윈도우 프로그래머를 위한 PE 포맷 가이드
실행 압축의 원리

목차

소개

실행 압축이란 실행 파일을 압축시켜 저장하고, 실행을 하면 그것을 자동으로 해제해서 원본 파일이 그대로 실행되도록 해주는 프로그램이다. 이는 약간의 로딩 속도를 손해 보는 대신 공간과 편리함이라는 두 마리 토끼를 모두 잡은 재미있는 기술이다. 이번 시간에는 이러한 실행 압축의 뒤에 숨어있는 원리에 대해서 살펴본다.

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 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

인이라는 것은 활 쏘는 것과 같다. 활을 쏠 때는 자세를 바르게 한 후에 쏘는 법이다. 화살이 과녁에 맞지 않으면 자기를 이긴 자를 원망할 것이 아니라 (과녁에 맞지 않은 까닭을) 도리어 자기 자신에게서 찾는다.

맹자에 나오는 말이다. 우리는 많은 경우에 문제의 원인을 외부에서 찾으려고 한다. 버그가 발생하면 항상 라이브러리 탓을 먼저 하는 개발자들이 많은 걸로 봐서는 개발자들에게도 이러한 경향이 다분히 강한 것 같다. 하지만 이러한 자세는 문제 해결에는 큰 도움이 되지 않는다. 왜냐하면 대부분의 경우에 문제의 핵심 원인이 자신에게 있는 경우가 많기 때문이다. 물론 원인이 정말 외부에 있다고 하더라도 이러한 겸손한 자세는 자신의 코드를 한번 더 돌아볼 수 있는 기회를 만들어 준다. 천하를 다스리는 일도 자신을 다스린 연후에나 가능한 것이다.

Introduction

우리가 생활 속에서 결정하는 많은 일들에는 늘 기회 비용이 따른다. 시간은 중첩될 수 없기 때문에 WoW를 하면서 동시에 데이트를 즐길 순 없는 것이다. 이것은 컴퓨터 속 세상에도 똑같이 적용된다. 공간을 많이 사용하면 빠르게 만들 수 있고, 느리게 만들어도 된다면 공간은 적게 사용할 수 있다. 항상 그 사이에서 결정을 해야 한다. 일상 생활에서 이러한 기회 비용을 잘 살리는 사람들은 인기가 있다. 마찬가지로 컴퓨터 속 세상에 사용되는 기술에도 이러한 것은 똑같이 적용된다. 트레이드오프를 잘 한 기술을 대중의 선택을 받게 되고 업계 표준이 되는 경우가 많다.

우리가 이번 시간에 살펴볼 실행 압축이란 기술도 이러한 트레이드오프가 잘 이루어진 기술 중에 하나다. 과거 디스크 공간이 귀하던 시절이 있었다. 그 시절엔 한 바이트라도 아끼는 것이 미덕이었다. 압축을 해서 저장을 하는 것은 일상 생활이었다. 하지만 실행 파일을 압축해서 저장한다면 매번 다시 압축을 해제해서 실행을 해야 한다는 불편함이 있었다. 이러한 공간 절약의 욕구와 편리함을 모두 충족시키면서 약간의 로딩 속도를 감수하면서 탄생한 기술이 실행 압축이다.

이전에 살펴보았던 자동 풀림 압축 파일과 실행 압축의 가장 큰 차이점은 결과물에 있다. 자동 풀림 압축 파일은 최종 결과물이 압축 해제된 파일이다. 반면에 실행 압축은 압축되기 전의 파일이 그대로 메모리에서 실행되도록 해야 한다. 따라서 그때보다는 좀 더 실행 파일의 메모리 구조에 대해서 잘 알아야 하고 좀 더 복잡하게 스텁 코드를 다루어야 한다. 이번 시간에는 upx를 통해서 실행 압축을 하는 방법과 그 속에 숨겨진 원리에 대해서 살펴본다.

upx

이제는 역사의 뒤안길로 사라진 워크맨이 휴대용 카세트 레코더의 대명사였다면, 실행 압축 프로그램의 대명사는 upx라고 할 수 있겠다. upx는 실행 압축이라는 말이 생소하던 시절부터 존재했으며, 지금까지 많은 사람들이 사용하고 있는 프로그램이다. 이번 시간 우리는 PE 포맷의 실행 압축에 대해서만 살펴보겠지만 upx는 PE 포맷외에도 다양한 플랫폼의 다양한 실행 파일에 대한 압축을 지원한다. 또한 오픈 소스 프로젝트라 누구나 원하기만 한다면 언제든지 소스코드를 다운로드 받아서 살펴볼 수 있다.

사실 이번 달 내용을 구구절절 읽는 것 보다 upx 소스 코드를 받아서 살펴보는 것이 더 큰 도움이 되지 않을까 생각해 본다. 딱딱한 원리 보다는 눈으로 직접 보면서 실습을 해보는 것이 이해가 빠르기 때문에 upx를 통해서 그 속에 숨겨진 원리를 찾아보도록 하자. upx는 http://upx.sourceforge.net에서 무료로 다운로드 받을 수 있다.

<리스트 1>에 upx 압축을 실습해볼 간단한 예제 프로그램이 나와 있다. 프로그램을 만든 다음 최적화 옵션을 끄고, 멀티 쓰레드 CRT를 사용해서 릴리즈 버전으로 컴파일한다. 필자가 사용한 Visual Studio 2005에서는 이렇게 컴파일해서 생성한 프로그램의 크기가 48k였다.

