

Software
Cracking
/
Reverse engineering

COPY
Right Left

※전체 목차는 정해지지 않았음.

Index

- 제1장. 리버스 기본
 - 1-1. 소프트웨어 리버스/크랙의 정의
 - 1-2. 법률적인 검토 및 최근 국 내외 판례
 - 1-3. 방향
- 제2장. 윈도우 운영체제 기본편
 - 2-1. Inside Windows OS
 - 2-2. Windows 운영체제 구성 요소
 - 2-3. 메모리 관리
 - 2-4. 오브젝트와 핸들 및 프로세스/스레드
 - 2-5. 윈도우 API 와 실행 파일 구조
- 제3장. 윈도우 크랙 기본편
 - 3-1. 소프트웨어 크랙에 이용되는 툴
 - 3-2. 크랙에 필요한 어셈블리어 기초
 - 3-3. 소프트웨어 크랙 준비
 - 3-4. OllyDbg Plug-in 소개
 - 3-5. 날짜제한 Protection 의 크랙
 - 3-6. OllyDbg Plug-in 제작
 - 3-7. ID/Password 기반 Protection 의 크랙 및 Key-Generator 제작
 - 3-8. 크랙에 필요한 어셈블리어 이해
 - 3-9. SoftICE 프로그램 소개와 플러그인
- 제4장. 윈도우 크랙 응용편
 - 4-1. P-Code 크랙
 - 4-2. Visual Basic 크랙 1
 - 4-3. Visual Basic 크랙 2
 - 4-4. Java 기반 프로그램의 크랙
- 제5장. 윈도우 크랙 심화편
 - 5-1. 게임 보안툴의 리버싱
 - 5-2. 다형성 소프트웨어의 리버싱
 - 5-3. 프로젝트 패스워드 크랙
 - 5-4. Microsoft Word/PowerPoint/Excel 패스워드 크랙
 - 5-5. 암호화 통신 프로그램의 리버싱
 - 5-6. 난독 프로그램 코드의 리버싱
- 제6장. 윈도우 Anti-Crack/Anti-Reverse Engineering
- 제7장. 윈도우 리버스 엔지니어링 기본
- 제8장. 리눅스 운영체제 기본편
- 제9장. 리눅스 크랙 기본편
- 제10장. 리눅스 크랙 응용편
- 제11장. 리눅스 크랙 심화편
- 제12장. Palm/Cellvic/Windows CE 크랙 기본편
- 제13장. Palm/Cellvic/Windows CE 크랙 응용편
- 제14장. Mac 크랙 기본편
- 제15장. Mac 크랙 응용편
- 제16장. 윈도우 PE 파일 상세 구조편
- 제17장. 리눅스 실행 파일 구조편
- 제18장. 윈도우 Win32 API/Native API
- 제19장. 어셈블리어 기본편
- 제20장. SoftICE/OllyDbg/gdb/GNU Debug/windbg/macdbg 상세 매뉴얼

머리말(Preface)

목적(Purpose)

이 문서의 목적은 소프트웨어 리버스 엔지니어링과 크랙에 대한 실제적인 방법을 배우는 것이다. 리버스엔지니어링 및 크랙에 대해 이제 막 공부하려는 사람들이 쉽게 이해할 수 있도록 꼭 필요한 내용만(!!!) 상세하게 설명 하였다.

리버스 엔지니어링은 바이너리 소프트웨어를 분석하여 버그, 취약점, 프로그램 로직, 패치 변경 사항 등을 찾아내기 위한 것이며, 드물지만 바이너리로부터 원본 소스코드를 복원하는 것도 포함 한다. 리버싱을 통해 분석된 프로그램은 자신이 원하는 형태로 가공하거나 변경하는데 목적이 있다. 이렇게 변경을 가하는 것을 크랙이라 부르며, 주로 크랙은 리버싱을 통해 분석된 소프트웨어의 보호 장치(복사방지, 등록기술 등)를 풀거나 프로그램 로직을 변경하는 것을 말하고 있다. 이 모든 것은 텍스트 형태의 원본 소스코드를 가지고 있지 않기 때문에 발생된다.

아마도 아래와 같은 단어를 한번 이상 들어 보신 분이라면 그 활용 범위를 알 수 있을 것이다.

『바이러스 분석, 웜 분석, 루트킷 분석, 해킹툴 분석, 악성 프로그램 분석, App 분석, 프로그램 분석, 소프트웨어 공학, 백신 제작, 취약점 분석, 패치 분석, 역패치, 익스플로잇 제작, Exploit, 버그(Bug), 퍼징(Fuzzing), 키젠(Keygen), 멀티로더, 로더, 게임핵, 핵줄, 핵팩, 리버싱, 정품변환, 정품키, 크랙패치, 핵패치, 복사방지, 안티 디버깅, 안티 프로텍터, MUC, 크랙판, 해적판, 시리얼 키, 시디키, 과자, 크래킹, 풀패치, 등록번호, 데드리스팅, 리소스핵, Resource Hack, 메모리핵, 치트키, WPE, 트랜스마커, 언팩, MUP, 패쳐, 엔플러그, 핵슬러, nP 즐, Hs 즐, Xr 즐, 한글화, 머니핵, 노시디, 에뮬, 에뮬레이터, 동굴, 메모리패치, 후킹, Hook, 코드캐이빙, 스택 오버플로우, 힙 오버플로우, 포맷스트링 ……』

보안은 창과 방패라고 이야기 한다. 공격이 최선의 방어라고 하지만 어떻게 공격해 올지 미리 알고 있다면 충분히 위험을 피할 수 있을 것이다. 아무쪼록 이 문서를 통해 국내 소프트웨어 보안이 한 단계 발전하기를 바랄 뿐이다.

감사(Acknowledgements)

감사할 사람이 너무 많습니다.

참고자료(Reference)

참고한 서적과 인터넷 자료가 넘쳐 납니다. 그 분들의 노고에 진심으로 감사를 드립니다.

* 할 수 없는 것과 하지 않는 것은 전혀 다른 이야기입니다. 이 문서는 할 수 없는 분들을 위한 것들이 주로 담겨 있습니다.

제 1 장 Reverse Engineering/Crack

1.1 소프트웨어 리버스 엔지니어링 이란?

들어가며

원고 교정중

제 2 장 Windows OS 기본편

2.1 운영체제 기본 구성요소

들어가며

Windows 운영체제는 편리한 그래픽 사용자 인터페이스(GUI) 뒤에 엄청난 융통성과 강력함을 지닌 많은 이야기들이 숨어있습니다. 필자가 이야기라고 하는 이유는, 1990년대 원도우 운영체제가 탄생한 이 후 컴퓨팅 환경이 급격히 발전함에 따라 눈에 보이지 않은 많은 부분들이 이야기처럼 흘러 변해가고 있기 때문입니다. 오늘의 사실이 내일에는 하나의 이야기 거리로 화자되는 세상이 되어 버렸습니다.

소프트웨어 리버스 엔지니어링을 하기 위해 원도우 운영체제에 대해 배워야 하는 이유는 무엇일까요? 간단한 해답은 운영체제는 리버스 엔지니어링을 하기 위한 핵심 바탕이며, 우리가 원도우 응용프로그램을 리버스 엔지니어링 및 크랙을 시도할 때 모든 소프트웨어는 원도우라는 운영체제 위에서 실행되기 때문입니다. 물론 원도우 운영체제 자체를 리버싱 하려는 사람들도 해당될 것입니다. 그래서 원도우 운영체제의 기본적인 내용을 알고 리버스엔지니어링 및 크랙에 대하여 이야기를 시작하려고 합니다. 여기서는 Windows NT4 이상의 버전과 관련된 내용을 주로 다루고 있으며, 그 하위 버전은 잠깐식 언급하는 정도로 넘어가려고 합니다. 본문에서 말하는 원도우 NT(New Technology)는 Windows NT4 ~ Windows 2003 까지의 포괄적인 원도우 운영체제를 말하고 있습니다.

구성요소와 설계 기준[Components and Architecture]

원도우 운영체제의 특징으로는 7 가지로 압축해서 나타낼 수 있습니다.

(a). 보안성(Secure)

예전 원도우 하위버전과는 다르게 원도우 NT4 부터는 보안을 염두에 두고 디자인 했습니다. 미국 방성 C2 보안등급을 충족하고 있습니다. 물론 100% 완전한 보안성을 현재까지도 만족하고 있지는 않지만, 앞으로 많은 발전을 할 것입니다.

(b). 이식성(Portability)

원도우는 독창적인 언어로 쓰여진 것이 아니라 C 와 C++로 작성되었습니다. 원도우 3.x 나 95/98/Me 는 인텔 계열의 프로세서만을 대상으로 작성되었지만, 원도우 NT 부터는 다른 프로세서나 다른 플랫폼에서 실행할 수 있습니다. 이러한 이식성을 가능하게 만든 것은 원도우 NT 부터 도입되기 시작한 하드웨어 접근 계층(HAL: Hardware Abstraction Layer)이라는 물리적인 하드웨어 접근 통로를 만들었다는 것입니다. 즉 하드웨어 의존 요소를 완전히 분리시킴으로써, 새로운 프로세서나 플랫폼으로 이전 할 때는 HAL 코드만을 작성하면 된다는 것입니다.

(c). 32bit 구조(32-Bit Architecture)

현재는 64-bit 구조로 변경이 완료된 상태이지만, 널리 쓰이는 운영체제는 완전한 32bit 구조를 가지고 있습니다.

운영체제 기본 구성 요소

32bit 운영체제라고는 해도 64-bit 와 16-bit 와의 호환된 구조를 지원하고 있습니다.

④. 메모리 관리(Memory Management)

윈도우 NT 부터는 가상메모리 관리자를 채용하고 있습니다. 가상 메모리는 하드웨어에 직접 접근하지 않고 가상의 메모리 주소 공간에서 소프트웨어와 서로 통신하는 방법을 제공하고 있습니다. 이것은 서로 다른 응용프로그램이 독립된 공간에서 실행되도록 하며, 모든 메모리 접근은 물리적인 메모리 주소를 처리하기 위해 페이지 테이블(page table)이라 불리는 특별한 테이블로 프로세서가 참고합니다. 메모리는 개별 바이트를 위한 테이블 엔트리를 실제적으로 가지고 있으며, 프로세스는 페이지를 메모리로 나누고 있습니다. 좀더 자세한 내용은 뒷 부분에서 설명할 예정입니다.

⑤. 다중 스레드(Multithread)

프로세스(Process)는 실행중인 응용프로그램을 말하며, 응용프로그램이 실행되기 위해 필요한 메모리 주소 공간, 핸들 등의 모든 리소스 객체를 말하고, 스레드(Thread)는 프로세스 내의 독립된 실행 경로를 말합니다. 윈도우 NT 부터는 다중 스레드(Multithread)를 지원하는 선점형(Preemptive) 시스템입니다. 즉, 동시에 다양한 응용프로그램이 동시에 실행될 수 있는 구조를 갖추고 있습니다. 또한 Named Pipe, Remote Procedure Call, Socket, (Distributed)Component Object Model+ 등 IPC(InterProcess Communication)의 분산형 프로세싱 기법을 운영체제 내에서 지원하고 있습니다.

⑥. 다중 프로세서(Multiprocessor)

윈도우 NT 커널은 다중 프로세서를 지원하고 있어서, 향상된 성능과 대용량 데이터 처리를 가능하게 하며 CPU에 매우 의존적인 응용프로그램에 뛰어난 컴퓨팅 환경을 제공하고 있습니다. 기본적으로 IA-32, IA-64, EMT64, DEC Alpha 프로세서에서 다중 프로세서로 실행될 수 있습니다.

⑦. 호환성(Compatible)

DOS 응용프로그램처럼 16비트 코드는 멀티태스킹 시스템에서 동작하도록 설계되어 있지 않으므로 윈도우 NT는 가상 머신이라는 독립적인 공간에서 실행되도록 만들었습니다. 또한 64비트 운영체제인 비스타는 하위 운영체제와의 호환성을 갖도록 만들었습니다. 이처럼 윈도우 운영체제는 오래된 응용프로그램을 계속 사용 가능하도록 호환성을 지니고 있습니다.

윈도우 NT 구조(Windows NT Architecture)

모든 운영체제에는 사용자 코드와 운영체제 코드를 분리/격리시키는 방법이 있는데, 이것은 하드웨어의 메모리 관리 기능에 의해 지원됩니다. 이 기능은 사용자 프로그램이 우연히 또는 고의로 시스템을 파괴하려고 할 때 시스템을 보호하는 역할을하게 됩니다. 물론 제한적이기는 하지만 사용자 모드에서 시스템코드 하단의 하드웨어를 직접 접근하는 방법을 제공하고 있습니다. 윈도우에서는 사용자 응용프로그램 구간을 사용자 모드(User Mode)라고 하고 시스템에서 실행중인 구간을 커널 모드(Kernel Mode)라고 합니다. 커널 모드에서 실행중인 코드는 하드웨어와 시스템 자원을 직접 접근할 수 있지만 사용자 모드는 커널 모드를 통해서만 접근이 가능합니다. 그렇지만 *nix 계열처럼 커널과 사용자 모드간의 구분이 분명하지는 못합니다. 윈도우 NT는 개선된 클라이언트-서버 모델을 가지는 서브시스템과 실행부 구조로 이루어져 있습니다.

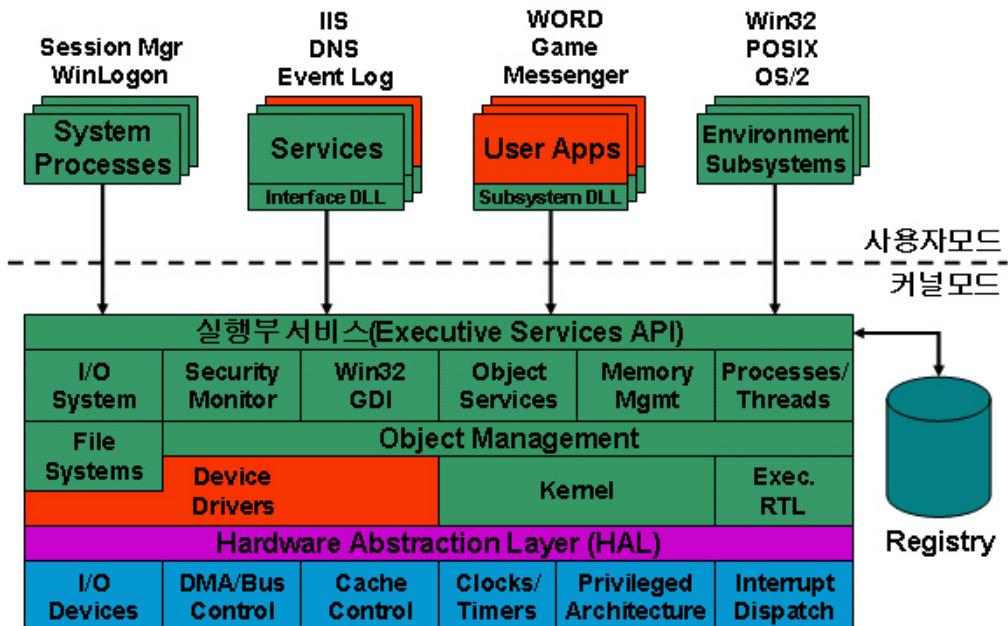


그림 1. Windows 2000 Architecture

이 문서에서 설명하는 내용은 모두 윈도우 32 비트 버전에 국한됩니다. 사실 64 비트 윈도우 버전은 리버싱 관점에서 매우 다른 형태를 지니고 있습니다. 64 비트 프로세서(특정한 Architecture 에 관계없이)는 포인터 및 워드변수가 64 비트 체계를 사용하기 때문에 어셈블리 언어상 매우 다른 형태가 됩니다. IA-32 어셈블리 언어의 32비트 버전을 설명하는 것이 64비트 운영체제를 이해하는 근간(根幹)이 될 것입니다.

가상 메모리와 페이징(Virtual Memory and Paging)

가상 메모리는 부족한 물리적인 메모리 공간을 디스크 공간으로 확장하여 사용하는 개념입니다. 가상 메모리 시스템은 물리적인 메모리가 가득 찰 때 일부 내용을 디스크나 다른 저장 시스템으로 교체하여 사용하는 것을 말합니다. 이렇게 함으로써 물리적으로 존재하는 것보다 많은 양의 메모리를 사용할 수 있습니다. 메모리는 IA-32 프로세서에서 기본적으로 4K 페이지 단위를 사용하며, 2M 나 4M 페이지 단위도 지원합니다. 이러한 페이지 테이블을 사용하여 얻을 수 있는 이점은 다중 주소 공간 생성이 가능하다는 것입니다. 프로세스 주소 공간에서 물리적인 메모리가 존재할 경우 직접 읽어 들여지고, 존재하지 않은 경우 페이지파일로 접근하여 읽어 들입니다. 이렇게 가상 주소공간에 존재하는 메모리 파일을 페이지 파일이라 부르며, 기본적으로 각 하드 디스크 드라이브 루트에 “pagefile.sys”로 저장되어 있습니다.

커널 메모리(Kernel Memory)

그림 및 원고 교정 후 다시 붙이기(filename:M2-sub-Kernel_memory)

메모리 관리 APIs(Memory Management APIs)

그림 및 원고 교정 후 다시 붙이기(filename:M2-sub-Memory_Management_api)

객체와 핸들(Objects and Handles)

그림 및 원고 교정 후 다시 붙이기(filename:M2-sub-Object_Handles)

프로세스와 스레드(Processes and Threads)

프로세스와 스레드를 설명하기 이전에 먼저 멀티태스킹(Multitasking)에 대해 간단히 알아 두어야 합니다. 멀티태스킹은 운영체제가 동시에 여러 응용프로그램을 실행시킬 수 있다는 것을 말합니다. 사실, 아주 정확하게 말하면 2 개 이상의 CPU 를 가진 디중 프로세서에서만 진정한 멀티태스킹이 이루어 질 수 있습니다. 단일 CPU 를 가진 컴퓨터에서는 실행중인 프로그램들을 빠르게 전환시켜 동시에 동작되고 있다는 것처럼 느끼게 만들어 주는 것뿐입니다. 윈도우 NT 에서는 이처럼 자연스럽게 프로그램간의 이동을 수행하는 선점형(Preemptive) 멀티태스킹을 채용하고 있습니다. 응용프로그램의 우선권을 기초로 데이터 읽기/쓰기를 결정하며, 내부적인 기준에 의해 주어진 시기에 응용프로그램의 실행을 결정하기도 합니다.

프로세스(Processes)는 응용프로그램이 사용하는 모든 자원을 소유하는 객체를 말합니다. 스레드(Thread)는 프로세스 내에서 독립적으로 실행되는 객체입니다. 스레드는 프로세스의 가상 메모리 공간, 코드와 전역 데이터를 공유하지만 서로간에 독립적으로 실행되고 독립적인 스택을 가지고 있습니다. 가장 간단한 예로 문서 출력을 하는 독립된 스레드를 실행시키면서 출력되는 동안 문서를 편집할 수 있습니다.

컨텍스트 스위칭(Context Switching)

스레드의 컨텍스트(Context)는 프로세서 레지스터 내용 및 스택포인터(ESP)와 스레드가 실행되고 있는 메모리 공간에 대한 포인터를 포함하여 스레드의 현재 상태를 스냅샷(Snapshot)한 데이터로 구성됩니다. 시스템이 동일한 프로세서에서 실행중인 스레드간 전환을 위해 현재 실행중인 스레드를 저장하고 새로 실행하기 원하는 스레드를

운영체제 기본 구성 요소

위한 컨텍스트를 재 작성해주어야 하는데, 이것을 컨텍스트 스위칭이라고 합니다. 이는 윈도우 NT 운영체제가 근본적으로 마이크로 커널을 사용하기 때문에 발생됩니다.

동기화 객체(Synchronization Objects)

그림 및 원고 교정 후 다시 붙이기(filename:M2-sub-Sync_Objects)

IPC 와 네트워킹(IPC and Networking)

그림 및 원고 교정 후 다시 붙이기(filename:M2-sub-IPC_Network)

에러 처리>Error processing

그림 및 원고 교정 후 다시 붙이기(filename:M2-sub-Error_processing)

프로세스 실행 단계(Process Initialization Sequence)

그림 및 원고 교정 후 다시 붙이기(filename:M2-sub-Process_Initialization_step)

제 3 장 Windows 크랙 기본편

3.1 소프트웨어 크랙에 이용되는 툴

들어가며

소프트웨어 크랙은 단조로운 작업으로 Protection 정도에 따라 다르기도 하지만 끈기와 많은 시간을 필요로 합니다. 그 시간과 노력을 줄이기 위해서는 크랙 툴을 다루는 법을 충분히 습득하고 있을 필요가 있습니다. 그러나 크랙 툴은 복잡한 기능을 가진 것이 많고 습득하려면 많은 시간을 필요로 합니다. 최근 들어 리버스 엔지니어링 툴들이 기능이나 사용자 편의성을 대폭 향상 시켰다고는 하지만 여전히 툴을 습득하는데 많은 시간을 필요로 합니다. 그렇지만 소프트웨어 크랙을 쉽게 도와주는 플러그인(Plug-in)이 많이 공개되어 있고, 자체 제작하여 툴에 포함 시킬 수도 있습니다.

이번 장에서는 주로 Windows 어플리케이션을 크랙 할 때 이용되는 툴을 장르별로 소개합니다.

바이너리 에디터(Binary Editors, Hex Editors)

크랙 대상이 되는 바이너리 데이터의 참조, 편집을 행하는 데는 필수 툴입니다. 특히 실행가능 파일에 대한 패치로 이용됩니다. 상용 레벨 수준의 바이너리 에디터는 많이 공개되어 있기 때문에 자신에게 맞는 것을 선택하면 문제 없습니다. 단지 바이너리 내부 표시문자 코드로서 최소한 2 바이트 코드 문자인 Korean 와 Simplified Chinese, Traditional Chinese, Shift-JIS, Unicode 가 선택가능하고, 대용량 파일도 문제없이 읽어 들일 수 있는 것이 좋습니다.

아마도 바이너리 에디터로 널리 이용되고 있는 툴은 Hex WorkShop¹, Hex Edit², XVI32³ 등이 있을 것입니다. 이들 말고도 바이너리 에디터 기능 이외의 다양한 플러그인 기능을 내장한 툴도 많이 있으므로 본인이 직접 툴을 다운 받아 설치해보고 하나를 선택해서 완벽하게 사용할 줄 만 알면 됩니다. 각 툴은 바이너리 편집을 수행하는 데는 서로 다를 바 없고 UI나 참조기능이 조금 다를 뿐입니다. 툴은 단지 크랙을 좀더 쉽게 도와주는 도구에 지나지 않습니다. 크랙 및 리버스 엔지니어링의 원리와 과정을 잘 이해하는 것이 더 중요할 것입니다.

¹ Hex WorkShop: <http://www.bpssoft.com/downloads/index.html>

² Hex Edit: <http://www.expertcomsoft.com/download.htm>

³ XVI32: <http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm>

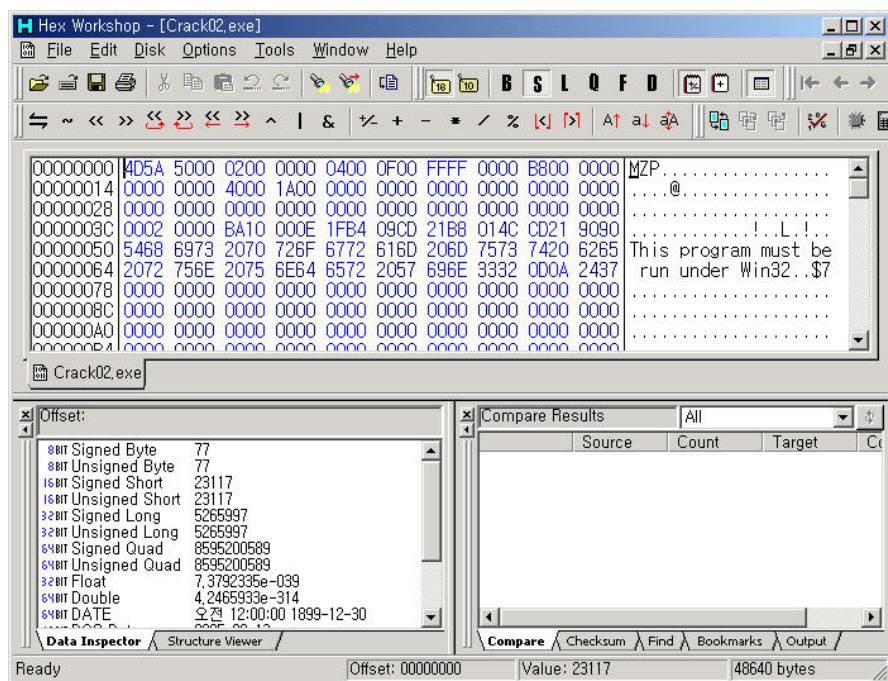


그림. Hex Workshop

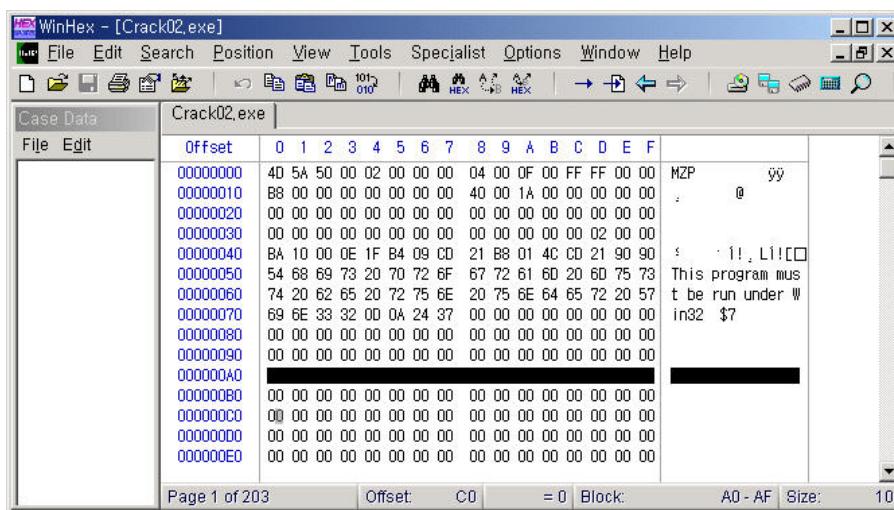


그림. WinHex

일본 크래커 중에는 【QuickBe⁴】를 추천하는 사람도 있습니다. 이것은 일본 【Team Next Generation】(1999년에 해산) 소속의 KK씨가 작성한 바이너리 에디터입니다. 크래킹에 정통한 분이 작성한 것인 만큼 FCJ 형식의 크랙 패치 작성 기능을 포함하여 크랙 면에서도 강력하며 불필요한 기능을 제거한 깔끔한 구조로 동작하기 때문에 공식배포 페이지가 존재하지 않는 현재도 일본에서는 끊임없이 사용되고 있습니다.

