

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

Reverse Engineering Code with IDA Pro



RE Study Group

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

본 문서는 **KISC (Korea Information Security Community)**의 소모임 **Reverse Engineering Study**에서 발표한 문서이며 본 저작권은 **KISC**에 있습니다.

Study에 참여 하였던 분들

애화몽, 엔에스엠, 태검, 고양이, 프리, 가리온, 인클, 프리윌리, 네오위즈

그 외 **Study**가 원활히 운영될 수 있도록 애써주셨던 **ZIZI** 님,
부족한 부분을 위해 강의자료를 준비 및 강의 해주신 제스리버, 이명수 님께
감사 드립니다.

목 차

| | |
|--|-----|
| 0x03. Portable Executable and Executable and Linking Formats | 1 |
| Portable Executable Format | 1 |
| 0x04. Walkthroughs One and Two | 21 |
| Introduction | 21 |
| 0x05. Debugging | 31 |
| Introduction | 31 |
| 0x06. Anti-Reversing | 36 |
| MASM | 36 |
| 0x08. Reversing Malware | 74 |
| 악성코드 파일 분석 방법절차 | 74 |
| 1 악성코드 분석 환경 구성 | 74 |
| 1.1 가상 환경 구성(VMware) | 74 |
| 1.2 SysAnalyzer | 75 |
| 1.3 Sysinternal suit | 76 |
| 1.4 기타 도구들 | 78 |
| 2 Manual UnPacking | 79 |
| 2.1 실행압축(Packing)이란? | 79 |
| 2.2 OEP(Original Entry Point) 유형 | 81 |
| 2.3 Stack을 이용한 packer의 mup 방법 (pusha/popa) | 82 |
| 3 Embed string Analysis | 94 |
| 3.1 악성코드에서 자주 사용하는 API 목록 | 94 |
| 3.2 Embed string 값을 이용한 분석방법 | 94 |
| 4 간단한 IDA Pro 사용법 | 97 |
| 4.1 The Main Window | 97 |
| 4.2 Name Tag | 100 |
| 4.3 Strings Tag | 100 |
| 4.4 Import Viewer Tag | 101 |
| 4.5 Cross- reference Tag | 101 |
| 4.6 Functions Tag | 102 |
| 4.7 Arrow Tag | 102 |
| 5 Debugging with OllyDBG&IDA | 103 |

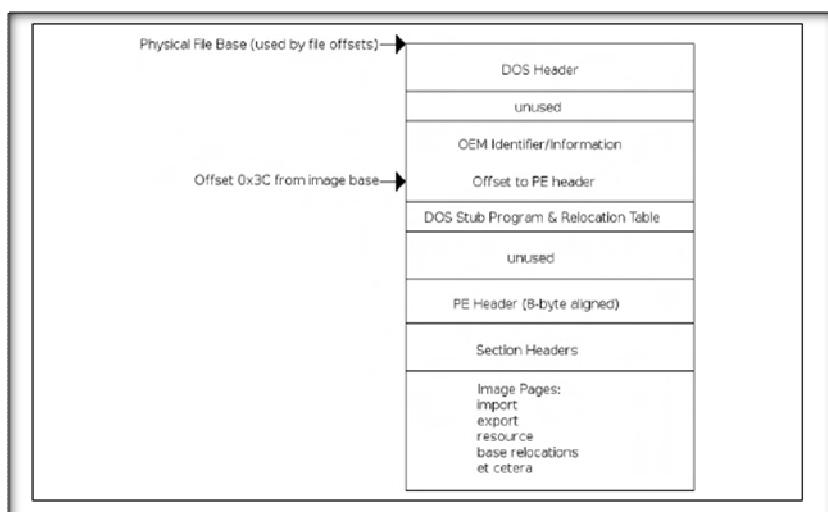
| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

| | | |
|-------|---|-----|
| 5.1 | Beep.sys 생성 및 실행 함수(sub_401B50) | 103 |
| 5.2 | nbjs.dll 생성 함수(sub_4012F0) | 104 |
| 5.3 | BITS 서비스 설정 초기화 함수(sub_4010C0) | 106 |
| 5.4 | nbjs.dll을 BITS 서비스 키에 등록 함수(sub_4011A0) | 106 |
| 5.5 | BITS 서비스 시작 설정 함수(sub_401000) | 108 |
| 5.6 | 고소장처리결과보고.exe 파일 삭제 함수(sub_401400) | 109 |
| 6 | 악성코드 증상 분석 | 111 |
| 6.1 | API 함수 분석 | 111 |
| 6.2 | 생성/변조/삭제 된 파일 리스트 | 113 |
| 0x09. | IDA Scripting and Plug-in..... | 115 |
| | IDC란?..... | 115 |
| | IDC Syntax | 116 |
| | Script Samples | 126 |
| | Useful IDC Functions..... | 128 |
| | Plug-ins..... | 129 |
| | Plug-in Syntax | 130 |
| | Setting up the Development Environment | 132 |
| | Simple Plug-in Examples | 133 |
| | Third-party Scripting Plug-ins..... | 135 |

0x03. Portable Executable and Executable and Linking Formats**Portable Executable Format**

Composition

- Dos Header - PE 파일의 시작부분으로 메모리상의 ImageBase에서 찾을 수 있음 (64Byte)
- Dos Stub code - DOS Header 바로 뒤에 위치 즉, ImageBase로부터 64Byte 떨어진 곳에서 찾을 수 있음
- PE- Header
 - DOS Header에 있는 e_lfanew 값을 이용하여 위치를 계산 할 수 있음
 - e_lfanew는 DOS Header의 마지막에 위치하며 크기는 4Byte
 - 저장된 값은 파일시작부터 PE Header 까지의 Offset
 - PEsignature(4byte)+FileHeader(20byte)+OptionHeader(224byte이상)
 - File Header의 값은 PE Header의 크기
- Section Table
 - PE Header 바로 뒤에 위치
 - 즉 시작점은 PE Header 주소 + File Header 값
- Section
 - Section table에 저장된 Section Header를 통해 확인
 - VirtualAddress -> 메모리상에서의 Section Address
 - PointToRawData -> 디스크상에서의 Section Address
 - 섹션 정렬단위 – FileAlignment , SectionAlignment (in PE Header)

**그림 1 Typical PE Layout**

■ ***Optional Header***

| Offset | Size | Filed Name |
|--------|------|-------------------------------|
| 0 | 2 | Magic |
| 2 | 1 | MajorLinkerVersion |
| 3 | 1 | MinorLinkerVersion |
| 4 | 4 | SizeOfCode |
| 8 | 4 | SizeOfInitializeData |
| 12 | 4 | SizeOfUninitializeData |
| 16 | 4 | EntryPoint |
| 20 | 4 | BaseOfCode |
| 24 | 4 | BaseOfData |
| 28 | 4 | ImageBase |
| 32 | 4 | SectionAlignment |
| 36 | 4 | FileAlignment |
| 40 | 2 | MajorOSVersion |
| 42 | 2 | MinorOSVersion |
| 44 | 2 | MajorImageVersion |
| 46 | 2 | MinorImageVersion |
| 48 | 2 | MajorSubsystemVersion |
| 50 | 2 | MinorSubsystemVersion |
| 52 | 4 | Win32VersionValue |
| 56 | 4 | SizeOfImage |
| 60 | 4 | SizeOfHeaders |
| 64 | 4 | Checksum |
| 68 | 2 | Subsystem |
| 70 | 2 | DLLCharacteristics |
| 72 | 4 | SizeOfStackReserve |
| 76 | 4 | SizeOfStarckCommit |
| 80 | 4 | SizeOfHeapReserve |
| 84 | 4 | SizeOfHeapCommit |
| 88 | 4 | LoaderFlags |
| 92 | 4 | NumberOfRVAsAndSizes |

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

다음 내용은 위의 테이블 (굵은표시) 의 세부내용입니다.

- ◆ Magic
 - 0x10B로 고정 , Optional Header를 구분하는 시그니처
- ◆ EntryPoint
 - PE 파일이 메모리에 로드 된 후 맨 처음으로 실행되어야 할 코드의 주소
 - **가상주소가 아닌 RVA 를 가진다.** (ImageBase로부터의 Offset)
 - **일반적으로 .txt Section의 시작점**
- ◆ ImageBase
 - 메모리에 로드된 PE파일의 시작점
- ◆ SectionAlignment
 - 메모리상에서 Section 사이즈 기준 (일반적으로 4096이상)
 - 즉 각 Section의 시작 번지는 SectionAlignment 의 정수배
- ◆ FileAlignment
 - 디스크상에서의 Section 사이즈 기준 (512 or 512 x 짹수배)
- ◆ SizeOfImage
 - 메모리상에 로드된 PE파일의 총 사이즈 (SectionAlignment 의 정수배)
- ◆ SizeOfHeader
 - 디스크상에서의 헤더의 총 사이즈
 - DOS Header에서 Padding을 포함한 Section헤더의 끝까지의 사이즈
 - FileAlignment의 정수배 – 메모리에 로드 되어도 변경 안됨
- ◆ SubsystemVersion
 - Major 4 , Minor 0 (Win32)
- ◆ SizeOfStack
 - 일반적으로 Reserve 0x1000 , Commit 0x1000
- ◆ SizeOfHeap
 - 일반적으로 Reserve 0x1000 , Commit 0x1000
- ◆ Subsystem – CUI 0x3 , GUI 0x2

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //
    WORD    Magic;
    BYTE   MajorLinkerVersion;
    BYTE   MinorLinkerVersion;
    DWORD  SizeOfCode;
    DWORD  SizeOfInitializedData;
    DWORD  SizeOfUninitializedData;
    DWORD  AddressOfEntryPoint;
    DWORD  BaseOfCode;
    DWORD  BaseOfData;
    //
    // NT additional fields.
    //
    DWORD  ImageBase;
    DWORD  SectionAlignment;
    DWORD  FileAlignment;
    WORD   MajorOperatingSystemVersion;
    WORD   MinorOperatingSystemVersion;
    WORD   MajorImageVersion;
    WORD   MinorImageVersion;
    WORD   MajorSubsystemVersion;
    WORD   MinorSubsystemVersion;
    DWORD  Win32VersionValue;
    DWORD  SizeOfImage;
    DWORD  SizeOfHeaders;
    DWORD  CheckSum;
    WORD   Subsystem;
    WORD   DllCharacteristics;
    DWORD  SizeOfStackReserve;
    DWORD  SizeOfStackCommit;
    DWORD  SizeOfHeapReserve;
    DWORD  SizeOfHeapCommit;
    DWORD  LoaderFlags;
    DWORD  NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

그림 2 구조체의 모습

■ Date Directory

PE 헤더의 마지막에 위치에 위치한 배열 (128Byte)

존재할 수도 있고 없을 수도 있다.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

그림 3 Date Directory

배열의 구성요소를 엘리먼트라 하며 그의 총 개수는 PE Header의 NumberOfRvaAndSize에 저장되어 있다.

각 엘리먼트는 Export Table, Import Table등의 개체(보통 16개)들의 위치,크기 정보를 담고 있다.

위치 값은 VA가 아닌 RVA(ImageBase로 부터의 상대주소)값이다.

■ Import Table

- ◆ OriginalFirstThunk
 - ILT(Import Lookup Table)의 주소를 저장[RVA]
- ◆ TimeStamp
- ◆ ForwarderChain
- ◆ FirstThunk – IAT(Import Address Table)의 주소를 저장[RVA]

```
typedef struct _IMAGE_THUNK_DATA32
{
    Union
    {
        DWORD ForwarderString;
        DWORD Function;
        DWORD Ordinal;
        DWORD AddressOfData;
    } u1;
} IMAGE_THUNK_DATA32
```

그림 4 ILT / IAT – IMAGE_THUNK_DATA로 구성된 배열

IMAGE_THUCK_DATA의 특징

4Byte union

IMAGE_IMPORT_BY_NAME의 주소를 가르킨다.(before binding)

AddressOfData(binding) 혹은 Ordinal, 또는 Function의 의미로 사용된다(IAT only)

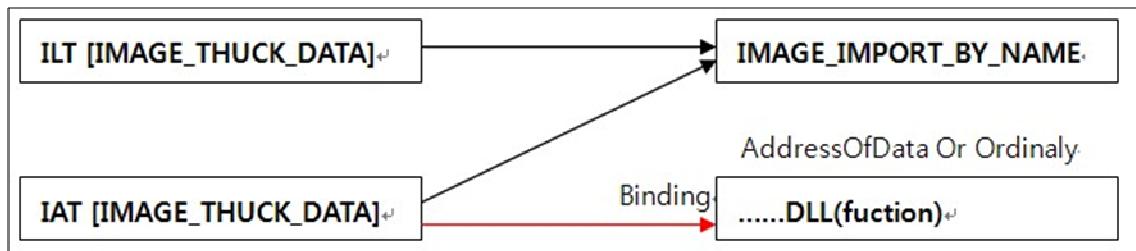


그림 5 ILT에서의 **IMAGE_THUCK_DATA**와 IAT에서의 **IMAGE_THUCK_DATA** 차이점

■ Export Table

PE 로더가 프로그램을 실행할 때 프로세스 주소 공간에 있는 관계된 DLLs를 로드하게 됩니다.

그런 후 PE loader는 메인 프로그램으로부터 import 된 function 들에 관한 정보를 추출합니다. PE loader는 이 정보를 이용하여 메인 프로그램으로 패치될 function 들의 주소를 위한 DLL들을 검색하게 됩니다. PE loader가 function 들의 주소를 찾는 공간의 DLL들은 export 테이블입니다.

DLL/EXE는 다른 DLL/EXE에 의해 사용될 function 을 export하는데, 이는 두 가지 방식으로 진행됩니다. 이름에 의한 export와 순서에 의한 export가 그것입니다.

만약에 DLL 내에 "GetSysConfig"라는 이름의 function이 있다고 한다면, 해당 DLL/EXE는 다른 DLLs/EXEs에게 해당 function을 호출하려면 GetSysConfig처럼 그 이름을 지정하라고 전달(얘기)합니다.

또 다른 방법은 Ordinal에 의한 것입니다.

Ordinal은 DLL내에서 함수의 유일한 정보로서 16-bit의 숫자입니다.

예를 들어 DLL이 Ordinal 16으로 함수를 export합니다. 이때 다른 DLL/EXE는 GetProcAddress에서 Ordinal 값으로 함수를 찾게 됩니다.

Ordinal에 의한 export는 되도록 사용하지 말아야 하는데 그 이유는 DLL 관리에 문제가 야기되기 때문입니다.

DLL이 upgrade/update되는 경우 변경하려고 하는 DLL에 의존적인 프로그램이 멈출 수 있기 때문에 Ordinal을 변경 할 수 없습니다.



그림 6 Export structure는 IMAGE_EXPORT_DIRECTORY

◆ Sequence

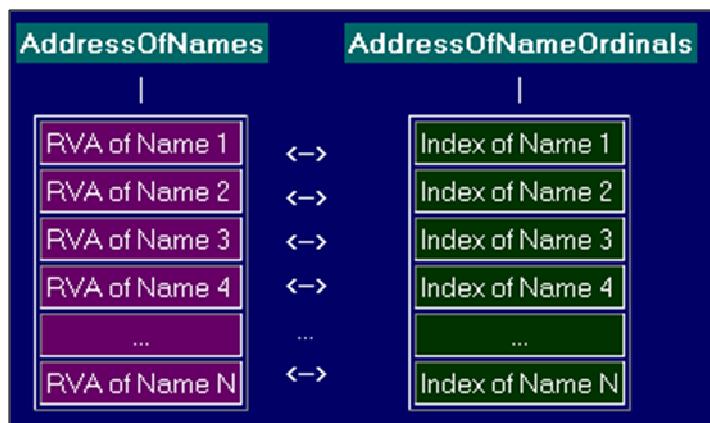


그림 7 Sequence

PE Header에서 data directory의 주소를 찾고 Export table의 위치를 얻습니다.

IMAGE_EXPORT_DIRECTORY의 멤버중 NumberOfNames 와 NumberOfFunction의 값을 찾습니다.

AddressOfName 과 AddressOfNameOrdinals 를 순차적으로 병행 검색하여 함수의 이름 혹은 Ordinal을 찾는다

이때 EOT(AddressOfNameOrdinals)에서 추출된 값은 EAT(ExportAddressTable)를 가르키는 인덱스 입니다.

◆ **Section Table**

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

그림 8 Section Table Header 구성

- ◆ Section Header는 IMAGE_SECTION_HEADER 구조체로 구성되어 있습니다.
PE 로더가 각 섹션을 메모리에 로드하고 속성을 설정하는데 필요한 정보를 가지고 있습니다.

◆ **Name**

- 섹션의 이름 Max 8byte,

◆ **VirtualAddress**

- 섹션이 로드될 가상주소(RVA)

◆ **SizeOfRawData**

- 디스크상에서의 섹션의 사이즈

◆ **PointerToRawData**

- 디스크상에서의 섹션시작 위치

◆ **Characteristics**

- 섹션의 속성 값 (Execution – Read - Write)

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

ELF (Executable and Linkable Format) File Format

- About ELF File Format ?

ELF 는 원래 UNIX 시스템 연구소에서 Application Binary Interface(ABI)의 한 부분으로

개발되고 공개되었다. Tool Interface Standards committee(TIS)는 ELF 표준을 다양한

운영체제를 위해서, 32 비트 인텔 아키텍처 환경에서 동작하는 이식 가능한 목적파일로

선택하였다. ELF 표준은 다양한 운영체제에 걸쳐서 사용될 수 있는 이진 인터페이스를

프로그래머에게 제공함으로써, 소프트웨어 개발에 연계성을 주기 위해 만들어졌다.

따라서 서로 다른 여러 인터페이스의 구현을 방지하고, 그럼으로써 프로그램을 다시 짜고

재컴파일해야 할 필요성을 줄이게 된다.

- Object File

목적파일은 어셈블러와 링커에 의해 생성된, CPU에 의해 직접 실행될 수 있는 프로그램의 이진 형식이다.

- ◆ Relocatable file

- 재배치 파일, 다른 object file 과 연결되어 executable file 이나 Shared object file 을

생성할 수 있는 정보를 가지고 있음

- ◆ Executable file

- Exe가 프로그램 실행에 필요한 정보를 가짐

- ◆ Shared object file

- 두 가지 방법의 linking 정보를 가짐 (ld[sd_cmd], dynamic linker)

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

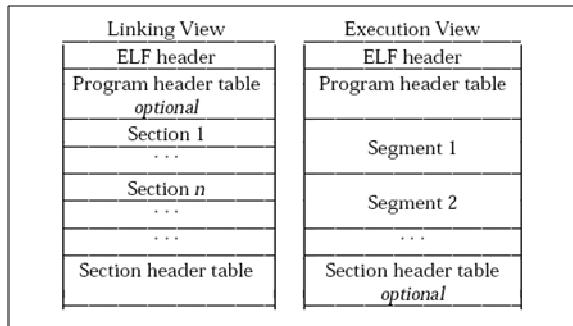


그림 9 Object File Format

(계속)

** Program header table을 ELF header 바로 뒤에 표시하고 각 섹션들 다음에 섹션헤더 테이블을 나타냈지만, 실제의 목적파일의 구성은 다를 수 있다.
섹션과 세그먼트들은 순서에 대한 규정이 없고 단지 ELF header 만 object file의 맨 처음에 위치해야 한다는 규칙만이 존재한다.

■ Data representation

| Name | Size | Alignment | Purpose |
|---------------|------|-----------|--------------------------|
| Elf32_Addr | 4 | 4 | Unsigned program address |
| Elf32_Half | 2 | 2 | Unsigned medium integer |
| Elf32_Off | 4 | 4 | Unsigned file offset |
| Elf32_Sword | 4 | 4 | Signed large integer |
| Elf32_Word | 4 | 4 | Unsigned large integer |
| unsigned char | 1 | 1 | Unsigned small integer |

그림 10 32-Bit Data Types

* * 목적 파일 형식은 8-bit 바이트와 32-bit 바이트 구조를 갖는 여러 프로세서를 지원한다. 그럼에도 불구하고 더 큰(혹은 더 작은) 구조의 프로세서로 확장할 수 있다. 목적 파일은 기계 독립적인 형식을 갖는 제어 자료를 표현해서 목적 파일을 식별할 수 있도록 하고 공통의 방법으로 내용을 변역할 수 있도록 한다. 파일의 나머지 자료 파일이 생성된 기계에 상관없이 목표 프로세서의 부호를 사용한다.

목적 파일 형식을 정의하는 모든 자료 구조는 상대적 클래스에 대하여 “natural size”와 정렬 지침을 따른다. 필요하다면 4-bit 오브젝트에 대해 4-bit 정렬로 하기 위하여 패딩을 하여 자료 구조를 4의 배수로 한다. 따라서 자료 파일의 처음부터 적절한 정렬을 찾는다. 그래서 예를 들어 Elf32_Addr 구성을 갖는 자료 구조는 파일내에 4-bit로 정렬한

다. 이식성 때문에 ELF는 비트-필드를 사용하지 않는다.

■ **ELF Header**

헤더에서 자료 구조의 ELF 실제 크기를 갖고 있기 때문에 어떤 제어 자료구조는 커질 수

있다. 목적 파일 형식이 변화한다면 프로그램은 기대했던 것보다 커지거나 작아진 자료 구

조를 만나게 된다. 그러므로 프로그램은 여분의 정보를 무시할 수 있다. 손실된 정보는 문

맥에 따르게 되고 확장이 정의되어 있다면 지정에 따르게 될 것이다

```
#define EI_NIDENT      16

typedef struct {
    unsigned char   e_ident[EI_NIDENT];
    Elf32_Half     e_type;
    Elf32_Half     e_machine;
    Elf32_Word     e_version;
    Elf32_Addr    e_entry;
    Elf32_Off      e_phoff;
    Elf32_Off      e_shoff;
    Elf32_Word     e_flags;
    Elf32_Half     e_ehsize;
    Elf32_Half     e_phentsize;
    Elf32_Half     e_phnum;
    Elf32_Half     e_shentsize;
    Elf32_Half     e_shnum;
    Elf32_Half     e_shstrndx;
} Elf32_Ehdr;
```

그림 11 **ELF Header**

- ◆ **e_ident**
 - object file임을 나타냄
- ◆ **e_type**
 - object file type을 나타냄(NONE, REL, EXEC, DYN, LOPROC, HIPROC)
- ◆ **e_machine**
 - object file이 사용되는 machine의 종류를 나타냄
(M32, SPARC, 386, MIPS...)
- ◆ **e_version**
 - object file의 버전을 나타냄 (0, 1[current])

◆ **e_entry**

- 실제 프로그램이 실행되는 가상의 주소값을 나타냄

◆ **e_phoff**

- object file 내에서 program header의 시작위치를 나타냄

◆ **e_shoff**

- object file 내에서 섹션헤더 테이블의 시작위치를 나타냄

◆ **e_flags**

- 프로세서에 관련된 flag

◆ **e_ehsize**

- ELF Header의 크기

◆ **e_phentsize**

- Program header table의 각 entry size

◆ **e_phnum**

- Program header table의 전체 entry 개수

◆ **e_shentsize**

- Section Header의 각 entry size

◆ **e_shnum**

- section header의 entry 개수

◆ **e_shstrndx**

- section name string table을 나타내는 section header 내의 entry index

■ **Section Header**

Section Header table은 파일안의 모든 섹션들의 위치를 나타낸다..

| Name | Index/Value |
|---------------|-------------|
| SHN_UNDEF | 0 |
| SHN_LORESERVE | 0xFF00 |
| SHN_LOPROC | 0xFF00 |
| SHN_HIPROC | 0xFF1F |
| SHN_ABS | 0xFFFF1 |
| SHN_COMMON | 0xFFFF2 |
| SHN_HIRESERVE | 0xFFFF |

그림 12 Special Indexes for ELF Section Header Table Array

- ◆ SHN_UNDEF
 - 정의 되지 않았거나 없는 경우 혹은 관계가 없는 경우 의미없는 Section 참조 등을 나타낸다.
- ◆ SHN_LORESERVE
 - 예약된 인덱스 영역의 하위경계를 나타낸다.
- ◆ SHN_LOPROC/HIPROC
 - 프로세스에 의존적인 예약된 값
- ◆ SHN_ABS
 - Relocation에 독립적이고 관련된 symbol은 절대값을 가진다.
- ◆ SHN_COMMON
 - 공통된 일반적인 symbol
- ◆ SHN_HIRESERVE
 - 인덱스 영역의 상위 경계

** System은 SHN_LORESERVE ~ SHN_HIRESERVE 사이의 인덱스들을 예약하고 있으며 이러한 값은 Section table을 참조하지 않는다.

✧ Object file의 Section의 조건

- 모든 Section은 자신의 Section 헤더 하나를 가진다
- 모든 Section은 파일내에서 연속적인 바이트열을 가진다
- Object file의 모든 바이트를 설명하지 않는다

```
typedef struct {
    Elf32_Word      sh_name;
    Elf32_Word      sh_type;
    Elf32_Word      sh_flags;
    Elf32_Addr     sh_addr;
    Elf32_Off       sh_offset;
    Elf32_Word      sh_size;
    Elf32_Word      sh_link;
    Elf32_Word      sh_info;
    Elf32_Word      sh_addralign;
    Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

그림 13 Section Header

- ◆ sh_name
 - section의 이름 , section header string table의 인덱스
- ◆ sh_type
 - section의 내용과 의미
- ◆ sh_flags
 - section의 세부적 특성 , 1bit
- ◆ sh_addr
 - section이 메모리에 적재되었을 때 첫바이트 주소 ,적재되지 않는다면 0
- ◆ sh_offset
 - object file의 처음부터 section의 첫바이트 까지의 상대 offset section type이 SHT_NOBITS 인 경우, section은 object file 내에서 차지 하는 공간이 없고 이때의 sh_offset은 object file에서 개념적인 위치를 나타낸다
- ◆ sh_size
 - section size , type이 SHT_NOBITS가 아닌 경우 0
- ◆ sh_link
 - section header table의 인덱스 링크
- ◆ sh_info
 - section의 부가적인 정보
- ◆ sh_addralign
 - 특정 section 들은 주소값이 정렬되어야 한다(sh_addr%sh_addralign=0)
- ◆ sh_entsize
 - 특정 section들은 고정된 크기의 entry를 가지며 그 entry size이다(Symbol)

| Name | Value |
|--------------|------------|
| SHT_NULL | 0 |
| SHT_PROGBITS | 1 |
| SHT_SYMTAB | 2 |
| SHT_STRTAB | 3 |
| SHT_REL | 4 |
| SHT_HASH | 5 |
| SHT_DYNAMIC | 6 |
| SHT_NOTE | 7 |
| SHT_NOBITS | 8 |
| SHT_REL | 9 |
| SHT_SHLIB | 10 |
| SHT_DYNSYM | 11 |
| SHT_LOPROC | 0x70000000 |
| SHT_HIPROC | 0x7fffffff |
| SHT_LOUSER | 0x80000000 |
| SHT_HIUSER | 0xffffffff |

그림 14 Section Types, ah_type

- ◆ SHT_NULL
 - section header가 유효하지 않음
- ◆ SHT_PROGBITS
 - 프로그램에 정의된 정보를 담고 프로그램에 의해서만 사용
- ◆ SHT_SYMTAB SHT_DYNSYM
 - symbol table을 가진다
- ◆ SHT_STRTAB
 - section string table
- ◆ SHT_REL
 - 재배치 엔트리를 가진 section (가수를 가짐 +)
- ◆ SHT_HASH
 - symbol hash table을 가진 section
- ◆ SHT_DYNAMIC
 - dynamic link에 필요한 정보를 담고있는 섹션
- ◆ SHT_NOTE
 - 파일에 표시를 하는 정보를 가진다
- ◆ SHT_NOBITS
 - object file에 아무런 공간을 차지 하지 않음
- ◆ SHT_REL
 - 재배치 엔트리를 가진 section (가수가 없음)

- ◆ SHT_SHLIB
 - ABI를 따르지 않는다
- ◆ SHT_LOPROC/ SHT_HIPROC
 - 각 프로세서에 의존적인 정보를 담는다
- ◆ SHT_LOUSER
 - 프로그램에 예약된 하한값
- ◆ SHT_HIUSER
 - 프로그램에 예약된 상한값

| Name | Value |
|---------------|------------|
| SHF_WRITE | 0x1 |
| SHF_ALLOC | 0x2 |
| SHF_EXECINSTR | 0x4 |
| SHF_MASKPROC | 0xf0000000 |

그림 15 Section Attribute Flags, sh_flags

(계속)

섹션 헤더의 sh_flags는 1-bit 플래그로 섹션의 성격을 설명한다. 정의된 값은 다음과 같고 다른 값들은 예약되어 있다.

- ◆ SHF_WRITE
 - process가 실행 중일 때 쓰여질 수 있는 데이터를 가진다
- ◆ SHF_ALLOC
 - process가 실행 중일 때 메모리를 차지한다
- ◆ SHF_EXEC
 - 실행될 수 있는 기계어 명령을 가진다
- ◆ SHF_MASK
 - processor에 의존적인 의미를 지니는 값을 가진다

| Name | Type | Attributes |
|-----------|--------------|---------------------------|
| .bss | SHT_NOBITS | SHF_ALLOC + SHF_WRITE |
| .comment | SHT_PROGBITS | none |
| .data | SHT_PROGBITS | SHF_ALLOC + SHF_WRITE |
| .data1 | SHT_PROGBITS | SHF_ALLOC + SHF_WRITE |
| .debug | SHT_PROGBITS | none |
| .dynamic | SHT_DYNAMIC | see below |
| .dynstr | SHT_STRTAB | SHF_ALLOC |
| .dynsym | SHT_DYNSYM | SHF_ALLOC |
| .fini | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |
| .got | SHT_PROGBITS | see below |
| .hash | SHT_HASH | SHF_ALLOC |
| .init | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |
| .interp | SHT_PROGBITS | see below |
| .line | SHT_PROGBITS | none |
| .note | SHT_NOTE | none |
| .plt | SHT_PROGBITS | see below |
| .relname | SHT_REL | see below |
| .relaname | SHT_REL | see below |
| .rodata | SHT_PROGBITS | SHF_ALLOC |
| .rodata1 | SHT_PROGBITS | SHF_ALLOC |
| .shstrtab | SHT_STRTAB | none |
| .strtab | SHT_STRTAB | see below |
| .syms | SHT_SYMTAB | see below |
| .text | SHT_PROGBITS | SHF_ALLOC + SHF_EXECINSTR |

그림 16 Special Sections

다양한 섹션이 프로그램과 제어 정보를 갖고 있다. 위에 있는 섹션은 시스템에서 사용하고 종류와 속성을 가진다.

- ◆ .bbs
 - program의 메모리 영역에 초기화 되지 않은 데이터
- ◆ .comment
 - 버전 제어정보
- ◆ .data/.data1
 - program의 메모리 영역에 초기화된 데이터
- ◆ .debug
 - symbolic debugging에 대한 정보
- ◆ .dynamic
 - dynamic linking에 필요한 정보
- ◆ .dynstr
 - dynamic linking에 필요한 문자열
- ◆ .dynsym
 - dynamic symbol table

- ◆ .fini
 - process의 종료 코드에 제공될 실행 명령어를 가지고 있다
- ◆ .got
 - global offset table
- ◆ .hash
 - symbol hash table
- ◆ .init
 - process 초기화에 사용될 실행 명령어를 가지고 있다
- ◆ .interp
 - 프로그램 분석기의 경로명
- ◆ .line
 - symbolic debugging을 위한 라인번호
- ◆ .note
 - 이 정보는 옵션, 다른 프로그램의 호환성을 검사할 필요가 있을 때..
- ◆ .plt
 - 프로시저 연결 테이블
- ◆ .rename/.reaname
 - 재배치 정보를 담고 있다
- ◆ .rodata/.rodata1
 - 프로세스의 이미지에서 쓰기 불가능한 세그먼트에 사용되는 전용데이터

- ◆ .shstrtab
 - 섹션의 이름
- ◆ .strtab
 - 심볼 테이블과 관련된 문자열
- ◆ .symtab
 - 심볼 테이블을 담고 있다
- ◆ .text
 - 프로그램의 텍스트 또는 실행 가능한 명령

■ Program Header

실행 혹은 공유 오브젝트 파일의 프로그램 헤더는 자료 구조의 배열로 시스템이 프로그램을 실행시키기 위하여 필요한 세그먼트나 다른 정보를 설명한다. 오브젝트 파일의 세그먼트는 하나 이상의 섹션을 갖고 있다. 프로그램 헤더는 실행 파일과 공유 오브젝트 파일에만 의미가 있다. 파일에서 ELF헤더의 e_phentsize와 e_phnum으로 프로그램 헤더의 크기를 지정한다.

