윈도우 프로그래머를 위한 PE 포맷 가이드

DLL 로딩하기

목차

목차 1
License 1
소개 1
연재 가이드 1
연재 순서 2
필자소개 2
필자 메모 2
Introduction 2
재배치 섹션 3
DLL 로더의 전체적인 구조 5
파일을 메모리 속으로 7
재배치 수행하기 9
DLL에서 임포트한 함수 연결하기 10
페이지 속성 변경하기 12
LoadLibrary/FreeLibrary 구현 하기 13
GetProcAddress 구현 하기 14
CDllLoader 사용하기 15
도전 과제 16
참고자료 16

License

소개

PE 포맷에는 로딩 주소가 바뀐 경우에 고쳐주어야 하는 정보에 대한 목록을 재배치 섹션에 저장해 두고 있다. 이번 시간에는 PE 포맷의 재배치 섹션의 구조에 대해서 배우고 이 정보를 바탕으로 커스텀 DLL 로더를 제작해 본다.

연재 가이드

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

DLL 로딩하기

응용분야: 보안 프로그램

연재 순서

2007. 08. 실행파일 속으로
2007. 09. DLL 로딩하기
2007. 10. 실행 파일 생성기의 원리

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.

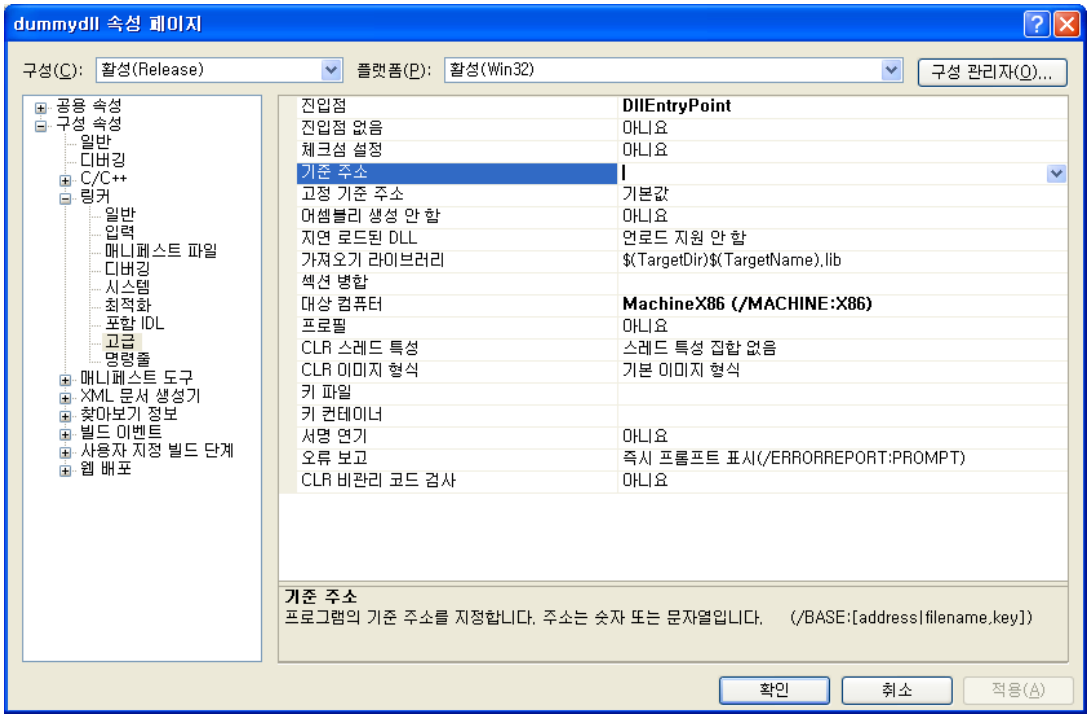
필자 메모

DLL을 직접 로딩해서 얻을 수 있는 가장 큰 장점은 보안성이다. 최근의 악성 소프트웨어의 경우 다양한 루트킷 기술을 사용한다. 대부분이 후킹 범주에 속하는 기술들이다. 특정 DLL의 IAT, EAT 등을 변조하고 코드 섹션을 바꾸기도 하고, 커널 함수 테이블이나 커널 코드를 변조하는 일도 심심찮게 일어난다. 당연히 LoadLibrary란 시스템 콜을 사용한 호출도 우리가 원하지 않는 곳으로 제어권을 넘겨 주기도 한다. 또한 그것이 악성 코드일 수도 있다. 커스텀 DLL 로더를 사용한다면 그러한 후킹을 우회해서 DLL을 사용할 수 있다. 또한 메모리상의 각종 DLL 들이 패치된(후킹된) 상태인지 점검하는 용도로 사용할 수도 있다.

Introduction

최근에 작업한 DLL 프로젝트가 있다면 프로젝트의 속성 창에 들어가 보도록 하자. 그곳에서 링커에서 고급 탭을 선택한다. <화면 1>과 같은 다이알로그가 나올 것이다. <화면 1>은 지난 시간에 만들었던 dummydll의 프로젝트 속성 창이다. 기준 주소를 살펴보자. 그곳이 비어있다면 해당 DLL의 로딩 번지는 0x10000000이 된다.

한 프로그램에서 사용하는 A, B, C, D란 네 개의 DLL이 있다고 생각해 보자. 네 개의 기준 주소가 모두 기본 값으로 지정되었다면 A, B, C, D 모두 0x10000000 번지에 로딩되려고 할 것이다. 하지만 프로세스의 주소 공간에 0x10000000 번지는 하나밖에 없다. 결국 나머지 세 개의 DLL은 다른 주소에 로딩될 수 밖에 없다. 로딩 주소가 바뀌게 되면 절대 주소를 사용한 것은 반드시 바뀐 주소에 해당하는 값으로 고쳐 주어야 한다. 그렇지 않으면 0x10000000 번지에 로딩된 다른 DLL의 메모리 영역을 참조하게 된다. PE 포맷에서는 이렇게 고쳐 주어야 하는 곳을 재배치 섹션에 모아서 저장해 두고 있다.



