문서타입	공개
공개유형	공개

About Handle in Windows system

최초 작성자	노용환 (fixbrain@gmail.com)
최초 작성일	2006-02-15
최종 수정일	2006-02-15
버전	1.0.0.0

목차

문서 개정이력

개정 날짜	개정 내용	개정인	비고
2006-02-15	최초 작성	노 용 환	1.0.0.0
2006-02-28	PspCidTable 구조 정리 추가	노 용 환	1.0.0.1

1. 요약

이 문서에서는 Windows 에서 사용하는 HANDLE 이 무엇이며 Windows 커널이 이를 관리하는 방식에 설명한다.
또한 Handle 을 이해하기 위해서는 필수적인 Windows 의 object 관리 방식에 대한 설명도 함께 할 것이다.

2. Object Manager

윈도우 커널이 관리하는 프로세스, 이벤트, 파일 객체등을 일컬어 Object 라 한다.
개념적인 내용은 각자 알아서 공부하도록 하고, 이 Object 는 executive object 와 kernel object 의 두가지 종류가 있다.

아래의 내용은 Windows Internals 4th edition 의 Chapter 3 의 내용과 실제 xp sp2 가 적용된 시스템에서 테스트한 내용이다.

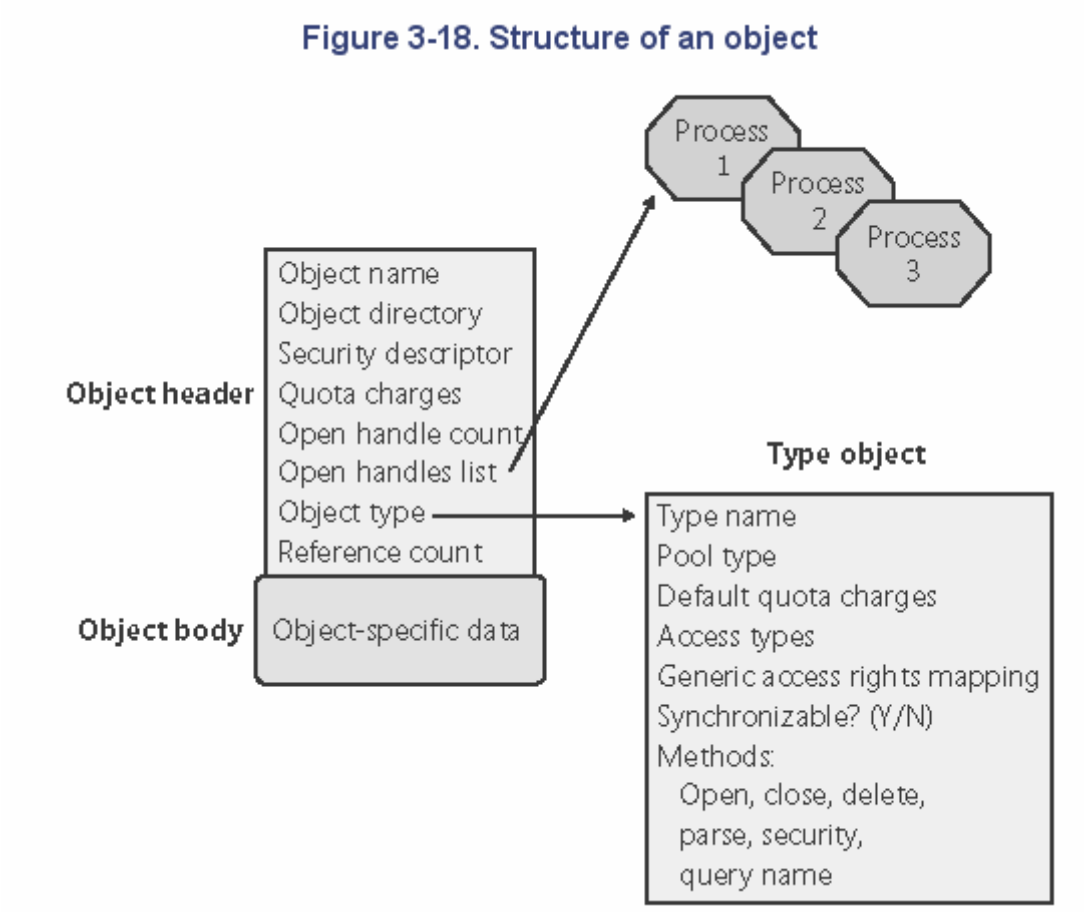
2.1. Executive Object

Executive object 의 종류

Object Type	Represents
Symbolic link	A mechanism for referring to an object name indirectly.
Process	The virtual address space and control information necessary for the execution of a set of thread objects.
Thread	An executable entity within a process.
Job	A collection of processes manageable as a single entity through the job.
Section	A region of shared memory (known as a file mapping object in Windows).
File	An instance of an opened file or an I/O device.
Access token	The security profile (security ID, user rights, and so on) of a process or a thread.

Object Type	Represents
Event	An object with a persistent state (signaled or not signaled) that can be used for synchronization or notification.
Semaphore	A counter that provides a resource gate by allowing some maximum number of threads to access the resources protected by the semaphore.
Mutex*	A synchronization mechanism used to serialize access to a resource. (Kernel mode 에서는 mutant 임)
Timer	A mechanism to notify a thread when a fixed period of time elapses.
IoCompletion	A method for threads to enqueue and dequeue notifications of the completion of I/O operations (known as an I/O completion port in the Windows API).
Key	A mechanism to refer to data in the registry. Although keys appear in the object manager namespace, they are managed by the configuration manager, in a way similar to that in which file objects are managed by file system drivers. Zero or more key values are associated with a key object; key values contain data about the key.
WindowStation	An object that contains a clipboard, a set of global atoms, and a group of desktop objects.
Desktop	An object contained within a window station. A desktop has a logical display surface and contains windows, menus, and hooks.

2.2. Object structure



오브젝트는 Object header 와 Object body 로 구성된다.

Object body 는 object type 에 따라서 내용이 바뀐다.

Table 3-4. Standard Object Header Attributes	
Attribute	Purpose
Object name	Makes an object visible to other processes for sharing
Object directory	Provides a hierarchical structure in which to store object names
Security descriptor	Determines who can use the object and what they can do with it (Note: it might be null for objects without a name.)

Table 3-4. Standard Object Header Attributes	
Attribute	Purpose
Quota charges	Lists the resource charges levied against a process when it opens a handle to the object
Open handle count	Counts the number of times a handle has been opened to the object
Open handles list	Points to the list of processes that have opened handles to the object (not present for all objects)
Object type	Points to a type object that contains attributes common to objects of this type
Reference count	Counts the number of times a kernel-mode component has referenced the address of the object

Object type 에 따라 유니크한 object body 를 가지는데 몇가지 공통적인 object service 가 있다.
비록 모든 오브젝트가 generic object service 를 지원하긴 하지만 각각의 오브젝트들은 자신만의 create, open, query 등의 서비스를 가지고 있다.
예를 들어 I/O 시스템의 경우 file object 에 대한 create file 서비스나 process manager 는 process object 에 대한 process create 서비스를 가지고 있는 것이다.