외국의 많은 사람들은 【Hiew⁵】를 추천하고 있습니다. 다른 GUI(Graphic User Interface)바이너리 편집 툴과는 달리 CUI(Command User Interface)입력 형태를 취하고 있습니다. 처음 크랙 및 리버싱을 하시는 분들이 어려울 수도 있지만 바이너리 편집 이외의 다양한 기능(어셈블, 역어셈블, 파일 헤더 분석 등)과 속도 및 키보드 매크로

⁴ QuickBe: <http://cowscorpion.com/dl/QuickBe.html>

⁵ Hiew: <http://webhost.kemtel.ru/~sen/>

소프트웨어 크랙에 이용되는 툴

기능이 매우 뛰어나서 현재까지도 많은 크랙, 리버싱 문서에 등장하고 있습니다. 한글 윈도우 명령창에서는 코드 페이지를 영문(chcp 437)으로 변경하면 라인 문자 깨짐을 방지할 수 있습니다.

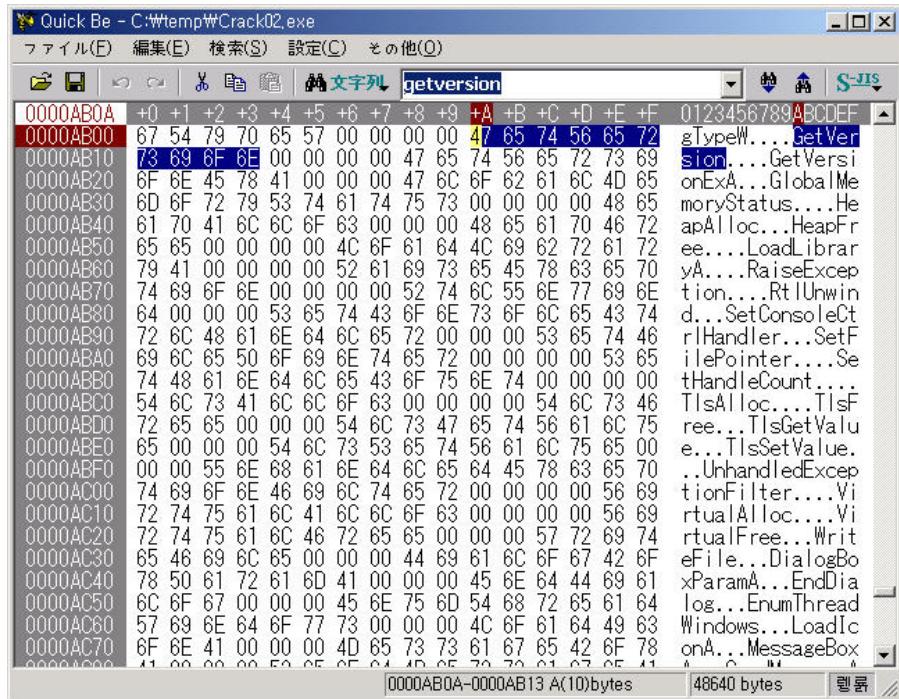


그림. QuickBe

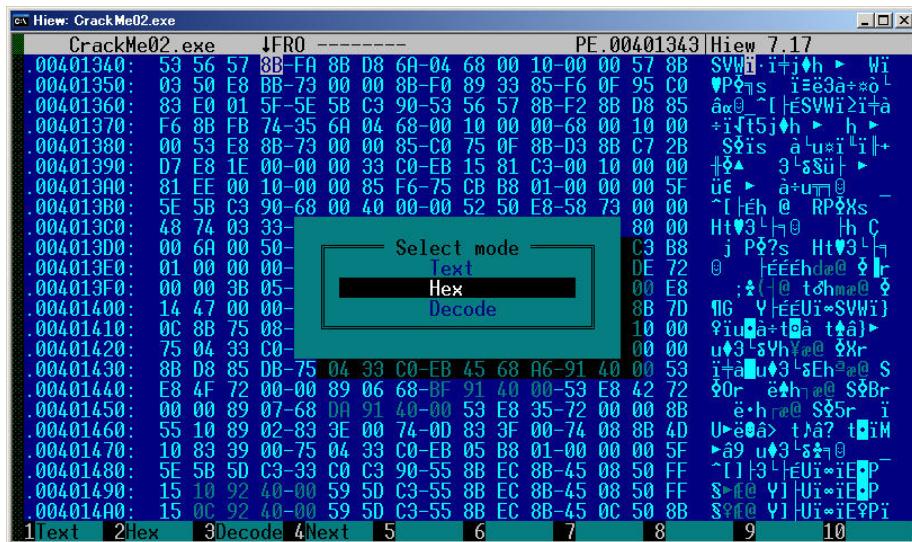


그림. Hiew

디버거 (Debuggers)

디버거는 원래 소프트웨어 개발상에서 인위적으로 버그를 발생시키기 위해 사용되는 것입니다. 디버그 대상이 되는 프로그램을 실행하면서 그 동작 흐름을 확인할 수 있기 때문에 프로그램의 제어 플로우(흐름)를 이해하기에는 더없이 유용한 툴입니다. 제 3 자인 크래커는 소스 코드를 가지고 있지 않기 때문에 대신 실행가능 파일로부터 얻을 수 있는 역 어셈블 코드를 대상으로 하여 디버그를 시행합니다. 크래커는 코드 중에서 체크 루틴을 찾기 위해 브레이크 포인트(Breakpoint-흔히 “브뽀”라고 말함) 설정을 행하는 등 원하는 부분을 찾기 위해 노력합니다. 그 뒤 체크 루틴을 한 명령씩 실행해 가면서 레지스터 메모리 등의 값을 모니터링, 편집하면서 세부 동작을 파악합니다.

무료로 배포되고 있는 【OllyDbg】⁶는 유저 모드 디버그로서 널리 사용되고 있습니다. 이 툴은 기능이나 사용자 조작이 매우 편리해서 지금까지의 Defact Standard였던 【SoftIce】⁷를 대신하는 디버거로 대체되고 있는 듯 합니다. 2006년 시점에서의 최신 버전은 1.10입니다. 현재는 후속 판으로 64bit 디버깅이 가능한 버전 2.0 개발⁸이 진행되고 있습니다. 2년 넘도록 2.0 버전이 릴리즈 되지 않는 이유는 여러 가지 있지만 개발 진행 사항을 보면 기대해볼 만 합니다. 제작자인 Oleh Yuschuk는 우크라이나 태생으로 현재 독일에서 살고 있습니다.

OllyDbg 의 장점중의 하나는 외부 플러그인 제작이 가능하다는 것입니다. 예를 들어 디버거를 탐지하는 프로그램을 디버깅 하기 위해서는 디버깅 모드를 숨길 필요가 있습니다. 이런 디버깅 숨김 플러그인을 외부에서 불러들여와 사용할 수 있다는 것입니다.

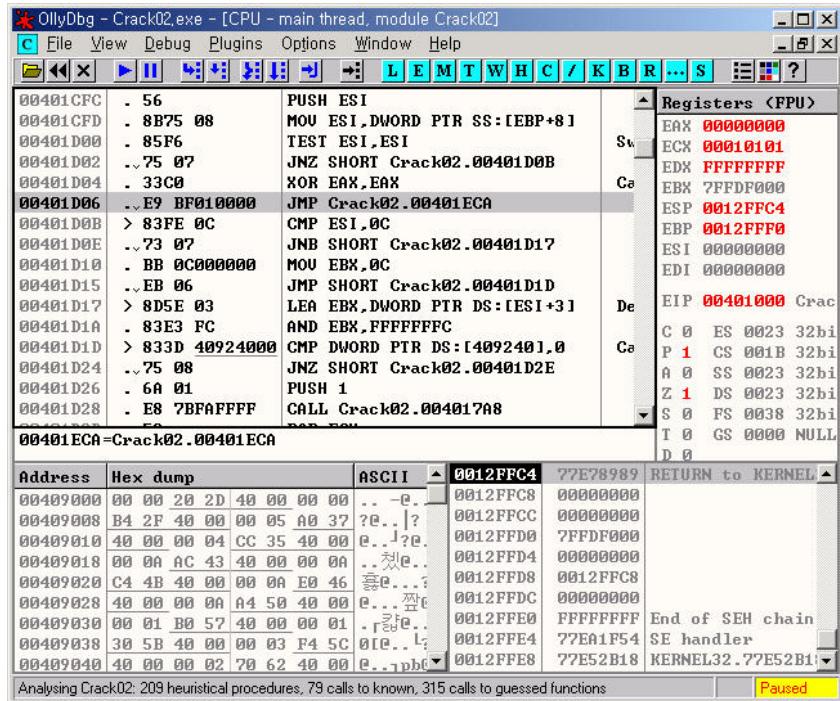


그림. OllyDbg

⁶ Olldby: <http://www.ollydbg.de/>

⁷ SoftICE: <http://www.compuware.co.kr/products/driverstudio/softice/>

⁸ Ollydbg: <http://www.ollydbg.de/whatsnew.htm>

소프트웨어 크랙에 이용되는 툴

디버거를 논한다면 Compuware 사의 【SoftICE】를 빼 놓을 수 없습니다. 디바이스 드라이버 개발 지원 툴로서 설계된 커널 모드 디버거입니다. 그렇기 때문에 매우 강력한 디버그가 가능하고 윈도우 크랙을 수행하는데 필수 툴입니다. 단, 일반적으로 조작이 복잡한 디버거 중에서도 대단히 복잡하고, 게다가 조작방법이 커맨드 라인 입력과 단축키 만으로 되어 있기 때문에 습득에 매우 많은 시간을 필요로 합니다. 또한 윈도우 커널과 직접 통신을 하기 위해 윈도우와 함께 시작되어야만 합니다. 그렇기 때문에 보통은 소프트웨어 크랙이나 디버깅을 위해 별도 시스템을 구축하거나 Dual 부팅을 하는 사람들이 많습니다. 개발 중단을 예고하고 있지만 64bit 이전 윈도우 환경에서는 막강한 기능을 가지고 있기 때문에 SoftICE는 다른 장에서 상세하게 설명하도록 하겠습니다.

위에서 소개한 디버거는 세부적인 개별 명령까지 상세하게 따라가는 것에 중점을 둔 것이지만, 이 외에도 프로그램 구동을 감시하고 상세한 로그를 출력하는 것에 주력한 것도 있습니다. Compuware사의 【SmartCheck】⁹나 MutekSolution사의 【BugTrapper】¹⁰가 대표적입니다. 이들 프로그램은 호출하는 외부 함수와 그 인수 반환 값을 상세하게 출력할 수 있기 때문에 VB, VC, .NET 표준함수의 처리와 동적 라이브러리(DLL), 윈도우 이벤트 추적 등 윈도우 Visual 프로그램 계열 소프트웨어에 특히 효과를 발휘합니다. 디버거는 작성된 프로그램 언어에 종속적인 경우도 많으므로 실행파일이 작성된 언어를 파악하고 효과적인 디버거를 선택하는 것도 중요합니다. 안타까운 사실은 많은 좋은 디버거 소프트웨어들이 개발이 중단되거나 MS, IBM, SUN과 같은 기업 솔루션에 내장 되버렸다는 것입니다.

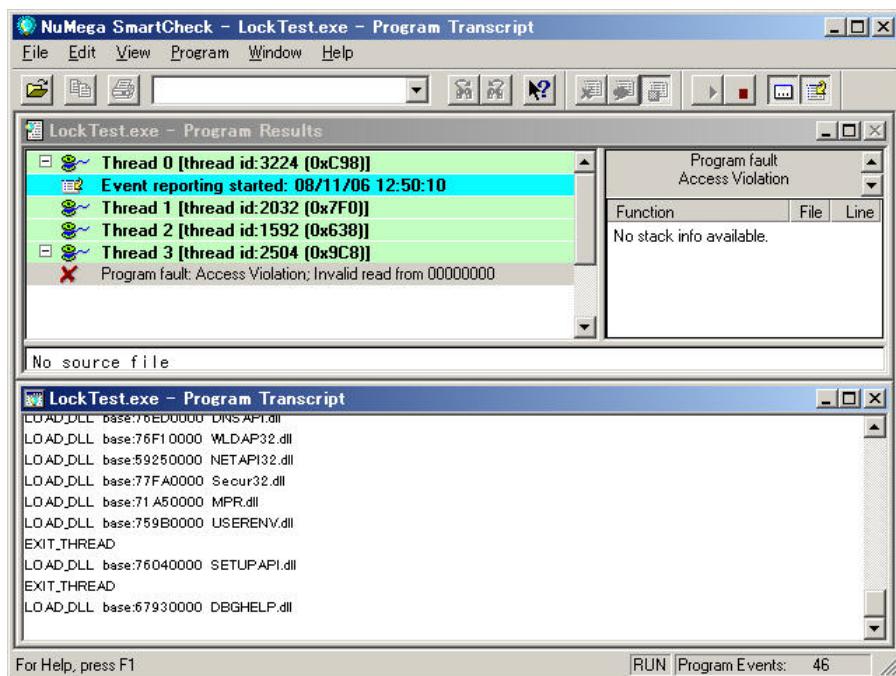


그림. SmartCheck

모든 툴은 악성 바이러스에 감염되는 상태가 없도록 반드시 공식 배포 사이트나 신뢰된 사이트를 통해 다운로드 하시기 바랍니다.

⁹ SmartCheck: <http://www.compuware.com/products/devpartner/studio.htm>

¹⁰ Mutek Solution 사의 【BugTrapper】: 2002년에 사명이 【Identify Software Ltd.】로 변경되어, 현재는 AppSight라고 불리는 개발지원 솔루션에 포함되어 있습니다. (<http://www.identify.com/products/index.php>)

역 어셈블러 (Disassemblers)

역 어셈블러를 쉽게 말하면, CPU 가 해석 가능한 기계어가 포함된 실행 파일을 사람이(적어도 16 진수 값의 나열보다는) 이해하기 쉬운 어셈블리 언어 형태로 변환하는 것입니다. 이전에 설명한 디버거에도 역어셈블 기능을 포함한 것이 있지만 전문적인 역 어셈블러들은 그것보다 상세한 정보를 출력하는 것이 많습니다.

역 어셈블러에 의해 출력되는 역어셈블 목록은 원본 실행 파일의 사이즈와 비교하면 상당히 큰 텍스트 데이터가 됩니다. 이것을 처음부터 차례대로 눈으로 따라가다 보면 시간이 많이 걸리기 때문에 통상은 다른 툴과 병행하여 사용합니다. 또 기계어는 디버깅 툴이 지원하는 CPU(or 호환 CPU)의 기계어밖에 대응하지 않습니다.

DataRescue사가 배포하는 【IDA Pro】¹¹ 는 Z80 으로부터 Pentium4 까지 50 종이 넘는 CPU에 대응하는 막강한 상용 역 어셈블러입니다. 어셈블러를 해석하는 기능이 매우 강력하기 때문에, 타 어셈블러에 비해 분석 시 상당한 시간을 요하지만 아주 상세한 역어셈블 목록을 출력할 수 있습니다. IDA의 최대 매력은 FLIRT(Fast Library Identification and Recognition Technology)라 불리는 기계어의 코드로부터 컴파일러 특유의 라이브러리 함수를 산출해 낼 수 있는 강력한 기능입니다. 이것에 의해 여분의 코드를 추적할 필요가 없어지기 때문에 시간과 노력이 절약됩니다. 단지 기능이 매우 다양하기 때문에 툴을 습득하는 데에 시간이 많이 걸린다는 것이 단점입니다.

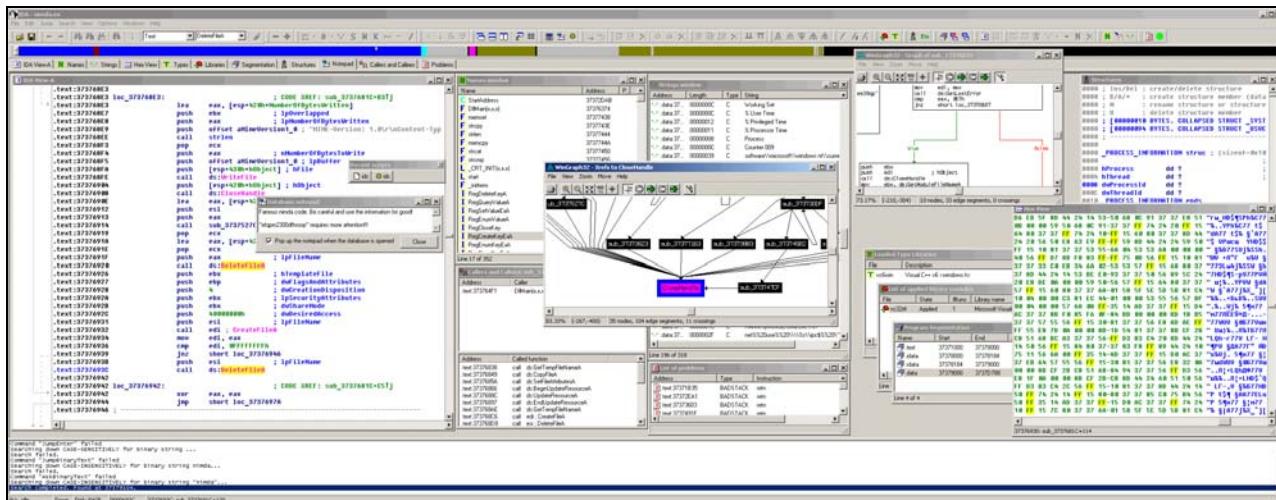


그림. IDA Pro

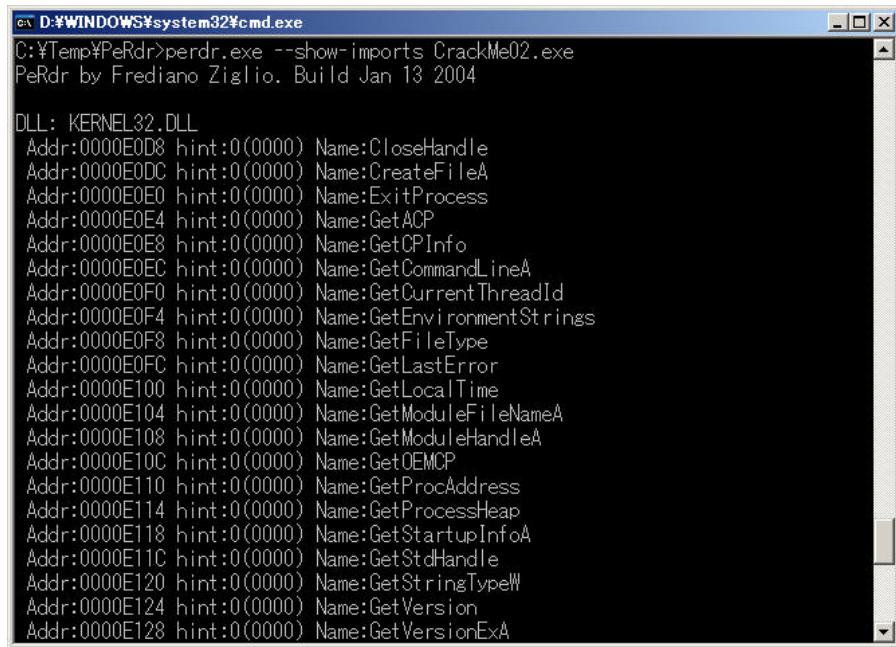
그 외에도 호출된 주소를 열거하는 cross reference 기능이나 참조문자열 표시가 가능한 【PeRdr】¹² 등이 있습니다. OllyDbg 등장 이전은 독특한 인터페이스의 디버거 기능도 갖춘 【W32Dasm】¹³ 가 특히 크래커들 사이에서 인기가 있었습니다. 현재까지도 제 3 자에 의해 기능이 추가된 크랙판이 나돌고 있는 것을 보면 그 인기를 짐작할 수 있습니다.

¹¹ IDA Pro: <http://www.datarescue.com/idabase/>

¹² PeRdr: http://sourceforge.net/project/showfiles.php?group_id=26371

¹³ W32Dasm: <http://www.google.com/search?hl=en&q=W32Dasm>

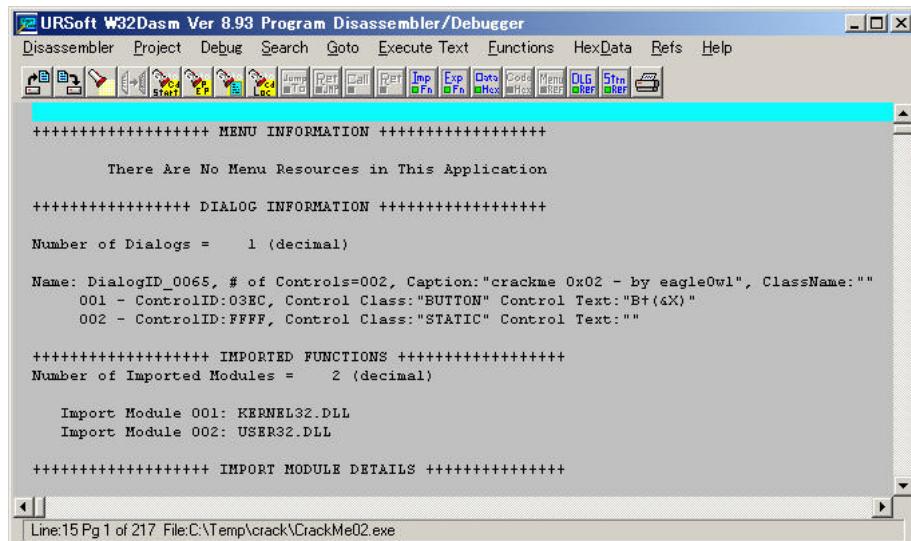
소프트웨어 크랙에 이용되는 툴



```
D:\WINDOWS\system32\cmd.exe
C:\Temp\PeRdr>perdr.exe --show-imports CrackMe02.exe
PeRdr by Frediano Ziglio. Build Jan 13 2004

DLL: KERNEL32.DLL
  Addr:0000E008 hint:0(0000) Name:CloseHandle
  Addr:0000E0DC hint:0(0000) Name>CreateFileA
  Addr:0000E0E0 hint:0(0000) Name:ExitProcess
  Addr:0000E0E4 hint:0(0000) Name:GetACP
  Addr:0000E0E8 hint:0(0000) Name:GetCPIInfo
  Addr:0000E0EC hint:0(0000) Name:GetCommandLineA
  Addr:0000E0F0 hint:0(0000) Name:GetCurrentThreadId
  Addr:0000E0F4 hint:0(0000) Name:GetEnvironmentStrings
  Addr:0000E0F8 hint:0(0000) Name:GetFileType
  Addr:0000E0FC hint:0(0000) Name:GetLastError
  Addr:0000E100 hint:0(0000) Name:GetLocalTime
  Addr:0000E104 hint:0(0000) Name:GetModuleFileNameA
  Addr:0000E108 hint:0(0000) Name:GetModuleHandleA
  Addr:0000E10C hint:0(0000) Name:GetOEMCP
  Addr:0000E110 hint:0(0000) Name:GetProcAddress
  Addr:0000E114 hint:0(0000) Name:GetProcessHeap
  Addr:0000E118 hint:0(0000) Name:GetStartupInfoA
  Addr:0000E11C hint:0(0000) Name:GetStdHandle
  Addr:0000E120 hint:0(0000) Name:GetStringTypeW
  Addr:0000E124 hint:0(0000) Name:GetVersion
  Addr:0000E128 hint:0(0000) Name:GetVersionExA
```

그림. PeRdr



```
URSoft W32Dasm Ver 8.93 Program Disassembler/Debugger
Disassembler Project Debug Search Goto Execute Text Functions HexData Rets Help
File Edit View Insert Options Tools Help
+++++ MENU INFORMATION ++++++
There Are No Menu Resources in This Application
+++++ DIALOG INFORMATION ++++++
Number of Dialogs = 1 (decimal)
Name: DialogID_0065, # of Controls=002, Caption:"crackme 0x02 - by eagleOwl", ClassName:""
  001 - ControlID:03EC, Control Class:"BUTTON" Control Text:"B+(4X)"
  002 - ControlID:FFFF, Control Class:"STATIC" Control Text:""
+++++ IMPORTED FUNCTIONS ++++++
Number of Imported Modules = 2 (decimal)
Import Module 001: KERNEL32.DLL
Import Module 002: USER32.DLL
+++++ IMPORT MODULE DETAILS ++++++
Line:15 Pg 1 of 217 File:C:\Temp\crack\CrackMe02.exe
```

그림. W32DASM

그 외에도 특정 프로그래밍 언어로 작성된 실행 파일을 대상으로 한정한 역 어셈블러도 존재합니다. Delphi로 작성된 실행 파일을 대상으로 한 【DeDe】¹⁴는 크랙 대상이 Delphi언어로 작성된 것이라면 이것 하나로 요리할 수 있을 정도로 막강한 기능을 가지고 있습니다. 독자 형식으로 정의된 품 구조 등을 해석하여 윈도우 이벤트와 대응된 명령 코드 분석, 바이너리 편집 등 상세한 정보를 수집할 수 있습니다. 그러나 2002년 DeDe가 업데이트 중단이 되고 난 후(또한 공식 배포 사이트인 www.balbaro.com 접속 불가) Delphi6과 컴파일러 구조가 변경된 Delphi2006 버전으로 작성된 실행 파일 분석시 DeDe가 오류를 발생 시킬 수 있으니 주의해야 합니다. 현재는 Delphi2006의 내부 디버깅 기능으로도 충분하다고 봅니다.

¹⁴ DeDe: <http://www.balbaro.com/dede/>

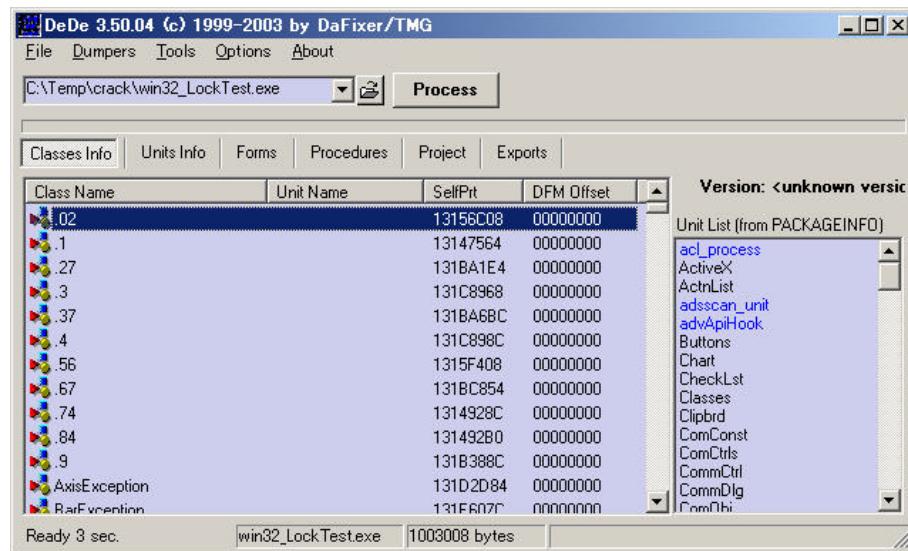


그림. DeDe

아래 표는 잘 알려진 리버싱 툴의 Disassembly architecture에 대한 것입니다.