화면 1 DLL의 기본 로드 주소 변경 창

PE 포맷에 재배치 내용이 포함되어 있고, 재배치 과정은 전적으로 운영체제가 알아서 처리해주기 때문에 기준 주소를 정하는 작업이 무의미하게 느껴질 수 있다. 하지만 한 가지 더 생각해야 하는 것은 누가 하든 재배치 작업을 하는 데는 CPU 시간이 소모되고, 그러한 것들이 모이면 무시 못할 시간이 된다는 점이다. 이 시간은 고스란히 프로그램의 로딩 속도를 지연시키는 요인이 된다. 따라서 한 프로젝트를 위해서 만들어진 DLL이라면 기준 주소를 적절하게 분배해서 재배치 작업이 발생하지 않도록 만들어 주는 것이 좋은 습관이다.

자신이 만든 DLL이 아니라서 기준 주소를 변경할 수 없는 경우라면 Visual C++ 유틸리티에 포함되어 있는 rebase란 도구를 사용하면 쉽게 기준 주소를 변경할 수 있다. dummydll.dll의 기준 주소를 0x30000000으로 바꾸고 싶다면 아래와 같이 명령 프롬프트에서 입력하면 된다.

```
rebase –d 0x30000000 dummydll.dll
```

재배치 섹션

재배치 섹션을 분석하기 전에 가장 먼저 해야 할 일은 PE 포맷에서 재배치 섹션을 찾는 것이다. 재배치 섹션의 정보는 NT 헤더의 OptionalHeader 구조체의 DataDirectory에 저장되어 있다. DataDirectory[IMAGE_BASE_RELOCATION] 부분이 재배치 섹션의 정보를 담고 있는 곳이다.

재배치 섹션은 여러 개의 재배치 블록을 가지고 있다. <그림 1>에 재배치 블록의 구조가 나와 있다. 각 재배치 블록은 4바이트의 기준 주소와 4바이트의 블록 크기 그리고 N개의 재배치 정보를 가지고 있다. 각 재배치 정보는 2바이트로 구성되며 상위 4비트는 재배치 타입을, 하위 12비트는 재배치 수행 번지에 대한 오프셋을 저장하고 있다.

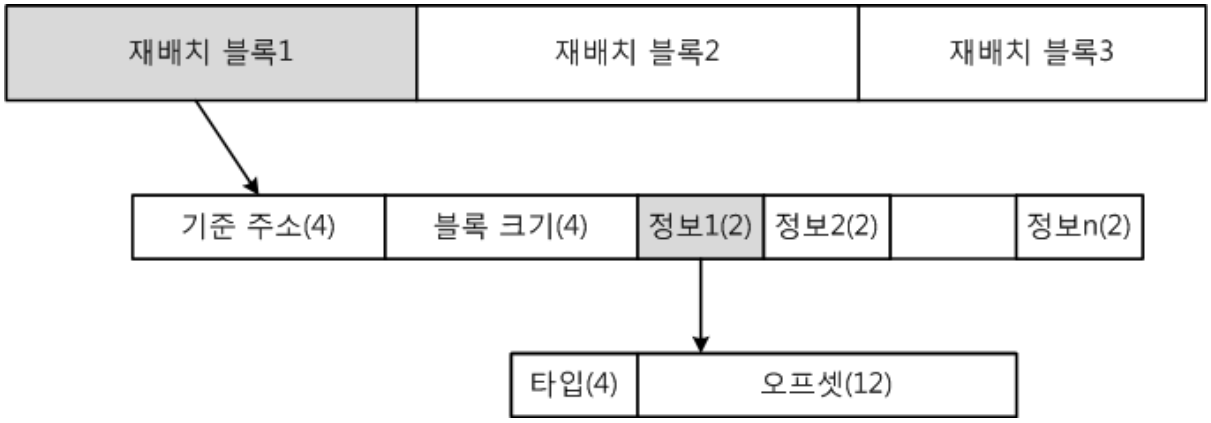


그림 1 재배치 정보

재배치 블록에 대해서 좀 더 살펴보도록 하자. 재배치 블록에 포함된 재배치 정보의 개수는 "(블록 크기 – 8) / 2"를 해서 구하면 된다. 재배치를 수행해야 할 대상 주소는 "기준 주소 + 오프셋"이 된다. 이 값은 RVA이기 때문에 프로그램을 로딩한 베이스 주소를 더해준 값이 실제 재배치 주소가 된다. 재배치 정보에 포함된 타입은 사실상 큰 의미가 없다. 32비트 PE 포맷의 경우는 IMAGE_REL_BASED_HIGHLOW가 사용되며, 64비트 PE 포맷의 경우에는 IMAGE_REL_BASED_DIR64가 사용된다.