Table 3-5. Generic Object Services	
Service	Purpose
Close	Closes a handle to an object
Duplicate	Shares an object by duplicating a handle and giving it to another process
Query object	Gets information about an object's standard attributes
Query security	Gets an object's security descriptor
Set security	Changes the protection on an object

Table 3-5. Generic Object Services	
Service	Purpose
Wait for a single object	Synchronizes a thread's execution with one object
Wait for multiple objects	Synchronizes a thread's execution with multiple objects

2.2.1. Type Object

Viewing Object Header and Type Objects

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 817bd9c8  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1000cf0  HandleCount: 238.
    Image: System
```

```
kd> !object 817bd9c8
Object: 817bd9c8  Type: (817bd040) Process
    ObjectHeader: 817bd9b0
    HandleCount: 2  PointerCount: 60
```

```
kd> dt _OBJECT_HEADER
   +0x000 PointerCount     : Int4B
   +0x004 HandleCount      : Int4B
   +0x004 NextToFree       : Ptr32 Void
   +0x008 Type             : Ptr32 _OBJECT_TYPE
   +0x00c NameInfoOffset   : UChar
   +0x00d HandleInfoOffset : UChar
   +0x00e QuotaInfoOffset  : UChar
   +0x00f Flags            : UChar
   +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body             : _QUAD

kd> dt _OBJECT_HEADER 817bd9b0
```

```
   +0x000 PointerCount     : 60
   +0x004 HandleCount      : 2
   +0x004 NextToFree       : 0x00000002
   +0x008 Type             : 0x817bd040
   +0x00c NameInfoOffset   : 0 ''
   +0x00d HandleInfoOffset : 0 ''
   +0x00e QuotaInfoOffset  : 0 ''
   +0x00f Flags            : 0x22 '"'
   +0x010 ObjectCreateInfo : 0x80562c80
   +0x010 QuotaBlockCharged : 0x80562c80
   +0x014 SecurityDescriptor : 0xe10016e4
   +0x018 Body             : _QUAD

kd> dt _OBJECT_TYPE 817bd040
   +0x000 Mutex            : _ERESOURCE
   +0x038 TypeList         : _LIST_ENTRY [ 0x817bd078 - 0x817bd078 ]
   +0x040 Name             : _UNICODE_STRING "Process"
   +0x048 DefaultObject    : (null)
   +0x04c Index            : 5
   +0x050 TotalNumberOfObjects : 0x15
   +0x054 TotalNumberOfHandles : 0x4e
   +0x058 HighWaterNumberOfObjects : 0x15
   +0x05c HighWaterNumberOfHandles : 0x4f
   +0x060 TypeInfo         : _OBJECT_TYPE_INITIALIZER
   +0x0ac Key              : 0x636f7250
   +0x0b0 ObjectLocks      : [4] _ERESOURCE

kd> dt _OBJECT_TYPE_INITIALIZER 817bd040+60
   +0x000 Length           : 0x4c
   +0x002 UseDefaultObject : 0 ''
   +0x003 CaseInsensitive  : 0 ''
   +0x004 InvalidAttributes : 0xb0
   +0x008 GenericMapping   : _GENERIC_MAPPING
   +0x018 ValidAccessMask  : 0x1f0fff
   +0x01c SecurityRequired : 0x1 ''
   +0x01d MaintainHandleCount : 0 ''
```