역어셈블러/디버거 이름	역어셈블 방법
OllyDbg	Recursive traversal
W32DASM	Recursive traversal
IDA Pro	Recursive traversal
PEBrowse Professional	Recursive traversal
SoftICE(Numega)	Linear sweep
WinDbg(Microsoft)	Linear sweep

디컴파일러 (Decompilers)

디컴파일러란, 기계어나 오브젝트언어, 바이트 코드 등의 명령 코드군을, C 언어 등의 고급언어로 복원을 시행하는 툴입니다.

디컴파일러는 의외로 오랜 역사를 가지고 있습니다. 1960년 미국 해군 전자 연구소에 소속한 J.K.Donnelly와 H.Englandner가, NELIAC이라는 ALGOL 파생언어를 대상으로 하여 개발한 【Donnelly-Neliac(D-Nelac) decompiler】가 최초로 되어 있습니다. 디컴파일러는 역 어셈블러와 비슷하지만 저급언어에서 고급언어로의 변환이라는 고도한 처리가 가해지는 점에서 크게 다릅니다. 역 어셈블 목록과 비교하면 가독성이 현격히 높아지기 때문에 프로그램을 소스 레벨로 조사할 때에 큰 위력을 발휘합니다.

일반적으로 기계어로부터 소스 코드를 복원하는 것은 매우 어렵습니다. 그래서인지 실용성 있는 레벨의 쓸만한 것은 존재하지 않는 것 같습니다. 그러나 Java 바이트 코드나 .NET framework 상에서 동작하는 소프트웨어의 중간언어 MSIL(Microsoft Intermediate Language)에 대해서는 실용적인 디컴파일러가 존재합니다.

소프트웨어 크랙에 이용되는 툴

【DJ Java Decompiler】¹⁵는 Java 대응 디컴파일러 【Jad】¹⁶의 Front-end 툴로서, 클래스 파일 단위에서 디컴파일을 수행할 수 있습니다. 실제로 자작 프로그램에 대하여 디컴파일을 시행해 보면 알겠지만, 원본 소스코드와 거의 유사할 정도로 높은 정밀도로 소스 코드를 복원하는 것이 가능합니다. 【DJ Java Decompiler】를 이용한 크랙 방법은 차후 소개할 예정입니다.

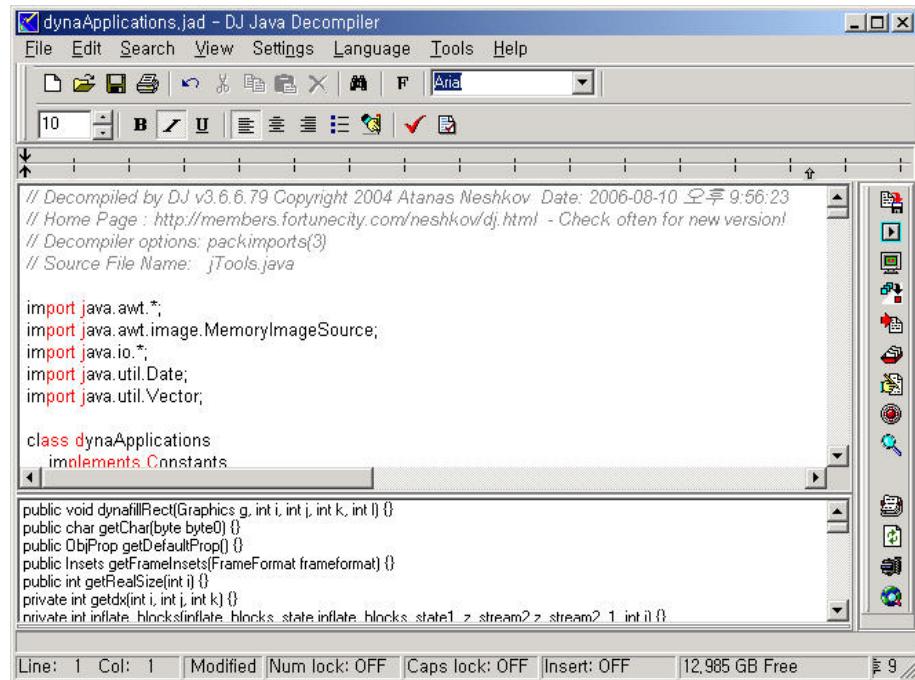


그림. DJ Java Decompiler

.NET용 디컴파일러로서 【.NET Reflector】¹⁷가 있습니다. 이것은 .NET으로 작성된 소프트웨어라면 작성언어가 C#, VB.NET, Delphi¹⁸라고 해도 문제없이 디컴파일이 가능합니다. 게다가 작성언어에 관계없이 출력하는 소스 코드 형태를 C#, VB.NET, Delphi, IL, MC++, Chrome형태로 출력할 수 있기 때문에 C#로 작성한 프로그램을 VB.NET 소스 코드 형식으로 컨버팅하는 것도 가능합니다.

¹⁵ DJ Java Decompile: <http://members.fortunecity.com/neshkov/dj.html>

¹⁶ JAD: <http://www.kpdus.com/jad.html>

¹⁷ .NET Reflector: <http://www.aisto.com/roeder/dotnet/>

¹⁸ Delphi8: 정확히 【Delphi 8 for the Microsoft .NET Framework】. Delphi8부터 .NET에서 컴파일 가능.

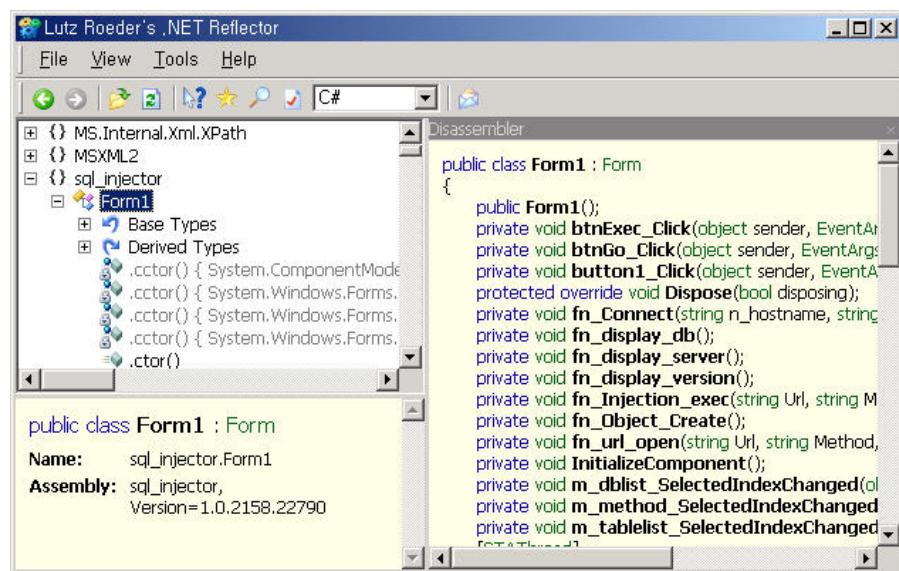


그림.. .NET Reflector

리소스 편집기 (Resource Editors)

실행 파일이나 DDL 파일 등에 포함되는 리소스 정보를 표시하거나 편집하기 위한 툴입니다. 주로 영어/중국어/일본어판 소프트웨어를 한글화하는 경우에 쓰이지만, 크랙 과정에서 몇몇 단서를 파악하기 위해 쓰이기도 합니다. 여기서 말하는 리소스는 리소스 파일(*.rc, *.res)로 정의된 실행가능 파일이나 DLL 파일 등에 포함되는 Windows 표준 리소스(Dialog, Cursor, Bitmap Menu, Sound File 등)을 가리킵니다.

그러나 Visual Basic으로 작성된 프로그램은 리소스 파일을 사용하지 않는 구조이기 때문에 리소스 에디터에 의한 편집은 불가능합니다. 이것은 프로그램으로 리소스를 작성(특히 윈도우가 많음)하고 있는 경우에 대해서도 같다고 할 수 있습니다. 또한 비표준 실행파일 포맷(흔히 실행파일 압축 -UPX, PECompat 등)에 대해서도 편집이 불가능합니다. 이러한 파일들은 표준 실행 파일 포맷 형태로 다시 복원한 후 편집을 수행할 수 있습니다.

대표적인 리소스 에디터는 【Resource Hacker】¹⁹를 들 수 있습니다. 이것은 초창기 리소스 편집기로 많이 사용했던 것이며 한글판이 있는데다가 무료로 배포 되고 있습니다. 2002년 개발이 중단되고 소스코드를 물려받아 eXeScope, XN Resource Editor로 변경되었습니다.

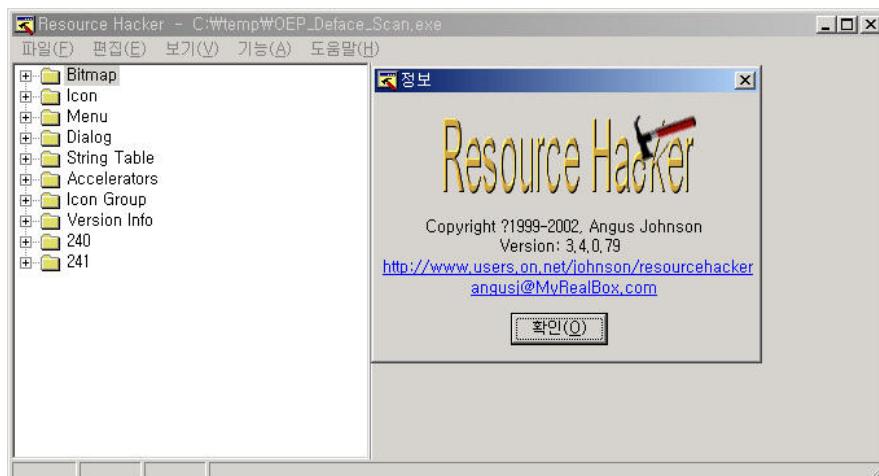


그림. Resource Hacker

쉐어웨어인 【eXeScope】²⁰는 일본인이 개발한 리소스 에디터로 애용자가 많은 것으로 알려져 있습니다. 리소스 에디터 기능 이외에 실행 파일 헤더(File header) 표시나 외부 참조 DLL 함수 표시가 가능하기 때문에 【Resource Hacker】 보다는 이쪽을 더 많이 사용하고 있습니다.

최근 【XN Resource Editor】²¹도 사용자가 많아지고 있습니다. Delphi2006으로 작성된 이 프로그램은 소스코드가 공개되어 있으며 깔끔한 UI와 Delphi언어로 작성된 Dialog 폼을 모두 편집할 수 있습니다. 또한

¹⁹ Resource Hacker: <http://www.users.on.net/johnson/resourcehacker/>

²⁰ eXeScope: <http://hp.vector.co.jp/authors/VA003525/Eindex.htm>

²¹ XN Resource Editor: <http://www.wilsonc.demon.co.uk/d10resourceeditor.htm>

eXeScope 대부분의 기능도 포함하고 있습니다.

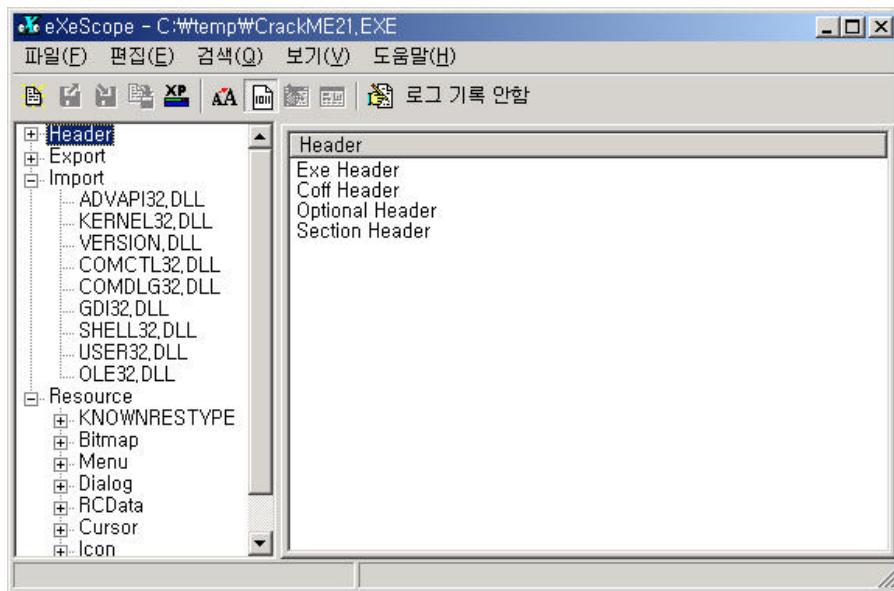


그림. eXeScope

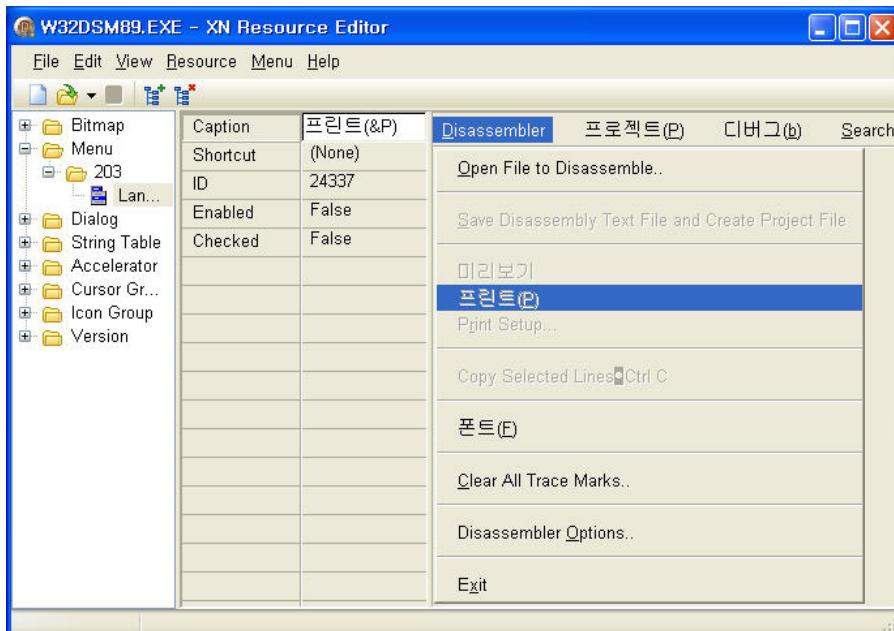


그림. XN Resource Editor

리소스 에디터는 상용 제품도 많이 있지만 위의 공개용 소프트웨어로도 충분히 사용 가능합니다. 상용제품으로는 Restorator 2006²²이나 Resource Builder²³ 등이 쓸만한 것 같습니다. 리소스 편집을 손쉽게 하기 위한 편리한 기능과 파일 탐색, 검색, 파일압축/해제 등의 기능도 존재합니다.

그 외 운영체제(Mac, Palm, Linux) 리소스 에디터는 별도로 존재합니다. 관련 운영체제 크랙편에서 다룰 예정이므로 참고하시기 바랍니다.

²² Restorator 2006: <http://www.bome.com/Restorator/download.html>

²³ Resource Builder: <http://www.resource-builder.com/index.html>

모니터링 툴 (Monitoring Tools)

SysInternals가 배포하고 있는 레지스트리 액세스 감시 툴 【Regmon】²⁴이나 파일 액세스 감시 툴 【Filemon】²⁵은, 레지스트리, 파일 액세스를 감시하고 로그를 저장할 수 있는 툴입니다. 각각 Windows9x, NT/2000/2003, XP 판이 준비되어 있고, 배포 사이트에는 그 외에도 유용한 모니터링 툴이 다수 공개되어 있습니다. 1996년부터 원도우에 사용되는 각종 유용한 툴을 배포하고 있던 Sysinternals사가 최근 2006년 7월에 마이크로소프트사에 인수되면서 Microsoft Technet사이트에 통합되었습니다. 향후 몇몇 툴은 원도우에 내장된 기능으로 동작할 수 도 있을 것 같습니다.

#	Time	Process	Request	Path
118...	57.38181305	explorer.exe:1704	OpenKey	HKCU\CLSID\{9d3c85d1-f877-11d0-b083-004
118...	57.38925171	explorer.exe:1704	OpenKey	HKCR\CLSID\{9d3c85d1-f877-11d0-b083-004
118...	57.38927841	explorer.exe:1704	CloseKey	HKCR\CLSID\{9d3c85d1-f877-11d0-b083-004
118...	57.38930130	explorer.exe:1704	Enumerate...	HKCR\CLSID
118...	57.38932037	explorer.exe:1704	QueryKey	HKCU
118...	57.38934708	explorer.exe:1704	OpenKey	HKCU\CLSID\{9d66bb5e-8a2c-40b8-a6bd-2de0
118...	57.38938522	explorer.exe:1704	OpenKey	HKCR\CLSID\{9d66bb5e-8a2c-40b8-a6bd-2de0
118...	57.38940430	explorer.exe:1704	QueryKey	HKCR\CLSID\{9d66bb5e-8a2c-40b8-a6bd-2de0
118...	57.38943863	explorer.exe:1704	OpenKey	HKCU\CLSID\{9d66bb5e-8a2c-40b8-a6bd-2de0
118...	57.38945770	explorer.exe:1704	OpenKey	HKCR\CLSID\{9d66bb5e-8a2c-40b8-a6bd-2de0
118...	57.38946915	explorer.exe:1704	CloseKey	HKCR\CLSID\{9d66bb5e-8a2c-40b8-a6bd-2de0
118...	57.38948441	explorer.exe:1704	Enumerate...	HKCR\CLSID
118...	57.39547723	explorer.exe:1704	QueryKey	HKCU

그림 15. Regmon

#	T..	Process	Request	Path	Result	Other
3381	?	IEXPLORE.EXE:2252	QUERY INFORMATION	C:\WINNT\system32\url.dll	SUCCESS	Attributes: A
3382	?	IEXPLORE.EXE:2252	OPEN	C:\WINNT\system32\url.dll	SUCCESS	Options: Open Access: All
3383	?	IEXPLORE.EXE:2252	QUERY INFORMATION	C:\WINNT\system32\url.dll	SUCCESS	Attributes: A
3384	?	IEXPLORE.EXE:2252	SET INFORMATION	C:\WINNT\system32\url.dll	SUCCESS	FileBasicInformation
3385	?	IEXPLORE.EXE:2252	READ	C:\WINNT\system32\url.dll	SUCCESS	Offset: 0 Length: 12
3386	?	IEXPLORE.EXE:2252	QUERY INFORMATION	C:\WINNT\system32\url.dll	SUCCESS	Length: 106496
3387	?	IEXPLORE.EXE:2252	QUERY INFORMATION	C:\WINNT\system32\url.dll	SUCCESS	Length: 106496
3388	?	IEXPLORE.EXE:2252	CLOSE	C:\WINNT\system32\url.dll	SUCCESS	
3389	?	IEXPLORE.EXE:2252	QUERY INFORMATION	C:\Program Files\Internet E...	NOT FOUND	Attributes: Error
3390	?	IEXPLORE.EXE:2252	QUERY INFORMATION	C:\Documents and Settings...	NOT FOUND	Attributes: Error
3391	?	IEXPLORE.EXE:2252	QUERY INFORMATION	C:\WINNT\system32\OLE...	SUCCESS	Attributes: A
3392	?	IEXPLORE.EXE:2252	OPEN	C:\WINNT\system32\OLE...	SUCCESS	Options: Open Access: Exec...
3393	?	IEXPLORE.EXE:2252	CLOSE	C:\WINNT\system32\OLE...	SUCCESS	
3394	?	IEXPLORE.EXE:2252	QUERY INFORMATION	C:\Program Files\Internet E...	NOT FOUND	Attributes: Error
3395	?	IEXPLORE.EXE:2252	QUERY INFORMATION	C:\Documents and Settings...	NOT FOUND	Attributes: Error
3396	?	IEXPLORE.EXE:2252	QUERY INFORMATION	C:\WINNT\system32\OLE...	SUCCESS	Attributes: A
3397	?	IEXPLORE.EXE:2252	OPEN	C:\WINNT\system32\OLE...	SUCCESS	Options: Open Access: All
3398	2	IEXPLORE.DLL:2252	QUERY INFORMATION	C:\WINNT\system32\OLE...	SUCCESS	Length: 11264

그림 16. Filemon

크랙을 목적으로 할 경우는 사용기간이나 시리얼 넘버 등 소프트웨어 Protection에 관련된 정보의 보관장소를 밝혀내기 위해 사용되지만, 이외에도 윈도우 루트킷이나 악성 프로그램들의 행동을 검출할 때에도 사용되고 있습니다. 응용범위는 넓고 조작방법도 간단하기 때문에 능숙하게 다루는 데에는 문제가 없으리라 봅니다.

²⁴ Regmon: <http://www.sysinternals.com/Utilities/Regmon.html>

²⁵ Filemon: <http://www.sysinternals.com/Utilities/Filemon.html>

소프트웨어 크랙에 이용되는 툴

상용제품으로는 MultiMon²⁶이라는 제품도 사용해 볼 만 합니다. 시스템 파일, 디바이스, 레지스트리, 네트워크, 키보드, 클립보드 모니터링 할 수 있을 뿐만 아니라 특정 프로세스별로 개별 로깅이 가능합니다.

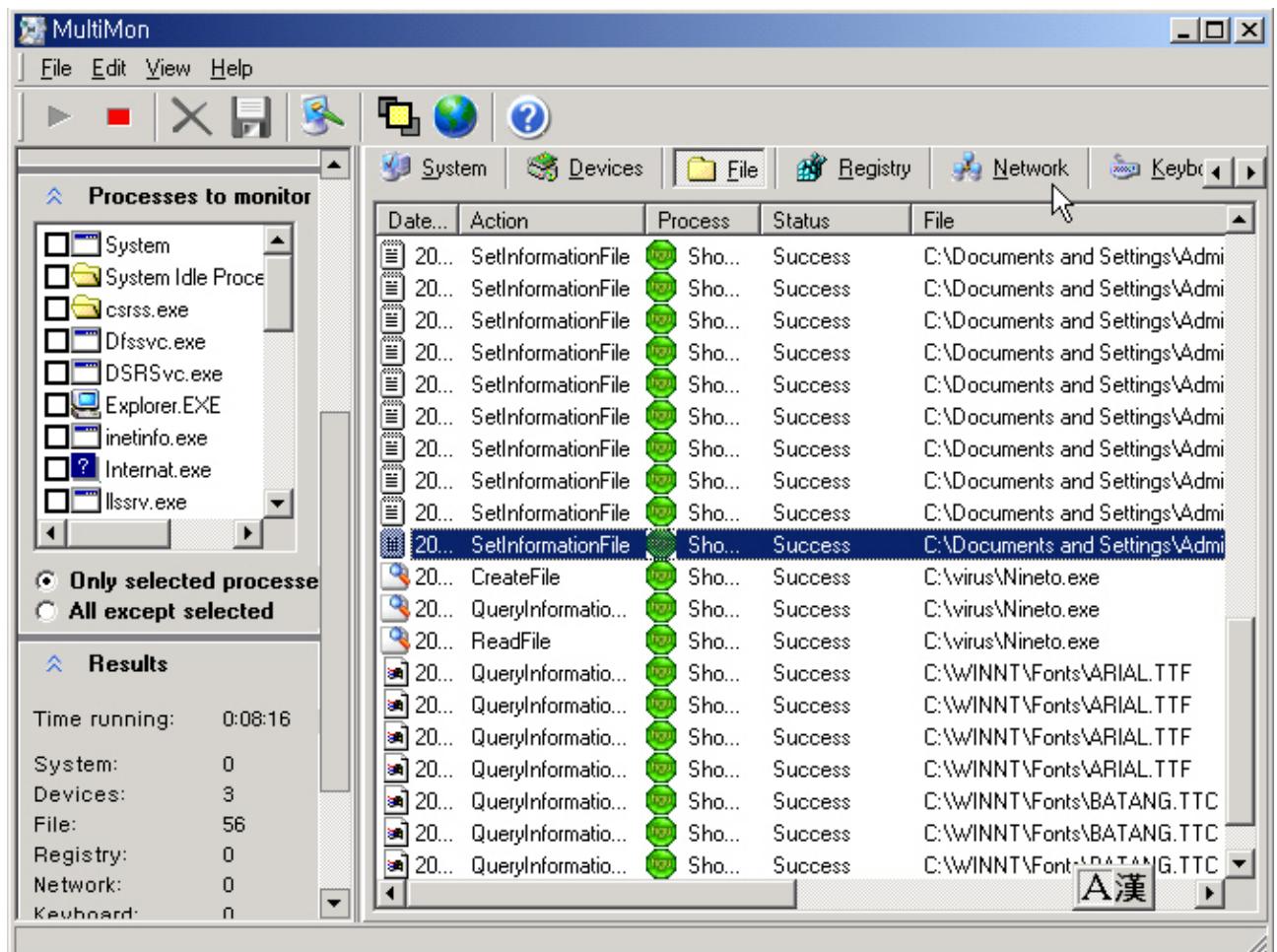


그림. MultiMon

²⁶ MultiMon: <http://www.resplendence.com/multimon>

파일 스캐너 (File Scanners)

크랙을 행할 때에는 먼저 타겟이 되는 실행가능 파일이 어떤 컴파일러로 작성되었는지, 타겟이 팩(Packs)되어 있는 경우 어떤 패커(Packers)가 이용되었는지를 조사하는 것부터 시작됩니다. 사용된 컴파일러(Compilers)에 따라 크랙 기법도 다르기 때문에 패커에 있어서도 언패커(Un-Packers)가 존재하는지 확인 가능하다면 시간을 단축 시킬 수 있습니다. 그러나 컴파일러와 패커의 종류는 버전 차이를 포함하면 엄청난 수에 이르기 때문에 파일 스캐너라고 불리는 툴을 이용하여 확인합니다.

【PE iDentifier】²⁷은 Compiler, Packer, Cryptors를 버전 별로 포함하여 600 종류 이상(계속 업데이트 중)을 판별할 수 있는 매우 우수한 파일 스캐너입니다. 본래의 기능 이외에 파일 헤더 정보 분석, Hex Viewer, 역 어셈블 기능도 갖추고 있고, 플러그 인을 통해 언팩을 수행하는데 유용한 OEP(Original Entry Point)의 확인도 가능합니다.



그림. PeiD

【Stud_PE】²⁸는 CGSoftLabs에서 제공되는 툴로 PeiD와 함께 대표적인 파일 스캐너입니다. PeiD의 Plugin을 동일하게 사용할 수 있으며 파일 헤더분석, Dos 헤더 분석, Sections분석, Functions 분석 및 리소스 편집도 가능합니다. 또한 바이너리 파일 비교를 통해 파일 변경 부분을 확인 할 수 있습니다.

