런-타임 압축기 패커 구현

Development of run-time compressor packer

개발기간	2014.01.03 - 2014.01.14
개발환경	Windows 7
개발도구	Visual Studio 2013
사용언어	C
현재버전	1.0

Contents

서론

이 문서를 작성하고 있는 시점인 현재, 우리나라는 시스템 및 역공학(reverse engineering) 관련 분야의 불모지이다. 인터넷에는 대부분의 유용한 정보들이 모두 해외에 있으며, 영어로 작성되어 있다. 설령 한국어로 작성된 자료가 있더라도 해당 주제의 개요나 툴들의 사용법만을 알려주는 글들이 있을 뿐 어떠한 주제를 상세하게 다루는 글은 찾아보기 힘들다. 이 문서는 런-타임 압축기인 패커(packer)의 기본적인 골격 구현에 대해 다루고 있다. 비록 이 프로그램은 현재까지도 유명한 UPX, ASPack, UPack, PESpin 그리고 Themida 등의 패커와 비교할 바는 못되며, 데이터 압축은 아직 구현에 성공하질 못했지만 역공학을 공부하거나 패커를 직접 구현하려고 시도하는 학생 또는 직장인들에게 시행착오를 줄이고 조금이나마 보탬이 되기를 바라며 이 문서를 작성했다.

가장 먼저 패커가 무엇인지에 대해 간단하게 소개를 하고, 설명을 위해 본인이 작성한 소스 코드를 기준으로 함수 별로 각 소단원을 나누었다. 코드를 위주로 설명을 진행했으나, 설명을 할 때는 코드 별로 줄줄이 설명하지 않았고 중요한 내용 및 짚고 넘어가야 할 부분만을 설명한다. 따라서 본 문서와 관련 없는 지식에 관한 설명은 최대한 배제하였다. 윈도우즈 운영체제의 역공학 및 PE 구조에 대해 어느 정도 공부한 사람을 대상으로 작성했으며, 글 만으로는 이해하기가 힘든 내용의 설명을 돕기 위해 일부 글 사이에는 그림을 첨부했다. 문서의 내용이 정말 유용하거나 방대한 지식을 담고 있지는 않지만 이런 정보가 필요한 이들에게 도움이 되길 바란다.

본론

패커란 ?

패커(packer)는 실행 압축을 위한 압축기이다. 압축에는 크게 일반 압축과 실행 압축 두 가지 방식이 존재한다. 일반 압축은 압축의 결과를 그대로 사용할 수는 없으며, 압축을 풀어주는 매개 프로그램을 반드시 필요로 한다. 압축이 해제되고 나서야 해당 프로그램을 정상적으로 사용할 수가 있는 것이다. 반면, 실행 압축은 결과물에 메모리 내에서 압축을 해제하는 루틴이 포함되므로 압축 해제 프로그램이 필요가 없으며 압축이 된 상태로 즉시 사용할 수 있다. 예를 들자면 .exe 파일을 실행 압축하게 되면 결과물은 .exe 파일로 생성되며, 압축된 이 결과물을 그대로 실행할 수 있다는 것이다.

실행 압축이 탄생한 배경은 MS-DOS 시절까지 거슬러 올라간다. 이 시절의 컴퓨터에서 사용되던 보조기억장치의 용량은 턱없이 부족했으며, 리소스를 조금이라도 절약하고자 실행 압축을 사용했

었다. 그러나 현대의 보조기억장치는 상당히 발전을 하였고 이에 따라 실행 압축은 큰 의미가 없어지게 되었다. 아래는 일반 압축과 실행 압축의 차이점을 비교한 표이다.

항목	일반 압축	실행 압축
대상 파일	모든 파일	PE 파일(.exe, .dll, .sys)
압축 결과 파일	압축(zip, tar 등) 파일	PE 파일(.exe, .dll, .sys)
압축 해제 방식	전용 압축 해제 프로그램	내부 디코딩(decoding) 루틴
실행 여부	자체 실행 불가	자체 실행 가능
장점	모든 파일에 대해 높은 압축율로 압축 가능	별도의 압축 해제 프로그램 없이 바로 실행 가능
단점	전용 압축 해제 프로그램이 없으면 압축 해제 불가	실행할 때마다 디코딩 루틴이 실행되므로 실행시간이 느려짐

표 1. 일반 압축과 실행 압축 비교

시대의 변화에 따라 실행 압축이 의도한 원래 의미는 사라지고 실행 파일 내부의 코드 및 데이터와 리소스들을 숨기기 위한 목적으로 변경이 되었다. 게임 개발 회사에서는 해커들로부터 게임이 역공학을 통해 크래킹 당하는 것을 방지하기 위해 실행 압축을 사용하게 되었으며, 악성 코드 제작자들은 파일의 불순한 의도를 숨기고 피해자의 컴퓨터가 바이러스를 평범한 실행 압축된 파일로 인식하게 하기 위해 사용한다.

이제 패커와 프로텍터를 조금 더 자세히 정의해보자. 패커는 두 가지 종류가 존재한다. 단순히 파일의 크기를 줄이기 위해 실행 압축을 하는 순수한 의도의 패커와 위에서 설명한 경우와 같이 악성 코드 제작자들이 백신으로부터 자신의 악성 코드를 보호하기 위해 파일 구조를 심하게 훼손시키는 불순한 의도의 패커가 존재한다. 전자의 경우에 포함되는 패커로는 UPX, ASPack 등이 있으며, 후자의 경우에 포함되는 패커로는 UPack, PESpin 등이 있다.

조금 특이한 유형의 패커도 존재한다. 압축 목적보다는 프로그램 내부의 코드, 데이터 및 리소스를 감추기 위해 프로그램 개발 회사에서 주로 사용하는 유형이다. 이런 과정에서 파일은 실행 압축 후에는 크기가 더 커지기도 하는데, 이런 부류의 실행 압축 유틸리티는 프로텍터(protector)라고 부른다. 대표적인 프로텍터로는 더미다(Themida)가 있다. 프로텍터는 역공학을 방해하기 위해

Anti-Debugging, Anti-Emulating, Code Obfuscation, Debugger Detection 등의 다양한 기능을 제공한다.