```
+0x01e MaintainTypeList : 0 ''
+0x020 PoolType         : 0 ( NonPagedPool )
+0x024 DefaultPagedPoolCharge : 0x1000
+0x028 DefaultNonPagedPoolCharge : 0x290
+0x02c DumpProcedure    : (null)
+0x030 OpenProcedure    : (null)
+0x034 CloseProcedure   : (null)
+0x038 DeleteProcedure  : 0x8058c87d     nt!PspProcessDelete+0
+0x03c ParseProcedure   : (null)
+0x040 SecurityProcedure : 0x8056c71e     nt!SeDefaultObjectMethod+0
+0x044 QueryNameProcedure : (null)
+0x048 OkayToCloseProcedure : (null)
```

오브젝트 매너저가 user mode 를 위한 어떠한 서비스도 제공하지 않기 때문에 Type Object 는 user mode 에서 접근하는 것이 불가능하다.
그러나 Table 3.6 에 정의된 속성들에 대해서는 native service (windows api)를 통해 접근 가능하다.

Table 3-6. Type Object Attributes

Attribute	Purpose
Type name	The name for objects of this type ("process," "event," "port," and so on)
Pool type	Indicates whether objects of this type should be allocated from paged or nonpaged memory
Default quota charges	Default paged and nonpaged pool values to charge to process quotas
Access types	The types of access a thread can request when opening a handle to an object of this type ("read," "write," "terminate," "suspend," and so on)
Generic access rights mapping	A mapping between the four generic access rights (read, write, execute, and all) to the type-specific access rights

Table 3-6. Type Object Attributes	
Attribute	Purpose
Synchronization	Indicates whether a thread can wait for objects of this type
Methods	One or more routines that the object manager calls automatically at certain points in an object's lifetime

2.2.2. Object Methods

Table 3-7 은 object Method 를 나타낸다.
Object Method 들은 객체가 생성되거나 소멸될 때 자동으로 호출되어지는 C++ 의 생성자나 소멸자처럼 동작한다. Object Manager 는 이 개념을 좀 더 확장해서 다른 상황에서도 Object method 가 호출되도록 만든다.

Table 3-7. Object Methods	
Method	When Method Is Called
Open	When an object handle is opened
Close	When an object handle is closed
Delete	Before the object manager deletes an object
Query	name When a thread requests the name of an object, such as a file, that exists in a secondary object namespace
Parse	When the object manager is searching for an object name that exists in a secondary object namespace
Security	When a process reads or changes the protection of an object, such as a file, that exists in a secondary object namespace

2.2.3. Object Handles and the Process Handle Table

프로세스가 O 이름을 통해서 Object 를 생성하거나 open 하면 Handle 을 리턴한다. 이 Handle 은 커널 객체를 나타내는 값인데 이 핸들을 통해서 Object 에 접근하는 것이 이름을 통해서 접근하는 것보다 빠르다. (이름 찾기를 안해도 되니..)
유저 모드 프로세스는 object 에 접근하기 위해 반드시 Handle 이 필요하다.
그렇지 않으면 object 에 접근할 방법이 없기 때문이다. 커널 모드 프로세스는 object 에 직접 접근할 수 있지만 Object 매니저에게 해당 Object 를 사용할 것이라는 것을 알려줘야 한다.
(ObReferenceObjectByPointer, ObDereferenceObject.. 등등의 함수로..)

Object Handle 을 사용하는 것은 몇가지 추가적인 이점이 있다.
첫째로 파일 핸들이든, 이벤트 핸들이든, 프로세스 핸들이든 간에 동일한 인터페이스를 제공한다는 것이다.

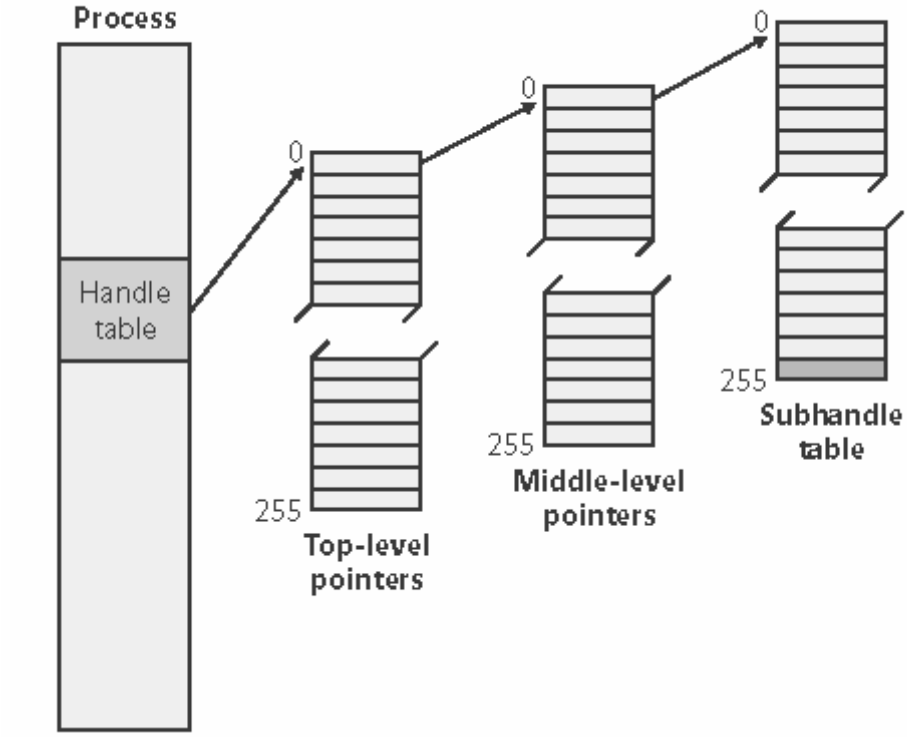
둘째로 object manager 는 handle 을 생성하고, handle 이 참조하는 object 를 배치하는데 독자적인 권한을 가질 수 있다는 의미이다.

Object handle 은 executive process block (EPROCESS) 가 가리키고 있는 handle table 의 인덱스이다.

프로세스의 핸들테이블은 프로세스가 open 하고 있는 모든 object 들에 대한 포인터들을 담고 있다.
Handle Table 은 x86 의 MMU(Memory Management Unit) 과 유사한 구조의 3 단계 구조를 가지고 있다.

Window 2000 에서는 프로세스가 생성될 때 object manasger 는 top level handle table 을 생성한다. Top level handle table 은 middle level table 들에 대한 포인터를 담고 있다.
Middle level table 은 subhandle tables 를 가리키는 포인터들의 배열을 가지고 있다.
마지막 lowest level 은 첫번째 subhandle table 을 가지고 있다.
Figure 3-20 은 windows 2000 의 핸들 테이블 아키텍쳐를 표현하고 있다.

Figure 3-20. Windows 2000 process handle table architecture



Windows 2000 에서는 object manager 는 object handle 값의 하위 24 비트를 8 비트 필드로 나누어 각 필드값들을 3 레벨 테이블의 인덱스로 사용한다.

Windows XP 와 Windows Server 2003 은 프로세스가 생성될 때 lowest level handle table 만을 생성하며 다른 레벨의 handle table **은 필요에 의해** 생성한다.

Windows 2000 에서는 subhandle table 은 255 개의 사용가능한 엔트리로 구성된다.

Windows XP 와 Windows Server 2003 에서는 subhandle table 은 PAGE -1 에 가능한한 많은 엔트리로 구성되며 이것은 handle auditing 에 사용된다.
예를 들어, x86 시스템에서 페이지는 4096 바이트이며 이는 handle table entry (8byte) 사이즈로 나누면 512 인데 여기서 1 을 빼면 전부 511 개의 엔트리가 lowest level handle table 에 존재하게 되는 것이다.

