

XP/S2K3/Vista/Win7 bugs help malware to survive

CodeGate 2009

발표 : Kris Kaspersky

번역, 재작성 : window31 (<http://www.window31.com>)

이 내용은 Kris Kaspersky 가 CodeGate 2009 에서 "XP/S2K3/Vista/Win7 bugs help malware to survive" 라는 제목으로 발표한 내용 PPT자료를 한글로 각색한 글입니다. 공개된 자료는 PPT 파일이 전부였기 때문에 좀더 원활한 설명을 위해 원본 그대로의 번역 보다는 필자의 기타 부연설명이 많이 들어갔습니다. 추가적으로 풀 소스 코드도 함께 첨부되어 있습니다. 소스코드의 내용은 발표 때는 공개되었지만 배포자료에 함께 제공되지 않은 틀을 PPT 자료에 근거하여 코딩해 본 것들입니다. 이번 CodeGate 에서 Kris 의 통역자가 중간에 통역을 포기하고 그만두었기 때문에 논란이 있었고(Kris 의 러시아어 섞인 영어발음과 지나치게 빠른 스피킹에 통역하기가 어려웠을 것이므로 통역자를 완전히 탓하기도 그렇습니다) 그 탓에 발표 내용을 완전히 흡수하지 못한 분들이 아쉬움을 표명하고 있는데, 좋은 발표가 그런 식으로 묻히는 것에 대한 아쉬움에, 이와 같은 문서를 작성하게 되었습니다.

주제 1 : 잘못된 스레드 시작 번지

주제 2 : 엔트리 포인트가 없는 EXE 파일

프롤로그

OS 를 사용하다 보면 버그가 나오고 핫픽스가 나오고 하는 상황이 반복된다. 그런 과정에서 악성코드는 신규 기법을 발견하게 되고 그것은 Undocumented 형태로 새로운 공격 구현에 이용된다. 이런 식으로 악성코드 개발자는 Undocumented 형태의 기술도 자유로이 이용할 수 있지만 안티바이러스를 개발하는 입장에서는 악성코드가 새로운 기술을 사용했다 하더라도, 그것이 Undocumented 라면 함부로 그 기술을 적용할 수가 없다. 왜냐하면 모든 OS와 플랫폼을 고려하여 개발해야 하므로 악성코드 차단과 동시에 안정성이라는 과제를 항상 가지고 가야 하기 때문이다. 따라서 아무렇게나 마구잡이로

돌아만 가면 괜찮은 악성코드를 만드는 입장보다는 훨씬 불리한 상황에서 있다. 그래서 Kris 는 악성 코드가 새로운 기능을 만들 때 그 원리가 Undocumented feature 라도 고민하지 않고 마음껏 만들 수 있지만, 보안 개발자는 그렇게 하지 못해서 너무 불리한 싸움이라는 점을 강조했다. 또, 그런 Undocumented 로 개발된 악성 코드는, Reverse Engineering 하는 입장에서도 처음 보는 기술이라 악성코드를 분석하기도 더욱 힘들다는 얘기도 추가적으로 덧붙이고 있다.

주제 1 : 잘못된 스레드 시작 번지

CreateRemoteThread()를 이용하여 원격에서 만들어진 스레드는 보통, DLL 을 가지고 있다. 하지만 DLL 주입 없이 타 프로세스에 코드만 풀어놓고 CreateRemoteThread() 로 스레드를 돌린다면, DLL 리스트에는 해당 DLL 이 나오지 않기 때문에, 백신이나 프로세스 익스플로러 같은 뷰어 툴에서 확인할 수가 없다. 그리고 그 같은 미 감지 상태에서 스레드는 계속 돌아가고 있으며, 공격자는 원하는 행위를 지속할 수 있다 (Kris Kaspersky 는 이런 행위를 ring3 rootkit 이라고도 표현했다). 이런 스레드는 어떻게 감지해야 하는 것일까?

스레드의 메모리 Type 검사

CreateRemoteThread() 를 이용하면 타 프로세스에 메모리 공간을 할당하는 작업이 이루어진다. 그리고 VirtualAllocEx() 을 이용하기 때문에 그 영역은 HEAP 공간이 된다. 따라서 이 영역을 VirtualQuery()를 이용하여 메모리 Type 을 살펴보면, MEM_PRIVATE 라는 값을 가지게 된다. 반면에 정상적인 DLL 이나 EXE 내부의 코드에 의해 생성된 스레드는 HEAP 이 아닌 Code Section 에 포함되어 있다. 그러므로 정상인 경우는 MEM_PRIVATE 가 아닌 MEM_IMAGE Type 을 가지게 된다. 따라서 먼저 스레드의 시작 번지를 리스팅한 후, 그 번지의 메모리 속성을 살펴보면 MEM_PRIVATE 가 나올 경우, 그것은 CreateRemoteThread() 를 이용한 ring3 rootkit 이라고 판단할 수 있다. 따라서 우리는 스레드의 시작 번지를 얻어온 후 그 Type 을 검사한다면, 공격자의 원격 스레드인지 아닌지를 판별할 수 있다.