```
typedef struct {
    Elf32_Word      p_type;
    Elf32_Off       p_offset;
    Elf32_Addr     p_vaddr;
    Elf32_Addr     p_paddr;
    Elf32_Word     p_filesz;
    Elf32_Word     p_memsz;
    Elf32_Word     p_flags;
    Elf32_Word     p_align;
} Elf32_Phdr;
```

그림 17 Program Header

- ◆ p_type
 - 배열요소가 어떤 종류의 세그먼트를 설명하는지와 어떻게 해석되어야 하는지 나타냄
- ◆ p_offset
 - 파일의 처음부터 세그먼트의 첫 바이트가 위치하는 곳까지의 옵셋
- ◆ p_vaddr
 - 세그먼트의 첫 바이트가 위치하게될 가상 메모리의 주소
- ◆ p_paddr
 - 물리적인 주소지정이 필요한 시스템을 위해 예약됨
- ◆ p_filesz
 - object file내에서 세그먼트 바이트의 수를 담고있다
- ◆ p_memsz
 - 세그먼트가 메모리에 적재 되었을때의 바이트 수
- ◆ p_flags
 - 세그먼트와 관계된 플래그
- ◆ p_align
 - 메모리와 파일내에서 세그먼트가 정렬되는 값

적재 가능한 프로세스 세그먼트에서는 `p_vaddr`과 `p_offset`은 페이지의 나머지 값에 일치한다. 메모리와 파일에서 정립되어져야 할 값을 나타낸다. 0과 1은 정렬을 요구하지 않는다. 그렇지 않다면 `p_align`은 양수이고 2의 지수승이고, 그리고 `p_vaddr`은 `p_offset`, `p_align`의 `modulo`와 일치해야 한다.

각 엔트리는 프로세스 세그먼트를 설명하거나 아니면 보충 정보를 주지만 프로세스 이미지에 영향을 주지 않는다. 세그먼트의 엔트리는 순서와 무관하지만 밑에서 분명하게 언급하는 경우는 제외한다. 정의된 형식의 값은 다음과 같다. 다른 값들은 미래에 사용하기 위하여 예약해 놓는다.

| Name | Value |
|------------|------------|
| PT_NULL | 0 |
| PT_LOAD | 1 |
| PT_DYNAMIC | 2 |
| PT_INTERP | 3 |
| PT_NOTE | 4 |
| PT_SHLIB | 5 |
| PT_PHDR | 6 |
| PT_LOPROC | 0x70000000 |
| PT_HIPROC | 0x7fffffff |

그림 18 Segment Type, `p_type`

- ◆ PT_NULL
 - 프로그램 헤더 테이블을 무시할 수 있는 엔트리를 가질 수 있다
- ◆ PT_LOAD
 - `p_filesz`와 `p_memsz`에 표시되는 적재 가능한 세그먼트를 나타냄
- ◆ PT_DYNAMIC
 - 동적 연결 정보를 담고 있다
- ◆ PT_INTERP
 - 인터프리터로서 호풀하기 위한 종료 경로명의 위치와 크기를 담고 있다
- ◆ PT_NOTE
 - 추가 정보의 위치와 크기를 담고 있다
- ◆ PT_SHLIB
 - 정의되지 않음
- ◆ PT_PHDR
 - 파일과 메모리상의 프로그램내에서 프로그램 `pej` 테이블 자신의 위치와 크기를 담고 있다
- ◆ PT_LOPROC/HIPROC
 - 프로세서 의존적인 정보

0x04. Walkthroughs One and Two**Introduction**

- 4장의 주 핵심은 코드의 흐름을 따라가며 다음과 같은 소 주제를 두며 설명하겠다.
 - ◆ Understanding Execution Flow
 - ◆ Tracing Functions
 - ◆ Recovering Hard Coded Password
 - ◆ Finding Vulnerable Functions

```
.text:00401270 ; int __cdecl main(int argc,const char *argv,const char *envp)
.text:00401270 Dst    = byte ptr -80h
.text:00401270 argc   = dword ptr 8
.text:00401270 argv   = dword ptr 0Ch
.text:00401270 envp   = dword ptr 10h
.text:00401270     push    ebp
.text:00401271     mov     ebp, esp
.text:00401273     sub     esp, 80h
.text:00401279     push    offset aReverseEngineer
.text:0040127E     call    sub_401554
.text:00401283     add     esp, 4
.text:00401286     push    offset aPleaseProvideT
.text:0040128B     call    sub_401554
.text:00401290     add     esp, 4
.text:00401293     push    80h      ; Size
.text:00401298     push    0          ; Val
.text:0040129A     lea     eax, [ebp+Dst]
.text:0040129D     push    eax      ; Dst
.text:0040129E     call    _memset
.text:004012A3     add     esp, 0Ch
.text:004012A6     lea     ecx, [ebp+Dst]
.text:004012A9     push    ecx
.text:004012AA     push    offset a127a    ; "%127s"
.text:004012AF     call    _scanf
.text:004012B4     add     esp, 8
.text:004012B7     lea     edx, [ebp+Dst]
.text:004012BA     push    edx      ; Str2
.text:004012BB     call    sub_4011C0
.text:004012C0     add     esp, 4
.text:004012C3     movsx  eax, al
.text:004012C6     test   eax, eax
.text:004012C8     jge    short loc_4012D9
.text:004012CA     push    offset aYouFailed_
.text:004012CF     call    sub_401554
.text:004012D4     add     esp, 4
.text:004012D7     jmp    short loc_4012E6
.text:004012D9 loc_4012D9: ; CODE XREF: _main+58
.text:004012D9     push    offset aYouWon_Goodbye
.text:004012DE     call    sub_401554
.text:004012E3     add     esp, 4
...text:004012E6 loc_4012E6: ; CODE XREF: _main+67
.text:004012E6     mov     eax, 1
.text:004012EB     mov     esp, ebp
.text:004012ED     pop    ebp
.text:004012EE     retn
.text:004012EE main endp
```

그림 19 Binary Reversing Example

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

바이너리를 리버싱 하는 과정에서 가장 중요한 것은 어떠한 바이너리를 어떻게 재구현하는지 가 될것이다.

이제부터 위의 예제 코드를 시작으로 **Step by step** 접근을 한다. 그리고 분석을 시작함으로 인해 메모를 준비하는 것은 간단한 Tip이 될수있다. 인간의 기억력은 한계가 있기 때문에 복잡하고 섬세한 코드들을 일일이 기억하기엔 한계를 느낀다.

우선 예제 코드에서 위의 6줄을 보면 `text:00401270`으로 모두 같은 값을 가지며 우리가 알고 있던 기존의 어썸의 코드들과는 다름을 알 수 있다. 이 부분이 왜 예제 코드에 있느냐는 우선 호출 규약이라 불리는 **calling convention**을 먼저 알 필요가 있다.

참고 : <http://kkamogui.springnote.com/pages/368023>

<http://www.codeengn.com/Archive/100>

Calling Convention 은 함수를 호출하는 규약으로 여기에는 몇 가지 방식이 존재한다.

- ◆ **Stdcall(pascal)**방식 : 스택에 파라미터를 역순으로 삽입하고 함수를 호출하며 스택의 정리 작업을 호출된 함수에서 수행한다.
- ◆ **Cdecl** 방식 : 스택에 파라미터를 넣는 방식은 **Stdcall**과 같으나 스택의 정리 작업을 호출한 함수에서 수행한다. (C 언어에서 주로 사용)
- ◆ **Fastcall** 방식 : 몇 개의 파라미터는 레지스터를 통해서 넘기고 나머지는 스택을 사용하는 방식

Stdcall 방식은 **Callee**(호출된 함수, 수신자)에서 스택 정리를 함으로 **Caller**(호출하는 함수)와 **Callee** 모두 파라미터의 개수를 알고 있어야 정상적인 처리가 가능하다. 반면에 **cdecl** 방식은 **Caller**에서 스택 정리를 함으로 **Callee**는 파라미터의 개수를 정확히 몰라도 된다.

```
int DoSomething( int a, int b )
{
    int c;
    c = a+b;
    return c;
/* 어셈블리어로 변경된 코드
push ebp
mov ebp,esp
push ecx
mov eax,[ebp+08h]
add eax,[ebp+0Ch]
mov [ebp-04h],eax
mov eax,[ebp-04h]
mov esp,ebp
pop ebp
retn
*/
}
int main(int argc, char* argv[])
{
    DoSomething( 1, 2 );
/* 어셈블리어로 변경된 코드
push ebp
mov ebp,esp
push 00000002h
push 00000001h
call SUB_L00401000
add esp,00000008h <== 스택을 정리하는 부분
pop ebp
retn
*/
}
```

그림 20 C언어 -> 어셈블리어

위의 내용은 컴파일러의 디버깅모드시 간단히 보여지는 화면입니다.

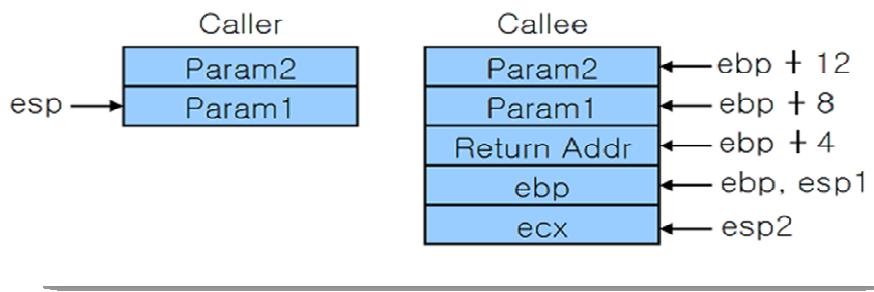


그림 21 Caller, Callee 스택상태

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

위의 C코드와 어셈 코드에서 Caller 와 Callee의 스택 상태를 나타낸 것이다. 여기서 주의 해서 볼 것은 스택에 대한 access가 ESP가 아닌 EBP를 기준으로 움직임을 확인 할 수 있는데 그 이유는 ESP 레지스터는 코드 중간 중간에 호출될 수 있기 때문에 그 위치가 가변적이다. 그래서 EBP를 기준으로 삼아서 스택의 위치를 indexing하게 되면 스택의 TOP가 바뀌더라도 EBP는 항상 초기 스택의 TOP 위치를 가지게 된다. (초기 스택의 TOP는 EBP=ESP)

예제 코드를 보다가 갑자기 Stack과 Caller, Callee, ESP, EBP 등을 언급한 이유는 , 호출하고 호출되어지는 함수의 호출 관계에서 Callee의 스택을 다시 Caller의 스택으로 되돌려 줘야하는데 이때 스택의 TOP를 저장하고 복원하는 작업을 Prologue 라 한다. 즉 위의 예제 코드에서는 main함수가 돌아가야 하는 위치를 표시해 주고 있는 것이다.

00401270 ~ 73 은 스택에 ebp를 저장하고(push ebp) esp에 ebp값을 가르키도록 위치하고 (mov ebp, esp) 스택에서 esp를 0x80만큼 이동시킨다. (sub esp, 80h)

예제 코드를 아무런 어셈 지식없이 코드를 읽다보면 offset 뒤에 우리가 읽고 이해할 수 있는 문자열들이 보인다. aReverseEnginee, aPleaseProviedT, a127s ;"%127s", aYouFailed_, aYouWon_Goodbye가 그것이다. 그런데 이것만 봐도 예제 코드가 먼가 입력을 받아서 맞으면 YouWon 을 틀리면 Failed를 우리에게 보여준다는 것을 가볍게 짐작할 수 있다. 이정도를 알았다면 리버싱에 감각이 전혀 없는 것은 아니라고 생각한다. 머리를 너무 비우고 코드를 읽었다면 한번 더 읽어보기를 권장한다.

이제 본격적으로 어셈 코드들의 흐름에 몸을 실어보자.

```

.text:00401279      push    offset aReverseEnginee
.text:0040127E      call    sub_401554
.text:00401283      add     esp, 4
.text:00401286      push    offset aPleaseProvideT
.text:0040128B      call    sub_401554

```

우리가 익숙한 Push가 나왔다. 스택에 offset 들을 저장하라는 것이다. 다음에 push가 나오면 가볍게 또 원가 스택에 집어 넣는구나 라고 생각해 주면 된다. 그리고 앞에서 잠시 언급했던 call...은 무었인가를 호출한다. 그런데 call이 2개가 있는데 모두 sub_401554를 가리키고 있다. 찾아가보도록 한다.

```
.text:00401554 ; int printf(const char *,...)
.text:00401554_printf      proc near ; CODE XREF: sub_401000+65
.text:00401554; sub_401000+C0
```

쉽게 알아볼수있는 문자열이 존재한다. Printf 누구나 본적이 있는 그 함수... sub_401554는 무엇인가 출력하기 위해서 호출되어지는 것이다. 0x04장은 좁고 깊은 지식보다는 폭넓고 얕은 깊이로 흘러가는 chapter이기 때문에 가볍게 출력을 위해서 호출되는 sub_401554라고 생각해 두기 바란다.

```
.text:00401290    add    esp, 4
.text:00401293    push   80h    ; Size
.text:00401298    push   0       ; Val
.text:0040129A    lea    eax, [ebp+Dst]
.text:0040129D    push   eax    ; Dst
.text:0040129E    call   _memset
.text:004012A3    add    esp, 0Ch
.text:004012A6    lea    ecx, [ebp+Dst]
.text:004012A9    push   ecx
.text:004012AA    push   offset al27s    ; "%127s"
.text:004012AF    call   _scanf
```

위 어셈 코드의 포인트는 호출되는 `_memset`과 `_scanf`이다. 여기서는 버퍼를 채우기 위해서 `memset`을 호출하고 버퍼를 읽기 위해서 `sacnf`를 사용한다. `Memset`이 채우는 버퍼에 대한 것은 `call _memset` 앞단의 `push` 명령어들을 보면 알 수 있다. (`0x80=128 OR 0x00`에서 `Dst` 스택 버퍼의 바이트들, `NULL`)

지금까지는 어셈 명령어들 위주로 코드의 흐름을 따라 내려 왔다면 이제 남은 것은 이 코드들이 무엇을 하는지에 대한 접근을 시작한다. 즉 `Reversing What the Binary Does` 으로 들어가게된다.

```
.text:004012A9    push    ecx
.text:004012AA    push    offset a127s    ; "%127s"
.text:004012AF    call    _scanf
.text:004012B4    add     esp, 8
.text:004012B7    lea     edx, [ebp+Dst]
.text:004012BA    push    edx    ; Str2
.text:004012BB    call    sub_4011C0
.text:004012C0    add     esp, 4
.text:004012C3    movsx  eax, al
.text:004012C6    test   eax, eax
.text:004012C8    jge    short loc_4012D9
```

앞에서 잠시 언급했던 `scanf`가 다시 나왔지만 위의 코드에서 주인공은 다시 만나서 반가운 `scanf`가 아니다. `Scanf`의 리턴값을 처리하는 부분이 나온다. 즉 스택과 호출된 `sub_4011C0`를 비교한다. 드디어 나왔다. 비.교. 정확한 입력값에 대해서 YouWon_Goodbye를 출력하는 예제라고 도입부에 설명해 두었던 것을 기억하면 비.교.라는 단어가 왜 중요한지 충분히 인식했을 것이다.

Call `sub_4011C0` 이전에서 `scanf`의 리턴값을 위해 `esp`를 8만큼 증가시키고 `lea edx, [ebp+Dst]`를 통해서 `edx`에 [] value를 저장하고 `push edx`를 통해서 스택에 `edx`값을 저장하는 과정이 진행된다. 그리고 아직 알 수 없는 의문의 호출 `sub_4011C0`의 과정이 진행되고 그 결과값이 `esp`가 4만큼 증가된 공간에 할당된다.

이번에는 앞서 언급했던 `sub_4011C0` SubRoutine의 흐름을 따라가 본다.

```
.text:004011C0 ; int __cdecl input_process(char *Str2)
.text:004011C0 input_process proc near ; CODE XREF: _main+4B
.text:004011C0 Dst    - byte ptr -80h
.text:004011C0 var_7F- byte ptr -7Fh
.text:004011C0 var_7E- byte ptr -7Eh
.text:004011C0 var_7D- byte ptr -7Dh
.text:004011C0 var_7C- byte ptr -7Ch
.text:004011C0 var_7B- byte ptr -7Bh
.text:004011C0 var_7A- byte ptr -7Ah
.text:004011C0 var_79- byte ptr -79h
.text:004011C0 var_78- byte ptr -78h
.text:004011C0 var_77- byte ptr -77h
.text:004011C0 var_76- byte ptr -76h
.text:004011C0 var_75- byte ptr -75h
.text:004011C0 var_74- byte ptr -74h
.text:004011C0 var_73- byte ptr -73h
.text:004011C0 var_72- byte ptr -72h
.text:004011C0 var_71- byte ptr -71h
.text:004011C0 var_70- byte ptr -70h
```

```
.text:004011C0 Str2  = dword ptr 8
.text:004011C0          push    ebp
.text:004011C1          mov     ebp, esp
.text:004011C3          sub     esp, 80h
.text:004011C9          push    80h      ; Size
.text:004011CE          push    0         ; Val
.text:004011D0          lea     eax, [ebp+Dst]
.text:004011D3          push    eax      ; Dst
.text:004011D4          call    _memset
.text:004011D9          add     esp, 0Ch
.text:004011DC          mov     [ebp+var_70], 0
.text:004011E0          mov     [ebp+var_75], 73h
.text:004011E4          mov     [ebp+Dst], 74h
.text:004011E8          mov     [ebp+var_76], 73h
.text:004011EC          mov     [ebp+var_7F], 68h
.text:004011F0          mov     [ebp+var_7A], 6Dh
.text:004011F4          mov     [ebp+var_7C], 69h
.text:004011F8          mov     [ebp+var_7B], 73h
.text:004011FC          mov     [ebp+var_71], 64h
.text:00401200          mov     [ebp+var_74], 77h
.text:00401204          mov     [ebp+var_7E], 69h
.text:00401208          mov     [ebp+var_7D], 73h
.text:0040120C          mov     [ebp+var_78], 70h
.text:00401210          mov     [ebp+var_73], 6Fh
.text:00401214          mov     [ebp+var_72], 72h
.text:00401218          mov     [ebp+var_79], 79h
.text:0040121C          mov     [ebp+var_77], 61h
.text:00401220          mov     ecx, [ebp+Str2]
.text:00401223          push    ecx      ; Str2
.text:00401224          lea     edx, [ebp+Dst]
.text:00401227          push    edx      ; Str1
.text:00401228          call    _strcmp
.text:0040122D          add     esp, 8
.text:00401230          test   eax, eax
.text:00401232          jz     short loc_401247
.text:00401234          push    offset aInvalidPassword ;
"\n***** INVALID PASSWORD *****\n"
.text:00401239          call    printf
.text:0040123E          add     esp, 4
.text:00401241          or     al, 0FFh
.text:00401243          jmp    short loc_40125D
```

```
.text:00401245      jmp     short loc_40125D
.text:00401247 loc_401247: ; CODE XREF: input_process+72
.text:00401247          mov     eax, [ebp+Str2]
.text:0040124A          push    eax
.text:0040124B          push    offset aSIIsCorrect_ ; "%s is correct.\n\n"
.text:00401250          call    printf
.text:00401255          add    esp, 8
.text:00401258          call    sub_401000
.text:0040125D loc_40125D: ; CODE XREF: input_process+83
.text:0040125D; input_process+85
.text:0040125D          mov     esp, ebp
.text:0040125F          pop    ebp
.text:00401260          retn
.text:00401260 input_process      endp
```

위의 예제 코드를 위에서 아래로 가볍게 훑어보도록 하자. 그러면 지금까지 보지 못한 어AAP 코드들이 몇 가지가 보임을 알 수 있다. (call _strcmp, call sub_401000)

_strcmp 직관적으로 봤을 때 떠오르는 단어가 있다면 절반은 먹고 들어간다고 할 수 있다. 이 함수는 바로..... stringcompare를 떠올렸다면 센스가 충만한 것이다. 우리가 최초에 봤던 예제 코드에서 언급했듯이 입력받은 값을 비교해서 YouWon_Goodbye을 얻어내는 것이 목적이었다. 즉 우리에게는 비교하는 함수가 필요했던 것이다. 그것도 문자열을 비교하는!! 예제 코드에서 push ecx 와 push edx를 주석으로 Str2, Str1으로 설명해놓고 있으며 이 두 스택의 값을 _strcmp 함수를 이용해서 비교한다. _strcmp 리턴값이 처리되는 것은 바로 아래 부분 jz 어AAP 명령어에서 처리된다. Jz는 비교값이 0 일때 지정된 장소로 점프하는 어AAP 명령어로서 _strcmp의 리턴값에 대한 test eax, eax의 결과가 0이 된다면 (두 문자열 Str2, Str1이 서로 다르다면), 지정된 장소 short loc_401247으로 점프를 하게 된다. 만약 0이 아닐 경우는 jz 구문 아래로 차례대로 진행이 된다.

short loc_401247에 잠시 다녀오면 좋겠지만 어차피 차례로 코드의 흐름을 따라간다면 다시 살펴볼 기회가 있기 때문에 여기서는 잠시 언급만하고 지나쳐 가겠다. 하지만 그곳에 우리가 원하는 최종 목적이 있음을 꼭 기억해 두어야 한다.

Call printf가 또 나왔다. 무슨 뜻일까? 당연히 위의 INVALID PASSWORD 부분을 출력하라는 것이다. 여기까지 왔으면 척이면 착으로 알고 있을 것이라 믿어 의심치 않는다. 왜냐하면 지금까지 call에 의해 호출될 함수의 인수들은 바로 앞에서 모두 정의되었기 때문이다. 자, 이제 우리는 굉장히 익숙한 'or' 명령어를 마주하게 되었다. 여기서 or 연산자는 매칭되는 비트들 중에서 어느 하나 또는 둘 모두가 1이면 연산 결과를 1로 설정하는 명령어이다.

그리고 이하 jmp 구문이 2개가 나오는데 모두 한곳으로 점프하도록 가르키고 있다. Short loc_40125D를 가르키고 있는데 여기를 잠시 살펴보면,

```
.text:0040125D loc_40125D: ; CODE XREF: input_process+83
.text:0040125D    input_process+85
.text:0040125D        mov     esp, ebp
.text:0040125E        pop    ebp
.text:00401260        retn
.text:00401260 input_process      endp
```

여기서 중요한 것은 mov esp, ebp이다. 최초 ebp와 같은 위치에서 시작해서 함수마다 불려다니면서 일하던 esp가 다시 ebp와 같은 위치로 돌아왔다. 그것은 최초 예제의 목적을 성공적으로 수행했다는 것이다. 우리의 목적은 Short loc_40125D를 찾아가면 되는 것이다.

이제 다시 앞에서 잠시 언급했던 jmp short loc_401247를 살펴보자.

```
.text:00401247 loc_401247: ; CODE XREF: input_process+72
.text:00401247        mov     eax, [ebp+Str2]
.text:0040124A        push   eax
.text:0040124B        push   offset aSIIsCorrect_ ; "%s is correct.\n\n"
.text:00401250        call   printf
.text:00401255        add    esp, 8
.text:00401258        call   sub_401000
```

위에 매우 짧은 예제 코드가 언급되어 있는데 이 짧은 명령어들 사이에 또 서브루틴을 호출하는 call sub_401000이 있다. 이 다음은 무엇을 하겠는데 바로 call sub_401000를 살펴보도록 하겠다. 그런데 책에서 설명하고 있는 서브루틴 sub_401000의 코드가 매우 길다는 것을 알 수 있다.

일단 prologue인 00401000에서부터 SecondCheck이다. 즉 call sub_401000 이전 과정에서 문자열을 비교했는데 틀려서 이 서브루틴으로 들어왔다는 것이다. 더더욱 슬픈 것은 본 서브루틴이 지금까지 모든 예제코드나 서브루틴보다 길다는 것이다. 일단 시작했으니 짧은 지식을 동원해서라도 코드의 흐름을 파고 들어가보자.

코드의 첫 시작은 앞에서 보아온 예제들과 동일하다 스택에 push 명령을 이용해서 값을 저장하고 있다. 그러다 00401015 ~ 1F 구간에서 익숙지 않은 명령어가 보인다.

Rep movsd

Movsw

Movsb

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

Rep는 스트링 관련 명령어들과 함께 사용된다. (rep, repz, repe, repnz, repne등이 있다.) Movsb, movsw, movsd는 스트링 관련 명령어들로서 movs는 mov string을 의미하며 뒤에 붙는 b,w,d는 자료형의 크기를 나타낸다. (1,2,4 바이트를 나타냄) 즉, 한 메모리 주소에서 다른 곳으로 값을 복사한다.

명령어 설명에서 다시 코드의 흐름으로 넘어가면 _memset이 두 번 call되는 것을 볼 수 있는데 메모리에 eax와 ecx값들이 저장되면서 스택의 esp값이 +12씩 두 번 증가하고 있다.
(add esp, 0ch)

본 서브루틴에서 점프로 올 수 있는 위치가 8개 지점이 표시되어 있다. 우선 0040104C부터 코드를 따라가 보자. 이 루틴에는 call 되는 함수가 printf, scanf, _strncat, _strcmp 4개의 함수가 있음이 단순히 코드만 따라가면서 확인 할 수 있다. 여기서 생소한 함수인 strncat에 대해 알아보고 계속해서 코드를 따라가 보겠다.

Strncat는 길이를 지정하여 두 개의 문자열을 합치는 기능을 수행하는 함수이다.

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

0x05. Debugging

Introduction

5장은 디버깅에 대한 기본과 IDA를 이용한 디버깅 기능들에 대해서 설명해 주고 있다. 디버깅은 소프트웨어에서 버그의 위치를 찾는 일련의 모든 행동들을 말한다. 우리가 다루는 IDA 역시 디버거로서 다른 프로그램들을 실행하고 그 실행을 모니터링 할 수 있다.

디버거는 2가지 타입으로 나뉠 수 있는데 **user mode** 와 **kernel mode**이다. 유저 모드의 경우 디버거가 프로세스 단위로 동작을 한다. 그리고 메모리에 대한 접근이 제한되기 때문에 커널 메모리에 접근하여 디버깅 해야하는 커널 코드를 디버깅 하는데 불편함을 가지고 있다. 커널 모드의 경우 커널 레벨에서 코딩되어 있는 드라이버나 모듈, 시스템 레벨 등에서 코드를 확인할 때 유용하다. 유저 모드의 디버거로는 IDA pro, OllyDbg 등이 있으며 커널 모드 디버거로는 Windbg, SoftICE가 있다.

디버깅하면 가장 먼저 떠오르는 것은 **printf?** 이라면 곤란해진다. 물론 괜찮은 방법 중에 하나지만 무한 삽질과 인내를 요구하기에 **breakpoint**에 대해서 잠시 생각해보고 넘어가자.

- ◆ Breakpoint

BP는 프로그램 내에서 우리가 원하는 위치에서 프로그램의 실행을 멈추게 한다.

BP를 사용하게 되면 프로그램의 실행이 멈추게 되고 제어권이 디버거로 넘어가게 된다.

그리고 Breakpoint는 일반적으로 2 가지 타입으로 나뉜다.

- ◆ Hardware breakpoint

Hardware Breakpoint는 CPU에 있는 Debug Register를 이용하여 브레이크 포인트를 거는 방법이다. Software Breakpoint는 실행을 잡아낼 수 밖에 없지만, Hardware Breakpoint는 실행/읽기/쓰기를 잡아내는 것이 가능하다. 하지만 Hardware Breakpoint를 사용할 때, Breakpoint 걸 주소를 지정할 수 있는 공간은 DR0, DR1, DR2, DR3 이렇게 4개 밖에 없다.

DR4,5는 사용하지 않고 DR7은 Debugger Control register(IA-32 process 기준)이며 2 가지 레벨로 나뉜다. (Local(DR0,2,4,6), Global(DR1,3,5,7) level) local bit가 활성화 된 DR은 새로운 task에서 원치 않는 BP 설정을 피하기 위해서 모든 task switch 상황시 프로세서에 의해서 자동으로 리셋된다. Global bit가 활성화된 DR은 task switch에 의해서 리셋되지 않는다. 그러므로 이러한 DR7은 모든 task에서 Hardware breakpoint 활성화 유무와 형태에 대한 정보를 나타낼 수 있는 것이다.

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

Breakpoint 가 trigger 되면 16 ~ 23 bit 가 정의되고 각 BP 는 2 개의 bit 를 할당 받아 execution(00b), data write(01b), data read or write(11b) 를 표시한다. 10b 는 IO read or write 를 의미하지만 hardware 는 이를 지원하지 않는다. 24 ~ 31 bit 는 BP 에 의해서 보여지는 메모리 영역의 크기를 나타낸다. – 이 부분을 더 자세히 알고 싶다면 IA32 레퍼런스를 참조하기를 바란다. 여기서 자세한 언급은 피하겠다.... 복잡하니까 이런 것이 있다는 정도만 기억하자.

[IA- 32 : Intel Architecture 32]

◆ Software breakpoint

하드웨어 BP 와는 다르게 디버거 레지스터를 사용하지 않고 디버거가 직접 관리하는 BP 이다. Software BP 의 동작 방식은 설정한 BP 의 명령어의 첫 바이트를 인터럽트 명령어 INT 3 (0xcc)로 바꾸고 원래 명령을 다른 장소에 저장한다. 그래서 해당 명령이 실행이 되면 INT3 이 실행되면서 IDT 에 지정되어 있는 Interrupt3 Handler 가 호출되어 OS 는 이것을 적절한 처리하고 디버거에 현재 상황(CPU 의 레지스터 등)을 알려주고 디버거는 이것을 가지고 처리한다.

[IDT : Interrupt Descriptor Table]

디버깅에서 Software breakpoint 는 hardware BP 보다 자주 사용된다. 그 가장 큰 이유는 단 4 개의 레지스터를 사용하여 4 개의 BP 를 설정하는 hardware BP 와는 달리 제한이 없기 때문이다.

Hardware BP 는 메모리 영역에서 설정할 수 있어서 메모리에 대한 접근을 breaking 해서 테이블 사용이나 메모리 충돌을 확인해 볼 수 있다.

참고 : <http://zesrever.xstone.org/10>

◆ Single stepping

- Single stepping 은 디버거에서 명령어를 하나씩 실행해가는 과정을 말한다. 이 방법으로 프로그램 전체를 확인하는 것은 매우 힘들다. 이 방법은 코드의 일부를 자세히 살펴 볼 때 활용된다.

◆ Watches

- 소스 레벨 디버깅에서 변수들은 추상적인 이름의 위치를 가진다. 그리고 어셈블리에서 변수들은 일반적으로 메모리 위치들이다. 컴파일러는 변수들을 레지스터로 최적화시킬 수 있다. Watches 는 변수를 보여주거나 유용한 표현식을 보여주는 기능이다.

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

◆ xceptions : Exceptions

- 프로그래머가 에러를 찾기 위해서 사용하는 기능이라 할 수 있다. 디버거는 Exceptions에서 정지를 시킬 수도 있지만 어플리케이션으로 넘길 수도 있다. 즉 Exceptions이 의미하는 것이 무조건 error는 아닌 것이다

Windows에서 0xc00000005는 Access Violation으로 프로세스가 잘못 맵핑된 address에 접근을 했다는 뜻이다

Exception C0000005 (ACCESS_VIOLATION reading [41414141])

위와 같은 결과는 프로세스가 0x41414141 address에 잘못 맵핑되었다는 뜻이다.

◆ Tracing

Tracing은 방식에 따라 프로그램을 실행하는 과정이나 정보를 기록하는 과정을 추적하는 기능이다. Instruction tracing으로 명령어나 레지스터 값 등을 하나하나 추적해 갈 수도 있지만 이러한 작업은 속도가 매우 느리다. 그래서 function tracing을 이용해서 BP를 설정하여 추적하거나 call이 나올 때까지 single stepping 할 수 있다. 즉, 특정 함수가 호출될 때, 프로그램 실행을 멈추고 디버거는 인자값아니 레지스터 값을 기록하고 Exception이 처리된다. 그리고 이 두 방식 외에도 basic block tracing이 있다.

IDA Pro에는 debugger가 포함되어 있으며 plug-in 기능을 제공한다. 그리고 디버거는 로컬에서 뿐만 아니라 네트워크로 연결된 원격에서도 동작할 수 있다. IDA Pro가 지원하는 디버깅 환경은 Win 32 Local/Remote, Win 64 Remote, Linux Remote(x86 only), OSX Remote(x86 only), WinCE Remote (ARM only) 등이다.

◆ Debugger Setup Options

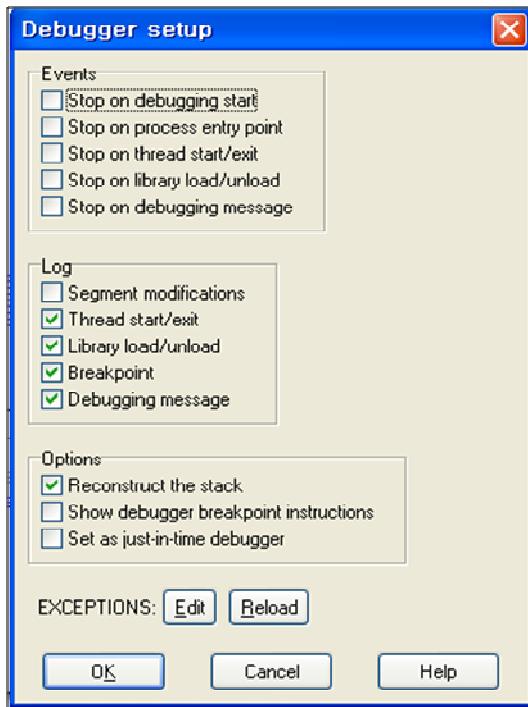


그림 22 Debugger Setup Option

◆ Debugger Application Setup

- 특정 어플리케이션을 커맨드 모드에서 실행하는 것과 같이 실행할 수 있는 기능 (Option, Port 등 설정 가능)

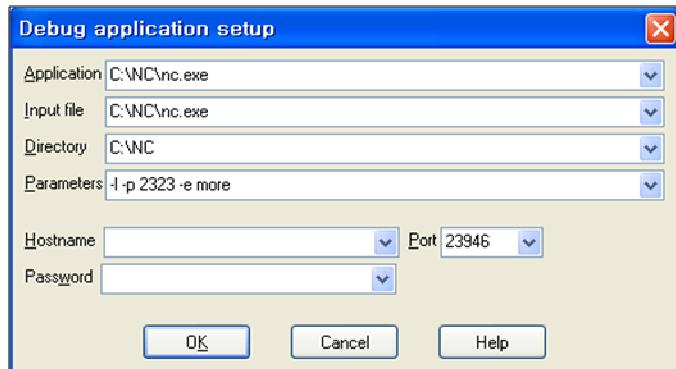


그림 23 Debugger Application Setup

◆ Debugger Hotkey List

- F9 : Debugger 시작
- F2 : BP 설정 및 해제
- F6 : 다음 윈도우로 이동
- F7 : 단계별 코드 진행, call 문 포함 (Ollydbg 와 같음)
- F8 : 함수 별 코드 진행, call 문 바로 처리 (Ollydbg 와 같음)

◆ Conditional Breakpoint

- Conditional Breakpoint에서는 software BP뿐 아니라 hardware BP까지 설정이 가능하며 condition box에서 BP가 처리되는 조건을 만들 수 있다.
`Esi==0xc8 || esi == 0xcc` 와 같은 조건을 주면 해당 조건을 만족할 때 아래의 action에 해당하는 동작을 취하게 된다.

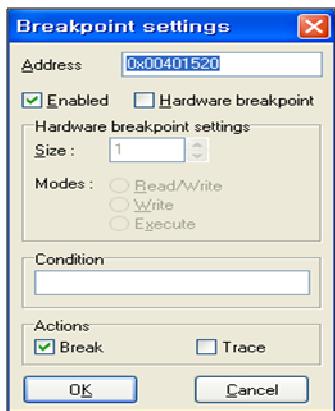


그림 24 BreakPoint Setting

◆ Exception handling

- 스택에서 BP에 의해 메모리 등이 충돌하게 되면 Exception handling이 나타나게 되고 여기서 우리는 stop할 것인지 pass할 것인지 설정해 줄 수 있다.

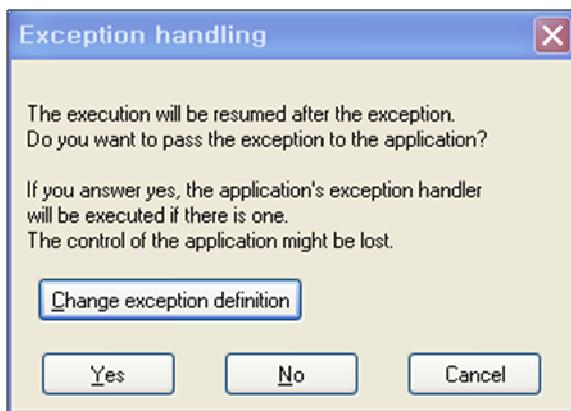


그림 25 Exception Handling

| | | |
|---------------------------------|--|------------------------------|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|------------------------------|

0x06. Anti- Reversing

MASM

■ EDIT PLUS 를 이용한 MASM32 컴파일 환경 구축

MASM32 어셈블러를 이용한 Anti- Reversing Code 구현에 앞서 간단히 MASM32 이란 무엇이며 어떻게 환경을 구축하여 테스트를 하는지에 대한 설명을 할 것이다.

MASM32 이란 마이크로소프트사의 어셈블러 툴이다 그외 여러가지 어셈블러들이 존재하지만 일반적으로 API 를 쓰기 위한 어셈블러로 MASM 을 이용하게 된다. MASM 은 어디서 구할수 있으며 환경설정은 어떻게 할것인가? 다음 사이트에서 MASM 을 구할수 있다.

<http://www.masm32.com/masmdl.htm>

현재 최신 버전은 9 버전으로 파일명은 m32v9r.zip 으로 존재한다. 참고로 필자의 Visual Studio 의 버전은 6 을 쓰며 또한 필요한 툴은 EDIT PLUS 가 있어야 한다 각자의 툴은 알아서 구하길 바란다. 먼저 MSAM 을 설치해 본다.

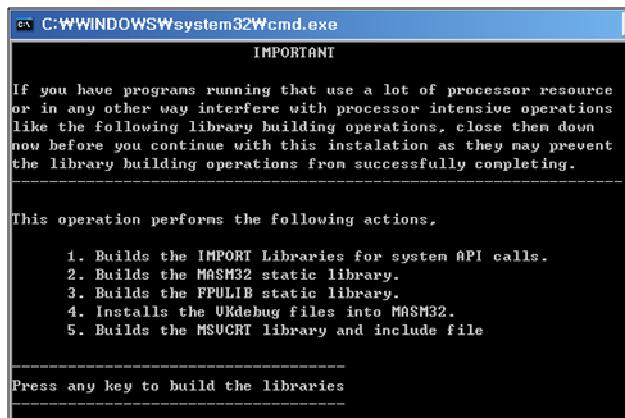


그림 26 MASM Install

간단히 Enter 키를 누름으로써 설치를 할 수 있다.

다음은 EDIT- PLUS 에서 설정이다. 도구- > 기본설정- > 사용자도구에서 그룹과 도구 항목을 설정 할 수 있다.

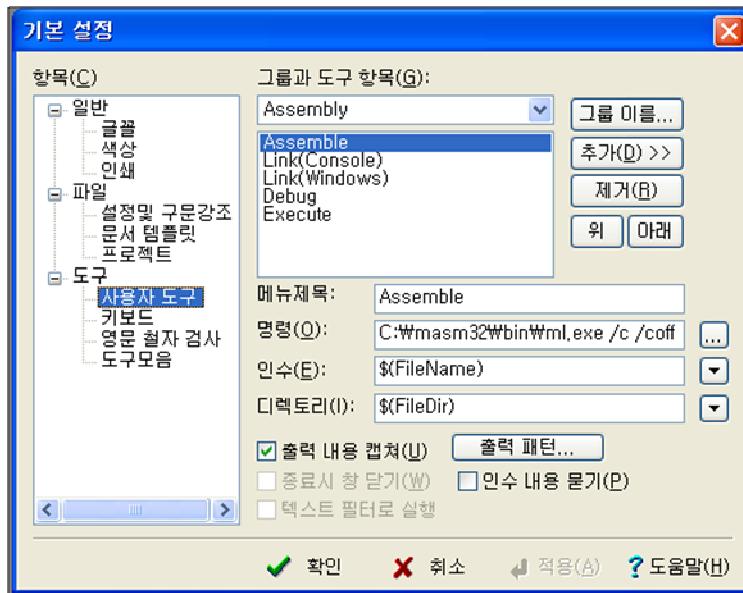


그림 27 EditPlus 컴파일 환경설정

아래는 이와 같은 설정을 나열 해 보여주고 있다.

Edit- Plus Assembly 설정 (MASM32)

1. Assemble

명령 : C:\masm32\bin\ml.exe /c /coff /Zi, 인수 : \${FileName}, 디렉토리 : \${FileDir}, 출력 내용 캡쳐 : 체크

2. Link (Console)

명령 : C:\masm32\bin\link.exe /SUBSYSTEM:CONSOLE /DEBUG, 인수 : \${FileNameNoExt}.obj, 디렉토리 : \${FileDir}

출력 내용 캡쳐 : 체크

3. Link (Windows)

명령 : C:\masm32\bin\link.exe /SUBSYSTEM:WINDOWS /DEBUG, 인수 : \${FileNameNoExt}.obj, 디렉토리 : \${FileDir}

출력 내용 캡쳐 : 체크

4. Debug

명령: C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin\MSDEV.EXE, 인수 : \${FileNameNoExt}.exe

디렉토리 : \${FileDir}

5. Execute

또한 EditPlus 에서는 MASM 문법에 대한 구문강조 파일이 존재한다. 해당 파일은 아래 사이트에서 받을수 있다. 아래 Site 에서 다운 받은뒤 EDIT- PLUS 가 설치된 폴더에 복사하면 된다 다음은 해당 구문강조에 대한 설정을 하는 장면이다.

- 다운 : <http://www.editplus.com/dn.cgi?asm2.rar>

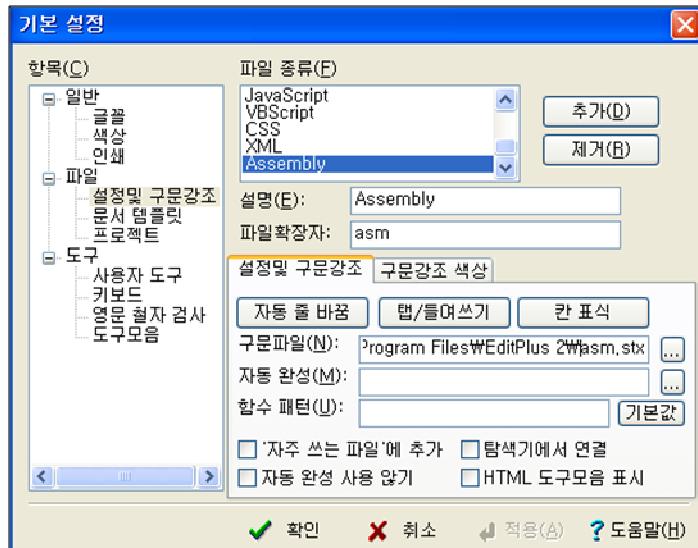


그림 28 MASM STX 설정

해당 설정을 끝 마쳤다면 이제 실질적인 ASM 코드를 작성 하는 시간이다. 처음에는 간단한 문법 설명을 위하여 IsDebuggerPresent 라는 안티 디버깅 코드를 대상 으로 설명 하고 그 외 안티 코드들을 설명할 때 덧붙이는 형식으로 진행한다. 이제 아래 그림처럼 새 파일을 하나 생성 하여 본다.

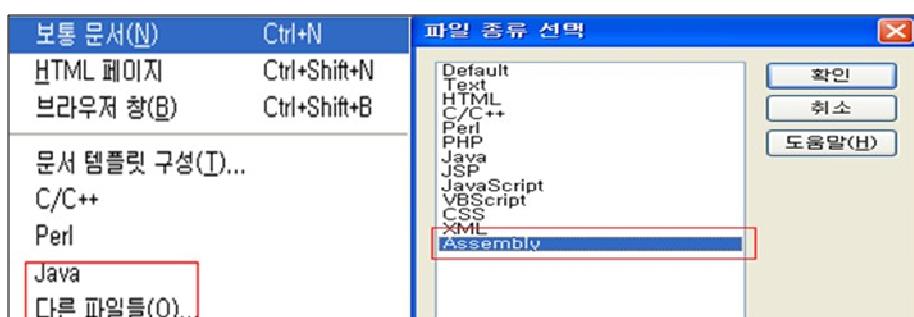


그림 29 파일생성 예제

■ Anti-Reverse

리버서드를 아주 괴롭히는게 이 안티리버싱 기법이다.. 언팩을 하든 프로그램을 분석하든 이 안티리버싱 코드가 있으면 어떻게든 리버서들을 방해하고 혼란을 가중시키기 위하여 여러가지 방법을 동원하여 괴롭힙니다. 물론 방어자 입장에서는 자신의 코드가 쉽게 분석 당하지 않기 위하여 더없이 고마운 기법이긴 하지만 말입니다.

■ Optional Header

대부분의 기초적인 디버거 탐지 기법에는 PEB 의 BeingDebugged 값을 체크를 하는 기법을 가지고 있습니다. PEB(Process Environment Block)의 위치는 TEB(Thread Environment Block)에서 0x30 만큼 떨어진 곳에 존재 합니다. IsDebuggerPresent 는 유저모드의 디버거가 프로세서를 디버깅하고 있는지 flag 값을 체크 하여 확인합니다. 여기서 간단히 PEB 과 TEB 에 대해서 알아 보겠습니다. PEB 은 유저레벨 프로세서의 대한 추가적인 정보를 가지고 있는 구조체이고 TEB 은 쓰레드에 대한 정보를 가지고 있다고 말할 수 있습니다 이 구조체를 어떻게 구동되며 핸들링 하는지는 얘기가 길어 지므로 생략합니다. 여기서는 PEB 의 구조체 값 중 BeingDebugged 값을 체크하여 디버깅 유무를 판별 한다고 생각 하시면 될까 같습니다. 그 외에도 PEB 구조체 값을 이용하여 판별 할수 있습니다. 그건 추후에 나오는 기법에 대하여 언급 하겠습니다. 아래의 코드를 바탕으로 기본적이 MASM 문법과 IsDebuggerPresent 의 기능을 살펴 보겠습니다.

◆ IsDebuggerPresent Code