리스트 1 upxtest 프로그램 소스 코드

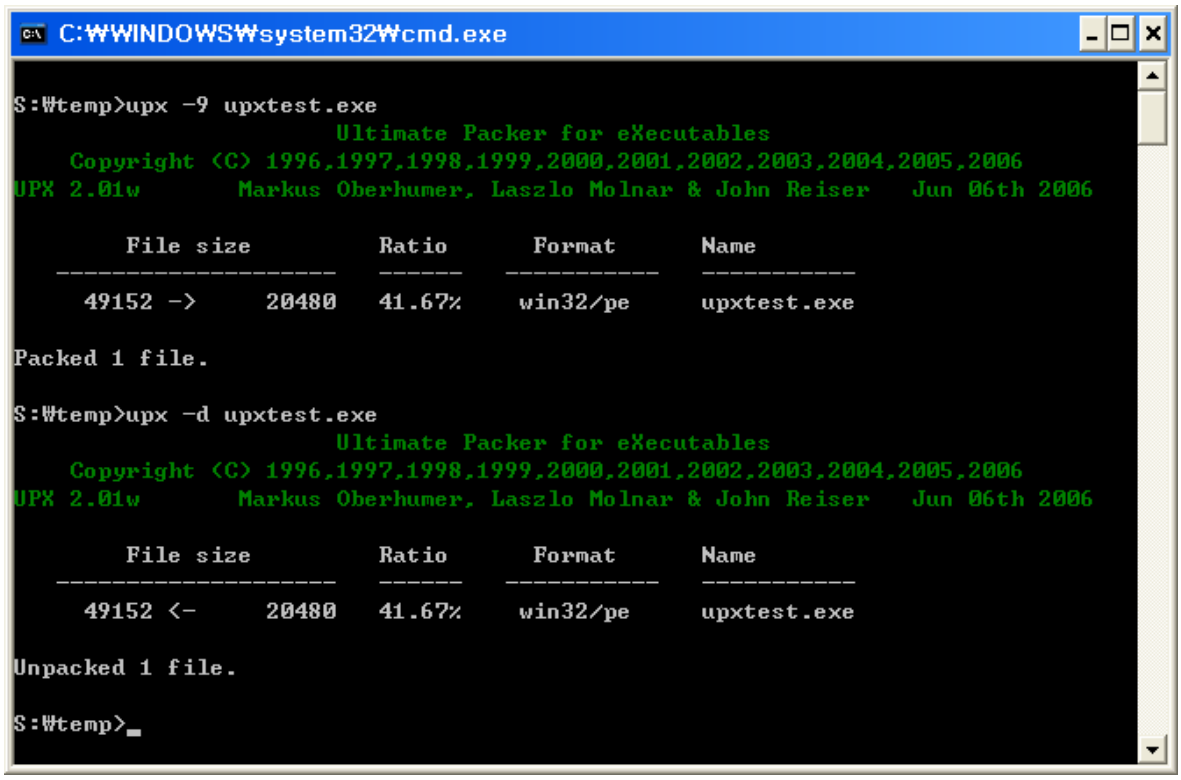
```
#include <windows.h>
#include <tchar.h>

int main()
{
	TCHAR msg[] = _T("Hello World");
	DWORD written;

	WriteConsole(GetStdHandle(STD_OUTPUT_HANDLE)
			, msg
			, sizeof(msg) / sizeof(TCHAR)
			, &written
```

```
                    , NULL);
    return 0;
}
```

압축할 때에는 upx -9를, 압축된 파일을 다시 복원할 때에는 upx –d를 사용하면 된다. 생성된 upxtest.exe 파일을 압축했다 해제하는 방법이 <화면 1>에 나와 있다. 필자가 사용한 버전의 upx로 압축할 경우 48k였던 실행 파일의 크기가 19k 정도로 줄어 들었다. upx의 보다 자세한 설명은 upx –h를 통해서 살펴보도록 하자.

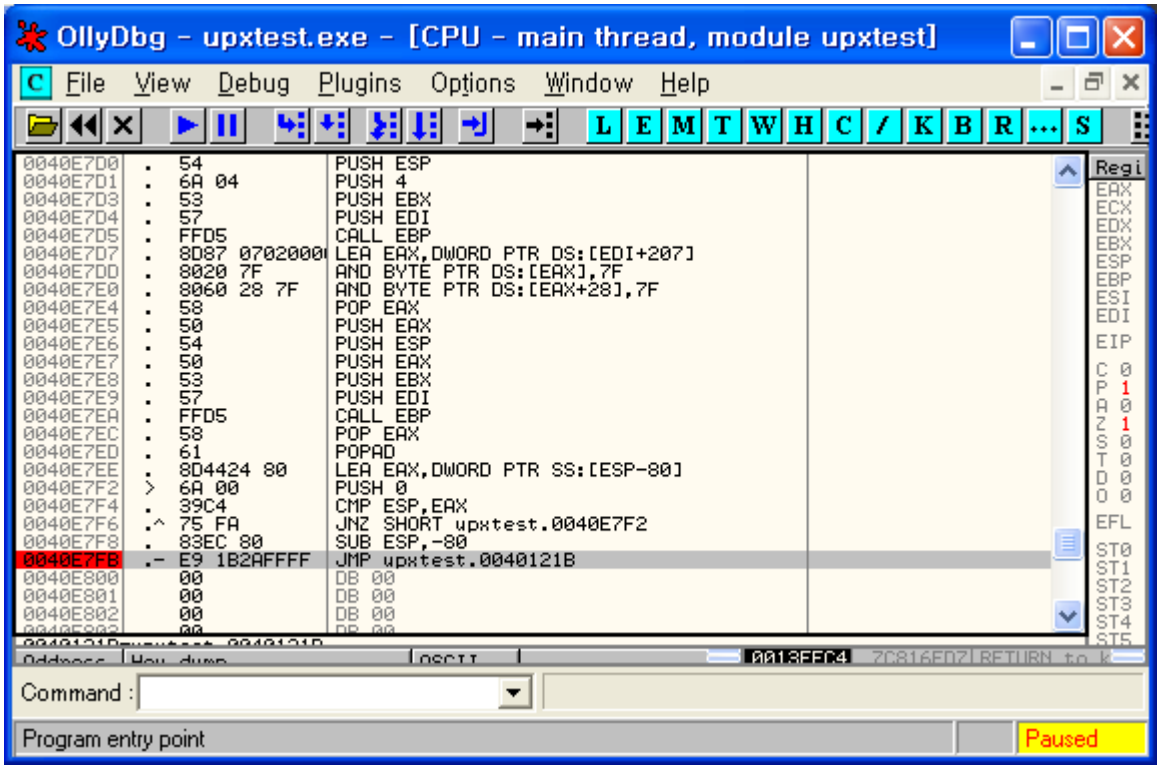


화면 1 upx 압축 및 해제 화면

매뉴얼 언팩

실행 압축을 하는 것을 '팩(pack)', 그것을 역으로 해제하는 작업을 '언팩(unpack)'이라는 말로 표현한다. 실행 압축 파일의 구조를 이해하는데 가장 도움이 되는 것은 실행 압축된 파일을 직접 손수 언팩시켜 보는 것이다. 도구를 사용하지 않고도 언팩을 할 수 있지만 여기서는 눈으로 확인하기 쉽게 하기 위해서 디버거를 사용해서 언팩을 해보도록 한다. 필요한 도구는 ollydbg와 ollydump다. ollydbg는 Windows용 디버거로 http://www.ollydbg.de에서, ollydump는 ollydbg에서 실행 파일 덤프를 떠주는 플러그인으로 http://www.openrce.org/downloads/details/108/OllyDump에서 무료로 다운로드 받을 수 있다.