재배치 블록의 내용을 모두 파악했다면 재배치를 수행하는 방법은 간단하다. 프로그램의 기준 주소(NT헤더의 OptionalHeader.ImageBase)와 로드한 실제 주소의 차이를 delta라고 할 때에, 각 정보에 해당하는 실제 재배치 주소를 구해서 그 곳에 delta 만큼 더해주면 된다.

pFile	Raw Data	Value
00000800	00 10 00 00 0C 00 00 00 25 30 45 30 00 20 00 00	.....%0E0. ..
00000810	0C 00 00 00 0C 30 10 30 00 00 00 00 00 00 00 00	.....0.0.....
00000820	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000830	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000840	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

화면 2 dummydll.dll의 재배치 섹션

<화면 2>에는 지난 시간에 우리가 만들었던 dummydll.dll의 재배치 섹션이 나와있다. 정보를 간단히 분석해 보도록 하자. 제일 처음에 있는 4바이트 00001000이 기준 주소가 된다. 다음에 오는 0000000C는 재배치 블록의 크기를 나타낸다. 12바이트 이기 때문에 실제로 이 곳에 포함된 재배치 정보는 2개라는 것을 알 수 있다. 다음에 오는 3025와 3045가 이 재배치 블록에 포함된 정보가 된다. 앞에 있는 3은 재배치 타입으로 IMAGE_REL_BASED_HIGHLOW를 나타낸다. 실제 재배치를 수행해야 할 두 군데의 주소는 00001025와 00001045가 된다. 이어지는 내용은 다음 블록에 관한 정보다. 다음 블록의 기준 주소는 00002000이고 크기는 0000000C가 된다. 재배치 정보는 각각 300C와 3010이 된다.

박스 1 엔디안 이야기

엔디안이란 바이트를 저장하는 순서를 말한다. 크게 리틀 엔디안와 빅 엔디안이라는 두 종류의 방식이 있다. 리틀 엔디안은 최하위 바이트를 메모리 상의 가장 낮은 번지에 저장하는 방식이고, 빅 엔디안은 최상위 바이트를 메모리 상의 가장 낮은 번지에 저장하는 방식이다. 0x01020304라는 데이터를 저장한다고 했을 때 리틀 엔디안은 0x04, 0x03, 0x02, 0x01 순서로 저장하고, 빅 엔디안은 0x01, 0x02, 0x03, 0x04 순으로 저장한다. 우리가 사용하는 x86 계열의 Intel CPU는 리틀 엔디안 방식을 사용한다.

<화면 2>의 데이터를 다시 한번 보자. 처음 4바이트가 0x00, 0x10, 0x00, 0x00이다. Intel CPU는 리틀 엔디안 방식을 사용한다고 했기 때문에 화면상에 표시된 내용을 거꾸로 읽어 들여야 한다. 그렇게 읽어 들이면 실제 데이터는 0x00001000이 된다. 만약 빅 엔디안이라면 이 값은 0x00100000이 될 것이다.

DLL 로더의 전체적인 구조

이번 시간에 제작해 볼 DLL 로더 클래스의 전체적인 구조가 <리스트 1>에 나와있다. 로더 클래스의 전체 구조를 살펴 보도록 하자. 로더 클래스에서 가장 핵심적인 내용은 DLL 정보를 저장하는 CDllInfo 클래스와 로드한 DLL의 목록을 저장하고 있는 m_loadDlls다.

CDllInfo 클래스에는 로드한 DLL의 경로, 해당 DLL에서 사용하고 있는 다른 DLL 목록, 로드 횟수, 그리고 DLL을 로드하는데 사용된 메모리 공간에 대한 포인터가 저장된다. 로드가 성공적으로 이루어지면 로드 과정에서 생성된 CDllInfo가 m_loadDlls에 추가된다.

리스트 1 CDllLoader 클래스

```
class CDllLoader
{
private:
  typedef BOOL (CALLBACK *DllMainFunc)(HINSTANCE, DWORD, LPVOID);
  typedef std::basic_string<TCHAR> tstring;

  class CDllInfo
  {
  public:
    tstring path; // dll 경로
    list<HMODULE> modules; // 사용한 DLL 목록
    DWORD ref; // 로드 횟수

    CVirtualMemory baseMem; // 베이스 주소
    CVirtualMemory headerMem; // 헤더 저장 공간
    CAutoArray<CVirtualMemory> *sectionMem; // 섹션 내용 저장 공간

  public:
    PIMAGE_NT_HEADERS NTHeader()
    {
      PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER) baseMem.Get();
      PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS) GetPtr(dos, dos->e_lfanew);
      return nt;
    }

    PIMAGE_SECTION_HEADER SectionHeader()
    {
      PIMAGE_NT_HEADERS nt = NTHeader();
```

```
    return (PIMAGE_SECTION_HEADER) GetPtr(nt, sizeof(*nt));
  }

  IMAGE_DATA_DIRECTORY &DataDirectory(DWORD type)
  {
    PIMAGE_NT_HEADERS nt = NTHeader();
    return nt->OptionalHeader.DataDirectory[type];
  }

  PVOID Base()
  {
    return (PVOID) baseMem;
  }

  HINSTANCE Instance()
  {
    return (HINSTANCE) Base();
  }

  BOOL DllMain(DWORD reason)
  {
    PIMAGE_NT_HEADERS nt = NTHeader();
    DllMainFunc func = (DllMainFunc) GetPtr(Base(), nt->OptionalHeader.AddressOfEntryPoint);
    return func(Instance(), reason, 0);
  }
};

typedef boost::shared_ptr<CDllInfo> DllInfoPtr;
typedef map<PVOID, DllInfoPtr> Base2DllInfoMap;
typedef map<PVOID, DllInfoPtr>::iterator Base2DllInfoMit;

Base2DllInfoMap m_loadDlls; // 로드한 dll 목록
CRITICAL_SECTION m_csLoadDlls;

private:
bool LoadFile(CDllInfo * dll); // dll 파일 로드
void UnloadFile(CDllInfo * dll); // dll 파일 메모리 해제
bool Reloc(CDllInfo *dll); // 재배치
DWORD IsBound(CDllInfo * dll); // 바운드된 dll인지 검사
bool BuildIAT(CDllInfo * dll); // IAT 작성
bool ProtectDll(CDllInfo * dll); // 섹션 보안 속성 조절
DWORD GetModuleTimeStamp(PVOID base); // 특정 모듈의 타임 스탬프 값 리턴


// base 주소에 해당하는 정보 구조체 리턴
CDllInfo * GetDll(PVOID base)
{
  Base2DllInfoMit it = m_loadDlls.find(base);
  if(it == m_loadDlls.end())
    return NULL;
  return it->second.get();
}

// 경로에 해당하는 정보 구조체 리턴
CDllInfo * GetDll(LPCTSTR path)
{
  Base2DllInfoMit it = m_loadDlls.begin();
  Base2DllInfoMit end = m_loadDlls.end();
  for( ; it != end; ++it)
  {
    if(_tcscmp(it->second->path.c_str(), path) == 0)
      return it->second.get();
  }

  return NULL;
}

void FreeDll(CDllInfo * dll);
void FreeMapItem(Base2DllInfoMap::value_type item) { FreeDll(item.second.get()); }
```

```
  CDllLoader(const CDllLoader &);
  CDllLoader &operator=(const CDllLoader &);

public:
  CDllLoader();
  ~CDllLoader();

  PVOID LoadLibrary(LPCTSTR path);
  bool FreeLibrary(PVOID base);
  FARPROC GetProcAddress(PVOID base, LPCSTR name);
};
```