Windows XP 와 Windows Server 2003 에서는 mid-level handle table 은 subhandle table 포인터의 full page 로 구성된다. 따라서 subhandle tables 의 개수는 page 사이즈와 플랫폼의 pointer 사이즈에 의존적이다.

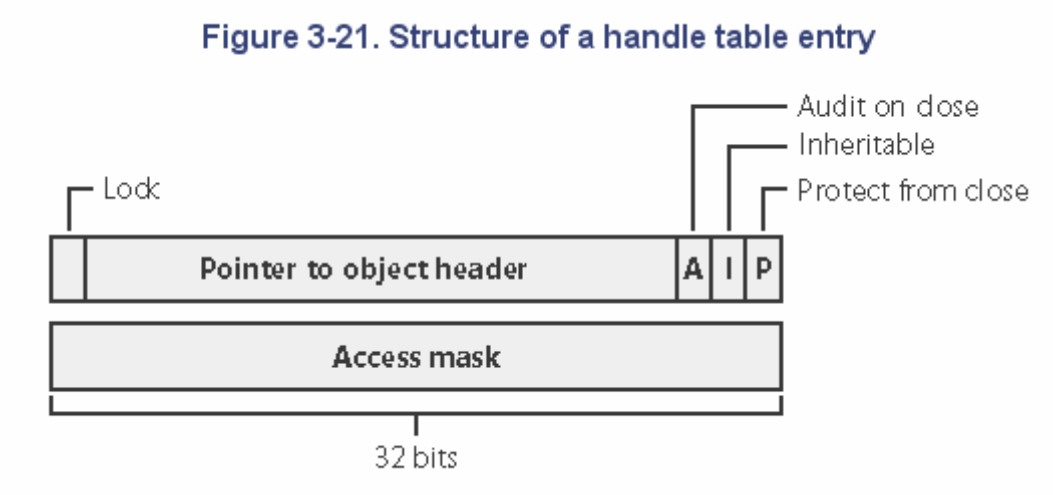


Figure 3-21 에서 보듯이 x86 시스템에서 각각의 handle table entry 는 object 에 대한 포인터(flag 포함)와 granted access mask, 두개의 32 비트 멤버로 구성된다.
64 비트 시스테에서는 handle table entry 는 12 바이트 이며, 64 비트 짜리 object header 포인터와 32 비트 access mask 로 구성된다.
Windows 2000 에서 첫번째 32 비트 멤버는 object header 에 대한 포인터와 4 개의 플래그로 구성된다.
Object header 는 항상 8 바이트로 정렬(align)되므로 하위 3 비트는 flag 로 자유롭게 사용될 수 있다.
엔트리의 최상위 1 비트는 Lock 을 위한 비트이다. 오브젝트 매니저가 handle 을 object pointer 로 변환할 때 이 handle entry 가 변환중이라는 표시를 위해서 이 플래그를 세팅한다.
모든 오브젝트는 시스템 영역에 존재하기 때문에 최상위 비트는 세팅되어야 한다. (0x80000000 보다 수치적으로 높은 영역은 /3GB 부트 스위치가 켜져있다 하더라도 보장되기 때문에..)
그러므로 오브젝트 매니저는 object 의 정확한 포인터 값을 얻기 위해 이 비트를 세팅할 수 있는거다.
오브젝트 매니저는 handle table lock 을 통해서 프로세스의 모든 핸들 테이블을 잠궈야 하는 경우가 있다. - 프로세스가 새로운 핸들을 생성하거나 존재하는 핸들을 close 하는 경우 -

Windows xp 와 windows server 2003 의 경우 lock 비트는 object pointer 의 low-order 비트이다. Windows 2000 에서 low-order bit 에 존재하던 flag 는 access mask 의 unused bit 에 있게 된다.
첫번째 flag 는 caller 가 이 핸들을 닫을수 있는지 없는지를 가리킨다.

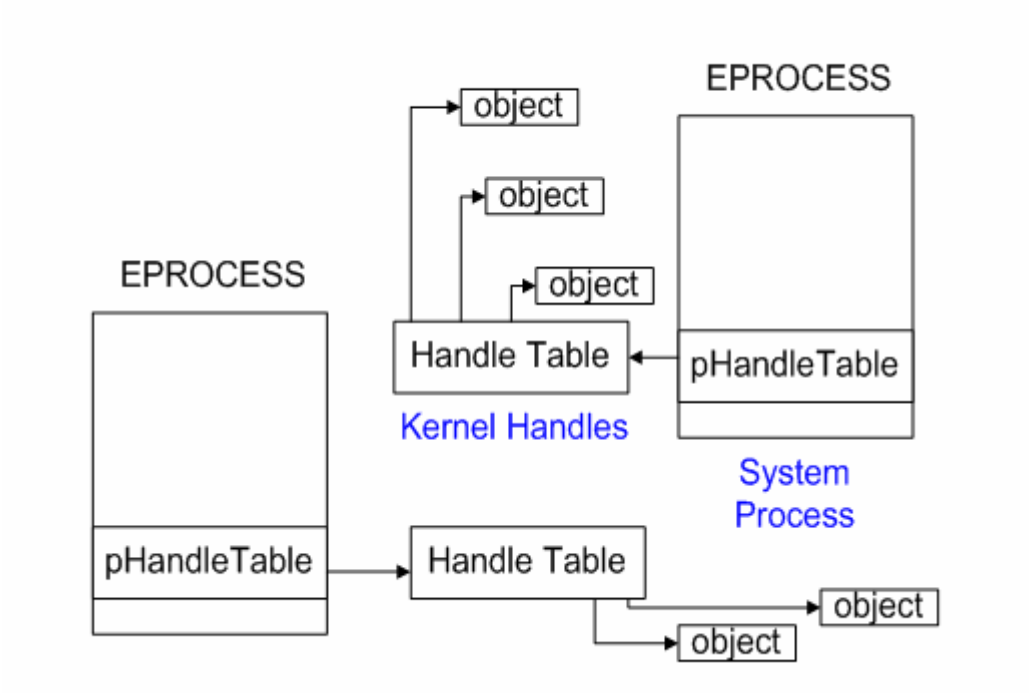
두번째 flag 는 이 프로세스에 의해 생성되는 프로세스가 이 프로세스의 핸들테이블의 복사본을 받을 것인지를 나타낸다. (inherit)
이미 얘기했듯이 handle 의 상속(inheritance) 는 핸들을 생성하거나 SetHandleInformation 함수를 통해서 명시될 수 있다.
세번째 flag 는 object 를 close 할때 audit message 를 생성할지 말지를 가리키며 이 플래그는 windows 가 노출하지 않는 정보이며 내부적으로만 사용된다.

System components and device drivers often need to open handles to objects that user-mode applications shouldn't have access to. This is done by creating handles in the *kernel handle table* (referenced internally with the name ObpKernelHandleTable). The handles in this table are accessible only from kernel mode and in any process context.
This means that a kernelmode function can reference the handle in any process context with no performance impact. The object manager recognizes references to handles from the kernel handle table when the high bit of the handle is set-that is, when references to kernel-handle-table handles have values greater than 0x80000000. On Windows 2000, the kernel-handle table is an independent handle table, but on Windows XP and Windows Server 2003 the kernel-handle table also serves as the handle table for the System process.

3. Handle

Handle 은 x86 시스템에서 다들 잘 알다시피 포인터와 같은 단순한 숫자에 불과하지만 Windows object manager 는 이 숫자값을 HANDLE Table 의 인덱스로 사용하므로써 참조하고자 하는 커널 객체에 대한 빠른 참조를 가능하게 한다.

3.1. Handle 과 커널 오브젝트의 관계



[그림 3.1]

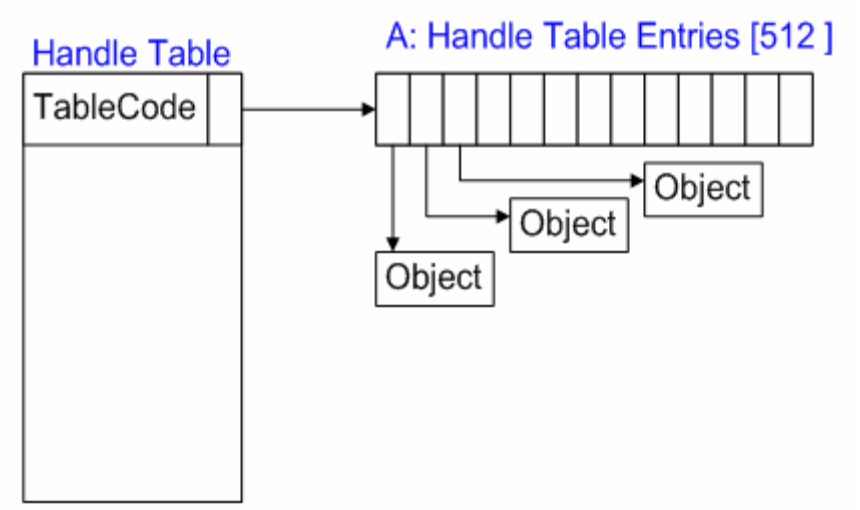
그림 3.1 은 프로세스의 핸들테이블과 커널 오브젝트간의 관계를 간략히 표현한 것이다.

windows 는 커널 오브젝트들을 Handle table 이라는 특별한 관리구조를 통해서 관리하며 Handle table 로부터 특정 커널 오브젝트를 참조하기 위한 key 가 바로 핸들이라는 것이다.
이게 핵심이다 !!

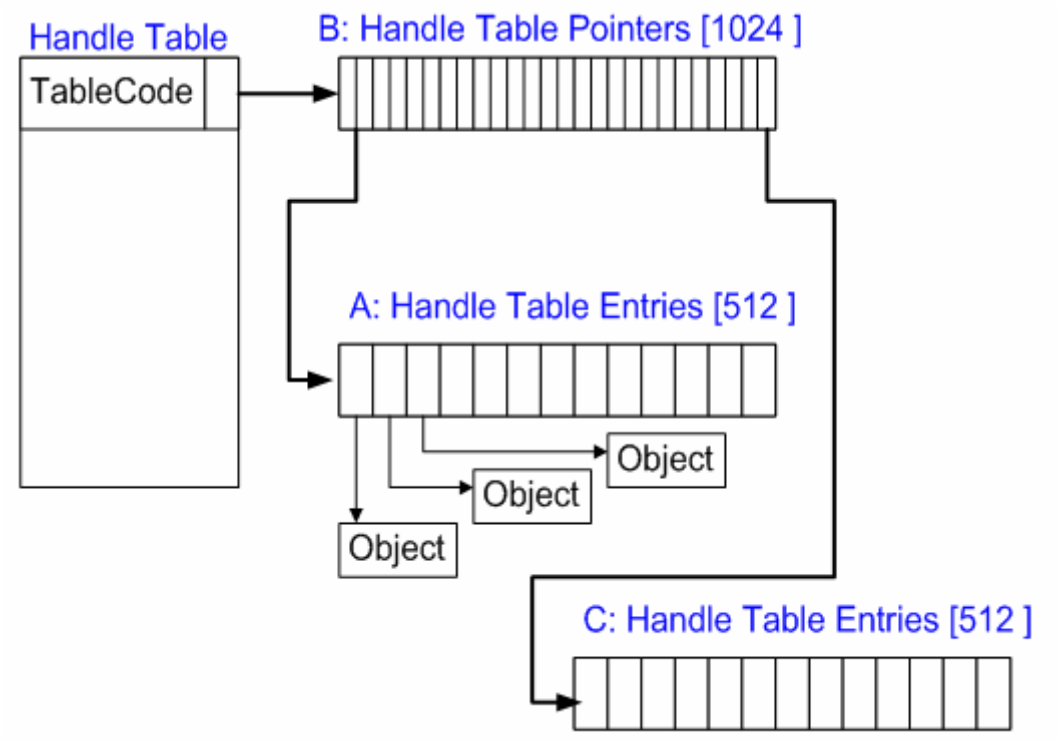
3.2. 핸들과 3 level handle table 간의 관계

2.2.3 에서 설명한 것 처럼 핸들 테이블은 3 layer 구조로 구성된다.
따라서 그림 3.1 에서 Handle Table 로부터 object 로의 이행은 분명히 3 단계의

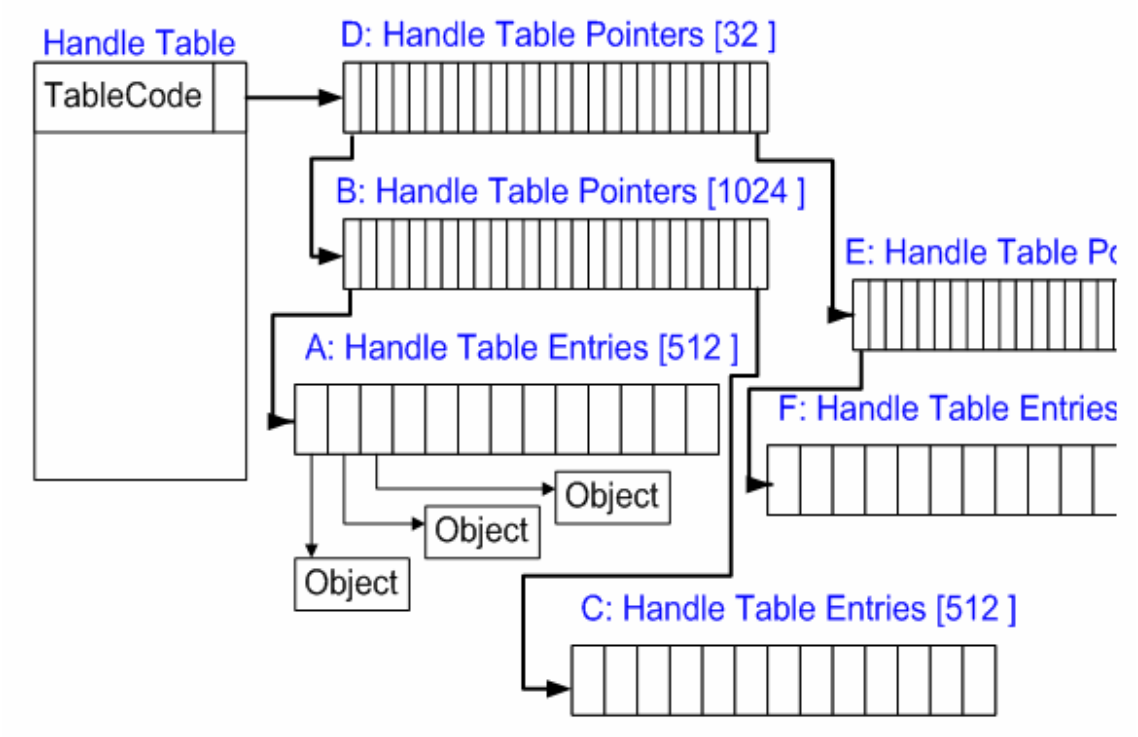
레이어를 찾아 가는 과정이 있을 것이다.
또한 System process 들이 참조하는 핸들 테이블을 따로 존재하는데 (그림 3.1 에서 보듯이) 이에 대한 설명은 ObpKernelHandleTable 에 대한 디버깅과정에서 확인 할 수 있을 것이다.
(개인적인 디버깅에 의한 결과물이므로.. 틀릴 수도 있다. -_-)



[그림 3.1.1] One level handle table



[그림 3.1.2] Two level handle table



[그림 3.1.3] three level handle table

위의 그림들을 보면 object manager 가 handle table 을 관리하는 방식에 대해 이해할 수 있을 것이다.
또한 위의 그림은 xp, 2003 시스템 기준으로 작성한 것이다.
Win 2k 같은 경우는 핸들 테이블 엔트리가 모두 255 개로 고정되어있다.
또한 xp 와 2k 의 _HANDLE_TABLE_ENTRY 와 _HANDLE_TABLE 구조체는 서로 약간씩 다르다. 2k 의 경우 _HANDLE_TABLE 구조체에 _HANDLE_TABLE_ENTRY 에 대한 포인터가 있으나 xp 의 경우 이 필드가 존재하지 않으며 TableCode 라는 필드를 통해서 핸들 테이블 엔트리로 접근해야 한다.
이 내용에 대해서는 실제 디버깅에 의해 설명할 것이다.

참고로 어떤 프로세스가 Event object 를 사용하는 경우 해당 프로세스의 _EPROCESS 의 ObjectTable 필드를 참조해서 핸들테이블을 찾은 후 위의 그림의 순서대로 object 를 참조하게 된다.
(아마 맞을것이다..실제로 디버깅을 해보지는 않았지만 … ^^)

4. ObpKernelHandleTable - WinXP sp2

Window nt 커널은 ObpKernelHandleTable 이라는 심볼을 익스포트 하고 있다.
ObpKernelHandleTable 은 _HANDLE_TABLE 구조체 포인터 이며 이 심볼은 system process 가 참조하는 object 들에 대한 핸들테이블 엔트리를 관리하고 있다.

4.1. Examine System Process's Handle Table

Kd 명령중에 !process 명령을 통해서 Idle 프로세스의 _EPROCESS 를 보자.
!process 를 인자 없이 실행하면 현재 프로세스가 나온다는데 뭐 Idle 프로세스가 나와서 일단 넘어간다. (!process 0 0 하면 안나오네??)