Address	Type	Size	Pages	Protection	Path
01B71000	Free	61440			
01B80000	Image	49152	7	ERWC	D:\code\launching\TestGameHS\HSD11\
01B80000	Image	4096		-R--	
01B81000	Image	20480		E---	
01B86000	Image	4096		-R--	
01B87000	Image	4096		-RW-	
01B88000	Image	4096		-RWC	
01B89000	Image	8192		-RW-	
01B8B000	Image	4096		-R--	
01B8C000	Free	16384			
01B90000	Private	65536	2	-RW-	
01B90000	Private	16384		-RW-	
01B94000	Reserve	49152		-RW-	
01B98000	Free	131072			

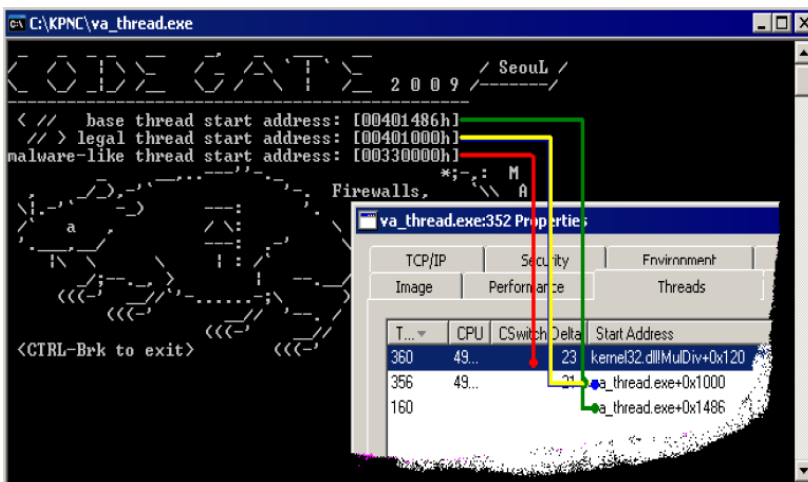
<화면 1> MEM_IMAGE 영역. Image 라고 표시되어 있다. 정상적인 DLL 영역이다.

Virtual Memory Map (PID=4592 "EnumTh.exe")						
Refresh! Expand Regions! Copy to Clipboard!						
003E0000	Private	4096	1	ERW-		
003E0000	Private	4096		ERW-		
003B1000	Free	323584				
00400000	Image	49152	6	ERWC	D:_code\launching\Kaspersky\Launching\	
00400000	Image	4096		-R--		
00401000	Image	20480		ER--		
00406000	Image	4096		-R--		
00407000	Image	4096		-RW-		

<화면 2> Private 영역. 이곳은 리모트 스레드의 영역이다. 실행 속성인 E 속성을 가진 것도 보인다.

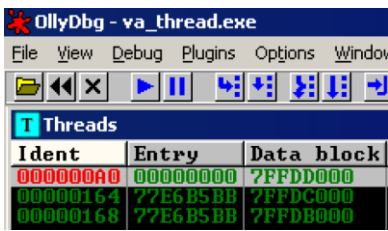
Procexp, OllyDBG 의 버그

Procexp 에는 스레드 리스트를 보여주는 기능을 가지고 있다. 하지만 그 기능에는 버그가 존재한다. 정상적인 스레드는 시작 번지를 제대로 표기해 주지만, 이와 같이 리모트로 생성된 스레드는 엉뚱한 값으로 표기해 준다. <화면 3> 을 보면 그 내용이 나와 있다. 2개의 스레드를 가진 프로세스에 1개의 리모트 스레드를 추가로 생성한 모습이다. Malware like thread 라고 표기된 스레드는 리모트로 생성된 스레드이며 실제 위치는 0x330000 번지에서 돌아가고 있지만 Procexp 는 이 번지를 Kernel32.dll!MulDiv+0x120 (0x7943B700) 라고 가리키고 있다. 잘못된 리스팅이다.



<화면 3> Procexp의 버그 (page 9, Kris's PPT)

OllyDBG 의 경우도 마찬가지다. OllyDBG도 스레드 리스트를 보여주는 기능을 보유하고 있으나 역시 엉뚱한 번지를 리스팅해 준다. OllyDBG 의 경우는 하나도 맞는게 없다. Procexp 의 경우보다 훨씬 더 심하다.



<화면 4> OllyDBG 의 Threads 윈도우 (page 11, Kris's PPT)

시작 번지를 찾는 세미 휴리스틱 알고리즘

그렇다면 이와 같은 리모트 스레드의 경우는 어떤 식으로 시작 번지를 찾을 수 있을까? 아직 정식 문서화된 방법으로는 이런 스레드의 시작 번지는 찾을 수가 없다. 따라서 스택을 다루는 약간의 테크닉이 가미된 방법을 동원해야 된다. 물론 이 방법은 Undocumented 한 내용이다.

