, as shown in Listing 5-15. Here, the global flag gfSpyHookFilter is set to the client-supplied value, and the previous setting is returned. The default value is FALSE; that is, the filter is disabled.

```
BOOL SpyHookPause (BOOL fPause)
    {
    BOOL fPause1 = (BOOL)
                   InterlockedExchange ((PLONG) &gfSpyHookPause,
                                        ( LONG) fPause);
    if (!fPause) SpyHookReset ();
    return fPause1;
    }
```

LISTING 5-14. *Switching the Protocol On and Off*

```
BOOL SpyHookFilter (BOOL fFilter)
    {
    return (BOOL) InterlockedExchange ((PLONG) &gfSpyHookFilter,
                                       ( LONG) fFilter);
    }
```

LISTING 5-15. *Switching the Protocol Filter On and Off*

You already know the variable `gfSpyHookFilter` from the discussion of the `SpyWriteFilter()` function in Listing 5-8. If `gfSpyHookFilter` is `TRUE`, this function helps `SpyHookProtocol()` (see Listing 5-7) to drop all API calls that involve handles not previously registered by the spy device.

THE IOCTL FUNCTION SPY_IO_HOOK_RESET

The IOCTL `SPY_IO_HOOK_RESET` function resets the protocol mechanism to its original state, clearing the data buffer and discarding all registered handles. The `SpyHookReset()` function called by the `SpyDispatcher()` is merely a wrapper around `SpyWriteReset()`. Both functions are included in Listing 5-16. `SpyHookReset()` features additional serialization by means of the mutex calls `SpyHookWait()` and `SpyHookRelease()` (see Listing 5-7).

THE IOCTL FUNCTION SPY_IO_HOOK_READ

The API hook logger writes the protocol data to the `abData[]` buffer inside the global `SPY_PROTOCOL` structure shown in Listing 5-9. This byte array is designed as a circular buffer. That is, it features a pair of pointers for read and write access, respectively. Whenever one of the pointers moves past the end of the buffer, it is reset to the buffer base. The read pointer always tries to catch up with the write pointer, and if both point to the same location, the buffer is empty.

`SPY_IO_HOOK_READ` is by far the most important hook management function offered by the spy device. It reads arbitrary amounts of data from the protocol data buffer and adjusts the read pointer appropriately. This function should be called frequently while the protocol is enabled, to avoid buffer overflows. Listing 5-17 shows the function set handling this IOCTL request. The basic handlers are `SpyReadData()` and `SpyReadLine()`. The difference between them is that the former returns the requested amount of data, if available, whereas the latter retrieves single lines only. Line mode can be very convenient when the read data must be filtered by a client application. Callers of `SPY_IO_HOOK_READ` pass in a Boolean value that decides whether block mode (`FALSE`) or line mode (`TRUE`) is requested.

```
void SpyWriteReset (PSPY_PROTOCOL psp)
    {
    KeQuerySystemTime (&psp->sh.liStart);

    psp->sh.dRead    = 0;
    psp->sh.dWrite   = 0;
    psp->sh.dCalls   = 0;
    psp->sh.dHandles = 0;
    psp->sh.dName    = 0;
    return;
    }

// -----------------------------------------------------------------

void SpyHookReset (void)
    {
    SpyHookWait    ();
    SpyWriteReset  (&gpDeviceContext->SpyProtocol);
    SpyHookRelease ();
    return;
    }
```

LISTING 5-16. *Resetting the Protocol*

```
DWORD SpyReadData (PSPY_PROTOCOL psp,
                   PBYTE         pbData,
                   DWORD         dData)
    {
    DWORD i = psp->sh.dRead;
    DWORD n = 0;

    while ((n < dData) && (i != psp->sh.dWrite))
        {
        pbData [n++] = psp->abData [i++];
        if (i == SPY_DATA_BUFFER) i = 0;
        }
    psp->sh.dRead = i;
    return n;
    }

// -----------------------------------------------------------------

DWORD SpyReadLine (PSPY_PROTOCOL psp,
                   PBYTE         pbData,
                   DWORD         dData)
    {
    BYTE  b = 0;
```