```
kd> !process
PROCESS 8055b580  SessionId: none  Cid: 0000    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1000cf0  HandleCount: 238.
    Image: Idle

kd> !process 4 0
Searching for Process with Cid == 4
PROCESS 817bd9c8  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1000cf0  HandleCount: 238.
    Image: System
```

이상한 점은 Idle 프로세스와 System 프로세스가 분명히 다른 프로세스이긴 하지만 동일한 오브젝트 테이블을 참조하고 있으며 동일한 DirBase 값을 가진다는 것이다.
결국 DirBase 가 같다는 것은 동일한 가상 메모리 공간을 가진다고 볼 수 있으므로 ..
암튼 그렇다. 심심하면 봐야겠다.

- Examine Idle process

```
kd> dd 8055b580+c4 l 1
8055b644  e1000cf0
kd> dt _HANDLE_TABLE e1000cf0
   +0x000 TableCode        : 0xe1002000
   +0x004 QuotaProcess     : (null)
   +0x008 UniqueProcessId  : 0x00000004
---생략
   +0x038 NextHandleNeedingPool : 0x800
```

+0x03c HandleCount : 238

- Examine System Process

```
kd> dd 817bd9c8+c4 l 1
817bda8c  e1000cf0
```

Idle process 와 System 프로세스의 _EPROCESS.ObjectTable 을 추적해 보면 동일한 위치다.
위에서도 알수있듯이 당연한 결과다.

4.2. Examine ObpKernelHandleTable

```
kd> x nt!ObpKernelHandleTable
805622b8 nt!ObpKernelHandleTable = <no type information>
kd> dd 805622b8 l 1
805622b8  e1000cf0
```

ObpKernelHandleTable 은 _HANDLE_TABLE 의 포인터이므로 해당 위치가 가리키는 곳을 덤프해야 한다.
결과에서도 알 수 있듯이 ObpKernelHandleTable 이나 System, Idle 프로세스의 HandleTable 이 동일함을 확인 할 수 있다.

결과적으로 Windows 2k 이상의 Windows 는 system process 가 사용하는 핸들 테이블에 대해서 별도의 심볼(ObpKernelHandleTable)을 통해 사용할 수 있도록 하고 있음을 유추할 수 있다.

5. PspCidTable in Windows XP sp2

Win2k 커널에는 또 따른 핸들 테이블이 존재한다.
PspCidTable 이 그 테이블인데 이것은 모든 프로세스/스레드의 CID (Client ID) 를 위한 핸들 테이블이다.
이제부터 PspCidTable 에 대한 분석을 시작해 보자.
CID 를 관리하는 테이블이니 OpenProcess() 함수가 가장 적당한 함수일 것이다.

5.1. Win32 API OpenProcess

OpenProcess() 는 Kernel32.dll 이 export 하는 유저레벨 api 이다.
이 함수는 PID 값을 통해 해당 프로세스의 핸들을 리턴하는 함수이다.
다시 말하면 OpenProcess() 는 프로세스의 PID 로 커널이 관리하는 오브젝트, 즉 프로세스 오브젝트를 찾아갈 수 있는 핸들테이블의 인덱스를 리턴하는 함수라고 할 수 있을 것이다.

어쨌든 Kernel32.OpenProcess() 를 호출하면 Ntdll.NtOpenProcess() 가 호출하게 된다. ntdll.NtOpenProcess() 는 Fast System Call/SW Interrupt 등의 방식을 통해 권한상승을 시도할 것이고, ntoskrnl.NtOpenProcess() 함수가 호출된다 .
자세한 내용은 Technical report - Windows system call 분석 문서를 참조하기 바란다.

5.2. ntoskrnl.ZwOpenProcess() / NtOpenProcess()

5.2.1. ntoskrnl.ZwOpenProcess( )

```
nt!ZwOpenProcess:
804df724 b87a000000      mov     eax,0x7a
804df729 8d542404        lea     edx,[esp+0x4]
804df72d 9c              pushfd
804df72e 6a08            push    0x8
804df730 e871170000      call    nt!KiSystemService (804e0ea6)
804df735 c21000          ret     0x10
```

알다시피 ZwOpenProcess 는 Fast call stub 이다. 실제 시스템 콜에 대한 구현은 NtXXX() 함수를 봐야 한다.
KiSystemService() 를 트레이싱 하다 보면 NtOpenProcess() 를 호출한다.

5.2.2. ntoskrnl.NtOpenProcess()

```
805766a8 8d45dc          lea     eax,[ebp-0x24]
805766ab 50              push    eax
805766ac ff75d4          push    dword ptr [ebp-0x2c]
805766af e871000000      call    nt!PsLookupProcessByProcessId (80576725)
805766b4 8bf8            mov     edi,eax
805766b6 3bfe            cmp     edi,esi
805766b8 0f8cb9f40200    jl      nt!NtOpenProcess+0x1ed (805a5b77)
805766be 8d45e0          lea     eax,[ebp-0x20]
```

NtOpenProcess() 를 트레이싱하다가 보면 PsLookupProcessByProcessId() 함수를 호출한다.