모든 스레드의 시작 번지는 스택의 특성상 ESP의 특정 위치에 항상 존재하게 된다. 그 위치는 스택의 바닥에서부터 세번째 DWORD 값이거나 두번째 DWORD 값이다. <화면 5>를 보자. 방금 얘기한 내용대로, 스레드의 시작 주소는 밑에서 세번째 위치나 두번째 위치에 항상 자리잡고 있다. (OllyDBG 로 Attach 한 후 스택 윈도우 맨 밑에까지 가 보면 항상 스레드의 엔트리가 기록되어 있다는 것을 쉽게 확인할 수 있다)



<화면 5> 스레드의 시작 번지가 담긴 곳

따라서 이런 알고리즘으로 구현이 가능하다.

- 1) 만일 스택의 바닥에서부터 세번째 값이 0이 아니라면 그곳이 스레드의 시작 번지이다.
- 2) 만일 스택의 바닥에서부터 세번째 값이 0이라면 두번째 값이 스레드의 시작 번지이다.

```
th_addr = ((buf[GET_FZ -3]) ? buf[GET_FZ -3] : buf[GET_FZ -2]);
printf("start address : %08Xh\n",th_addr?th_addr:0xDEADBEEF);
```

<코드 1> 간단 알고리즘 page 14, Kris's PPT

그럼 우리에게 필요한 작업은 알고리즘은 다음과 같다.

- 1) 스레드 리스트를 구하여, 스레드 체크를 원하는 프로세스로 접근
- 2) 해당 프로세스의 esp 를 얻어오는 것
- 3) 스택의 두번째 또는 세번째 값을 이용하여 스레드의 시작 번지 획득
- 4) 얻어온 스레드의 시작 번지의 메모리 Type 을 검사하여 MEM_PRIVATE 인지 검사

먼저 스레드 리스트는 toolhelp32 라이브러리로 쉽게 구할 수 있다. Thread32First/Thread32Next API 를 이용한다.

```
DWORD dwCurPid = GetCurrentProcessId();
THREADENTRY32 pth;
HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, dwCurPid);
pth.dwSize = sizeof(THREADENTRY32);

Thread32First(hSnapshot, &pth);

while(Thread32Next(hSnapshot, &pth))
{
    if (pth.th32OwnerProcessID == dwCurPid)
    {
        PrintThreadInfo(pth.th32ThreadID, pth.th32OwnerProcessID);
        printf("pth.tpBasePri: %X\n", pth.tpBasePri);
    }
}
CloseHandle(hSnapshot);
```

<코드 2> 스레드 리스트 출력

그리고 PrintThreadInfo() 안에서 각 스레드에 대한 정보를 획득한다. 우리가 필요한 정보는 esp 의 위치와 내용이다. 그것은 GetThreadContext()로 얻을 수 있다. 그 API 에는 스레드 핸들이 인자로 필요한데 이것은 OpenThread() 로 얻을 수 있다. OpenThread() 는 플랫폼 SDK 의 버전에 따라 링크되어 있

지 않을 수도 있다 (VC 6++ 기준). 따라서 GetProcAddress() 로 함수 포인터를 얻는 작업이 필요하다.

```
-----  
typedef HANDLE ( __stdcall *OPENTHREAD)(  
    DWORD dwDesiredAccess, // access right  
    BOOL bInheritHandle,   // handle inheritance option  
    DWORD dwThreadId       // thread identifier  
);  
OPENTHREAD fnOpenThread = NULL;  
fnOpenThread = (OPENTHREAD)GetProcAddress(GetModuleHandle("kernel32.dll"),  
    "OpenThread");  
-----
```