(continued)

```
    DWORD i = psp->sh.dRead;
    DWORD n = 0;

    while ((b != '\n') && (i != psp->sh.dWrite))
        {
        b = psp->abData [i++];
        if (i == SPY_DATA_BUFFER) i = 0;
        if (n < dData) pbData [n++] = b;
        }
    if (b == '\n')
        {
        // remove current line from buffer
        psp->sh.dRead = i;
        }
    else
        {
        // don't return any data until full line available
        n = 0;
        }
    if (n)
        {
        pbData [n-1] = 0;
        }
    else
        {
        if (dData) pbData [0] = 0;
        }
    return n;
    }

// -----------------------------------------------------------------

DWORD SpyHookRead (PBYTE pbData,
                   DWORD dData,
                   BOOL  fLine)
    {
    DWORD n = 0;

    SpyHookWait ();

    n = (fLine ? SpyReadLine : SpyReadData)
            (&gpDeviceContext->SpyProtocol, pbData, dData);

    SpyHookRelease ();
    return n;
    }

// -----------------------------------------------------------------
```

```
NTSTATUS SpyOutputHookRead (BOOL   fLine,
                            PVOID  pOutput,
                            DWORD  dOutput,
                            PDWORD pdInfo)
    {
    *pdInfo = SpyHookRead (pOutput, dOutput, fLine);
    return STATUS_SUCCESS;
    }
```

LISTING 5-17. *Reading from the Protocol Buffer*

The `SpyOutputHookRead()` and `SpyHookRead()` functions are trivial. `SpyHookRead()` adds the usual mutex serialization and chooses between `SpyReadLine()` and `SpyReadData()`, and `SpyOutputHookRead()` postprocesses its results as demanded by the IOCTL framework.

THE IOCTL FUNCTION SPY_IO_HOOK_WRITE

The IOCTL `SPY_IO_HOOK_WRITE` function allows the client to write data to the protocol buffer. An application can use this feature to add separators or additional status information to the protocol. The implementation is shown in Listing 5-18. `SpyHookWrite()` is yet another wrapper with additional mutex serialization. The `SpyWriteData()` function it calls is the basic protocol generator of the spy device. All `SpyWrite*()` helper functions (e.g., the `SpyWriteFormat()`, `SpyWriteNumber()`, `SpyWriteChar()`, and `SpyWriteLarge()` functions used by `SpyHookProtocol()` in Listing 5-7) are ultimately built upon it.

```
DWORD SpyWriteData (PSPY_PROTOCOL psp,
                    PBYTE         pbData,
                    DWORD         dData)
    {
    BYTE  b;
    DWORD i = psp->sh.dRead;
    DWORD j = psp->sh.dWrite;
    DWORD n = 0;

    while (n < dData)
        {
        psp->abData [j++] = pbData [n++];
        if (j == SPY_DATA_BUFFER) j = 0;

        if (j == i)
            {
            // remove first line from buffer
```

(continued)

```
            do  {
                b = psp->abData [i++];
                if (i == SPY_DATA_BUFFER) i = 0;
                }
            while ((b != '\n') && (i != j));

            // remove half line only if single line
            if ((i == j) &&
                ((i += (SPY_DATA_BUFFER / 2)) >= SPY_DATA_BUFFER))
                {
                i -= SPY_DATA_BUFFER;
                }
            }
        }
    psp->sh.dRead  = i;
    psp->sh.dWrite = j;
    return n;
    }

// -----------------------------------------------------------------

DWORD SpyHookWrite (PBYTE pbData,
                    DWORD dData)
    {
    DWORD n = 0;

    SpyHookWait ();

    n = SpyWriteData
            (&gpDeviceContext->SpyProtocol, pbData, dData);

    SpyHookRelease ();
    return n;
    }
```

LISTING 5-18. *Writing to the Protocol Buffer*

Note how `SpyWriteData()` handles overflow situations. If the read pointer advances slowly, the write pointer may lap it. In this situation, two options are available:

1. Write access is disabled until the read pointer is advanced.
2. Buffered data is discarded to make space.

The spy device chooses the second option. If an overflow occurs, the entire protocol line at the current read pointer position is dropped by advancing the read pointer to the next line. If the buffer contains just a single line (which is highly

improbable), only the first half of the line is discarded. The code handling these situations is marked in Listing 5-18 by appropriate comments.