파일을 메모리 속으로

이제 DLL을 로딩하기 위한 기본 지식은 모두 갖추었다. 이제 실제적으로 DLL을 로딩해 보도록 하자. 그 첫 번째 단계는 파일의 내용을 메모리에 적절하게 로딩하는 것이다. 지난 시간에 설명했듯이 섹션 데이터의 파일 오프셋과 메모리에 로딩되는 주소는 일치하지 않기 때문에 주의해야 한다.

<리스트 2>에는 PE 이미지를 메모리에 로드하고 그것을 다시 해제하는 함수가 나와있다. LoadFile은 파일을 메모리로 올리는 일을 하고, UnloadFile은 로드된 파일 데이터를 메모리에서 제거하는 일을 한다. 지난 시간에 설명한 PE 포맷의 기본 구조만 정확하게 이해하고 있다면 소스는 쉽게 분석할 수 있다. 중요한 필드에 대해서만 다시 살펴보도록 하자.

NT 헤더의 OptionalHeader 구조체에 있는 ImageBase는 이미지의 기본 로딩 주소를 담고 있다. 해당 주소에 로드된다면 재배치 과정을 수행할 필요가 없다. 같은 구조체에 있는 SizeOfImage는 전체 이미지를 메모리에 로딩하기 위해서 확보해야 하는 공간을 담고 있다. SizeOfHeaders는 헤더를 저장하기 위해서 확보해야 하는 공간을 의미한다. FileHeader의 NumberOfSections는 PE 파일에 담겨있는 전체 섹션의 개수를 가지고 있다. 각 섹션 헤더 구조체의 PointerToRawData와 SizeOfRawData는 각각 섹션 데이터가 있는 파일의 오프셋과 크기를 나타낸다. VirtualAddress와 Misc.VirtualSize는 각각 메모리에 로딩되어야 할 가상 주소와 메모리에서 섹션이 차지하는 공간을 나타낸다.

리스트 2 LoadPEImage, UnloadPEImage 함수

```
bool CDllLoader::LoadFile(CDllInfo * dll)
{
  ifstream file(dll->path.c_str(), ios::in | ios::binary);
  if(!file.is_open())
    return false;

  IMAGE_DOS_HEADER dos;
  IMAGE_NT_HEADERS nt;

  file.read((char *) &dos, sizeof(dos));
  if(dos.e_magic != IMAGE_DOS_SIGNATURE)
    return false;

  file.seekg(dos.e_lfanew);
  file.read((char *) &nt, sizeof(nt));
  if(nt.Signature != IMAGE_NT_SIGNATURE)
    return false;

  // 이미지 공간 예약 한다.
  dll->baseMem.Alloc((LPVOID)(SIZE_T) nt.OptionalHeader.ImageBase
        , nt.OptionalHeader.SizeOfImage
        , MEM_RESERVE
        , PAGE_READWRITE);
```

```
  if(dll->baseMem == NULL)
  {
    // 이미지를 로드할 베이스 주소가 사용 중인 경우 다른 주소를 할당한다.
    dll->baseMem.Alloc(NULL
           , nt.OptionalHeader.SizeOfImage
           , MEM_RESERVE
           , PAGE_READWRITE);

    if(dll->baseMem == NULL)
      return false;
  }

  // 헤더를 로드할 공간을 확보한다.
  dll->headerMem.Alloc(dll->baseMem
           , nt.OptionalHeader.SizeOfHeaders
           , MEM_COMMIT
           , PAGE_EXECUTE_READWRITE);

  if(dll->headerMem == NULL)
    return false;

  file.seekg(0);
  file.read((char *) dll->headerMem.Get(), nt.OptionalHeader.SizeOfHeaders);

  // 섹션 정보를 파악한다.
  PIMAGE_SECTION_HEADER sec = dll->SectionHeader();

  int sectionCnt = nt.FileHeader.NumberOfSections;
  dll->sectionMem = new CAutoArray<CVirtualMemory>(sectionCnt);
  CAutoArray<CVirtualMemory> &sectionMem = *dll->sectionMem;

  for(int i=0; i<sectionCnt; ++i)
  {
    // 섹션을 위한 메모리 공간을 확보한다.
    sectionMem[i].Alloc(GetPtr(dll->baseMem, sec[i].VirtualAddress)
             , sec[i].Misc.VirtualSize
             , MEM_COMMIT
             , PAGE_EXECUTE_READWRITE);

    if(sectionMem[i] == NULL)
      return false;

    // 로드할 섹션 데이터가 있는 경우에 파일에서 읽어 온다.
    if(sec[i].SizeOfRawData > 0)
    {
      file.seekg(sec[i].PointerToRawData);
      file.read((char *) sectionMem[i].Get(), sec[i].SizeOfRawData);
    }
  }

  return true;
}

void CDllLoader::UnloadFile(CDllInfo * dll)
{
  if(dll->sectionMem)
    delete dll->sectionMem;

  dll->headerMem.Free();
  dll->baseMem.Free();
}
```