패커 개발을 위해서

패커를 개발하기 위해서는 어떤 것들이 필요하며, 결과물이 나오기 전까지는 어떤 과정으로 원본 프로그램이 처리되어야 할까? 가장 먼저 결정해야 할 것은 어느 운영체제 하에서 개발을 진행하는가이다. 리눅스가 될 수도 있고 윈도우즈가 될 수도 있다. 리눅스에서 개발하고 싶다면 ELF 구조를 공부해야 되며, 윈도우즈에서 개발하고 싶다면 PE 구조를 공부해야 한다. 본 문서에서는 PE 파일 패커에 대해 다루었으므로 개발 환경은 윈도우즈라고 가정한다. 그 다음은 해당 운영체제의 시스템 API를 익히는 것이다. 일반적으로 파일 구조는 플랫폼에 종속적이며 해당 플랫폼은 대체로 자신의 파일 구조에 대하여 미리 정의된 구조체나 API들을 제공한다. 구체적인 예로 리눅스는 ELF 구조를 위해 Elf32_Ehdr, Elf64_Ehdr 등의 구조체들을 제공하며 메모리의 권한을 다루기 위한 함수로 mprotect 함수가 있다. 윈도우즈는 PIMAGE_NT_HEADERS, IMAGE_SECTION_HEADER 등의 구조체들을 제공하며 mprotect와 거의 동일한 기능을 수행하는 VirtualProtect 함수가 있다.

사실, 시스템 API들을 굳이 외울 필요는 없으며 필요할 때마다 인터넷에서 정보를 찾아 참고하면 된다. 그러나 본격적으로 패커의 소스 코드를 작성하기 위해서는 압축을 하고자 하는 원본 프로그램이 어떤 식으로 처리되어야 하는지에 대한 큰 흐름은 미리 생각해야 한다. 제일 먼저 해야 할 작업은 당연히 파일을 메모리로 불러오는 것이며, 우리는 PE 파일을 다룰 것이므로 그 파일이 PE 파일이 맞는지 아닌지 여부를 제일 먼저 확인해야 한다. PE 파일이라면 패킹을 시작하고 아니라면 정상적인 PE 파일을 입력하라는 오류 메시지와 함께 프로그램을 종료하면 될 것이다.

실행 압축된 파일, 즉 패킹된 파일은 실행되면 메모리에서 자신 스스로 압축을 해제하는 디코딩 루틴을 제일 먼저 수행시킨다. 이 디코딩 루틴을 포함할 새로운 섹션이 필요한데 본인은 '.Min' 이라는 이름의 섹션을 새로 생성하여 이 곳에 디코딩 루틴을 담았다. 데이터 압축 해제를 제외하고 디코딩 루틴이 수행하는 일 중, 가장 중요한 일이 하나 더 남았는데 IAT(Import Address Table)를 복구하는 일이다. IAT가 필요한 이유는 여러 환경의 변화 때문인데 리눅스를 조금 알고 있다면 PLT(Procedure Linkage Table)와 GOT(Global Offset Table)을 생각하면 된다. MS-DOS 시절의 프로그램은 실행에 필요한 모든 요소들을 내부에 모두 가짐으로써 용량이 비교적 컸다. 그러나 윈도우즈의 프로그램은 DLL을 사용함으로써 이러한 단점을 보완하였고, 대부분의 프로그램은 DLL로부터 실행에 필요한 함수를 임포트(import)하여 사용한다. 윈도우즈의 서비스 팩 업데이트 또는 DLL 업

데이트 등의 여러 원인에 의해 DLL 내부에서도 함수들의 주소가 변경이 된다. 만약 동적으로 IAT를 구성하지 않고 파일 내부에 DLL의 함수들의 주소를 하드코딩하고 그대로 사용한다고 가정해 보자. 정말 끔찍한 일이 발생한다. 위에서 예로 든 변화가 일어날 때마다 프로그램을 다시 컴파일 해야 할 것이며, 그냥 둔다면 온전하게 실행되는 프로그램은 거의 없을 것이다. 윈도우즈에서는 어떤 프로그램을 실행하면 PE 로더가 해당 프로그램에서 임포트하는 DLL들로 채워져 있는 IDT(Import Directory Table)로부터 필요한 DLL을 조사하여 임포트되는 함수들을 하나씩 IAT에 채워주지만, 패킹된 프로그램은 내부의 디코딩 루틴이 실행되는 시점에 PE 로더는 이미 작업을 마친 상태이므로 직접 IAT를 채워줘야 한다. 정리해보면 디코딩 루틴을 담을 공간을 위해 하나의 새로운 섹션을 생성하고, 내부에 데이터 압축 해제 루틴과 IAT 복구 루틴을 직접 넣어줘야 한다. 다만, 서론에서 소개했듯이 데이터 압축은 구현하지 않았으므로 IAT 복구 루틴만을 넣어준다.

IAT 복구를 위해서는 IDT에 들어있는 DLL들의 이름, 각 DLL의 IAT가 위치하고 있는 주소는 어디이며 각 DLL에 포함되어 있는 함수 이름을 어딘가에 저장할 필요가 있다. 이 정보들을 보관할 공간이 필요하다. 그래서 본인은 하나의 섹션을 더 추가하고 이 섹션의 이름을 '.Dong' 으로 명명했다. .Min 섹션에 담겨있는 디코딩 루틴이 수행되며 .Dong 섹션에 백업해둔 정보들을 참조하여 IAT 복구가 진행된다. 여태까지 설명한 내용을 요약하여 앞으로 해야 할 일을 나열해보면 아래와 같다.

① 패킹할 파일 읽기
② 읽은 파일이 정상적인 PE 파일인지 확인
③ 두 개의 섹션 추가
④ IDT(Import Directory Table) 백업
⑤ 함수 이름 백업
⑥ 기존의 IDT를 지우고 새로운 IDT 작성
⑦ 디코딩 루틴 삽입

그림 1. 패커의 전체적인 흐름

본인은 그림 1의 각 항목 별로 함수를 정의했으며 앞으로 전개할 내용에서는 함수 단위로 설명을 할 것이다. 그러나 설명했던 내용들이 잘 이해가 되지 않는다면 PE 파일 구조에 대해 공부를 조

금 더 하고 계속 진행하는 것을 추천한다.