<코드 3> OpenThread

이제 스레드의 핸들을 얻고, esp 와 해당 컨텍스트의 버퍼를 따오는 작업을 진행하자. CONTEXT 구조체를 선언한 뒤, esp 는 VirtualQueryEx() 로 스택의 위치를 파악하고, 할당된 스택 영역만큼 읽어오자. 그리고 스택의 바닥에서 두번째 또는 세번째 값을 가져오면 그것이 스레드의 시작 번지이므로, 그 값의 메모리 Type 을 검사한다는 것이 요점이다. 그 내용이 <코드 4>에 기록되어 있다. 궁극적인 목적은 스레드의 시작 번지가 MEM_IMAGE 인지 MEM_PRIVATE 인지 체크하기 위해서다. MEM_PRIVATE 는 다시 얘기하지만 DLL 없이 리모트로 주입한 스레드이며 악성코드로 간주할 수 있다.

```
-----  
#define GET_FZ 4  
hThread = fnOpenThread(THREAD_GET_CONTEXT, 0, dwThreadId);  
GetThreadContext(hThread, &context);  
hProcess = OpenProcess(PROCESS_VM_READ|PROCESS_QUERY_INFORMATION, 0, dwOwnerPid);  
  
if (hProcess)  
{  
    VirtualQueryEx(hProcess, (void*)context.Esp, &mbi, sizeof(mbi));  
  
    DWORD stack = (DWORD) mbi.BaseAddress + mbi.RegionSize;  
  
    DWORD read = 0;  
    BOOL bRead = ReadProcessMemory(hProcess,  
        (char*)stack - GET_FZ*sizeof(DWORD), buf, GET_FZ*sizeof(DWORD), &read);  
}
```

```

if (bRead)
{
    DWORD st_adr = 0;
    st_adr = ((buf[GET_FZ-3])?buf[GET_FZ-3]:buf[GET_FZ-2]);

    printf("start address      : %08Xh\n",st_adr?st_adr:0xDEADBEEF);
    printf("point to args       : %08Xh\n",
           ((buf[GET_FZ-3])?buf[GET_FZ-2]:buf[GET_FZ-1]));

    VirtualQueryEx(hProcess, (void*)st_adr, &mbi, sizeof(mbi));

    printf("type                : %s\n",
           (mbi.Type==MEM_IMAGE)?"MEM_IMAGE":
           (mbi.Type==MEM_MAPPED)?"MEM_MAPPED":
           (mbi.Type==MEM_PRIVATE)?"MEM_PRIVATE":"UNKNOWN");

    printf("%08Xh %08Xh %08Xh %08Xh\n",
           buf[GET_FZ-3], buf[GET_FZ-2], buf[GET_FZ-1], buf[GET_FZ-0]);
}
}
}

```

<코드 4> core routine.

이제 휴리스틱 알고리즘이 완성되었다. 결과를 보자.

```

ex "D:\_code\launching\Kaspersky\Launching
press any key after you excute MRT.exe...
start address      : 00401345h
point to args      : 00000000h
type               : MEM_IMAGE
00000000h 00401345h 00000000h 00000020h
pth.tpBasePri: 8
start address      : 00401000h
point to args      : 00000000h
type               : MEM_IMAGE
00401000h 00000000h 00000000h 00000020h
pth.tpBasePri: 8
enjoy debugging !

```

<화면 6> 정상 스레드 리스트

```

C:\WINDOWS\system32\cmd.exe - EnumTh.exe

D:\_code\launching\Kaspersky\Launching>EnumT
press any key after you excute MRTI.exe....i
start address      : 00401345h
point to args      : 00000000h
type               : MEM_IMAGE
00000000h 00401345h 00000000h 00000020h
pth.tpBasePri: 8
start address      : 003A0000h
point to args      : 003B0000h
type               : MEM_PRIVATE
003A0000h 003B0000h 00000000h 00000020h
pth.tpBasePri: 8
start address      : 00401000h
point to args      : 00000000h
type               : MEM_IMAGE
00401000h 00000000h 00000000h 00000020h
pth.tpBasePri: 8
enjoy debugging ?

```

<화면 7> 리모트 스레드 탐지된 모습

<화면 6>의 경우는 두개의 스레드가 정상적으로 가동되고 있는 모습이며 (0x401345 은 main thread , 0x401000 은 CreateThread 로 생성한 일반적인 스레드이다), <화면 7>은 그 두개 외에 1개의 리모트 스레드가 추가로 탐지된 모습이다. Type 이 MEM_PRIVATE 인 것으로 보아 리모트 스레드가 확실하며 그 코드는 0x3A0000 번지에 숨어 있다. 메인 프로세스는 0x401000 번지부터 시작하는 것으로 보아 그 번지와 전혀 연관이 없는 0x3A0000의 스레드는 누군가에 의해 생성된 스레드이다. 이런식으로 디텍션할 수 있다.

```

Lfp99 - EnumTh.exe - [CPU - thread 0000F24]
File View Debug Plugins Options Window Help
Paused
003A0000 56      push esi
003A0001 8B7424 08    mov esi, dword ptr ss:[esp+8]
003A0005 56      push esi
003A0006 FF96 84000000 call near dword ptr ds:[esi+84]
003A000C 68 E8030000 push 3E8
003A0011 FF96 88000000 call near dword ptr ds:[esi+88]
003A0017 ^ EB EC    jmp short 003A0005
003A0019 90      nop
003A001A 90      nop
003A001B 90      nop

```

<화면 8> 0x3A0000 번지에 숨어 있는 스레드.

우회방법

이 같은 세미 휴리스틱 알고리즘에 탐지되지 않고 리모트 스레드를 생성할 수 있는 방법이 없을까? 간단하다. ESP의 두번째 또는 세번째에 위치해 있는 스레드 시작 주소를, 스레드 실행과 동시에 제거 해버리는 기능을 만들면 된다. 스레드가 시작되면, 이제 그 주소는 더 이상 필요가 없기 때문에 이렇게 제거해 버려도 프로그램에는 아무 영향을 끼치지 않는다. 따라서 그 번지를 0 으로 만들거나 혹은

영동한 쓰레기 번지를 넣는 형태로 페이킹, 어뷰징할 수도 있다. 따라서 다음 세대의 malware 는 아마도 이런 트릭을 사용할 것으로도 예상된다.