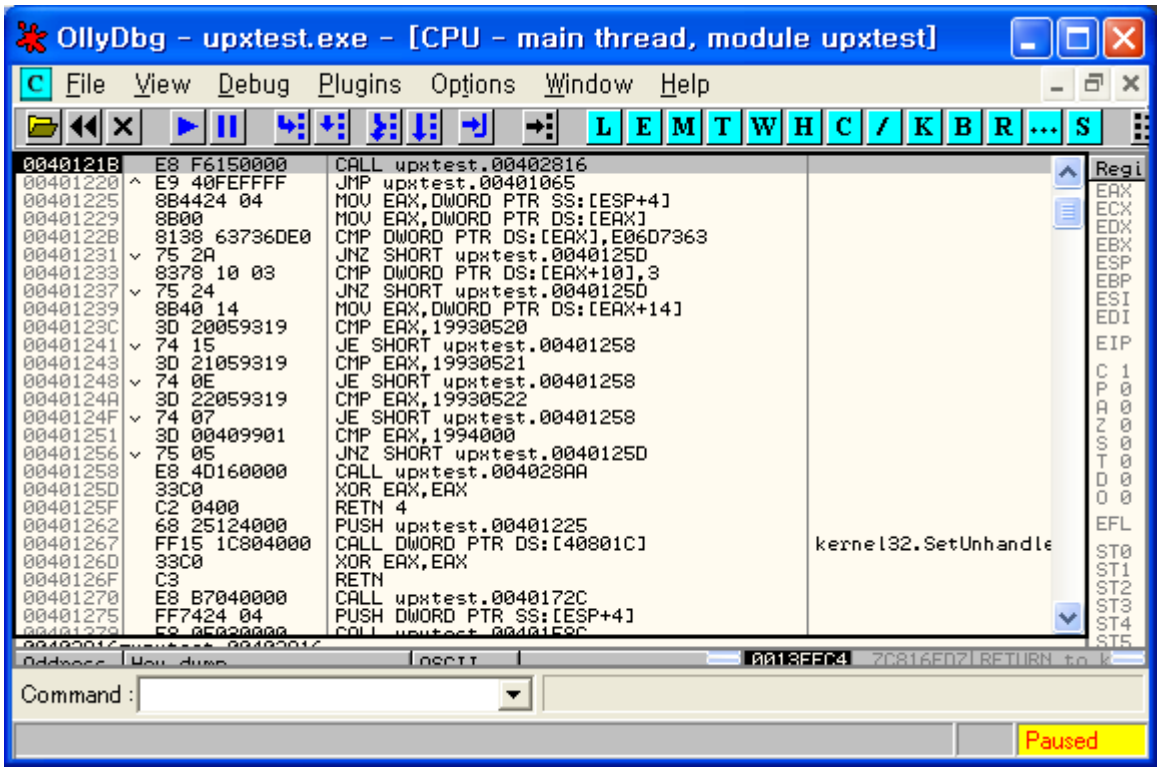
upx로 압축한 upxtest.exe를 ollydbg로 불러와 보자. 그럼 프로그램 진입점에서 브레이크 포인터가 걸린다. 스크롤바를 아래로 내려가다 보면 0x00 연속적으로 나타나는 부분이 보인다. 그 부분 바로 위에 JMP 명령어가 있는 곳에 브레이크 포인터를 걸도록 한다. 여기까지 실행하면 <화면 2>와 같이 된다.



화면 2 압축 해제 끝 부분에 브레이크 포인터

이제 F9를 눌러서 프로그램을 진행시킨다. 우리가 브레이크 포인터를 걸어둔 JMP 명령어에서 프로그램 실행이 중지될 것이다. 여기서 F8를 눌러 안으로 트레이스 해보도록 하자. 그러면 <화면 3>과 같은 내용이 출력된다. 여기가 실제 원본 프로그램의 진입점이다. 궁금하다면 원본 프로그램의 코드와 대조해 보도록 하자. 오른쪽 버튼을 눌러서 'dump debugged process'메뉴를 실행한다. 화면에 나타난 대화상자에서 'Rebuild Import'에서 체크를 해제한 다음 Dump 버튼을 눌러서 적당한 장소에 파일로 저장한다.

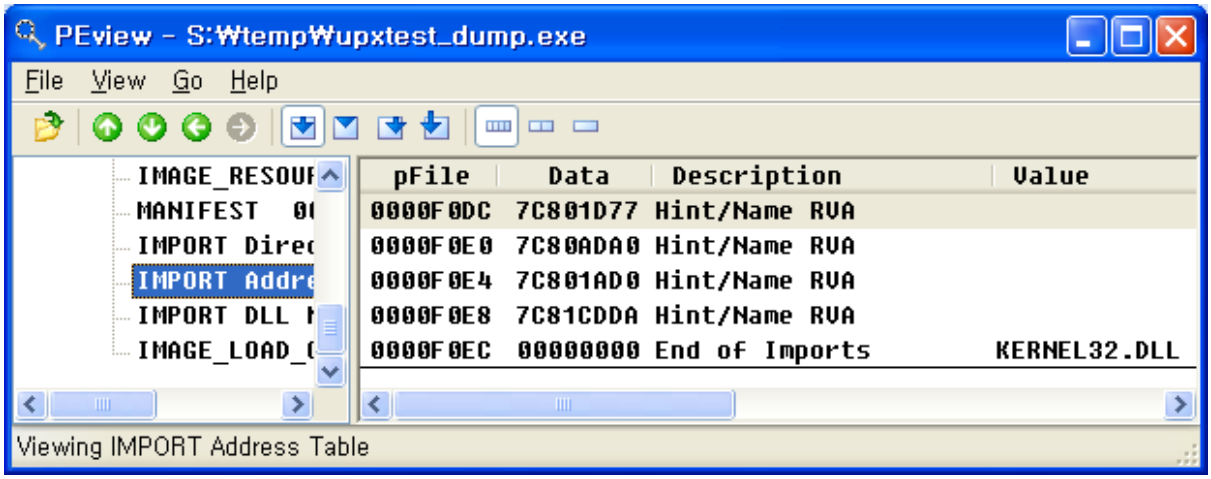
'Rebuild Import' 기능은 빌드된 임포트 테이블 정보를 다시 원래대로 돌려 놓는 기능이다. 로더는 실행파일을 불러올 때 파일에 기록된 정보를 토대로 IAT를 빌드한다. 메모리에 올라온 이미지의 IAT는 실제 함수 주소를 가리키고 있는 것이다. 따라서 메모리를 덤프해서 파일로 저장하고 다시 실행시키려고 한다면 로더가 IAT 정보를 찾을 수 없기 때문에 오류가 발생한다. 'Rebuild Import' 기능을 사용하지 않는 이유는 임포트 테이블을 새롭게 만드는 과정에서 메모리 구조와 덤프된 파일간의 구조가 틀리게 되고, 이를 처음 이 과정을 하는 사람은 결과를 이해하기가 쉽지 않기 때문이다. 우리는 덤프된 파일의 IAT를 손으로 직접 수정할 것이다.