패킹할 파일 읽기

파일을 읽기 위해 LoadFileToPack 함수를 정의했다. 메모리 사상 함수인 CreateFileMapping 함수와 MapViewOfFile 함수를 사용하여 원본 파일을 읽기 전용으로 불러와 Mapped 포인터에 시작 주소를 저장한다. 그리고 패킹되어 파일로 생성될 결과물을 보관하는 공간을 위해 HeapAlloc 함수를 사용하여 메모리를 할당 후, BasePointer에 시작 주소를 저장한다. 최종적으로 이 메모리 공간에 파일의 모든 내용을 읽은 후 저장한다. 즉, 원본 파일을 참조해야 할 때는 Mapped 포인터를 사용하고 패킹을 위해 변경된 사항들은 BasePointer에 갱신되어, 이 메모리의 내용들이 마지막에는 패킹된 결과물 파일로 생성된다.

PE 파일 여부 확인

읽은 파일이 정확한 PE 파일인지 여부를 먼저 확인하는 방법은 내부의 시그니처(signature)를 확인하는 것이다. 보통 매직 넘버라고도 부르는 이것은 파일의 특정 오프셋에 반드시 존재하는 몇몇 기호를 의미한다. PE 파일에는 두 가지의 매직 넘버가 존재한다. 하나는 MS-DOS의 개발자였던 Mark Zbikowski의 머리 글자를 따서 만든 DOS 헤더에 존재하는 'MZ', 또 다른 하나는 NT 헤더에 존재하는 'PE' 헤더이다. 두 가지 모두 헤더의 가장 첫 자리에 위치하고 있으며, 없다면 프로그램이 정상 실행되지 않는다.

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
    WORD   e_magic;                 // Magic number
    …
    LONG   e_lfanew;                // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

그림 2. Windows에 실제로 정의되어 있는 DOS 헤더 구조체

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD                   Signature;
    IMAGE_FILE_HEADER       FileHeader;
    IMAGE_OPTIONAL_HEADER   OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

그림 3. Windows에 실제로 정의되어 있는 NT 헤더 구조체

위의 소스 코드는 DOS 헤더와 NT 헤더의 실제 구조체 모습이며, DOS 헤더의 경우는 중요하지 않은 원소를 표시하지 않았다. 볼드체로 표시되어 있는 e_magic과 Signature에 'MZ'와 'PE'가 각각 들어간다. 이 외에도 PE 파일이 정상적으로 실행되기 위한 File Alignment나 Section Alignment와 같은 PE 스펙(specification)들이 약간 더 존재하지만 그 경우들은 고려하지 않았다. 매직 넘버를 눈으로 직접 확인하고 싶다면 헥스 에디터로 PE 파일을 열어서 확인해보면 된다. 파일 구조의 제일 상단에 위치하고 있으므로 쉽게 확인할 수 있을 것이다.

새로운 두 개의 섹션 생성

백업할 정보들과 디코딩 루틴을 담기 위해 두 개의 섹션이 추가로 필요하다는 것을 위에서 설명하였다. 섹션들을 생성하기 이전에 PE 헤더의 정보들을 쉽게 참조하게 위하여 각 구조체 헤더들의 포인터를 선언하고 읽기 전용으로 맵핑한 원본 파일의 주소와 결과물을 담고 있는 메모리 주소를 적절하게 대입해주었다.

이제 섹션을 생성해보자. 섹션에도 DOS 헤더나 NT 헤더와 같이 여러 속성(attribute)들이 존재하며 하나의 값이라도 잘못 되었을 시, 프로그램이 정상 실행이 되지 않는다. 미리 전역으로 선언해 둔 .Dong과 .Min 섹션을 CreateSection 함수에서 먼저 0으로 초기화한다. 그리고 필요한 최소한의 속성들의 값을 넣어줘야 한다. Virtual Size, RVA, Size of Raw Data, Pointer to Raw Data 그리고 Characteristics이다. 이 문서를 읽는 대부분은 이들이 무엇인지 알고 있겠지만 각 속성들이 어떤 기능을 하는지 다시 간략하게 알아보자. 프로그램은 실행이 되지 않을 때는 보조기억장치에 데이터 형태로 존재하고 있다가 실행이 되면 프로세스 형태로 메모리 위에 올라온다. 이 다섯 속성은 그것과 연관되어 있다. Virtual Size는 프로그램이 보조기억장치로부터 로드 되어 메모리 상에 올라왔을 때 섹션 크기를 의미한다. 즉, 프로그램이 실행되었을 때 메모리 위에서의 해당 섹션 크기를 의미하는 것이다. RVA는 메모리 상에서 섹션의 시작 주소를 의미하며 Size of Raw Data는 보조기

억장치 에서의 섹션 크기를 의미한다. Pointer to Raw Data는 보조기억장치 내에서 섹션의 시작 오프셋을 가리킨다. 마지막으로 Characteristics는 섹션에 부여된 권한을 의미한다. 읽기 권한만 설정되어 있으면 이 섹션을 읽기만 가능하며, 읽기쓰기 모두 설정되어 있다면 이 섹션을 읽고 쓰는 것이 가능하다.

```
pSectionHeader += (pFileHeader->NumberOfSections - 1);
DongISH.VirtualAddress=pSectionHeader->VirtualAddress+pSectionHeader-
>Misc.VirtualSize;
```

그림 4. 새 섹션의 Virtual Address(RVA) 계산

Virtual Size 값은 본인이 직접 정의해둔 값으로 설정했다. 0x2000 바이트면 .Dong 섹션에서는 IDT 정보와 함수 이름을 담을 공간이 충분할 것이며 .Min 섹션에서는 디코딩 루틴을 담고도 남을 것이다. RVA 속성을 계산하는 것도 어렵지 않다. 위 코드는 .Dong 섹션의 RVA 속성을 계산하는 코드인데 바로 위에 위치하는 섹션의 RVA 값과 Virtual Size 값을 얻어와 더한 뒤 넣어주면 된다. 그런데 여기서 가장 중요한 문제가 발생한다. 알다시피 섹션의 위치는 Section Alignment의 배수여야 한다. 이것이 지켜지지 않으면 프로그램은 정상 실행이 되지 않는다. 따라서 현재 PE 파일에 명시되어 있는 Section Alignment 속성의 배수를 만들어 주기 위하여 RVA 속성 값을 조정해야 한다. Size of Raw Data와 Pointer To Raw Data는 File Alignment 속성의 배수가 되면 된다.

