윈도우 프로그래머를 위한 PE 포맷 가이드
실행 파일 프로텍터

목차

저작권

소개

요즘 출시되는 많은 프로그램들은 해커의 공격에 대비해서 상용 프로텍터로 실행 파일을 보호한다. 이를 통해 해커의 공격을 100% 막을 순 없지만 어느 정도 저지할 순 있기 때문이다. 지난 시간에 살펴 보았던 원리를 기초로 간단한 실행 파일 프로텍터를 제작해 보도록 한다.

연재 가이드

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

연재 순서

2007. 08. 실행 파일 속으로

2007. 09. DLL 로딩하기
2007. 10. 실행 파일 생성기의 원리
2007. 11 코드 패칭
2007. 12 바이러스
2008. 01 진화하는 코드
2008. 02 실행 압축의 원리
2008. 03 실행 파일 보안의 원리
2008. 04 실행 파일 프로텍터

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

시시각각 조여오는 프로젝트 데드라인, 산더미처럼 쌓인 버그 리포트, 반드시 해결해야만 하는, 하지만 해결 방법은 오리 무중인 문제들에 둘러싸여 하루를 보내다 보면 사는 건지 전쟁을 하는 건지 의문이 드는 순간이 한 두 번이 아니다. 입가에 머금은 녹차 향을 즐길 여유도 없이 삼켜야만 하는 현실이 안타깝다. 물론 마음 한 켠엔 능력이 부족하니 손발이 고생이란 생각이 들기도 한다. 이러한 필자의 마음도 모른 채 시간은 늘 째깍째깍 자기 갈 길을 재촉해서 떠나버린다.

Introduction

이번 시간에는 그 동안 배웠던 내용을 바탕으로 간단한 실행 파일 프로텍터를 제작해 본다. 일단 기반 구조가 만들어지면 그곳에 기능을 추가하는 것은 비교적 자유롭기 때문에 전체적으로 기반 구조를 만드는 것에 초점을 맞추어서 살펴보도록 하자.

프로텍터 제작에 앞서 가장 먼저 생각해야 할 부분은 메모리의 구조다. <그림 1>에 일반적인 실행 파일 프로텍터의 메모리 구조가 나와 있다. 물론 대부분의 상용 프로텍터는 원본 IAT와 리소를 보호하는 기능까지 가지고 있는 경우가 대부분이지만 여기서는 문제를 단순화 시키기 위해서 코드만을 보안한다는 가정을 가지고 그려본 것이다.

<그림 1>의 프로텍터의 가장 큰 특징은 스텁 코드가 2단계로 구성된다는 것이다. 1차 스텁 코드의 역할은 2차 스텁 코드를 메모리 상에 올리는 일을 한다. 보통 2차 스텁 코드는 VirtualAlloc등을 통해 할당된 별도의 메모리에서 수행된다. 2차 스텁 코드에 대한 메모리 할당과 재배치, API 주소 연결 작업이 끝나면 1차 스텁 코드는 2차 스텁 코드로 점프한다. 2차 스텁 코드는 기본적으로 파일이 실행되기에 안전한 환경인지를 점검하고 암호화된 코드를 풀어서 복원한 다음 원본 코드를 실행시킨다.