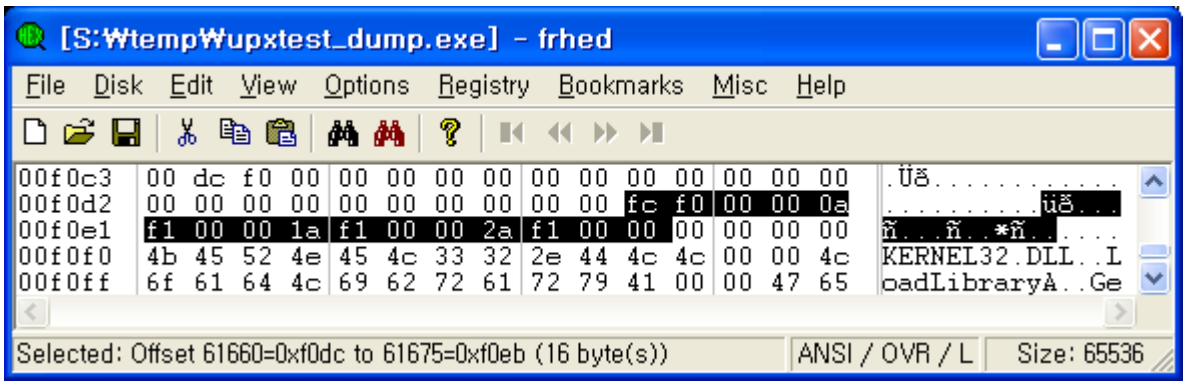
화면 3 원본 프로그램의 진입점

덤프된 파일을 그대로 실행해보자. 당연히 실행이 되지 않는다. 왜냐하면 앞서 설명했듯이 IAT가 빌드된 상태이기 때문이다. <화면 4>에 덤프한 파일의 IAT가 나와 있다. 0x7C801D77과 같이 실제 함수 주소가 기록되어 있는 것을 볼 수 있다. 이것을 로더가 알 수 있는 형태로 복구 시켜주어야 한다. 자동으로 복구시켜주는 툴을 사용할 수도 있지만 좀 더 잘 이해하기 위해서 손으로 직접 이 부분을 수정해보자. pFile 부분에 있는 오프셋을 찾아가 원본 파일에 있는 내용으로 고쳐주면 된다. <화면 5>와 <화면 6>에 수정하는 방법과 결과가 나와있다.

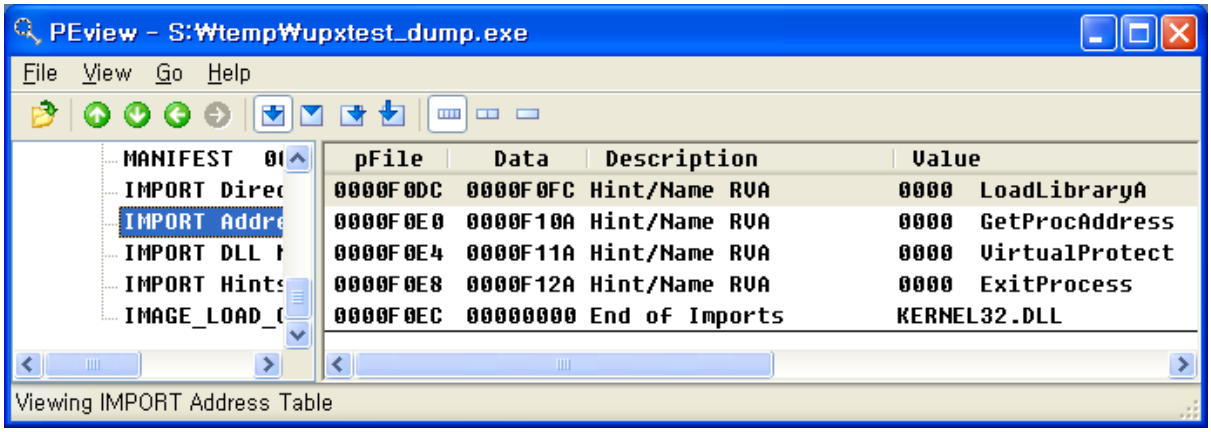
IAT를 수정하고 나면 정상적으로 실행이 된다. 물론 이것으로 완벽하게 언팩된 것은 아니다. 다른 컴퓨터 다른 운영체제로 옮겨서 실행해 보면 되지 않을 수도 있다. 왜 그럴까? 한번씩 그 이유를 고민해 보도록 하자.



화면 4 덤프한 파일의 IAT



화면 5 헥사 에디터로 IAT를 수정하는 화면



화면 6 덤프한 파일의 IAT를 복구해준 화면

실행 압축의 원리

이제 실행 압축이 어떻게 이루어지는 것인지 이론적인 지식을 조금 살펴보도록 하자. 가장 먼저 이해해야 하는 것은 실행 압축 전, 후의 섹션 구조다. <표 1>과 <표 2>에는 각각 압축 전, 후의

파일에 포함된 섹션이 나와 있다. 각 섹션의 크기와 메모리 위치 용도 등을 살펴보도록 하자.

표 1 원본 파일 섹션 구조

섹션명	RVA	Virtual Size	Raw Size	용도
.text	0x1000	0x62ab	0x7000	코드 저장
.rdata	0x8000	0x1b46	0x2000	읽기 전용 데이터 저장
.data	0xa000	0x187c	0x1000	읽기/쓰기 데이터 저장
.rsrc	0xc000	0x00b0	0x1000	리소스 저장

표 2 upx로 압축된 파일 섹션 구조

섹션명	RVA	Virtual Size	Raw Size	용도
.UPX0	0x1000	0x9000	0x0000	원본 코드 복원용
.UPX1	0xa000	0x5000	0x4800	압축해제 스텁 코드 및 압축된 원본 코드
.rsrc	0xf000	0x1000	0x0200	읽기/쓰기 데이터 저장

일단 간단하게 표에 나타난 메모리 크기(Virtual Size)를 비교하면 근소하게 upx로 압축한 파일이 크다는 것을 알 수 있다. 이는 메모리 상에 압축된 코드와 압축이 해제된 코드가 공존하는 구조이기 때문이다.

표에 나타난 내용을 토대로 각 이미지가 로딩된 경우의 메모리 구조를 그려보면 <그림 1>과 같이 된다. 왼쪽은 압축하기 전의, 오른쪽은 압축한 후의 메모리 구조다. upx 스텁 코드는 압축된 내용을 .UPX0 섹션에 푼 다음 원본 이미지의 시작 주소로 점프하는 일이 전부다.