```
PAGE:004A21C4 loc_4A21C4:                ; CODE XREF: NtOpenProcess+1836D↓j
PAGE:004A21C4                    cmp     byte ptr [ebp-1Ah], 0
```

```
PAGE:004A21C8                   jnz     loc_52AD07
PAGE:004A21CE                   cmp     byte ptr [ebp-19h], 0
PAGE:004A21D2                   jz      loc_50F4F4
PAGE:004A21D8                   mov     [ebp-30h], esi
PAGE:004A21DB                   cmp     [ebp-28h], esi
PAGE:004A21DE                   jnz     loc_508913
PAGE:004A21E4                   lea     eax, [ebp-24h]
PAGE:004A21E7                   push    eax             ; struct _EPROCESS pointer
PAGE:004A21E8                   push    dword ptr [ebp-2Ch] ; PID
PAGE:004A21EB                   call    PsLookupProcessByProcessId
PAGE:004A21F0
```

위의 코드는 NtOpenProcess() 를 리버스 엔지니어링 하는 내용이다. (주석은 내가 달아놓은 것이다)
004A21E7 에서 _EPROCESS pointer 와 PID 값을 push 하고, undocumented api 인 PsLookupProcessByPid() 함수를 호출한다.

5.2.3. PsLookupProcessByProcessID()

PsLookupProcessByProcessId() 가 호출된 시점의 스택은 아래와 같을 것이다. (__stdcall 인 경우)

HIGHER ADDRESS

_EPROCESS pointer	Ebp + 0xc
PID	Ebp + 0x8
Ret	Ebp + 0x4
Ebp	Ebp + 0x0
Local 변수 공간	Ebp - 0x?

LOWSER ADDRESS

```
PAGE:004A2057                                   ; sub_584D43+3F↓p
PAGE:004A2057
PAGE:004A2057 arg_0           = dword ptr  8    ; IN HANDLE ProcessId
PAGE:004A2057 arg_4           = dword ptr  0Ch  ; OUT PEPROCESS *Process
PAGE:004A2057
PAGE:004A2057 ; FUNCTION CHUNK AT PAGE:0052D52C SIZE 00000011 BYTES
```

```
PAGE:004A2057
PAGE:004A2057                   mov     edi, edi
PAGE:004A2059                   push    ebp
PAGE:004A205A                   mov     ebp, esp
PAGE:004A205C                   push    ebx
PAGE:004A205D                   push    esi
PAGE:004A205E                   mov     eax, large fs:124h
PAGE:004A2064                   push    [ebp+arg_0]             // PID
PAGE:004A2067                   mov     esi, eax
PAGE:004A2069                   dec     dword ptr [esi+0D4h]
PAGE:004A206F                   push    dword_491660
                                ; push    dword ptr [nt!PspCidTable (80562ce0)]
PAGE:004A2075                   call    sub_49702D
                                ; call    nt!ExMapHandleToPointer (8056861e)
PAGE:004A207A                mov     ebx, eax
```

PsLookupProcessByProcessId() 를 다시 디스어셈블 해보면 exported 되지 않은 함수인 ExMapHandleToPointer() 함수를 호출함을 볼 수 있다.
ExMapHandleToPointer() 의 파라미터를 유심히 볼 필요가 있다.
PID 와 ntoskrnl.PspCidTable 을 인자로 받아들임을 확인 할 수 있다.
함수 이름이나, 인자, 그리고 리턴된 후의 코드를 살펴보면 ExMapHandleToPointer() 함수는 PspCidTable 을 통해서 Pid 가 유효한 Object (process object) 인지 확인하고, 유효하다면 핸들 포인터를 리턴하는 함수가 아닐까.. 한다. (아님 말고. -_-)

5.2.4. ExMapHandleToPointer()

HIGHER ADDRESS

PID	Ebp + 0xc
PspCidTable	Ebp + 0x8
Ret	Ebp + 0x4
Ebp	Ebp + 0x0
Local 변수 공간	Ebp - 0x?

LOWSER ADDRESS

```
PAGE:0049702D  sub_49702D          proc  near                ;  CODE  XREF:
```

```
sub_4973AE+13↓p
PAGE:0049702D                                   ; sub_497961+3C↓p ...
PAGE:0049702D
PAGE:0049702D arg_0           = dword ptr  8        ; PspCidTable
PAGE:0049702D arg_4           = dword ptr  0Ch      ; PID
PAGE:0049702D
PAGE:0049702D ; FUNCTION CHUNK AT PAGE:004AB9E2 SIZE 00000007 BYTES
PAGE:0049702D ; FUNCTION CHUNK AT PAGE:004BA395 SIZE 00000012 BYTES
PAGE:0049702D ; FUNCTION CHUNK AT PAGE:0050D133 SIZE 00000014 BYTES
PAGE:0049702D ; FUNCTION CHUNK AT PAGE:005392B3 SIZE 0000005C BYTES
PAGE:0049702D
PAGE:0049702D                 mov     edi, edi
PAGE:0049702F                 push    ebp
PAGE:00497030                 mov     ebp, esp
PAGE:00497032                 push    edi
PAGE:00497033                 mov     edi, [ebp+arg_4] ; PID
PAGE:00497036                 test    di, 7FCh
// pid 의 하위 16 비트와  0x07fc 를 and  연산
// 0000 0011 1001 1000               398
// 0000 0111 1111 1100               7fc
// ---------------------------------------- TEST
// 0000 0011 1001 1000               ---- ZF = 0
// 이게 뭘까?
//
PAGE:0049703B                 jz      loc_4AB9E2            // 함수 종료
PAGE:00497041                 push    ebx
PAGE:00497042                 mov     ebx, [ebp+arg_0]
PAGE:00497045                 push    esi
PAGE:00497046                 push    edi     // PID
PAGE:00497047                 push    ebx     // PspCidTable
PAGE:00497048                 call    loc_4944F1 // ExpLookupHandleTableEntry()
PAGE:0049704D                 mov     esi, eax
PAGE:0049704F                 test    esi, esi
PAGE:00497051                 jz      loc_4BA395
```

5.2.5. ExpLookupHandleTableEntry()

HIGHER ADDRESS

PID	Ebp + 0xc
PspCidTable	Ebp + 0x8
Ret	Ebp + 0x4
Ebp	Ebp + 0x0
Local 변수 공간	Ebp - 0x?

LOWSER ADDRESS

Arg_8 : CID

Arg_4 : PspCidTable

```
PAGE:004944F1 loc_4944F1:                             ; CODE XREF: sub_494530+1A↓p
PAGE:004944F1                                         ; sub_49702D+1B↓p ...
PAGE:004944F1                 mov     edi, edi
PAGE:004944F3                 push    ebp
PAGE:004944F4                 mov     ebp, esp
// CID && 0x0FFFFFFFC
// -> CID && 11111111111111111111111111111100
// ==> 하위 두 비트 클리어
// ➔ xp 에서는 하위 두 비트를 통해서 핸들 테이블 레벨값으로 사용됨을 확인
PAGE:004944F6                 and     [ebp+arg_8], 0FFFFFFFCh
PAGE:004944FA                 mov     eax, [ebp+arg_8]
PAGE:004944FD                 mov     ecx, [ebp+arg_4]
PAGE:00494500                 mov     edx, [ebp+arg_8]
// CID >> 2
//      => 최하위 두비트를 쉬프트
//      ➔ 하위 두 비트를 제외한 나머지 30 비트가 첫번째 핸들 테이블 엔트리
//              포인터로 사용됨
PAGE:00494503                 shr     eax, 2
// PspCidTable + 0x38 과 Cid 비교
//      ➔  +0x038 NextHandleNeedingPool : Uint4B
//      cmp     cid, PspCidTable.NextHandleDeedingPool
```

```
PAGE:00494506                   cmp     edx, [ecx+38h]
PAGE:00494509                   jnb     loc_49DE45
PAGE:0049450F                   push    esi
PAGE:00494510                   mov     esi, [ecx]
PAGE:00494512                   mov     ecx, esi
PAGE:00494514                   and     ecx, 3
PAGE:00494517                   and     esi, 0FFFFFFFCh
PAGE:0049451A                   sub     ecx, 0
PAGE:0049451D                   jnz     loc_496EDA
PAGE:00494523                   lea     eax, [esi+eax*8]
PAGE:00494526
PAGE:00494526 loc_494526:                                       ; CODE XREF:
NlsLeadByteInfo+2A0D↑j
PAGE:00494526                   pop     esi
PAGE:00494527
PAGE:00494527 loc_494527:                                       ; CODE XREF:
NlsLeadByteInfo+9963↑j
PAGE:00494527                   pop     ebp
PAGE:00494528                   retn    8
```

ExMapHandleToPointer() 함수는 다시 ExpLookupHandleTableEntry() 함수를 통해서 PspCidTable 로부터 PID, 즉 CID 의 존재를 검사하고, 유효한 object 가 존재하면 object 의 _HANDLE_TABLE_ENTRY 를 리턴한다.