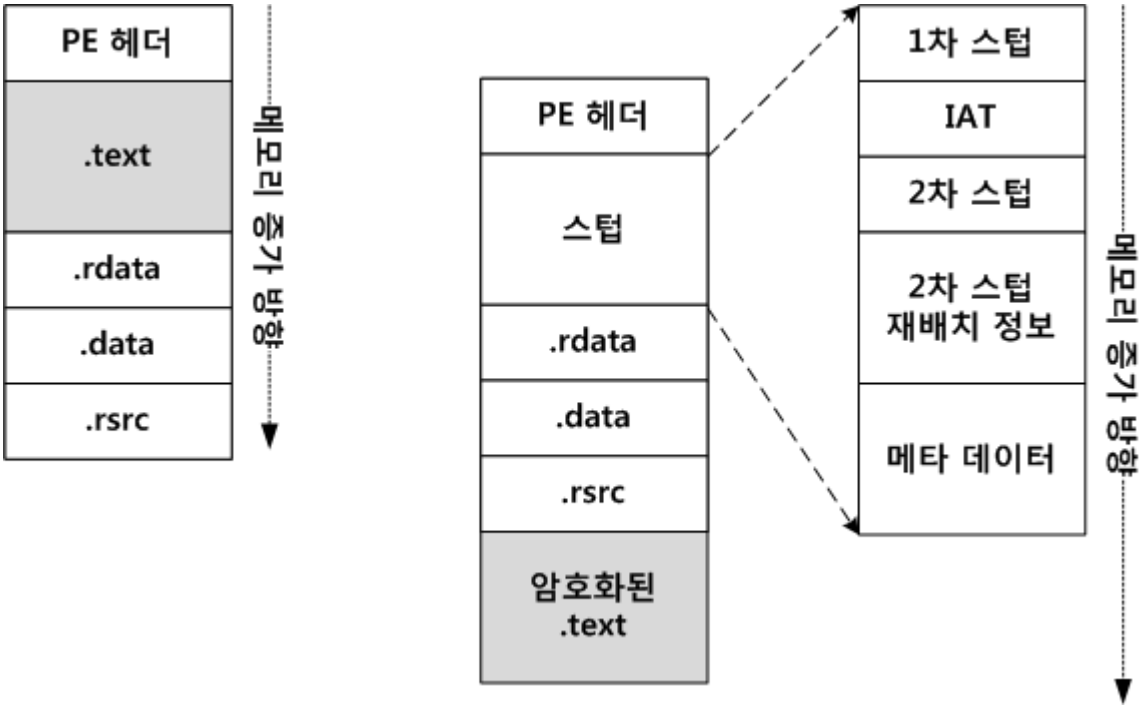


그림 1 일반적인 프로텍터의 메모리 구조

<그림 1>과 같이 2단계로 구성되도록 만들 경우에 스텁 코드의 위치가 자유롭다는 것과 스텁 코드 보안이 강화된다는 장점이 있는 반면 구현이 쉽지 않다는 단점이 있다. 실제로 2단계의 스텁 코드를 완전 자동화 시키기 위해서는 중간 계층의 코드 관리기를 별도로 만들어야 하는 복잡함이 따른다.

구현 상의 복잡함 때문에 이번 시간에 제작해볼 Z 프로텍터의 경우에는 <그림 2>와 같은 메모리 구조를 가지도록 만들었다. 스텁 코드는 하나로 구성되고, 원본 코드에 영향을 주지 않기 위해서 파일 끝에 위치한다. 구조만 살펴 본다면 지난 번에 작성했던 바이러스와 별반 다를 게 없어 보인다. 하지만 내부적으로는 많이 다른 구조를 가진다. 별도의 IAT를 가지고 프로텍터에 의해서 재배치 되어서 실행 파일에 추가된다는 점이 큰 차이라 할 수 있다.

스텁은 실제로 세 부분으로 구성된다. 실행될 코드와 코드에서 사용하는 API 정보를 담은 IAT와 스텁 코드가 수행되는데 필요한 정보를 담고 있는 메타 데이터가 그것이다. 프로텍터는 원본 파일의 IAT를 스텁 IAT로 교체할 것이다. 원본 실행 파일의 IAT는 스텁 코드가 실행될 때 빌드해준다.

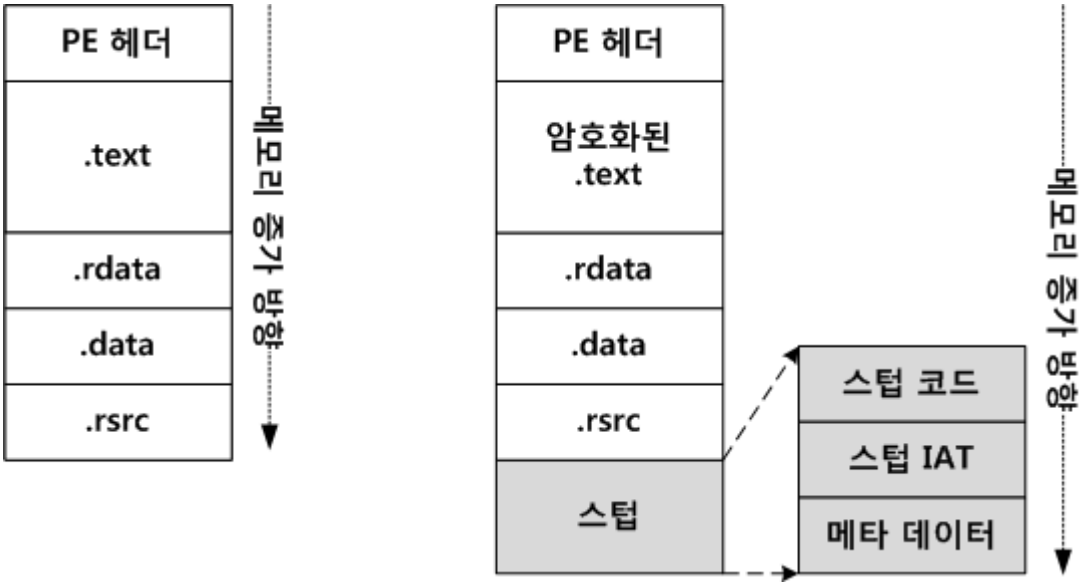


그림 2 간단한 프로텍터의 메모리 구조

IAT 처리하기

사각형 몇 개 그려놓고 그림으로 살펴보면 쉬워 보이지만 실제 세부 구현으로 들어가면 만만한 작업이 하나도 없다. 그 중에서도 특히 IAT를 처리하는 것은 꽤나 복잡한 작업이다. 컴파일러가 전적으로 관여해서 만들어내기 때문이다. 간단하게 문제의 복잡도를 살펴보기 위해서 컴파일러가 GetModuleHandle(0)을 번역한 어셈블러를 살펴보자.

```
// GetModuleHandle(0);
mov esi, 0x00402000
push 0
call esi
```

눈에 보이는 것처럼 0x00402000이란 숫자가 문제가 된다. 이 코드가 정상적으로 동작하기 위해서는 0x00402000에 GetModuleHandle의 주소가 들어 있거나, 0x00402000을 GetModuleHandle의 주소가 들어있는 메모리 번지로 변경해 주어야 한다. 첫 번째 방법은 우리가 만들 프로텍터 스텁 코드가 사용할 수 있는 영역이 제한적이라는 점 때문에 사용할 수 없다. <그림 2>에 나와 있는 것처럼 우리가 만든 프로텍터의 스텁 코드가 쓸 수 있는 공간은 마지막 섹션 밖에는 없다. 0x00402000이란 번지를 우리 맘대로 쓸 수 없는 것이다. 두 번째 방법의 경우는 구현이 가능하긴 하지만 매우 복잡해진다는 점이 치명적이다. 디스어셈블러를 사용해서 모든 패턴을 조사하면 될 것 같지만 더 어려운 문제는 call과 0x00402000이 떨어져 있다는 점이다.