```
.386
.model flat, stdcall
option casemap :none      ; case sensitive
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.data
DbgNotFoundTitle db "Debugger status:",0h
DbgFoundTitle db "Debugger status:",0h
DbgNotFoundText db "Debugger not found!",0h
DbgFoundText db "Debugger found!",0h
```

```
.code
start:

    CALL IsDebuggerPresent

    CMP EAX,1
    JE @DebuggerDetected

    PUSH 40h
    PUSH offset DbgNotFoundTitle
    PUSH offset DbgNotFoundText
    PUSH 0
    CALL MessageBox

    JMP @exit
@DebuggerDetected:

    PUSH 30h
    PUSH offset DbgFoundTitle
    PUSH offset DbgFoundText
    PUSH 0
    CALL MessageBox

    @exit:

    PUSH 0
    CALL ExitProcess

end start
```

IsDebuggerPresent 소스를 **EDIT- PLUS stx**에 적용

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

.386 은 어떠한 인스트럭션 를 가질 것인가에 대한 정의 정도가 되겠습니다 위와 같이 .386 이 되어 있다면 386 의 인스트럭션을 가질 것이며 .486 .586 등이 존재 합니다 이 와 같은 정의는 각각의 사양에 따른 아키텍쳐가 조금씩 차이가 나기 때문입니다.(가령 386 에서는 32 비트 레지스터와 32 비트 주소 버스 및 외부 데이터 버스의 특징을 가지고 있으며 펜티엄 계열 에서는 실행 속도를 향상시킨 새로운 마이크로 구조 설계를 기반 하는 차이입니다 이에 대한 설명은 대부분의 어셈블리어 책에 초반부에 설명이 되어 있습니다) 다음은 .model 지시자은 메모리 모델에 해당하는 지시자 입니다. 원도우는 항상 flat 모델을 쓰고 있으므로 flat 이라 지정해주며 뒤에 stdll 은 콜링컨베션중 스탠다드 콜에 해당하는 정의입니다. Flat 의 간단한 정의는 no segment, 32bit address,protected mode only 라고 정의되어 있습니다. Option 에 대해서는 잘 모르겠군요. 다음으로는 include 입니다 이건 조금이라도 프로그래밍을 하신분들이 라면 아실꺼라 생각하고 생략합니다. .data 지시자는 데이터 세그먼트를 지정하는 지시자 입니다. DB(Define Byte) 라고 해서 원바이트 형으로 데이터를 정의 하겠다는 뜻입니다. C 에서는 char 형에 해당합니다. .code 지시자는 이제부터 코드 세그먼트를 지시하고 있다는 뜻입니다. 그뒤 start 는 일종의 코드 엔트리를 가리키는 레이블입니다. 코드 중에 @exit: 라고 정의 된 부분이 있고 그것을 호출하는 부분을 쉽게 정의하기 위한 형태라고 생각 하시면 되겠습니다. 다른 부분은 일반적인 어셈 코드와 다를 바가 없습니다.

이제는 PEB 구조체와 TEB 구조체의 값을 간단히 살펴보겠습니다. 주로 windbg를 이용하여 확인 하지만 간단히 값을 확인 할때는 likekd를 이용하면 편합니다. Livekd는 아래 사이트에서 다운로드 받을수 있습니다. 이 툴의 동작 방식은 커널 메모리 덤프를 실시간으로 떠서 동작합니다. 단지 메모리 덤프를 통하여 커널을 분석할 때 유용한 툴입니다. 그전에 Debugging Tools for Widows가 먼저 설치 되어 있어야 합니다. 참고로 dt명령어는 커널 구조체의 값을 확인할 때 쓰이는 명령어 입니다

➤ 참고 : <http://www.sysinternals.com/utilities/livekd.html>

```
+0x200 SystemDefaultActivationContextData : Ptr32 Void
+0x204 SystemAssemblyStorageMap : Ptr32 Void
+0x208 MinimumStackCommit : Uint4B
:
kd> dt _TEB
+0x000 NtTib : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId : _CLIENT_ID
+0x028 ActiveRpcHandle : Ptr32 Void
+0x02c ThreadLocalStoragePointer : Ptr32 Void
+0x030 ProcessEnvironmentBlock : Ptr32 _PEB
+0x034 LastErrorValue : Uint4B
+0x038 CountOfOwnedCriticalSection : Uint4B
+0x03c CsrClientThread : Ptr32 Void
+0x040 Win32ThreadInfo : Ptr32 Void
+0x044 User32Reserved : [26] Uint4B
+0x0ac UserReserved : [51] Uint4B
+0x0c0 WOW32Reserved : Ptr32 Void
+0x0c4 CurrentLocale : Uint4B
+0x0c8 FpSoftwareStatusRegister : Uint4B
+0x0cc SystemReserved1 : [54] Ptr32 Void
+0x1a4 ExceptionCode : Int4B
+0x1a8 ActivationContextStack : _ACTIVATION_CONTEXT_STACK
```

그림 30 TEB 구조체

```
0: kd> dt _PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged : UChar
+0x003 SpareBool : UChar
+0x004 Mutant : Ptr32 Void
+0x008 ImageBaseAddress : Ptr32 Void
+0x00c Ldr : Ptr32 _PEB_LDR_DATA
+0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData : Ptr32 Void
+0x018 ProcessHeap : Ptr32 Void
+0x01c FastPebLock : Ptr32 _RTL_CRITICAL_SECTION
+0x020 FastPebLockRoutine : Ptr32 Void
+0x024 FastPebUnlockRoutine : Ptr32 Void
+0x028 EnvironmentUpdateCount : Uint4B
+0x02c KernelCallbackTable : Ptr32 Void
+0x030 SystemReserved : [1] Uint4B
+0x034 At1ThunkSListPtr32 : Uint4B
+0x038 FreeList : Ptr32 _PEB_FREE_BLOCK
+0x03c TlsExpansionCounter : Uint4B
+0x040 TlsBitmap : Ptr32 Void
+0x044 TlsBitmapBits : [2] Uint4B
+0x04c ReadOnlySharedMemoryBase : Ptr32 Void
```

그림 31 PEB 구조체

여기서는 0x030 오프셋 만큼의 위치에 PEB의 포인터가 하며 PEB의 0x002 오프셋 위치에 BeingDebugged 가 존재합니다. EDIT- PLUS에서 설정한 컴파일 환경을 이용하여 위의 코드를 컴파일 하여 OLLY로 좀 더 디테일하게 알아 보겠습니다.

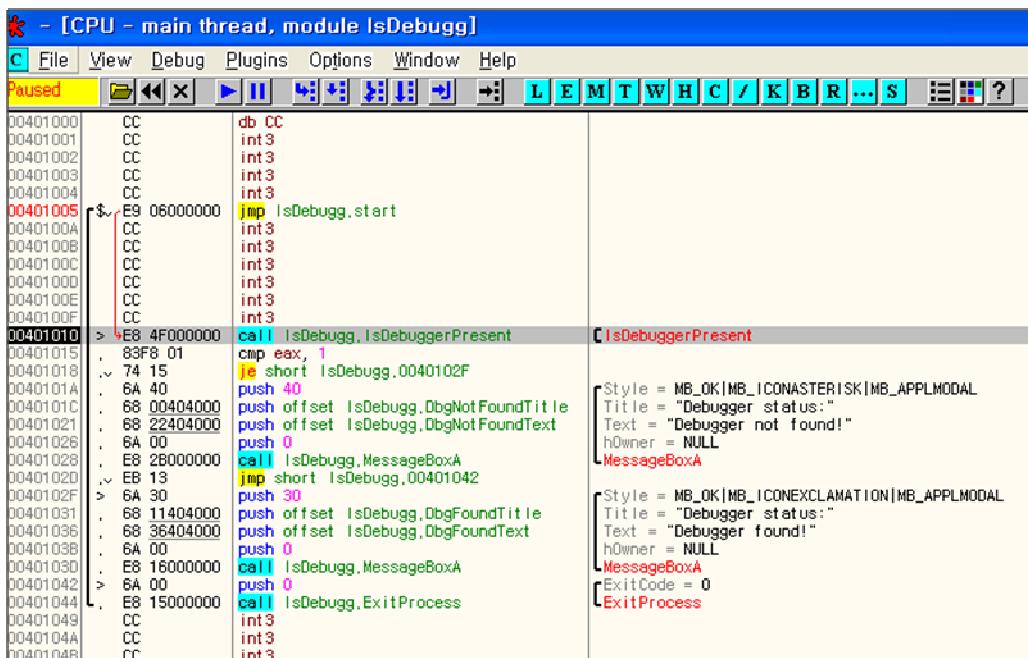


그림 32 OLLY로 열어본 IsDebuggerPresent

위의 코드에서 IsDebuggerPresent()가 호출된뒤에 리턴값이 1 이라면 디버거를 찾았다는 메시지를 띄워 준다는 걸 알 수 있습니다. 좀 더 자세히 콜문의 어셈블리를 알아 보겠습니다.

| | | | |
|----------|----------------|--------------------------------|------------------|
| 7C813091 | 90 | NOP | |
| 7C813092 | 90 | NOP | |
| 7C813093 | 64:A1 18000000 | MOV EAX, DWORD PTR FS:[18] | TEB의 위치 |
| 7C813099 | 8B40 30 | MOV EAX, DWORD PTR DS:[EAX+30] | PEB의 위치 |
| 7C81309C | 0FB640 02 | MOVZX EAX, BYTE PTR DS:[EAX+2] | BeingDebugged 위치 |
| 7C8130A0 | C3 | RETN | |
| 7C8130A1 | 90 | NOP | |
| 7C8130A2 | 90 | NOP | |
| 7C8130A3 | 90 | NOP | |

그림 33 IsDebuggerPresent call 세부 루틴

위의 달랑 세줄에서 위에 설명한 것들에 대한 것을 실행합니다. 일반적으로 FS 세그먼트 영역은 데이터 세그먼트 일종이다. 그리고 FS의 시작위치 즉 FS:[0] 은 TEB의 시작 위치를 가리킨다. 이렇게 FS:[18]은 TEB의 위치를 가르키며 그 주소값을 EAX에 담고 Eax+0x30 은 PEB의 위치를 다시금 EAX 에 담고 다시 EAX+0X2 값 BeingDebugged 값을 담아서 리턴 하고 있다. 그런데 이상하지 않은가 올리에서는 FS:[18] 을 TEB의 위치로 담고 있다. 그래서 LiveKD를 이용하여 TEB의 주소값을 출력 시킨 값과 FS:[18]의 값이 서로 일치 하는 것을 알 수 있다. 이와 같은 현상이 왜 일어 나는지 나중에 좀 더 심도 있게 다루어 볼 예정이다.

```
Couldn't resolve error at 'TEB'  
0: kd> !TEB  
TEB at 7ffd000  
    ExceptionList:      0012e174  
    StackBase:          00130000  
    StackLimit:         000f8000  
    SubSystemTib:       00000000  
    FiberData:          00001e00  
    ArbitraryUserPointer: 00000000  
    Self:               7ffd000  
    EnvironmentPointer: 00000000  
    ClientId:           00001828 . 00001b24  
    RpcHandle:          00000000  
    Tls Storage:        00000000  
    PEB Address:        7ffd9000  
    LastErrorValue:     0  
    LastStatusValue:    c0000033  
    Count Owned Locks: 0  
    HardErrorMode:      0
```

```
FS:[00000018]=[7FFDF018]=7FFDF000  
EAX=00000000
```

■ IsDebuggerPresent 우회

이를 우회 하는 방법에는 여러 가지 방법이 존재합니다 우선 해당 플래그 값을 수정 하면 됩니다. 하지만 올리에서 해당 안티 디버깅에 대한 플러그인 이 존재 합니다. 해당 플러그인은 다음 사이트에서 다운로드 받을수 있습니다.

➤ 다운 : <http://www.openrce.org/downloads/details/111/IsDebuggerPresent>

올리의 플러그인 폴더에 압축을 풀면 다음과 같은 플러그가 생기는 걸 확인 할 수 있습니다

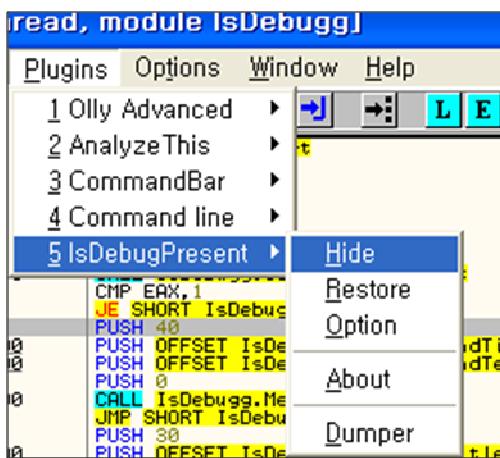


그림 34 OllyPlugin IsDebuggerPresent

위 그림에서 Hide는 IsDebuggerPresent 를 우회 한다는 말이고 Restore은 다시금 복원 한다는 의미입니다.

■ PEB.NtGlobalFlag

PEB 은 BeingDebugged 플래그 외에도, PEB 는 NtGlobalFlag 라는 필드를 갖고 있는데 NtGlobalFlag 는 PEB 으로부터 0x68 위치에 존재합니다. LiveKD 를 이용하여 PEB 의 구조체 값을 확인 해보았습니다.

```
+0x04c ReadOnlySharedMemoryBase : Ptr32 Void
+0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
+0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
+0x058 AnsiCodePageData : Ptr32 Void
+0x05c OemCodePageData : Ptr32 Void
+0x060 UnicodeCaseTableData : Ptr32 Void
+0x064 NumberOfProcessors : Uint4B
+0x068 NtGlobalFlag : Uint4B
+0x070 CriticalSectionTimeout : _LARGE_INTEGER
+0x078 HeapSegmentReserve : Uint4B
+0x07c HeapSegmentCommit : Uint4B
+0x080 HeapDeCommitTotalFreeThreshold : Uint4B
+0x084 HeapDeCommitFreeBlockThreshold : Uint4B
```

그림 35 PEB 의 **NtGlobalFlag**

이 플러그인의 값은 디버깅 중이 아니라면 0x0 값이 담기지만 디버깅 중이라면 0x70 값이 담겨집니다. 그와 같은 것을 이용하여 디버깅 탐지를 합니다

```
.386
.model flat, stdcall
option casemap :none      ; case sensitive

include c:\masm32\include\windows.inc
include c:\masm32\include\user32.inc
include c:\masm32\include\kernel32.inc
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib

.data
DbgNotFoundTitle db "Debugger status:",0h
DbgFoundTitle db "Debugger status:",0h
DbgNotFoundText db "Debugger not found!",0h
DbgFoundText db "Debugger found!",0h
.code
start:

ASSUME FS:NOTHING
MOV EAX,DWORD PTR FS:[30h]
ADD EAX,68h
MOV EAX,DWORD PTR DS:[EAX]
CMP EAX,70h
JE @DebuggerDetected

PUSH 40h
PUSH offset DbgNotFoundTitle
PUSH offset DbgNotFoundText
PUSH 0
CALL MessageBox

JMP @exit
@DebuggerDetected:

PUSH 30h
PUSH offset DbgFoundTitle
```

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

```
PUSH offset DbgNotFoundText
```

```
PUSH 0
```

```
CALL MessageBox
```

```
JMP @exit
```

```
@DebuggerDetected:
```

```
PUSH 30h
```

```
PUSH offset DbgFoundTitle
```

```
PUSH offset DbgFoundText
```

```
PUSH 0
```

```
CALL MessageBox
```

```
@exit:
```

```
PUSH 0
```

```
CALL ExitProcess
```

```
end start
```

여기서 **ASSUME FS:NOTHING** 이란 구문은 세그먼트 레지스트의 주소값을 재할당 하는 것
이 아니라 이 디렉티브를 맞나면 실행 시에 어셈블러가 주소를 계산하는 방법을 변경한다.
즉 **FS** 세그먼트에 **NOTHING**이라는 속성을 붙이는 의미이다..

해당 내용을 더 깊어 알아 보았는데 다음과 같은 사이트에서 친절히 설명 해주고 있다.

➤ 참고 : <http://www.winasm.net/forum/index.php?showtopic=2082>

간단히 축약해서 얘기하자면(맞는 얘기인지도 모르겠다, 필자의 영어실력은 가히 밑바닥 수준이라서...) FS:[0] 은 Exception handler 를 Default 를 가지고 있다. MASM 컴파일러는 기본적으로 이 레지스터를 사용할 때 ERRORTOOLBOX 를 뱉기 때문에 위와 같이 ASSUME FS:NOTHING 을 써주면 그와 같은 ERRORTOOLBOX Check 를 Remove 해준다고 설명이 되어 있다. 다음으로 FS:[30] 는 PEB 의 위치에서 ADD EAX,68h 에는 NtGlobalFlag 가 존재한다. 해당 코드를 올리로 열어 보았다.

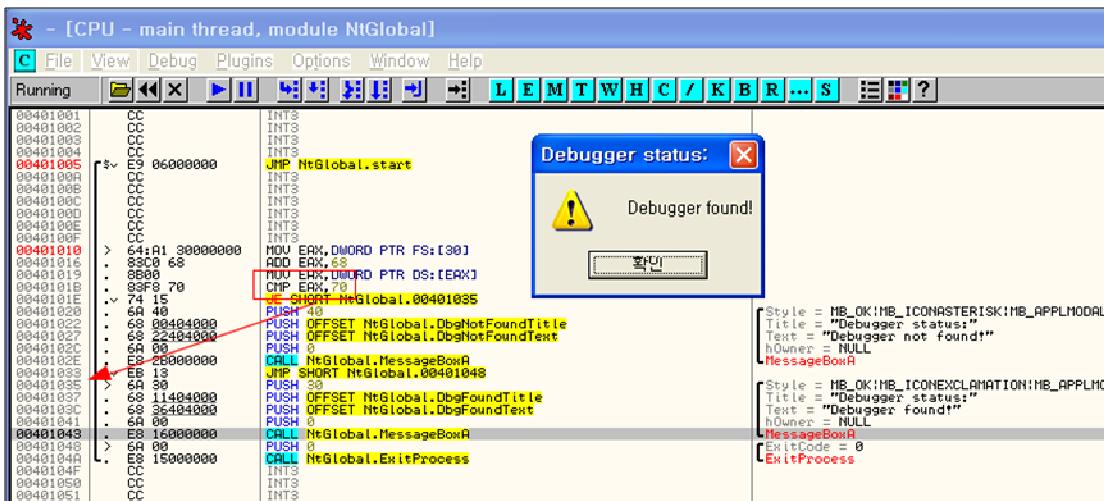


그림 36 Olly 로 확인한 NtGlobalFlag 루틴

■ PEB.NtGlobalFlag 우회

해당 탐지를 우회는 간단하다. 직접 코드 패치를 하여 바꾸거나 혹은 플래그 값을 수정하거나 이다. 하지만 이것도 역시 올리에서 플러그인 형태로 지원을 하고 있다. 다음 사이트에서 다운로드 하여 쓰시면 됩니다.

- 다운 : http://www.openrce.org/downloads/details/241/Olly_Advanced

해당 플러그 인을 실행하고 NtGlobalFlag 를 체크 뒤 확인 해주면 자동적으로 리턴 시 플래그 값을 수정하여 디버거 탐지를 우회하게 됩니다. 그 외 코드 패치 등 다양한 방법 이 존재 합니다. 툴로서 편리한 안티리버싱을 할수 있습니다

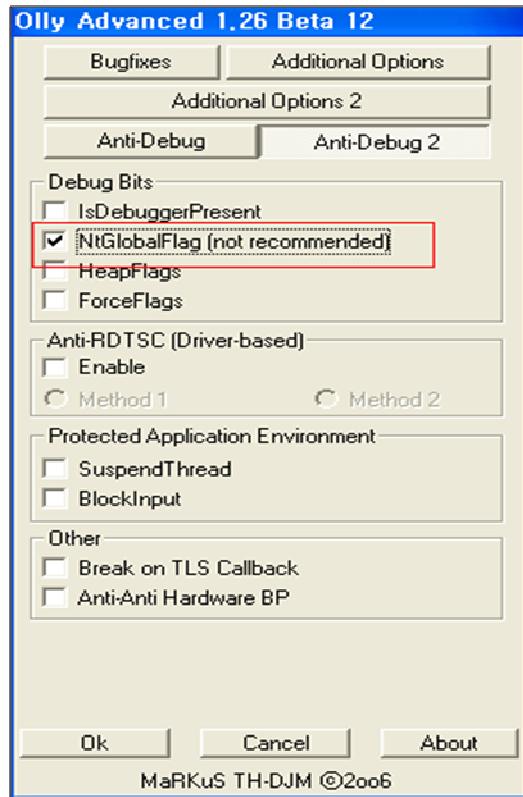


그림 37 Olly Advanced NtGlobalFlag

■ *Heap.HeapFlag , Heap.ForceFlags*

ProcessHeap 은 PEB 에서 0x18 만큼 떨어져 있고 이 플래그는 프로세서가 디버깅 될 때 heap 이 만들어 지고 이때 설정 되는 플래그들은 HeapFlag 와 ForceFlag 입니다. 처음 프로그램이 힙 영역을 만들면 ForceFlags 에는 0x0 값이 HeapFlag 에는 0x2 값이 설정 됩니다. 하지만 디버깅 중이라면 NtGlobalFlag 에 따라 두개의 플래그 값이 변경 됩니다. 만약 디버깅이 탐지 되어 변경 되었다면 ForceFlags 에는 0x40000060 값이 할당 되며 HeapFlag 에는 0x50000062 값이 할당 됩니다. 이때 변경된 값을 체크 하므로써 디버깅의 유무를 판별합니다.

```
0: kd> dt _PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged      : UChar
+0x003 SpareBool          : UChar
+0x004 Mutant              : Ptr32 Void
+0x008 ImageBaseAddress   : Ptr32 Void
+0x00c Ldr                 : Ptr32 _PEB_LDR_DATA
+0x010 ProcessParameters  : Ptr32 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData      : Ptr32 Void
+0x018 ProcessHeap         : Ptr32 Void
+0x01c FastPebLock        : Ptr32 _RTL_CRITICAL_SECTION
+0x020 FastPebLockRoutine : Ptr32 Void
+0x024 FastPebUnlockRoutine : Ptr32 Void
+0x028 EnvironmentUpdateCount : UInt4B
+0x02c KernelCallbackTable : Ptr32 Void
```

그림 38 PEB 의 *ProcessHeap* 구조체 offset

여기서 *ProcessHeap* 의 구조체를 살펴보면 언급한 두개의 값들이 존재합니다.