섹션의 수가 두 개 증가했기 때문에 NT 헤더 내부에 존재하는 FILE 헤더의 Number of Sections 속성의 값을 2 증가시키고 OPTIONAL 헤더의 Size of Image 속성도 0x4000(0x2000 * 2)만큼 더해줘야 한다. 새 섹션들의 설정 작업이 끝났으니 BasePointer 포인터를 통해 파일로 나올 결과물을 담고 있는 메모리에 이 섹션들의 헤더 값을 써줄 일만 남았다. 그러기 위해서는 기존에 존재하고 있던 섹션 헤더들의 밑에 여유 공간이 있는지 먼저 확인을 하고, 공간이 부족하다면 확보를 해야 한다.

```
* DOS 스텁(stub) 크기 = NT 헤더 시작 위치 - DOS 헤더 크기
* 헤더의 패딩(padding) 오프셋 = DOS 헤더 크기 + DOS 스텁 크기 + NT 헤더 크기 + 섹션 헤더 크기 * 섹션 헤더 개수 + Bound Import Table 크기
* 섹션 헤더가 위치할 오프셋 = 헤더의 패딩 오프셋 - Bound Import Table 크기
```

그림 5. 새로운 두 섹션이 위치하게 될 오프셋 계산

BIT(Bound Import Table)은 DLL의 로딩 속도를 조금 향상시켜 주지만 섹션에 반드시 포함되어야 하는 것은 아니다. 따라서 BIT 정보를 없애고 섹션 헤더를 위한 추가 공간을 확보하기로 한다. BIT에 대한 정보는 OPTIONAL 헤더 내에 Data Directory의 11번째에 존재한다. RVA 값이 0으로 되어 있다면 BIT는 존재하지 않으며 아니라면 BIT는 존재한다. BIT가 존재한다면, BIT의 값들을 모두 지우고 PE 헤더와 PE 바디 사이에 섹션 헤더 두 개가 들어갈 충분한 공간이 있는지 확인한다. 공간이 불충분 하다면 확보할 공간을 File Alignment의 배수에 맞춰서 마련한 뒤에, PE 바디를 밑으로 끌어내린다. 그리고 Data Directory의 BIT 정보를 모두 없애고 .Dong과 .Min 섹션을 마련된 공간에 넣어주면 된다.

IDT(Import Directory Table) 백업

IDT에는 해당 프로그램이 임포트하는 DLL의 이름과 IAT 및 INT(Import Name Table)의 위치 정보 등이 들어있다. 두 가지 다른 속성이 더 있지만 중요한 정보들이 아니므로 무시해도 좋다. 디코딩 루틴에서 IAT를 복구할 때 IDT에 들어있는 정보들을 사용하기 때문에 .Dong 섹션에 보관해야 한다. IDTBackUp 함수에서 이 일을 처리한다. 단순히 IDT의 시작 위치를 받아와서 모든 정보들을 .Dong 섹션으로 복사한다.

함수 이름 백업

각 DLL 들로부터 임포트해오는 함수 이름을 알아내려면 IDT의 정보를 분석할 줄 알아야 한다. 여기서 필요한 것은 Import Name Table 속성이다. 임포트된 DLL에서 사용되는 함수 이름을 가리키는 RVA들이 파일 어딘가에 테이블 형태로 구성되어 있고, Import Name Table 속성은 이 테이블의 시작 주소를 가리키고 있다. 따라서 IDT의 각 원소들을 조사하며 INT에서 함수 이름들을 하나씩 알아내어 저장하면 될 것이다. 정확한 이해를 위해 PE 뷰어를 통해 파일 내부를 살펴보자.

RVA	Data	Description	Value
00007604	00007990	Import Name Table RVA	
00007608	FFFFFFFF	Time Date Stamp	
0000760C	FFFFFFFF	Forwarder Chain	
00007610	00007AAC	Name RVA	comdlg32.dll
00007614	000012C4	Import Address Table RVA	

그림 6. notepad.exe의 IDT 중 comdlg32.dll의 부분

그림 6을 보면 Import Name Table RVA의 값은 7990으로 되어있다. 당연히 RAW가 아닌 RVA 값이며 7990를 추적해보자.

00007990	00007A7A	Hint/Name RVA	000F PageSetupDlgW
00007994	00007A5E	Hint/Name RVA	0006 FindTextW
00007998	00007A9E	Hint/Name RVA	0012 PrintDlgExW
0000799C	00007A50	Hint/Name RVA	0003 ChooseFontW
000079A0	00007A40	Hint/Name RVA	0008 GetFileTitleW
000079A4	00007A8A	Hint/Name RVA	000A GetOpenFileNameW
000079A8	00007A6A	Hint/Name RVA	0015 ReplaceTextW
000079AC	00007A14	Hint/Name RVA	0004 CommDlgExtendedError
000079B0	00007A2C	Hint/Name RVA	000C GetSaveFileNameW
000079B4	00000000	End of Imports	comdlg32.dll

그림 7. notepad.exe에서 comdlg32.dll 로부터 임포트한 함수 이름들

7990의 값은 7A7A, 7994의 값은 7A5E이다. 이런 식으로 79B4까지 4 바이트 값들이 들어있다. 이 값들은 함수의 이름을 가리키는 RVA 값이다. 가령, 7994에 들어있는 7A5E 값은 'FindTextW' 라는 함수 이름의 문자열을 가리키고 있으며 Image Base 속성에 7994를 더하면 실제로 'FindTextW' 문자열의 주소가 나오게 된다. 프로그램을 실행 시키고 디버거로 Image Base + 7994 주소를 추적해 보면 'FindTextW' 문자열을 볼 수 있을 것이다. 이제 함수 이름들을 어떻게 옮길지에 대한 감이 잡혔을 것이라 확신한다. 추가로 설명할 것이 있다면, IDT 내부의 각 원소들은 윈도우즈 에서는 PIMAGE_IMPORT_DESCRIPTOR 으로 정의되어 있으며 IDT의 끝은 널 구조체로 끝이 나며 INT의 끝 값도 널 값이다. 따라서 소스 코드에서는 읽은 구조체의 Name RVA 속성과 Import Address Table RVA 속성이 0이라면 루프를 종료하도록 했다. Name RVA 속성은 함수 이름들의 문자열 주소와 같이 DLL 이름의 문자열을 가리키는 RVA 값이며, Import Address Table RVA 속성은 IAT의 시

작 주소를 가리키는 값이다. 실제 구조체 내부에는 Name RVA가 Name으로 Import Address Table RVA가 FirstThunk로 명명되어 있다.