아마 이번 연재를 계속 읽어온 독자라면 IAT를 사용하지 말고 바이러스를 만들 때 사용했던 방법을 사용하면 안 되는가에 대한 의문을 가질 것 같다. 바이러스 제작 때 사용했던 방법에는 두 가지 큰 문제가 존재한다. 복잡한 코드를 만들기가 쉽지 않다는 것과 실험에 의한 특수한 가정에 기반한다는 점이 그것이다. 실행 파일 프로텍터의 경우는 범용적으로 사용되는 프로그램이기 때

문에 이러한 단점을 용납하기는 쉽지 않다.

여기서는 이 문제를 특수한 형태로 제작된 DLL을 사용해서 해결했다. 컴파일러가 제작해주는 IAT를 포기하고 다른 실행 파일로 이식하기 쉽게 만들기 위해서 우리가 직접 IAT를 제작하도록 할 것이다. 다소 복잡한 방법이기 그림을 보면서 전체 구조를 머릿속에 담아 두도록 하자.

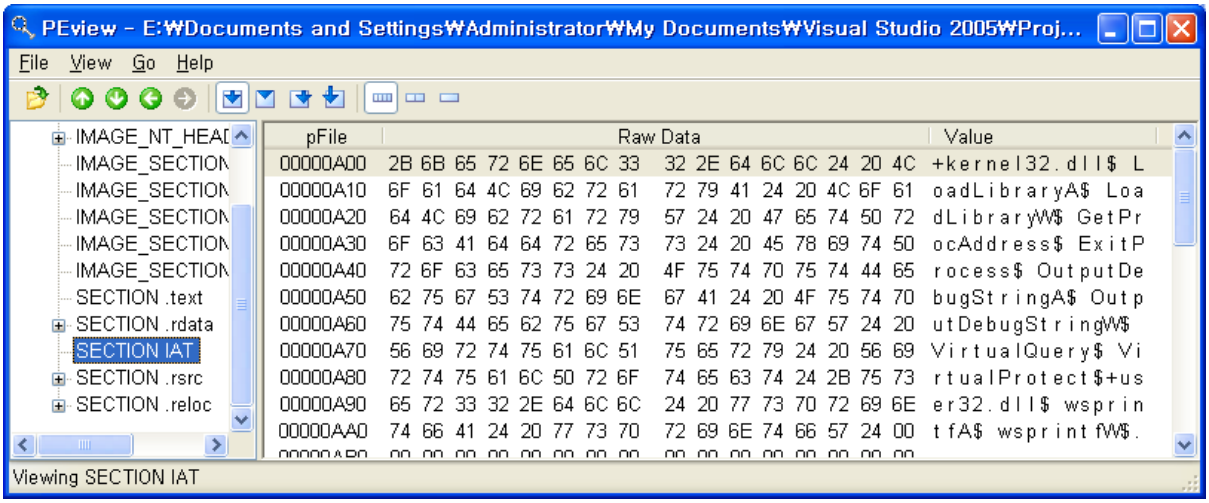
우리가 사용할 특수한 DLL을 제작할 때 가장 먼저 해야 할 일은 자신이 사용할 API 목록을 제작하는 것이다. 목록을 제작하는 방법이 <리스트 1>에 나와 있다. IAT란 섹션을 만들어서 그곳에 사용하고자 하는 함수목록을 가지고 있는 DLL과 함수 명을 적어준다. DLL 이름는 +로 시작해서 $으로 끝나고 함수 이름은 스페이스로 시작해서 $으로 끝나도록 만들어준다. 그리고 함수 진입점에 실제 함수 주소가 저장될 공간을 마련해 준다. 각 함수당 4바이트의 공간이 필요하고, DLL이 바뀔 때 NULL주소가 들어가야 하기 때문에 DLL이 바뀌는 지점에는 추가적인 공간이 필요하다는 점을 기억하자.