```
0: kd> dt _Heap
ntdll!_HEAP
+0x000 Entry           : _HEAP_ENTRY
+0x008 Signature       : UInt4B
+0x00c Flags            : UInt4B
+0x010 ForceFlags       : UInt4B
+0x014 VirtualMemoryThreshold : UInt4B
+0x018 SegmentReserve  : UInt4B
+0x01c SegmentCommit    : UInt4B
+0x020 DeCommitFreeBlockThreshold : UInt4B
+0x024 DeCommitTotalFreeThreshold : UInt4B
+0x028 TotalFreeSize    : UInt4B
+0x02c MaximumAllocationSize : UInt4B
```

그림 39 *Process_Heap* 구조체

◆ Process_Heap Code

```
.386
.model flat, stdcall
option casemap :none      ; case sensitive

include c:\masm32\include\windows.inc
include c:\masm32\include\user32.inc
include c:\masm32\include\kernel32.inc

includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib

.data
DbgNotFoundTitle db "Debugger status:",0h
DbgFoundTitle db "Debugger status:",0h
DbgNotFoundText db "Debugger not found!",0h
DbgFoundText db "Debugger found!",0h
.code

start:

ASSUME FS:NOTHING
MOV EAX,DWORD PTR FS:[18h]    ;TEB
MOV EAX,DWORD PTR [EAX+30h]      ;PEB
MOV EAX,DWORD PTR[EAX+18h] ;Process_Heap
CMP DWORD PTR DS:[EAX+10h],0 ;Force_Flags
JNE @DebuggerDetected

MOV EAX,DWORD PTR FS:[18h]    ;TEB
MOV EAX,DWORD PTR [EAX+30h]      ;PEB
MOV EAX,DWORD PTR[EAX+18h] ;Process_Heap
CMP DWORD PTR DS:[EAX+0ch],2 ;Force_Flags
JNE @DebuggerDetected
```

```
PUSH 40h
PUSH offset DbgNotFoundTitle
PUSH offset DbgNotFoundText
PUSH 0
CALL MessageBox
```

```
JMP @exit
@DebuggerDetected:
```

```
PUSH 30h
PUSH offset DbgFoundTitle
PUSH offset DbgFoundText
PUSH 0
CALL MessageBox
```

```
@exit:
```

```
PUSH 0
CALL ExitProcess
```

```
end start
```

■ **Heap.HeapFlag , Heap.ForceFlags** 우회

```
MOV [NtAddr],EAX
```

```
MOV EAX,offset MinusOne
```

```
PUSH EAX
```

```
MOV EBX,ESP
```

```
PUSH 0
```

```
PUSH 4
```

```
PUSH EBX
```

```
PUSH 7
```

```
PUSH DWORD PTR[EAX]
```

```
CALL [NtAddr]
```

```
POP EAX
```

```
TEST EAX,EAX
```

```
JNE @DebuggerDetected
```

```
PUSH 40h
```

```
PUSH offset DbgNotFoundTitle
```

```
PUSH offset DbgNotFoundText
```

```
PUSH 0
```

```
CALL MessageBox
```

```
JMP @exit
```

```
@DebuggerDetected:
```

```
PUSH 30h
```

```
PUSH offset DbgFoundTitle
```

```
PUSH offset DbgFoundText
```

```
PUSH 0
```

```
CALL MessageBox
```

```
@exit:
```

◆ NtQueryInformationProcess Code

```
MOV [NtAddr],EAX

MOV EAX,offset MinusOne
PUSH EAX
MOV EBX,ESP

PUSH 0
PUSH 4
PUSH EBX
PUSH 7
PUSH DWORD PTR[EAX]
CALL [NtAddr]

POP EAX

TEST EAX,EAX
JNE @DebuggerDetected

PUSH 40h
PUSH offset DbgNotFoundTitle
PUSH offset DbgNotFoundText
PUSH 0
CALL MessageBox

JMP @exit

@DebuggerDetected:

PUSH 30h
PUSH offset DbgFoundTitle
PUSH offset DbgFoundText
PUSH 0
CALL MessageBox

@exit:
```

위의 코드에서 ? 는 아마 변수를 선언할 때 값을 할당하지 않은 상태일꺼라 생각이 듭니다. 어째든 위의 코드에서 커널 API 를 수행하기 위해서 LoadLibrary 와 GetProcAddress 를 이용하여 해당 함수 주소를 얻은뒤 앞서 설명한 인자값을 토대로 호출하고 디버깅 중이라면 리턴되는 값을 체크하여 디버거 존재 유무를 판별 하고 있습니다. 다음 그림은 위의 코드를 Olly 에서 확인한 장면입니다.

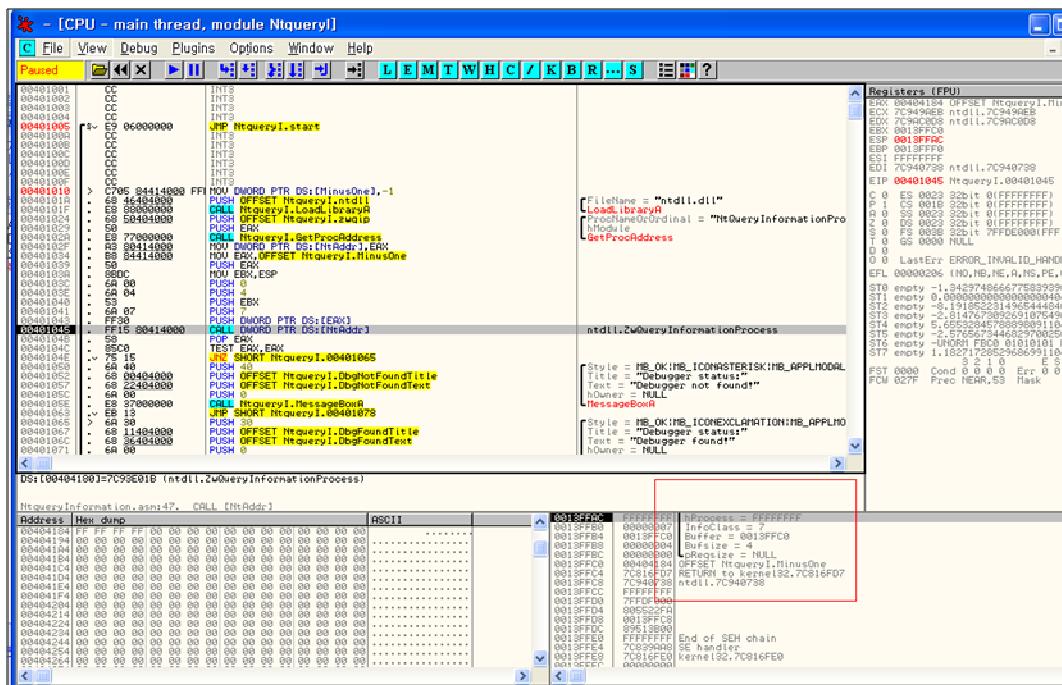


그림 41 NtQueryInformationProcess in Olly

그림에서 알수 있듯이 인자 값이 저런식으로 들어가 호출하게 되어야지 디버깅 체크를 할수 있습니다.

■ NtQueryInformationProces 우회

이 방법을 우회하는 방법 역시 여려가지가 존재 할수 있습니다. 다만 올리 플러그인중 Olly Advanced 는 NtQueryInformationProcess 의 인자 값중 하나인 hProcess 를 0 으로 만듬으로써 해당 루틴을 우회하게 될꺼라고 생각했으나 잘 안 되는군요. 일단은 그냥 주 루틴이 나오면 리턴되는 부분에 EAX 값을 0 을 설정하면 우회가 됩니다.

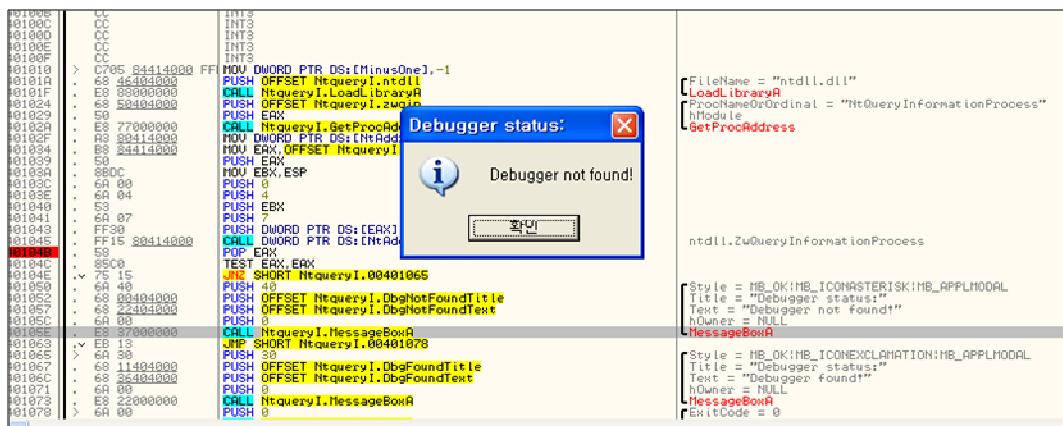


그림 42 NtQueryInformation 우회

■ Debugger Interrupts

이 방법은 디버거가 하나의 소프트 브레이크 포인트를 수행하려고 할 때 INT3(0xcc)를 삽입 함으로써 수행이 된다. 인터럽터가 수행이 된다는 말은 즉 예외처리가 일어 난다는 의미이다. 하지만 디버거가 디버깅 수행중 일 때 같은 OP 코드인 INT3를 만난다면 예외 처리없이 수행을 하게된다. 이 Debugger Interrupts 는 바로 이러한 것을 차단하여 안티 디버깅을 수행한다. 즉 Interrupt 수행중에 같은 인터럽터 코드를 만났을 때 exception을 하지 않는 경우를 디버깅 중이라고 판별하게 된다. 좀 더 유연한 이해를 위해 아래 코드를 참고 한다

◆ Debugger Interrupts

```

.386
.model flat, stdcall
option casemap :none      ; case sensitive

include c:\masm32\include\windows.inc
include c:\masm32\include\user32.inc
include c:\masm32\include\kernel32.inc

includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib

```

```
.data
msgTitle db "Execution status:",0h
msgText1 db "No debugger detected!",0h
msgText2 db "Debugger detected!",0h
.code
```

```
start:
```

```
ASSUME FS:NOTHING
PUSH offset @Check
PUSH FS:[0]
MOV FS:[0],ESP
```

```
; Exception
```

```
INT 3h
```

```
PUSH 30h
PUSH offset msgTitle
PUSH offset msgText2
PUSH 0
CALL MessageBox
```

```
PUSH 0
CALL ExitProcess
```

```
; SEH handleing
@Check:
POP FS:[0]
ADD ESP,4
```

```
PUSH 40h
PUSH offset msgTitle
PUSH offset msgText1
PUSH 0
CALL MessageBox
```

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

```
PUSH 0
CALL ExitProcess

; SEH handleing
@Check:
POP FS:[0]
ADD ESP,4

PUSH 40h
PUSH offset msgTitle
PUSH offset msgText1
PUSH 0
CALL MessageBox
```

위에 코드에서 @Check 는 exception 이 일어 날 때 등록되는 SEH 이다. INT 3 이 수행될 때 익셉션이 일어 나지 않는다면 Debugger detect 메시지가 출력 될것이다. 다음은 올리에서 살펴본 모습이다.

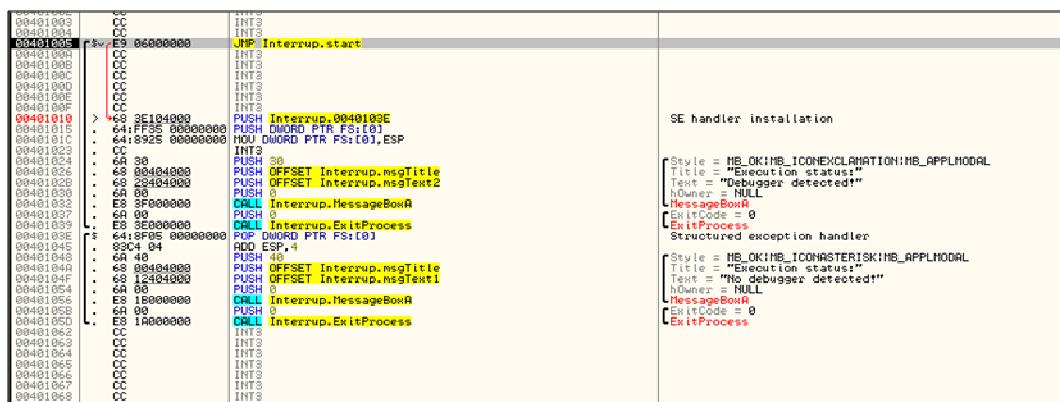


그림 43 Debugger Interrupt in Olly

저기에서 INT3 을 유심히 살펴 보면 저기가 분기문이라고 이해하시는 편이 더 수월 할수도 있습니다 INT3 이 exception 을 일으킨다면 0x0040103e 로 향하게 될것이며 그렇지 않을경우 0x00401024 코드로 향하게 될것입니다.

■ **Debug Interrupts** 우회

우회 방법은 간단합니다. 올리에서 디버깅 옵션에 보면 INT 3 을 만났을 때 무시하지 않고 SEH를 following 하라는 옵션을 선택하여 줍니다.

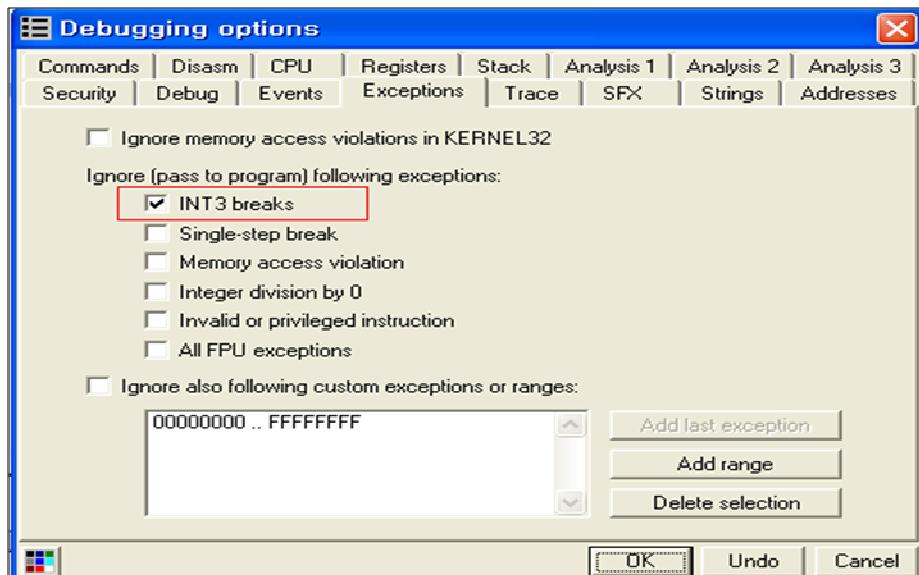


그림 44 Debugger Interrupts 우회 옵션

다음과 같이 우회를 하게 된다면 Exception Handle로 빠지는 루틴을 보실수 있습니다.

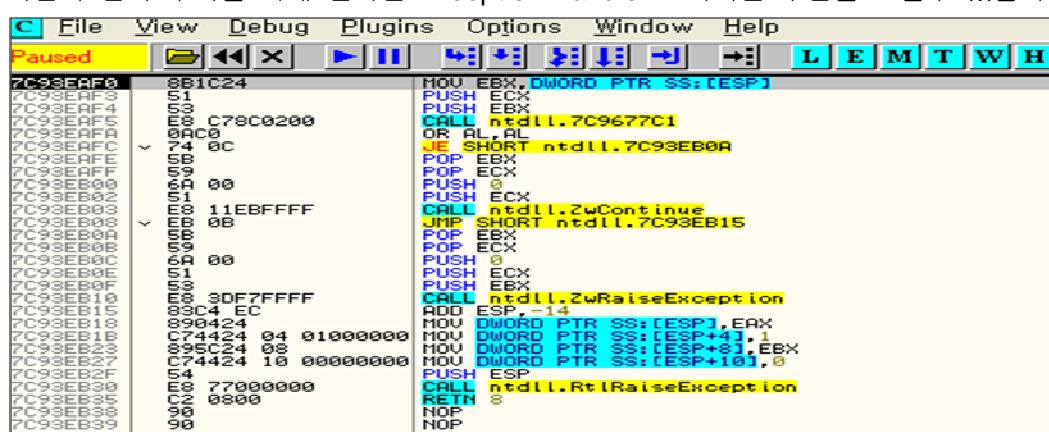


그림 45 Exception Handle 을 호출하는 루틴

■ Hardware Break Point

이왕 인터럽터나 SEH에 대해서 나온 만큼 브레이크 포인트에 대한 안티 리버싱 기술에 대하여 살펴 보겠습니다. 사실 위에서 언급한 Debugger interrupts만으로도 소프트 브레이크 포인터에 대한 탐지가 가능합니다. 물론 언급한 내용외에 코드를 체크하거나 체크섬을 검사하여 탐지를 할수 있습니다. 이번 장에서는 Hardware Break Point를 어떻게 탐지하며 우회를 할 것인가에 대하여 다루겠습니다. 시작하기에 앞서 먼저 디버그 레지스터리에 대한 이해가 필요합니다. 디버그 레지스터는 총 7개가 구성이 됩니다. (DR0~DR7) 또한 DR4와 DR5는 사용되지 않고 있으며 DR0~DR3까지는 브레이크 포인트가 걸린 주소값을 담는 용도입니다. 그래서 하드웨어 브레이크 포인트가 4개밖에 존재하지 않습니다. 이러한 하드웨어적인 방식은 DR0~DR3중 하나에 브레이크 포인트로 사용될 주소로 설정하여 준후 DR7 레지스터에 브레이크가 발생하게 될 조건을 명시하게 됩니다. 좀더 중요한 DR7 레지스터리에 알아봅시다. 아래 출처에서 설명되어 있어서 굳어 오듯이 설명하겠습니다.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----------|----------|----------|----------|----------|----------|----------|----|----|--------|----|----|----|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|---|-----|---|---|---|---|---|---|
| LEN 3 | R/W 3 | LEN 2 | R/W 2 | LEN 1 | R/W 1 | LEN 0 | R/W 0 | 0 | 0 | G D | 0 | 0 | 1 | G E | L E | G L | L 3 | G L | L 2 | G L | L 1 | G L | L 0 | 0 | DR7 | | | | | | |

그림 46 하드웨어-브레이크-포인트-탐지

➤ Image 출처 : <http://slaxcore.tistory.com/entry/>

DR7은 Debug Control Register입니다. 브레이크 포인터 타입 및 활성화 여부 브레이크 포인트 길이 등이 저장됩니다. 아래는 위 그림에서 RW0~RW3의 설정값

| | |
|----|---|
| 00 | Break on instruction execution only |
| 01 | Break on data writes only |
| 10 | Undefined |
| 11 | Break on data reads or writes but not instruction fetches |

그림 47 RW0~RW3

다음 그림은 LEN0~LEN3에 설정되는 브레이크 포인터 Data에 대한 길이 값이 저장됩니다.

| | |
|----|------------------|
| 00 | one-byte length |
| 01 | two-byte length |
| 10 | Undefined |
| 11 | four-byte length |

그림 48 LEN0~LEN3

또한 (L0~L3,G0~G3)은 새로운 브레이크 포인터에 대한 설정값들이 설정됩니다. 앞서 SEH에 대하여 간략하게 설명드렸습니다. 좀더 깊게 이해하기 위하여 정덕영님의 Windows 구조와 원리 p162에 나와 있는 글로써 인용을 하겠습니다.

프로세스가 메모리와 각종 자원에 대한 구별을 가진다면 스레드는 목적한 코드를 일정 시간 동안 수행하는 실행 단위로 구별할 수 있으며, 각각의 스레드가 가지는 독립적인 요소 중 하나는 에러 핸들러이다.

각각의 스레드는 자신이 목적하는 코드를 수행하게 되며, 만약 이코드가 수행하던 중 예외적인 상황이 발생하게 되면 운영체제와 컴파일러는 이 예외적 상황을 처리할 수 있는 기회를 스레드에게 제공하고 있다.C나 C++와 같은 컴파일러에서는 우리에게 _try,_except,catch,thro와 같은 키워드를 제공함으로써 예외적 상황에 대해 프로그래머가 처리할 수 있는 기회를 제공해 주고 있다. 하지만 여기서 간과해서는 안되는 일은 이는 컴파일만으로는 되지 않는다는 점이다.

즉, 어떠한 스레드에서 잘못된 메모리를 참조하는 것과 같은 행위를 하게 되면 마이크로프로세서에서는 예외가 발생하게 되며, 이때 이 예외를 어떻게 처리하는지는 OS가 해주어야 하는 몫이다. Windows에서는 이러한 예외가 발생하면 그 예외를 발생시킨 스레드가 그에 대한 에러 처리를 할 수 있도록 해주고 있으며, 이를 구조화된 예외처리라고 부른다.

간단한 SEH 처리 과정을 코드와 더불어 설명하겠습니다.

◆ SimpleSEH.cpp

```
#include <windows.h>
ULONG G_nValid;

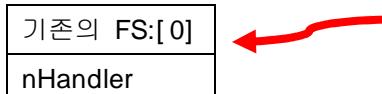
EXCEPTION_DISPOSITION __cdecl except_handler(           //예외 처리
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void *EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void *DisatcherContext
)
{
    ContextRecord->Eax = (ULONG)&G_nValid;
    return ExceptionContinueExecution;
}

int main(int argc,char *argv[])
{
    ULONG nHandler = (ULONG)except_handler;
    PCHAR pTest = (PCHAR)0;
    __asm      //SEH 설치
    {
        push nHandler //예외 핸들링을 수행할 함수 주소
        push FS:[0]
        mov FS:[0],ESP
    }
    __asm
    {
        mov eax,0
        mov [eax],'a'      //SEH 발생 잘못된 메모리 참조
    }
    __asm      //SEH 제거
    {
        mov eax,[ESP]
        mov FS:[0],EAX
        add esp,8
    }
    return 0;
}
```

FS:[0]

Mov FS:[0],ESP

위의 코드는 사용자가 만든 SEH 핸들러에 대한 처리를 정의하고 새로운 SHE를 발생시키기 위하여 0의 주소값을 가리키게 하여 SHE를 발생 시키는 코드입니다. SEH 설치 부분을 먼저 살펴 보겠습니다.



현재 ESP 값이 기존의 FS:[0] 값으로 가리키게 됩니다. 이 과정은 우리가 직접 넣은 push 명령으로 넣은 데이터들은 실제 Windows에서 EXCEPTIN_REGISTRATION_RECORD라는 구조체로 정의하고 있는 데이터입니다. 구조체의 정의는 이렇습니다.

현재 ESP 값이 기존의 FS:[0] 값으로 가리키게 됩니다. 이 과정은 우리가 직접 넣은 push 명령으로 넣은 데이터들은 실제 Windows에서 EXCEPTIN_REGISTRATION_RECORD라는 구조체로 정의하고 있는 데이터입니다. 구조체의 정의는 이렇습니다

```
Typedef struct  
EXCEPTIONREGISTRATIONRECORD  
{  
    DWORD prev_structure; //이전에 설치된 Exception handler  
    DWORD ExceptionHandler; //해당 에러 핸들러  
}
```

이런식으로 싱글리스트 형태로 이전의 Exception handler 가 다음의 Exception handler 를 가리키게 됩니다. 다음으로는 에러 처리에 해당하는 코드를 살펴보겠습니다. 코드를 보면 에러가 발생한 시점의 Eax 값을 전역변수 G_nValid에 담아서 ExceptionContinueExecution 을 리턴하고 있습니다. 이값은 0 값으로 VC 헤더 파일인 EXCPT.H에 정의 되어 있습니다.

```
Typedef enum _EXCEPTION_DISPOSITION{  
    ExceptionContinueExecution, //0  
    ExceptionContinueSearch, //1  
    ExceptionNestedException, //2  
    ExceptoinCollidedUnwind //3  
}
```

이러한 값은 악성 코드들에 많이 이용 됩니다. 보통 악성코드들은 앞서 살펴본 디버거 레지스터리 값을 바꿀 때 사용되어 지는 `ContextRecord` 을 이용하여 디버거 레지스터리의 값을 바꾸어 여러가지에 응용하며 또한 `ExceptionContinuExecution(0)` 값을 리턴함으로써 예러가 발생할 당시의 `Register` 을 바꾸어 줄 수 있는데 보통 EIP 값을 수정하여 예러가 발생한 시점에 악성코드를 실행한 뒤(exception handler) EIP 를 수정하여 보통의 OEP 를 가장하는 기법을 이용합니다.(실제로 디버거를 이용하여 디버깅을 할 때 OEP) 다음은 구조체 CONTEXT 의 일부인 디버거 레지스터리를 정의 하고 있는 부분입니다.



그림 49 Context Record 구조체중 디버거 레지스터리

Hardware Break Point 탐지에 필요한 개념을 이해 하였으면 아래의 코드로 어떻게 탐지를 할수 있는지 얘기를 해보겠습니다.

```
.386
.model flat, stdcall
option casemap :none      ; case sensitive

include c:\masm32\include\windows.inc
include c:\masm32\include\user32.inc
include c:\masm32\include\kernel32.inc

includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib

.data
    KoreaSecurity db "Korea Security",0h
    Protect db "보호하고 싶은가?",0h
    DbgNotFoundTitle db "Debugger status:",0h
    DbgFoundTitle db "Debugger status:",0h
    DbgNotFoundText db "Debugger hardware bpx not found!",0h
    DbgFoundText db "Debugger hardware bpx found!",0h
.data?
    OrgEbp    dd ?
    OrgEsp    dd ?
    SaveEip   dd ?
.code
```

org

```
start:  
; Setup SEH  
MOV EAX,offset @Exit  
MOV DWORD PTR[OrgEbp],EAX ;EAX에 @Exit 함수주소를 담고 있음  
MOV DWORD PTR[SaveEip],EBP; 현재 EBP를 SaveEip라고 담고 있음.  
ASSUME FS : NOTHING  
  
PUSH offset @DetectHardwareBPX ;Exception handler  
PUSH FS:[0]  
MOV DWORD PTR[OrgEsp],ESP ;현재 스택을 OrgEsp에 담고 있음.  
MOV FS:[0], ESP ; 위에서 설명한 과정  
; Fire SEH  
XOR EAX,EAX ;EAX 값을 0 초기화  
XCHG DWORD PTR DS:[EAX],EAX ;0이라는 주소값 참조 exception 발생  
CALL @Protected;Hardware Break Point로부터 보호하고 싶은 영역  
@Exit:  
POP FS:[0]  
ADD ESP,4  
PUSH 0  
CALL ExitProcess  
@Protected:  
PUSH 30h  
PUSH offset KoreaSecurity  
PUSH offset Protect  
PUSH 0  
CALL MessageBox  
@DetectHardwareBPX:  
PUSH EBP;핸들러가 발생할 때의 처리  
MOV EBP,ESP  
MOV EAX,DWORD PTR SS:[EBP+10h] ;Context Record 값 세번째인자  
; Restore ESP, EBP, EIP  
MOV EBX,DWORD PTR[OrgEbp] ;@exit 함수 주소  
MOV DWORD PTR DS:[EAX+0B8h],EBX;Context Record의 Ebp 값  
MOV EBX,DWORD PTR[OrgEsp];fs[0]과 esp가 바뀌기 전의 esp 값
```

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

```

MOV DWORD PTR DS:[EAX+0C4h],EBX;Context Record 의 esp 값
MOV EBX,DWORD PTR[SaveEip];
MOV DWORD PTR DS:[EAX+0B4h],EBX;context Record 의 eip 값

; Check DRx registers

CMP DWORD PTR DS:[EAX+4h],0
JNE @hardware_bpx_found
CMP DWORD PTR DS:[EAX+8h],0
JNE @hardware_bpx_found
CMP DWORD PTR DS:[EAX+0Ch],0
JNE @hardware_bpx_found
CMP DWORD PTR DS:[EAX+10h],0
JNE @hardware_bpx_found
PUSH 40h
PUSH offset DbgNotFoundTitle
PUSH offset DbgNotFoundText
PUSH 0
CALL MessageBox
@hbpexit:
MOV EAX,0 ;위에서 설명한 ExceptionContinuExecution(0) 값
LEAVE
RET
@hardware_bpx_found:
PUSH 30h
PUSH offset DbgFoundTitle
PUSH offset DbgFoundText
PUSH 0
CALL MessageBox
JMP @hbpexit

end start

```

다소 이때까지 본 코드보다는 길어 보이지만 복잡하지는 않다. 앞전에 설명한대로 SHE 를 설치하고 SEH 처리에서는 디버거 레지스터리를 검사하여 Context Record 값중 ebp, esp, eip 를 변경하여 다시금 변경된 레지스터리 값을 복구하는 과정이다. 이를 올리로 본 화면은 이렇다. 아래 그림에서 보다시피 보호 하고 싶은 영역에 하드웨어 브레이크 포인트를 설정하였다. 확인하는 방법은 올리에서 Debug → Hardware Breakpoints 를 선택하면 된다.

■ Hardware Break Point 우회

앞선 설명을 다 이해 했다면 우회하는 방법은 간단합니다. 디버깅 체크루틴에서 강제적으로 디버거 레지스터리 값을 0 으로 만들어 주거나 혹은 Exception 을 유발 시키는 코드에 대해서 NOP 을 처리해 해당 루틴이 실행하지 않도록 합니다.

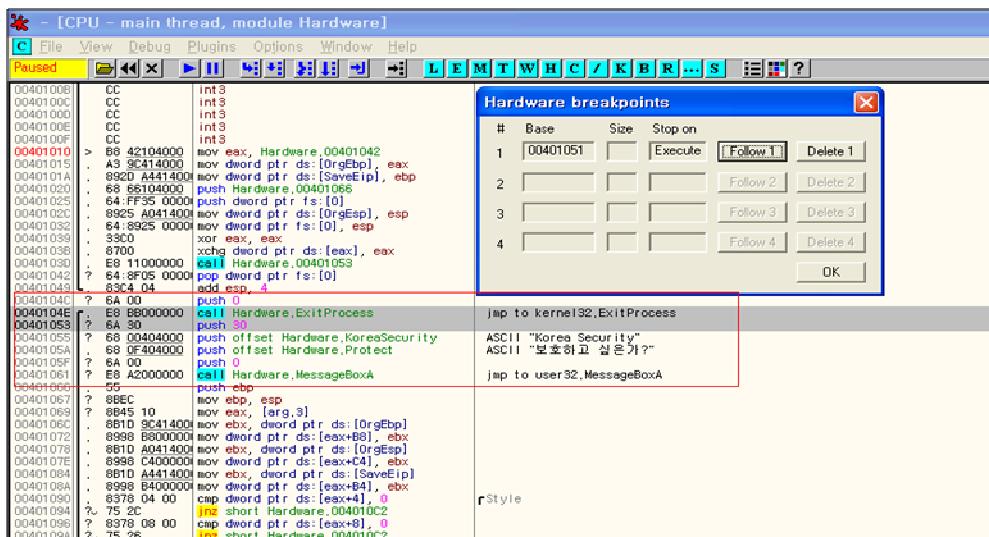


그림 50 Protect 영역에 Hardware break Point

F9 를 눌러 run 을 하여 보자.

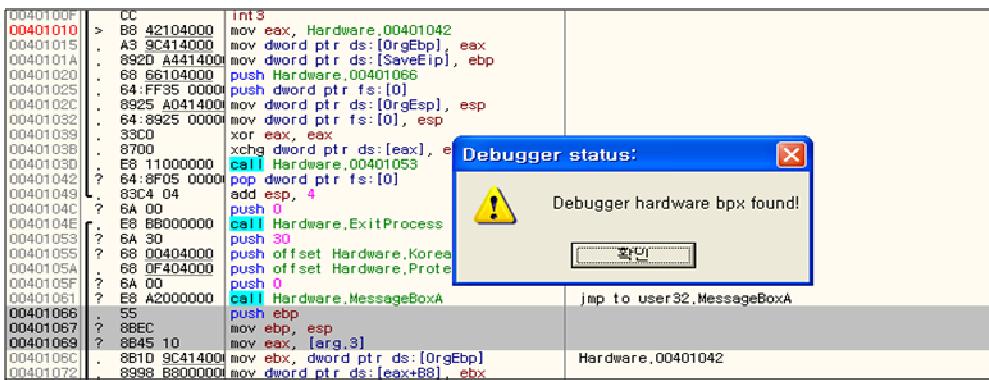


그림 51 Hardware break Point 발견

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

■ **Garbage Code & JunkCode**

지금부터 설명한 앤티 리버싱 방법은 앞선 기술과 달리 리버서들을 심리적으로 짜증나고 지치게 만드는 기법입니다. 단지 보호 될 코드를 분석할 때 시간을 끄는 용도로 사용됩니다. 두 개념은 다소 다른 의미를 지니고 있습니다. 먼저 Garbage 코드는 코드중간에 의미없는 코드들을 집어 넣어 리버서들의 집중력(?)과 혼란을 가중 시켜 버리며 Junkcode 는 가령 디버거는 코드를 디어셈 할 때 정의된 OPCODE에 의해 디어셈을 화면에 DISPLAY 합니다. 그것을 착안하여 실행하지 않도록 하는 OPCODE를 중간에 넣으므로써 전혀 엉뚱한 디어셈 코드를 출력함으로써 리버서를 방해 합니다. 앞서 작성한 SHE 를 유발시키는 코드를 보호될 코드 영역으로 정하고 Garbage 코드를 삽입 해 올리로 살펴보겠습니다.

보호되어야 할 코드

XOR EAX,EAX

XCHG DWORD PTR DS:[EAX],EAX

위와 같은 코드가 보호 되어야 할 것이라면 중간에 Garbage 코드(프로그램에 상관없이 의미 없는 코드)를 삽입 합니다.

Garbage 코드 삽입

PUSH EAX

MOV EBX,3

POP EAX

SUB EBX,3

XOR EAX,EAX

MOV EAX,0DEADh

SHR EAX,4

XCHG DWORD PTR DS:[EAX],EAX

간단하게 적용한 모습입니다. 물론 저정도 코드는 중간에 삽입된게 의미가 없다는걸 한눈에 알아 볼수 있지만 이러한 코드들이 무더기로 중간중간에 삽입이 되어 있다면 골치가 아플 것입니다. 다음은 올리에서 적용된 모습입니다

| | | | |
|----------|------------------|---|-----------------------------|
| 00401020 | 68 75104000 | PUSH Hardware.00401020 | |
| 00401025 | 64:FF35 00000000 | PUSH DWORD PTR FS:[0] | |
| 0040102C | 8925 A0414000 | MOV DWORD PTR DS:[0x414000],ESP | |
| 00401032 | 64:8925 00000000 | MOV DWORD PTR FS:[0],ESP | |
| 00401039 | 58 | PUSH EBX | |
| 0040103A | BB 00000000 | MOV EBX,0 | |
| 00401040 | 83EB 03 | POP EBX | |
| 00401043 | 33C0 | XOR EBX,EBX | |
| 00401045 | B8 ADE0000 | MOV EBX,0DE0AD | |
| 0040104A | C1E8 04 | SHR EBX,4 | |
| 0040104D | 87E0 | XCHG DWORD PTR DS:[EBX],EBX | |
| 00401054 | ED 11000000 | CALL Hardware.00401054 | |
| 0040105B | 93C4 04 | ADD ESP,4 | JMP to kernel32.ExitProcess |
| 0040105E | 6A 00 | PUSH 0 | ASCII "Korea Security" |
| 00401060 | E8 C1000000 | CALL Hardware.ExitProcess | |
| 00401065 | 6A 30 | PUSH 30 | |
| 00401067 | 68 00404000 | PUSH OFFSET Hardware.KoreaSecurity | |
| 0040106C | 68 0F404000 | PUSH OFFSET Hardware.Protect | |
| 00401071 | 6A 00 | PUSH 0 | JMP to user32.MessageBoxA |
| 00401073 | ED A0000000 | CALL Hardware.MessageBoxA | |
| 00401079 | 8BEC | POP EBSP | Hardware.00401054 |
| 0040107B | 8B45 10 | MOV EBX,ESP | |
| 0040107E | 8B1D 90C414000 | MOV EBX,DWORD PTR DS:[0x90C414000] | |
| 00401084 | 8998 B8000000 | MOV DWORD PTR DS:[EBX+B8],EBX | |
| 0040108A | 8B1D A0414000 | MOV EBX,DWORD PTR DS:[0x414000] | |
| 00401090 | 8998 C4000000 | MOV DWORD PTR DS:[EBX+C4],EBX | |
| 00401096 | 8B1D A4414000 | MOV EBX,DWORD PTR DS:[0x4414000] | |
| 0040109C | 8998 B4000000 | MOV DWORD PTR DS:[EBX+B4],EBX | |
| 004010A2 | 3378 04 00 | CMP DWORD PTR DS:[EBX+4],0 | Style |

그림 52 Garbage코드가 삽입된 올리화면

다음은 Junk Code 를 삽입 하여 보겠습니다. 여기에 해당하는 내용은 제스리버님의 홈페이지의 테스트를 참고로 하였습니다. 코드는 아래와 같습니다.