A SAMPLE HOOK PROTOCOL READER

To help you write your own API hook client applications, I have added a very simple sample application that reads the hook protocol buffer and displays it in a console window. The pause, filter, and reset functions can be issued by pressing keys **P, F,** and **R** on the keyboard, and the output can be filtered according to a series of user-specified function name patterns. The application is called "SBS Windows 2000 API Hook Viewer," and its source code is available on the book's companion CD in the directory tree `\src\w2k_hook`.

CONTROLLING THE SPY DEVICE

For convenience, the `w2k_hook.exe` application uses a couple of simple wrappers for the various `SPY_IO_HOOK_*` IOCTL functions, summarized in Listing 5-19. These utility functions make the code much more readable and minimize the probability of parameter errors during the development of a spy device client application.

```
BOOL WINAPI SpyIoControl (HANDLE hDevice,
                          DWORD  dCode,
                          PVOID  pInput,
                          DWORD  dInput,
                          PVOID  pOutput,
                          DWORD  dOutput)
    {
    DWORD dInfo = 0;

    return DeviceIoControl (hDevice, dCode,
                            pInput,  dInput,
                            pOutput, dOutput,
                            &dInfo,  NULL)
           &&
           (dInfo == dOutput);
    }

// -----------------------------------------------------------------

BOOL WINAPI SpyVersionInfo (HANDLE            hDevice,
                            PSPY_VERSION_INFO psvi)
    {
    return SpyIoControl (hDevice, SPY_IO_VERSION_INFO,
                         NULL, 0,
```

(continued)

```
                         psvi, SPY_VERSION_INFO_);
    }

// ----------------------------------------------------------------

BOOL WINAPI SpyHookInfo (HANDLE         hDevice,
                         PSPY_HOOK_INFO pshi)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_INFO,
                         NULL, 0,
                         pshi, SPY_HOOK_INFO_);
    }

// ----------------------------------------------------------------

BOOL WINAPI SpyHookInstall (HANDLE hDevice,
                            BOOL   fReset,
                            PDWORD pdCount)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_INSTALL,
                         &fReset, BOOL_,
                         pdCount, DWORD_);
    }

// ----------------------------------------------------------------

BOOL WINAPI SpyHookRemove (HANDLE hDevice,
                           BOOL   fReset,
                           PDWORD pdCount)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_REMOVE,
                         &fReset, BOOL_,
                         pdCount, DWORD_);
    }

// ----------------------------------------------------------------

BOOL WINAPI SpyHookPause (HANDLE hDevice,
                          BOOL   fPause,
                          PBOOL  pfPause)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_PAUSE,
                         &fPause, BOOL_,
                         pfPause, BOOL_);
    }

// ----------------------------------------------------------------
```

```
BOOL WINAPI SpyHookFilter (HANDLE hDevice,
                           BOOL   fFilter,
                           PBOOL  pfFilter)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_FILTER,
                         &fFilter, BOOL_,
                         pfFilter, BOOL_);
    }

// ----------------------------------------------------------------

BOOL WINAPI SpyHookReset (HANDLE hDevice)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_RESET,
                         NULL, 0,
                         NULL, 0);
    }

// ----------------------------------------------------------------

DWORD WINAPI SpyHookRead (HANDLE hDevice,
                          BOOL   fLine,
                          PBYTE  pbData,
                          DWORD  dData)
    {
    DWORD dInfo;

    if (!DeviceIoControl (hDevice, SPY_IO_HOOK_READ,
                          &fLine, BOOL_,
                          pbData, dData,
                          &dInfo, NULL))
        {
        dInfo = 0;
        }
    return dInfo;
    }

// ----------------------------------------------------------------

BOOL WINAPI SpyHookWrite (HANDLE hDevice,
                          PBYTE  pbData)
    {
    return SpyIoControl (hDevice, SPY_IO_HOOK_WRITE,
                         pbData, lstrlenA (pbData),
                         NULL,   0);
    }
```