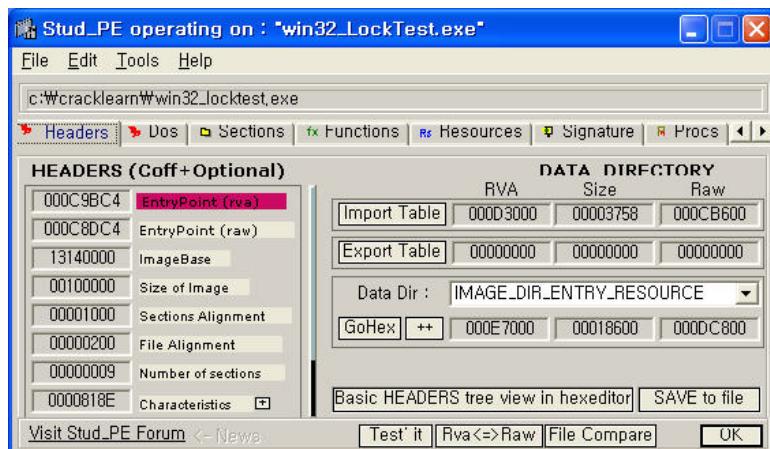


그림. Stud_PE

²⁷ PE iDentifier: <http://peid.has.it/>

²⁸ Stud_PE: <http://www.cgsoftlab.ro/studpe.html>

언 패커 (Un-Packers)

크랙 대상에 따라서는 패커라고 불리는 툴로 실행 파일이 압축, 암호화되어 있는 수가 많습니다. 이런 경우 역어셈블러나 디버거가 정상으로 동작하지 않기 때문에, 언패커라고 불리는 툴로 패커로 보호되어 있는 상태를 해제하고 원래 바이너리 데이터로 복원하는 작업을 수행 합니다. 잘 알려진 패커라면 간단한 클릭 만으로 언팩할 수 있는 언팩 툴이 존재하지만, 알려지지 않은 압축방법이 이용되고 있는 경우는 수작업에 의한 언팩을 수행하여야 합니다. 수작업에 의한 언팩 방법의 경우 【툴 언팩】과 구별하여 부릅니다.

파일 패킹 종류는 앞에서도 설명했듯이 600 종류 이상으로 사실상 모든 패킹을 언팩할 수 있는 단일 툴은 존재하지 않습니다. 잘 알려진 언팩커는 【UPXpack, Antiaspack, ASPack, ACProtect, ARMMadilo, ASProtect, DEPE, EXEStealth, EXEPack, EXEZip, FSG, Generic, MEW, Morphine, PeCompact, Pedimisherunpack, PEncrypte, PESpin, tELockt, unpepack, unaspck, UnMEW, unFSG, qunpack, autounpack, y0dadepack 너무 많아 생략 …】 등이 존재합니다. 파일 스캐너로 먼저 팩킹된 정보를 확인 한 후 언팩 툴이 존재하는지 검색해 보고 없다면 매뉴얼 언팩을 시도하는 것이 좋습니다.

OllyDbg 플러그인으로 공개되어 있는 덤프 툴 【OllyDump】²⁹은 메모리상에 로드된 코드를 파일로 저장할 수 있습니다. SoftICE에 대해서도 【icedump】라는 플러그인이 공개되어 있는데, 이것은 원래 메모리 덤프 기능뿐만 아니라 Capture가 불가능한 스크린샷 기능이나 SoftICE의 검출회피(Anti-Debug meltice대책)등의 부가기능을 제공합니다.

매뉴얼 언팩의 경우 부가적으로 많은 부분을 염두에 두고 작업해야 합니다. 덤프된 파일을 윈도우 상에서 실행시키기 위한 작업과 Anti-Debug 체크 루틴을 해제하여야 하는 등 시간이 많이 소요됩니다.

²⁹ OllyDump: <http://dd.x-eye.net/>

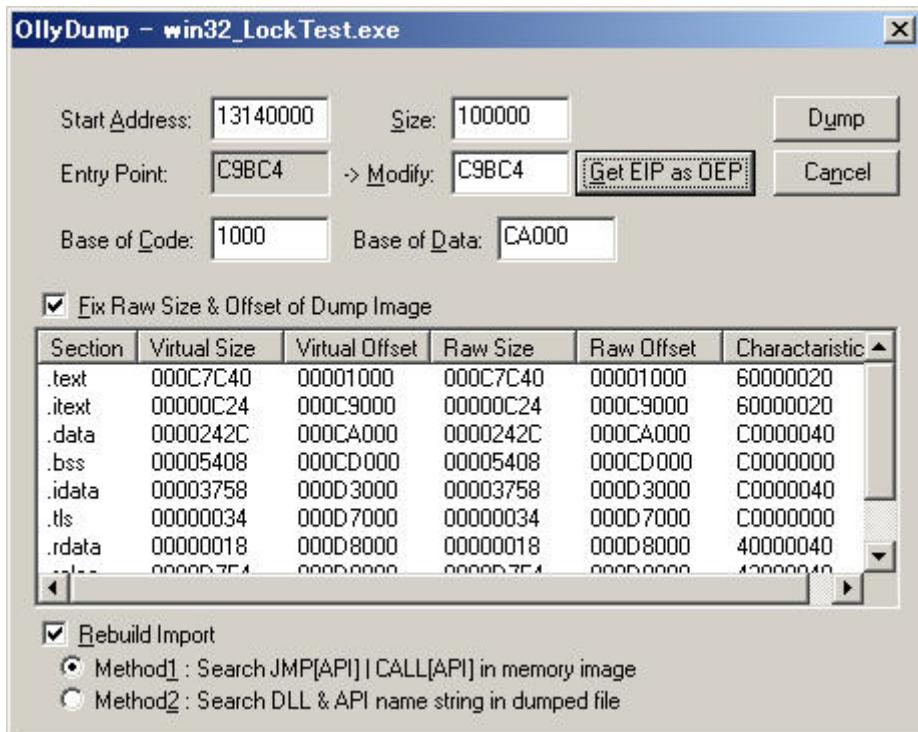


그림. OllyDump

리빌더 (Rebuilders)

덤프 툴에서 메모리 위에 저장된 본래의 코드를 추출해도 실행 파일로 정의되어 있는 Import 함수의 정보(IAT:Import Address Table)는 복원되지 않습니다. 그래서 Import 정보를 복원하는 IAT 리빌드 툴을 이용하여 Import 정보를 재 구축을 할 필요가 있습니다. 이렇게 윈도우에서 실행 가능한 파일 구조로 만들어 주는 툴을 리빌더라고 합니다.

【Revirgin】³⁰는 크랙 포럼에서 활동하던 +Tsehp 에 의해 작성된 리빌더(Rebuilder) 툴입니다. 최종 버전은 1.5로 WinNT에서만 동작하기 때문에 Win9x계열에서는 1.3을 이용할 필요가 있습니다. 현재 더 이상 업데이트가 이루어지지 않고 있습니다. 동종 툴로 약간 인터페이스가 뛰어난 【Import REConstructor】³¹, 【LordPE】³²가 있습니다. 보통 리빌드만을 전문적으로 하는 툴은 특별히 존재하지 않고 PE파일 편집 프로그램에서 플러그인 형태나 서브 기능으로 제공하는 경우가 많습니다. 또한 공개된 리빌더 소스를 가지고 리버서들이 각자 입맛에 맞게 변형하는 형태라서 동작원리가 비슷하다고 보면 됩니다. 리빌드는 차후 크랙과정과 윈도우 PE파일 포맷 부분에서 상세하게 소개할 예정입니다.

³⁰ Revirgin: <http://programmerstools.org/node/354>

³¹ Import Reconstructor: <http://mackt.cjb.net/>

³² LordPE: <http://y0da.cjb.net/>

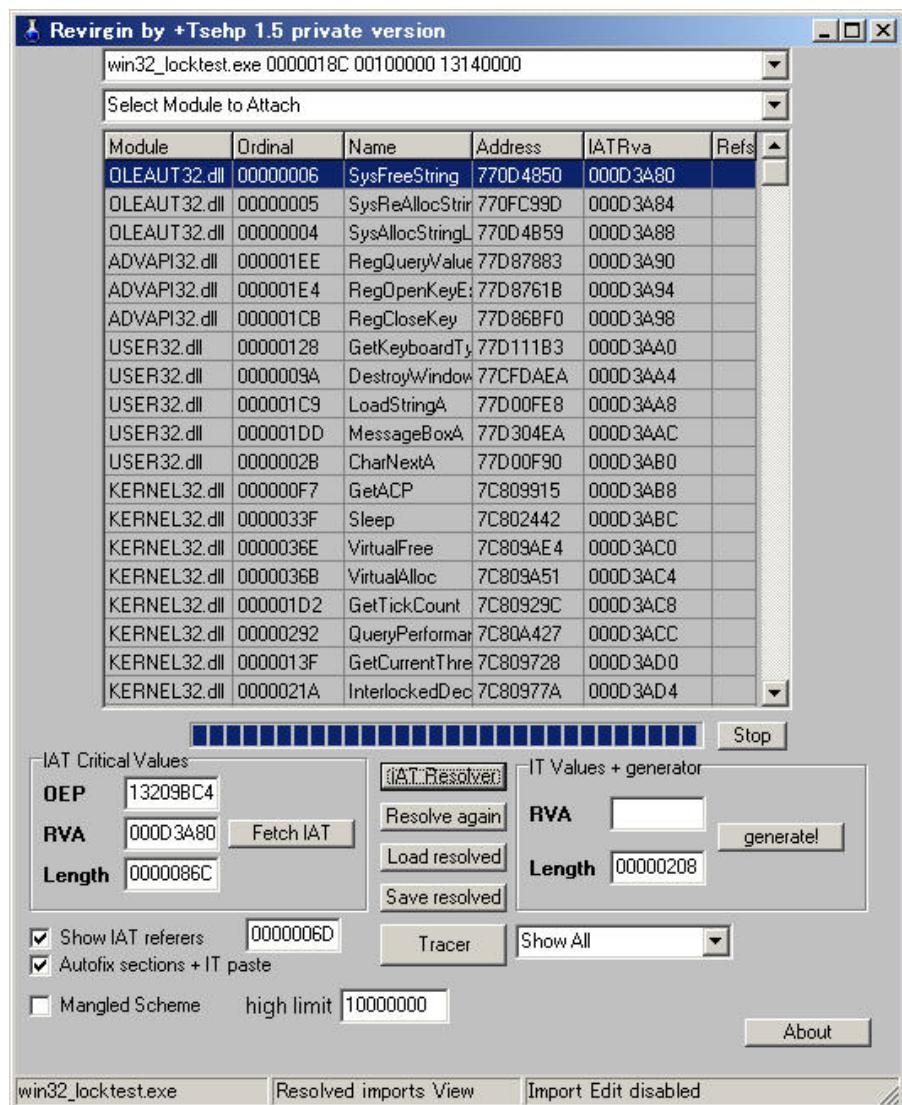


그림. Revirgin

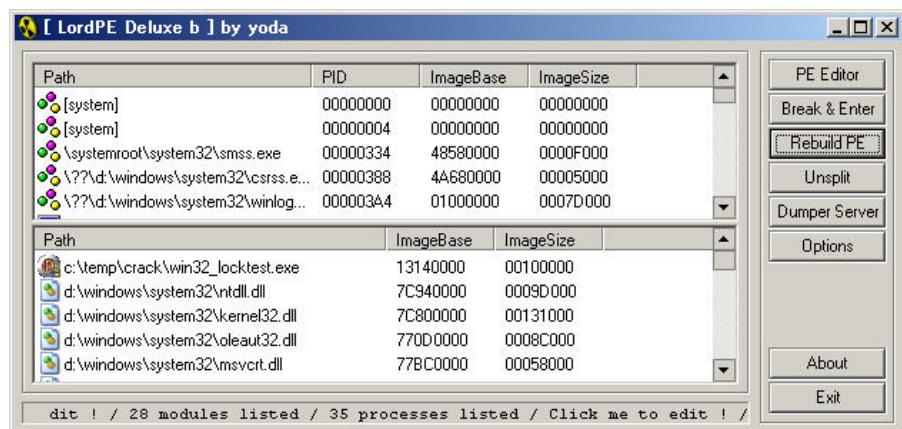


그림. LordPE

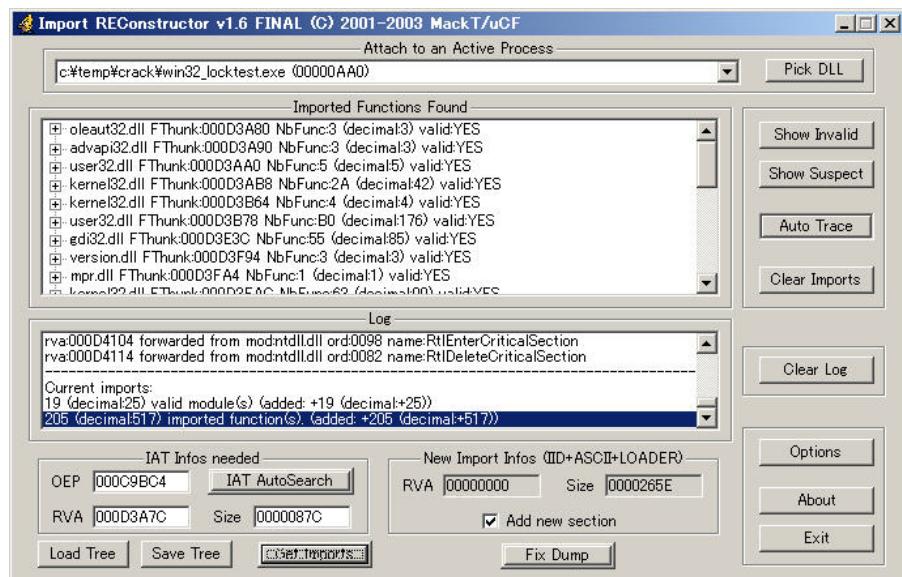


그림. Import REConstructor

패처 (Patchers)

크래커가 파일의 바이너리 수정을 통해 크랙에 성공한 경우 오리지널 버전과 크랙 버전과의 달라진 정보, 즉 크랙 패치 부분이 존재합니다. 외산 소프트웨어를 한글화하는 리소스 편집 등에도 패치파일이 이용되지만, 패치는 유저의 편리성을 생각하여 실행 파일 형태(즉 바이너리 형식)로 배포됩니다. 그럴 경우 패치 파일 작성에는 패처(Patcher)라는 툴을 이용합니다. 주로 이용되는 패처 툴로는 DUP2(diablo2oo2's Universal Patcher)³³, Universal Patcher³⁴, aPE³⁵등이 이용됩니다.

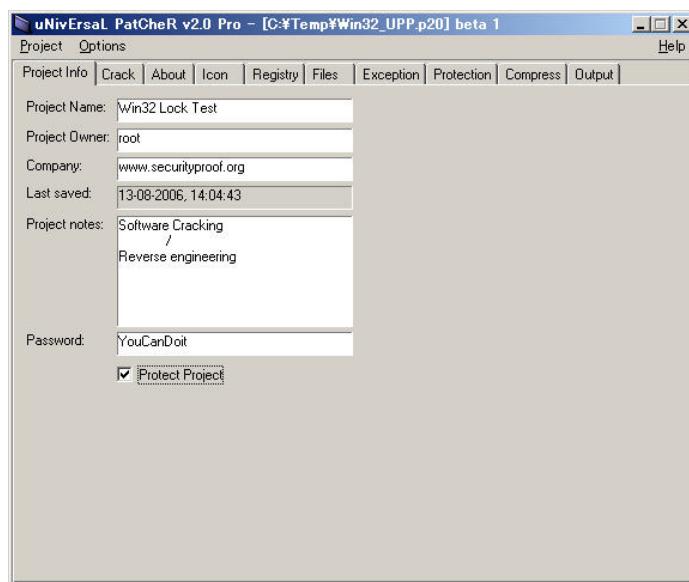


그림. Universal Patcher

³³ dUP2: <http://navig8.to/diablo2oo2>

³⁴ uNivErsal Patcher: <http://www.aaochg.prv.pl/>

³⁵ aPE: <http://ap0x.jezgra.net/patchers.html>

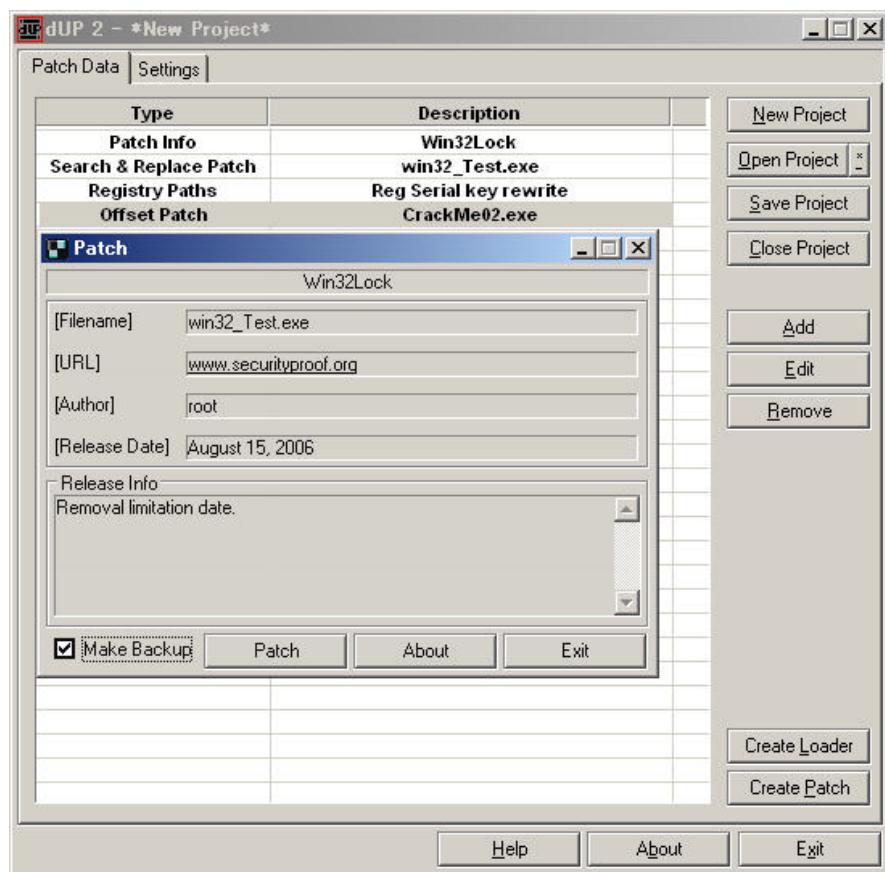


그림. dUP2

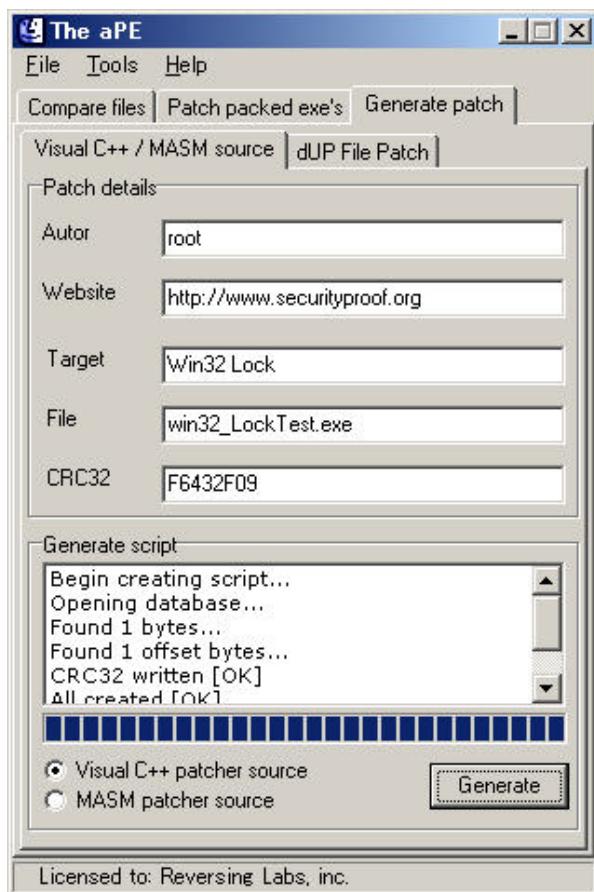


그림. aPE

그러나 크랙 패치를 실행 파일로 만들어 주는 경우는 드문 일입니다. 그 이유로는 크랙 정보교환 수단이 주로 BBS 나 IRC 가 사용되고 있기 때문에 텍스트 형식이나 어셈블 파일 형식이 편리하다는 것입니다. 또한 크랙 패치는 다소 불법적인 요소를 포함하고 있기도 하며 텍스트 파일 형태로 배포시 바이러스 및 악성 루틴 추가 위험성이 없어진다는 것입니다. 그래서 텍스트 파일 형태를 불러와 패치하는 전문적인 툴들이 존재합니다. 역 어셈블러나 바이너리 편집기, 패치 작성 프로그램 등에서 크랙 되어야 하는 부분들을 텍스트 형태로 저장한 후 패처 툴에서 파일을 불러와 크랙 패치를 수행합니다.

안티 프로텍터(Anti-Protectors)

크랙이나 리버싱 엔지니어링을 시도하지 못하도록 많은 소프트웨어는 적절한 보호 장치를 마련하고 있습니다. 그러므로 이러한 보호장치를 건너 뛸 수 있는 안티 프로텍터들이 필요하게 됩니다. 안티 프로텍터를 구분하면 【Anti-Debugging, Anti-Traceing, Anti-Dumping, Anti-Generator OEP, Anti-Inline, Anti-Invisible, Anti-Hide, Anti-Hooking, Anti-Hardware, API Import redirection, CRC checksum, Mutex checking, File name changeing protection, OEP protection code, Code markers inline protection, Clear PE header data, Fake header data, HardLock section locking】로 나눌 수 있으며, 각각의 보호 기법에 따라 크랙 하는 방법과 사용하는 툴이 달라 질 수 있습니다. 그러나 이러한 안티 프로텍터를 회피한다고 해도 내부 프로그램 알고리즘을 분석하지 못한다면 원하는 크랙 키 값이나 리버싱에 필요한 로직을 얻지 못할 수도 있습니다.

파일 편집 툴 (File Snipping/Injecting/Code Ripper/Data Ripper)

크랙 진행 중에 파일의 특정 부분을 잘라내거나 실행 순서를 다시 맞추기 위해 파일을 조각 낸다거나 파일 일부에 코드를 삽입하는 경우가 많습니다.

###이걸 여기서 다 설명해야 되나 말아야 되나, 나중에 크랙 과정 중에 잠깐식 소개 하는게 날을려나, 고민되네.

유용한 툴 (Useful Tools)

크랙을 수행할 때에 있으면 유용한 툴들이 많이 존재합니다. 그 중에 몇몇 툴들을 소개하면 아래와 같습니다.

a). CFF Explorer

Deniel Pisteli가 프리웨어로 제공중인 CFF Explorer³⁶은 다양한 기능들을 제공하고 있습니다.

³⁶ CFF Explorer: <http://www.ntcore.com/>

소프트웨어 크랙에 이용되는 툴

X86/x64/Itanium기반의 PE 파일 및 메모리 덤프 기능, 리빌더, 리소스 편집기, 헥사 에디터, Import 추가 기능등을 제공합니다. 특히 .NET 리소스를 편집하는데 매우 유용합니다.

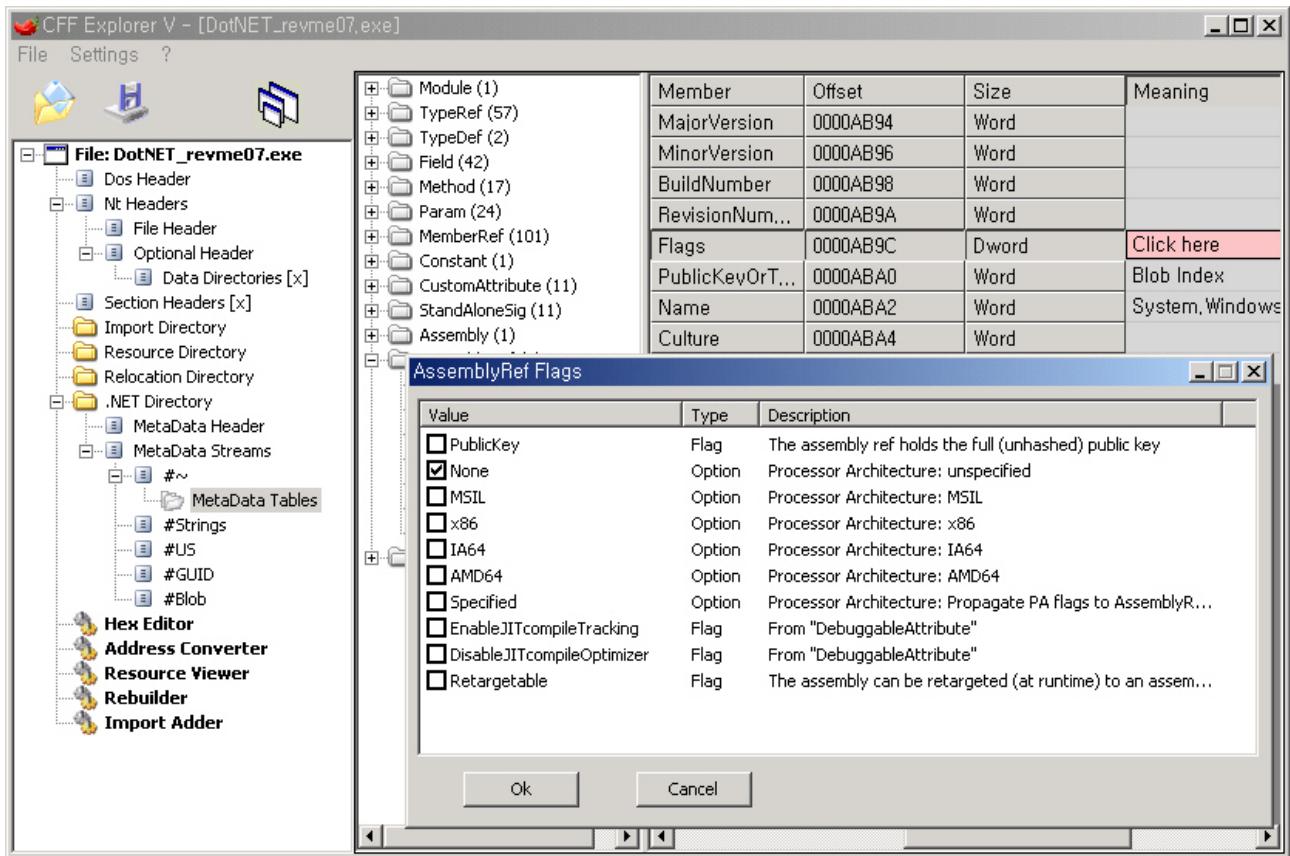


그림. CFF Explorer

정리 다시 해서 문서추가 해야 함.