```
#include <windows.h>

int main(int argc,char *argv[])
{
    __asm
    {
        jmp here+1;
        here:
        __emit 0xe9//__emit 은 뒤에 바이트를 코드에 포함
        mov eax,1
    }
    return 0;
}
```

코드를 보면 중간에 박아서 디버거가 제대로된 코드를 뿌려주지 못하도록 하고 있습니다.

다음은 올리로 확인한 모습입니다.

```
C File View Debug Plugins Options Window Help
Paused [File] [View] [Debug] [Plugins] [Options] [Window] [Help]
[File] [View] [Debug] [Plugins] [Options] [Window] [Help]
00401019: . 8D7D C0    lea edi, [local.16]
0040101C: . B9 10000000  mov ecx, 10
00401020: . DD CCCCCCCC  mov eax, cccccccc
00401025: . F3:A8    rep stos dword ptr es:[edi]
00401028: . E9 D10000000 jmp SEH_Stru.0040102E
00401029: E9          db E9
0040102A: B8          db B8
0040102B: > 0100    add dword ptr ds:[eax], eax
0040102C: 0000    add byte ptr ds:[eax], al
0040102D: ? 33C0    xor eax, eax
0040102E: ? SF       pop edi
0040102F: SE          pop esi
00401030: SB          pop ebx
00401031: 83C4 40    add esp, 40
00401032: ? 3BEC    cmp ebp, esp
00401033: ? E8 1E000000 call SEH_Stru._chkesp
00401034: ? 8BE5    mov esp, ebp
00401035: ? 50       pop ebp
00401036: C3          ret
00401037: CC          int3
00401038: CC          int3
00401039: CC          int3
00401040: CC          int3
```

그림 53 Junk Code 삽입

MOV EAX,1 디어셈 코드는 온데 간데 보이질 않고 덩그러니 JMP코드가 자리 잡고 있습니다. 그런데 JMP하는 주소값이 이상하네요. 저정도야 금방 눈치채겠지만 그래도 모르는 상태에서 본다면 다소 의아해 할 것 입니다. 해당 옵코드 0xe9 를 0x90(NOP)으로 처리한다면 원래 OP CODE인 mov eax,1을 복원 할 수 있습니다. 다음은 해당 코드를 복원 한 모습입니다. NOP으로 바꾼뒤 올리에서 Ctrl+a 혹은 마우스 우클릭시 Analysis → Alnalse code 을 클릭하면 OPCODE를 재분석 하게 됩니다

```
00401019: 8D70 C0    lea edi, [local_16]
0040101C: B9 10000000  mov ecx, 10
00401021: B8 CCCCCCCC  mov eax, CCCCCCCC
00401026: F3:A8      rep stos dword ptr es:[edi]
00401028: E9 01000000  jmp SEH_Stru,0040102E
0040102E: 90          nop
0040102F: B8 01000000  mov eax, 1
00401031: 33C0        xor eax, eax
00401035: 5F          pop edi
00401036: 5E          pop esi
00401037: 5B          pop ebx
00401038: B3C4 40     add esp, 40
0040103B: 3BEC        cmp ebp, esp
0040103D: E8 1E000000  call SEH_Stru,_chkesp
00401042: 8BE5        mov esp, ebp
00401044: 5D          pop ebp
00401045: C3          ret
00401046: CC          int3
00401047: CC          int3
00401048: CC          int3
```

그림 54 Junk Code 패치

■ Reference

EDIT PLUS 를 이용한 MASM 환경 구축

<http://mysilpir.net/entry/EditPlus- Assembly- %EC%84%A4%EC%A0%95- MASM>

ANTI REVERSE

<http://zesrever.xstone.org/><http://slaxcore.tistory.com><http://beist.org/research/public/artofunpacking/artofunpacking.pdf><http://openrce.org>

정덕영님의 원도우 구조와 원리

THX to zersrever,slaxcore,ashine,ap0x,정덕영 FROM Hong10

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

0x08. Reversing Malware

악성코드 파일 분석 방법절차

많은 종류의 다양한 악성코드들을 분석하다보면 좀더 편리하고 좀더 빠른 방법을 통한 분석절차에 대해서 고민하게 됩니다. 여러 AV 업체에서는 좀더 빠른 시간 안에 악성코드의 유니크한 패턴을 추출하고 복원하는 방법에 대해서 고민하고, 관리자들의 경우 악성코드 감염시 미치는 영향과 그 대상에 대해 관심을 갖는 경우가 많습니다. 이 문서에서는 어떤 정형화된 절차(예: 외형분석, 정적분석, 동적분석, 상세분석과 같은 절차)가 아닌 다분히 개인적인 편이에 따른 방법을 통해서 악성코드를 분석하도록 하겠습니다.

1 악성코드 분석 환경 구성

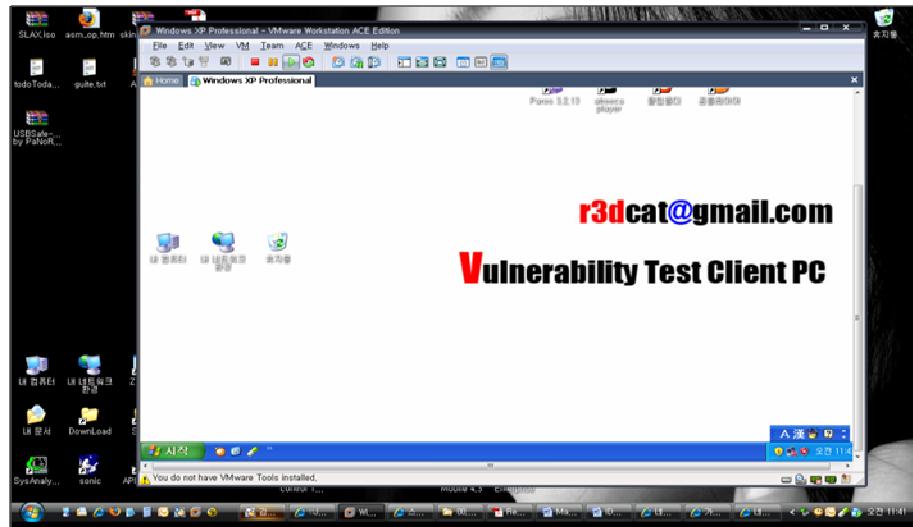
악성코드 분석을 위해서는 분석 작업을 하는 Host 가 감염되지 않고 통제 가능한 환경을 구성하는 것이 중요합니다. 그리고 악성코드의 증상을 분석하기 위해서 각종 모니터링 도구를 통해 악성코드 실행시 레지스트리 정보 수정 및 파일 입출력 정보, 네트워크 연결 정보 등의 로그를 수집해야 합니다.

1.1 가상 환경 구성(**VMware**)

- http://www.vmware.com/download/desktop_virtualization.html

VMWARE 워크스테이션은 EMC 사의 자회사인 VMware에서 제작한 가상화 소프트웨어입니다. 가상 머신(Virtual Machine)을 통해 Host OS(실제 환경)과 Guest OS(가상 환경)을 구분하여 악성코드 실행시 Host OS(실제 환경)에 영향을 받지 않고 분석 할 수 있는 환경을 구성하도록 도와줍니다. 예를 들어 악성코드 분석할 때 snapshot 기능을 이용하여 쉽게 시점별로 복원을 하거나, IRCbot 분석 등을 할 경우 쉽게 감염된 다수의 Guest OS(가상 환경) 들로 Bot NET을 구성 할 수 있습니다.

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|



[그림 55] **VMware** 구동 화면

1.2 SysAnalyzer

- <http://labs.idefense.com/software/malcode.php>

SysAnalyzer 도구는 악성코드가 시스템에서 구동되는 동안에 주어진 시간에 이후에 변경된 정보를 수집, 비교, 분석 보고해 주는 자동화된 툴입니다. SysAnalyzer 의 주된 임무는 지정된 시간에 걸쳐 시스템의 스냅샷을 비교하는 작업을 수행합니다.

- 1) Delay : 스냅샷 전.후 사이의 값 지정
- 2) Sniff Hit : HTTP 접속 및 IRC 접속 정보를 확인
- 3) Api Logger- 분석 바이너리에 인젝션 되는 DLL에서 호출되는 API 목록
- 4) Directory Watcher- 모니터링 시점에 생성되는 모든 파일 확인



[그림 56] **SysAnalyzer** 실행 초기 화면

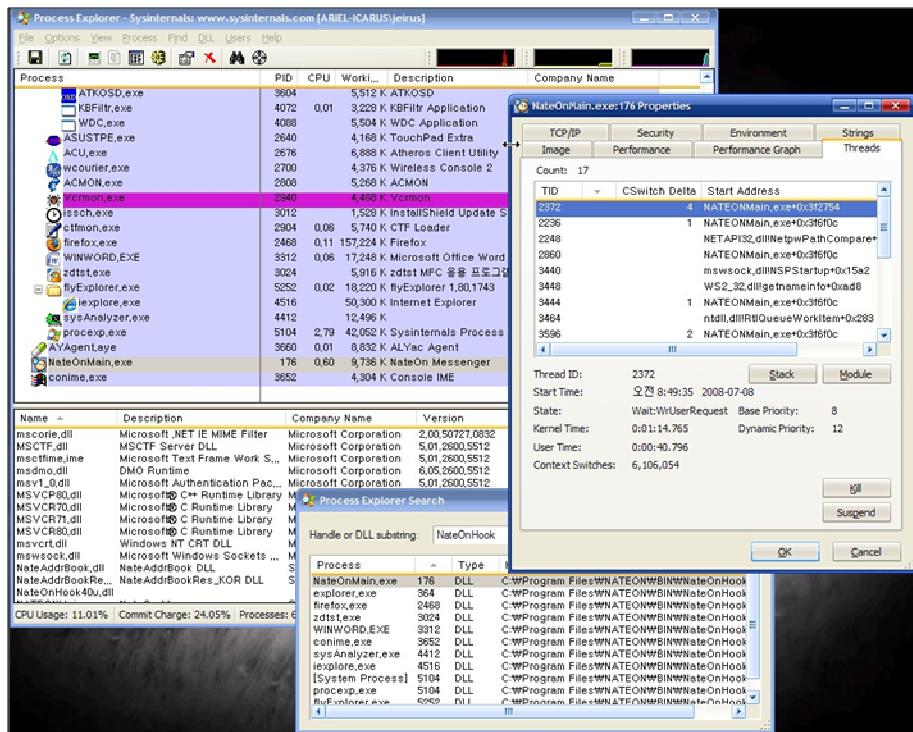
SysAnalyzer 는 비교된 스냅샷을 통해 실행된 악성코드에 다음과 같은 정보를 얻을 수 있습니다.

- 1) 실행된 프로세스
- 2) 악성코드에 의해 오픈된 포트
- 3) explorer.exe 나 Internet Explorer에서 로드된 DLL
- 4) 커널에 로드된 모듈, 변경/생성된 레지스트리 키

1.3 **Sysinternal suit**

➤ [http://technet.microsoft.com/ko-kr/sysinternals/default\(en-us\).aspx](http://technet.microsoft.com/ko-kr/sysinternals/default(en-us).aspx)

작업 관리자에서는 나타나지 않는 프로세스(**Unnamed Process**)까지 볼 수 있으며, 각각의 프로세스가 익은 핸들과 DLL을 모두 확인할 수 있습니다.



[그림 57] Process Explorer 실행 화면

`sysinternals`에서 배포하고 있는 통합 패키지인 `suit`에는 아래와 같은 개별 도구들을 포함하고 있습니다.

| 구분 | 설명 |
|-----------------|---|
| Autoruns | 원도우즈 시작 시 자동으로 실행되는 프로그램 감시 |
| Filemon | 대상 프로그램이 읽거나 쓰는 파일 감시 |
| Regmon | 대상 프로그램이 읽거나 쓰는 레지스트리 감시 |
| RootkitRevealer | 루트킷 탐지 프로그램 |
| Tcpviews | 네트워크를 통해 주고 받는 TCP 패킷 감시 |
| Tdimon | 현재 시스템의 모든 TCP/UDP 입출력 상황 감시 |
| ProcessMonitor | 프로세스 목록 감시(Filemon, Regmon, Process 모니터 통합) |

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

1.4 기타 도구들

위에서 명시한 도구 외에도 아래와 같은 툴들도 많이 사용되고 있습니다. 별도로 살펴보시길 권합니다.

- 1) **Icesword** 숨겨진 프로세스 목록탐지 및 파일, 레지스트리,BHO 등을 볼수 있는 종합툴
- 2) **MultiMon** 위의 모든 대상에 대한 통합 실시간 감시
- 3) **Regshot** 레지스트리 분석 도구(실행 전과 후의 스냅샷 비교)
- 4) **Smartsniff** 간단한 네트워크 스니퍼
- 5) **Winalysis** 위의 모든 대상에 대한 통합 분석(스냅샷 비교)

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

2 Manual UnPacking

악성코드를 분석하기 위한 환경이 모두 구성이 되었으면 이제 악성코드 샘플을 가지고 본격적인 분석작업에 들어가게 됩니다. 그렇지만 악성코드 제작자의 경우 악성코드의 패턴 추출 지연 및 코드분석 지연을 목적으로 해당 악성코드에 대한 분석을 어렵게 하기 위해 안티 디버깅 기법과 실행압축(Packing) 된 형태로 배포 합니다.

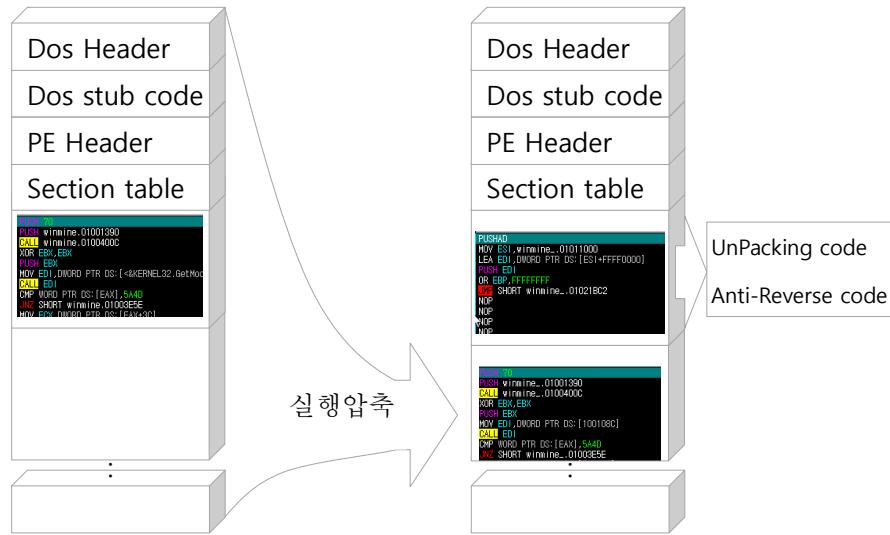
그래서 분석을 위한 첫 번째 단계는 **UnPacking** 부터 시작 됩니다.

2.1 실행압축(Packing)이란?

Packing은 단어적 의미 그대로 ‘포장’, ‘짐꾸리기’라는 뜻으로 사용됩니다. 초기의 실행압축 기술은 컴퓨터가 플로피 디스크나 적은 용량의 하드드라이브가 쓰일 당시에 실행파일의 용량을 줄이는 목적으로 사용되었습니다.

우선 배포시 플로피 디스크 1 장에 배포가 가능하도록 사이즈를 줄이고, 압축 파일의 별도의 해제 과정 없이 바로 실행이 가능하도록 할 필요가 있었습니다.

그러나 현재는 주로 역공학(Reverse Engineering: 이하 리버싱)으로부터 코드를 보호할 목적으로 사용되고 있습니다. 그러면 어떻게 리버싱으로부터 코드를 보호 할 수 있을까요? 압축을 한 파일을 살펴보면 원본 코드 소스가 마치 ‘포장’ 되듯이 다른 코드소스(패커 소스)의 데이터 부분에 감싸져있는 것을 알 수 있습니다. 예를 들어 설명하자면 노트패드를 패킹 할 경우 원래 노트패드의 코드 소스는 암호화되어 새로운 파일의 데이터 영역에 저장 됩니다. 이렇게 패킹 된 프로그램은 전체적으로 패킹 되기 전과는 다른 모습으로 저장되어 있습니다. 그리고 패킹된 프로그램을 실행하면 노트패드가 실행되기 전 패커의 언패킹 코드가 먼저 실행되어 데이터 영역에 암호화 되어 저장된 코드를 원래의 노트패드 코드로 복호화하여 메모리에 로딩합니다. 그렇게 하여 패킹된 노트패드 역시 기존의 노트패드와 같은 코드로 실행 되는 것입니다. 이렇게 패킹 되어진 노트패드의 첫번째 코드 실행 위치와 같은 원본 소스의 진입지점을 OEP(Original Entry Point)라고 부릅니다. 이 정보는 다음에서 살펴 볼 MUP에서 중요한 정보로 활용 됩니다.



[그림 58] Packing 된 실행파일 형태

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

2.2 OEP(*Original Entry Point*) 유형

위에서 살펴 보았듯이 OEP란 패킹된 프로그램의 실제 코드 시작부분입니다.

| 구 분 | OEP 형태 |
|-------------------|---|
| Visual Basic | PUSH VB- Crack.00405C78 CALL <JMP.&MSVBVM50.# 100> |
| Visual C++ 6.0 | PUSH EBP MOV EBP,ESP SUB ESP,44 PUSH ESI CALL NEAR DWORD PTR DS:<&KERNEL32.GetCommandLineA> MOV ESI,EAX |
| Visual C++ 7.0 | PUSH 70 PUSH notepad.01001898 CALL notepad.01007568 |
| Borland Delphi | PUSH EBP MOV EBP,ESP ADD ESP,- 10 MOV EAX,Test_.004529A8 CALL Test_.00406578 |
| MASM | PUSH 0 CALL <JMP.&kernel32.GetModuleHandleA> |

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

2.3 Stack을 이용한 packer의 mup 방법 (pusha/popa)

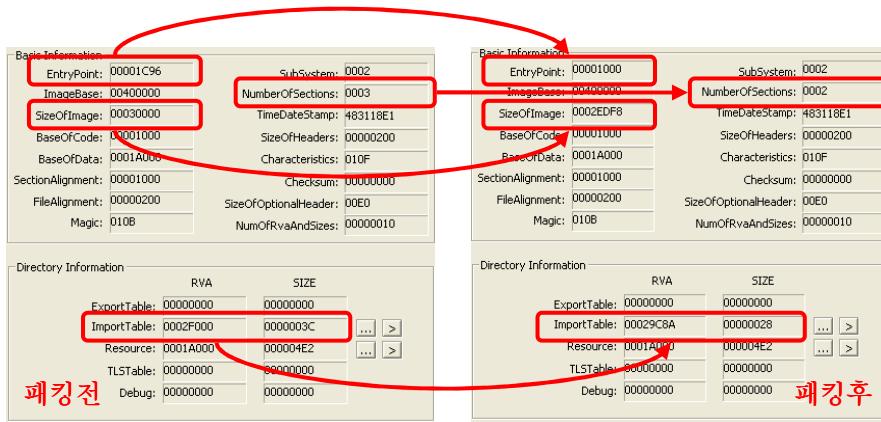
1) MUP(Manual UnPacking) 란?

위에서 살펴 보았듯이 악성코드 개발자들은 분석을 어렵게 하기 위해서 원본 소스 파일을 패킹 합니다. 그래서 분석을 용이하도록 하기 위해서는 해당 실행파일의 실행압축(Packing)을 해제할 필요가 있습니다. 그렇지만 실행압축 기술을 사용하는 수많은 Packer 프로그램이 존재하고 모든 패커에 대한 언패커가 존재하지는 않기 때문에 수 작업을 통한 MUP(Manual UnPacking) 작업이 필요하게 됩니다.

2) MUP 절차

패커는 패킹 과정에서 언패킹 코드를 삽입 한 후 진입지점(Entry Point)를 수정하여 프로그램 시작전 언패킹 코드를 우선 실행하도록 코드 흐름을 변경 합니다. 이 과정에서 아래와 같은 필드 값 역시 변경될 수 있습니다.

| 필드 | 변경 형태 |
|--------------------------|--------------------------------------|
| AddressOfEntryPoint | 언팩킹을 위한 해제 루틴으로 EP(entry point)가 변경됨 |
| NumberOfSection | 섹션의 수가 증가 할 수 있음 |
| SizeOfImage | 메모리에 불러들어오는 사이즈의 값이 변경됨 |
| SizeOfHeaders | 헤더들의 사이즈 줄어들 수 있음 |
| FileAlignment | 파일상에서의 섹션의 배치간격이 변경(줄어들 수 있음) |
| Import Table | 위치가 변경되고, 사이즈는 줄어들 수 있음 |
| Relocation Table | 재배치 정보가 추가됨 |
| Load Configuration Table | 정보가 사라질 수 있음 |
| Import Address Table | 정보가 사라질 수 있음 |

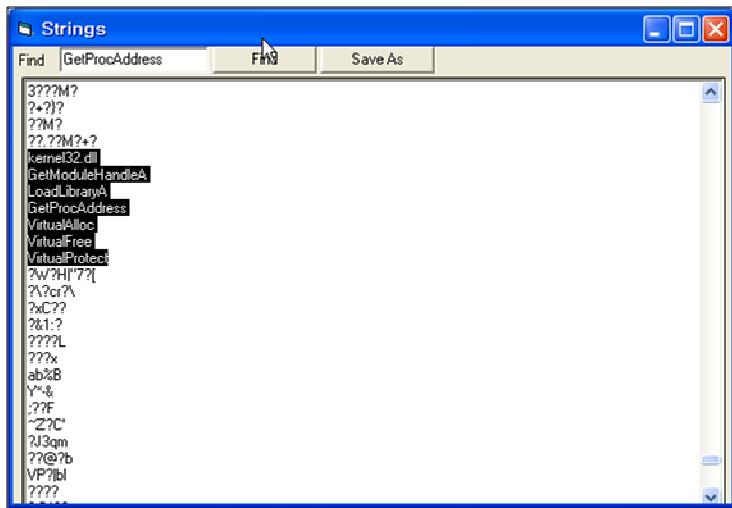
[그림 59] **Packing** 된 실행파일 필드 정보 값을 **peid**로 변경 내역 확인

변경된 필드 값을 살펴보았습니다. 이제는 수동 실행 압축 해제를 하기 위해서 다음과 같은 순서로 진행하도록 하겠습니다.

1. 어떤 실행압축 프로그램(Packer)을 사용하였는지 확인 (패킹 여부 확인)
2. OEP(Original Entry Point)를 찾고 실행압축이 해제된 코드를 Dump
 - ✓ SEH 예외처리 이용방법
 - ✓ Code Section 접근 방법 확인
 - ✓ Stack Memory 를 통한 Register 복원 방법 등등
3. Dump 한 코드 파일의 OEP(Original Entry Point)를 수정
4. IAT(Import Address Table) 복원

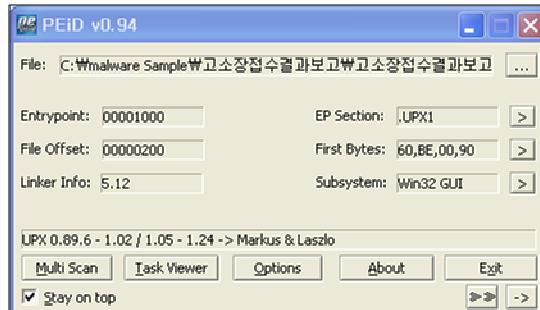
3) 실행압축 프로그램 확인

먼저 해당 악성코드가 실제로 패킹 되었는지를 확인 해봅니다.

[그림 60] **Strings**를 이용한 패커 여부 추정

악성코드 파일의 **Strings**을 분석한 결과 **VirtualAlloc()**, **GetModuleHandle()**, **LoadLibraryA()**를 사용하는 것으로 미루어보아 패킹된 것을 짐작할 수 있습니다.

어떤 Packer 프로그램을 사용하였는지 한번 살펴보도록 하겠습니다. 여기서는 PEid 프로그램을 이용하여 해당 정보를 알아보겠습니다.

[그림 61] **PEID**를 이용한 패커 정보 확인

PEid에서 볼 수 있듯이 UPX 0.89.6 Packer 프로그램을 사용한 것을 알 수 있습니다.

보통 Packing이 이루어지기 전 실제 바이너리 코드를 Packing 된 바이너리 코드의 어떤 영역(일명 Section)에 압축 혹은 암호화해서 저장합니다. 그리고 나서 Entry Point를 실행압축된 코드를 해제하기 위한 주소로 바꾸고 실행 시에는 실행압축된 코드가 압축해제되어 실제 코드가 실행될 수 있도록

UnPacking 하는 코드가 포함됩니다. 결국은 실제코드가 Packing 되어 있더라도 실행 시에는 반드시 Unpacking 된 후에 실행이 되어야 합니다. 그렇기 때문에 실제 코드가 UnPacking 되는 시점과 위치를 파악하면 UnPacking 된 원본코드를 찾을 수 있습니다. 그럼 다음에서는 실제 코드를 획득하기 위해 OEP(Original Entry Point)를 찾고 해당 코드에 대하여 Dump 를 수행해보도록 하겠습니다.

4) **OEP(Original Entry Point)** 찾기

OEP 라는 항은 실행 시 Packing 된 바이너리 코드가 모두 압축해제 된 후 원래의 실제 코드가 수행되기 위한 시작점입니다. 일단 이 시작점을 찾기 위해 우리는 ollyDBG 를 사용하여 해당 악성코드의 디버깅 해보도록 하겠습니다. 먼저 악성코드를 디버거를 통하여 열었더니 다음과 같은 메시지가 출력되었습니다.



[그림 62] ollydbg의 경고 메세지창

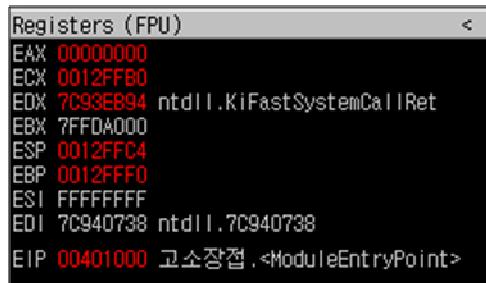
이와 같이 창이 나타나는 이유는 Entry Point 가 실제 실행되는 영역 밖에 있기 때문에 압축되어 있거나 혹은 변조되어 있을 수 있다는 것을 알려주기 위해 출력하는 메시지창 입니다. 확인 버튼을 누르면 실제 어셈블리 코드가 나타나게 됩니다. 우리가 유심히 살펴봐야 할 것은 맨 첫 줄에 나타난 코드입니다.

| Address | Hex dump | Disassembly | Comment |
|----------|-----------------|-------------------------------------|---------|
| 00401000 | \$ 60 | PUSHAD | |
| 00401001 | . BE 00908B00 | Mov ESI, 8B9000 | |
| 00401006 | . 80BE 008084F1 | LEA EDI,DWORD PTR DS:[ESI+FFB48000] | |
| 0040100C | . 57 | PUSH EDI | |
| 0040100D | . 83CD FF | OR EBP, FFFFFFFFFF | |
| 00401010 | .~ EB 3A | JMP SHORT 고소장점.0040104C | |

[그림 63] 첫번째 PUSHAD(악성코드의 진입지점)

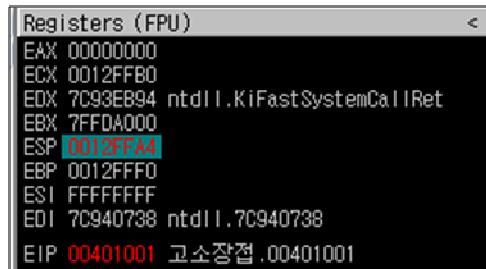
'PUSHAD' 라는 명령어가 보입니다. 이 명령어는 현재의 실행 환경 값을 백업하는 명령입니다. 모든 레지스터에 저장된 값을 Stack 에 모두 저장합니다. 그리고 이후 UnPacking 을 수행하고 나서 원래 실행 파일이 실행되기 바로 직전 실행 환경 값을 다시 복원하기 위해서 Stack 에 저장된 레지스터 값을 'POPAD' 명령로 복원하게 됩니다.

아래의 레지스터는 맨 처음 어셈블리 코드가 열렸을 때 값입니다. F7 를 눌러 보겠습니다.



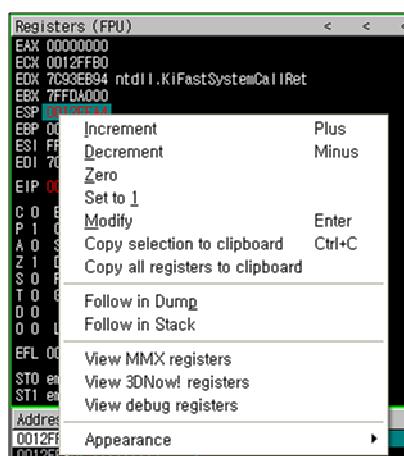
[그림 64] EP에서의 레지스터 값

F7 를 눌러 보겠습니다. 그러면 아래와 같이 ESP 스택 포인터 값이 '0012FFCA' 에서 '0012FFA4'로 변경됨을 알 수 있습니다.



[그림 65] ESP 레지스터 값 세팅

이제는 아래와 같이 해당 ESP 레지스터가 가리키고 있는 영역을 값을 살펴보도록 하겠습니다.



[그림 66] ESP 주소에 대한 dump 출력

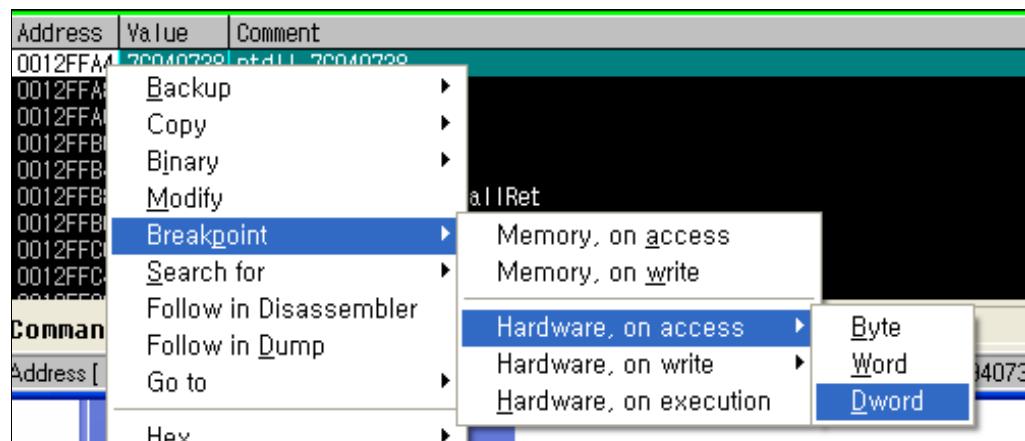
ESP 레지스터 값을 가리킨 후 ‘Follow in Dump’ 명령어를 수행하면 해당 ESP 레지스터에 저장된 값이 가리키고 있는 영역에 저장된 값이 아래와 같이 나타납니다.

The screenshot shows two windows side-by-side. On the left is the 'Registers (FPU)' window, which lists CPU registers (EAX, ECX, EDX, etc.) with their current values. Several registers are highlighted in red: EAX, ECX, EDX, EBP, and EIP. On the right is a table titled 'Stack dump' showing memory addresses and their corresponding values. The first few rows show the stack frame, with EBP pointing to the base of the stack (0012FFC4). The EIP register value (00401001) is also present in the stack dump table under address 0012FFC4. Red boxes highlight the EIP entry in both the registers and the stack dump table.

| Address | Value | Comment |
|----------|--------------|------------------------------|
| 0012FFC4 | EAX 7C940738 | ntdll!_7C940738 |
| 0012FFC5 | ECX FFFFFFFF | |
| 0012FFC6 | EDX 7C93EB94 | ntdll!_KiFastSystemCall!Ret |
| 0012FFC7 | EBP 0012FFB0 | |
| 0012FFC8 | 0012FFC4 | |
| 0012FFC9 | 0012FFB4 | 7FFDA000 |
| 0012FFCA | EDX 7C93EB94 | ntdll!_KiFastSystemCall!Ret |
| 0012FFCB | ECX 0012FFB0 | |
| 0012FFCC | EAX 00000000 | |
| 0012FFCD | 7C81604F | RETURN to kernel!32.7081604F |

[그림 67] Stack에 저장된 레지스터 값

각 레지스터의 값들이 Stack에 저장되어 있는 것을 확인 할 수 있습니다. 저장된 레지스터 값은 UnPacking 된 후 실제 코드가 실행되기 전에 복원을 해야하기 때문에 저장된 레지스터의 위치에 Stack 포인터가 접근하기 하는 순간에 브레이크 포인트를 걸어보도록 하겠습니다. 아래와 같이 처음 4byte를 선택한 후 마우스 우클릭하여 하드웨어 메모리 브레이크 포인트를 설정합니다.



[그림 68] H/W BP 설정

그리고 나서 F9 키를 눌러서 디버깅을 진행하도록 합니다. 그러면 다음의 위치에서 브레이크 포인트가 걸리게 됩니다.

The screenshot shows the assembly dump window with three entries. The first entry is a NOP instruction (90). The second entry is a jump instruction (JMP) to address 004291A3. The third entry is a DB instruction (00). The second entry (JMP) is highlighted in red, indicating it is the point where the hardware breakpoint was triggered.

| Address | Hex dump | Disassembly | Comment |
|----------|----------------|-------------------|---------|
| 0040104E | . 90 | NOP | |
| 0040104F | .- E9 4F810200 | JMP 고소장접.004291A3 | |
| 00401054 | 00 | DB 00 | |

[그림 69] 첫번째 Bp

004291A3 으로 Jump 하는 구문이 있는 위치에서 디버깅이 멈추었습니다. ‘POPAD’ 명령어를 통해 레지스터가 복원된 이 후 이기 때문에 다음 위치로 Jump 하면 바로 OEP 임을 예상할 수 있습니다. F8 키를 눌러서 계속 디버깅 해보도록 하겠습니다.

| Address | Hex dump | Disassembly | Comment |
|----------|-------------|-----------------------------|---------|
| 004291A3 | 60 | PUSHAD | |
| 004291A4 | E8 00000000 | CALL 고소장점.004291A9 | |
| 004291A9 | 83C4 04 | ADD ESP,4 | |
| 004291AC | 8B6C24 FC | MOV EBP,WORD PTR SS:[ESP-4] | |

[그림 70] 두번째 pushad

그렇지만 우리가 앞서 살펴본 OEP의 형태와는 다른 모습을 보임을 알 수 있습니다. 다시 ‘PUSHAD’를 통해서 레지스터 정보를 저장함을 알 수 있습니다. 다시 F9로 진행을 해보도록 하겠습니다.

| Address | Hex dump | Disassembly | Comment |
|----------|---------------|-------------------|---------|
| 0042946A | - E9 2788F0FF | JMP 고소장점.00401C96 | |
| 0042946F | 60 | PUSHAD | |
| 00429470 | 6A 40 | PUSH 40 | |
| 00429472 | 68 00100000 | PUSH 1000 | |

[그림 71] 두번째 Bp

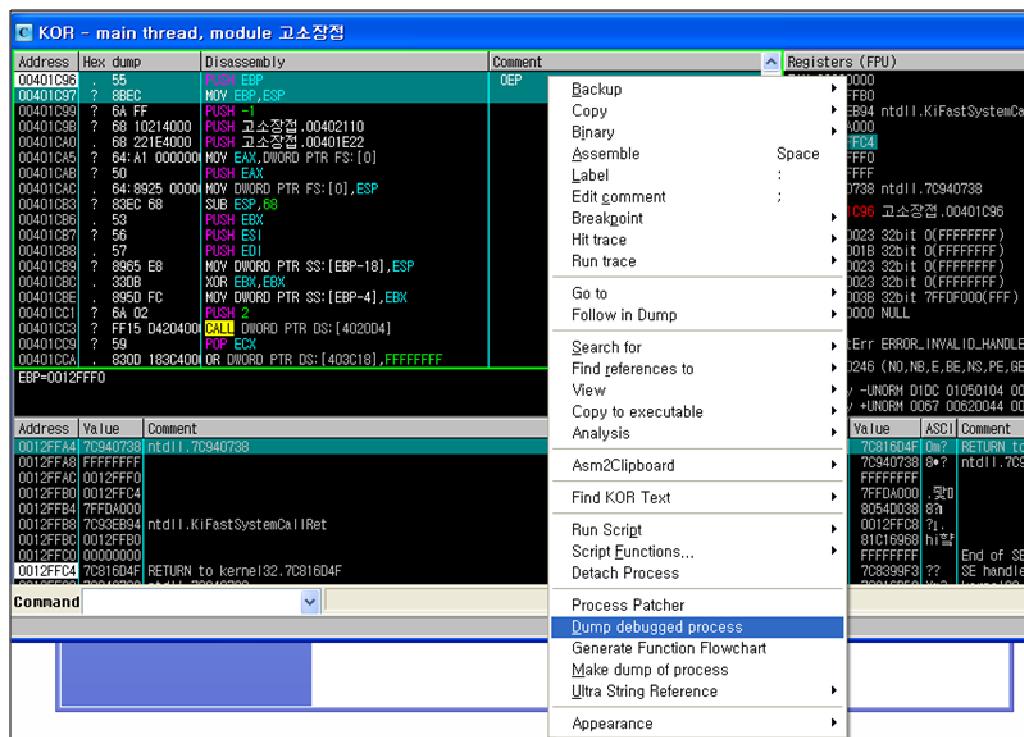
다시 ‘POPAD’ 명령어로 메모리 레지스터가 복원된 후에서 Bp 가 걸림을 확인 했습니다. 한 스텝을 더 진행을 해보도록 하겠습니다.

| Address | Hex dump | Disassembly | Comment |
|----------|----------------|--------------------------|---------|
| 00401C96 | . 55 | PUSH EBP | OEP |
| 00401C97 | ? 88EC | MOV EBP,ESP | |
| 00401C99 | ? 6A FF | PUSH -1 | |
| 00401C9B | ? 68 10214000 | PUSH 고소장점.00402110 | |
| 00401CA0 | . 68 221E4000 | PUSH 고소장점.00401E22 | |
| 00401CA5 | ? 64:A1 000000 | MOV EAX,WORD PTR FS:[0] | |
| 00401CAB | ? 50 | PUSH EAX | |
| 00401CAC | . 64:8925 0000 | MOV DWORD PTR FS:[0],ESP | |

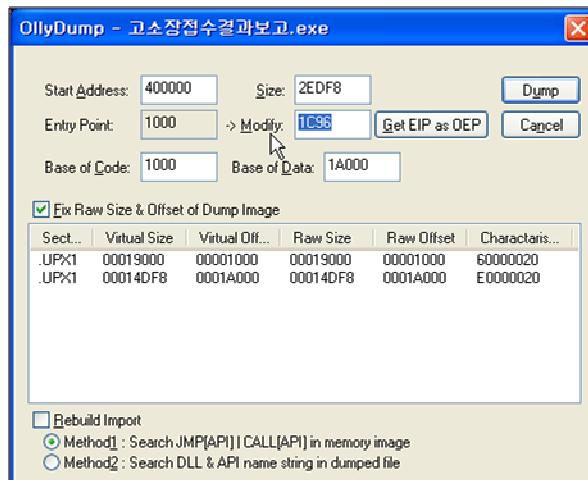
[그림 72] OEP code

실제 코드가 수행될 엔트리 포인트 위치로 이동하였습니다. 이 종으로 같은 패킹된 구조였기 때문에 unPacker를 이용한 unPacking이 이루어지지 않은 것입니다.

여기서부터 원본 코드에 대한 DUMP를 수행합니다. 해당 OEP에서 마우스 우클릭 후 ollyDump 플러그인 프로그램을 이용하여 아래와 같이 DUMP를 수행합니다.

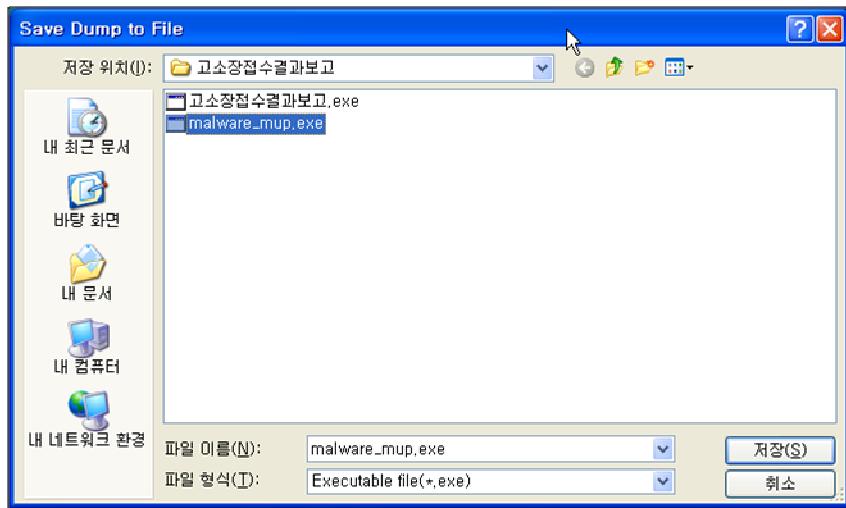


[그림 73] ollyDump plug 실행①



[그림 74] ollyDump plug 실행②

OEP 값인 '1C96'을 따로 기록 한 후 Rebuild Import 메뉴는 체크를 해제하고 Dump 를 실행 합니다.

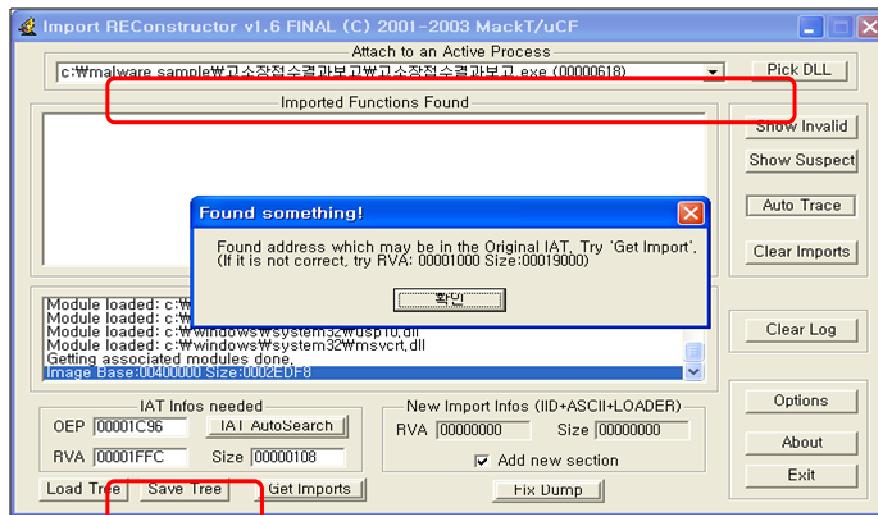


[그림 75] malware_mup로 dump 저장

5) IAT(Import Address Table) 복원

현재 원본 code에 대한 dump를 마쳤지만 IAT(Import Address Table: 이하 IAT)에 대한 정보를 생성해주지 않아 해당 파일은 실행이 되지 않습니다. IAT는 프로그램이 외부 함수를 사용하려 할 때 Windows는 메모리 주소 공간으로 함수가 들어있는 DLL을 로드하고나서 원하는 함수에 대한 코드 위치를 IAT에 전달합니다. 그렇기 때문에 IAT가 제대로 생성되지 않았을 경우 해당 함수를 로딩할 수 없어 에러가 발생 합니다. 그렇기 때문에 IAT에 대한 복원작업을 진행해야 합니다.

ImpREC를 이용하여 현재 ollyDBG로 실행되고 있는 프로세스를 Attach합니다. 그리고 사전에 적어둔 OEP 위치 정보를 OEP에 입력합니다.



[그림 76] IAT 복원작업①

ollyDBG 로 IAT 의 정보를 확인해보도록 하겠습니다.

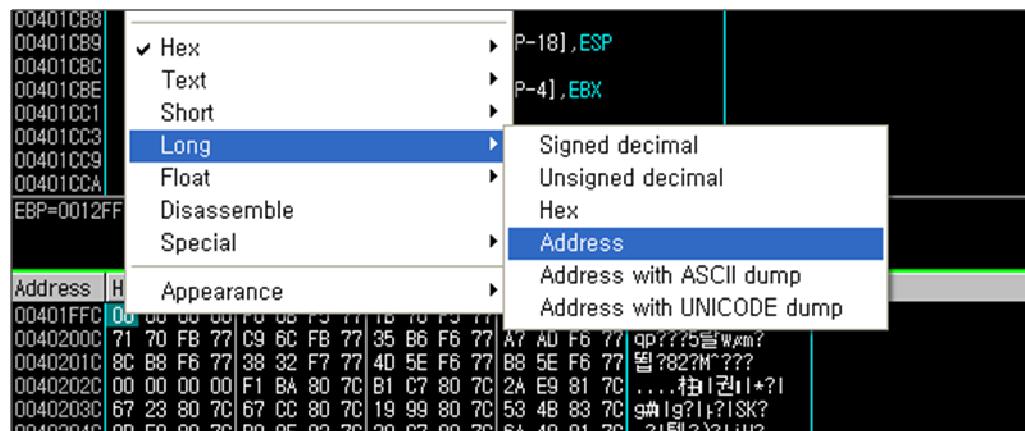
우선 dump 창에서 ‘ctrl+g’ 핫키를 사용하여 IAT 의 RVA 위치로 이동합니다.

이미지 베이스 값이 400000 이기 때문에 401FFC 로 이동 합니다.



[그림 77] IAT 내용 확인①

이동한 주소의 dump 창에서 오른쪽 버튼을 클릭 한 후 Long- >Address 로 정보를 쉽게 볼 수 있도록 설정 합니다.



[그림 78] IAT 내용 확인②

Address 형으로 Dump 창을 재정렬한 후 내용을 살펴보면 아래와 같이 Import 되는 함수 주소들이 기록되어 있음을 알 수 있습니다.

| Address | Value | Comment |
|----------|----------|-------------------------------|
| 00401FFC | 00000000 | |
| 00402000 | 77F56BF0 | ADVAPI32.RegCloseKey |
| 00402004 | 77F5761B | ADVAPI32.RegOpenKeyExA |
| 00402008 | 77F5EBE7 | ADVAPI32.RegSetValueExA |
| 0040200C | 77FB7071 | ADVAPI32.CreateServiceA |
| 00402010 | 77FB6CC9 | ADVAPI32.ChangeServiceConfigA |
| 00402014 | 77F6B635 | ADVAPI32.ControlService |
| 00402018 | 77F6A0A7 | ADVAPI32.OpenSCManagerA |
| 0040201C | 77F6B88C | ADVAPI32.OpenServiceA |
| 00402020 | 77F73238 | ADVAPI32.StartServiceA |
| 00402024 | 77F65E40 | ADVAPI32.CloseServiceHandle |
| 00402028 | 77F65E88 | ADVAPI32.QueryServiceStatus |

Command

Address [401FFC ~ 401FFF Range: 4] Byte [H: 0 D: 0] Word [H: 00 D: 0] DWord [H: 0000 D: 0]

[그림 79] IAT 내용 확인③

IAT 의 시작지점부터 마지막 함수 호출 지점까지의 사이즈를 살펴보면 아래와 같이 108 인 것을 알 수 있습니다.

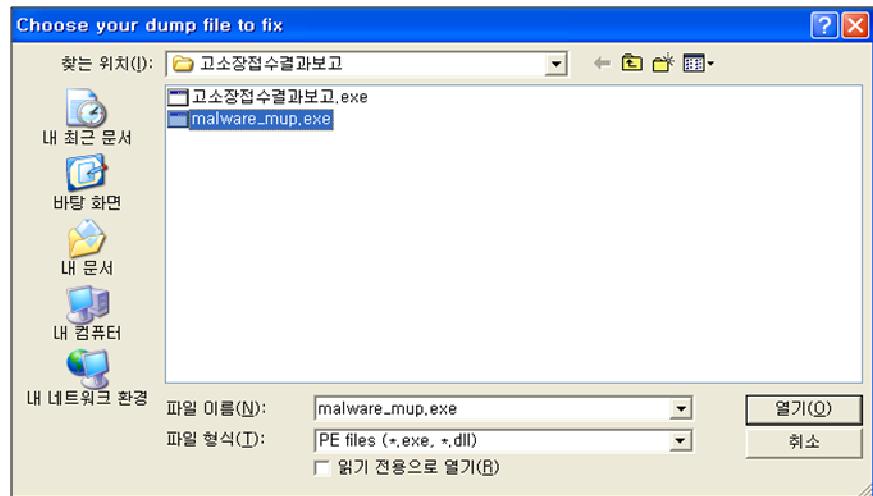
| Address | Value | Comment |
|----------|----------|---------------------------|
| 0040208C | 7C940331 | ntdll.RtlGetLastError |
| 00402090 | 7C801EEE | kernel32.GetStartupInfoA |
| 00402094 | 7C80B529 | kernel32.GetModuleHandleA |
| 00402098 | 7C802442 | kernel32.Sleep |
| 0040209C | 7C80AC28 | kernel32GetProcAddress |
| 004020A0 | 7C801D4F | kernel32.LoadLibraryExA |
| 004020A4 | 7C80FE2F | kernel32.GlobalFree |
| 004020A8 | 7C80AA66 | kernel32.FreeLibrary |
| 004020AC | 7C801625 | kernel32.DeviceIoControl |
| 004020B0 | 00000000 | |
| 004020B4 | 77BE9E7E | |
| 004020B8 | 77C117AC | |

Command

Address [401FFC ~ 4020B3] Range [B8]

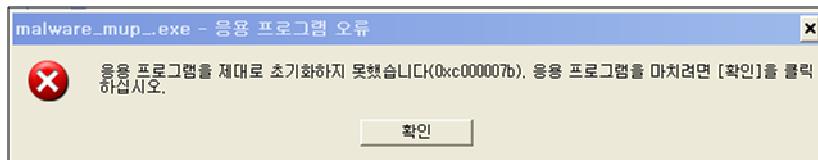
[그림 80] IAT 내용 확인④

OllyDBG 를 통한 확인을 마치고 다시 ImpREC 의 ‘Fix Dump’를 이용해서 악성 코드의 IAT 를 복원하도록 합니다.



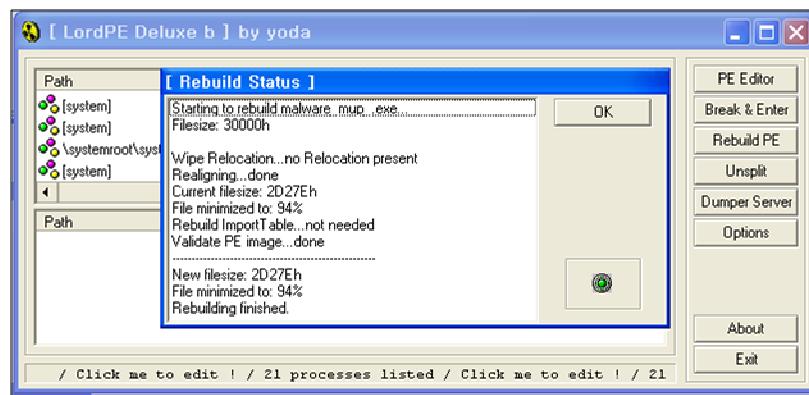
[그림 81] IAT 복원작업②

ImpREC 를 통해 IAT 를 복원하였지만 IAT 전체에 대한 복원이 원활하지 않아 실행이 되지 않음을 알 수 있습니다.



[그림 82] IAT 복원작업③

LordPE 의 ‘Rebuild PE’ 기능을 이용하여 IAT 를 재복원 합니다.



[그림 83] IAT 복원작업④

복원을 통해 MUP 작업이 모두 완료 되었습니다. 이제 내부의 문자열을 통해서 악성코드에서 사용하는 API 값과 ASCII 값으로 정적 분석을 진행 하도록 하겠습니다.

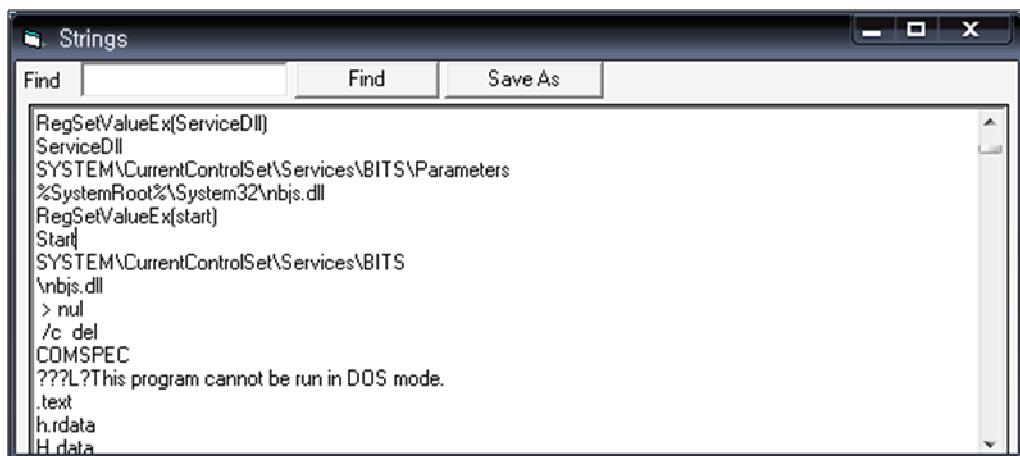
3 Embed string Analysis

표준 출력 명령어인 strings 를 이용하여 바이너리 파일 내부에서 출력 가능한 문자열만을 추출할 수 있습니다. 악성코드의 내부 문자열을 통해서 사용하는 API 와 하드코딩된 문자열로 악성코드의 기능을 유출해 볼 수 있습니다. 아래는 악성코드에 자주 사용되는 API 목록입니다.

3.1 악성코드에서 자주 사용하는 API 목록

| API | 설명 |
|----------------------|---|
| CreateFile() | 파일 생성 및 Open |
| ReadFile() | 파일 내용을 읽을 때 사용 |
| WriteFile() | 파일에 내용을 쓸 때 사용 |
| GetSystemDirectory() | 시스템 디렉토리 경로를 얻어올 때 사용 |
| RegCreateKey() | 레지스트리 키 생성에 사용 |
| RegDeleteKey() | 레지스트리 키 삭제에 사용 |
| RegOpenKey() | 레지스트리 키를 열 때 사용 |
| RegCloseKey() | 열려진 레지스트리 키를 닫을 때 사용 |
| RegSetValueEX() | 열려있는 레지스트리 키(hKey)를 이용하여 lpValueName 에 명시된 항목 이름의 데이터 형식이나 내용을 설정하는데 사용 |

3.2 Embed string 값을 이용한 분석방법



[그림 84] *malware_mup.exe*의 내부 문자열

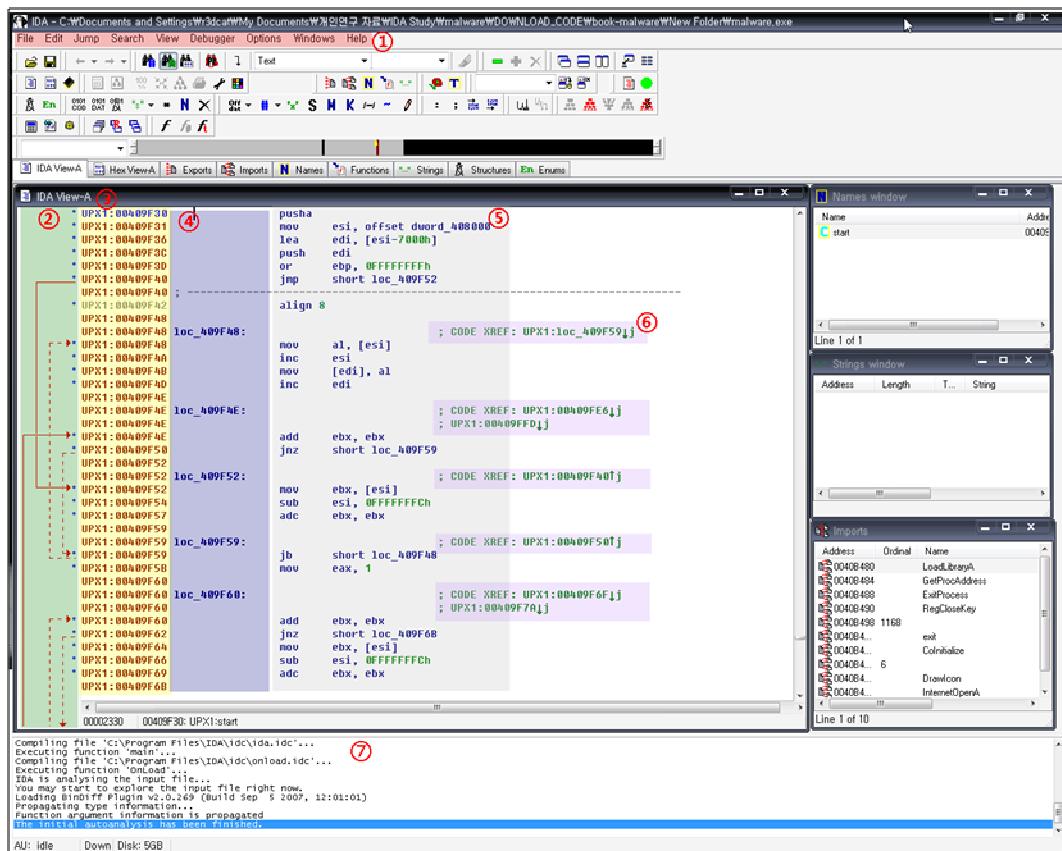
| 문자열 | 설명 |
|---|--|
| RegSetValueEx(ServiceDll) SYSTEM\ CurrentControlSet\ Services\ BITS\ Parameters %SystemRoot%\ System32\ nbjs.dll | nbjs.dll 악성코드를 BITS(Background Intelligent Transfer Service)에 등록 정상파일 qmqr.dll 을 악성파일 nbjs.dll 교체함 ※ BITS: 유형 상태의 네트워크 대역폭을 사용하여 백그라운드에 있는 파일을 전송하는 서비스 |
| RegSetValueEx(start) SYSTEM\ CurrentControlSet\ Services\ BITS \ nbjs.dll | 서비스 시작 유형을 자동으로 설정하여 재부팅 시 악성코드가 실행되도록 변경 |
| /c del > nul | 특정 파일을 유저 몰래 삭제 |
| Start beep service ok beep | Beep 서비스를 시작함. Beep 서비스 역시 악성코드로 교체된 것으로 추정됨 |
| darkshell \ darkshell.sys | Darkshell.sys 파일을 생성하는 것으로 추정됨 |
| InternetReadFile InternetCloseHandle InternetOpenUrlA InternetOpenA wininet.dll | 특정 인터넷 통신을 통해 파일을 불러오는 것으로 보임 |
| OLLYDBG ntdll.dll FileMonClass 18467- 41 \ \ .\ SICE \ \ .\ SIWVID \ \ .\ NTICE \ \ .\ REGSYS \ \ .\ REGVXG | 악티 디버깅 모듈로 의심되는 구문이 존재함. 모니터링 프로그램이나 디버깅 프로그램이 검출 되면 에러메세지를 출력하고 종료되는 구문으로 추정됨 |

\ \ .\ FILEVXG
\ \ .\ FILEM
\ \ .\ TRW
\ \ .\ ICEEXT
Debugger detected - please close it down and restart!
Windows NT users: Please note that having the WinIce/SoftIce service installed means that you are running a debugger!
Monitor detected - please close it down and restart!
Windows NT users: Please note that having the FileMon/RegMon service installed means that you are running a monitor!

4 간단한 **IDA Pro** 사용법

앞서 우리는 간단하게 악성코드인 고소장접수결과.exe 파일의 내부 문자열을 통해 그 기능을 간단하게 유추해보았습니다. 이제부터는 본격적으로 IDA 와 OllyDBG 를 이용해서 분석을 하는데 앞서 간단하게 IDA 의 각 기능에 대해서 살펴본 후 분석을 하도록 하겠습니다.

4.1 The Main Window



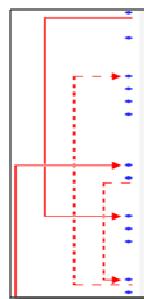
[그림 85] IDA Main Window

- ① IDA 의 메뉴바로 IDA 에 구성된 대부분의 기능들이 있습니다.



[그림 86] 메뉴바 영역

- ② 분기문에 의한 점프 위치를 나타내는 화살표 영역
함수의 분기점이나 코드의 Jump 구문에 의해 이동해지는 위치를 화살표를 통해 표시해주고 있습니다.



[그림 87] 화살표 영역

- ③ PE 파일의 섹션 이름에 따른 Virtual address 영역
이 주소는 디버깅을 진행 할 때 보여지는 주소값과 같습니다.

| |
|---------------|
| UPX1:00409F30 |
| UPX1:00409F31 |
| UPX1:00409F36 |
| UPX1:00409F3C |
| UPX1:00409F3D |

[그림 88] Virtual address 영역

- ④ 코드 로케이션 영역
코드에서 Jump 문에 의해 분기할 때 해당 로케이션으로 이동 합니다.

loc_409F48:

[그림 89] 코드 로케이션 영역

- ⑤ 역어셈블링 코드 영역
분석할 대상 프로그램의 역어셈블링 코드입니다.