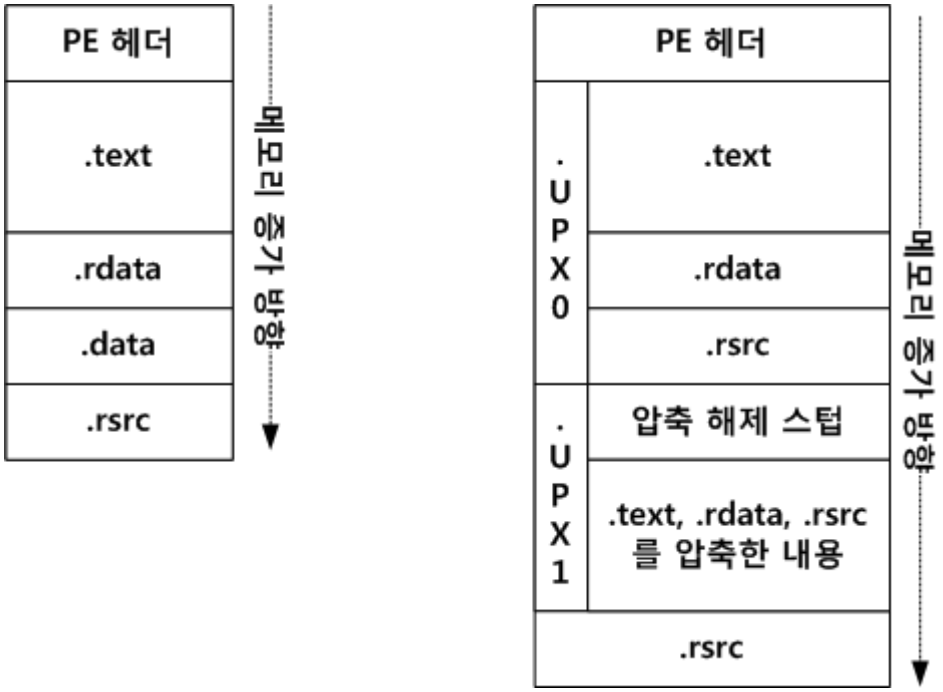
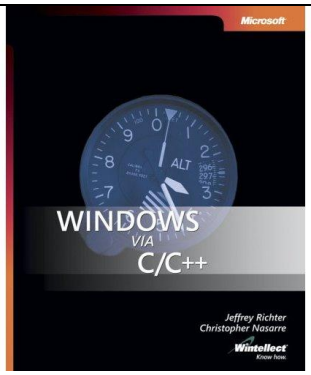


그림 1 원본 파일 과 upx로 압축된 파일의 메모리 구조

재미있는 사실은 .data 섹션이다. 표에 나타난 것처럼 .data 섹션은 전역 변수 등을 저장하기 위한 용도로 사용된다. 따라서 이는 파일일 때에는 크기가 0이다. 그런데 upx로 압축된 이미지를 살펴보면 .data 섹션이 없는 것을 볼 수 있다. 왜 그럴까? 이는 .data 섹션은 메모리 읽고 쓸 수 있는 메모리 공간만 확보해주면 되기 때문이다. upx로 압축된 이미지에서 .data 섹션은 .UPX1 섹션이 된다. 압축 해제를 한 후에는 스텁코드와 압축된 데이터가 더 이상 필요하지 않기 때문이 그 곳을 저장 공간으로 사용하는 것이다. 살펴보면 .data 섹션과 .UPX1 섹션의 시작 주소가 같다는 것을 알 수 있다. 물론 항상 모든 경우에 .UPX1 섹션이 .data 섹션이 되는 경우는 아니다. .data 섹션의 경우에는 이런 형태로 처리된다는 것만 알아두자.

스텁 코드가 하는 일은 간단하지만 저러한 메모리 구조를 가지도록 이미지를 재구성 하는 일은 간단하지만은 않다. 운영체제의 로더가 관여하는 내용은 그대로 유지해 주어야 하기 때문이다. 로더가 참조하는 로드 설정 테이블(Load Configuration Table), IAT, Windows XP 이후부터 공용 컨트롤을 사용하기 위한 매니페스트 리소르를 압축한다면 실행 파일의 동작이 정상적으로 진행되지 않을 것이기 때문이다. 따라서 이러한 로더가 관여하는 부분에 대해서는 압축을 피해가도록 만들어야 한다. 리소스 섹션의 경우는 구조가 복잡하기 때문에 여기서는 간단하게 로드 설정 테이블(Load Configuration Table)과 IAT에 대해서만 살펴보도록 하자.

박스 1 Windows via C/C++



Jeffrey Richter 아저씨가 몇 년 만에 C/C++개발자를 위한 Windows 프로그래밍 책을 새로 출간했다. 이름도 근사하게 Windows via C/C++로 바뀌었다. Windows 환경에서 시스템 프로그래밍을 하는 독자라면 반드시 읽어야 할 필독서다. 아직 Jeffrey Richter 아저씨의 책을 접해보지 않았다면 당장 지르도록 하자. 다소 부담스러운 가격이긴 하지만 무엇을 상상하든 그 이상의 가치를 보여줄 책임에 틀림없다.

로드 설정 테이블 처리

로드 설정 테이블은 로더가 이미지를 로딩할 때 참조하는 정보들을 기록해둔 테이블이다. OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG]을 통해서 정보를 찾을 수 있다. 파일에서 해당 위치를 찾아가면 IMAGE_LOAD_CONFIG_DIRECTORY 구조체가 저장되어 있다. 구조체의 원형이 <리스트 2>에 나와 있고, 각 필드 별 의미는 <표 3>에 나와 있다.

리스트 2 IMAGE_LOAD_CONFIG_DIRECTORY 원형

```
typedef struct {
    DWORD   Size;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   GlobalFlagsClear;
    DWORD   GlobalFlagsSet;
    DWORD   CriticalSectionDefaultTimeout;
    DWORD   DeCommitFreeBlockThreshold;
    DWORD   DeCommitTotalFreeThreshold;
    DWORD   LockPrefixTable;            // VA
    DWORD   MaximumAllocationSize;
    DWORD   VirtualMemoryThreshold;
    DWORD   ProcessHeapFlags;
```