0060FFD0	00000000		0012FFD0	7FFDE000	
0060FFD4	0060FFC4		0012FFD4	0000000F	
0060FFD8	00000000		0012FFD8	0012FFC8	
0060FFDC	FFFFFFFF	End of SEH chain	0012FFDC	F8A77CF8	
0060FFE0	77E6B798	SE handler	0012FFE0	FFFFFFFF	End of SEH chain
0060FFE4	77E66070	kernel32.77E66070	0012FFE4	77E6B798	SE handler
0060FFE8	00000000		0012FFE8	77E523D8	kernel32.77E523D8
0060FFEC	00000000		0012FFEC	00000000	
0060FFF0	00000000		0012FFF0	00000000	
0060FFF4	00401000	va_threa.00401000	0012FFF4	00000000	
0060FFF8	0000C001		0012FFF8	00401486	va_threa.<ModuleEntry
0060FFFC	00000000		0012FFFC	00000000	

<화면 9> 선택한 두 부분을 스택에서 지워 버린다.

주제 2 : 엔트리 포인트가 없는 EXE 파일

EXE 파일은 대부분 엔트리 포인트를 가지고 있다. 이 엔트리 포인트는 프로그램이 시작될 때 가장 처음 호출되는 번지이지만, 사실은 엔트리 포인트보다도 먼저 호출되는 코드가 있다. 그것은 바로 Static 으로 빌드된 DllMain() 함수와 TLS CallBack 이다. 이 영역은 EXE 의 EP가 호출되기 전에 먼저 초기화 되는 부분이라 이 부분에 페이킹 코드를 넣고, 우리가 원하는 대로 엔트리 포인트 를 바꿀 수 있다. 이런 기술에 대해 TLS CallBack 의 경우는 흔하게 사용되지만, DllMain()의 경우는 거의 알려져 있지 않다. 따라서 DllMain()을 이용한 "외부 TLS CallBack" 을 소개해 보도록 하겠다.

DLL 디자인

이해를 쉽게 도모하기 위하여 간단한 크랙미를 목적으로 예제를 만들어 보자. 프로그램의 목적은 디버깅을 방지하기 위한 것이다. 따라서 디버거 없이 그냥 실행하면 정상적으로 실행되지만, 디버거로 불러서 실행하면 디버깅 감지 메시지를 출력하는 프로그램을 제작할 것이다. 그래서 우리에게, 디버거로 실행하면 엔트리 포인트를 디버깅 감지 메시지를 출력하는 코드로 변경시켜버리는 작업이 필요하다. DllMain() 이 그 역할을 한다. DllMain()의 DLL_PROCESS_ATTACH: 에다가 <코드 5>와 같은 코드를 작성하자.

```
-----
#define PE_off  0x3C
#define EP_off  0x28
```

```

bool DllAttach(HANDLE hModule)
{
    LPBYTE base_x = (LPBYTE)GetModuleHandle(0);
    DWORD pe_off = *((DWORD*)(base_x + PE_off));
    DWORD ep_off = *((DWORD*)(base_x + pe_off + EP_off));
    LPBYTE ep_adr = base_x + ep_off;

    // check int3
    if (*ep_adr == 0xCC)
        return false;

    // ....

    return true;
}

```

<코드 5> DllAttach() 의 내용(상)

GetModuleHandle()로 EXE의 ImageBase 를 얻어온 후, PE 로 접근하여 엔트리 포인트를 얻는다. ep_adr 에는 오리지널 엔트리 포인트가 담긴다. 디버거로 프로그램을 열었을 때, 맨 처음 화면에 걸리며 멈추는 부분이 바로 이 번지 영역이다. 디버거는 여기서 int3 로 프로그램을 멈추어 놓으므로, 우리는 if (*ep_adr == 0xCC) 라는 코드로 바이너리에 0xCC 가 박혀있는지 검사한다. 그리고 int3 상태라면 디버거가 물고 있다고 간주하며 그상태에서 리턴시켜버린다.