박스 2 정렬

NT헤더의 OptionalHeader에는 두 개의 정렬 필드가 있다. 파일 크기 정렬을 나타내는 FileAlignment와 메모리 에서의 정렬을 나타내는 SectionAlignment 필드가 그것이다. 정렬이란 지정된 값의 배수가 되게 크기나 오프셋을 맞추어 주는 행위를 말한다. 쉽게 말하면 정렬 기준에

맞춰서 올림한다고 생각하면 된다.

정렬 기준이 8인 경우의 예를 살펴 보자. 이 경우에 14란 값을 단위에 맞게 정렬하면 16이 된다. 7을 정렬하면 8, 2를 정렬하면 8, 22를 정렬하면 24가 된다. 그렇다면 이렇게 정렬하는 코드는 어떻게 만들 수 있을까? 정렬 단위가 2의 배수인 경우에는 아래와 같은 간단한 코드를 사용하면 된다. 2의 배수가 아닌 경우라면 AND 대신 나누기 연산을 하면 된다. src가 정렬에 맞추려는 값이고, align이 정렬 단위가 된다. dest에 정렬된 값이 들어간다.

```
int src = 7;
int align = 8;
int dest = (src + align - 1) & ~(align-1);
```

재배치 수행하기

DLL을 메모리에 로딩했다면 이제 할 것은 절대 주소를 가진 코드들을 수정해 주는 것이다. 이 작업은 앞서 설명한 재배치 섹션을 통해서 이루어진다. 재배치를 수행하는 실제 함수 코드가 <리스트 3>에 나와 있다. 이해가 잘 되지 않는다면 앞쪽에 있는 재배치 섹션의 구조를 다시 한번 보도록 하자. 기본 로딩 주소에 로드된 경우와 재배치 섹션이 없는 경우는 재배치를 수행할 수 없다는 사실을 명심해야 한다.

리스트 3 Reloc 함수

```
bool CDllLoader::Reloc(CDllInfo *dll)
{
  PVOID base = dll->Base();
  PIMAGE_NT_HEADERS nt = dll->NTHeader();
  DWORD_PTR relDelta = (DWORD_PTR) base - (DWORD_PTR) nt->OptionalHeader.ImageBase;

  // 기본 이미지 베이스 주소에 로드된 경우라면 재배치를 할 필요가 없다.
  if(relDelta == 0)
    return true;

  // 재배치 섹션이 없는 경우라면 재배치를 수행할 수 없다.
  IMAGE_DATA_DIRECTORY rdd = dll->DataDirectory(IMAGE_DIRECTORY_ENTRY_BASERELOC);
  if(rdd.Size == 0)
    return false;

  int relCnt;
  WORD *relInfo;
  DWORD relType;
  DWORD relOffset;
  PVOID relBase;
  DWORD_PTR *relPatch;

  IMAGE_BASE_RELOCATION *reloc;
  reloc = (IMAGE_BASE_RELOCATION *) GetPtr(base, rdd.VirtualAddress);
  while(rdd.Size > 0)
  {
    // 재배치 기준 주소, 재배치 개수, 정보 목록을 구한다.
    relBase = GetPtr(base, reloc->VirtualAddress);
    relCnt = (reloc->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) / sizeof(WORD);
    relInfo = (WORD *) GetPtr(reloc, sizeof(IMAGE_BASE_RELOCATION));

    // 재배치를 수행한다.
    for(int i=0; i<relCnt; ++i)
    {
      relType = relInfo[i] >> 12;
```

```
    if(relType == IMAGE_REL_BASED_HIGHLOW)
    {
      relOffset = relInfo[i] & 0x0fff;
      relPatch = (DWORD_PTR *) GetPtr(relBase, relOffset);
      *relPatch += relDelta;
    }
   }

   // 다음 재배치 블록으로 이동한다.
   rdd.Size -= reloc->SizeOfBlock;
   reloc = (IMAGE_BASE_RELOCATION *) GetPtr(reloc, reloc->SizeOfBlock);
  }

  return true;
}
```

DLL에서 임포트한 함수 연결하기

윈도우 개발자라면 누구나 알고 있듯이 DLL도 다른 DLL을 연결해서 사용한다. 따라서 로드하려는 DLL에서 사용하는 다른 DLL도 같이 메모리에 올려주어야 한다. 또한 로딩하려는 DLL에서 사용하는 외부 DLL 함수들의 주소를 구해서 IAT에 기록해 주어야 한다. 함수 주소를 IAT에 기록해 주지 않거나 잘못된 주소를 기록한다면 어김없이 잘못된 연산 오류를 만날 것이다.

<리스트 4>에 IAT를 만들어주는 BuildIAT 함수가 나와있다. 함수 내부를 살펴보면 다른 DLL들을 로드하기 위해서 윈도우에서 제공하는 LoadLibrary를 사용하는 것을 볼 수 있다. 이 부분을 보고 DLL 로더가 완전하지 않다고 판단할 수도 있다. 물론 이곳에 우리가 만들 CDllLoader::LoadLibrary를 사용해도 된다. 하지만 일부 시스템 DLL은 메모리 상에 두 번 맵핑되면 오류가 발생하는 경우가 있다. 로더의 잘못이라기 보다는 DLL이 원래 그렇게 설계된 것이 주된 이유다. 따라서 그런 부분을 안전하게 처리하기 위해서는 기본으로 제공하는 LoadLibrary를 사용하는 것이 좋다.