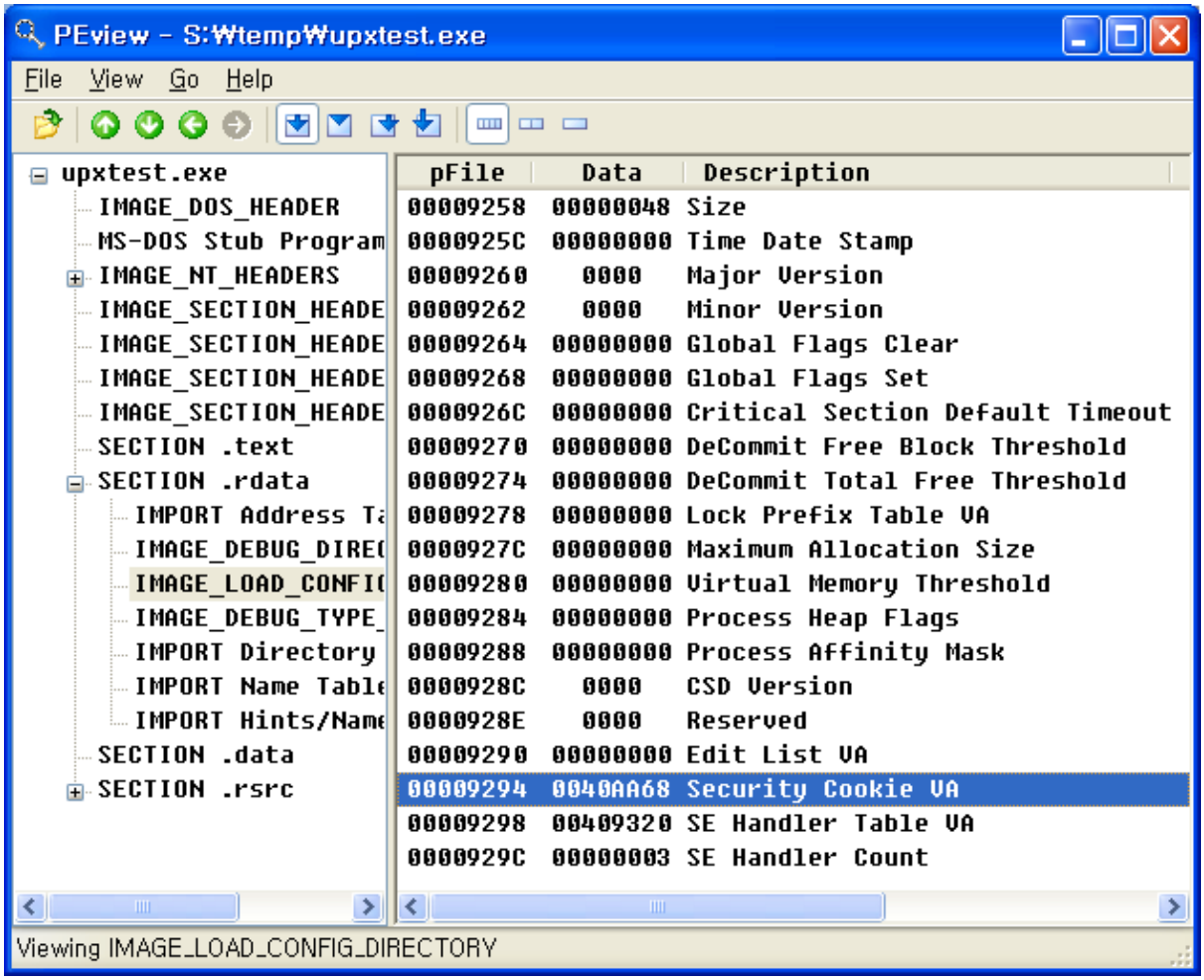
```
    DWORD     ProcessAffinityMask;
    WORD      CSDVersion;
    WORD      Reserved1;
    DWORD     EditList;                    // VA
    DWORD     SecurityCookie;              // VA
    DWORD     SEHandlerTable;              // VA
    DWORD     SEHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY32, *PIMAGE_LOAD_CONFIG_DIRECTORY32;
```

표 3 IMAGE_LOAD_CONFIG_DIRECTORY32 구조체의 필드별 의미

오프셋	사이즈	필드명	설명
0	4	Characteristics	파일 속성을 나타낸다. 현재는 사용하지 않음.
4	4	TimeDateStamp	1970년 1월 1일을 기준으로 지금까지 경과된 시간을 초단위로 저장한다.
8	2	MajorVersion	메이저 버전
10	2	MinorVersion	마이너 버전
12	4	GlobalFlagsClear	프로세스 시작 전 전역 로더 플래그를 초기화할지 결정한다.
16	4	GlobalFlagsSet	로더가 프로세스를 로딩할 때 설정할 전역 플래그 값
20	4	CriticalSectionDefaultTimeout	크리티컬 섹션 타임아웃 초기 값
24	8	DeCommitFreeBlockThreshold	시스템으로 메모리를 반환하기 위한 해제 임계 값 (바이트 단위)
32	8	DeCommitTotalFreeThreshold	해제된 메모리 총 량 (바이트 단위)
40	8	LockPrefixTable	[x86 전용] LOCK 접두어가 사용된 명령어의 가상 주소 리스트다. 이것들은 싱글 프로세스 시스템에서는 NOP으로 대체된다.
48	8	MaximumAllocationSize	최대 할당 크기 (바이트 단위)
56	8	VirtualMemoryThreshold	최대 가상 메모리 크기 (바이트 단위)
64	8	ProcessAffinityMask	0이 아닌 값으로 설정할 경우에 SetProcessAffinityMask를 프로세스 시작 시에 호출하는 것과 같음 (.exe 전용)

72	4	ProcessHeapFlags	프로세스 힙 플래그. HeapCreate의 첫 번째 인자와 같은 의미다. 이 값은 프로세스 기본 힙에 적용된다.
76	2	CSDVersion	서비스 팩 버전 ID
78	2	Reserved	예약된 값 (반드시 0)
80	8	EditList	시스템에서 사용하기 위해 예약된 값
60/88	4/8	SecurityCookie	Visual C++의 GS(스택 가드)를 구현하기 위해서 사용되는 쿠키 포인터.
64/96	4/8	SEHandlerTable	[x86 전용] 이미지에 포함된 구조적 예외처리 핸들러(SEH)의 RVA를 저장하고 있는 테이블의 가상 주소
68/104	4/8	SEHandlerCount	[x86 전용] 이미지에 포함된 구조적 예외처리 핸들러(SEH)의 개수

표에서 눈여겨 봐야 할 부분은 LockPrefixTable, SecurityCookie, SEHandlerTable이다. 왜냐하면 이 세 필드만이 가상 주소를 담고 있기 때문이다. 나머지는 대부분 단순한 정수 값들이다. 또한 이 필드들은 Visual C++ 컴파일러의 기능과 밀접한 관련이 있다. 이 두 필드의 의미를 upxtest.exe를 통해서 좀 더 자세히 살펴보도록 하자. upxtest.exe의 IMAGE_LOAD_CONFIG_DIRECTOYR가 <화면 7>에 나와있다.