리스트 1 스텁 코드에서 사용하기 위한 API 목록

```
#pragma data_seg("IAT")
char IAT[] =
        "+kernel32.dll$ LoadLibraryA$ LoadLibraryW$"
        " GetProcAddress$ ExitProcess$ OutputDebugStringA$"
        " OutputDebugStringW$ VirtualQuery$ VirtualProtect$"
        "+user32.dll$ wsprintfA$ wsprintfW$";
#pragma data_seg()

#define DD(); __asm { __asm nop __asm nop __asm nop __asm nop }

void Entry()
{
    __asm call Stub1

    DD(); // 메타 데이터 오프셋
    DD(); // LoadLibraryA 주소
    DD(); // LoadLibraryW 주소
    DD(); // GetProcAddress 주소
    DD(); // ExitProcess 주소
    DD(); // OutputDebugStringA 주소
    DD(); // OutputDebugStringW 주소
    DD(); // VirtualQuery 주소
    DD(); // VirtualProtect 주소
    DD(); // kernel32.dll 종료
```

```
    DD(); // wsprintfA 주소
    DD(); // wsprintfW 주소
    DD(); // user32.dll 종료
}
```

이렇게 만든 후에 DLL을 빌드하면 <화면 1>에 나타난 것과 같이 IAT 섹션이 추가되고 그곳에 우리가 사용할 함수 이름이 가지런히 들어가 있게 된다. Z 프로텍터는 이 정보를 참조해서 IAT를 생성한다.



화면 1 컴파일된 DLL

<리스트 2>와 <리스트 3>에 IAT 섹션의 문자열을 분석해서 자료 구조를 생성하는 코드가 나와 있다. 가장 핵심적인 자료 구조는 DLL_ENTRY다. DLL_ENTRY는 가져다 쓰는 각각의 DLL을 저장하는 용도로 사용된다. DLL_ENTRY의 name은 dll 이름이, funcs에는 그 dll에서 가져다 쓰는 함수 이름이 저장된다. 사용하는 DLL의 전체 목록은 m_iat에 저장된다. m_iatSize는 IAT를 기록하기 위해서 할당해야 하는 크기를 나타내고, m_functionSize는 룩업 테이블(IMPORT LOOKUP TABLE)을 저장하기 위해서 할당해야 하는 크기를 나타낸다.

코드가 금방 이해하기 힘들다면 우선 <그림 3>을 살펴보도록 하자. <그림 3>은 우리가 생성할 최종적인 IAT 구조를 나타내고 있다. 왼쪽 그림은 메모리 상에서 각 데이터가 위치하고 있는 구조를, 오른쪽은 각 구조가 어떤 형태로 연결이 되어 있는지를 보여준다.

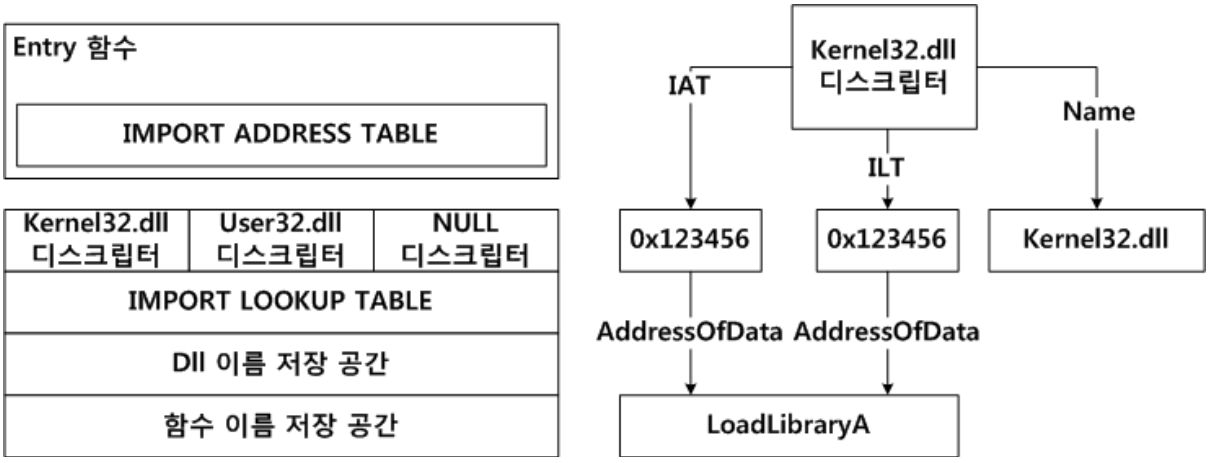


그림 3 IAT 구조

리스트 2 IAT 생성에 사용되는 자료 구조

```
typedef struct _DLL_ENTRY
{
	string name; // dll 이름
	StringVec funcs; // 함수 이름 목록
} DLL_ENTRY, *PDLL_ENTRY;

typedef std::vector<DLL_ENTRY> DllEntryVec;
typedef std::vector<DLL_ENTRY>::iterator DllEntryVit;

DllEntryVec m_iat; // DLL, 함수 이름 저장 공간
DWORD m_iatSize; // IAT 크기
DWORD m_functionSize; // 함수 크기
```

리스트 3 IAT 생성 함수

```
void CStubCode::MakeIAT(LPBYTE data)
{
	char name[MAX_PATH];
	int index = 0;
	FSM_STATE state = START;
	DLL_ENTRY entry;

	m_iat.clear();
	m_iatSize = sizeof(IMAGE_IMPORT_DESCRIPTOR);
	m_functionSize = 0;

	while(*data)
```

```
{
	switch(state)
	{
	case START:
		if(*data == '+')
			state = DLL;
		else if(*data == ' ')
			state = FUNC;
		break;

	case DLL:
		if(*data == '$')
		{
			name[index] = '₩0';
			index = 0;
			state = START;

			entry.name = name;
			entry.funcs.clear();
			m_iat.push_back(entry);

			m_iatSize += sizeof(IMAGE_IMPORT_DESCRIPTOR);
			m_iatSize += entry.name.length() + 1;
			m_functionSize += sizeof(DWORD);
		}
		else
		{
			name[index] = *data;
			++index;
		}
		break;

	case FUNC:
		if(*data == '$')
		{
			name[index] = '₩0';
			index = 0;
			state = START;
```

```
                    m_iat.back().funcs.push_back(name);

                    m_iatSize += strlen(name) + 1 + sizeof(WORD);
                    m_functionSize += sizeof(DWORD);
                }
                else
                {
                    name[index] = *data;
                    ++index;
                }
                break;
            }
            ++data;
        }

        m_iatSize += m_functionSize;
}
```