소스를 분석할 때 한 가지 주의 깊게 보아야 하는 것은 IAT(Import Address Table)와 ILT(Import Lookup Table)를 둘 다 사용한다는 점이다. 일반적인 경우에는 IAT와 ILT가 동일한 내용을 담고 있다. 하지만 DLL이 바인드된 경우에는 IAT가 ILT와 다른 내용을 담고 있다. 바인드와 관련된 내용은 지면 관계상 지난 시간에 소개하지 못했다. 간단한 내용이기 때문에 궁금하신 분들은 참고 자료나 필자의 블로그(http://www.jiniya.net)에 올려진 내용을 참고하도록 하자.

리스트 4 BuildIAT 함수

```
bool CDllLoader::BuildIAT(CDllInfo * dll)
{
  PVOID base = dll->baseMem;
  IMAGE_DOS_HEADER *dos = (IMAGE_DOS_HEADER *) base;
  IMAGE_NT_HEADERS *nt = (IMAGE_NT_HEADERS *) GetPtr(dos, dos->e_lfanew);
  IMAGE_DATA_DIRECTORY idd = nt->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];

  if(idd.Size == 0)
    return true;

  IMAGE_IMPORT_DESCRIPTOR     *idesc     =     (IMAGE_IMPORT_DESCRIPTOR     *)     GetPtr(dos,
idd.VirtualAddress);
  IMAGE_THUNK_DATA *iat, *ilt;

  FARPROC func;
  HMODULE module;
  char *dllName;
  bool complete = false;
```

```
  // 임포트한 DLL 이름을 출력한다.
  while(idesc->Name)
  {
    dllName = (char *) GetPtr(base, idesc->Name);
    module = ::LoadLibraryA(dllName);
    if(!module)
      goto $cleanup;

    // 각 DLL에서 임포트한 함수 명을 출력한다.
    iat = (IMAGE_THUNK_DATA *) GetPtr(base, idesc->FirstThunk);
    if(idesc->OriginalFirstThunk)
      ilt = (IMAGE_THUNK_DATA *) GetPtr(base, idesc->OriginalFirstThunk);
    else
      ilt = iat;

    for( ; ilt->u1.AddressOfData; ++ilt, ++iat)
    {
      if(ilt->u1.Ordinal & 0x80000000)
      {
        func = ::GetProcAddress(module, (LPCSTR) (ilt->u1.Ordinal & 0x7fffffff));
      }
      else
      {
        PIMAGE_IMPORT_BY_NAME     ibn     =     (PIMAGE_IMPORT_BY_NAME)     GetPtr(base,     ilt-
>u1.AddressOfData);
        func = ::GetProcAddress(module, (LPCSTR) ibn->Name);
      }

      if(!func)
        goto $cleanup;

      iat->u1.Function = (DWORD)(DWORD_PTR) func;
    }

    ++idesc;
  }

  complete = true;

$cleanup:
  if(!complete)
  {
    for_each(dll->modules.begin(), dll->modules.end(), ::FreeLibrary);
    dll->modules.clear();
  }

  return complete;
}
```

박스 3 코딩은 손으로 하는 거야

처음 C++을 배울 때였다. 필자는 그 때 C언어에 익숙한 때라 C++은 금방 배울 줄 알았다. C++과 관련된 책 몇 권을 읽고는 C++도 C와 유사하다는 어줍잖은 생각을 했었다. 학과의 C++ 수업에서 처음으로 과제를 할 때였다. 책은 많이 읽었지만 한번도 C++ 프로그램을 작성해 보지 않았던 필자는 컴파일 에러 때문에 과제를 하는데 상당히 오랜 시간이 걸렸다. 시간을 갉아 먹은 에러는 다름 아닌 클래스 선언 끝에 세미콜론을 찍지 않는 것과 같이 사소한 것들이었다. 책을 읽는 과정에서 프로그램 하나만 작성해 보아도 겪지 않았을 그런 에러들이었다. 그 때 옆에 있던 선배가 이런 말을 했다. "코딩은 눈이나 귀로 하는 게 아니라 손으로 하는 거야".

그렇다. 결국 코드를 작성하는 일은 손으로 한다. 아무리 좋은 책을 많이 읽고, 아무리 훌륭한 사람에게 좋은 이야기를 많이 들었다고 하더라도 자신이 알고 있는 진정한 지식은 자신이 직접 해 본 것뿐이다. 나머지는 모두 불완전한 지식이다.

페이지 속성 변경하기

DLL을 메모리에 로딩하기 위한 마지막 단계는 할당한 페이지의 속성을 변경해 주는 것이다. CDllLoader에 있는 ProtectDll 함수가 이 작업을 해준다(<리스트 5> 참고). 우리는 VirtualAlloc을 사용해서 메모리를 직접 할당했기 때문에 공유 섹션이나 기록시 복사(copy on write)와 같은 특성을 제공하지 못한다. VirtualAlloc으로 할당된 메모리에 적용할 수 있는 속성은 실행(PAGE_EXECUTE), 실행읽기(PAGE_EXECUTE_READ), 실행읽기쓰기(PAGE_EXECUTE_READWRITE), 읽기전용(PAGE_READONLY), 읽기쓰기(PAGE_READWRITE)가 모두다. 각각을 섹션의 속성에 맞게 지정해 주면 된다. IMAGE_SCN_MEM_DISCARDABLE 속성이 지정된 섹션은 로딩이 완료되고 나면 더 이상 쓸 일이 없는 섹션이기 때문에 메모리에서 제거해 준다.