기타 툴 (Other Tools)

크랙을 도와주는 많은 툴이나 크랙을 수행하려는 프로그램 중에는 하드 디스크 데이터를 삭제하거나 시스템 파일을 손상시키는 악성 로직이 존재하는 것도 있습니다. 바이러스를 리버스 엔지니어링 시도할 때 혹은 크랙툴을 다운로드 받아 실행할 때는 항상 이러한 위험에 노출되어 있습니다. 이런 경우 【VMWare】³⁷나 【Virtual PC】³⁸ 같은 가상 윈도우PC 가 효과를 발휘합니다. 가상 윈도우PC 상에서 크랙/리버스 엔지니어링 환경을 구축하면 분석시 악성 로직이 실행 되어도 분석 이전 상태로 언제든지 복원할 수 있습니다. 이런 경우 시스템이 Crash 되어도 피해를 최소화 할 수 있습니다. VMWare/VirtualPC의 경우 운영체제를 이미지 파일로 저장하고 있으므로 손쉽게 서로 다른 운영체제를 하나의 시스템에서 동시에 부팅 할 수 있습니다.

³⁷ VMWare: <http://www.vmware.com/>

³⁸ Virtual PC: <http://www.microsoft.com/windows/virtualpc/default.mspx>

소프트웨어 크랙에 이용되는 툴



그림. Microsoft Virtual PC

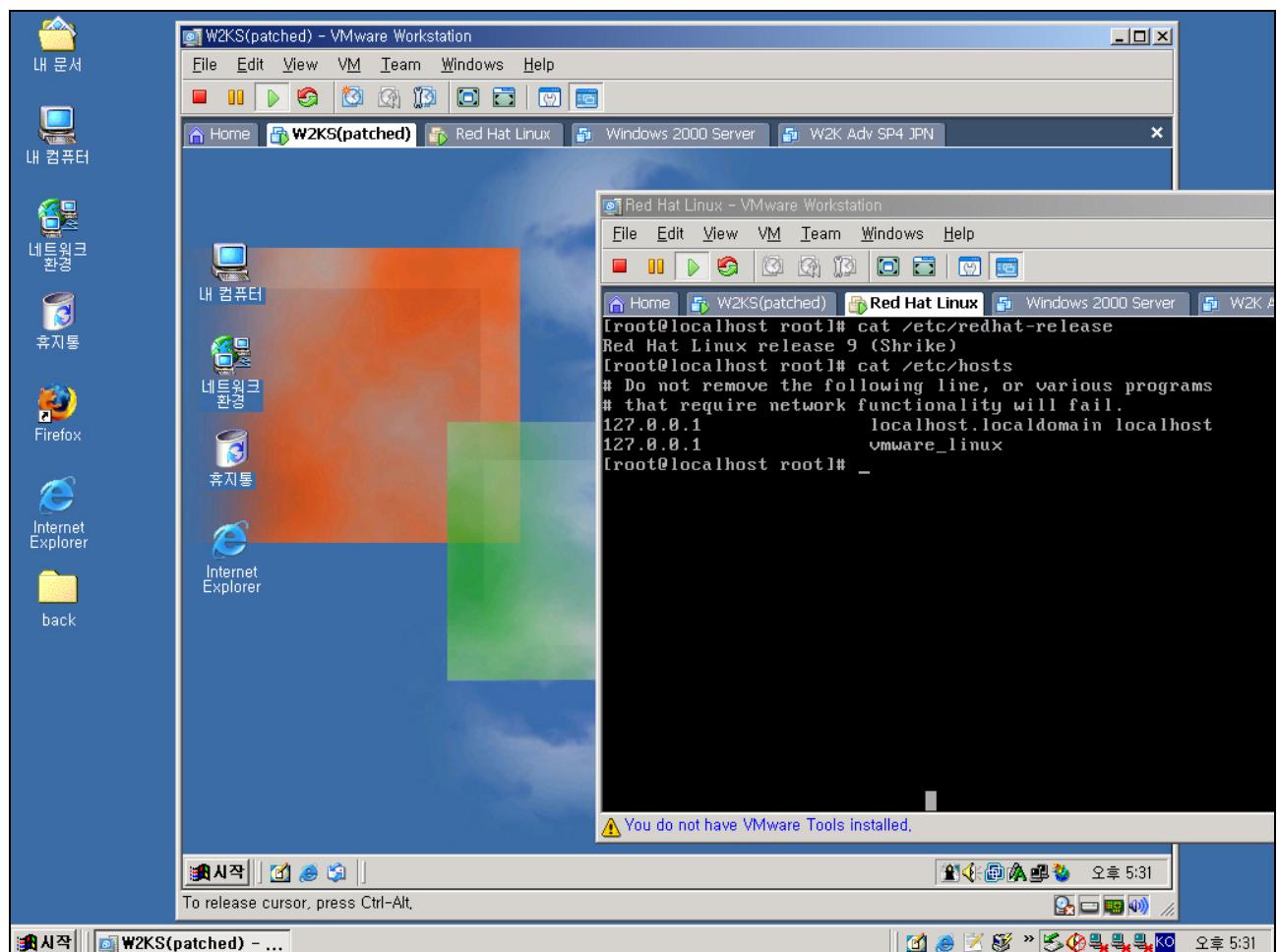


그림. VMWare

그리고 ...

적어도 한 개 이상의 프로그래밍 언어를 습득하면 좋습니다. 창조적인 리버서(해커, 크래커)로 성장하려면 반드시 필요한 자질이며, 자신만의 리버싱 기술을 표현하려면 없어서는 안될 기술이기 때문입니다. 리버스 엔지니어링을 잘하는 사람치고 프로그래밍을 못하는 사람은 아직 만나보지 못했습니다.

또한 리버스 엔지니어링을 성공적으로 하기 위해서는 리버싱 하려는 대상 프로그램에 대한 충분한 지식을 갖추면 좋습니다. 인터넷 전화를 리버싱 하려는 사람이 H.323, MGCP 와 같은 네트워크 프로토콜에 대한 이해가 없다면 얻을 수 있는 정보는 매우 작습니다. 마찬가지로 Windows 루트킷을 리버싱 하려는 사람이 Windows API 에 대한 지식이 없다면 분석은 상당히 어려울 것입니다. 자신이 리버싱 하려는 대상에 대한 충분한 사전 지식을 먼저 습득한다면 성공확률은 높아 질 것입니다.

마지막으로 많은 인내심이 필요합니다. 타고난 리버서가 아니라면 소프트웨어의 흐름과 제어 구조를 알아가기 위해 단순하고도 기나긴 시간이 필요할 것입니다. 디버깅이나 프로그램 분석을 어렵도록 만든 소프트웨어를 만난다면 원 제작자와의 길고 긴 두뇌싸움이 시작될 것입니다. 이러한 두뇌싸움에서 일찍 승리하려면 최신의 Anti 기법을 항상 습득하고 리버싱과 크랙을 많이 하여 다양한 경험을 쌓는 방법 밖에는 없습니다.

제 3 장 Windows 크랙 기본편

3.2 크랙에 필요한 어셈블리어 기초(Assembler Basic)

어셈블리 언어의 기초

소프트웨어 크랙에는 어셈블리 언어에 대한 지식이 반드시 요구 됩니다. 운영체제에 따라 CPU 모델에 따라 조금씩 차이가 있는 어셈블리 언어를 알아야 합니다. 그러나 가장 기본적인 부분임에도 관련된 정보가 적고, 크랙을 시작하고 싶은 사람에게 있어서도 첫 번째 장벽이 되고 있는 것이 현실입니다.

이번 장에서는 Windows 크랙에 필요한 어셈블리 언어를 설명 하고자 합니다. 어셈블리 언어 자체는 매우 간결하고 단순하므로, 기본적인 부분이라도 알고 시작하면, 크랙 수행 중에 스스로 레퍼런스를 참고하여 이해해 나갈 수 있습니다. 크랙은 경험이 많은 부분을 차지하고 있습니다. 단순한 어셈블리어의 설명보다 실제 크랙을 수행하면서 코드 상에서 전개되는 흐름을 하나씩 이해하다 보면 자연스럽게 어셈블리언어가 습득되게 됩니다.

어셈블리 언어란? (Assembly language?)

확장자가 EXE 인 실행 파일을 바이너리 에디터로 열면 숫자들의 나열이 보입니다. 이것은 헤더, 데이터, 그리고 프로그램 처리 로직이 전부 숫자로 표시되어 있기 때문입니다. EXE 가 실행되면 OS 는 메모리 위에 프로그램을 적절히 옮겨 놓고 여러 가지 처리를 먼저 한 후 프로그램에게 제어권을 넘기고 숫자의 나열로 표현된 CPU 에 대한 명령이 실행되게 됩니다. 숫자의 나열은 기계어라고 불리는 것으로 CPU 가 이해할 수 있는 유일한 언어입니다.

어셈블리 언어는 기계어에 매우 가까운 문법으로 숫자에 이름을 붙이거나 하여 인간이 이해하기 쉽도록 한 언어입니다. 기본적으로 기계어와 어셈블리 언어는 1 대 1로 대응하고 있습니다.

프로그램 해석은 기계어를 보면서 진행하는 것도 가능하지만, 숫자의 나열이므로 인간에게 있어서는 매우 이해하기 힘듭니다. 물론 디버깅이나 리버싱을 많이 해본 사람은 자주 쓰는 기계어 코드를 외우는 경우도 있습니다. 그러나 디버거나 디스어셈블러에서 어셈블리 언어 레벨로 변환하여 해석하는 것이 실제적인 방법입니다. 리버스 엔지니어링 능력을 더욱더 키우고자 한다면, 본인이 작성하는 프로그램을 디버깅 할 때 원본코드와 어셈블리어 코드를 항상 같이 비교하면서 보면 도움이 될 것입니다. 소스 코드가 실제로 어떻게 어셈블리 명령으로 바뀌는지 확인하면 나중에는 어셈블리 명령만을 통해서도 프로그램 흐름을 금방 이해할 수 있게 됩니다.

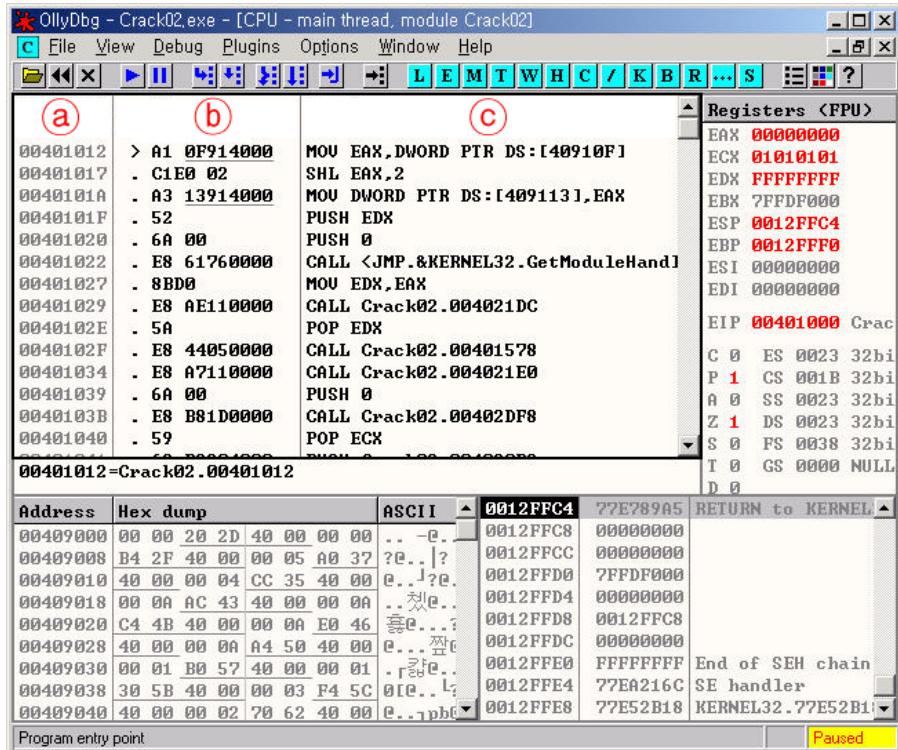


그림 1. OllyDbg

그림 1 은 OllyDbg 라는 디버거로 Crack02.exe 실행 파일을 불러 들인 화면입니다. 왼쪽 열 “ⓐ”영역에 있는 【00401012】 등의 숫자는 프로그램 메모리 주소를 나타내며, RVA(Relative Virtual Address)와 파일 포인터 간의 적절한 변환을 거치면 실행파일상의 오프셋과 매치됩니다. 그 중앙 “ⓑ”영역에 있는 【A1 0F914000】 와 같은 숫자가 기계어입니다. 맨 우측 “ⓒ”영역에 있는 【MOV EAX,DWORD PTR DS:[40910F】 와 같은 기호가 어셈블리 언어라고 불리는 것입니다.

2 진수와 16 진수(Binary and Hexadecimal)

일상 속에서 우리들이 사용하는 숫자는 10 진수라는 것입니다. 10마다 자릿수가 올라가기 때문에 10 진수라고 합니다. 그러나 컴퓨터 세계는 디지털, 즉 0과 1만의 세계입니다. 컴퓨터는 내부적으로 0과 1만으로 모든 처리를 하고 있는 것입니다. 때문에 숫자를 표현하는 데도 0과 1만으로 나타낼 필요가 있습니다. 그래서 2진수라는 것이 등장합니다. 2진수는 2마다 자릿수가 올라가는 것으로, 각 자릿수는 0과 1밖에 없습니다. 2진수는 “0, 1, 10, 11, 100, 101, 111, 1011 …”과 같이 나아갑니다. 몇 자릿수 정도라면 2진수라도 어떻게든 읽을 수 있지만 숫자가 조금 올라가도 몇십자릿수가 되는데(예를 들어 10 진수 100은 2진수로 변환하면 1100100), 그렇게 되면 우리들은 이해하는데 매우 어렵게 되며 표기하는 데도 효율적으로 문제가 됩니다.

그래서 2 진수와 잘 맞고 인간에게도 비교적 이해하기 쉬운 16 진수라는 것을 사용합니다. 16 진수는 16마다 자릿수가 올라가는 표현 방법으로, 0~9의 숫자와 A~F의 문자를 이용합니다.

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11,12,13,14,15,16,17,18,19,1A,1B,1C,1D,1F,20….

16 진수 1 자릿수로 2 진수 4 자릿수를 표현할 수 있습니다. 16 진수 F 가 2 진수의 1111 이 되며, 이런 점에서 10 진수보다 16 진수 쪽이 2 진수와 잘 맞는다고 말할 수 있습니다.

10 진수	2 진수	16 진수
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

표 1. 숫자의 진수 표현

여러 종류의 수치 표현이 나오지만 혼란을 피하기 위해서는 16 진수를 나타낼 때는 숫자 첫머리에 0(Zero)을 붙이고, 숫자 어미에는 h³⁹을 붙여서 표현 합니다. 예를 들면 【0FAh, 016h】 와 같이 합니다. 또 2 진수를 나타낼 때는 숫자 어미에 b⁴⁰를 붙여서 표현하도록 합니다. 예를 들어 【100b】 와 같이 합니다. 10 진수에는 아무것도 붙이지 않습니다.

10 진수	10	16	17	31	32	255	256
16 진수	0Ah	010h	011h	01Fh	020h	0FFh	0100h
2 진수	1010b	1 0000b	1 0001b	1 1111b	10 0000b	1111 1111b	1 0000 0000b

표 2. 다양한 진수 표현

³⁹ h 는 영어로 16 진수를 나타내는 hexadecimal 의 약자입니다.

⁴⁰ b 는 영어로 2 진수를 나타내는 binary 의 약자입니다.

크랙에 필요한 어셈블리 기초

2 진수와 10 진수, 16 진수 사이의 상호 변환은 Windows 보조프로그램 계산기를 사용하면 됩니다. 윈도우 시작메뉴의 【프로그램】》【보조 프로그램】》【계산기】를 실행합니다. 계산기 메뉴의 【보기】》【공학용】에 체크합니다. 그러면 【Hex-16 진수】 , 【Dec-10 진수】 , 【Oct-8 진수】 , 【Bin-2 진수】의 라디오 버튼이 달린 계산기로 변경됩니다.

예를 들어, 10 진수에서 16 진수로 변환할 경우는 【Dec】의 라디오 버튼이 온 되어 있는 상태에서 숫자를 치고 16 진수를 가리키는 【Hex】 라디오 버튼을 클릭 하면 됩니다. 마찬가지로 2 진수로 변환 하려면 【Bin】 버튼을 활성화 해주면 됩니다.

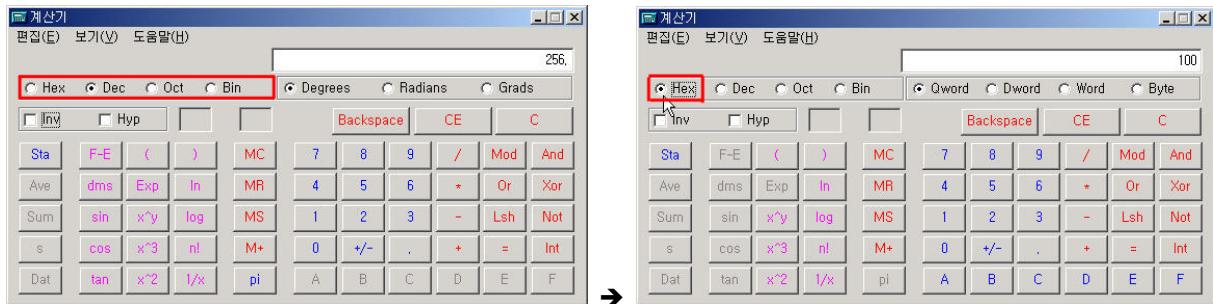


그림 2. 진법 계산기

비트와 바이트(Bit and Byte)

어셈블리 언어를 사용한 후에 나오는 단위에 대하여 설명합니다.

- 비트(bit): 컴퓨터에서 사용되는 최소 단위입니다. 2 진수에서 1 자릿수, 즉 0이나 1입니다.
- 니블(nibble): 2 진수에서 4 자릿수, 즉 4 비트가 1 니블이 됩니다. 16 진수면 1 자릿수가 됩니다.
- 바이트(Byte): 2 진수에서 8 자릿수, 즉 8 비트가 1 바이트입니다. 16 진수면 2 자릿수가 됩니다.
- 워드(Word): 2 진수에서 16 자릿수, 즉 16 비트가 1 워드입니다. 16 진수면 4 자릿수가 됩니다.
- 더블 워드(double word, Dword): 2 진수에서 32 자릿수, 즉 32 비트가 1 더블 워드입니다.

16 진수면 8 자릿수가 됩니다

- 쿼드 워드(Quad word, Qword): 2 진수에서 64 자릿수, 즉 64 비트가 1 쿼드 워드입니다.

16 진수면 16 자릿수가 됩니다.

비트	바이트	워드	더블 워드	쿼드 워드	나타내는 범위
1bit					00–01h
8bits	1Byte				00–0FFh
16bits	2Bytes	1Word			00–0FFFFh
32bits	4Bytes	2Words	1Dword		00–0xFFFFFFFFh
64bits	8Bytes	4Words	2Dwords	1Qword	00–0xFFFFFFFFFFFFFFh

표 3. 각 단위의 관계

양수와 음수(Positive number and Negative number)

프로그램 중에서는 음수를 사용하는 경우도 있습니다. 여기서는 하나의 단위를 반씩 분할하여 수를 양과 음으로 나타내는 방법을 설명하겠습니다.

8 비트 숫자를 예를 들어 생각해 보겠습니다. 8 비트는 00000000b~11111111b, 000h~0FFh, 즉 0~255 까지 256개 숫자를 나타낼 수 있습니다. 이것을 반으로 분할하면 -128~127의 숫자로 나타낼 수 있습니다. -1은 0보다 1 작은 수입니다. 그래서 0에서 1을 빼면 Overflow가 되어 버리기 때문에, 이것을 OFFh라고 합니다. 그리고 작아질수록 0FEh, 0FDh …로 줄어 듭니다. 그러면 아래와 같은 표가 만들어집니다.

-128	-127	…	-3	-2	-1	0	1	2	3	…	126	127
080h	081h	…	0FDh	0FEh	0FFh	00h	01h	02h	03h	…	07Eh	07Fh

표 4. 양수와 음수 표현

여기에서 080h~0FFh는 2 진수로 나타내면 최상위 비트가 전부 1이 되고 00h~07Fh는 최상위 비트가 전부 0이 됩니다. 이 최상위 비트를 양(+)과 음(-)의 부호를 나타내는 비트라 생각할 수 있습니다.

OFFh가 -1을 나타내는지 255를 나타내는지는 프로그램상의 흐름에 의하여 달라집니다. 지금은 이런 방법이 있다는 것만 알아 두십시오.

레지스터와 플래그(Register and Flag)

리버스 엔지니어링을 하기 위해서는 CPU 내부에 존재하는 특별한 메모리들을 잘 알고 있어야 합니다. CPU에는 계산 등에 사용되는 정보를 일시적으로 보존하여 두는 장소가 있습니다. 이것을 레지스터라고 부릅니다. 레지스터는 정해져 있고 용도도 어느 정도 한정되어 있습니다. x86 계열 CPU에서는 8비트, 16비트, 32비트 단위로 레지스터를 사용할 수 있으며 각각 이름이 붙어 있습니다. 또한 레지스터는 범용 레지스터, 플래그 레지스터 등 몇 개의 종류가 있습니다.

① 범용 레지스터

범용 레지스터에는 【EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI】⁴¹ 등이 있습니다. 이것은 전부 32비트 레지스터입니다. 또한 【EAX, ECX, EDX, EBX】는 32비트 중 16비트만을 취급할 때는 【AX】로 지정하며 또한 【AX】 중 상위 8비트를 취급할 때에는 【AH】, 하위 8비트를 취급할 때는 【AL】로 지정 합니다.

⁴¹ ExtendedAccumulatorX, ExtendedBaseX, ExtendedCounterX, ExtendedDataX

크래ք에 필요한 어셈블리어 기초

【ECX, EDX, EBX】에 대하여도 【EAX】와 같이 지정할 수 있습니다.

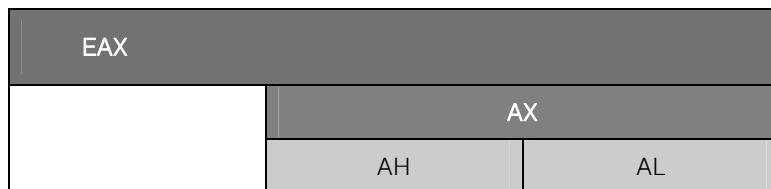


표 5. 32 비트 범용 레지스터에의 액세스

【ESP】(스택 포인터), 【EBP】(베이스 포인터), 【ESI, EDI】는 각각 32 비트 레지스터로, 그 하위 16 비트를 지정하는데 【SP, BP, SI, DI】라는 이름으로 사용할 수 있습니다. 8 비트씩 지정은 안됩니다.

레지스터는 어셈블리 언어 명령과 밀접한 관계가 있으므로 어느 정도 용도가 한정되어 있습니다. 상세한 내용은 명령 해설 부분에서 설명하겠습니다.

레지스터	설명
EAX(Accumulator)	누산기로 불리며, 메모리 산술 연산 결과가 저장되는 레지스터
EBX(Base)	주소 지정을 확장하기 위해 인덱스(Index)로 사용되는 레지스터
EDX(Data)	입출력 연산에서 사용되는 데이터 레지스터
ECX(Counter)	카운터로 루프가 반복되는 횟수를 제어하는데 사용되는 레지스터
ESI(Source Index)	보내는 측의 포인터 지시자로 메모리 고속 복사에 이용
EDI(Destination Index)	받는 측의 포인터 지시자로 메모리 고속 복사에 이용
EBP(Base Pointer)	함수 인자와 지역 변수를 가리킬 때 사용되는 레지스터
ESP(Stack Pointer)	스택의 맨 윗 부분을 가리킬 때 사용되는 레지스터

② 플래그 레지스터

플래그 레지스터는 CPU 상태를 나타내기 위한 레지스터로 프로그램에서 직접 조작하는 경우는 별로 없습니다. 예를 들어 계산 결과가 Overflow(계산 할 수 있는 자릿수를 넘은)되면 OF(Overflow 플래그)가 세트 됩니다. 각 플래그 레지스터는 1 비트로 표시하고 있으며, 전부 0이나 1 상태를 나타내고 있습니다. 플래그가 클리어 되는 때 플래그는 0이 되고, 세트 될 때는 플래그는 1이 되는 것을 표현하고 있습니다.

대표적인 플래그에 CF, PF, ZF, OF, DF 등이 있습니다.

CF	Carry Flag	“캐리 플래그”라고 불리며, 산술 연산에서 결과의 최상위 비트가 Carry 또는 borrow 한 경우(자릿수 올림이나 자릿수 빌림)에 세트(1)되고, 안된 경우에는 클리어(0) 됩니다.
PF	Parity Flag	“패리티 플래그”라고 불리며, 산술 연산에서 결과의 최하위 바이트에 1의 비트가 짝수개의 경우에 세트되고, 홀수개의 경우에 클리어 됩니다.
ZF	Zero Flag	“제로 플래그”라고 불리며, 산술연산의 결과가 제로일 경우에 세트 되고,

크랙에 필요한 어셈블리어 기초

		제로가 아닐 경우에 클리어 됩니다.
OF	Overflow Flag	“오버 플로우 플래그”라고 불리며, 연산에 오버 플로우가 일어난 경우(연산결과가 너무 크거나, 너무 작을 경우)에 세트 되고, 그렇지 않은 경우에 클리어 됩니다.
DF	Direction Flag	“디렉션 플래그”라고 불리며, 스트링 명령(MOVS, CMPS, SCAS, LODS, STOS)이 동작하는 방향을 결정하는 플래그입니다.
SF	Sign Flag	“사인 플래그”라고 불리며, 연산결과가 양수 일 때에 클리어 되고 음수 일 때 세트 됩니다.

③ 인스트럭션 포인터

EIP는 인스트럭션 포인터라고 불립니다. 다음에 어떤 주소의 명령을 실행할지를 가리키는 포인터입니다.

④ 세그먼트 레지스터

CS, DS, ES, FS, SS, GS 등이 있습니다. 메모리 블럭을 지정하는 것이지만, Windows 32비트 환경에서는 거의 신경 쓸 필요가 없는 레지스터입니다.

⑤ 디버그 레지스터

이름 그대로 디버거가 사용하는 레지스터로, 프로그램 실행 상황을 확인하기 위한 것입니다. 리버싱을 하지 못하도록 안티 디버그(Anti Debug)에 사용되는 경우도 있습니다.

여기서 소개하지 못한 레지스터는 있지만, OS를 제작한다든지 하는 게 아니라면 그렇게까지 알 필요는 없습니다. 이 이후로 여기서 소개한 각 레지스터와 어셈블리 명령의 함수를 조금씩 이해해 간다면 크랙을 위한 지식으로서는 충분하다고 할 수 있습니다.