실제로 생성된 구조체를 사용해서 IAT를 기록하는 부분이 <리스트 4>에 나와 있다. 원본 코드에서 IAT 관련 부분만 발췌한 것이다. 인자로 넘어오는 buf는 스텁 데이터가 기록될 버퍼를, bufSize는 버퍼 크기를, imageBase는 기록될 이미지의 기준 주소를, codeBase는 기록될 이미지에서 코드가 들어갈 부분의 RVA를 나타낸다. functions는 Entry에 포함된 실제 IAT가 저장되는 부분의 시작 주소를, lookups는 룩업 테이블의 시작 주소를, dllNames는 dll 이름을 기록할 부분의 시작 주소를, funcNames는 함수 이름을 기록할 부분의 시작 주소를 나타낸다. 각 오프셋의 의미와 전체 구조(<그림 3> 참고)를 정확하게 이해하고 있다면 코드를 쉽게 이해할 수 있다. 한 가지 중요한 사실은 IAT, ILT, 디스크립터 테이블 모두 마지막에 NULL을 채워주어야 한다는 점이다.

리스트 4 IAT 기록 함수

```
void CStubCode::Write(LPBYTE buf, SIZE_T bufSize, DWORD imageBase, DWORD codeBase)
{
    IMAGE_IMPORT_DESCRIPTOR idesc;
    LPBYTE iat = buf + CodeSize();

    LPDWORD lookups = (LPDWORD) (iat + (m_iat.size() + 1) * sizeof(idesc));
    LPBYTE dllNames = (LPBYTE) lookups + m_functionSize;
    LPBYTE funcNames = dllNames;
    LPDWORD functions = (LPDWORD)(buf + m_entry + 9);

    StringVit sit;
```

```
StringVit send;

DllEntryVit it;
DllEntryVit end = m_iat.end();
for(it = m_iat.begin(); it != end; ++it)
	funcNames += (*it).name.length() + 1;

idesc.ForwarderChain = 0;
idesc.TimeDateStamp = 0;
for(it = m_iat.begin(); it != end; ++it)
{
	idesc.OriginalFirstThunk = Offset(lookups, buf, codeBase);
	idesc.FirstThunk = Offset(functions, buf, codeBase);
	idesc.Name = (DWORD)(dllNames - buf) + codeBase;
	memcpy(iat, &idesc, sizeof(idesc));
	iat += sizeof(idesc);

	strcpy((char *) dllNames, (*it).name.c_str());
	dllNames += (*it).name.length() + 1;

	send = (*it).funcs.end();
	for(sit = (*it).funcs.begin(); sit != send; ++sit)
	{
		*functions = Offset(funcNames, buf, codeBase);
		++functions;

		*lookups = Offset(funcNames, buf, codeBase);
		++lookups;

		funcNames[0] = 0;
		funcNames[1] = 0;
		strcpy((char *) funcNames+2, (*sit).c_str());
		funcNames += (*sit).length() + 3;

	}

	*functions = 0;
	++functions;
```

```
            *lookups = 0;
            ++lookups;
        }

        idesc.OriginalFirstThunk = 0;
        idesc.FirstThunk = 0;
        idesc.Name = 0;
        memcpy(iat, &idesc, sizeof(idesc));
}
```

메타 데이터

메타 데이터는 스텁 코드와 원본 실행 파일 사이를 연결해 주는 데이터다. 프로텍터에 의해서 훼손된 원본 데이터 중에서 나중에 필요한 데이터를 저장해 두는 공간이라고 생각하면 쉽다. 필수적으로 저장해야 하는 정보로는 원본 실행 파일의 진입점과 IAT다. <리스트 5>에는 Z 프로텍터에 사용된 메타 데이터의 구조체가 나와 있다. 원본 코드 오프셋은 암호화된 부분을 복호화 시키기 위해서 사용된다.

리스트 5 메타 데이터 구조체

```
typedef struct _METADATA
{
    DWORD codeOffset; // 원본 코드 오프셋
    DWORD codeSize; // 원본 코드 사이즈

    IMAGE_DATA_DIRECTORY import; // 원본 IAT
    DWORD originalEntry; // 원본 진입점
} METADATA, *PMETADATA;
```

이런 필수적인 정보 외에도 추가적으로 필요한 정보는 어떠한 것이든 포함시킬 수 있다. 예를 들어 사용자가 지정한 암호를 입력해야만 프로그램이 실행되도록 해주는 프로텍터라면 메타 데이터에 사용자가 입력한 암호가 저장되어야 할 것이다.