스택에서 엔트리 찾기

그리고 만약 디버거로 오픈 상태가 아니라면 계속 다음 루틴으로 이어와 그때부터 엔트리 포인트를 바꾸는 작업을 수행한다. <코드 6>을 보자. DllMain()이 수행된 뒤, EXE 의 실제 엔트리 포인트로 이동하게 되는데, 그 엔트리 포인트는 스택의 어딘가에 담겨 있다. 따라서 우리는 스택의 맨 위에서부터 끝까지 검색하며 정상 엔트리 포인트를 우리가 만든 또다른 엔트리 포인트로 변경해야 한다. VirtualQuery() 로 MEMORY_BASIC_INFORMATION 구조체의 BaseAddress 멤버 변수를 얻으면 그 영역이 현재 스레드에서 사용중인 스택의 꼭대기 영역이다. 여기서부터 RegionSize 만큼 탐색을 시작한다. 4 byte 씩 줄여가며 스택을 뒤진다. 그리고 오리지널 엔트리 포인트(ep_adr)와 같은 값이 걸릴 경우, 복귀할 엔트리 포인트의 주소로 간주하고 우리가 미리 알고 있는 NEW_EP 로 변경시켜버린다. 그러면 디버거로 열지 않았을 때는 NEW_EP 로 엔트리 포인트가 변경되어 실행될 것이다.

```

-----
#define NEW_EP 0x40102C
#define EP_KEY 0xA181818A // fake for IDA Pro
DWORD ep_key = EP_KEY;

// Modify Entry point
MEMORY_BASIC_INFORMATION mbi;
VirtualQuery((LPCVOID)&hModule, &mbi, sizeof(mbi));
LPBYTE lpBaseAddress = ((LPBYTE)mbi.BaseAddress);
DWORD dwRegionSize = mbi.RegionSize - sizeof(DWORD);

for(dwRegionSize; dwRegionSize > 0; dwRegionSize-=sizeof(DWORD))
{
    // 스택에서 ep가 있으면 그것을 new EP 로 바꾼다.
    if (((DWORD)ep_adr)!=( *(DWORD*)(lpBaseAddress+dwRegionSize)))
    {
        (*(DWORD*)(lpBaseAddress+dwRegionSize)) = NEW_EP ^ EP_KEY;
        (*(DWORD*)(lpBaseAddress+dwRegionSize)) ^= ep_key;
    }
}
-----

```

<코드 6>DllAttach()의 내용(하)

IDA 속이기

PPT 자료의 코드에 보면, NEW_EP 를 스택에 바로 대입하지 않고 EP_KEY 를 이용해 한번 XOR 하는 작업을 진행한다. Kris 의 자료에 구체적으로는 언급되어 있지 않지만, 이런 작업을 하는 이유에 대해 알아보자. 만일 여기서 XOR 을 수행하지 않고 바로 NEW_EP 로 번지를 바꾸어 버린다면, 디스어셈블링 할 때 <화면 10> 처럼 이 번지가 확연히 보이게 된다. [eax+ecx] 가 스택에 담겨있는 엔트리 포인터이며 이 번지를 NEW_EP 인 0x40102C 번지로 바꾸는 모습이 너무도 쉽게 노출된다.

```

.text:10001031 loc_10001031:                                ; CODE XREF: sub_10001010+18↑j
.text:10001031      lea     edx, [esp+20h+Buffer]
.text:10001035      push   1Ch                ; dwLength
.text:10001037      lea     eax, [esp+24h+Address]
.text:10001038      push   edx                ; lpBuffer
.text:1000103C      push   eax                ; lpAddress
.text:1000103D      call   ds:VirtualQuery
.text:10001043      mov     edx, [esp+20h+Buffer.RegionSize]
.text:10001047      mov     ecx, [esp+20h+Buffer.BaseAddress]
.text:10001048      lea     eax, [edx-4]
.text:1000104E      test   eax, eax
.text:10001050      jbe    short loc_10001063
.text:10001052
.text:10001052 loc_10001052:                                ; CODE XREF: sub_10001010+51↓j
.text:10001052      cmp     esi, [eax+ecx]
.text:10001055      jnz    short loc_1000105E
.text:10001057      mov     dword ptr [eax+ecx], 40102Ch
.text:1000105E
.text:1000105E loc_1000105E:                                ; CODE XREF: sub_10001010+45↑j
.text:1000105E      sub     eax, 4
.text:10001061      jnz    short loc_10001052
.text:10001063
.text:10001063 loc_10001063:                                ; CODE XREF: sub_10001010+40↑j

```

<화면 10> 쉽게 노출되는 NEW_EP

따라서 NEW_EP ^ EP_KEY; 처럼 XOR 로 한번 암호화 해 주고 다시 바로 복호화 해 주면, 번지가 노출되지 않고 <화면 11>과 같이 이상한 값으로 나오게 된다.

```

.text:10001031 loc_10001031:                                ; CODE XREF: sub_10001010+18↑j
.text:10001031      lea     edx, [esp+20h+Buffer]
.text:10001035      push   1Ch                ; dwLength
.text:10001037      lea     eax, [esp+24h+Address]
.text:10001038      push   edx                ; lpBuffer
.text:1000103C      push   eax                ; lpAddress
.text:1000103D      call   ds:VirtualQuery
.text:10001043      mov     edx, [esp+20h+Buffer.RegionSize]
.text:10001047      mov     ecx, [esp+20h+Buffer.BaseAddress]
.text:10001048      lea     eax, [edx-4]
.text:1000104E      test   eax, eax
.text:10001050      jbe    short loc_10001072
.text:10001052
.text:10001052 loc_10001052:                                ; CODE XREF: sub_10001010+60↓j
.text:10001052      cmp     esi, [eax+ecx]
.text:10001055      jnz    short loc_1000106D
.text:10001057      mov     dword ptr [eax+ecx], 0A1C191A6h
.text:1000105E      mov     edx, dword_10007030
.text:10001064      xor     edx, 0A1C191A6h
.text:1000106A      mov     [eax+ecx], edx
.text:1000106D
.text:1000106D loc_1000106D:                                ; CODE XREF: sub_10001010+45↑j
.text:1000106D      sub     eax, 4