0000DF80	80001266	Ordinal	1266
0000DF84	800009D2	Ordinal	09D2
0000DF88	800017A4	Ordinal	17A4
0000DF8C	80000FEE	Ordinal	0FEE

그림 8. Ordinal로 임포트된 어느 DLL의 함수들

간혹 프로그램의 INT가 0인 경우가 있다. 이런 경우는 IAT에 함수의 이름들이 들어있는 경우이므로 IAT를 분석하면 된다. 그림 6과 7에서 보였듯이 단순히 RVA를 추적하여 함수 이름을 알아내고 저장하는 작업만 반복하면 된다. 그런데 다른 예외적인 경우로는 그림 8과 같이 함수를 Ordinal로 임포트 하는 경우이다. 이 의미는 DLL 내부에는 함수가 이름이 아닌 고유 번호로도 저장이 되어 있으며, 함수를 이름으로 임포트하는 것이 아닌 이 고유 번호로 임포트 하는 것이다. 함수 이름이 정상적인 이름으로 임포트 되었는지 또는 Ordinal로 임포트 되었는지 알 수 있는 방법은 RVA의 가장 앞 비트를 보는 것이다. 만약 INT 내에 RVA 값의 맨 앞 비트가 0x80이라면 이것은 함수를 Ordinal로 임포트 했다는 의미이다. 그러므로 이름과 Ordinal 중 어떤 방식으로 임포트 되었는지 반드시 먼저 구분해야 한다. 따라서 함수 이름을 추적하기 전에 가장 먼저 0x80000000과 RVA를 앤드 연산 후에 진행하도록 한다. 이제 Ordinal로 임포트 된 함수의 이름을 알아내는 과정을 살펴보자. 아래에 간략하게 정리해 놓았다.

① LoadLibrary 함수를 이용하여 모듈의 시작 주소를 얻음
② 모듈의 Data Directory의 0번째 항목으로부터 EXPORT Table의 RVA를 얻음
③ 모듈의 베이스 주소와 ②에서 얻은 값을 더하여 EDT를 찾아감
④ Ordinal Table RVA 값을 얻어내어 0번째 원소부터 탐색
⑤ 찾고자 하는 Ordinal에서 Base 값을 뺀 값과 일치하는 값을 가진 인덱스를 탐색
⑥ Name Pointer Table RVA 값을 얻어내어 ⑤에서 구한 인덱스 위치의 값을 얻음
⑦ 모듈의 베이스 주소와 ⑥에서 얻은 값을 더하여 함수 이름을 얻음

그림 9. Ordinal 값으로 함수 이름을 찾는 과정

EDT(Export Directory Table)는 DLL의 익스포트(export) 정보를 담고 있다. 그러므로 당연히 각 DLL 마다 하나만 존재할 수 있다. 그리고 그림 9에서의 모듈은 DLL을 의미한다. DLL에는 함수들의 Ordinal 값이 배열 형태로 저장되어 있는데, Ordinal Table RVA는 이 배열의 시작 주소를 가리키며 Ordinal과 마찬가지로 함수의 이름들도 배열 형태로 저장되어 있는데, Name Pointer Table RVA는 이 배열의 시작 주소를 가리킨다. EDT의 실제 구조체는 PIMAGE_EXPORT_DIRECTORY이며 위의 두 배열은 각각 AddressOfNameOrdinals와 AddressOfNames라는 이름의 멤버로 들어있다. Ordinal을 사용하여 함수 이름을 구하는 과정을 보면 알겠지만 두 배열은 상관관계가 있다. Ordinal로 함수들을 임포트하는 프로그램 중 가끔씩 패킹이 제대로 되지 않는 경우가 발생하는데 원인은 아직 파악하질 못했다.

```
[DLL 이름] + [함수 이름] + [함수 이름] + … + [함수 이름] + [0xDE, 0xAD]
[DLL 이름] + [함수 이름] + [함수 이름] + … + [함수 이름] + [0xDE, 0xAD]
[DLL 이름] + [함수 이름] + [함수 이름] + … + [함수 이름] + [0xDE, 0xAD]
                                                    …
[DLL 이름] + [함수 이름] + [함수 이름] + … + [함수 이름] + [0xDE, 0xAD]
[DLL 이름] + [함수 이름] + [함수 이름] + … + [함수 이름] + [0xDE, 0xAD]
```

그림 10. IAT 복구를 위한 .Dong 섹션 내 백업 데이터의 형식

.Dong 섹션의 백업 데이터는 위와 같이 구성된다. 끝 부분에 0xDE와 0xAD를 붙여준 것은 DLL 별 묶음을 표시하기 위한 것인데 자세한 내용은 디코딩 루틴을 설명할 때 함께 설명하겠다. 이 모든 과정이 끝나고 나면 기존의 INT 영역에 존재하고 있던 함수 이름들도 모두 지운다. 이해가 잘 되지 않는다면 FuncNameBackUp 함수에 구현되어 있는 코드를 참고하면 된다.