OllyDbg 를 사용한 명령의 확인(Confirmation of detail instruction that use OllyDbg)

어셈블리 언어에는 덧셈을 하는 ADD 뿐만 아니라 서브 루틴을 호출하는 CALL을 시작으로 매우 많은 명령이 있습니다. 그러나 잘 사용되는 명령은 한정되어 있기 때문에 외워야 하는 것은 그렇게 많지 않습니다. 여기서는 크랙에 필요한 최소한의 명령만을 설명하도록 하겠습니다.

① OllyDbg 인스톨

OllyDbg는 사용하기 쉽고 기능도 풍부한 디버거입니다. 현재 크랙, 리버싱 세계에서 표준이 된 디버거입니다. W32DASM, SoftICE, IDA Pro도 많이 쓰이지만, 사용하기 편하고 앞으로 발전 가능성이 높은 OllyDbg로 설명하도록 하겠습니다. 물론 상업용 디버거가 더 좋은 것은 사실이지만 개인적으로 구매하기에는 비용이 너무 많이 들 뿐만 아니라 리버싱, 크랙을 배우는 사람 입장에서는 공개 소프트웨어를 이용하는 편이 좋습니다.

크랙에 필요한 어셈블리어 기초

인터넷의 거의 모든 강좌가 공개 소프트웨어 위주로 설명되어 있기 때문입니다. OllyDbg로 디버깅이 어려운 부분은 다른 툴(SoftICE, PE Explorer, IDA Pro)로 설명하도록 하겠습니다.

디버거는 프로그램의 동작을 한 개의 명령씩 순차적으로 실행 가능하기 때문에 어셈블리 언어에서는 각 명령의 동작을 확인하는데 최적입니다.

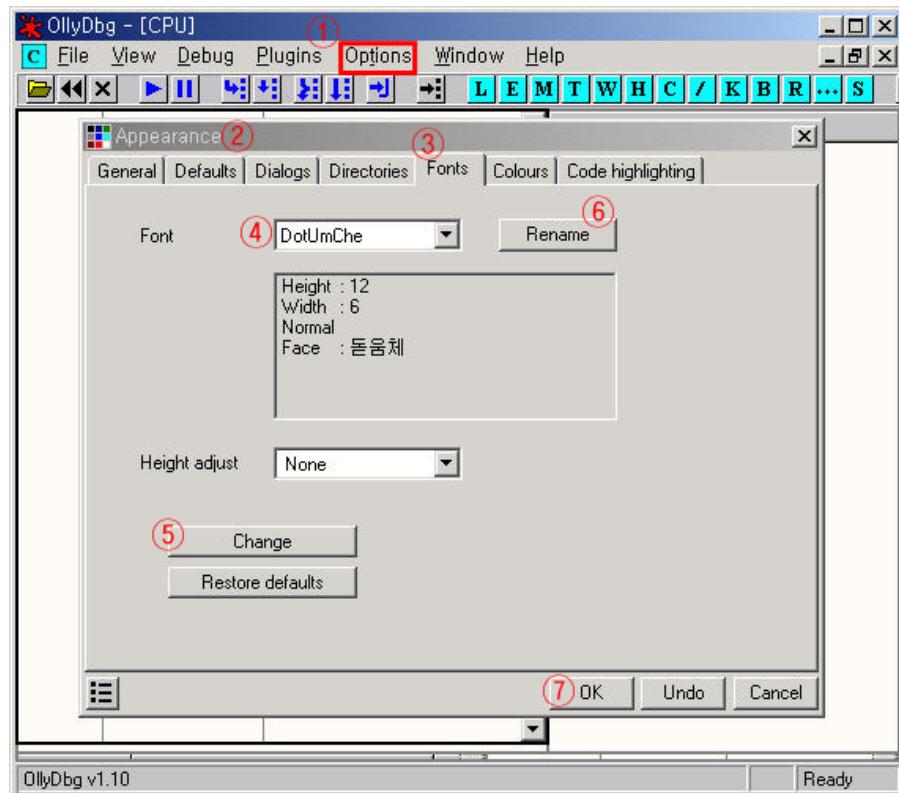
여기서부터는 OllyDbg를 사용하여 어셈블리 명령이 실제 어떻게 처리 되는지 확인하면서 설명해 나가겠습니다.

먼저 OllyDbg를 설치합니다. OllyDbg의 사이트(<http://www.ollydbg.de/>)에서 OllyDbg를 다운로드 합니다.

【Download】》【Download OllyDbg 1.10(final version)】에서 다운로드 할 수 있습니다. 다운로드 받은 파일을 적당한 폴더에 저장한 후 압축을 해제 하십시오.

OllyDbg는 기본 영문판이지만, 리버스 포럼⁴²에서 이미 한글화된 패치가 공개되어 있기 때문에 그것을 사용하셔도 됩니다. 마찬가지로 중국어, 일본어 문자열을 보고 싶다면 OllyDbg 폰트 플러그인을 다운로드 받아 OllyDbg 루트 디렉터리에 저장하시면 됩니다.

그리고 OllyDbg는 기본적으로 한글을 표시할 수 없으므로 한글폰트로 변경할 경우가 발생 될 수 있습니다. OllyDbg를 실행한 후 메뉴에서 【Options】》【Appearance】》【Fonts】탭을 선택합니다. Fonts 탭 화면에서 Font 드롭다운 메뉴의 【Font 5】를 선택하시고 하단의 【Change】버튼을 눌러 주십시오. 폰트 선택이 나타나면 한글 표시가 되는 폰트를 선택해 주십시오. 필자는 폰트를 【굴림체】, 글꼴 스타일 【보통】, 사이즈를 【9】, 스크립트를 【한글】로 했지만 OllyDbg에서 보기 편한 자신만의 글꼴 환경으로 설정하면 됩니다. 최종 선택이 끝나면 【Font 5】이름을 적당히 변경해 주시면 됩니다. 자세한 설정은 그림 3을 참조하십시오.



⁴² 리버스 포럼: <http://ampm.ddns.co.kr/~reverse/>

그림 3. 폰트 변경화면

그리고 OllyDbg 윈도우의 어디서든지 오른쪽 클릭 → 【Appearance】 → 【Font (all)】 → 【DotUmChe】를 선택하면 정의된 폰트로 OllyDbg 내 모든 윈도우에 반영됩니다. 상세한 내용은 아래 그림 4를 참조하십시오.

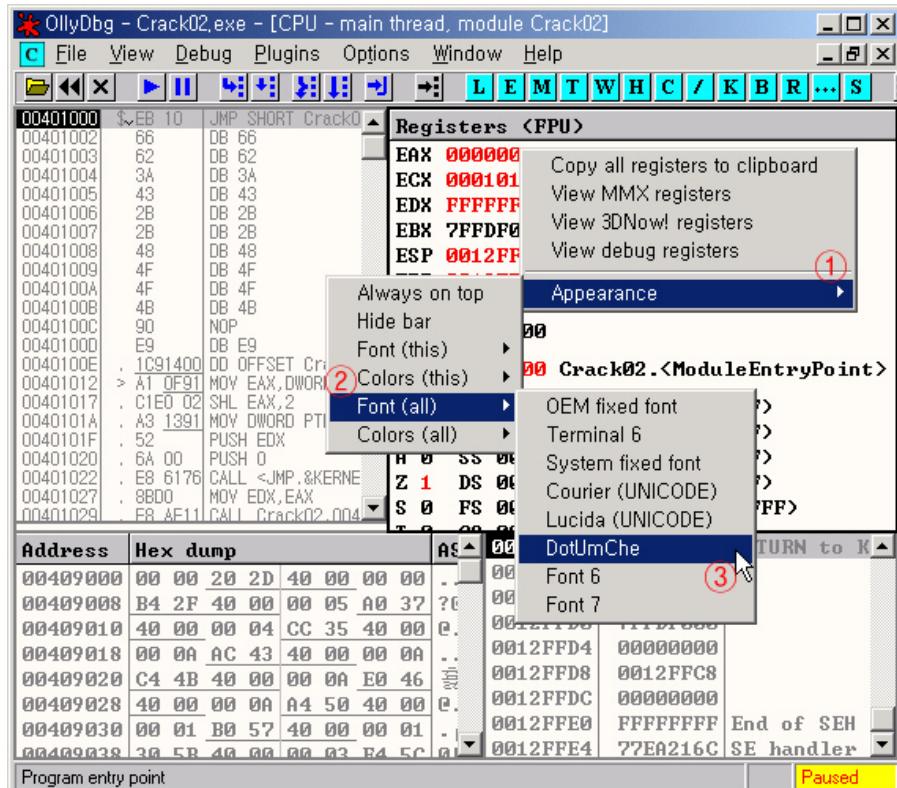


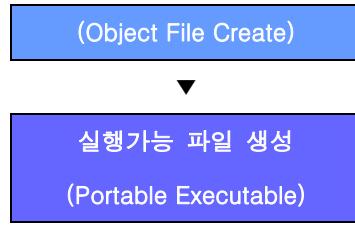
그림 4. OllyDbg 전체 폰트 변경

이것으로 기본 준비가 완료 되었습니다. 한글 리소스가 존재하는 경우 위와 같이 폰트를 변경하시면 됩니다. 나머지는 실제 어셈블리 언어를 설명 해 가면서 OllyDbg 의 기본적인 사용법도 설명하도록 하겠습니다.

샘플 어셈블리어 작성[Sample assembly language creation]

우선 어셈블리어를 설명하기 이전에, 본 문서에서 사용할 어셈블리어 샘플 파일을 작성하여 윈도우에서 컴파일해보도록 하겠습니다. 어셈블리언어로 작성된 파일을 윈도우 운영체제에서 실행 가능하도록 하려면 아래와 같은 과정을 거쳐야 합니다.





어셈블리 소스파일은 텍스트 편집기 등에서 어셈블리어 문법에 맞게 작성하시면 됩니다. 작성된 소스파일은 어셈블러(Assembler) 프로그램을 사용하여 오브젝트 파일(.obj)을 생성시키고 링커(Linker)라는 프로그램을 이용하여 실행 가능한 파일(.exe, .com, .sys, .dll등)로 만들어 주면 됩니다. 주로 사용되는 어셈블러 프로그램으로는 NASM, MASM, TASM, FASM, GoASM, RADASM등이 있고, 링커 프로그램도 GoLink, MASM Link, Radasm link등 다양하게 존재합니다. 필자는 설치과정과 특별한 환경 설정이 필요 없는 NASM⁴³, GoLink⁴⁴를 가지고 설명할 것입니다.

먼저 NASM 어셈블러를 다운로드(<http://sourceforge.net/projects/nasm>) 받아 적당한 폴더에 압축을 해제 합니다. 마찬가지로 GoLink 링커 프로그램도 다운로드(<http://www.jorgen.freeserve.co.uk/Golink.zip>) 받아 NASM 폴더에 압축을 해제합니다. NASM과 GoLink 실행파일을 동일 폴더에 복사하는 이유는 오브젝트 파일 생성과 링커 과정을 손쉽게 하기 위함입니다. 필자는 “D:\Assembler”폴더로 복사하였습니다. 개별 폴더에 각각 압축을 해제 하셔도 무방합니다.

첫 번째로 MOV 어셈블리어를 설명하기 위한 예제 입니다. 텍스트 편집기를 이용하여 아래와 같이 작성한 후 mov.s라는 파일로 저장합니다.

```

Notepad++ - D:\Assembler\mov.s
파일(F) 편집(E) 찾기(S) 보기(V) 형식(M) 언어(L) 설정(I)
매크로 실행 Plugins Window ?
mov.s
1 section .bss
2 buf resd 1
3 section .text
4     global start
5 start:
6     mov eax,7
7     mov ebx,eax
8     mov dword [buf],eax
9     mov edx,12345678h
10    mov dword [buf],edx
11
12    push 0
13    extern ExitProcess
14    call ExitProcess
15
  
```

그림 5. mov.s 어셈블리 파일

⁴³ NASM: <http://sourceforge.net/projects/nasm>

⁴⁴ GoLink: <http://www.jorgen.freeserve.co.uk/>

크랙에 필요한 어셈블리어 기초

작성이 완료되면 저장된 mov.s 파일을 오브젝트 파일로 변환하여 주기 위해 명령어 창에서 아래와 같이 실행합니다.

```
D:\WAssembler>nasmw.exe -fwin32 mov.s      ➔ -f: output format  
                                                ➔ win32: Microsoft Win32 (i386) object files
```

오브젝트 파일(mov.obj) 생성이 완료되면 Windows 라이브러리와 연결하여 실행파일로 만들어 줍니다.

```
D:\WAssembler>golink.exe -entry start mov.obj kernel32.dll
```

The screenshot shows a Windows Command Prompt window titled 'cmd.exe' running on 'C:\WINNT\system32'. The user has run the following commands:

```
C:\WINNT\system32>D:\WAssembler>nasmw -fwin32 mov.s  
D:\WAssembler>dir /w mov.obj | findstr "mov.obj"  
mov.obj  
D:\WAssembler>golink -entry start mov.obj kernel32.dll  
GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk  
Output file: mov.exe  
Format: win32 size: 1,536 bytes  
D:\WAssembler>dir /w mov.exe | findstr "mov.exe"  
mov.exe  
D:\WAssembler>
```

그림 6. 실행파일 만들기

위 모든 과정이 에러 없이 성공했다면 이제 크랙에 필요한 어셈블리어를 하나씩 설명하도록 하겠습니다.

○ MOV 명령

MOV 명령은 가장 많이 쓰여지는 명령 중에 하나입니다. 전송 명령 이라고 불리며, 레지스터나 메모리 등으로부터 다른 레지스터나 메모리에 값을 옮겨(복사)줍니다.

위에서 생성한 mov.exe 를 OllyDbg 에 로드해 봅시다. 먼저 좀 전에 인스톨 한 OllyDbg 를 실행하여 【File】 > 【Open】 로 mov.exe 를 불러 옵니다.

크랙에 필요한 어셈블리어 기초

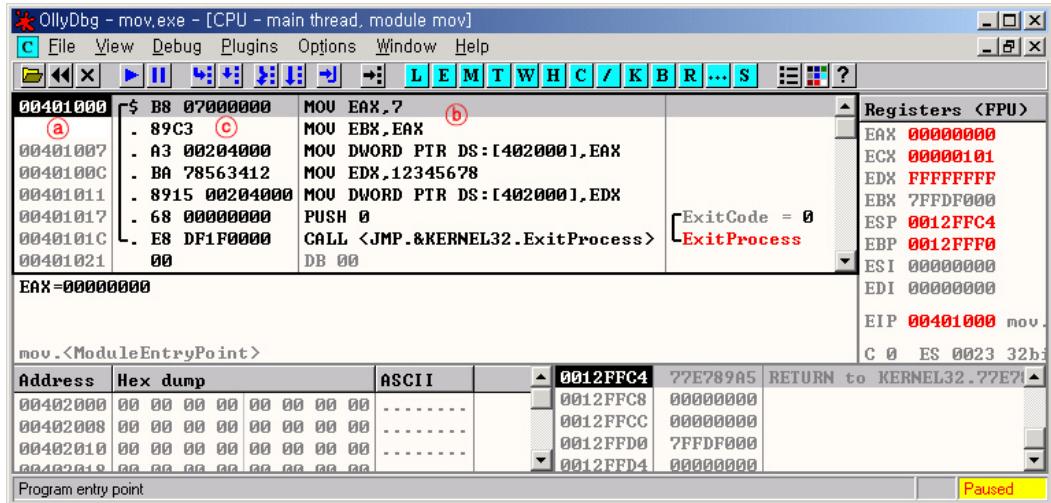


그림 7. OllyDbg에서 mov.exe를 불러온 화면

OllyDbg 전체 화면에서 왼쪽 위의 윈도우를 자세히 보면, 가장 왼쪽 00401000 부분(@영역)은 프로그램이 어느 위치에 있는지 알 수 있는 주소를 나타냅니다. 가장 오른쪽 【MOV EAX,7】 부분(⑥영역)이 명령을 디스어셈블한 결과입니다. 중앙(©영역)에 있는 것이 어셈블리어로 컴퓨터가 직접 해석 할 수 있는 기계어를 나타냅니다. 모든 숫자는 기본적으로 16 진수로 표현합니다.

그럼, 실제로 OllyDbg로 하나씩 명령을 실행(Step 실행)해 가면서 MOV 명령의 동작을 확인해 봅시다.

먼저 첫 번째 명령 【MOV EAX,7】을 보면, 이것은 EAX에 7이라는 숫자를 옮기라는 명령입니다. OllyDbg의 오른쪽 위의 윈도우에 있는 레지스터 윈도우로 EAX의 현재 값을 확인해 보십시오. 현재는 【00000000】입니다. 이 상태에서 키보드 F7 키(Step into)을 누르면, EAX의 값이 7로 변화하고(그림 8의 ②부분) 있는 것을 알 수 있을 것입니다.

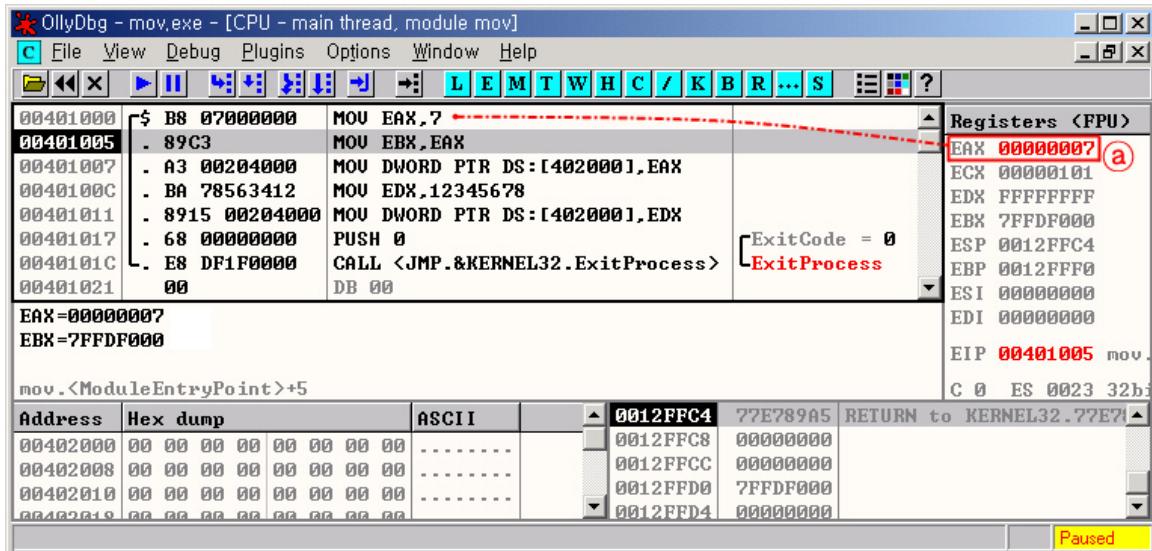


그림 8. EAX 레지스터 값의 변화

계속하여 【MOV EBX,EAX】를 실행해 봅시다. 여기서도 레지스터 윈도우에서 EAX와 EBX 값을 확인해 두십시오.

크랙에 필요한 어셈블리 기초

F7 키를 누르면 EBX가 7로 변화한 것을 알 수 있을 것입니다. 즉, EAX 값(7)을 EBX에 전송(복사)했기 때문입니다.

다음 명령을 봅시다. 【DWORD PTR DS:[402000】이라는 것은 402000h라는 주소의 더블 워드(4 바이트) 메모리를 가리키고 있습니다. 【MOV DWORD PTR DS:[402000],EAX】의 명령으로 EAX에 저장된 값을 402000h라는 주소에 복사합니다. OllyDbg의 왼쪽 하단 윈도우에 주소와 그 내용이 표시되어 있습니다. F7을 눌러서 주소 402000h에 있는 Hex 덤프 값의 변화를 봅시다. 402000h 주소에 07 값이 저장되었을 것입니다.

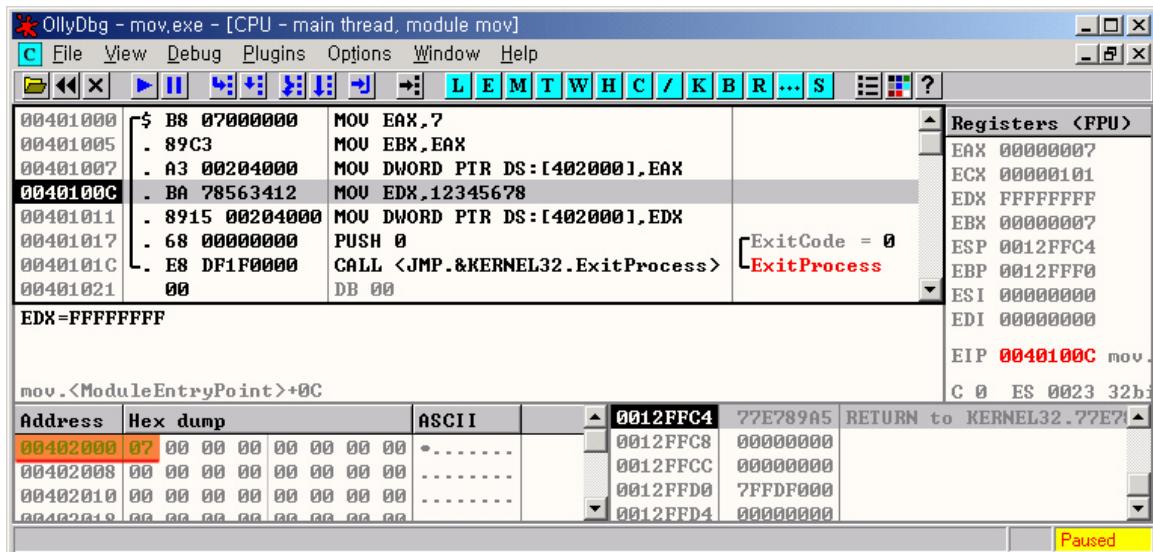


그림 9. 402000h 주소로 7 값을 복사

다음에 【MOV EDX,12345678】을 실행해 보십시오. 물론 EDX에 12345678이 복사됩니다. 계속하여 402000h에 있는 값을 확인하여, 【MOV DWORD PTR DS:[402000],EDX】을 실행해 봅시다. 그러면 402000h가 【78 56 34 12】라는 값으로 변해 있을 것입니다. 12345678h를 복사했는데 역순으로 저장되어 있습니다.

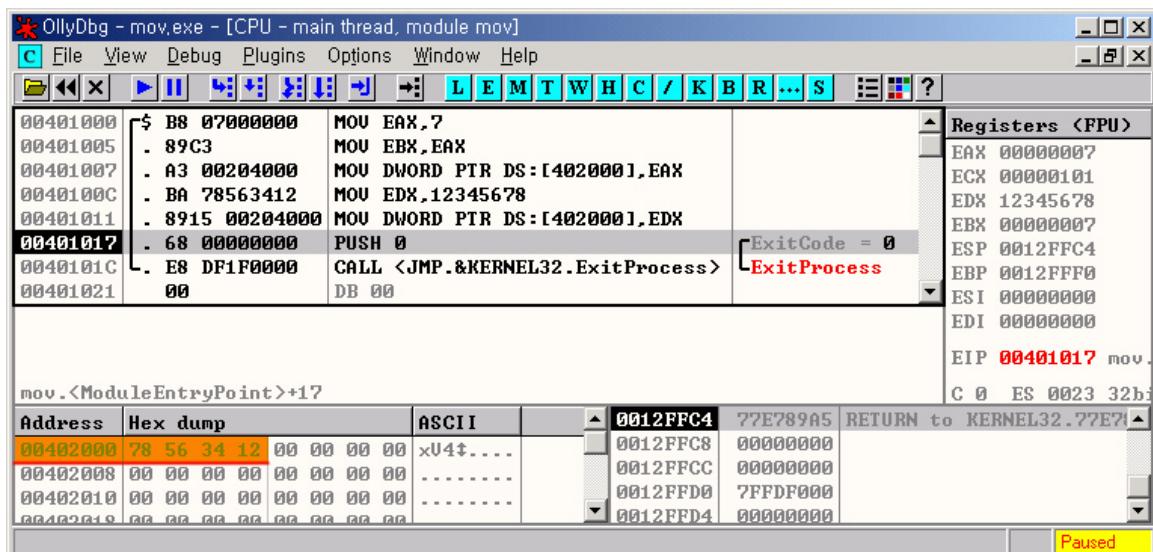
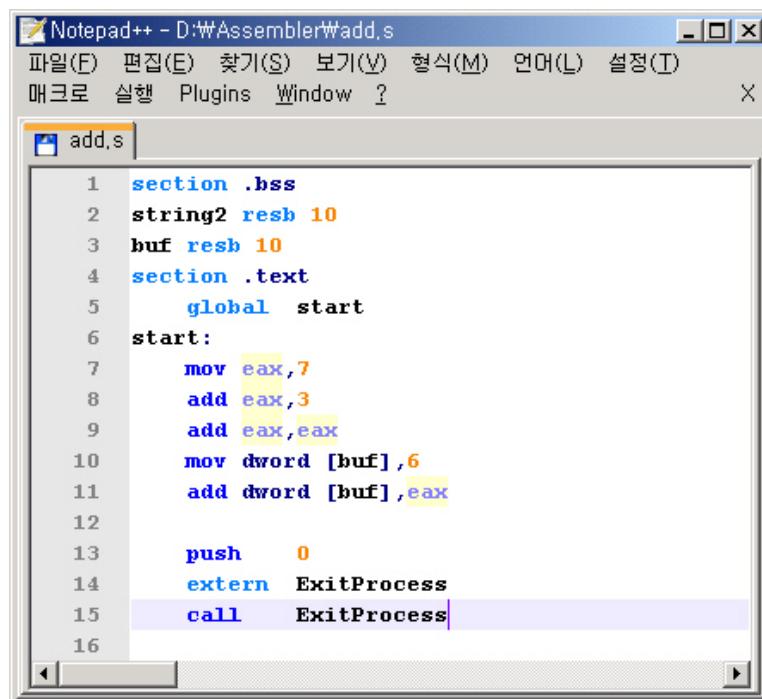


그림 10. 역순으로 저장(Little-Endian)

크랙에 필요한 어셈블리어 기초

이것은 x86 CPU 에서는 리틀 엔디안(Little-Endian)이라는 방법으로 숫자를 취급하고 있기 때문입니다. 리틀 엔디안 이라는 방법은 CPU 내부에서 1 바이트마다 역순으로 처리하는 것입니다. 예를 들어 워드 값을 취급할 경우 1234h 라는 숫자는 메모리상에서 【34 12】처럼 보존되며, 더블 워드 값 12345678h 라는 숫자는 【78 56 34 12】로 보존됩니다. 메모리 내용을 읽을 때 틀리지 않도록 주의 하여야 합니다. 이와 반대되는 개념으로 빅 엔디안(Big-Endian)이 존재합니다. 이는 CPU 에 따라 메모리에 저장되는 방식이 다르기 때문입니다. 인텔 프로세서나 DEC 의 알파 프로세서는 리틀 엔디안 방식을 사용하며, RISC 기반 프로세서와 Motorola 마이크로 프로세서는 빅 엔디안 방식을 사용합니다. 그러나 네트워크 통신은 모두 빅 엔디안 기법을 사용합니다. 이 기종 장비간의 통신을 원활히 하기 위해 빅 엔디안 방법만을 사용하기로 정한 것입니다. 예를 들어 네트워크를 통해 수신되는 IP 주소는 빅 엔디안 방식이므로 인텔 계열 운영체제에서는 꺼꾸로 읽지 않도록 주의해야 합니다.