```
8057675d 83bfa401000000  cmp     dword ptr [edi+0x1a4],0x0
80576764 7414            jz      nt!PsLookupProcessByProcessId+0x55 (8057677a)
80576766 8bcf            mov     ecx,edi
80576768 e83c5df7ff      call    nt!ObReferenceObjectSafe (804ec4a9)
8057676d 84c0            test    al,al
8057676f 7409            jz      nt!PsLookupProcessByProcessId+0x55 (8057677a)
80576771 8b450c          mov     eax,[ebp+0xc]
```

만일 정상적으로 핸들이 존재한다면 object 를 레퍼런싱 하고, 그에 대한 포인터를 리턴할 것이다.

5.3. Examine PspCidTable

앞의 리버스 엔지니어링을 통해서 PspCidTable 에는 분명히 모든 프로세스/스레드 Cid 관련 정보가 있다는 것을 알 수 있었다.

처음에는 이 PspCidTable 이 _HANDLE_TABLE 인지, _HANDLE_TABLE_ENTRY 에 대한 포인터인지 확신이 안서서 무척 어려웠으나 삽질끝에 알아낸 결과는 _HANDLE_TABLE 에 대한 포인터라는 것이다. (지금 생각하면 당연히 _HANDLE_TABLE 인데..TT)
또한 인터넷이나 관련 서적들은 대부분 win2k 의 _HANDLE_TABLE, _HANDLE_TABLE_ENTRY 에 대한 설명만 있다.
문제는 XP, 2003 은 2k 와는 좀 달라서 상당히 애먹었다.

Non exported api 인 ExpLookupHandleTableEntry() 함수가 PspCidTable 로부터 handle 을 가지고, _HANDLE_TABLE_ENTRY 포인터를 리턴하는 함수이다.
즉 얘가 가장 분석해야 할 함수라는 거다.

```
kd> x nt!PspCidTable
80562ce0 nt!PspCidTable = <no type information>
kd> dd nt!PspCidTable l 1
80562ce0  e1000890
kd> dt _HANDLE_TABLE e1000890
   +0x000 TableCode        : 0xe1003000
   +0x004 QuotaProcess     : (null)
   +0x008 UniqueProcessId  : (null)
   +0x00c HandleTableLock  : [4] _EX_PUSH_LOCK
   +0x01c HandleTableList  : _LIST_ENTRY [ 0xe10008ac - 0xe10008ac ]
   +0x024 HandleContentionEvent : _EX_PUSH_LOCK
   +0x028 DebugInfo        : (null)
   +0x02c ExtraInfoPages   : 0
   +0x030 FirstFree        : 0x720
   +0x034 LastFree         : 0x590
   +0x038 NextHandleNeedingPool : 0x800
   +0x03c HandleCount      : 287
   +0x040 Flags            : 1
   +0x040 StrictFIFO       : 0y1
```

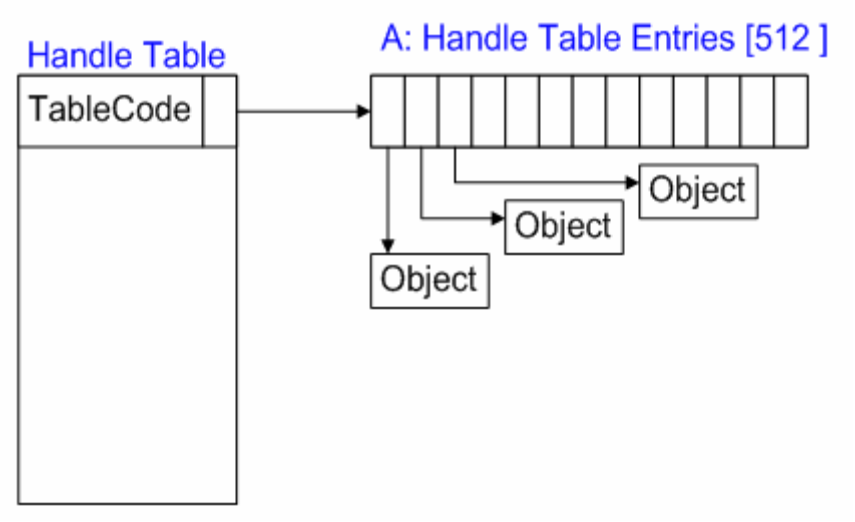
위의 덤프에서도 알 수 있듯이 _EPROCESS 의 HANDLE_TABLE 과는 좀 다르다.
일단 QuotaProcess, UniqueProcessId 가 없고, HandleTableList 도 없다.
결국 PspCidTable 은 좀 튀는 모양의 _HANDLE_TABLE 이다.
2k 같은 경우 _HANDLE_TABLE 구조체 안에 _HANDLE_TABLE_ENTRY 에 대한 포인터가

있어서 이를 통해서 3 layered handle table 을 추적해 나가면 object header 를 찾아갈 수 있는데 xp 에는 아예 _HANDLE_TABLE_ENTRY 가 없기 때문에 뭘 참조해야 하는지 몰랐다.

XP 의 경우 TableCode 필드가 handle table 의 인덱스 값이다.
앞에서 나온 핸들이나 핸들 테이블에 대한 개념이 확실하지 않으면 이해 하기 어려울 수 있으니 정신차리고 봐야 할 것이다.

```
kd> dd nt!PspCidTable l 1
80562ce0  e1000890
kd> dt _HANDLE_TABLE e1000890
   +0x000 TableCode        : 0xe1003000
   +0x004 QuotaProcess     : (null)
   +0x008 UniqueProcessId  : (null)
   +0x00c HandleTableLock  : [4] _EX_PUSH_LOCK
   +0x01c HandleTableList  : _LIST_ENTRY [ 0xe10008ac - 0xe10008ac ]
   +0x024 HandleContentionEvent : _EX_PUSH_LOCK
   +0x028 DebugInfo        : (null)
   +0x02c ExtraInfoPages   : 0
   +0x030 FirstFree        : 0x720
   +0x034 LastFree         : 0x590
   +0x038 NextHandleNeedingPool : 0x800
   +0x03c HandleCount      : 287
   +0x040 Flags            : 1
   +0x040 StrictFIFO       : 0y1
```

PspCidTable 의 TableCode 는 0xe1003000 이고, HandleCount 는 287 이다.



[그림 3.1.1] One level handle table

[그림 3.1.1] 을 참조해 보면 TableOfCode 값이 _HANDLE_TABLE_ENTRY 를 가리키고 있다.

리버싱을 통해 내린 결론은 TableOfCode 의 하위 2 비트가 handle table index 로 사용되고 있었다. (windows internals 책에서 나온 내용과는 좀 다르다. -_-)
그림 3.1.1 이 가장 정확한 내용이다.

_HANDLE_TABLE.TableOfCode 하위 2 비트를 통해서 subhandle table 인덱스인지, middle level index 인지, top level 인덱스 인지 판단하게 된다.

```
#define HANDLE_TABLE_LEVEL_MASK        0x03
switch (_HANDLE_TABLE.TableOfCode & HANDLE_TABLE_LEVEL_MASK )
{
Case 0: Subhandle table index;
Case 1: Middle level index;
Case 2: Top level Index;
}
```

이런 식이다.

그리고 _HANDLE_TABLE.TableOfCode 의 하위 2 비트를 제외한 값이 Handle Table Entry 배열의 포인터이다. (그림 3.1.1 에서 처럼)

Windows xp, 2003 의 경우 subhandle entry 는 앞에서 언급했듯이 PAGE_SIZE 가 허용하는 만큼 생성된다고 했다.
즉 (PAGE_SIZE / sizeof(_HANDLE_TABLE_ENTRY)) -1 만큼의 Handle Table Entry 가 존재한다는 것이다.

PspCidTable 의 경우 TableOfCode = 0xe1003000 이므로
0xe1003000 & 0x03 = 0x00
이므로 subhandle table index 로 쓰인다는 것을 알 수 있다.
이것은 [그림 3-1] 의 경우와 완전히 동일한 경우이다.
따라서 _HANDLE_TABLE_ENTRY 배열의 시작 주소는 HandleTable->TableCode 의 하위 두비트를 클리어 시키면 얻을 수 있다.