```

pusha
mov    esi, offset dword_408000
lea    edi, [esi-7000h]
push   edi
or    ebp, 0FFFFFFFh
jmp   short loc_409F52

align 8

        ; CODE XREF: UPX1:loc_409F59↓j
mov    al, [esi]
inc    esi
mov    [edi], al
inc    edi

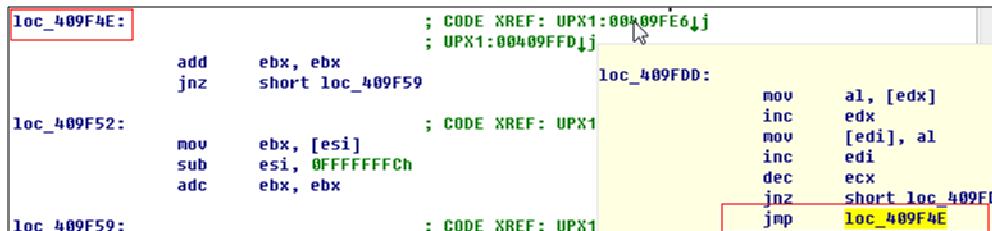
        ; CODE XREF: UPX1:00409FE6↓j
        ; UPX1:00409FFD↓j
add   ebx, ebx
jnz   short loc_409F59

```

[그림 90] 역어셈블링 코드 역역

⑥ 코드 레퍼렌시 영역

프로그램의 흐름 중 분기문에 의해서 해당 부분(loc_409F4E)을 참조한 위치 주소 값(loc_409FDD)을 표시 합니다. 더블 클릭으로 이동하거나 마우스 휠 버튼의 스크롤을 통해서 해당 부분을 살펴 볼 수 있습니다.



[그림 91] 코드 레퍼렌시 영역

⑦ 상태 및 로그 영역

이 상태 윈도우는 IDA의 마지막 행동 및 진행상황을 출력합니다.

```

Compiling file 'C:\Program Files\IDA\idc\ida.idc'...
Executing function 'main'...
Compiling file 'C:\Program Files\IDA\idc\onload.idc'...
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
Loading Bindiff Plugin v2.0.269 (Build Sep 5 2007, 12:01:01)
Propagating type information...
Function argument information is propagated
The initial autoanalysis has been finished.

```

[그림 92] 상태 및 로그 영역

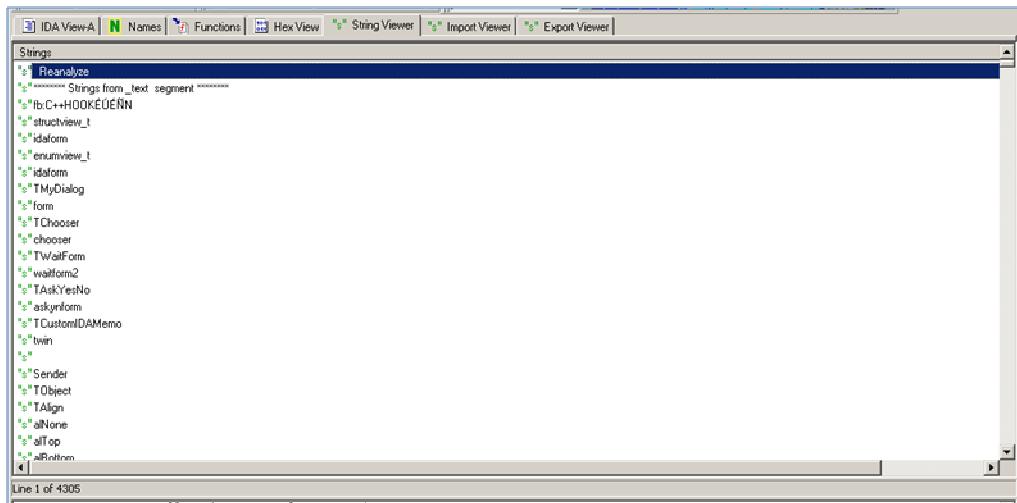
4.2 Name Tag

IDA로 해당 파일에서 찾을 수 있는 모든 이름 정보를 출력하고 있으며, Enter 혹은 더블클릭을 통해 해당 위치로 바로 이동이 가능합니다.

| ICON | L | 내 용 | 예 제 |
|------|-----------------|---------------|------------------------------------|
| L | .text:00401B5C | 라이브러리 함수 | L __GetExceptDLLInfo |
| F | .text:00401F03 | 일반 함수 | F WinMain |
| C | .text:004016BB | 명령 | C __getInstance |
| A | .data:005A8748 | ascii 문자열 | A aFbCHooksren |
| D | .text:004FD3FC | 데이터 | D `__tpdsc__[System::AnsiString *] |
| I | .idata:005D8D38 | imported name | I __imp_WinHelpA |

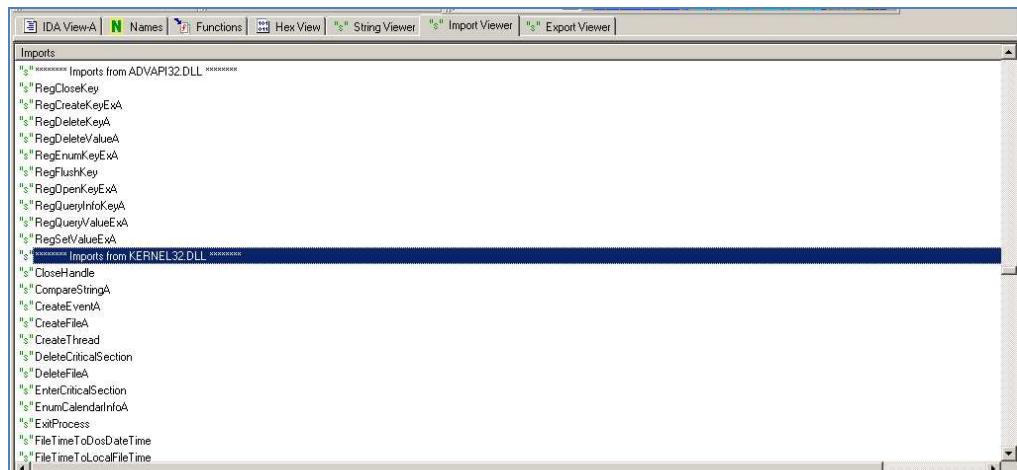
4.3 Strings Tag

IDA로 해당 파일에서 찾을 수 있는 모든 문자열을 출력해주며, Enter 혹은 더블클릭을 통해 해당 위치로 바로 이동이 가능합니다.



[그림 93] 내부 문자열 출력창

4.4 Import Viewer Tag



[그림 94] Import Viewer 출력창

4.5 Cross-reference Tag

| Direction | Type | Address | Instruction | xrefs to sub_403428 |
|-----------|------|----------------|----------------------------------|---------------------|
| Up/Down | p | .text:00404791 | call sub_403428 ; Call Procedure | |
| Up/Down | p | .text:004047C5 | call sub_403428 ; Call Procedure | |
| Up/Down | p | .text:004047F9 | call sub_403428 ; Call Procedure | |

[그림 95] Cross-reference 출력창

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

4.6 Functions Tag

| IDA View-A Hex View N Names Functions Strings Structures Enums | | | | | | |
|--|---------|----------|----------|---------------|---|---|
| Function name | Segment | Start | Length | R | F | L |
| Graphics::TIcon::ImageNeeded(void) | .text | 0047DABC | 00000081 | R . L . B . | | |
| sub_47DB70 | .text | 0047DB70 | 00000086 | R B . | | |
| Graphics::TIcon::NewImage(uint,Classes::TM...) | .text | 0047DC28 | 00000073 | R . L . B . | | |
| unknown_lname_254 | .text | 0047DC3C | 00000017 | R . L . . . | | |
| Graphics::TIcon::SelHeight(int) | .text | 0047DCB4 | 00000018 | R . L . . . | | |
| nullsub_32 | .text | 0047DCCC | 00000001 | R | | |

[그림 96] Functions 출력창

| ICON | 내 용 |
|------|--|
| R | function returns to the caller |
| F | far function |
| L | library function |
| S | static function |
| B | BP based frame. IDA will automatically convert all frame pointer [BP+xxx] operands to stack variables |
| T | function has type information |

4.7 Arrow Tag

| 화살표 종류 | 내 용 |
|--------|---------------------------|
| 빨간 화살표 | 소스 위치와 목적지 위치를 가리킴 |
| 까만 화살표 | 선택된 부분을 나타냄 |
| 회색 화살표 | 다른 모든 종류의 화살표들을 통합적으로 나타냄 |

5 Debugging with OllyDBG&IDA

IDA 의 간단한 사용법을 참고하여 OllyDBG 와 IDA 로 악성코드 함수의 기능들을 차례로 분석하도록 하겠습니다.