화면 7 upxtest.exe의 IMAGE_LOAD_CONFIG_DIRECTORY

LockPrefixTable은 앞서 설명 했듯이 LOCK 접두어가 사용된 명령어의 가상 주소 목록을 나타낸다. 로더는 이미지를 로딩하는 시스템이 싱글 코어 시스템이라면 LOCK을 사용할 필요가 없기 때문에 성능 향상을 위해서 해당 부분을 NOP으로 변경할 수 있다. 우리가 만든 실행 파일은 LOCK 접두어를 사용하는 명령어가 없기 때문에 이 값이 0으로 지정되어 있다. 필자가 테스트해본 바로는 아직까지 이 필드는 사용되지 않는 것 같았다. LOCK 접두어를 사용하는 프로그램을 만들어도 해당 필드 값은 0이었기 때문이다.

SecurityCookie 필드는 스택 가드(/gs) 기능을 구현하기 위해서 사용된다. 스택 가드란 스택 오버플로를 감지하는 기능을 말한다. Visual Studio 2005의 프로젝트 옵션에서 컴파일러의 코드 생성 부분을 살펴보면 버퍼 보안 검사라는 항목이 있다. 이 곳에서 사용하는 것이 security cookie라는 것이다. security cookie의 원리를 간단히 설명하면 return 주소 다음에 security cookie를 스택에 집어넣어서 그 값이 변경되는지를 체크하는 것이다.

security cookie가 사용된 경우의 스택 구조가 <그림 2>에 나와있다. 지역 변수의 크기를 넘어서

서 리턴 주소까지 덮어쓰는 경우에는 security cookie값이 변경된다. 컴파일러는 함수 리턴 직전에 이 security cookie값을 검사하는 코드를 추가한다. security cookie값이 스택에 푸시할 때의 값과 다르다면 스택 오버플로가 발생한 것으로 생각하는 것이다.

그림 2 security cookie가 사용된 스택 구조

SEHandlerTable과 SEHandlerCount는 Safe SEH와 관련이 있는 필드다. 구조적 예외처리 핸들러는 예외처리 핸들러를 리스트 형태로 관리한다. 그 중에 가장 상단에 있는 예외처리 핸들러의 주소는 fs:0에 들어 있다. 따라서 fs:0에 핸들러를 기록하고 예외를 발생시키면 어떤 코드든지 실행할 수 있다는 이야기가 된다. 이는 다른 프로그램을 공격할 수 있는 좋은 경로가 되기 때문에 Windows XP 이후부터는 IMAGE_LOAD_CONFIG_DIRECTORY의 SEHandlerTable이 존재하는 경우에는 그곳에 등록된 예외처리 핸들러만 사용하도록 강제한다. 즉, 임의의 예외처리 핸들러를 fs:0에 기록하고 예외를 발생시키더라도 SEHandlerTable에 등록된 예외처리 핸들러가 아니라면 실행되지 않는다는 것을 의미한다.

백 번 말로 설명을 듣는 것 보다 한 번 실행해서 결과를 보는 것이 이해하기 쉽다. <리스트 3>에는 Safe SEH를 테스트할 수 있는 프로그램 소스 코드가 나와 있다. 아래 코드를 컴파일하고 실행하면 화면에 아무것도 출력이 되지 않는다. 등록한 예외처리 핸들러인 ExceptionHandler가 실행되지 않았기 때문이다. 링커 옵션에 /SAFESEH:NO를 추가한 다음 컴파일하고 실행하면 Exception과 Complete 메시지가 출력되는 것을 볼 수 있다. /SAFESEH:NO는 Safe SEH 기능을 오프 시키는 링커 옵션이다. 각각의 경우에 생성된 이미지를 보면 첫 번째 경우에는 IMAGE_LOAD_CONFIG_DIRECTORY에 SEHandlerTable이 존재하는 것을 볼 수 있지만 두 번째 이미지의 경우에는 그것이 없다는 것을 알 수 있다.

리스트 3 Safe SEH 테스트 프로그램

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

```
#include <stdio.h>

typedef struct _EXTENDED_EXCEPTION_REGISTRATION
{
	DWORD PrevHandler;
	DWORD ExceptionHandler;
	DWORD SafeExit;
	DWORD SafeEBP;
} EXTENDED_EXCEPTION_REGISTRATION
, *PEXTENDED_EXCEPTION_REGISTRATION;

extern "C" EXCEPTION_DISPOSITION __cdecl ExceptionHandler
(
	EXCEPTION_RECORD *rec
	, PEXTENDED_EXCEPTION_REGISTRATION frame
	, PCONTEXT ctx
	, void *
)
{
	printf("Exception₩n");

	ctx->Eax = rec->ExceptionCode;
	ctx->Eip = frame->SafeExit;
	ctx->Ebp = frame->SafeEBP;
	ctx->Esp = (DWORD)(DWORD_PTR) frame;
	return ExceptionContinueExecution;
}

int main()
{
	__asm
	{
		push ebp
		push offset Complete
		push ExceptionHandler
		push fs:[0]
		mov fs:[0], esp
```

```
    }

    DWORD *ptr = (DWORD *) 0x0;
    *ptr = 1;

Complete:
    printf("Complete₩n");

    __asm
    {
        pop fs:[0]
        add esp, 12
    }
}
```

우리가 생성한 upxtest.exe의 SEHandlerTable 필드 값은 0x409320이다. 이미지의 기본 로딩 주소가 0x400000이기 때문에 RVA로 계산하면 0x9320이 된다. 이 부분을 파일에서 찾은 것이 <화면 8>에 나와 있다. SEHandlerCount가 3이기 때문에 앞에서부터 세 개를 읽어 보면 각각 0x2680, 0x519c, 0x67bc가 된다. 이들 각각은 RVA다. 이들 주소를 IDA로 디스어셈블해서 살펴보면 실제로 구조적 예외처리 핸들러(SEH) 임을 알 수 있다. <화면 9>, <화면 10>, <화면 11>에 각각의 위치에 대한 코드가 나와 있다.

```
00009320 80 26 00 00 9C 51 00 00  BC 67 00 00 00 00 00 00
00009330 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

화면 8 upxtest.exe의 SEHandlerTable

```
.text:00402680 _except_handler4 proc near               ; DATA XREF: __SEH_prolog4↑o
.text:00402680                                          ; _IsNonwritableInCurrentImage+A↓o
.text:00402680
.text:00402680 var_11          = byte ptr -11h
.text:00402680 var_10          = dword ptr -10h
.text:00402680 var_C           = dword ptr -0Ch
.text:00402680 var_8           = dword ptr -8
.text:00402680 var_4           = dword ptr -4
.text:00402680 arg_0           = dword ptr  4
.text:00402680 arg_4           = dword ptr  8
.text:00402680 arg_8           = dword ptr  0Ch
.text:00402680
.text:00402680                 sub     esp, 14h
.text:00402683                 push    ebx
.text:00402684                 mov     ebx, [esp+18h+arg_4]
```

화면 9 upxtest.exe_except_handler4