기존 IDT 제거 및 새 IDT 작성

원본 프로그램에 존재하던 IDT를 제거하는 이유는 두 가지가 있다. 첫 번째로 IAT를 디코딩 루틴을 통해 직접 복구할 것이기 때문에 기존 IDT가 더 이상 필요가 없기 때문이며, 두 번째로 어떤 DLL과 함수들을 사용하는지 감추기 위해서다. 그러나 디코딩 루틴에서 IAT를 복구하기 때문에 LoadLibrary 함수와 GetProcAddress 함수는 기본적으로 필요하다. 따라서 이 함수들을 포함하고 있는 KERNEL32.dll는 반드시 임포트해야 한다. 그 외의 DLL은 필요없다. KERNEL32.dll 을 임포트 하려면 몇 가지 준비 과정이 있다. IDT의 Name 속성을 채워주기 위해서 "KERNEL32.dll" 문자열을 가지고 있을 공간 그리고 Import Address Table RVA 속성을 위해서 함수 이름 "LoadLibrary"와 "GetProcaddress" 문자열을 가지고 있을 공간이 필요하다. 여태까지 위의 단원에서는 .Min 섹션에 디코딩 루틴만을 넣을 것이라 했지만 이 공간들도 .Min 섹션의 첫 부분에 할당한다. 이 작업들은 모두 RemoveAndWriteIDT 함수에서 처리한다.

```
00016000  4B 45 52 4E 45 4C 33 32  2E 64 6C 6C 00 21 60 01  KERNEL32.dll.!`.
00016010  00 30 60 01 00 41 60 01  00 52 60 01 00 00 00 00  .0`..A`..R`.....
00016020  00 00 00 4C 6F 61 64 4C  69 62 72 61 72 79 41 00  ...LoadLibraryA.
00016030  00 00 47 65 74 50 72 6F  63 41 64 64 72 65 73 73  ..GetProcAddress
00016040  00 00 00 56 69 72 74 75  61 6C 50 72 6F 74 65 63  ...VirtualProtec
00016050  74 00 00 00 45 78 69 74  50 72 6F 63 65 73 73 00  t...ExitProcess.
```

그림 11. Min 섹션의 상단에 IDT 구성을 위해 삽입된 데이터

문자열 정보들이 삽입되면 그림 11과 같이 구성된다. '함수 이름 백업' 절에서 설명했듯이 PE 로더가 알아서 IAT를 채워주기 때문에 INT를 0으로 지정하고 IAT에 INT를 구성해도 된다. INT의 원소들이 가리키는 각각의 데이터는 PIMAGE_IMPORT_BY_NAME 구조체이다. 이 구조체의 멤버는 Hint와 Name으로 구성되어 있으며 Hint는 현재 상황에서 의미가 없으므로 0x0000으로 값을 채우고 실제 함수 이름 문자열을 바로 뒤에 써준다. 빨간 박스로 표시된 부분이 IAT 영역이다. 물론 IAT 영역이긴 하지만, INT 구조체의 포인터 값들로 채워져 있다. 리틀 엔디안(little endian) 방식으

로 바이트가 저장되어 있지만 첫 번째 박스의 RVA를 계산해보면 00016021이다. 00016021에 있는 값은 '00 00 4C 6F 61 64 4C 69 62 72 61 72 79 41 00' 이며, 해석해보면 의미 없는 Hint값 0x0000과 "LoadLibraryA" 문자열이다. 4번째 박스까지 각각 힌트와 함수 이름 문자열로 구성되어 있다. 이 새로운 IDT를 메모리에 쓰고 Data Directory의 첫 번째에 위치한 IMPORT table 항목의 크기를 조절하고 나면 패킹된 프로그램은 KERNEL32.dll 만을 임포트하게 된다. IDT의 끝은 널 구조체로 끝난 다는 사실도 잊어서는 안된다. 위 패킹된 프로그램이 메모리에 올라와 디코딩 루틴이 실행된 뒤에는 저 IAT 영역이 어떻게 되어있는지 디버거로 살펴보자.

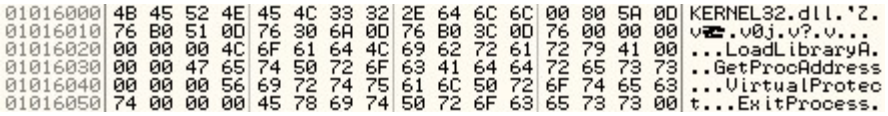

그림 12. 디코딩 루틴 수행 후에 채워진 .Min 섹션의 IAT 영역

원래는 구조체의 포인터 값들로 채워져 있었는데, 보다시피 760D5A80, 760D51B0 등의 실제 함수의 주소 값으로 채워져 있다. 각 주소는 LoadLibraryA, GetProcAddress, VirtualProtect, ExitProcess 함수와 대응된다. 실제로 함수의 주소가 맞는지 LoadLibraryA의 주소를 찾아가보자.

그림 13. 760D5A80에 위치한 LoadLibraryA 함수

그림 13을 통해 760D5A80 주소가 LoadLibraryA 함수의 주소인 것이 확인되었으며 PE 로더가 프로그램이 로드되며 IAT 영역을 실제 함수의 주소들로 채웠음을 확인했다.

디코딩 루틴 삽입

패커를 개발하는 과정에서 제일 재미있는 부분이다. 디코딩 루틴인 언패킹 코드를 만드는 과정이다. 패커는 메모리에서 압축이 해제된다. 따라서 고급 언어로 작성할 수는 없고 해당 프로그램이

실행되는 기반 아키텍처 언어로 프로그래밍 해야 한다. 본인은 인텔 아키텍처의 CPU를 사용하고 있으므로 인텔 어셈블리 언어를 사용해야 한다. 그러나 루틴을 작성할 때는 당연히 기계어로 삽입해야 한다. 따라서 어셈블리 언어로 먼저 코딩을 하고 기계어로 변환 후 그대로 패킹될 프로그램의 .Min 섹션에 삽입하면 된다. 기계어로 변환하기 위해서는 여러 가지 방법이 있을 수 있다. 아무 에디터에 어셈블리어를 작성하고 어셈블러를 이용하여 변환할 수도 있고, 더미 실행 파일을 디버거로 띄운 후 빈 공간에 어셈블리어를 작성할 수도 있다. 본인은 인텔 아키텍처의 매뉴얼을 직접 참고하여 테이블을 보면서 직접 기계어로 작성 후 그대로 삽입했다. 기계어 구문은 매뉴얼에 아주 상세히 나와있으며 인터넷에 관련 블로그가 있다면 참고해도 된다. 여기서는 루틴의 구성만을 살펴본다. 흐름 순서대로 코드를 조각조각 살펴보자.

```
; LoadLibraryA("KERNEL32.dll")
PUSH "KERNEL32.dll"
CALL DWORD PTR DS:[posLoadLibrary]

; VirtualProtect(VirtualAddressIDT, IDTSize, 0x04, GarbageMemory)
PUSH GarbageMemory
PUSH 0x04
PUSH IDTSize
PUSH VirtualAddressIDT
CALL DWORD PTR DS:[posVirtualProtect]
```