```

; int __stdcall WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)
WinMain@16 proc near

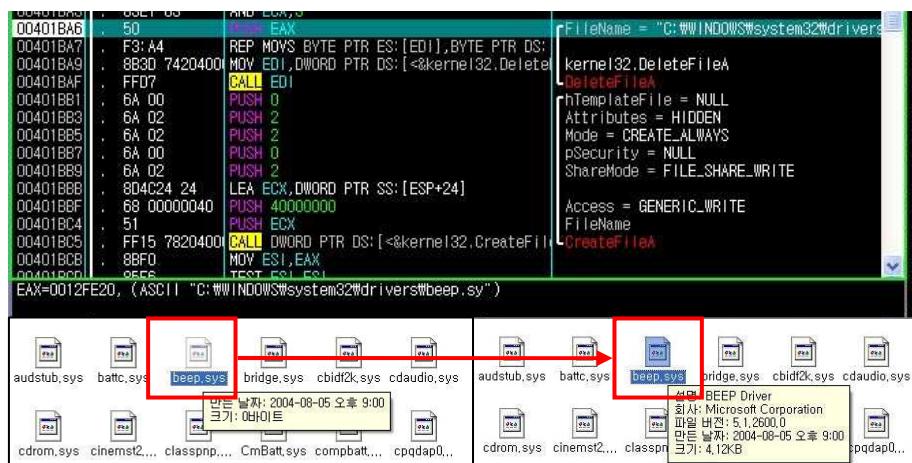
    push    0
    call    sub_401B50      ; beep.sys 생성 및 실행
    add     esp, 4
    call    sub_4012F0      ; nbjs.dll 생성
    call    sub_4012F0      ; BITS 서비스 설정 초기화
    call    sub_4011A0      ; nbjs.dll을 BITS 서비스 키에 등록
    call    sub_401000      ; BITS 서비스 자동시작 설정
    call    sub_401400      ; 고소장처리결과보고.exe 파일 삭제
    xor    eax, eax
    retn   10h
WinMain@16 endp

```

[그림 97] 고소장접수결과보고.exe WinMain()

5.1 Beep.sys 생성 및 실행 함수(sub_401B50)

OllyDBG 를 통해서 분석을 하면서 Beep.sys 드라이버 파일이 삭제 된 후 크기가 0 바이트 인 파일이 생성된 후 writeFile 을 통해서 4.12KB 짜리 악성 beep.sys 가 생성되는 것을 확인 할 수 있습니다.



[그림 98] beep.sys 파일 삭제 및 생성

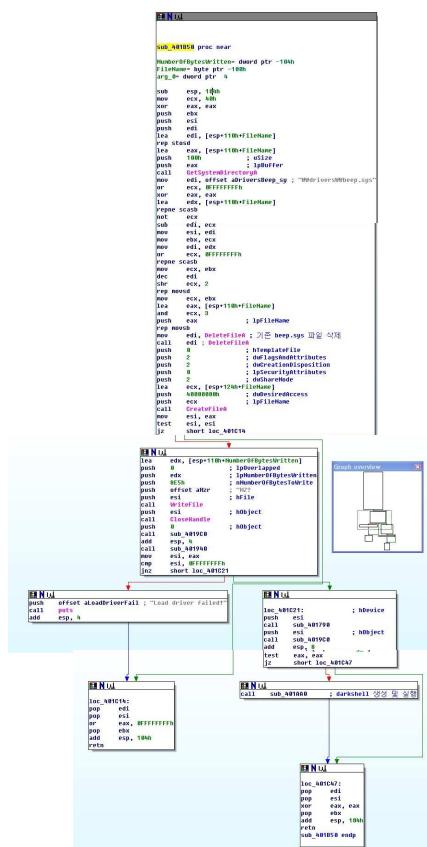
00401B05 . 6A 00 PUSH 0
00401B07 . 52 PUSH EDX
00401B08 . 68 E5080000 PUSH 8E5
00401B00 . 68 28314000 PUSH malware_.00403128
00401BE2 . 56 PUSH ESI
00401BE3 FF15 7C204000 CALL DWORD PTR DS:[<&kernel32.WriteFile]
00401BE9 . 56 PUSH ESI
00401BEA FF15 80204000 CALL DWORD PTR DS:[<&kernel32.CloseHandle]
00401BF0 . 6A 00 PUSH 0
00401BF2 . E8 C9FDFFFF CALL malware_.00401900
00401BF7 . 83C4 04 ADD ESP, 4
00401BFA . E8 41FDFFFF CALL malware_.00401940
00401BFF . 0FEC FAD NOW_ESI_CW

DS: [0040207C]=7C810F9F (kernel32.WriteFile)

| Address | Hex dump | ASCII |
|----------|---|------------------|
| 00403128 | 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 | MZL...J... |
| 00403138 | B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 | ?.....@..... |
| 00403148 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | |
| 00403158 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C8 00 |?.. |
| 00403168 | 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 63 | @?..??L?Th |
| 00403178 | 69 73 20 70 72 6F 67 72 61 60 20 63 61 6E 6E 6F | is program canno |
| 00403188 | 74 20 62 65 20 72 25 6E 20 69 6E 20 44 4E 53 20 | t to run in Dos |

[그림 99] WriteFile()로 beep.sys 생성완료

아래는 beep.sys 생성 함수 부분의 전체 구조입니다.

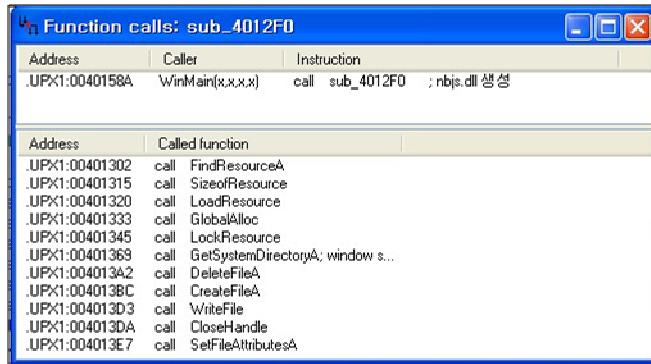


[그림 100] beep.sys 생성 및 실행함수 전체 구조

5.2 nbjs.dll 생성 함수(sub_4012F0)

IDA에서 nbjs.dll 생성 함수에서 사용되는 함수 목록을 살펴보았습니다.

우선 리소스 정보를 얻은 후 ‘시스템 경로’를 얻은 후 파일을 삭제하고, 파일을 생성하는 함수들이 존재하는 것을 알 수 있습니다.



[그림 101] nbjs.dll 생성함수의 함수목록

ollyDBG를 통해서 실제로 CreateFileA 함수를 이용하여 nbjs.dll을 생성하는 것을 확인 할 수 있습니다.



[그림 102] nbjs.dll 생성(CreateFileA)

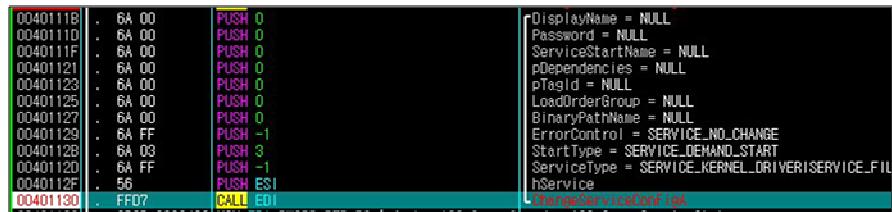
아래는 nbjs.dll 파일의 내용부분이 메모리에 로딩된 것을 확인 할 수 있습니다.

| Address | Hex dump | ASCII |
|----------|---|------------------|
| 00404080 | 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 | MZ?L...J... |
| 004040C0 | B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 | ?.....@..... |
| 00404000 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | |
| 004040E0 | 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 |r... |
| 004040F0 | 0E 1F BA 0E 00 B4 09 C0 21 B8 01 4C CD 21 54 68 | A?.???.L?Th |
| 00404100 | 69 73 20 70 72 6F 67 72 61 60 20 63 61 6E 6E 6F | is program canno |
| 00404110 | 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 | t be run in DOS |
| 00404120 | 60 6F 64 65 2E 00 00 0A 24 00 00 00 00 00 00 00 | mode....\$..... |
| 00404130 | 2B FD 34 4F 6F 9C 5A 1C 6F 9C 5A 1C 6F 9C 5A 1C | +?0o锚o锚o锚 |
| 00404140 | 87 83 50 1C 69 9C 5A 1C 14 80 56 1C 6E 9C 5A 1C | 锚Pi锚t锚Vn锚 |
| 00404150 | EC 80 54 1C 60 9C 5A 1C 90 BC 50 1C 6B 9C 5A 1C | ?Tm锚莫PK锚 |

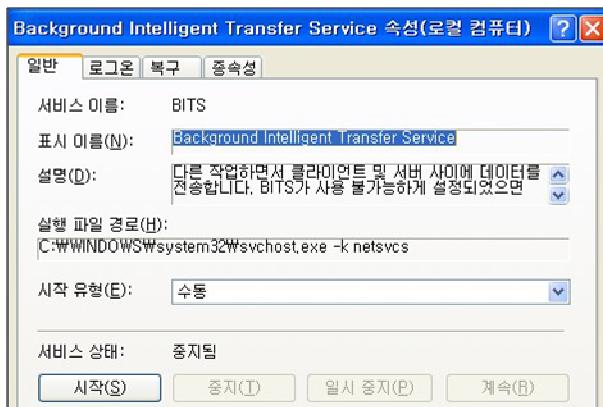
[그림 103] 로딩된 nbjs.dll 내용

5.3 BITS 서비스 설정 초기화 함수(*sub_4010C0*)

ChangeServiceConfigA로 서비스를 초기화하는 것을 알 수 있습니다.



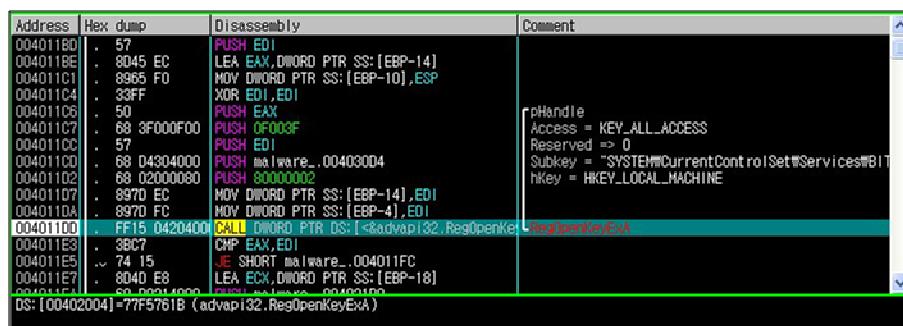
[그림 104] BITS 서비스 초기화



[그림 105] BITS 서비스 상태(수동/정지됨)

5.4 nbjs.dll을 BITS 서비스 키에 등록 함수(*sub_4011A0*)

RegOpenKeyExA로 BIT 서비스의 레지스트리 값을 Open 합니다.



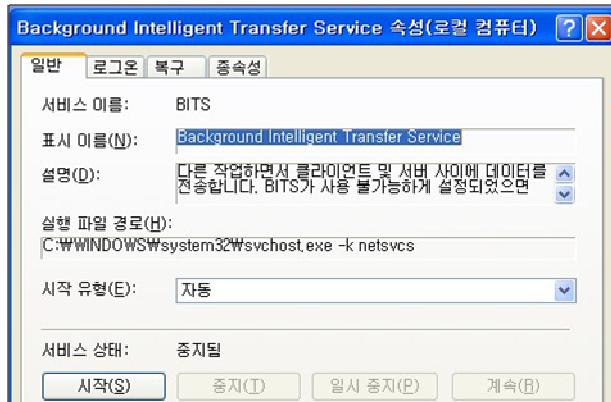
[그림 106] 레지스트리 값 Open

RegSetValueExA 를 이용하여 BITS 서비스 ‘시작유형’ 값을 ‘자동’(2)로 변경 하고 있습니다.

| Address | Hex dump | Disassembly | Comment |
|----------|-------------------|---|-----------------------|
| 004011F0 | . C745 E8 043C | MOV DWORD PTR SS:[EBP-18],malware_0040 | |
| 004011F7 | . E8 940A0000 | CALL <JMP.&msvcr7._CxxThrowException> | |
| 004011FC | > 8845 EC | MOV EAX,DWORD PTR SS:[EBP-14] | |
| 004011FF | . 8810 08204000 | MOV EBX,DWORD PTR DS:[<&advapi32.RegSet...] | |
| 00401205 | . 8055 E4 | LEA EDX,DWORD PTR SS:[EBP-1C] | |
| 00401208 | . 6A 04 | PUSH 4 | BufSize = 4 |
| 0040120A | . 52 | PUSH EDX | Buffer |
| 0040120B | . 6A 04 | PUSH 4 | ValueType = REG_DWORD |
| 0040120D | . 57 | PUSH ED1 | Reserved |
| 0040120E | . 68 CC304000 | PUSH malware_004030CC | ValueName = "Start" |
| 00401213 | . 50 | PUSH EAX | hKey |
| 00401214 | . C745 E4 0200 | MOV DWORD PTR SS:[EBP-1C],2 | |
| 00401218 | . FF03 | CALL EBX | RegSetValueExA |
| 0040121D | . 88F0 | MOV ESI,EAX | |
| 0040121F | . 56 | PUSH ES1 | Error |
| 00401220 | . F5 45 0000 0010 | CALL QWORD PTR DS:[EBP-10].advapi32.RegSet... EBX=??F5EBE7 (advapi32.RegSetValueExA) | CallQwordError |

[그림 107] BIT 서비스의 시작 유형 값을 변경함

변경된 레지스트리 값에 따라서 BITS 서비스의 시작 유형이 ‘자동’으로 바뀌었습니다. 해당 시스템이 재시작 할 경우 자동으로 서비스가 시작하도록 설정 되었습니다.



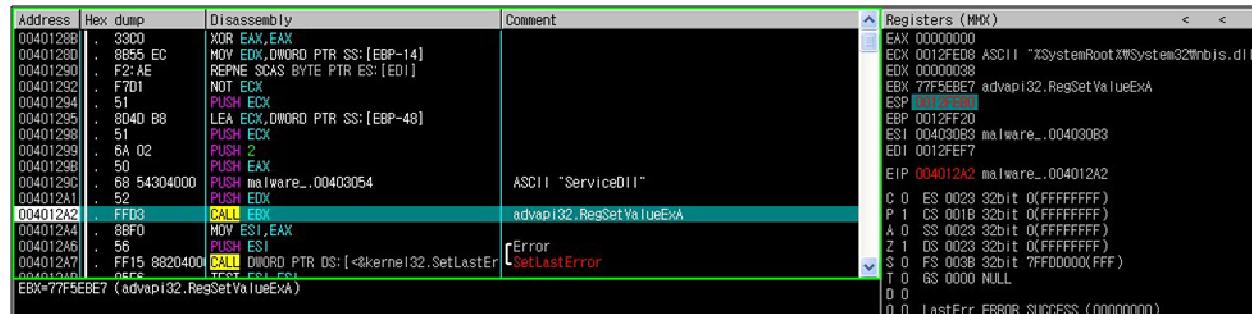
[그림 108] BIT 서비스의 시작 유형 값을 변경됨

악성코드가 실행되기 전의 BITS 의 ServiceDLL 값은 ‘qmgr.dll’ 파일입니다.



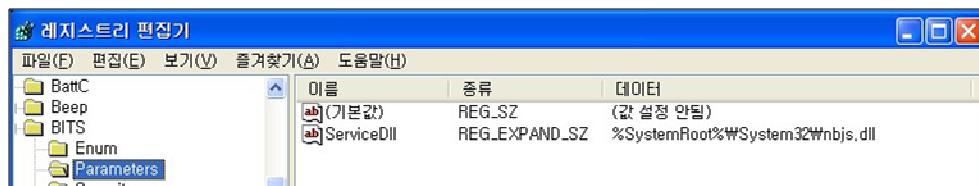
[그림 109] BIT 서비스의 원본 DLL 값

RegSetValueExA 를 이용하여 system32\ nbjs.dll 파일로 ServiceDll 값이 변경됩니다.



[그림 110] BIT 서비스의 DLL 값 변조

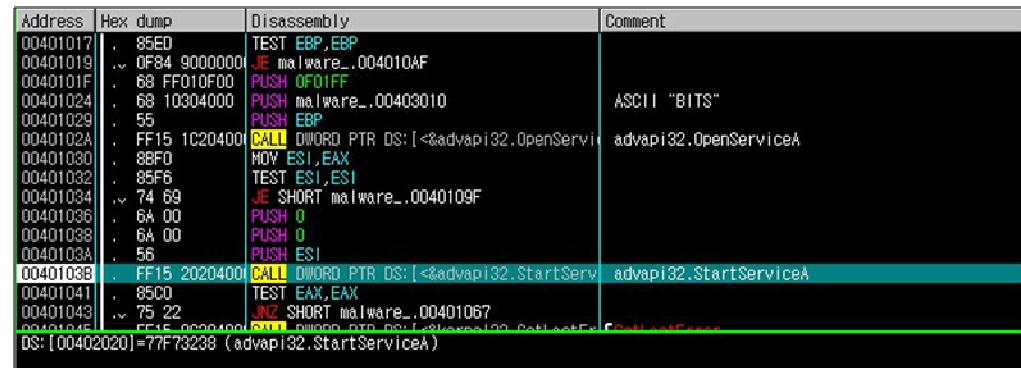
다음과 같이 서비스의 ServiceDll 값이 변경됨을 알 수 있습니다. BITS 서비스가 시작되면 qmgr.dll 파일 대신 nbjs.dll 파일이 실행 됨을 알 수 있습니다.



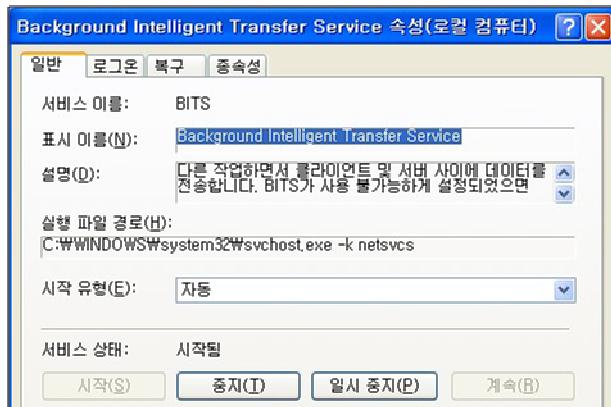
[그림 111] BIT 서비스의 변조된 DLL 값

5.5 BITS 서비스 시작 설정 함수(*sub_401000*)

StartServiceA 를 통해서 BITS 서비스를 실행합니다.



[그림 112] 변조된 BIT 서비스 실행



[그림 113] 변조된 BITS 서비스 실행

5.6 고소장처리결과보고.exe 파일 삭제 함수(sub_401400)

악성코드인 beep.sys 와 nbjs.dll 파일을 생성 한 후 실행시키고 malware_mup.exe (고소장처리결과보고.exe) 파일은 삭제됩니다.

Lstrcat 을 통해서 삭제시 사용할 명령 값들이 합쳐지는 것을 확인 할 수 있습니다.

| Address | Hex dump | Disassembly | Comment |
|---|---|---------------------------------------|---|
| 0040147B | 805424 64 | LEA EDX, DWORD PTR SS:[ESP+64] | |
| 0040147F | 808424 68010 | LEA EAX, DWORD PTR SS:[ESP+168] | |
| 00401486 | 52 | PUSH EDX | |
| 00401487 | 50 | PUSH EAX | |
| 00401488 | FFD6 | CALL ESI | |
| 0040148A | 808C24 68010 | LEA ECX, DWORD PTR SS:[ESP+168] | |
| 00401491 | 68 0C314000 | PUSH malware...,0040310C | String2 = "> null" |
| 00401496 | 51 | PUSH ECX | |
| 00401497 | FFD6 | CALL ESI | |
| 00401499 | 809424 68010 | LEA EDX, DWORD PTR SS:[ESP+168] | |
| 004014A0 | 808424 6C020 | LEA EAX, DWORD PTR SS:[ESP+260] | |
| 004014A7 | 52 | PUSH EDX | |
| 004014A8 | 50 | PUSH EAX | String2 = "C:\WINDOWS\system32\cmd.exe" |
| 004014A9 | FFD6 | CALL ESI | |
| 004014AB | B9 10000000 | MOV ECX, 10 | |
| 004014AC | 3200 | MOV EAX, ECX | |
| EAX=0012FE20, (ASCII "C:\WINDOWS\system32\cmd.exe") | | | |
| Address | Hex dump | ASCII | Address |
| 0012F010 | 00 00 00 00 A0 01 15 00 | 0A 00 00 00 20 2F 68 20 ...?4..... /c | 0012FB80 |
| 0012F020 | 20 64 65 6C 20 43 3A 50 40 41 4C 57 41 52 7E 31 | del C:\MALWAR~1 | 0012FB84 |
| 0012F030 | 5C 80 ED BC D2 C0 E5 7E 31 5C 40 41 4C 57 41 52 | #고소장~1\MALWAR | 0012FB88 |
| 0012F040 | 7E 31 2E 45 58 45 20 3E 20 6E 75 6C 00 16 17 00 | ~1.EXE > null .. | 0012FBBC |
| 0012F050 | 00 00 15 00 40 54 15 00 00 00 00 00 00 FF 12 00 | + 0t4 .. ? | 0012F0C0 |

[그림 114] cmdline 명령어 조합

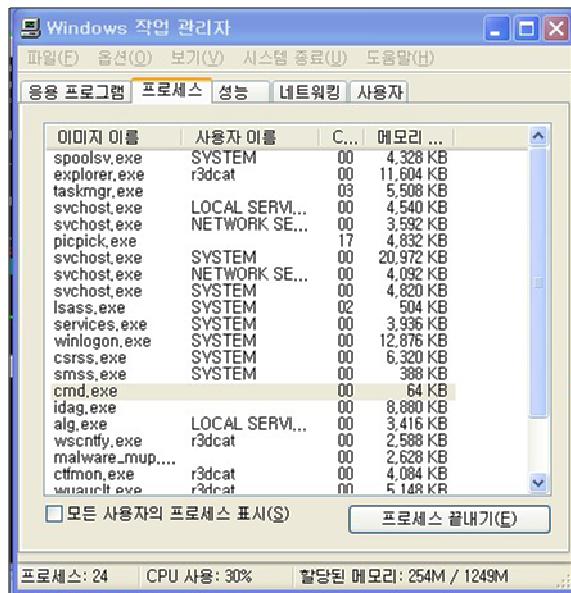
'/c del c:\ malware~ 1\ 고소장~ 1\ MALWAR~ 1.EXE > null' 와 같이 명령어가 차례로 생성되는 것을 알 수 있습니다.

생성한 명령어는 CreateProcessA함수를 통해서 CommandLine으로 실행이 됩니다.

| Address | Hex dump | Disassembly | Comment |
|---|-----------------|---|---|
| 0040150C | . 804424 20 | LEA EAX,DWORD PTR SS:[ESP+20] | |
| 00401510 | . 52 | PUSH EDX | pProcessInfo |
| 00401511 | . 50 | PUSH EAX | pStartupInfo |
| 00401512 | . 6A 00 | PUSH 0 | CurrentDir = NULL |
| 00401514 | . 6A 00 | PUSH 0 | pEnvironment = NULL |
| 00401516 | . 6A 0C | PUSH OC | CreateFlags = CREATE_SUSPENDED DETACHED_PROCESS |
| 00401518 | . 6A 00 | PUSH 0 | InheritHandles = FALSE |
| 0040151A | . 6A 00 | PUSH 0 | pThreadSecurity = NULL |
| 0040151C | . 808C24 880201 | LEA ECX,DWORD PTR SS:[ESP+288] | |
| 00401523 | . 6A 00 | PUSH 0 | pProcessSecurity = NULL |
| 00401525 | . 51 | PUSH ECX | CommandLine = "C:\Windows\system32\cmd.exe /c |
| 00401526 | . 6A 00 | PUSH 0 | ModuleFileName = NULL |
| 00401528 | . FF15 3C204000 | CALL DWORD PTR DS:[<kernel32.CreateProcessA | CreateProcessA |
| 0040152E | . 85C0 | TEST EAX,EAX | |
| 00401530 | . 74 28 | JE SHORT malware...,0040155A | |
| 00401532 | . 00E404 10 | MOV ECX,DWORD PTR [esp+10] | |
| CALL DWORD PTR DS:[<kernel32.ExitProcess] | | | |

[그림 115] CreateProcessA로 삭제구문 실행

Windows 작업 관리자에서 cmd.exe로 해당 프로세스가 실행되는 것을 알 수 있습니다.



[그림 116] 삭제구문 실행실행

이렇게 malware_mup.exe(고소장처리결과보고.exe) 파일은 beep.sys 드라이버와 nbjs.dll 파일을 생성하고 실행하는 역할을 하는 것을 알 수 있습니다.

6 악성코드 증상 분석

처음에 살펴보았던 모니터링 도구들을 통해서 악성코드를 실행한 후 악성코드가 생성하는 파일들과 레지스트리 값 변경, 생성 프로세스, 네트워크 연결 등을 살펴봄으로써 앞서 분석한 내용과 일치하는지 확인 합니다.

6.1 API 함수 분석

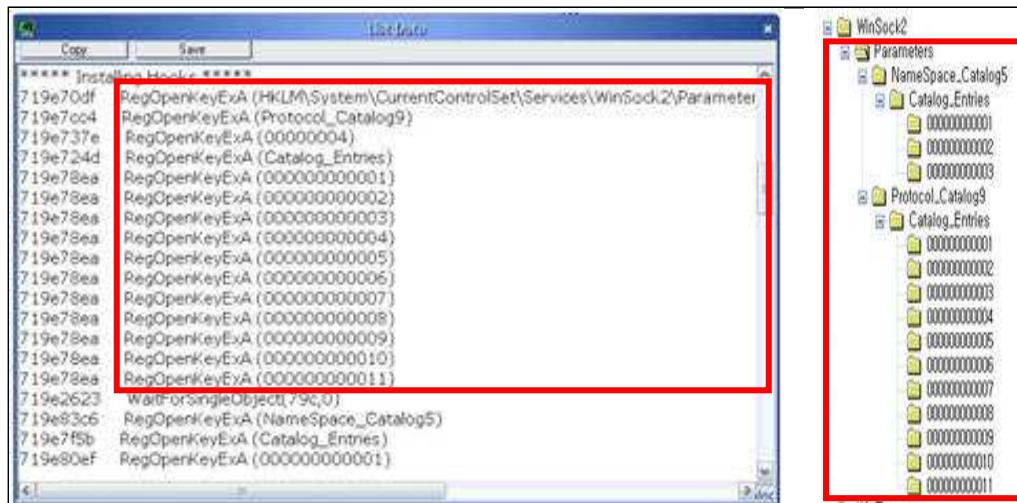
실제 악성코드(malware_mup.exe)을 SysAnalzer를 이용하여 분석해도록 하겠습니다.

우선 'Api Log'를 통해서 API 함수가 호출되는 로그를 살펴보면서 해당 파일의 기능함수를 알아보도록 하겠습니다.

1) Malware_mup.exe 실행

해당 프로그램이 실행되면 가장 먼저 RegOpenKeyEx() 함수를 이용하여 필요한 해당 키를 확인 합니다. 레지스트리의 winsock 파라미터로 네트워크 환경 값을 참조하는 것을 알 수 있습니다.

(HKLM\ System\ CurrentControlSet\ Services\ WinSock2\ Parameter 하위 키 포함)



[그림 117] RegOpenKeyEx() 및 실제 레지스트리 항목

2) C:\ windows\ system32\ drivers\ beep.sys 파일 생성

악성코드 드라이버인 beep.sys 파일을 생성 합니다.

| | | | |
|---|--|---|---|
| 719d1afa RegOpenKeyExA(HKEY\SYSTEM\CurrentControlSet\Services\Winsock2\Parameter 719d1996 GlobalAlloc() 7c80b511 ExitThread() 401bcb CreateFileA(C:\WINDOWS\system32\drivers\beep.sys) 401be9 WriteFile(h=0x8) 77f65f5e WaitForSingleObject(7e8,2bf20) | audstub.sys battc.sys beep.sys bridge.sys cbidfl2k.sys | 3KB 시스템 파일 16KB 시스템 파일 5KB 시스템 파일 70KB 시스템 파일 14KB 시스템 파일 | 2001-08-17 오후 ... 2001-08-27 오후 ... 2005-08-30 오전 ... 2005-08-30 오전 ... 2005-08-30 오전 ... |
|---|--|---|---|

[그림 118] CreateFileA()로 beep.sys 생성

3) C:\windows\system32\nbjs.dll 파일 생성

DDOS 기능을 하는 악성코드 파일인 nbjs.dll을 숨김 속성으로 생성 합니다.

| | | | |
|---|---|---|--|
| 77dabf8e RegOpenKeyExA(HKEY\Software\Microsoft\Rpc) 77f65f5e WaitForSingleObject(778,2bf20) 40199f CreateFileA(\\.\Re1986SDTOS) 4017e2 GlobalAlloc() 77f65f5e WaitForSingleObject(774,2bf20) 401339 GlobalAlloc() 4013c2 CreateFileA(C:\WINDOWS\system32\nbjs.dll) 4013d9 WriteFile(h=778) | narrator.exe narrhook.dll nbjs.dll nbtstat.exe ncobjapi.dll | 52KB 응용 프로그램 35KB 응용 프로그램 확장 82KB 응용 프로그램 확장 20KB 응용 프로그램 36KB 응용 프로그램 확장 | 2005-08-30 오전 12:00 2005-08-30 오전 12:00 2008-07-22 오후 6:37 2005-08-30 오전 12:00 2005-08-30 오전 12:00 |
|---|---|---|--|

[그림 119] CreateFileA()로 nbjs.dll 생성

4) BITS 서비스에 등록

BITS(Background Intelligent Transfer Service) 서비스는 다른 작업하면서 클라이언트 및 서버 사이에 데이터를 전송할 수 있는 서비스이며, 윈도우 로그인 시 대부분 자동 실행될 수 있도록 되어 있습니다. 해당 서비스에 임의로 등록하여 시스템을 재시작 할 경우 해당 악성코드가 실행하도록 설정 합니다.

| | | | |
|------------|----------------|---------------|------------------------------|
| BITS | 이름 | 종류 | 데이터 |
| Enum | ab(기본값) | REG_SZ | (값 설정 안됨) |
| Parameters | ab(ServiceDLL) | REG_EXPAND_SZ | C:\WINDOWS\system32\qmgr.dll |

악성코드 실행 전

| | | | |
|-------|----------------|---------------|--------------------------------|
| BattC | 이름 | 종류 | 데이터 |
| Beep | ab(기본값) | REG_SZ | (값 설정 안됨) |
| BITS | ab(ServiceDLL) | REG_EXPAND_SZ | %SystemRoot%\System32\nbjs.dll |

악성코드 실행 후

[그림 120] BITS 서비스 ServiceDLL 교체

5) 악성코드 파일(malware_mup.exe) 자신을 삭제

CreateProcessA((null),

C:\WINDOWS\system32\cmd.exe /c del C:\DOCUMENT~1
\ MALWAR~1.EXE > nul,0,(null))

악성코드 파일들(nbjs.dll, beep.sys)을 모두 생성 한 후 서비스에 등록한 malware_mup.exe (고소장접수결과.exe 의 언패킹 파일)은 CreateProcessA()를 통해서 자신을 삭제 합니다.