두 번째로 ADD 어셈블리어를 설명하기 위한 예제 입니다. 텍스트 편집기를 이용하여 아래와 같이 작성한 후 add.s 라는 파일로 저장합니다.



```
Notepad++ - D:\Assembler\add.s
파일(E) 편집(E) 찾기(S) 보기(V) 형식(M) 언어(L) 설정(I)
마크로 실행 Plugins Window ?
add.s
1 section .bss
2 string2 resb 10
3 buf resb 10
4 section .text
5 global start
6 start:
7     mov eax,7
8     add eax,3
9     add eax, eax
10    mov dword [buf],6
11    add dword [buf],eax
12
13    push 0
14    extern ExitProcess
15    call ExitProcess
16
```

그림 11. add.s 어셈블리 파일

작성이 완료되면 저장된 add.s 파일을 오브젝트 파일로 변환하여 주기 위해 명령어 창에서 아래와 같이 실행합니다.

```
D:\Assembler> nasmw.exe -fwin32 add.s
```

오브젝트 파일(add.obj) 생성이 완료되면 Windows 라이브러리와 연결하여 실행파일로 만들어 줍니다.

```
D:\Assembler> glink.exe -entry start add.obj kernel32.dll
```

크랙에 필요한 어셈블리어 기초

```
C:\>cmd.exe  
D:\Assembler>nasmw -fwin32 add.s  
D:\Assembler>dir /w add.obj | findstr "add.obj"  
add.obj  
D:\Assembler>golink -entry start add.obj kernel32.dll  
GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk  
Output file: add.exe  
Format: win32 size: 1,536 bytes  
D:\Assembler>dir /w add.exe | findstr "add.exe"  
add.exe  
D:\Assembler>
```

그림 12. 실행파일 만들기

○ 산술 연산 명령

산술 연산 명령이란 덧셈, 뺄셈, 곱셈, 나눗셈 등을 행하는 명령입니다. 여기서는 덧셈을 하는 ADD, INC 명령과 뺄셈을 하는 SUB, DEC 명령에 대하여 알아 보겠습니다.

먼저 ADD 명령부터 봅시다. 위에서 생성한 add.exe 를 OllyDbg 로 불러와 주십시오.

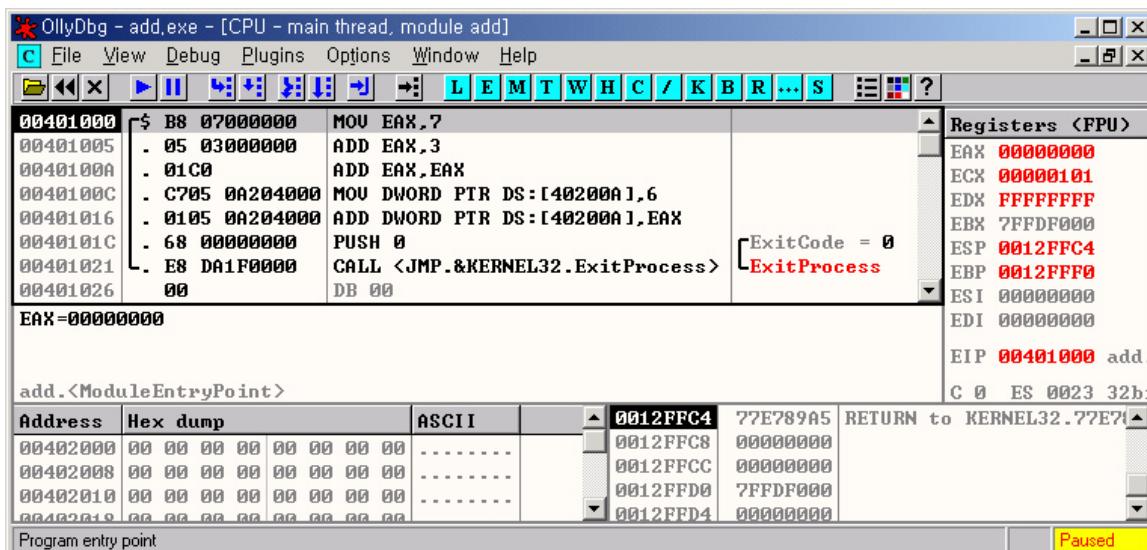


그림 13. add.exe 불러오기

OllyDbg 왼쪽 상단의 401000h 주소의 명령을 보면, 【MOV EAX,7】 은 앞의 MOV 명령처럼 EAX 에 7 을 옮겨줍니다. F7 키를 눌러 다음 단계로 진행하면 【ADD EAX,3】 명령으로 이동합니다. 이것은 EAX 에 3 를 더하는 명령으로, F7 키를 한번 더 눌러서 확인해 보면 EAX 가 7에서 Ah로 변화하는 것을 알 수 있습니다. 게다가 【ADD EAX,EAX】 에서 EAX 값 Ah 에 EAX 의 값 Ah 가 더해져 EAX 는 14h 가 됩니다. 레지스터 윈도우 값은 16 진수로 표기하고 있기 때문에 14h 로 되어 있습니다. (14h 는 10 진수에서는 20)

다음 명령을 봅시다. 【MOV DWORD PTR DS:[40200A],6】 로 되어 있습니다. OllyDbg 왼쪽 밑 윈도우의 40200A

크랙에 필요한 어셈블리어 기초

주소에 어떤 값이 들어 있는지 확인합니다. 이 시점에서 0이 되어 있다고 생각합니다. 여기서 F7 버튼을 누르면, 그 위치에 6이 대입됩니다. 다음에 【ADD DWORD PTR DS:[40200A],EAX】로 EAX의 값이 40200Ah의 위치에 더해집니다. $6h + 14h = 1Ah$ 이므로 40200Ah 값이 1Ah가 되어 있을 것입니다.

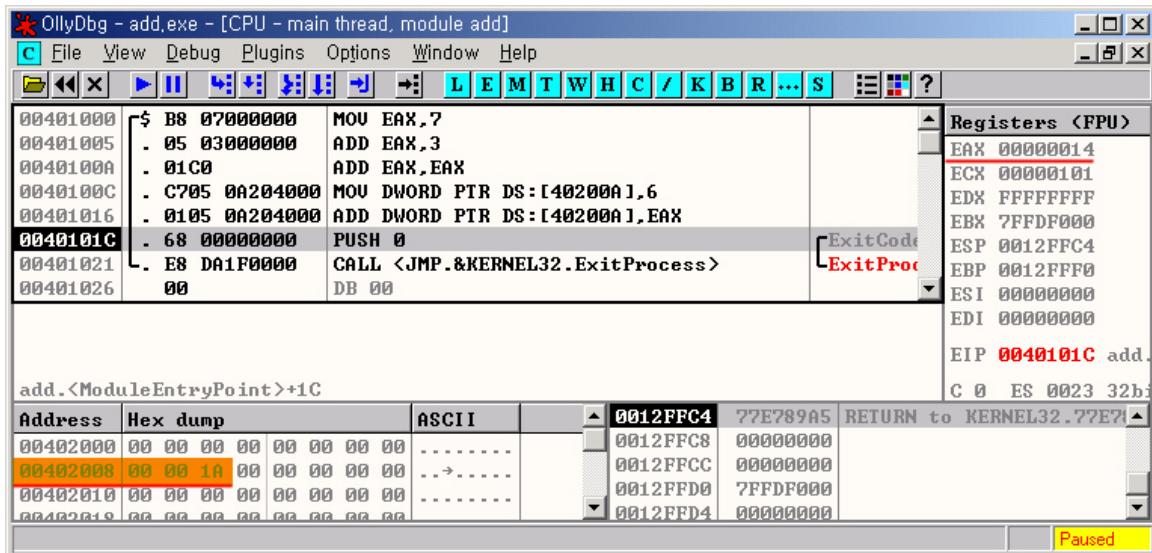


그림 13. ADD 명령 결과

다음으로 SUB 명령을 봅시다. SUB 는 뺄셈하는 명령어입니다. 예를 들어, 【SUB EAX,EBX】 가 있으면 EAX 에서 EBX 값을 빼서, 결과를 EAX 에 보관됩니다. 사용법은 ADD 와 완전히 같기 때문에 특별히 설명하지 않도록 하겠습니다.

다음으로 INC, DEC 2 개 명령에 대하여 알아 봅시다. INC(increment)는 【1 을 증가한다】 , DEC(decrement)는 【1 을 감소한다】 는 명령입니다. 텍스트 편집기를 이용하여 아래와 같이 작성한 후 incdec.s 라는 파일로 저장합니다.

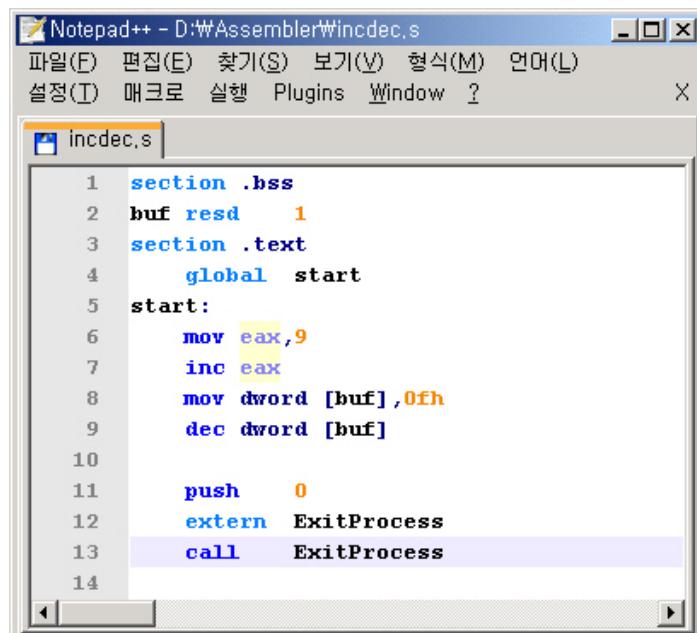


그림 14. incdec.s 어셈블리 파일

작성이 완료되면 저장된 incdec.s 파일을 오브젝트 파일로 변환하여 주기 위해 명령어 창에서 아래와 같이 실행합니다.

```
D:\WAssembler>nasmw.exe -fwin32 -O3 incdec.s ➔ -O(Optimize branch offsets)
```

오브젝트 파일(incdec.obj) 생성이 완료되면 Windows 라이브러리와 연결하여 실행파일로 만들어 줍니다.

```
D:\WAssembler>golink.exe -entry start incdec.obj kernel32.dll
```

```
C:\> C:\WINNT\system32\cmd.exe

D:\WAssembler>nasmw -fwin32 -O3 incdec.s

D:\WAssembler>dir /w incdec.obj | findstr "incdec.obj"
incdec.obj

D:\WAssembler>golink -entry start incdec.obj kernel32.dll

GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk
Output file: incdec.exe
Format: win32 size: 1,536 bytes

D:\WAssembler>dir /w incdec.exe | findstr "incdec.exe"
incdec.exe

D:\WAssembler>
```

그림 15. 실행 파일 만들기

OllyDbg에서 incdec.exe를 불러 옵시다. OllyDbg 불러온 파일에서 주소 401000h의 명령은 이제 금방 이해하시리라 생각합니다. EAX에 9h를 옮기는 명령입니다. 다음에 【INC EAX】는 EAX에 1이 증가 됩니다. 결과적으로 EAX는 Ah가 됩니다. 다음으로 401006h의 명령입니다. 주소 402000h의 더블 워드 위치에 0Fh가 대입됩니다. 다음 명령 【DEC DWORD PTR DS:[402000】은 지정한 주소 위치에 있는 값을 1을 감소 시킵니다. 즉 0Fh가 0Eh가 됩니다. INC와 DEC는 단순하지만 많이 쓰이는 명령입니다.

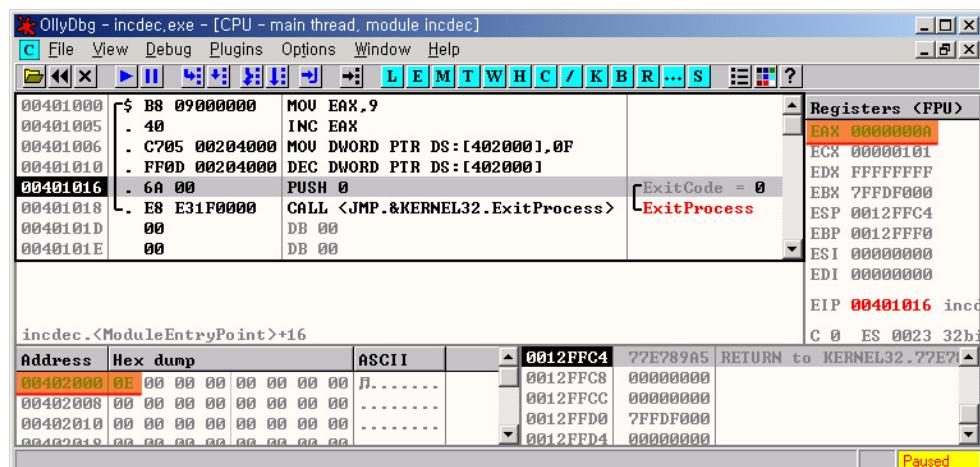


그림 16. INC, DEC 명령 결과

○ 논리 연산 명령

어셈블리 언어에서는 1 비트 단위 처리가 매우 중요합니다. 1 비트란 0이나 1로 나타나는 단위입니다. 이런 비트 연산을 사용하면 1비트 단위의 처리나 마스크 연산이 가능합니다. 여기서는 NOT, AND, OR, XOR 등의 비트연산을 알아보겠습니다.

먼저 가장 간단한 NOT 명령을 봅시다. NOT 명령은 부정연산이라고 불리며, 각 비트를 반전시키는 명령입니다. 예를 들어, 11100100b라는 숫자가 있다고 합시다. 이 숫자의 NOT 연산은 아래와 같습니다.

NOT	1	1	1	0	0	1	0	0
	0	0	0	1	1	0	1	1

0비트가 1로, 1비트가 0으로 전환되는 것으로 매우 단순합니다. 실제로 프로그램을 디버거로 쫓아가면서 해석해 나가면 숫자 표현은 16진수이므로 어렵게 느낄지도 모르겠습니다. NOT 명령뿐만 아니라 논리 연산이 사용되는 부분은 어느 정도 한정되어 있습니다. 익숙해 지면 2진수로 변환하지 않고 머리 속에서 계산할 수 있는 것도 많아집니다. 귀찮더라도 처음에는 Windows 계산기를 사용하여 하나씩 변환해 가면서 이해 하십시오.

다음으로 AND 명령을 봅시다. AND 명령은 각 비트가 모두 1일 때는 1이 되고, 그 이외에는 0이 됩니다. 10011000b와 10001011b의 AND 연산은 아래와 같습니다.

AND	1	0	0	1	0	1	1	0
	1	0	1	0	0	1	0	1
	1	0	0	0	0	1	0	0

다음으로 OR 명령입니다. OR 명령은 양 비트 중에 한 개라도 1이 있으면 1이 됩니다. 10011000b와 10001011b의 OR 연산은 아래와 같습니다.

OR	1	0	1	0	0	1	1	0
	1	0	1	1	0	1	1	1
	1	0	1	1	0	1	1	1

다음은 XOR 명령입니다. XOR 명령은 양 비트가 같으면 0, 다르면 1이 됩니다. 10011000b와 10001011b의 XOR 연산은 아래와 같습니다.

XOR	1	0	0	1	0	1	1	0
	1	0	1	0	0	1	0	1

크랙에 필요한 어셈블리어 기초

0	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

논리 연산이 실제 프로그램에서 어떻게 이루어지는지 알아 봅시다. 텍스트 편집기를 이용하여 아래와 같이 작성한 후 logical_calc.s라는 파일로 저장합니다.

The screenshot shows the Notepad++ application window with the file 'logical_calc.s' open. The code is as follows:

```
1 section .bss
2 buf resd 1
3 section .text
4     global start
5 start:
6     mov al,10011101b
7     not al
8     and al,01111001b
9     or al,11100011b
10    xor al,01101000b
11
12    push 0
13    extern ExitProcess
14    call ExitProcess
15
```

그림 17. logical_calc.s 어셈블리 파일

작성이 완료되면 저장된 logical_calc.s 파일을 오브젝트 파일로 변환하여 주기 위해 명령어 창에서 아래와 같이 실행합니다.

```
D:\WAssembler>nasmw.exe -fwin32 -O3 logical_calc.s ➔ -O(Optimize branch offsets)
```

오브젝트 파일(logical_calc.obj) 생성이 완료되면 Windows 라이브러리와 연결하여 실행파일로 만들어 줍니다.

```
D:\WAssembler>golink.exe -entry start logical_calc.obj kernel32.dll
```

The screenshot shows a Windows Command Prompt window with the following command history:

```
C:\> C:\WINNT\system32\cmd.exe
D:\WAssembler>nasmw -fwin32 -O3 logical_calc.s
D:\WAssembler>golink -entry start logical_calc.obj kernel32.dll
GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk
Output file: logical_calc.exe
Format: win32 size: 1,536 bytes
D:\WAssembler>
```

그림 18. 실행파일 만들기

크랙에 필요한 어셈블리 기초

OllyDbg에서 logical_calc.exe를 불러 옵시다. 첫 번째 【MOV AL,9D】는 AL 레지스터에 9Dh(10011101b)를 옮기는 명령입니다. 하위 8 비트 수치를 취급하기 때문에 AL 레지스터(EAX의 하위 8 비트)를 사용합니다. 다음으로 【NOT AL】에 의해 비트가 반전하여 01100010b가 됩니다. 이것은 16 진수로 62h입니다. OllyDbg에서 F7 키를 눌러서 확인해 보십시오. 다음으로 【AND AL,79】입니다. 79h는 2 진수로 1111001b입니다. 지금 AL에 01100010b(62h)가 들어있기 때문에 다음과 같이 됩니다.

	0	1	1	0	0	0	1	0
AND	0	1	1	1	1	0	0	1
	0	1	1	0	0	0	0	0

01100000b는 16 진수로 60h입니다. F7 키를 눌러 한 단계 진행하면, 아래 그림과 같이 AL 레지스터가 60h로 변화된 것을 알 수 있습니다.

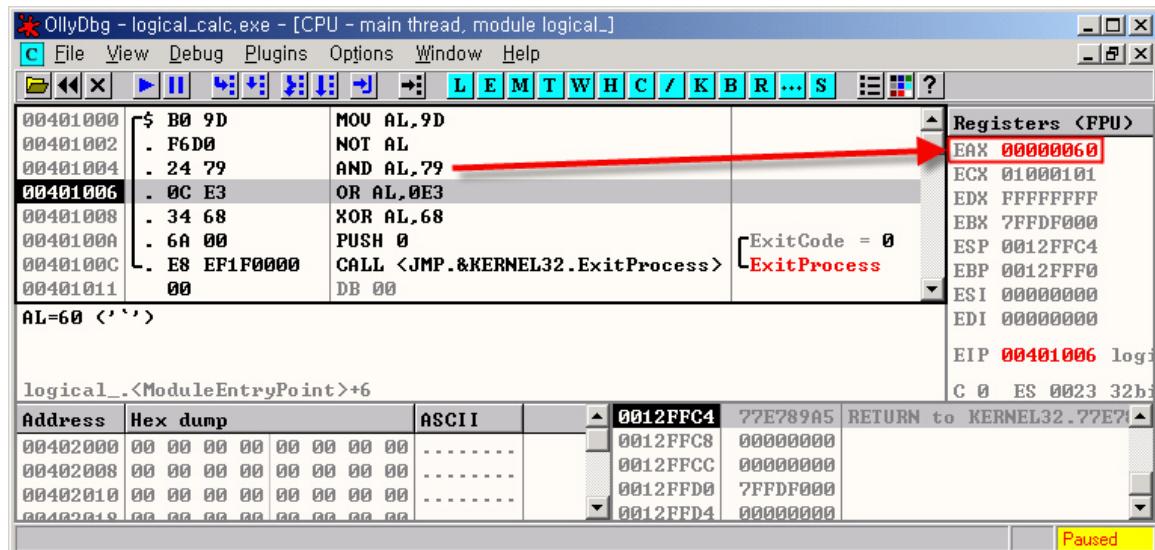


그림 19. AND 연산 결과

다음 명령 【OR AL,0E3】에서 0E3h는 11100011b이므로 다음과 같이 됩니다.

	0	1	1	0	0	0	0	0
OR	1	1	1	0	0	0	1	1
	1	1	1	0	0	0	1	1

11100011b는 16 진수로 0E3h가 됩니다. F7 키를 눌러 한 단계 진행하면, 아래 그림과 같이 AL 레지스터가 0E3h로 변화된 것을 알 수 있습니다.

크랙에 필요한 어셈블리 기초

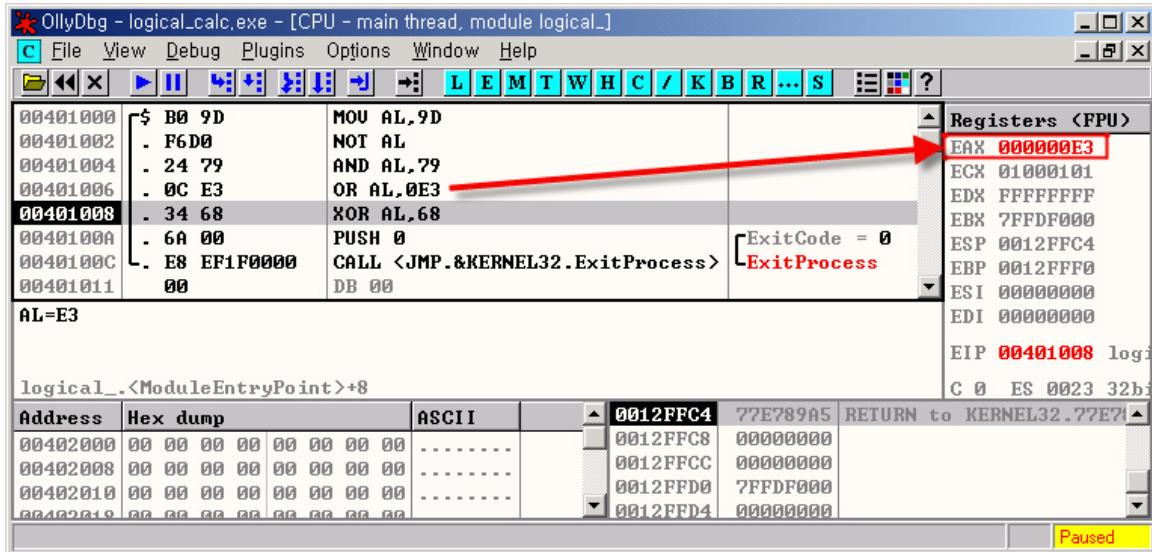


그림 20. OR 연산 결과

다음 명령 【XOR AL,68】에서, 68h 는 2 진수로 1101000b 이므로 다음과 같이 됩니다.

	1	1	1	0	0	0	1	1
XOR	0	1	1	0	1	0	0	0
	1	0	0	0	1	0	1	1

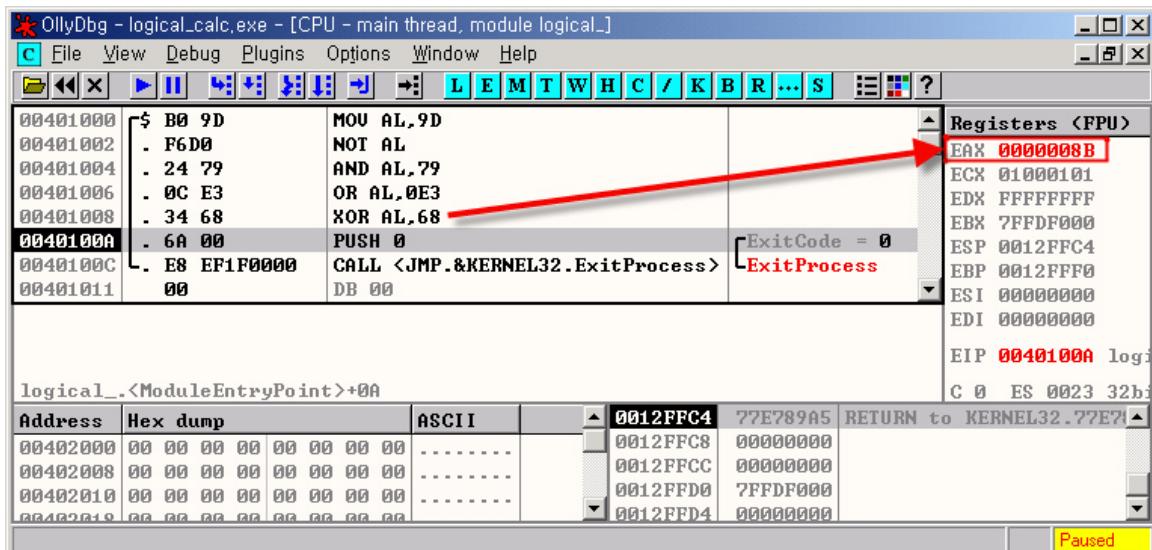


그림 21. XOR 연산 결과

하나씩 천천히 생각해 보면, 아주 단순하다는 것을 알 수 있습니다.

○ 스택(Stack)의 조작

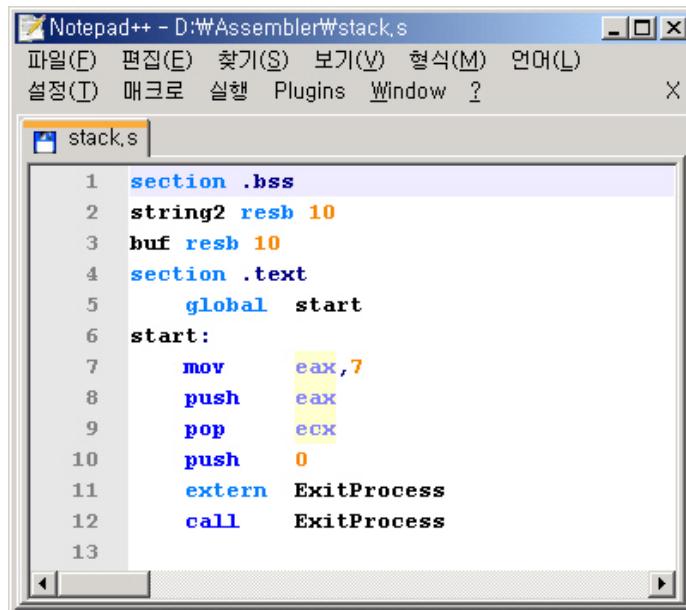
어셈블리 언어에서는 스택영역(Stack Area)이라고 불리는 메모리상의 영역을 이용하는 경우가 많습니다. 스택은

아래 목적으로 사용됩니다.