프로텍터가 DLL이나 OCX를 지원하도록 만들기 위해서는 원본 실행 파일의 재배치 정보도 저장해둘 필요가 있다. 더불어 IAT를 처리한 방법과 같은 형태로 스텁 코드에 대한 재배치 정보도 재구성해서 같이 저장해 주어야 한다.

스텁 코드

앞서 설명한 내용을 충분히 이해했다면 스텁 코드를 만드는 일은 간단한 작업이다. <리스트 6>에 Z 프로텍터의 스텁 코드 전문이 나와 있다. 이 코드는 프로그램 실행 전에 간단한 디버그 출력을

하고 코드 섹션을 복호화한 다음 원본 코드를 실행시켜 준다.

핵심은 함수들을 선언해서 사용하는 부분이다. 앞서 살펴보았던 IAT 처리하기 부분과 연결해서 살펴보도록 하자. 모든 데이터를 코드 섹션에 모으기 위해서 전역 변수는 .L 섹션에 선언하고 링커 명령어를 통해서 .text 섹션과 병합해 준다. 프로텍터가 재배치를 수행해주기 때문에 바이러스를 만들 때처럼 전역 변수를 복잡하게 사용할 필요는 없다.

한 가지 주의해야 할 점은 디버그 버전으로 빌드된 스텁 코드의 경우 단순히 .text 섹션만으로 동작하지 않는다는 점이다. 따라서 디버그 버전으로 빌드된 스텁 코드를 Z 프로텍터로 실행 파일과 병합시킬 경우에는 잘못된 연산 오류를 만날 수 있다. 스텁 코드는 항상 릴리즈 버전으로 빌드해서 사용하도록 한다.

리스트 6 프로그램 실행 전에 간단한 디버그 메시지를 출력하는 스텁 코드

```
#pragma comment(linker, "/section:.text,rwe")
#pragma comment(linker, "/merge:.L=.text")

#define METADATA_OFFSET 5
#define LOADLIBRARYA_OFFSET (METADATA_OFFSET + 4)
#define LOADLIBRARYW_OFFSET (METADATA_OFFSET + 8)
// ... 중략 ...

#define InitAPI(V, B, O) (*(FARPROC *) V = *(FARPROC *) GetPtr(B, O))

#pragma data_seg(".L")
HMODULE (WINAPI *g_LoadLibraryA)(LPCSTR) = NULL;
HMODULE (WINAPI *g_LoadLibraryW)(LPCWSTR) = NULL;
FARPROC (WINAPI *g_GetProcAddress)(HMODULE, LPCSTR) = NULL;
// ... 중략 ...

char g_msg[] = "Z 프로텍터₩n";
PVOID g_base = NULL;
PMETADATA g_meta = NULL;
#pragma data_seg()

void Decrypt()
{
    LPBYTE ptr;

    ptr = (LPBYTE) GetPtr(g_base, g_meta->codeOffset);
```

```
	DWORD oldProtect;
	g_VirtualProtect(ptr, g_meta->codeSize, PAGE_EXECUTE_READWRITE, &oldProtect);
	for(DWORD i=0; i<g_meta->codeSize; ++i)
		ptr[i] ^= 0x7f;

	g_VirtualProtect(ptr, g_meta->codeSize, oldProtect, NULL);
}

void Stub1()
{
	InitAPI(&g_LoadLibraryA, Entry, LOADLIBRARYA_OFFSET);
	InitAPI(&g_LoadLibraryW, Entry, LOADLIBRARYW_OFFSET);
	InitAPI(&g_GetProcAddress, Entry, GETPROCADDRESS_OFFSET);
	// ... 중략 ...

	g_OutputDebugStringA(g_msg);

	MEMORY_BASIC_INFORMATION mbi;
	g_VirtualQuery(Stub1, &mbi, sizeof(mbi));
	g_base = mbi.AllocationBase;

	DWORD *meta_offset;
	GetPtr(&meta_offset, Entry, METADATA_OFFSET);
	g_meta = (PMETADATA) GetPtr(g_base, *meta_offset);

	// 복호화
	Decrypt();

	// 원본 IAT 빌드
	BuildIAT();

	// 원본 코드로 점프
	DWORD oep;
	GetPtr(&oep, g_base, g_meta->originalEntry);
	__asm push oep
	__asm ret
}
```

Z 프로텍터

이제 실제 스텁 코드를 실행 파일과 연결 시키는 프로텍터 부분을 살펴보도록 하자. 이 부분은 기본적인 PE 파일의 구조에 대해서만 이해하고 있다면 어려운 부분이 아니다. 단지 작업 과정이 복잡하기 때문에 어떤 부분을 해야 하는지를 정확하게 알아둘 필요가 있다.

전체 작업 과정을 개략적으로 살펴보면 아래 나와 있는 것과 같다. IAT 생성 부분은 앞에서 살펴보았고, 재배치나 섹션을 추가하는 등의 작업은 이전 연재를 통해서 충분히 다룬 내용이기 때문에 특별히 어려운 부분은 없다. 각 작업 과정과 그 부분이 소스에서 어떻게 구현되어 있는지를 연결시켜 보도록 하자. 섹션의 정렬(align)을 맞추지 않는 것과 같은 사소한 실수 때문에 실행 파일이 동작하지 않는 일이 많기 때문에 PE 헤더를 수정할 때에는 항상 주의를 해야 한다. 전반적인 과정에 대한 소스 코드가 <리스트 7>에, NT 헤더에서 수정해야 할 부분에 대한 별도의 설명이 <표 1>에 나와 있다. <화면 2>에는 스텁 코드와 프로텍터를 빌드해서 실행 시키는 방법이 나와 있다.