```

<화면 11> XOR 후

그리고 이런 상황에서 XOR 을 이용할 때의 주의점이 있다. <코드 7>처럼 EP_KEY 를 두번 사용하면 컴파일러가 빌드를 할 때, 이것은 무의미한 코드로 간주하여 빌드 단에서 그 코드를 제외시켜버리므로, 컴파일 후에는 위의 <화면 10>과 똑같은 코드가 되고 만다. 따라서 <코드 8>처럼 DWORD ep_key

라는 별도의 변수를 두어 EP_KEY 를 대입시킨후 그것으로 두번째 XOR 을 수행해야 한다.

```
-----  
#define NEW_EP 0x40102C  
#define EP_KEY 0xA181818A  
  
(* (DWORD*)(lpBaseAddress+dwRegionSize)) = NEW_EP ^ EP_KEY;  
(* (DWORD*)(lpBaseAddress+dwRegionSize)) ^= EP_KEY;
```

<코드 7> 빌드 단에서 제외되는 코드, 이런식으로 처리하면 안된다.

```
-----  
#define NEW_EP 0x40102C  
#define EP_KEY 0xA181818A  
DWORD ep_key = EP_KEY; // fake for IDA Pro  
  
(* (DWORD*)(lpBaseAddress+dwRegionSize)) = NEW_EP ^ EP_KEY;  
(* (DWORD*)(lpBaseAddress+dwRegionSize)) ^= ep_key;
```

<코드 8> DWORD ep_key 추가하여 두번째 XOR 을 해야 한다.

익스포트 함수 추가

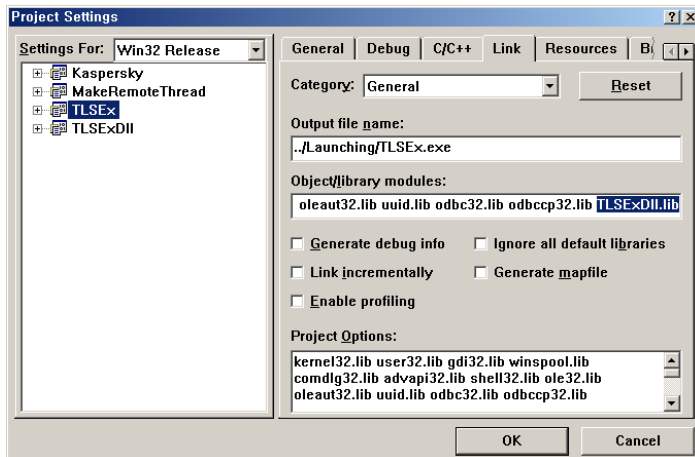
그리고 익스포트 함수의 추가가 필요하다. 이 함수는 EXE 에서 호출할 목적이다. 이 함수가 디버거가 걸리지 않았을 때의 엔트리 포인트가 될 것이다.

```
-----  
#define TLSEXDLL_API __declspec(dllexport)  
TLSEXDLL_API int fnTLSExDll(void)  
{  
    MessageBox(0, "debugger not found.", ":", MB_OK);  
    return 42;  
}
```

<코드 9> 익스포트 함수 추가

EXE 디자인

이제 EXE 를 제작해 보자. DLL 을 Import 에 추가시켜야 하므로 앞서 디자인한 DLL 의 Release 폴더에 있는 lib 파일을 가져와서 링크해야 한다.



<화면 12> 링크

그리고 DLL 의 익스포트 함수의 원형을 선언하고 메인 함수를 <코드 10>과 같이 구현한다.

```
_declspec(dllexport) int fnTLSExDll(void);
```

```
void Init()
```

```
{  
    MessageBox(0, "Debugger Detect !", "Hacker :p", MB_OK);  
}
```

```
int main(int argc, char* argv[])
```

```
{  
    __asm  
    {  
        nop  
        nop  
        nop  
        nop  
        nop  
    }
```

```

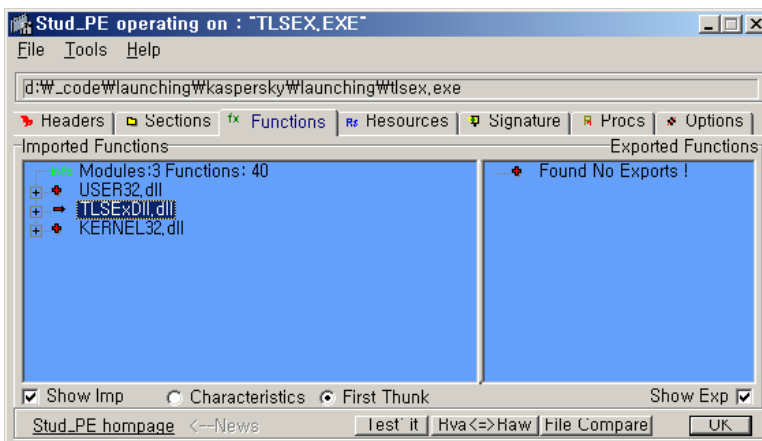
        call Init
        retn
        nop
        call ds:fnTLSExDll
        retn
    }

return 0;
}