그림 14. KERNEL32.dll의 시작 주소를 얻고 IDT 영역에 쓰기권한 부여

실제 소스 코드를 보면 알겠지만 위의 어셈블리어 코드처럼 직관적으로 작성을 할 수는 없다. 다만 이해를 돕기 위해 그림 14와 같이 프로그램을 작성한 것이다. 지금 생각해보면 IDT 영역을 복구 하지 않아도 프로그램이 제대로 동작할 것 같지만, 소스 코드를 작성했던 시점에는 원래의 IDT 영역까지도 모두 복구시키는 루틴까지 작성을 했었다. 따라서 VirtualProtect 함수를 사용하여 IDT 영역에 읽기쓰기 권한을 부여한다. 참고로 LoadLibraryA("KERNEL32.dll")을 수행하는 상단의 어셈블리어 두 줄은 의미가 없다. 왜 저 코드를 작성했는지 기억은 나지 않지만 무시해도 좋다. 다만 분기가 일어날 때 바이트 단위로 계산되어 이동하므로 코드가 깨지는 것을 방지하기 위해 코드를 지워서는 안된다.

```
XOR ECX, ECX
MOV EBX, IDTSize
LABEL1 :MOV DL, BYTE PTR DS:[posDongIDT + ECX]
        MOV BYTE PTR DS:[posOrgIDT + ECX], DL
        INC ECX
        CMP ECX, EBX
        JNE LABEL1
```

그림 15. Dong 섹션에 백업해둔 원본 IDT를 모두 제자리에 복구

EBX에 IDT의 크기 값을 넣어주면 LABEL1의 루프를 IDT의 크기 회수만큼 반복한다. 로직은 아주 단순하다. 단지 Dong 섹션에 백업되어 있던 IDT의 데이터를 한 바이트씩 가져와 원래의 IDT 영역에 넣어준다.

```
; VirtualProtect(IDTSection, IDTSectionSize, 0x04, GarbageMemory)
PUSH GarbageMemory
PUSH 0x04
PUSH IDTSectionSize
PUSH IDTSection
CALL DWORD PTR DS:[posVirtualProtect]

MOV ESI, 1
MOV EBX, posIATRVA
MOV EDI, ImageBase
ADD EDI, DWORD PTR [EBX]

; LoadLibraryA(LIBRARY_NAME)
PUSH [posName]
CALL DWORD PTR DS:[posLoadLibrary]

XOR ECX, ECX
MOV EDX, EAX

LABEL2:CMP BYTE PTR DS:[posName + ECX], 0
       JNE LABEL5
       INC ECX
       CMP BYTE PTR DS:[posName + ECX], 0xDE
       JNE LABEL3
       CMP BYTE PTR DS:[posName + ESI], 0xAD
       JNE LABEL3
       ADD ECX, 2
       MOV ESI, posName
       ADD ESI, ECX

; LoadLibraryA(LIBRARY_NAME)
```

```
PUSH ESI
MOV ESI, ECX
CALL DWORD PTR DS:[posLoadLibraryA]

MOV ECX, ESI
ADD EBX, Temp
MOV EDI, ImageBase
ADD EDI, DWORD PTR [EBX]
MOV EDX, EAX
MOV ESI, 1
JMP Label2

LABEL3:CMP BYTE PTR DS:[posName + ECX], 0
       JNE LABEL4
       JMP OEP

LABEL4:MOV ESI, posName
       ADD ESI, ECX

PUSH EDX

; GetProcAddress(hMod, FUNCTION_NAME)
PUSH ESI
MOV ESI, ECX
PUSH EDX
CALL DWORD PTR DS:[posGetProcAddress]

MOV EDX, ECX
POP EDX
MOV ECX, ESI
MOV DWORD PTR DS:[EDI], EAX
ADD EDI, 4
MOV ESI, 1
JMP LABEL2

LABEL5:INC ECX
       JMP LABEL2
```

그림 16. 디코딩 루틴의 가장 핵심 부분인 IAT 복구 코드

IAT 영역은 코드 영역에 존재하며 읽기 전용으로 권한이 부여되어 있다. 따라서 VirtualProtect 함수를 사용하여 쓰기 권한을 부여해 주어야 한다. 현재 코드에는 IDTSection 이라고 명명되어 있지만 실제로는 IAT 영역의 시작 주소이다. 원래라면 각 DLL 마다 IAT 시작 주소를 따로 구해야 하는 것이 맞지만 일반적으로 DLL 별 IAT 영역은 쭉 이어져 있으므로 다른 예외는 무시했다. 이제 Dong 섹션에 저장해둔 그림 10 형식의 데이터를 참조한다. Dong 섹션에는 IDT가 원래 순서대로 저장되어 있고 그림 10의 데이터들도 IDT의 순서에 맞게 저장되어 있다. posName은 Dong 섹션에 저장된 그림 10의 데이터의 가장 첫 부분을 가리키고 posIATRVA는 백업한 각 IDT의 IAT 부분

을 가리킨다. 제일 먼저 posName을 사용하여 DLL 이름을 읽어 LoadLibraryA 함수를 통해 모듈의 주소를 얻어낸다. 그리고 이어지는 함수 이름을 읽은 후 GetProcAddress 함수로 해당 함수의 시작 주소를 알아낸 뒤에 posIATRVA의 위치에 하나씩 저장한다. 이렇게 함수 이름을 읽어 나가면 0xDE, 0xAD를 마주치게 된다. 0xDE, 0xAD는 단순히 DLL과 함수 덩어리를 구분하기 위한 구분자로 사용하기 위해 중간에 삽입했다. 다른 방법으로 각 DLL 및 함수 덩어리 앞에 함수의 개수가 몇 개인지 삽입하고 이 횟수만큼 루프를 돌도록 해도 무방하다. IAT 복구가 완료되고 나면 OEP(Original Entry Point)로 점프하도록 했다. 이제 원래의 프로그램이 실행 될 것이다.