```
.text:0040519C _unwind_handler4 proc near              ; DATA XREF: _local_unwind4+14↑o
.text:0040519C
.text:0040519C arg_0           = dword ptr  4
.text:0040519C arg_4           = dword ptr  8
.text:0040519C arg_C           = dword ptr  10h
.text:0040519C
.text:0040519C                 mov     ecx, [esp+arg_0]
.text:004051A0                 test    dword ptr [ecx+4], 6
.text:004051A7                 mov     eax, 1
```

화면 10 upxtest.exe의 _unwind_handler4

```
.text:004067BC __unwind_handler proc near              ; DATA XREF: __local_unwind2+B↓o
.text:004067BC                                         ; __abnormal_termination+9↓o
.text:004067BC
.text:004067BC arg_0           = dword ptr  4
.text:004067BC arg_4           = dword ptr  8
.text:004067BC arg_C           = dword ptr  10h
.text:004067BC arg_10          = dword ptr  14h
.text:004067BC
.text:004067BC                 mov     ecx, [esp+arg_0]
.text:004067C0                 test    dword ptr [ecx+4], 6
.text:004067C7                 mov     eax, 1
```

화면 11 upxtest.exe의 _unwind_handler

이제 실행 압축 파일에서 이 로드 설정 테이블을 어떻게 다루어야 할지 생각해 보도록 하자. 정수 값을 제외하고 가상 주소 값 중에 로더가 로딩 중에 사용할 가능성이 있는 필드는 LockPrefixTable이 전부다. 나머지 두 필드는 로딩된 이후에 정상적으로 실행되는지를 체크하기 위해서 사용되는 것이다. 따라서 LockPrefixTable이 사용되지 않은 이미지라면 로드 설정 테이블을 별도로 복사해서 로더가 찾을 수 있도록 만들어 주면 된다. 만약 LockPrefixTable이 사용된다면 해당 위치와 함께 리스트도 같이 복사를 해주어야 할 것이다.

IAT 처리

IAT는 로더와도 연관이 있고, .text 식션의 코드와도 연관이 있다. 로더는 로딩하는 과정에 IAT 테이블의 정보를 읽어서 실제로 필드를 함수 주소로 채우는 일을 하고, .text 섹션의 코드는 그렇게 채워진 함수 주소를 사용해서 API에 접근한다. 그렇기 때문에 IAT를 처리하는 것은 로드 설정 테이블을 처리하는 것보다 조금 복잡하다. IAT를 분리시킨 다음 이동 시키면 로더는 인식하겠지만 이동된 위치 때문에 .text 섹션의 코드가 잘못된 곳을 참조하는 문제가 생기게 된다. 그렇다고 IAT를 같이 압축시켜 버리면 로더가 IAT를 찾지 못하는 문제가 생긴다.

upx는 이러한 문제를 해결하기 위해서 원본 IAT를 압축 시키고 스텁 코드에서 압축된 IAT 정보를 읽어서 빌드하는 방법을 사용한다. <그림 3>에는 upx로 압축된 파일의 IAT 구조가 나와있다. IAT가 두 벌이 존재하는 것을 볼 수 있다. .rsrc 섹션에 포함된 IAT는 로더가 로딩하는 기본 IAT로 스텁 코드를 실행하는데 필요한 최소한의 함수를 담고 있다. 이는 LoadLibrary, GetProcAddress와 같은 함수로 원본 IAT를 복구하기 위해서 사용한다. 원본 IAT는 upx 스텁 코드에서 압축을 해제한 다음 빌드한다. IAT를 빌드하는 방법은 우리가 DLL 로더를 제작할 때 살펴보았던 방법과 동일하

다. IAT 정보를 열거해가면서 해당 라이브러리를 로딩하고 함수 주소를 구해서 채워 넣어주면 된다.

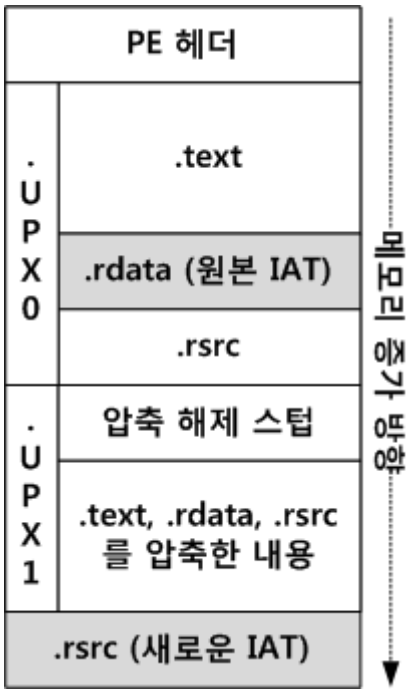


그림 3 upx로 압축된 파일의 IAT

앞서 언팩 시킬 때로 되돌아 가보자. 그 때 필자는 우리가 작업한 것이 완벽한 언팩이 아니라고 했다. 그 이유는 .rsrc 포함된 새로운 IAT만 복원 시켜 주었기 때문이다. .UPX0 섹션에 있는 원본 IAT의 경우는 그대로 빌드된 상태이다. 따라서 원본 IAT에 포함된 함수의 주소가 다른 컴퓨터로 이동한다면 정상적으로 실행되지 않는 문제가 발생한다. 이 문제는 upx가 IAT를 처리하는 방식에 기인한 것이기 때문에 매뉴얼 언팩된 파일에서 해결하기란 쉽지 않다.

도전 과제

예제 프로그램이 없는 것을 아쉬워하는 독자 분들이 많을 것 같다. 이번 연재의 주제인 실행 압축의 경우 실행 압축에 사용되는 코드보다 압축 알고리즘 자체에 사용되는 코드를 만드는 것이 더 힘들기 때문에 예제 프로그램을 만들지 않았다. 실행 파일 프로텍터 스텁 코드를 만들 때에 이번 시간에 배운 지식이 대거 활용될 것이기 때문에 다음 시간을 기대해 보아도 좋을 것 같다. 정히 실행 압축 프로그램의 실제 코드가 궁금한 독자 분들은 upx의 소스를 직접 분석해보는 것도 좋은 경험이 될 것이다.

이번 시간의 도전과제로는 upx를 언팩 할 때 사용했던 도구들을 직접 만들어 보는 것이다. 실행 파일의 덤프를 생성하는 프로그램이나 빌드된 IAT를 복구시켜주는 프로그램을 만들어 보도록 하자. 더불어 실행 파일의 덤프를 막는 방법은 없는지 한번씩 고민해보도록 하자.

참고자료

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

A Crash Course on the Depths of Win32™ Structured Exception Handling
http://www.microsoft.com/msj/0197/exception/exception.aspx