```
PHANDLE_TABLE_ENTRY pEntry =
        (PHANDLE_TABLE_ENTRY) ((HandleTable->TableCode & ~ TABLE_LEVEL_MASK));
```

따라서
0xe1003000 & ~ 0x3 = 0xe1003000
이므로 0xE1003000 이 sub handle table 배열의 시작 주소값이 된다.

```
kd> dd 0xe1003000
e1003000  00000000 fffffffe 817bd9c9 00000000
e1003010  817bd751 00000000 817bd309 00000000
e1003020  817bc021 00000000 817bcda9 00000000
e1003030  817bcb31 00000000 817bc8b9 00000000
e1003040  817bc641 00000000 817bc3c9 00000000
e1003050  817bb021 00000000 817bbda9 00000000
e1003060  817bbb31 00000000 817bb8b9 00000000
e1003070  817bb641 00000000 817bb3c9 00000000
```

0xE1003000 이 _HANDLE_TABLE_ENTRY 구조체 배열이므로
위 그림에서 붉은 박스 형태로 읽어내면 된다. 쉽다. ^_^

_HANDLE_TABLE_ENTRY 구조체는 아래와 같다.

```
typedef struct _HANDLE_TABLE_ENTRY {
    union {
        PVOID Object;
        ULONG ObAttributes;
    };
    union {
        union {
```

```
            ACCESS_MASK GrantedAccess;
            struct {
                USHORT GrantedAccessIndex;
                USHORT CreatorBackTraceIndex;
            };
        };
        LONG NextFreeTableEntry;
    };
} HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENTRY;
```

복잡해 보이지만 (4 byte) Object | (4byte) GrantAccess 구조이다.
위에서 설명한 대로 Object 필드는 object 에 대한 포인터 이다.
이 Object 라는 것은 _OBJECT_HEADER.Body 필드이며 다시 말해서 Object 필드는 _OBJECT_HEADER.Body 를 가리키는 포인터이다.

0xE1003000 은 Object 필드가 0 이므로 건너뛰고,
0xE1003000 + sizeof(_HANDLE_TABLE_ENTRY) 필드를 보자

```
kd> dd 0xe1003000
e1003000  00000000 fffffffe 817bd9c9 00000000
```

```
kd> dt _HANDLE_TABLE_ENTRY e1003000+0x8
   +0x000 Object           : 0x817bd9c9
   +0x000 ObAttributes     : 0x817bd9c9
   +0x000 InfoTable        : 0x817bd9c9
   +0x000 Value            : 0x817bd9c9
   +0x004 GrantedAccess    : 0
   +0x004 GrantedAccessIndex : 0
   +0x006 CreatorBackTraceIndex : 0
   +0x004 NextFreeTableEntry : 0
```

XP 의 경우 Object 필드의 하위 1 비트가 Lock 플래그이다.
따라서 아래와 같은 코드를 만들어 낼 수 있을 것이다.

```
#define XP_TABLE_ENTRY_LOCK_BIT        0x1
PVOID pObj = (PVOID) ( (ULONG) Entry->Object & ~ XP_TABLE_ENTRY_LOCK_BIT );
```

따라서 Object 필드가 가리키는 Object 의 주소는 0x817bd9c9 & ~ 1 이므로
817BD9C8 가 된다.

_HANDLE_TABLE_ENTRY.Object 필드는 커널 오브젝트 포인터 이므로 이 필드가 가리키고 있는 것이 어떤 오브젝트인지를 판단하기 위해서는 _OBJECT_HEADER.Type 필드를 참조해봐야 할 것이다.

```
kd> dt _OBJECT_HEADER
   +0x000 PointerCount      : Int4B
   +0x004 HandleCount       : Int4B
   +0x004 NextToFree        : Ptr32 Void
   +0x008 Type              : Ptr32 _OBJECT_TYPE
   +0x00c NameInfoOffset    : UChar
   +0x00d HandleInfoOffset  : UChar
   +0x00e QuotaInfoOffset   : UChar
   +0x00f Flags             : UChar
   +0x010 ObjectCreateInfo  : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body              : _QUAD
```

Body 필드의 Offset 은 0x18 이므로

```
kd> dt _OBJECT_HEADER 0x817BD9C8-0x18
   +0x000 PointerCount      : 60
   +0x004 HandleCount       : 2
   +0x004 NextToFree        : 0x00000002
   +0x008 Type              : 0x817bd040
   +0x00c NameInfoOffset    : 0 ''
   +0x00d HandleInfoOffset  : 0 ''
   +0x00e QuotaInfoOffset   : 0 ''
   +0x00f Flags             : 0x22 '"'
   +0x010 ObjectCreateInfo  : 0x80562c80
   +0x010 QuotaBlockCharged : 0x80562c80
   +0x014 SecurityDescriptor : 0xe10016e4
   +0x018 Body              : _QUAD
```

```
kd> dt _OBJECT_TYPE 0x817bd040
   +0x000 Mutex             : _ERESOURCE
   +0x038 TypeList          : _LIST_ENTRY [ 0x817bd078 - 0x817bd078 ]
   +0x040 Name              : _UNICODE_STRING "Process"
   +0x048 DefaultObject     : (null)
   +0x04c Index             : 5
   +0x050 TotalNumberOfObjects : 0x14
   +0x054 TotalNumberOfHandles : 0x4c
   +0x058 HighWaterNumberOfObjects : 0x15
   +0x05c HighWaterNumberOfHandles : 0x50
   +0x060 TypeInfo          : _OBJECT_TYPE_INITIALIZER
   +0x0ac Key               : 0x636f7250
   +0x0b0 ObjectLocks       : [4] _ERESOURCE
```

결국 이 오브젝트는 Process 오브젝트 인 것을 확인할 수 있다.

그렇다면 _HANDLE_TABLE_ENTRY.Object 를 _EPROCESS 로 캐스팅 해보면 어떤 프로세스인지 확인할 수 있을 것이다.

```
kd> dt _EPROCESS 817BD9C8
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
---- 생략
   +0x170 Session          : (null)
   +0x174 ImageFileName    : [16]  "System"
```

만일 엔트리보다 더 많은 스레드나 프로세스가 있다면 어떻게 될까?

Windows internals 에서는 xp 나 2003 의 경우 mid-level table 이나 top level table 은 필요에 따라서 생성된다고 했으므로 아마 새로운 테이블을 만들면 PspCidTable 의 TableCode 값을 mid-level 또는 Top-level 테이블을 인덱싱 하도록 수정하지 않을까 싶다. (테스트 해보지 않아서 확신하지는 못하겠다.)

6. Conclusion

Windows 에서 사용하는 HANDLE 이라는 것은 결국 커널이 관리하고 있는 객체를 참조하기 위한 정보이며 윈도우는 커널 객체들을 효율적으로 관리하기 위해서 3 중 구조의 테이블을 만들어 사용하고 있다.
이 테이블의 개념은 2k 나 xp 가 동일하게 적용되나 실제 구현에 관한 내용은 많은 차이점을 보이고 있다.

또한 윈도우는 핸들과 관련된 특별한 심벌을 정의해서 사용하고 있으며 대표적인 예가 ObpKernelHandleTable 과 PspCidTable 이다.
ObpKernelHandleTable 같은 경우 system 프로세스가 사용하는 핸들들에 대한 테이블이며 PspCidTable 은 모든 프로세스와 스레드 객체에 대한 핸들 테이블이다.

PspCidTable 의 경우 프로세스/스레드에 대한 정보를 담고 있기 때문에 이를 잘 사용하면 루트킷 탐지 혹은 루트킷 제작에 많은 도움이 될 수 있을 것이다.