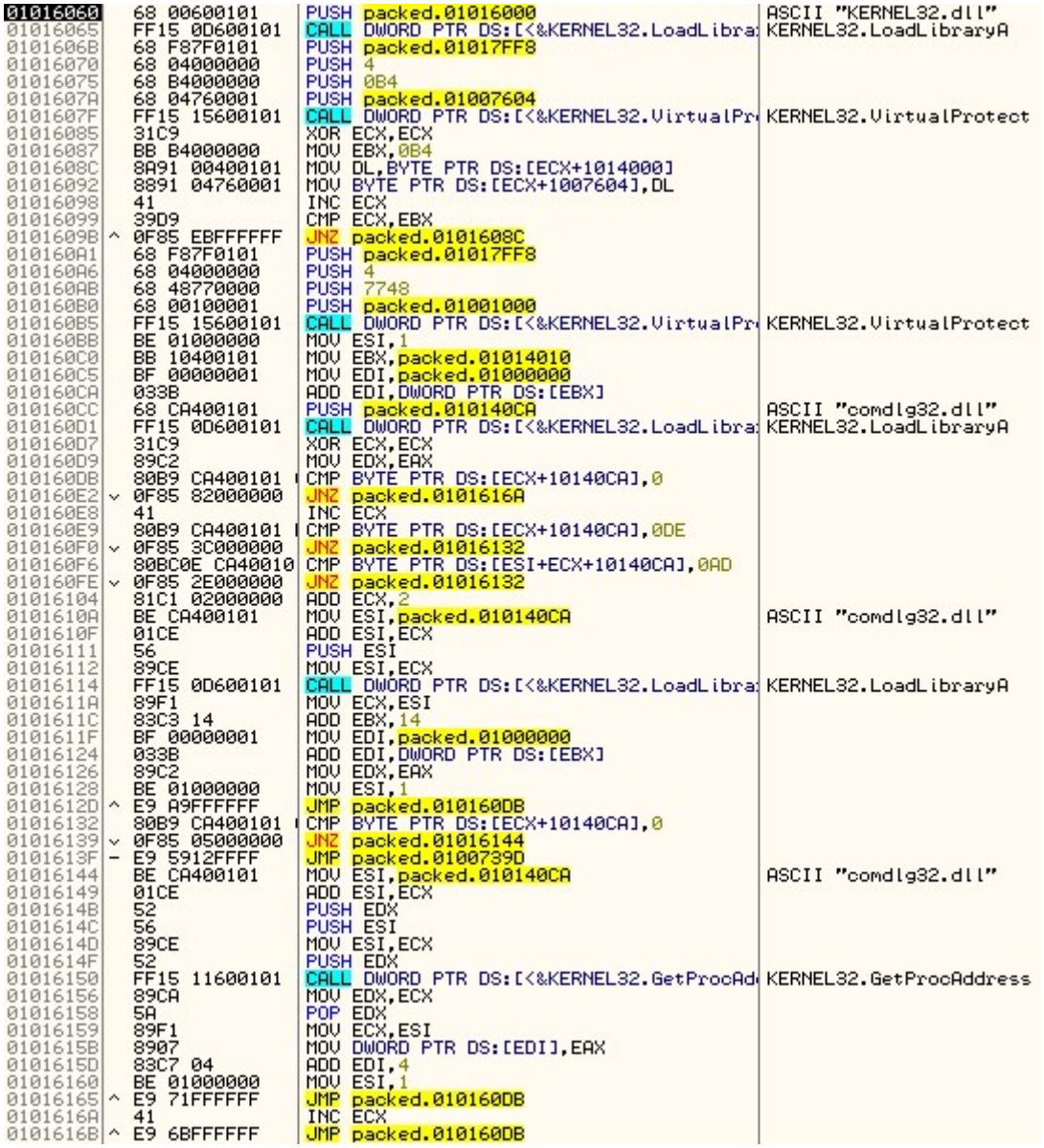

그림 17. 디버거를 통해 확인한 디코딩 루틴

패킹된 프로그램을 디버거에 띄운 후 본 코드 윈도우이다. 사용한 디버거는 올리디버거(ollydbg)이다. 프로그램은 메모리에 올라가 프로세스가 되었으며, 삽입한 기계어들이 잘 분석되어 어셈블리 코드로 나와있는 모습을 볼 수 있다. 이 절에서 설명한 모든 내용은 InsertUnpackCode 함수에서 처리한다.

결론

모든 .exe 프로그램을 테스트 해볼 수는 없었지만 간단한 메모 프로그램인 notepad.exe나 터미널 접속 프로그램인 putty.exe 등의 패킹은 정상적으로 이루어졌다. 문서 작성을 하며 본인이 작성한 코드들을 다시 살펴보니 굳이 할 필요 없는 일들도 보이며 이전에 해둔 작업을 내버려 두고 새롭게 진행하는 코드들도 꽤 보인다. 그리고 일부 PE 속성 값은 원본 프로그램을 분석하며 동적으로 설정을 해야 한다. 아무래도 본인이 패커의 동작을 실제로 구현해보는 것에 그쳤기 때문에 코드들이 다소 난잡하고 비효율적이며 여러 버그들이 존재할 수 있지만, 개발하며 겪은 여러 어려운 과정과 그 과정에서 본인도 몰랐던 지식을 습득했던 과정 및 소스 코드 작성에 의의를 둔다. 혹시 본인이 작성한 내용 중 잘못된 부분이 있다면 vrillon99@gmail.com으로 메일을 보내주면 정말 감사하겠다.

참고문헌

[1] 이승원,『 리버싱 핵심원리 』, 인사이트, 2012