LISTING 5-19. *Device I/O Control Utility Functions*

Before the functions in Listing 5-19 can be used, the spy device must be loaded and started. This operation is much the same as that outlined in Chapter 4 in conjunction with the memory spy application `w2k_mem.exe`. Listing 5-20 shows the application's main function, `Execute()`, which loads and unloads the spy device driver, opens and closes a device handle, and interacts with the device via IOCTL. If you compare Listing 5-20 to Listing 4-29, the similarities at the beginning and end are obvious. Only the middle sections, where the application-dependent code is located, are different.

```
void WINAPI Execute (PPWORD ppwFilters,
                     DWORD  dFilters)
    {
    SPY_VERSION_INFO svi;
    SPY_HOOK_INFO    shi;
    DWORD            dCount, i, j, k, n;
    BOOL             fPause, fFilter, fRepeat;
    BYTE             abData [HOOK_MAX_DATA];
    WORD             awData [HOOK_MAX_DATA];
    WORD             awPath [MAX_PATH] = L"?";
    SC_HANDLE        hControl          = NULL;
    HANDLE           hDevice           = INVALID_HANDLE_VALUE;

    _printf (L"\r\nLoading \"%s\" (%s) ...\r\n",
             awSpyDisplay, awSpyDevice);

    if (w2kFilePath (NULL, awSpyFile, awPath, MAX_PATH))
        {
        _printf (L"Driver: \"%s\"\r\n",
                 awPath);

        hControl = w2kServiceLoad (awSpyDevice, awSpyDisplay,
                                   awPath, TRUE);
        }
    if (hControl != NULL)
        {
        _printf (L"Opening \"%s\" ...\r\n",
                 awSpyPath);

        hDevice = CreateFile (awSpyPath,
                              GENERIC_READ    | GENERIC_WRITE,
                              FILE_SHARE_READ | FILE_SHARE_WRITE,
                              NULL, OPEN_EXISTING,
                              FILE_ATTRIBUTE_NORMAL, NULL);
        }
    else
        {
```

```
        _printf (L"Unable to load the spy device driver.\r\n");
        }
    if (hDevice != INVALID_HANDLE_VALUE)
        {
        if (SpyVersionInfo (hDevice, &svi))
            {
            _printf (L"\r\n"
                     L"%s V%lu.%02lu ready\r\n",
                     svi.awName,
                     svi.dVersion / 100, svi.dVersion % 100);
            }
        if (SpyHookInfo (hDevice, &shi))
            {
            _printf (L"\r\n"
                     L"API hook parameters:       0x%08lX\r\n"
                     L"SPY_PROTOCOL structure:    0x%08lX\r\n"
                     L"SPY_PROTOCOL data buffer:  0x%08lX\r\n"
                     L"KeServiceDescriptorTable:  0x%08lX\r\n"
                     L"KiServiceTable:            0x%08lX\r\n"
                     L"KiArgumentTable:           0x%08lX\r\n"
                     L"Service table size:        0x%lX (%lu)\r\n",
                     shi.psc,
                     shi.psp,
                     shi.psp->abData,
                     shi.psdt,
                     shi.sdt.ntoskrnl.ServiceTable,
                     shi.sdt.ntoskrnl.ArgumentTable,
                     shi.ServiceLimit, shi.ServiceLimit);
            }
        SpyHookPause  (hDevice, TRUE, &fPause ); fPause  = FALSE;
        SpyHookFilter (hDevice, TRUE, &fFilter); fFilter = FALSE;

        if (SpyHookInstall (hDevice, TRUE, &dCount))
            {
            _printf (L"\r\n"
                     L"Installed %lu API hooks\r\n",
                     dCount);
            }
        _printf (L"\r\n"
                 L"Protocol control keys:\r\n"
                 L"\r\n"
                 L"P    -  pause  ON/off\r\n"
                 L"F    -  filter ON/off\r\n"
                 L"R    -  reset protocol\r\n"
                 L"ESC  -  exit\r\n"
                 L"\r\n");

        for (fRepeat = TRUE; fRepeat;)
            {
```