1. 스텁을 추가할 섹션 생성
2. 메타 데이터 수집
3. 스텁 코드 기록
4. 스텁 코드 재배치
5. 스텁 IAT 생성 및 기록
6. 메타 데이터 기록
7. 추가할 섹션 헤더 생성 및 기록
8. NT 헤더 수정
9. 원본 코드 암호화

스텁 코드의 IAT 생성과 재배치를 담당하는 CStubCode, PE 포맷을 추상화한 CRawPeformat, 메모리 맵을 쓰기 편하게 만든 CMmap 등은 지면 관계상 담지 못했다. 이 부분이 궁금한 독자들은 이달의 디스크를 참고하도록 하자.

표 1 NT 헤더에서 수정해야 할 부분들

수정 부분	설명
AddressOfEntryPoint	스텁 코드의 진입점으로 변경해 주어야 한다.
SizeOfImage	추가된 스텁 코드의 크기만큼 증가시켜 주어야 한다.
IMAGE_DIRECTORY_ENTRY_IMPORT	추가된 스텁 코드의 임포트 정보를 가리키도록 변경해 주어야 한다.
NumberOfSection	스텁 코드를 별도의 섹션에 저장한 경우에는 1증가 시켜 준다.
BaseOfCode	스텁 코드가 저장된 섹션의 시작 주소로 변경해 준다.