공유 섹션의 경우 로드해서 동작을 시키더라도 문제가 발생하지 않는다. 다만 원래 DLL이 동작하도록 설계된 내용과 다르게 동작할 뿐이다. 결국 원래 의도대로 동작하지 않기 때문에 ProtectDll에서는 공유 섹션이 있는 경우를 에러로 처리하고 있다.

리스트 5 ProtectDll 함수

```
bool CDllLoader::ProtectDll(CDllInfo * dll)
{
  PIMAGE_NT_HEADERS nt = dll->NTHeader();
  PIMAGE_SECTION_HEADER sec = dll->SectionHeader();

  CVirtualMemory *mem;
  for(int i=0; i<nt->FileHeader.NumberOfSections; ++i)
  {
    if(sec[i].Characteristics & IMAGE_SCN_MEM_SHARED)
      return false;

    mem = &(*dll->sectionMem)[i];

    if(sec[i].Characteristics & IMAGE_SCN_MEM_DISCARDABLE)
    {
      mem->Free();
    }
    else
    {
      DWORD protect = PAGE_READONLY;
      if(sec[i].Characteristics & IMAGE_SCN_MEM_EXECUTE)
      {
        protect = PAGE_EXECUTE;
        if(sec[i].Characteristics & IMAGE_SCN_MEM_READ)
          protect = PAGE_EXECUTE_READ;
        if(sec[i].Characteristics & IMAGE_SCN_MEM_WRITE)
          protect = PAGE_EXECUTE_READWRITE;
      }
      else if(sec[i].Characteristics & IMAGE_SCN_MEM_WRITE)
         protect = PAGE_READWRITE;

      if(sec[i].Characteristics & IMAGE_SCN_MEM_NOT_CACHED)
        protect |= PAGE_NOCACHE;

      VirtualProtect(mem, mem->Size(), protect, NULL);
    }
  }
  return true;
}
```

LoadLibrary/FreeLibrary 구현 하기

이제 LoadLibrary와 FreeLibrary를 구현할 수 있는 기본 작업들을 모두 한 셈이다. 앞서 설명한 함수들을 순차적으로 호출하면 LoadLibrary와 FreeLibrary가 구현된다.

<리스트 6>에 LoadLibrary 함수가 나와있다. 넘어온 경로의 DLL이 이미 로드된 경우에는 재차 로드하지 않고 해당 DLL의 정보를 찾아서 로드 횟수만 증가 시킨 다음 리턴 한다. 로드한 적이 없는 경우라면 CDllInfo 구조체를 할당하고 로드 작업을 시도한다. 로드 과정은 메모리 할당, 재배치, 바운드 검사, 임포트 정보 분석, 섹션 보호 속성 설정, DllMain 호출 순으로 이어진다. 각 단계별로 실패할 경우에는 할당된 리소스를 회수해 주어야 한다. 단계별로 할당된 리소스가 다르기 때문에 해제 작업에 신경을 써 주어야 한다. 모든 과정이 성공적으로 수행도면 클래스 멤버 변수인 맵에 해당 내용을 추가한다.

리스트 6 LoadLibrary 함수

```
PVOID CDllLoader::LoadLibrary(LPCTSTR path)
{
  CDllInfo * loadedDll = GetDll(path);
  if(loadedDll)
  {
    InterlockedIncrement((LONG *) &loadedDll->ref);
    return loadedDll->Base();
  }

  DllInfoPtr dll(new CDllInfo);

  dll->ref = 1;
  dll->path = path;
  if(!LoadFile(dll.get()))
    return NULL;

  if(!Reloc(dll.get()))
  {
    UnloadFile(dll.get());
    return NULL;
  }

  DWORD bound = IsBound(dll.get());
  if(bound < 0)
  {
    UnloadFile(dll.get());
    return NULL;
  }

  if(!bound && !BuildIAT(dll.get()))
  {
    FreeDll(dll.get());
    return NULL;
  }

  if(!ProtectDll(dll.get()))
  {
    FreeDll(dll.get());
    return NULL;
  }

  if(!dll->DllMain(DLL_PROCESS_ATTACH))
  {
    FreeDll(dll.get());
    return NULL;
  }

  EnterCriticalSection(&m_csLoadDlls);
  m_loadDlls.insert(make_pair((PVOID)dll->baseMem, dll));
  LeaveCriticalSection(&m_csLoadDlls);
```

```
  return dll->baseMem;
}
```

<리스트 7>에는 FreeLibrary 함수가 나와있다. FreeLibrary 함수는 정말 간단하다. 넘어온 베이스 주소에 해당하는 CDllInfo 구조체를 찾아서 로드 횟수를 감소 시킨다. 로드 횟수가 0인 경우에는 해당 DLL과 관련된 리소스를 제거한다. 리소스 제거 작업이 완료되면 최종적으로 로드한 DLL을 저장하고 있는 맵에서 해당 정보를 제거한다.

리스트 7 FreeLibrary 함수

```
void CDllLoader::FreeDll(CDllInfo * dll)
{
  InterlockedDecrement((LONG *) &dll->ref);

  if(dll->ref == 0)
  {
    dll->DllMain(DLL_PROCESS_DETACH);

    for_each(dll->modules.begin()
        , dll->modules.end()
        , ::FreeLibrary);

    UnloadFile(dll);
  }
}

bool CDllLoader::FreeLibrary(PVOID base)
{
  CDllInfo * dll = GetDll(base);
  if(!dll)
    return false;

  FreeDll(dll);

  if(dll->ref == 0)
  {
    EnterCriticalSection(&m_csLoadDlls);
    m_loadDlls.erase(base);
    LeaveCriticalSection(&m_csLoadDlls);
  }

  return true;
}
```

GetProcAddress 구현 하기

지난 시간에 배웠던 익스포트 테이블의 구조에 대해서 이해하고 있다면 GetProcAddress를 구현하는 것은 어렵지 않다. GetProcAddress가 하는 일은 익스포트 테이블을 검색해서 입력으로 들어온 함수와 같은 녀석을 찾아서 리턴하는 것이 전부다. <리스트 8>에 GetProcAddress를 구현한 함수가 나와있다.