```
40152e CreateProcessA((null),C:\WINDOWS\system32\cmd.exe /c del C:\DOCUME~1\ma  
7c81628b WaitForSingleObject(770,64)
```

[그림 121] 악성코드파일 삭제

6) 필요한 DLL 파일 로드

```
76d94cd7 LoadLibraryA(VERSIÓN.dll)=77bb0000  
7c818e2c LoadLibraryA(advapi32.dll)=77f50000  
10001e25 LoadLibraryA(psapi.dll)=76ba0000  
10001e66 GetCurrentProcessId()=668  
76ba183b ReadProcessMemory(h=77c)  
76ba185a ReadProcessMemory(h=77c)  
76ba1878 ReadProcessMemory(h=77c)  
76ba17bb ReadProcessMemory(h=77c)  
4ad04f9d GetCurrentProcessId()=1984  
4ad05008 LoadLibraryA(ADVAPI32.dll)=77f50000  
7d606969 GetCurrentProcessId()=1984
```

[그림 122] 필요 DLL 로드

6.2 생성/변조/삭제 된 파일 리스트

악성코드 파일이 파일 시스템에 파일들을 생성/변조/삭제하는 리스트입니다.

앞서 살펴본바와 같이 악성코드 실행 후 2 개의 악성파일을 생성한 후 악성코드 은닉을 위해 윈도우에서 생성되는 주요 로그 파일들을 수정 합니다.

```
Deteled: C:\ WINDOWS\ system32\ drivers\ beep.sys  
Created: C:\ WINDOWS\ system32\ drivers\ beep.sys  
Modified: C:\ WINDOWS\ system32\ drivers\ beep.sys  
Modified: C:\ WINDOWS\ system32\ CatRoot2\ {F750E6C3- 38EE- 11D1-  
85E5- 00C04FC295EE}\ catdb  
Modified: C:\ WINDOWS\ system32\ config\ default.LOG  
Created: C:\ WINDOWS\ system32\ nbjs.dll  
Modified: C:\ WINDOWS\ system32\ nbjs.dll  
Modified: C:\ WINDOWS\ system32\ config\ system.LOG  
Created: C:\ DOCUME~ 1\ master\ LOCALS~ 1\ Temp\ JETADC7.tmp  
Modified: C:\ WINDOWS\ system32\ wbem\ Logs\ wbemess.log  
Created: C:\ DOCUME~ 1\ master\ LOCALS~ 1\ Temp\ JET1C.tmp  
Modified: C:\ WINDOWS\ system32  
Deteled: C:\ DOCUME~ 1\ master\ LOCALS~ 1\ Temp\ JET1C.tmp  
Deteled: C:\ DOCUME~ 1\ master\ LOCALS~ 1\ Temp\ JETADC7.tmp  
Created: C:\ DOCUME~ 1\ master\ LOCALS~ 1\ Temp\ ~ DF9F7B.tmp
```

Modified: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DF9F7B.tmp
Created: C:\ WINDOWS\ system32\ drivers\ beep.sys.new
Modified: C:\ WINDOWS\ system32\ drivers\ beep.sys.new
Created: C:\ WINDOWS\ system32\ dllcache\ beep.sys.new
Modified: C:\ WINDOWS\ system32\ dllcache\ beep.sys.new
Modified: C:\ WINDOWS\ system32\ dllcache\ beep.sys
Deleted: C:\ WINDOWS\ system32\ dllcache\ beep.sys.new
Modified: C:\ WINDOWS\ system32\ CatRoot2\ edb.chk
Modified: C:\ WINDOWS\ system32\ wbem\ Repository\ FS\ INDEX.MAP
Modified: C:\ WINDOWS\ system32\ wbem\ Repository\ FS\ OBJECTS.MAP

Modified:
C:\ WINDOWS\ system32\ wbem\ Repository\ FS\ MAPPING2.MAP
Modified: C:\ WINDOWS\ system32\ wbem\ Repository\ FS\ MAPPING.VER
Created: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DFDC57.tmp
Modified: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DFDC57.tmp
Created: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DFE10D.tmp
Modified: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DFE10D.tmp
Created: C:\ Program Files\ ESTsoft\ ALZip\ \$_Temp_\$\$\$\$
Deleted: C:\ Program Files\ ESTsoft\ ALZip\ \$_Temp_\$\$\$\$
Modified: C:\ Program Files\ ESTsoft\ ALZip
Created: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DFA940.tmp
Modified: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DFA940.tmp
Deleted: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DFA940.tmp
Modified: C:\ WINDOWS\ Debug\ UserMode\ userenv.log

Modified:
C:\ WINDOWS\ system32\ wbem\ Repository\ FS\ MAPPING1.MAP
Created: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DF9D6B.tmp
Modified: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DF9D6B.tmp
Deleted: C:\ DOCUME~1\ master\ LOCALS~1\ Temp\ ~DF9D6B.tmp

| | | |
|---------------------------------|--|------------------------------|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|------------------------------|

0x09. IDA Scripting and Plug-in

IDC란?

IDA에서 사용할 수 있는 C와 유사한 문법을 가진 Script 언어
IDC를 이용하여 IDA의 기능을 확장할 수 있음

IDA의 Help Document에 IDC관련 Help 제공

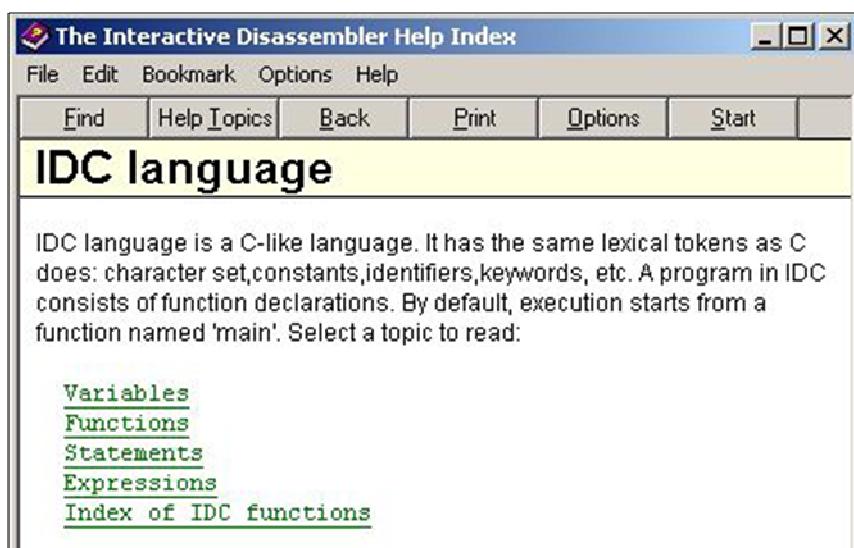


그림 123 IDC관련 Help 제공

Idc 실행방법

1. IDC Command

IDC 스크립트를 바로 실행 가능

기본적으로 Function을 사용할 수 없음

편법을 이용하여 Function 사용 가능

File | IDC Command (Shift + F2)

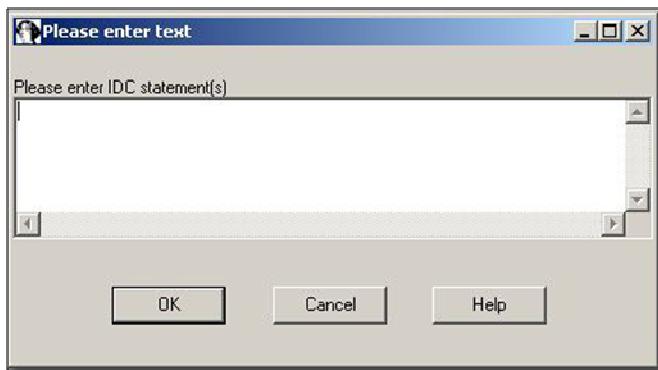


그림 124 IDC 실행

Loading the IDC File Script

IDC Script를 파일로 작성한 후 IDA에서 불러온다.

File | IDC File

IDC Syntax

IDC Scripting Language는 C언어 문법과 유사

모든 구문은 세미콜론(;)으로 끝난다

if, else if, while, do file , for, continue, break 등 C언어와 같은 키워드를 사용

주석

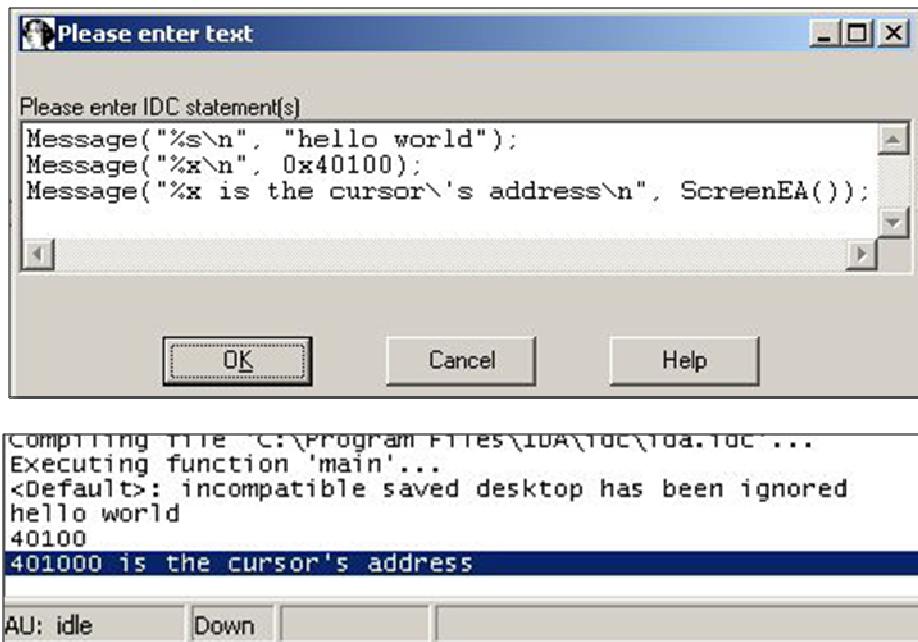
//, /* */

◆ OutPut

1. Message

```
void Message(string format, ... );
```

- IDA 하단의 메시지 창에 메시지를 출력한다.



2. Warning

- Warning 창을 팝업 시킨다.

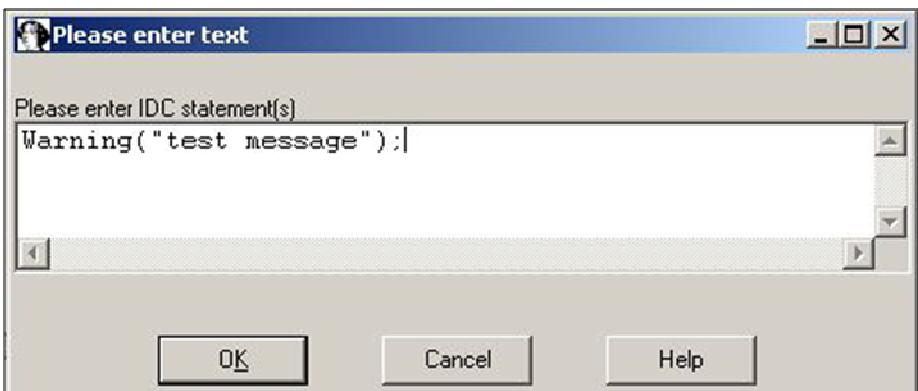


그림 125 IDC Dialog Box

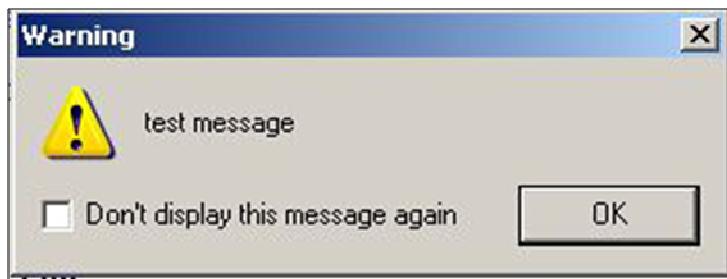


그림 126 Warning 창 화면

3. Fatal

- Fatal Error 창을 띄운 후 IDA를 종료한다.

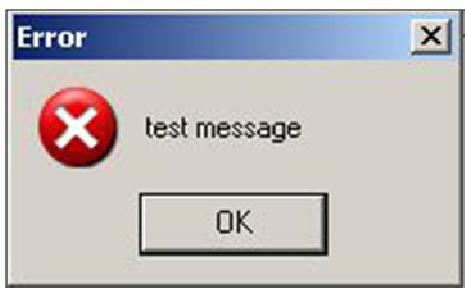


그림 127 Fatal 창 화면

◆ Variables

1. IDC의 모든 변수는 auto 타입으로 선언하여 사용

Ex) auto counter;

| The auto type can represent | Example data | Size |
|--|---------------|-------------------|
| 32Bit integer (64 bit in IDA Pro 64) | 0x00401000 | 32bit or 64bit |
| character string | "hello world" | 1023 characters |
| floating point number | 5.23 | 25 decimal digits |

2. 변수의 선언과 할당이 분리되어야 함

```
auto currentAddress;  
currentAddress=ScreenEA();
```

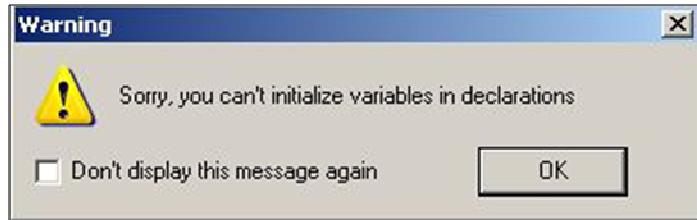


그림 128 선언과 할당을 같이 한 경우 에러 메세지

◆ Operators

1. All variables have local scope

- 함수 내에서만 사용 가능

EX) + - / * % << >> ++ --

2. 사용할 수 없는 연산자

EX) += ,(comma)

3. 문자열 연산

- IDC에서는 C언어 달리 '+' 기호로 문자열 합치는 것이 가능하다.

EX) Message("IDA" + "Script");

결과 : "IDAScript"

◆ Conditional

1. if else

- C 언어와 동일

```
auto currAddr;  
currAddr = ScreenEA();  
if( currAddr %2 )  
    Message("%x is oddWn", currAddr);  
else  
    Message("%x is evenWn", currAddr);
```

2. switch 문은 지원하지 않음

- 다중 if 문은 가능

◆ Loops

1. for, while, do while

- C 언어와 동일
 - for (expr1; expr2; expr3) statement
 - while (expression) statement
 - do statement while (expression);

- ◆ Loops Sample Code

1. Sample code using **a for loop**

```
auto origEA, currEA, funcStart, funcEnd;  
origEA=ScreenEA();  
funcStart=GetFunctionAttr(origEA, FUNCATTR_START);  
funcEnd=GetFunctionAttr(origEA, FUNCATTR_END);  
if( funcStart== -1)  
    Message("%x is not part of a functionWn", origEA);  
for(currEA=funcStart; currEA!=BADADDR; currEA=NextHead(currEA,  
funcEnd))  
{  
    Message("%8xWn", currEA);  
}
```

2. 결과

- 현재 함수내에 있는 모든 명령어의 주소를 출력

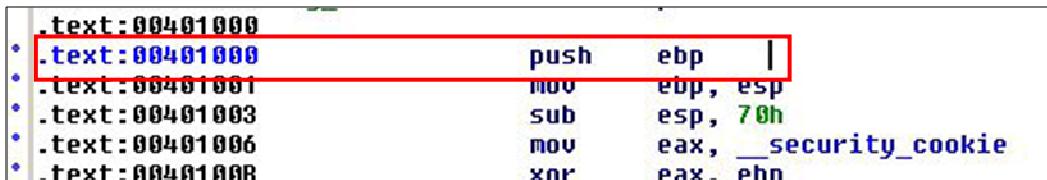
3. Sample code using a **while** loop

```
auto origEA, currEA, funcStart, funcEnd;  
origEA = ScreenEA();  
funcStart = GetFunctionAttr(origEA, FUNCATTR_START);  
funcEnd = GetFunctionAttr(origEA, FUNCATTR_END);  
if ( funcStart == -1 )  
    Message("%x is not part of a function\n", origEA);  
currEA = funcStart;  
while( currEA != BADADDR)  
{  
    Message("%8x\n", currEA);  
    currEA = NextHead(currEA, funcEnd)
```

4. 결과

- 현재 함수내에 있는 모든 명령어의 주소를 출력
- ◆ ScreenEA()
 - 현재 커서의 주소값을 반환

```
long ScreenEA();
```



```
.text:00401000  
.text:00401000      push   ebp  
.text:00401001      mov    ebp, esp  
.text:00401003      sub    esp, 70h  
.text:00401006      mov    eax, _security_cookie  
.text:00401008      xor    eax, ebp
```

그림 129 커서의 위치

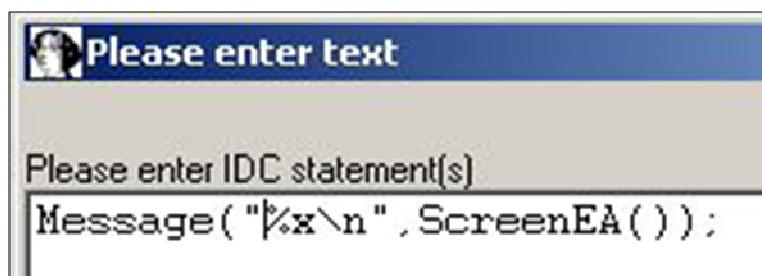


그림 130 IDC Dialog에 Script 입력

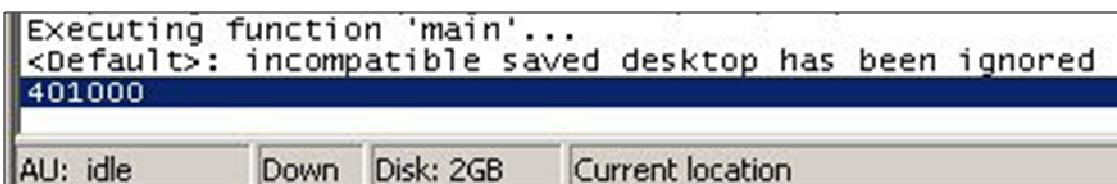


그림 131 메시지 창에 주소 출력

5. GetFunctionAttr

```
long GetFunctionAttr(long ea, long attr);
```

- 함수의 속성값 반환
- ea 가 속해있는 함수의 attr 속성을 반환한다.
- ea 가 함수내의 주소가 아니라면 -1을 반환하게 된다.

| attr 상수 | 의미 |
|------------------|---------------------------------------|
| FUNCATTR_START | function start address |
| FUNCATTR_END | function end address |
| FUNCATTR_FLAGS | function flags |
| FUNCATTR_FRAME | function frame id |
| FUNCATTR_FRSIZE | size of saved registers area |
| FUNCATTR_ARGSIZE | number of bytes purged from the stack |
| FUNCATTR_FPD | frame pointer delta |
| FUNCATTR_COLOR | function color code |

6. NextHead

```
long NextHead(long ea, long maxea);
```

다음 Instruction이나 Data의 주소를 반환

Uarguments

- ea : 시작지점
- maxea : 검사할 끝지점

지정된 범위내에 Instruction이나 data가 없다면 BADADDR을 반환

IA- 32같이 가변길이 명령어 아키텍쳐에서는 반복적으로 사용된다

RISC 아키텍처에서는 명령어길이만큼 더해주면 되기 때문에 NextHead 가 필요 없음

7. BADADDR

IDC에서 제공하는 상수

함수 호출이나 변수할당 초기화 등의 성공유무 판단시 사용

- 1 의 의미

◆ Functions

IDC에서도 반복되는 기능을 함수로 만들어 사용할 수 있다.

모든 Function은 static 형태로 정의되어야 한다.

IDC 를 파일형태로 만들어 사용하기 위해서는 반드시 Function을 작성해야 한다.

Sample code

```
static outputCurrentAddress(myString)
{
    auto currAddress;
    currAddress=ScreenEA();
```

IDC Dialog 창에서의 Function을 작성할 수 있는가?

- IDC Dialog 는 '_idc' 라는 함수 내에서 실행 되도록 되어있다.
- 함수 내에 함수를 선언하는 형태가 되므로 에러가 발생한다.

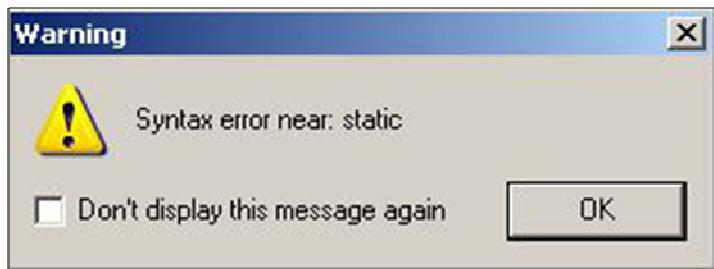


그림 132 IDC Dialog 내에 Function 사용시 에러 메세지

- 편법을 써서 함수를 사용 할 수 있다.
 - A. Willem Jan Hengeveld가 해결방법을 알아냄

➤ <http://www.xs4all.nl/~itsme/projects/disassemblers/ida.html>

IDC Dialog 창에서 Function 이용하는 방법

함수를 등록하고 '}' 를 이용하여 _idc 함수를 닫는다

사용한 함수 마지막을 닫지 않는다.

```
AddHotkey("Alt-f9", "outputCurrentAddress");
outputCurrentAddress();
} ← _idc function's closing brace
static outputCurrentAddress()
{ ← no closing brace
auto currAddress;
currAddress=ScreenEA();
Message("%x Wn", currAddress);
return currAddress;
```

◆ Global Variables

IDC는 Global Variable을 직접 지원하지 않는다.

외부에 변수를 선언할 수 없음

그러나 배열을 이용하여 Global 하게 사용할 수 있다.

8. 배열 생성

```
long CreateArray(string name)
```

9. 배열에 데이터 쓰기|long CreateArray(string name)

```
success SetArrayLong( long id, long idx, long value );  
success SetArrayString( long id, long idx, string str );
```

10. 배열에서 데이터 읽기

```
#define AR_LONG 'A' // 숫자  
#define AR_STR 'S' // 문자열  
long GetArrayId(string name);  
long GetArrayElement(AR_LONG, long id, long idx);  
string GetArrayElement(AR_STR, long id, long idx);
```

11. 배열에서 데이터 삭제

```
success DelArrayElement(long tag, long id, long idx);x;
```

12. Global Variable 을 편리하게 사용할 수 있는 라이브러리

- http://www.openrce.org/downloads/details/81/Common_Scripts
- common.idc

- InitGlobalVars()
- SetGlobalVarLong(index, value)
- SetGlobalVarString(index, string)
- GetGlobalVarLong(index)
- GetGlobalVarString(index)

Script Samples

- ◆ ResetColor IDC Script

```
#include <idc.idc>
static main(void)
{
    auto origEA, currEA, currColor, funcStart, funcEnd;
    origEA=ScreenEA();
    funcStart=GetFunctionAttr(origEA, FUNCATTR_START);
    funcEnd=GetFunctionAttr(origEA, FUNCATTR_END);
    Message("Welcome to resetColoe.idc\n");
    if(funcStart== -1 || funcEnd== -1)
    {
        Message("** Error : not in a function **\n");
        return -1;
    }
    Message("[*] Function: %s\n", GetFunctionName(funcStart));
    Message("[*] start = 0x%x, end == 0x%x\n", funcStart, funcEnd);
    for(currEA=funcStart; currEA != BADADDR; currEA =(NextHead(currEA,
        funcEnd)))
    {
        if(SetColor(currEA, CIC_ITEM, 0xffff00)==0)
        {
            Message("** Error : SetColor failed 0x%x **\n", currEA);
        }
    }
    Refresh();
    Message("resetColor is done\n");
}
```

1. GetFunctionName

함수 이름을 반환

```
string GetFunctionName(long ea);
```

Arguments

- ea : any address belonging to the function

return

- null string : function doesn't exist
- otherwise : returns function name

2. SetColor

```
success SetColor(long ea, long what, long color);
```

arguments

- ea : address of the item
- what : type of the item (one of CIC ... constants)
- Color : new color code in RGB (hex 0xBBGGRR)

returns

- 1 : ok
- 0 : failure

```
#define CIC_ITEM 1 // one instruction or data
#define DEFCOLOR 0xFFFFFFFF
#define CIC_FUNC 2 // function
#define CIC_SEGM 3 // segment
```

Useful IDC Functions

◆ Useful IDC Functions

1. Reading Memory

- long Byte(long ea)

ea 주소의 1 byte를 읽음

long Word(long ea)

- ea 주소의 word (2 bytes)를 읽음

long Dword(long ea)

- ea 주소의 double-word (4 bytes)를 읽음

위의 함수들은 실패시 -1 을 반환

실제 데이터가 -1 이라면?

- hasValue 매크로를 이용하여 판단 가능

```
#define hasValue(F) ( ( F & FF_IVL ) !=0 ) // any defined values?
```

값이 아니라면 0을 반환

2. Writing Memory

- void PatchByte (long ea, long value)

ea 주소의 1 byte를 value로 변경

- void PatchWord (long ea, long value)

ea 주소의 word (2bytes)를 value로 변경

- void PatchDword (long ea, long value)

ea 주소의 double-word(4bytes)를 value로 변경

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

Plug- ins

- ◆ IDA Plug- in

1. IDA Pro는 모듈을 통해 기능을 확장할 수 있음
2. IDA에서 사용 가능한 모듈

- Plug- in
- Loaders
OS의 로더와 유사한 기능을 수행
- Processor
다양한 CPU환경을 지원
opcoe를 인터프리팅하고 IDA에서 볼 수 있도록 Disassembly 생성
- Debuggers
IDA내부에서 동작하는 디버거

- ◆ Module/Plug- in Resources

1. Module/Plug- in Resources

- <http://www.hex-rays.com/idapro/idadown.htm>
- <http://www.hex-rays.com/forum>
- <http://binarypool.com/idapluginwriting/>
- http://www.hex-rays.com/idapro/unpack_pe/unpacking.pdf
- <http://hexblog.com>
- http://www.openrc.org/downloads/browse/IDA_Plugins

2. Plug- in은 C++로 작성하며 다양한 컴파일러와 개발환경을 지원

Visual C++

Borland C++ Builder

GCC C++ Compiler

Windows(32 and 64 bit)

Linux (32 and 64 bit)

Mac OS X (32 and 64 bit)

Plug-in Syntax

◆ Plug-in Syntax

- Plug in

*로딩이 가능한 라이브러리 (IDA에서는 필요할 때 로딩이 가능하다)

*C++로 작성

*PLUGIN_t라는 구조체를 통해 export 된다.

```
plugin_t PLUGIN =
{
    IDP_INTERFACE_VERSION,
    plugin_flags,
    init,
    term,
    run,
    comment,
    help,
    wanted_name,
    wanted_hotkey
};
```

◆ PLUGIN_t Structure

- IDP_INTERFACE_VERSION

SDK에 정의되어 있는 버전

- plugin_flags

IDA가 plug-in을 어떻게 처리할 것인지에 대한 정의

debugging plug-in의 경우 0 또는 PLUGIN_UNL이 설정됨

◆ Init

- plug-in이 로딩되었을 때 한번 실행됨

- plug-in을 위한 환경 설정에 사용됨

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

다음중 하나의 반환값을 가져야 한다.

PLUGIN_SKIP

plug-in을 로드하지 않도록 한다.

```
예) if (inf.filetype != f_PE)
    return PLUGIN_SKIP
```

PLUGIN_OK

plug-in을 로드한다

PLUGIN_KEEP

plug-in을 메모리에 남겨둔다.

- ◆ Term

- IDA가 종료될 때 실행됨
- plug-in이 동작중에 사용한 리소스를 반환하고 깨끗하게 정리하는데 사용
- 대부분은 이 함수를 사용하지 않음 (NULL로 셋팅)

- ◆ Run

- plug-in이 동작할때 실행됨
- plug-in 디렉토리에 있는 plugin.cfg에 정의된 arguments를 받을 수 있음
- 실질적으로 필요한 작업이 작성되어 있는 함수

- ◆ Comment

- plug-in에 대한 간략한 설명

- ◆ Help

- plug-in에 대한 설명, multi-line 가능

- ◆ Wanted_name

- IDA 메뉴에 보이는 plug-in 이름

- ◆ Wanted_hotkey

- plug-in의 단축키 설정

Setting up the Development Environment

- ◆ IDA Pro Plug- in Wizard
 - <http://jeru.ringzero.net/>
 - ◆ 지원 Modules (2008년 5월 31일 기준)
 - plugin modules
 - debugger plugin modules
 - ◆ 미지원 Modules (지원 예정)
 - processor modules
 - loader modules
 - hex- rays modules
- ◆ Install IDA Pro Plug- in Wizard
 - http://jeru.ringzero.net/?page_id=12
 - Setup for Visual C++ Express 2008
 - Put Contents into AppWiz 에 있는 내용을 다음 폴더에 복사
Microsoft Visual Studio 9.0\ VC\ VCWizards\ AppWiz
 - Put Contents into vcprojects 에 있는 내용을 복사
Microsoft Visual Studio 9.0\ VC\ Express\ VCProjects
 - IDA SDK의 include\ intel.hpp 수정
`#include "../idaidp.hpp" => #include "../module/idaidp.hpp"`
- ◆ 정상적으로 plug- in wizard가 설치되면 아래와 같이 Project types 에 IDA Pro 가 포함된다.

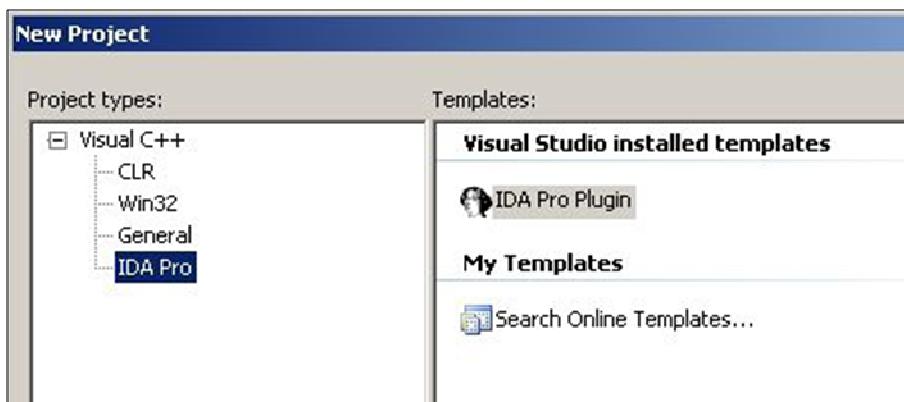


그림 133 Project Type

Simple Plug-in Examples

- ◆ Hello world Plug-in

```
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>

int init(void)
{
    return PLUGIN_OK;
}

void term(void)
{
    return;
}

void run(int arg)
{
    msg("Hello world!\\n");
    return;
}

char comment[] = "Short one line description about the plugin";
char help[] = "My plugin:\\n"
              "\\n"
              "Multi-line\\n"
              "description\\n";
/* Plugin name listed in (Edit | Plugins) */
char wanted_name[] = "helloworld";
/* plug-in hotkey */
char wanted_hotkey[] = "";
plugin_t PLUGIN =
{
    IDP_INTERFACE_VERSION,
    0, // plugin flags
    init, // initialize
    term, // terminate. this pointer may be NULL.
    run, // invoke plugin
```

```
comment, // comment about the plugin
help, // multiline help about the plugin
wanted_name, // the preferred short name of the plugin
wanted_hotkey // the preferred hotkey to run the plugin
};
```

- ◆ 컴파일과 링킹 과정이 정상적으로 끝나면 helloworld.plw 파일이 생성된다.
- ◆ 이 파일을 IDA가 설치된 디렉토리의 plugins 내에 복사
- ◆ plug-in이 정상적으로 설치되면 Edit | Plugins 에 플러그인 목록이 추가된 것이 확인됨

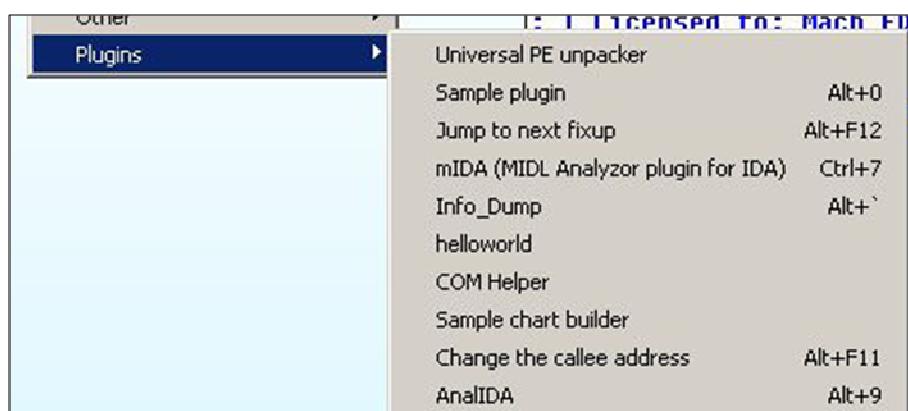


그림 134 helloworld plug-in이 추가된 화면

- ◆ Edit | Plugins | helloworld 를 클릭하면 plug-in 이 실행된다

```
Compiling file 'C:\Lab\IDAscript\globalvariable.idc'...
Executing function 'main'...
Hello world!
```

그림 135 helloworld plug-in 실행하여 메세지를 출력한 화면

| | | |
|---------------------------------|--|---|
| Reverse Engineering Season I | Reverse Engineering Code with IDA Pro | http://www.koreasecurity.org |
|---------------------------------|--|---|

Third- party Scripting Plug- ins

- ◆ IDAPython
 - <http://d-dome.net/idapython>
 - u Windows, Linux, Mac OS X 지원
- ◆ IDARub
 - <http://www.metasploit.com/users/spo0nM/idarub/>
 - 4.9 SDK 이후 새로 추가된 내용은 아직 지원하지 않음.
- ◆ Book
 - Reverse Engineering Code with IDA Pro's Chapter 9
- ◆ Web Site
 - <http://old.idapalace.net/>
 - <http://jeru.ringzero.net/>
 - <http://www.openrce.org/articles/>