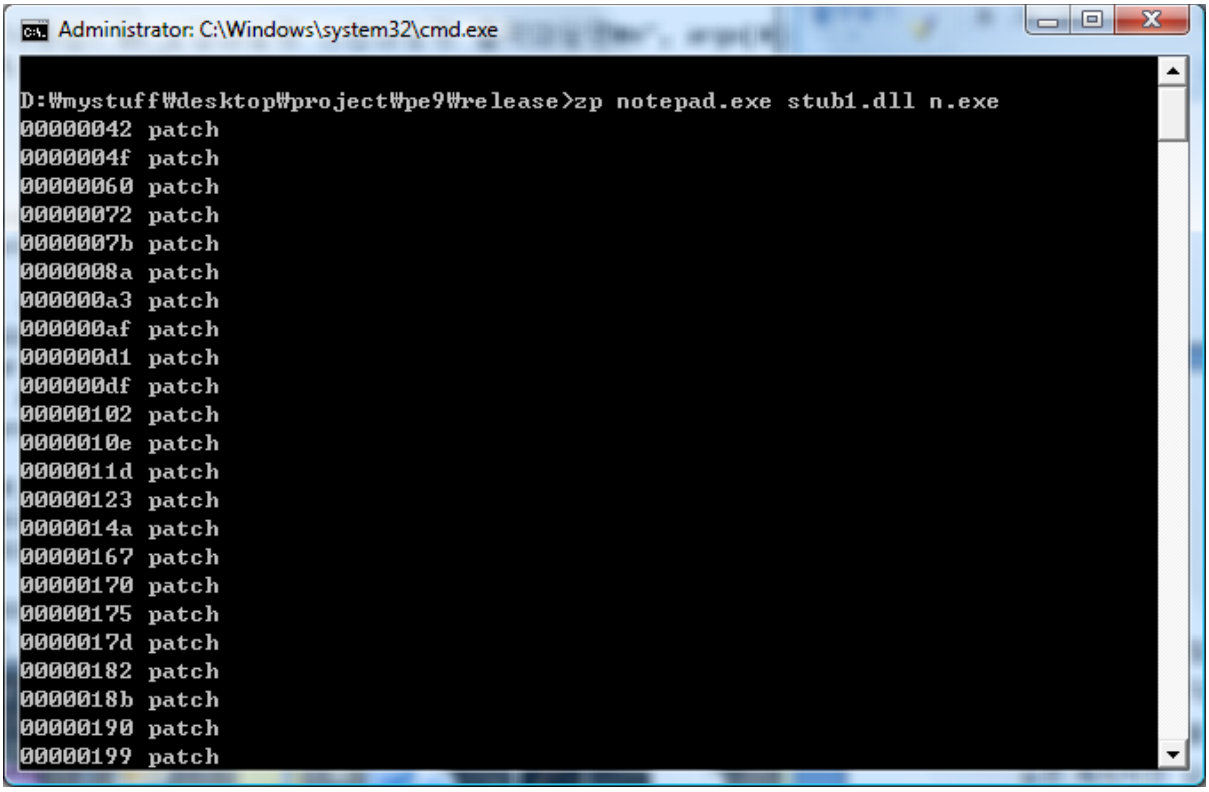
리스트 7 Z 프로텍터 소스 코드

```
int _tmain(int argc, TCHAR *argv[])
{
	if(argc < 4)
	{
		printf("사용법: %s 실행파일명 스텁파일명 출력파일명₩n", argv[0]);
		return 0;
	}

	try
	{
		CMmap mmap(argv[1], 0);
		CRawPeformat pe(mmap.Ptr());

		int nSec = pe.NumberOfSection();
		PIMAGE_SECTION_HEADER sec = pe.SectionHeader(nSec-1);
		DWORD eof = sec->PointerToRawData + sec->SizeOfRawData;
		if(eof < mmap.FileSize())
			return 0;

		PIMAGE_NT_HEADERS nt = pe.NTHeader();
		DWORD sAlign = nt->OptionalHeader.SectionAlignment;
		DWORD fAlign = nt->OptionalHeader.FileAlignment;

		PIMAGE_SECTION_HEADER codeSection = FindCodeSection(pe);
		if(codeSection == NULL)
		{
			printf("오류: 코드 섹션을 찾을 수 없습니다.");
			return 0;
		}

		CStubCode stub(argv[2]);

		METADATA meta;
		meta.codeOffset = codeSection->VirtualAddress;
		meta.codeSize = codeSection->Misc.VirtualSize;
		meta.originalEntry = pe.EntryPoint();
		meta.import = *pe.DataDirectory(IMAGE_DIRECTORY_ENTRY_IMPORT);
		stub.MetaData(&meta, sizeof(meta));
```

```
FILE *fp;
_tfopen_s(&fp, argv[3], _T("wb"));
if(!fp)
{
	printf("오류: 파일을 열 수 없습니다.₩n");
	return 0;
}

// 파일 전체 복사
fwrite(mmap.Ptr(), 1, mmap.FileSize(), fp);

// 스텁 섹션 헤더 기록
IMAGE_SECTION_HEADER secHdr;
secHdr.Characteristics = IMAGE_SCN_CNT_CODE
                       | IMAGE_SCN_MEM_EXECUTE
                       | IMAGE_SCN_MEM_READ
                       | IMAGE_SCN_MEM_WRITE;

secHdr.Misc.VirtualSize = stub.Size() + 10;
secHdr.NumberOfLinenumbers = 0;
secHdr.NumberOfRelocations = 0;
secHdr.PointerToLinenumbers = 0;
secHdr.PointerToRelocations = 0;
secHdr.PointerToRawData = Align(sec->PointerToRawData
                                + sec->SizeOfRawData
                                , fAlign);

secHdr.SizeOfRawData = Align(stub.Size() + 10, fAlign);
secHdr.VirtualAddress = Align(sec->VirtualAddress
                                + sec->Misc.VirtualSize
                                , sAlign);
memcpy(secHdr.Name, ".Z", 3);

fseek(fp, mmap.Offset(sec+1), SEEK_SET);
fwrite(&secHdr, sizeof(secHdr), 1, fp);

// NT 헤더 수정
IMAGE_NT_HEADERS hdr;
```

```
        memcpy(&hdr, pe.NTHeader(), sizeof(hdr));
        hdr.OptionalHeader.SizeOfImage = Align(hdr.OptionalHeader.SizeOfImage +
secHdr.Misc.VirtualSize, sAlign);
        hdr.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress =
secHdr.VirtualAddress + stub.CodeSize();
        hdr.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].Size =
stub.IATSize();
        hdr.OptionalHeader.AddressOfEntryPoint = secHdr.VirtualAddress + stub.Entry();
        hdr.FileHeader.NumberOfSections++;
        hdr.OptionalHeader.BaseOfCode = secHdr.VirtualAddress;

        fseek(fp, mmap.Offset(pe.NTHeader()), SEEK_SET);
        fwrite(&hdr, sizeof(hdr), 1, fp);

        // 코드 섹션 암호화
        CCodeEncryptor enc(pe, codeSection);
        fseek(fp, enc.Offset(), SEEK_SET);
        fwrite(enc.Ptr(), 1, enc.Size(), fp);

        // 스텁 데이터 기록
        PBYTE buf = new BYTE[secHdr.SizeOfRawData];
        if(!buf)
        {
            printf("오류: 메모리가 부족합니다.₩n");
            fclose(fp);
            return 0;
        }

        ZeroMemory(buf, secHdr.SizeOfRawData);
        stub.Write(buf
                    , secHdr.Misc.VirtualSize
                    , nt->OptionalHeader.ImageBase
                    , secHdr.VirtualAddress);

        fseek(fp, secHdr.PointerToRawData, SEEK_SET);
        fwrite(buf, 1, secHdr.SizeOfRawData, fp);

        fclose(fp);
    }
```

```
    catch(EWin32 &err)
    {
        printf("오류: %d₩n", err.ErrorCode());
    }

    return 0;
}
```



화면 2 Z 프로텍터 실행 화면

도전 과제

이번 시간에 진행한 내용은 실행 파일 프로텍터 제작을 위한 기반 구조에 관한 내용들이었다. 실제 프로텍터 제작은 여러분이 어느 정도로 강력한 스텁 코드를 제작하는가에 달려있다. 지난 시간에 배운 내용을 토대로 다양한 스텁 코드를 제작해보자. 심도 있는 PE 프로텍터를 제작해 보고 싶은 독자라면 상용 PE 프로텍터에 포함된 기능들을 구현해 보는 것도 도움이 될 것 같다.

참고자료

Make your owner PE Protector Part 1: Your first EXE Protector
http://www.codeproject.com/KB/cpp/peprotector1.aspx