한 가지 주의해야 할 점은 GetProcAddress의 두 번째 인자인 name 값이 두 가지 의미로 사용된다는 점이다. 문자열 포인터가 넘어온 경우에는 함수 이름을 의미하고, DWORD 값이 넘어온 경우에는 오디날을 의미한다. 둘 중 어떤 의미로 사용되었는지:는 name의 HIWORD가 0인지를 비교해서 알아낸다. 오디날인 경우는 그 값이 0이고, 문자열 포인터인 경우는 0이 아닌 값이 들어있다.

그렇다면 name으로 넘어온 문자열 포인터가 0x00001234와 같으면 어떻게 될까? 당연히 오류가

난다. 하지만 일반적인 경우에는 그런 일이 절대 발생하지 않는다. 왜냐하면 윈도우에서 0부터 0x10000번지 까지는 모두 사용할 수 없는 공간으로 예약해 두었기 때문이다.

리스트 8 GetProcAddress

```
FARPROC CDllLoader::GetProcAddress(PVOID base, LPCSTR name)
{
  CDllInfo * dll = GetDll(base);
  if(!dll)
    return NULL;

  PIMAGE_NT_HEADERS nt = dll->NTHeader();
  IMAGE_DATA_DIRECTORY edd = dll->DataDirectory(IMAGE_DIRECTORY_ENTRY_EXPORT);
  if(!edd.VirtualAddress || !edd.Size)
    return NULL;

  PIMAGE_EXPORT_DIRECTORY ied;
  ied = (PIMAGE_EXPORT_DIRECTORY) GetPtr(dll->Base(), edd.VirtualAddress);

  char *funcName;
  DWORD *funcs = (DWORD *) GetPtr(dll->Base(), ied->AddressOfFunctions);
  DWORD *names = (DWORD *) GetPtr(dll->Base(), ied->AddressOfNames);
  WORD *ordinals = (WORD *) GetPtr(dll->Base(), ied->AddressOfNameOrdinals);

  if(HIWORD(name) == 0)
  {
    WORD ordinal = LOWORD(name) - ied->Base;
    if(ordinal < ied->NumberOfFunctions && funcs[ordinal])
      return (FARPROC) GetPtr(base, funcs[ordinal]);
  }
  else
  {
    for(DWORD i=0; i<ied->NumberOfNames; ++i)
    {
      funcName = (char *) GetPtr(base, names[i]);
      if(strcmp(funcName, name) == 0 && funcs[ordinals[i]])
        return (FARPROC) GetPtr(base, funcs[ordinals[i]]);
    }
  }

  return NULL;
}
```

CDllLoader 사용하기

지금까지 제작한 CDllLoader를 사용해서 DLL을 로드해서 사용해 보도록 하자. 기본적인 윈도우 API와 사용법이 동일하기 때문에 별로 어려운 점은 없다. <리스트 9>에는 CDllLoader를 사용해서 지난 시간에 제작했던 dummydll.dll을 로드해서 사용하는 코드가 나와있다.

리스트 9 CDllLoader를 사용하는 코드

```
typedef void (*PrintMsgFunc)(const char *msg, DWORD len);
typedef int (*PlusFunc)(int , int);
int _tmain(int argc, _TCHAR* argv[])
{
  CDllLoader loader;
  PVOID base = loader.LoadLibrary(_T("dummydll.dll"));
  if(base)
  {
    PrintMsgFunc PrintMsg;
    PrintMsg = (PrintMsgFunc) loader.GetProcAddress(base, MAKEINTRESOURCEA(2));
    PrintMsg("Hello", 5);

    PlusFunc Plus;
    Plus = (PlusFunc) loader.GetProcAddress(base, "Plus");
    int a = Plus(3, 4);
    printf("%d\n", a);
```

```
    loader.FreeLibrary(base);
  }
  return 0;
}
```

도전 과제

우리가 제작한 DLL 로더는 몇 가지 제약 사항을 가지고 있었다. 제약 사항의 근본 원인은 메모리 할당을 VirtualAlloc으로 했기 때문에 생기는 제약이었다. 그렇다면 윈도우는 내부적으로 어떻게 실제로 DLL들을 로딩하는 것일까? 이에 대한 해답이 참고 자료에 있는 "What Goes On Inside Windows 2000: Solving the Mysteries of the Loader"란 글에 나와있다. 글을 읽어 보면 나와있는 것과 같이 윈도우는 내부적으로 NtMapViewOfSection이란 함수를 사용해서 메모리를 맵핑한다.

VirtualAlloc이 아닌 NtMapViewOfSection을 사용해서 DLL을 로딩하는 로더를 만들어 보자. 이번 시간에 만든 부분에서 메모리 할당 부분만 손보면 된다. NtMapViewOfSection 함수의 원형과 설명은 구글에서 검색하면 쉽게 찾을 수 있다.

참고자료

What Goes On Inside Windows 2000: Solving the Mysteries of the Loader
http://msdn.microsoft.com/msdnmag/issues/02/03/Loader/

"Windows 시스템 실행 파일의 구조와 원리"
이호동저, 한빛미디어

An In-Depth Look into the Win32 Portable Executable File Format
http://msdn.microsoft.com/msdnmag/issues/02/02/PE/

An In-Depth Look into the Win32 Portable Executable File Format, Part 2
http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/

Peering Inside the PE: A Tour of the Win32 Portable Executable File Format
http://msdn2.microsoft.com/en-us/library/ms809762.aspx