```

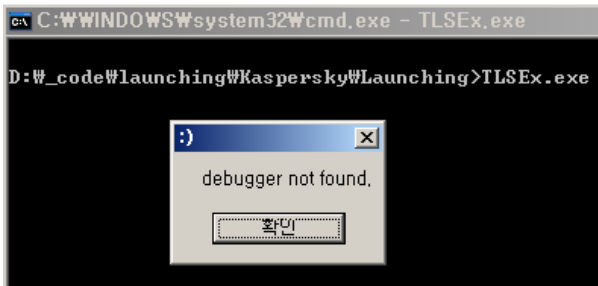
<코드 10> exe 의 코드

DLL 의 #define NEW_EP 0x40102C 에 해당하는 0x40102C 값은 call ds:fnTLSExDll 의 위치가 된다. 우리의 목적은 디버거로 실행되었을 때 call Init 이 실행되게 하고, 정상적으로 실행되었을 때 call ds:fnTLSExDll 가 실행되게 하는 것이다.

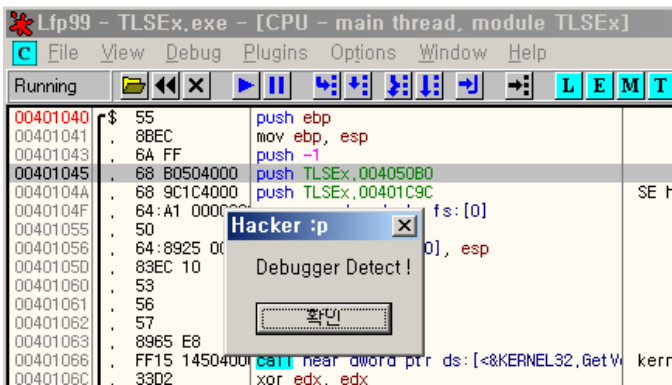


<화면 13> TLSEX.exe 의 Import 정보

이제 빌드하게 되면, DLL 을 Static 링크 하였으므로 EXE 에는 TLSEX.dll 이 Import 테이블로 붙게 된다. 그리고 TLSEX.exe 를 실행하면 EXE 파일의 엔트리 포인트가 실행되기 전에 시스템은 내부적으로 TLSEX.dll 을 LoadLibrary() 하게 되고, 그 시점에 TLSEX.dll 의 DllMain()이 호출된다. 그리고 그 안에서 엔트리를 변경하거나 디버깅 감지 시도를 할 수 있는 등 TLS CallBack 의 역할을 할 수 있게 된다 !



<화면 14> 정상 실행



<화면 15> 디버거로 실행

안티바이러스 개발자들의 악몽

이 방법은 TLS Callback 을 시뮬레이트 하는 방법이라 볼 수 있다. 구현 방법은 그다지 어렵지 않다(뭐 그렇다고 아주 쉬운 것도 아니다). 어쨌든 이 방법을 이용하여 안티바이러스의 디텍션을 피하는 악성 코드의 제작도 충분히 가능하다. 하지만 안티바이러스를 개발하는 입장에서는 이와 같은 PE Loader 형식에 대해 모두 시뮬레이트 하여 스캔할 수는 없다는 점이 문제점이다. 또한 이런 방법들은 대부분 Undocumented 한 것이라 구현하기에 더욱 골치 아프다

프로그램이 어디에선가 어떤 값들이 변경되고 그것이 OS 의 컨트롤을 벗어나는 일이 발생한다면, 그것은 정말 말도 안되는 것이다. 이런 식으로 바꾸어댄다면 스택의 어디에 어느 값이 EP가 될지 누가 알겠는가. EP 의 정확한 위치는 시스템에 따라 달라질 수밖에 없다. 따라서 그것을 빠르게 찾아낼 수 있는 정확한 방법은 없다. 이것은 안티바이러스 개발자에게 정말 큰 위협이다. 안티바이러스 개발자들은 헤아릴 수 없이 많은 양의 코드를 다시 작성해야만 할 것이다.

이 자료에 사용된 풀 소스 코드 위치 : <http://window31.com/entry/kasperskyCodegate2009>