(continued)

```
            if (n = SpyHookRead (hDevice, TRUE,
                                 abData, HOOK_MAX_DATA))
                {
                if (abData [0] == '-')
                    {
                    n = 0;
                    }
                else
                    {
                    i = 0;
                    while (abData [i] && (abData [i++] != '='));

                    j = i;
                    while (abData [j] && (abData [j] != '(')) j++;

                    k = 0;
                    while (i < j) awData [k++] = abData [i++];

                    awData [k] = 0;

                    for (i = 0; i < dFilters; i++)
                        {
                        if (PatternMatcher (ppwFilters [i], awData))
                            {
                            n = 0;
                            break;
                            }
                        }
                    }
                if (!n) _printf (L"%hs\r\n", abData);
                Sleep (0);
                }
            else
                {
                Sleep (HOOK_IOCTL_DELAY);
                }
            switch (KeyboardData ())
                {
                case 'P':
                    {
                    SpyHookPause (hDevice, fPause, &fPause);
                    SpyHookWrite (hDevice, (fPause ? abPauseOff
                                                   : abPauseOn));
                    break;
                    }
                case 'F':
                    {
                    SpyHookFilter (hDevice, fFilter, &fFilter);
                    SpyHookWrite  (hDevice, (fFilter ? abFilterOff
                                                     : abFilterOn));
```

```
                    break;
                    }
                case 'R':
                    {
                    SpyHookReset (hDevice);
                    SpyHookWrite (hDevice, abReset);
                    break;
                    }
                case VK_ESCAPE:
                    {
                    _printf (L"%hs\r\n", abExit);
                    fRepeat = FALSE;
                    break;
                    }
                }
            }
        if (SpyHookRemove (hDevice, FALSE, &dCount))
            {
            _printf (L"\r\n"
                     L"Removed %lu API hooks\r\n",
                     dCount);
            }
        _printf (L"\r\nClosing the spy device ...\r\n");
        CloseHandle (hDevice);
        }
    else
        {
        _printf (L"Unable to open the spy device.\r\n");
        }
    if ((hControl != NULL) && gfSpyUnload)
        {
        _printf (L"Unloading the spy device ...\r\n");
        w2kServiceUnload (awSpyDevice, hControl);
        }
    return;
    }
```

LISTING 5-20. *The Main Application Framework*

Note that the `Execute()` function in Listing 5-20 requests `GENERIC_READ` and `GENERIC_WRITE` access in the `CreateFile()` call, whereas the function in Listing 4-29 uses only `GENERIC_READ` access. The reason for this discrepancy is buried in the IOCTL codes used by these applications. Whereas the memory spy in Chapter 4 uses read-only functions throughout, the API hook viewer discussed here calls functions that modify system data and hence require a device handle with additional write access. If you examine the IOCTL codes in the third column of Table 5-3, you can see that most of them have the hex digit `E` at the fourth position from the right, whereas `SPY_IO_HOOK_INFO` and `SPY_IO_HOOK_READ` have the digit `6` there. According to Figure 4-6 in Chapter 4, this means that the latter pair of hook management functions require a device handle with read access, whereas the remaining ones require

read/write rights. The designer of a device driver must decide which read/write access right combinations are demanded by the I/O requests handled by the device. Patching the system's API service table is a radical write operation, so urging a client to obtain a handle with write access is certainly appropriate.

Most of the remaining code in Listing 5-20 should be self-explaining. Following are features that are worth noting:

- The `SPY_IO_HOOK_READ` function is operated in line mode, as the second argument of the `SpyHookRead()` call at the beginning of the big `for` loop shows.
- The user of the application can specify a series of pattern strings with embedded wildcards `'*'` and `'?'` on the command line. These patterns are compared sequentially with the function name within each protocol line using the helper function `PatternMatcher()` shown in Listing 5-21. If no pattern matches the name, the line is suppressed. To view the hook protocol unfiltered, the command `w2k_hook` * must be issued.
- After handling a protocol line, the application returns the rest of its time slice to the system by calling `Sleep (0),` so the time is available for other processes.
- If no protocol data is available, the application suspends itself for 10 msec `(HOOK_IOCTL_DELAY)` before polling the spy device again. This reduces the CPU load considerably in times with low usage of the Native API.
- In the main loop, the keyboard is polled as well. All keys except **P, F, R,** and **Esc** are ignored. **P** switches the pause mode on and off (default: on), **F** enables and disables filtering by handle (default: enabled), **R** resets the protocol, and **Esc** terminates the application.
- If one of the **P, F, R,** or **Esc** keys is pressed, a separator line is written to the hook protocol buffer using the `SPY_IO_HOOK_WRITE` function. This line indicates the state change resulting from the entered command. Writing the separator to the buffer is better than writing it directly to the console window because the state change might appear on the screen with some delay. For example, if the **P** key is pressed to halt the display, the application will continue to generate output until all data has been read from the protocol buffer. The separator generated by the **P** command will be appended after the last entry, so it appears at the correct location.
- Just like the `w2k_mem.exe` application in Chapter 4, `w2k_hook.exe` unloads the spy device only if the global flag `gfSpyUnload` is set. By default, it is *not* set—for the reasons explained in Chapter 4.

```
BOOL WINAPI PatternMatcher (PWORD pwFilter,
                            PWORD pwData)
    {
    DWORD i, j;

    i = j = 0;
    while (pwFilter [i] && pwData [j])
        {
        if (pwFilter [i] != '?')
            {
            if (pwFilter [i] == '*')
                {
                i++;
                if ((pwFilter [i] != '*') && (pwFilter [i] != '?'))
                    {
                    if (pwFilter [i])
                        {
                        while (pwData [j] &&
                               (!PatternMatcher (pwFilter + i,
                                                 pwData   + j)))
                            {
                            j++;
                            }
                        }
                    return (pwData [j]);
                    }
                }
            if ((WORD) CharUpperW ((PWORD) (pwFilter [i])) !=
                (WORD) CharUpperW ((PWORD) (pwData   [j])))
                {
                return FALSE;
                }
            }
        i++;
        j++;
        }
    if (pwFilter [i] == '*') i++;
    return !(pwFilter [i] || pwData [j]);
    }
```

LISTING 5-21. *A Simple String Pattern Matcher*

The examples shown in Figures 5-6 and 5-7 were generated by `w2k_hook.exe` with the name patterns `*file` and `ntclose` specified on the command line. This filters out all file management function calls plus `NtClose()`. It is important to keep in mind that the name patterns are applied to the protocol data *after* it has been generated, whereas the “garbage” filter of the spy device based on registered handles manipulates the protocol *before* it is written. If you exclude protocol entries by specifying name patterns on the `w2k_hook.exe` command line, this has absolutely no

effect on the protocol data generator. The only effect is that protocol entries are thrown away after having been retrieved from the protocol buffer.

Highlights and Pitfalls

The API hooking mechanism of Russinovich and Cogswell (Russinovich and Cogswell 1997) adapted here is clearly ingenious and elegant. The following are its most notable advantages:

- Installing and uninstalling a hook in the system's API service table is a simple pointer exchange operation.
- After the hook is installed, it receives the Native API calls of all processes running in the system, even of new ones started after the hook installation.
- Because the hook device runs in kernel-mode, it has maximum access to all system resources. It is even allowed to execute privileged CPU instructions.

The following are problem areas I encountered during the development of my spy device:

- The hook device must be designed and written with extreme care. Because all traffic occurring on the Native API level will pass through in the context of various application threads, it must be as stable as the operating system kernel itself. The smallest oversight may immediately crash the system.
- Only a small part of the kernel's API traffic is logged. For example, API calls originating from other kernel-mode modules don't pass through the system's `INT 2Eh` gate and hence don't appear in the hook protocol. Also, many important functions exported by `ntdll.dll` and `ntoskrnl.exe` are not part of the Native API, so they cannot be hooked in the service table.

The incomplete API coverage is clearly more restrictive than the demand for stability. Anyway, it is amazing how much useful data can be gained about the internals of an application by tracing its Native API calls. For example, I was able to gain deep insight into the NetWare Core Protocol (NCP) operations performed by Microsoft's NetWare redirector `nwrdr.sys` by simply observing its `NtFsControlFile()` traffic. Therefore, this approach to API monitoring is certainly the most proficient of the alternatives available to date for Windows 2000.