- ① 현재 레지스터 상태를 보존해 둘 때
- ② 서브 루틴에 인수를 건네줄 때
- ③ 일시적으로 데이터를 보존해 둘 때

스택 영역이 보통 메모리 영역과 다른 것은 데이터의 보존이나 추출 방법이 다를 뿐입니다. 실제로 스택 영역은 일반적인 메모리 접근과 동일한 방법으로 읽고 쓰기가 됩니다. 중요한 점은, 스택은 LIFO(Last-in First-out)라는 방식으로 사용된다는 것입니다. 다른 말로 후입선출법(後入先出法), 즉, 마지막에 저장한 것이 제일 먼저 추출된다는 것입니다. 예를 들어 책을 바닥에 두고, 그 위에 계속 책을 쌓아갑니다. 책 더미를 무너뜨리지 않고 빼려고 한다면, 위에서부터 순서대로 책을 뺄 수밖에 없습니다. 스택은 이것과 똑같습니다. PUSH라는 명령으로 데이터를 쌓아 나가다가, POP이라는 명령으로 위에서부터 순서대로 데이터를 빼 내 갑니다. 스택의 최상위 어드레스는 ESP(스택 포인터)라는 레지스터가 항상 지시하고 있습니다. 일부러 어드레스를 지정할 필요가 없으며, 이러한 특성 때문에 특수한 용도로 사용합니다(실제 x86 스택은 아래로 데이터가 계속 쌓여감. 그러므로 ESP는 맨 아래 부분을 가리키고 있음).

그러면 스택을 조작하는 기본명령인 PUSH 와 POP 에 대하여 알아 보겠습니다. 텍스트 편집기를 이용하여 아래와 같이 작성한 후 stack.s라는 파일로 저장합니다.



The screenshot shows a Notepad++ window with the file 'stack.s' open. The code is as follows:

```
1 section .bss
2 string2 resb 10
3 buf resb 10
4 section .text
5 global start
6 start:
7     mov    eax, 7
8     push   eax
9     pop    ecx
10    push   0
11    extern ExitProcess
12    call   ExitProcess
13
```

그림 22. stack.s 어셈블리 파일

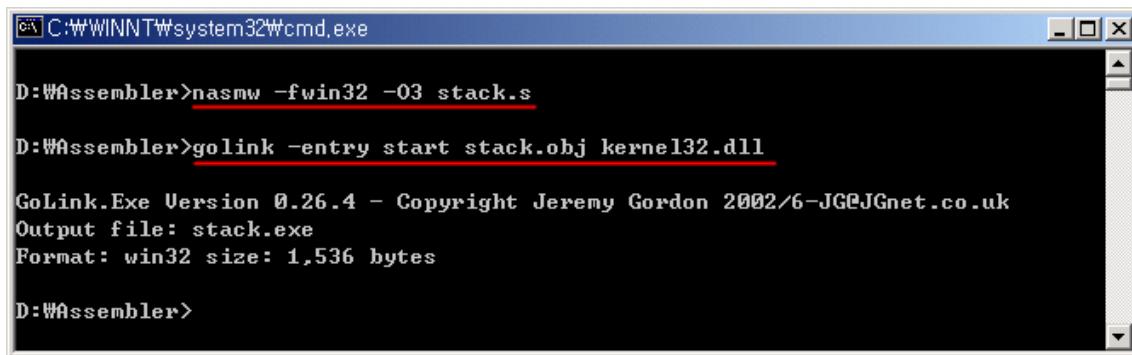
작성이 완료되면 저장된 stack.s 파일을 오브젝트 파일로 변환하여 주기 위해 명령어 창에서 아래와 같이 실행합니다.

```
D:\WAssembler>nasmw.exe -fwin32 -O3 stack.s ➔ -O(Optimize branch offsets)
```

크랙에 필요한 어셈블리어 기초

오브젝트 파일(stack.obj) 생성이 완료되면 Windows 라이브러리와 연결하여 실행파일로 만들어 줍니다.

```
D:\WAssembler>golink.exe -entry start stack.obj kernel32.dll
```



```
C:\WINNT\system32\cmd.exe

D:\WAssembler>nasmw -fwin32 -O3 stack.s
D:\WAssembler>golink -entry start stack.obj kernel32.dll

GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk
Output file: stack.exe
Format: win32 size: 1,536 bytes

D:\WAssembler>
```

그림 23. 실행파일 만들기

OllyDbg에서 stack.exe를 불러 옵시다. 불러온 화면에서 최초 MOV 명령은 EAX에 7을 대입(①영역)합니다. 이 시점에서 레지스터 윈도우의 ESP(스택 포인터) 값을 기억해 두십시오. 필자의 환경에서는 ESP는 0012FFC4로(②영역) 되어 있습니다. 여기에서 지금까지 설명하지 않았던 OllyDbg 오른쪽 밑 윈도우를 보십시오. 이것은 스택 영역을 표시(③영역)하고 있는 윈도우입니다. 아무것도 조작하고 있지 않는 상태라면, 제일 위에 ESP의 어드레스와 그 내용이 표시되어 있을 것입니다.

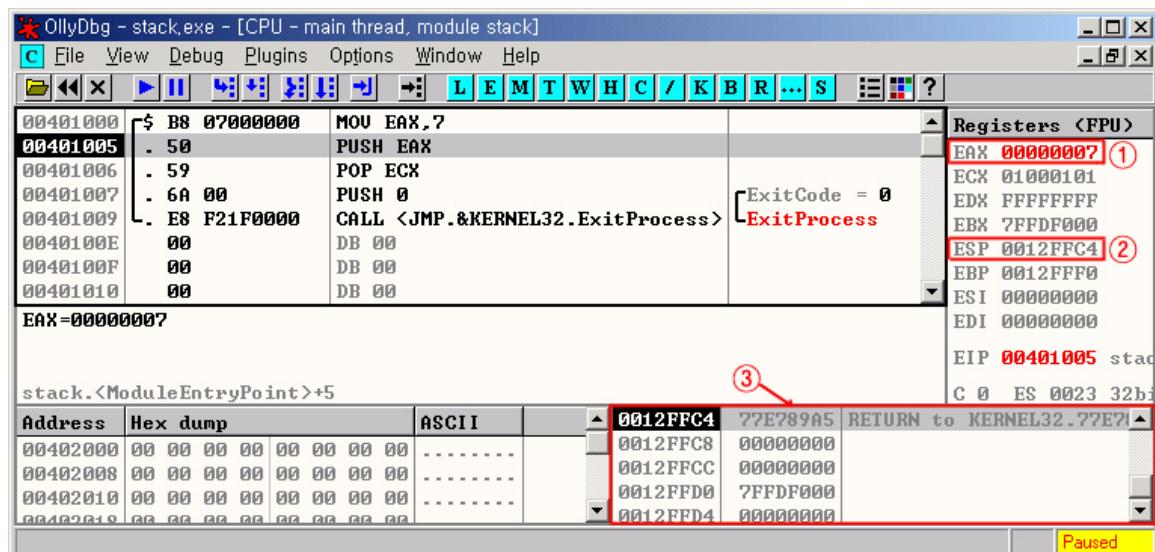


그림 24. ESP 주소

다음 【PUSH EAX】라는 명령을 한 단계 실행(F7 키)시켜 봅시다. ESP 값이 -4(4byte)가 되어 0012FFC0가 되고, 그 다음 단계는 좀 전에 스택에 쌓은 값이 ECX에 들어가 7이 되었습니다. PUSH로 스택에 값을 쌓고, POP에서 값을 추출하는 간단한 프로그램입니다.

일반적으로 스택에 값을 쌓는 경우는 어디에선가 저장된 값을 추출하기 위함입니다. 앞에서 설명한 대로 필자의 환경에서 이 프로그램 초기 ESP는 0012FFC4를 가리키고 있으며 그 내용은 아래와 같이 되어 있습니다.

크래ք에 필요한 어셈블리어 기초

0012FFC4	77E789A5	RETURN to KERNEL32.77E789A5
0012FFC8	00000000	
0012FFCC	00000000	
0012FFD0	7FFDF000	
0012FFD4	00000000	

그림 25. 초기 ESP

Windows 프로그램이 실행되면 kernel32라는 OS의 메인 모듈로부터 호출되고, 프로그램이 종료하면 프로그램을 최초로 호출했던 곳으로 되돌릴 필요가 있습니다. 그렇기 때문에 어디로 되돌릴지를 기억하고 있는 장소는 스택의 초기상태인 가장 최상위 위치(RETURN 처리 부분)가 됩니다. 지금 이런 설명을 해도 전혀 이해할 수 없을 거라 생각됩니다만, 일단 그런 것이 있다라는 정도만 기억해 두십시오. 여기에서 중요한 것은 스택은 어떤 처리에 의해 최종적으로 원래 상태로 돌아간다는 것입니다.

○ 조건 점프 명령

프로그램은 어떤 조건에 의해 다음에 무엇을 할 것인지를 제어하는 것이 필요합니다. 이런 곳에는 조건 점프 명령이 사용됩니다. 조건 점프 명령을 설명하기 이전에, 무조건 점프 명령인 JMP 명령에 대하여 설명 하겠습니다. JMP는 【JMP wxyz】 형태로 사용되며, wxyz라는 어드레스에 제어를 건네줄 수 있습니다. 아래 그림과 같이 “AAAA” ~ “XXXX”까지의 프로그램 흐름에서 “AAAA” → “BBBB” → “JMP WXYZ” → “WWWW” → “XXXX”라는 순서로 실행되게 됩니다. 무조건으로 WXYZ로 점프하는 것입니다.

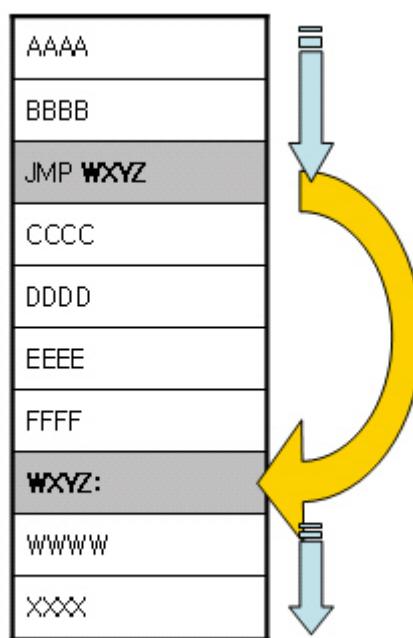


그림 26. 무조건 분기 JMP 예

그럼, 조건 점프에 대하여 알아 봅시다. 조건 점프를 하는 명령에 아래와 같은 것이 있습니다.

크래프트에 필요한 어셈블리어 기초

○ 조건 점프 명령어(x86 CPU)

【조건점프명령 OP1, OP2】

명령어 1	명령어 2	설명	부등호 조건	Flag 조건
JA	JNBE	크면 분기(부호 없이 비교[Unsigned])	"OP1 > OP2"면 분기	CF=0 & ZF=0
JAE	JNB	크거나 같으면 분기(부호 없이 비교[Unsigned])	"OP1 ≥ OP2"면 분기	CF=0 ZF=1
JB	JNAE	작으면 분기(부호 없이 비교[Unsigned])	"OP1 < OP2"면 분기	CF=1
JBE	JNA	작거나 같으면 분기(부호 없이 비교[Unsigned])	"OP1 ≤ OP2"면 분기	CF=1 ZF=1
JG	JNLE	크면 분기(부호 있는 비교[signed])	"OP1 > OP2"면 분기	ZF=0 & SF == OF
JGE	JNL	크거나 같으면 분기(부호 있는 비교[signed])	"OP1 ≥ OP2"면 분기	SF == OF
JL	JNGE	작으면 분기(부호 있는 비교[signed])	"OP1 < OP2"면 분기	SF != OF
JLE	JNG	작거나 같으면 분기(부호 있는 비교[signed])	"OP1 ≤ OP2"면 분기	ZF == 1 SF != OF
JE		같으면 분기	"OP1 = OP2"면 분기	ZF == 1
JNE		같지 않으면 분기	"OP1 ≠ OP2"면 분기	ZF == 0
JC		Carry Flag 가 Set 되면 분기		CF == 1
JNC		Carry Flag 가 해제되어 있으면 분기		CF == 0
JO		Overflow 가 Set 되어 있으면 분기		OF == 1
JNO		Overflow 가 해제되어 있으면 분기		OF == 0
JS		Sign Flag 가 Set 되어 있으면(음수이면) 분기		SF == 1
JNS		Sign Flag 가 해제되어 있으면(양수이면) 분기		SF == 0
JZ		Zero Flag 가 Set 되어 있으면 분기		ZF == 1
JNZ		Zero Flag 가 해제되어 있으면 분기		ZF == 0
JP		Parity Flag 가 Set 되어 있으면 분기		PF == 1
JNP		Parity Flag 가 해제 되어 있으면 분기		PF == 0

표. 조건 점프 명령어

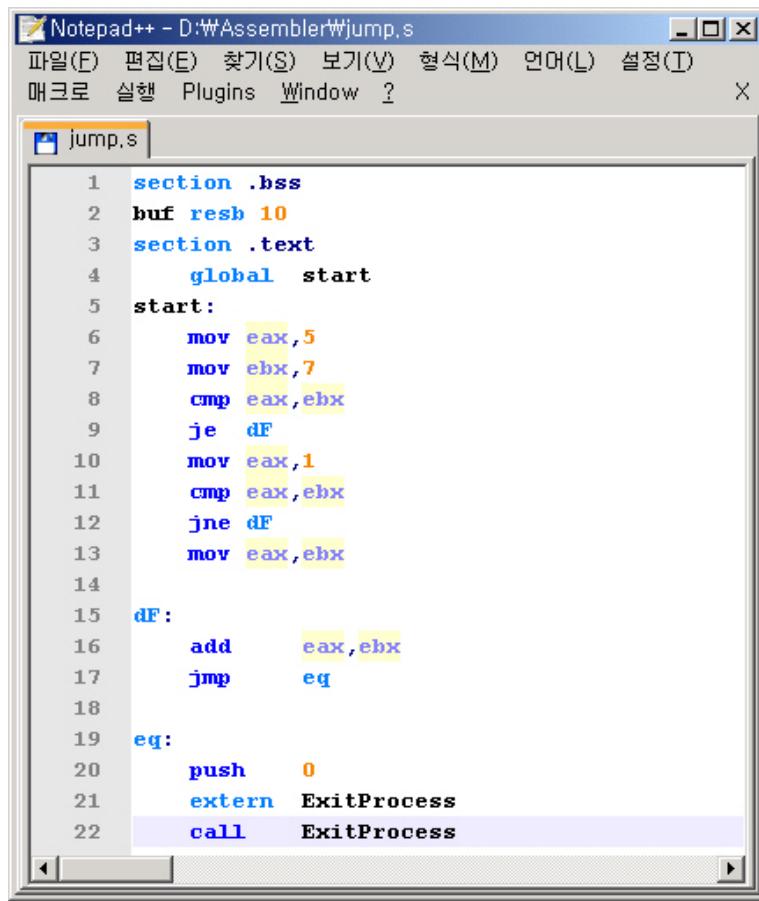
※ 주의사항

- ▶ ZF 는 $OP2 - OP1 = 0$ 인 경우 1로 Set 됩니다. 즉 $OP2$ 와 $OP1$ 이 같을 경우 Set(1)됩니다.
- ▶ CF 는 $OP2 - OP1$ 의 경우 $OP2$ 에서 자리 빌림이 발생할 때 Set(1)됩니다. 즉 레지스터의 최대값을 넘었을 때 변경됩니다.
- ▶ OF (Overflow flag)는 Sign bit(맨 앞 비트)가 변경될 때(즉, 부호가 바뀔 때) 발생합니다. 32bit 레지스터이면 Bit31 가 변경될 때 발생합니다.
- ▶ JE, JNE 명령은 CMP 를 통해 단일 Operand 를 통해 비교한 후 분기할 수 도 있음

대부분의 프로그램은 산술연산을 시작으로, 다양한 명령으로 플래그나 기타 레지스터의 값을 변화 시킵니다. 그 변화에 의해 생긴 상태를 판별하기 위해 조건 점프 명령을 사용하여 제어 분기를 행할 수 있습니다. 제어 분기는 크래킹에 있어서 가장 중요한 명령군이 됩니다.

크랙에 필요한 어셈블리 기초

그러면 조건분기명령을 조작하는 기본명령인 JE 와 JNE 에 대하여 알아 보겠습니다. 텍스트 편집기를 이용하여 아래와 같이 작성한 후 jump.s 라는 파일로 저장합니다.



```
Notepad++ - D:\Assembler\jump.s
파일(F) 편집(E) 찾기(S) 보기(V) 형식(M) 언어(L) 설정(I)
마크로 실행 Plugins Window ?
jump.s
1 section .bss
2 buf resb 10
3 section .text
4     global start
5 start:
6     mov eax,5
7     mov ebx,7
8     cmp eax,ebx
9     je dF
10    mov eax,1
11    cmp eax,ebx
12    jne dF
13    mov eax,ebx
14
15 dF:
16    add eax,ebx
17    jmp eq
18
19 eq:
20    push 0
21    extern ExitProcess
22    call ExitProcess
```

그림 27. jump.s 어셈블리 파일

작성이 완료되면 저장된 jump.s 파일을 오브젝트 파일로 변환하여 주기 위해 명령어 창에서 아래와 같이 실행합니다.

```
D:\Assembler>nasmw.exe -fwin32 -O3 jump.s ➔ -O(Optimize branch offsets)
```

오브젝트 파일(jump.obj) 생성이 완료되면 Windows 라이브러리와 연결하여 실행파일로 만들어 줍니다.

```
D:\Assembler>golink.exe -entry start jump.obj kernel32.dll
```



```
C:\WINNT\system32\cmd.exe
D:\Assembler>nasmw -fwin32 -O3 jump.s
D:\Assembler>golink -entry start jump.obj kernel32.dll
GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk
Output file: jump.exe
Format: win32 size: 1,536 bytes
D:\Assembler>
```

그림 28. 실행파일 만들기

크랙에 필요한 어셈블리어 기초

OllyDbg에서 jump.exe를 불러 옵시다. 아래와 같은 명령들이 보일 것입니다.

```
1 00401000 >/$ B8 05000000 MOV EAX,5
2 00401005 I. BB 07000000 MOV EBX,7
3 0040100A I. 39D8 CMP EAX,EBX
4 0040100C I. 74 0B JE SHORT jump.00401019
5 0040100E I. B8 01000000 MOV EAX,1
6 00401013 I. 39D8 CMP EAX,EBX
7 00401015 I. 75 02 JNZ SHORT jump.00401019
8 00401017 I. 89D8 MOV EAX,EBX
9 00401019 I> 01D8 ADD EAX,EBX
10 0040101B I. EB 00 JMP SHORT jump.0040101D
11 0040101D I> 6A 00 PUSH 0
12 0040101F \. E8 DC1F0000 CALL <JMP.&KERNEL32.ExitProcess>
```

그림 29. Jump.exe 디스어셈블리 덤프

먼저 1,2 번째 줄 명령으로 EAX에 5가 EBX에 7이 복사됩니다. 3 번째 줄에서 CMP 명령으로 EAX와 EBX를 비교합니다. CMP 명령은 실제로는 [EBX-EAX]라는 계산을 한 후 그 결과에 따라 플래그를 세트 합니다(EAX, EBX의 값은 변화하지 않음). 다음 4 번째 줄에서 JE 명령으로 CMP에서 비교한 값이 같으면 00401019h에 점프하고 같지 않으면 다음 5 번째 행으로 옮깁니다. 여기에서 EAX와 EBX는 같지 않으므로, 다음 명령 【MOV EAX,1】가 실행됩니다. 이 결과 EAX는 1이 됩니다. 다음 6 번째 줄 【CMP EAX,EBX】 명령에서 다시 한번 EAX와 EBX를 비교합니다.

여기에서 EAX와 EBX는 다르기 때문에 7 번째 줄 00401015h의 명령 【JNZ SHORT jump.00401019】라는 조건 점프 명령으로 00401019h로 점프하게 됩니다. 즉, 00401017h의 【MOV EAX,EBX】는 실행되지 않고, 점프한 곳의 어드레스인 00401019h의 【ADD EAX,EAX】가 다음에 실행되게 됩니다.

기본적으로 조건 점프 자체는 단순합니다. 처음에는 조건 점프 명령어 표를 참고 하면서 익숙해 지면 될 것입니다(위에서 기술한 표는 어셈블리 언어로 프로그래밍 할 때에 사용하는 명령이지만, 기계어 내부에서는 동일한 것으로 취급되는 것도 있으므로 디스어셈블리나 디버거로 읽어 들였을 때에는 눈에 띄지 않는 것도 있습니다.).

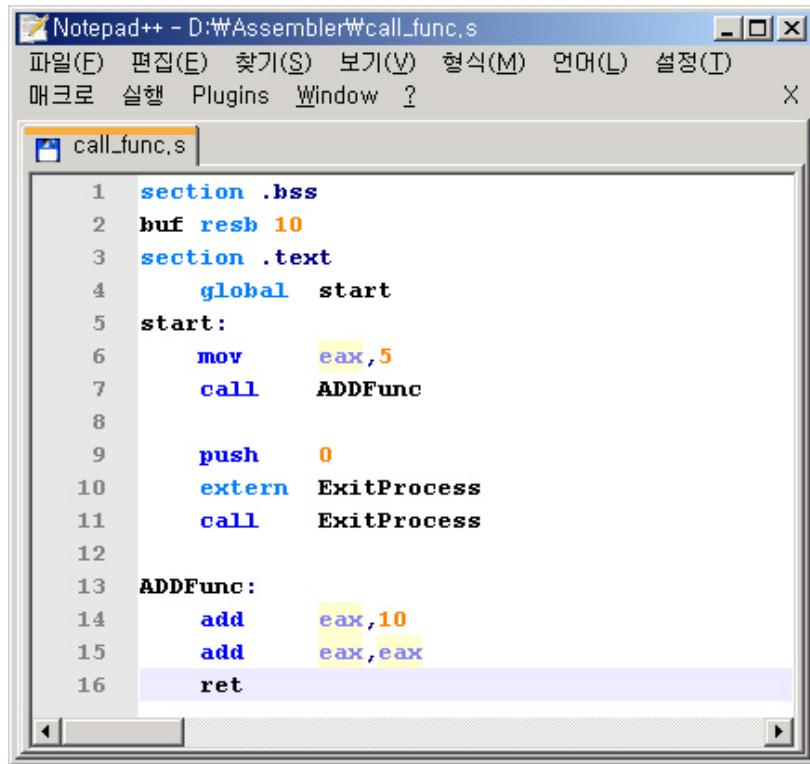
○ 서브루틴의 호출

서브루틴은 프로그래밍 중에서 몇 개의 명령군을 정리한 것입니다. 예를 들어, “어떤 수에 10을 더하고 2배 처리한다”라는 명령이 여러 번 있다고 한다면, 서브루틴이라는 형태로 만들어 프로그램의 여러 곳으로부터 호출할 수 있도록 할 수 있습니다. C언어의 서브 함수와 동일하다고 보면 됩니다.

어셈블리에서 서브루틴의 호출에는 CALL 명령을 사용합니다. 서브루틴에서 빠져 나올 때에는 RET 명령을 사용합니다.

그러면 서브루틴 명령 사용에 대하여 알아 보겠습니다. 텍스트 편집기를 이용하여 아래와 같이 작성한 후 call_func.s라는 파일로 저장합니다.

크랙에 필요한 어셈블리어 기초



The screenshot shows a Notepad++ window with the file 'call_func.s' open. The assembly code is as follows:

```
1 section .bss
2 buf resb 10
3 section .text
4 global start
5 start:
6     mov    eax,5
7     call   ADDFunc
8
9     push   0
10    extern ExitProcess
11    call   ExitProcess
12
13 ADDFunc:
14    add    eax,10
15    add    eax,eax
16    ret
```

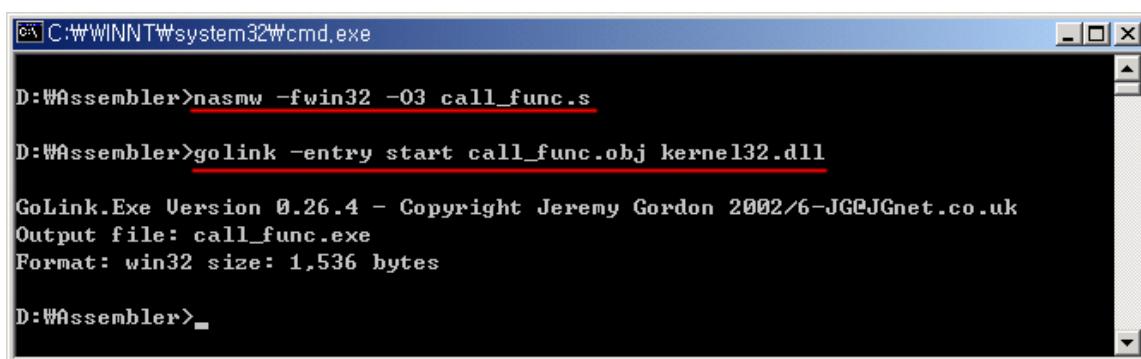
그림 30. call_func.s 어셈블리 파일

작성이 완료되면 저장된 call_func.s 파일을 오브젝트 파일로 변환하여 주기 위해 명령어 창에서 아래와 같이 실행합니다.

```
D:\Assembler>nasmw.exe -fwin32 -O3 call_func.s ➔ -O(Optimize branch offsets)
```

오브젝트 파일(jump.obj) 생성이 완료되면 Windows 라이브러리와 연결하여 실행파일로 만들어 줍니다.

```
D:\Assembler>golink.exe -entry start call_func.obj kernel32.dll
```



The screenshot shows a command prompt window with the following output:

```
C:\WINNT\system32\cmd.exe
D:\Assembler>nasmw -fwin32 -O3 call_func.s
D:\Assembler>golink -entry start call_func.obj kernel32.dll
GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk
Output file: call_func.exe
Format: win32 size: 1,536 bytes
D:\Assembler>
```

그림 31. 실행파일 만들기

만들어진 call_func.exe 를 OllyDbg 로 읽어 들이면 아래와 같은 명령을 볼 수 있습니다.

크랙에 필요한 어셈블리어 기초

그림 32. call_func.exe 디스어셈블리 덤프

첫 번째 시작 명령인 【MOV EAX,5】에서 EAX에 5를 대입하고, 다음 CALL 명령에서 004010011h 부터 시작하는 서브 루틴으로 이동하게 됩니다. 이동된 5 번째 줄 【ADD EAX,0A】로 EAX 가 0Fh(05h+0Ah)가 되고, 다시 【ADD EAX,EAX】를 통해 최종 EAX 값은 01Eh가 됩니다. [RETN]에서 CALL 명령을 호출했던 다음 명령으로 이동(3 번째 줄)하여 0040100Ah로 제어가 넘어갑니다. 기본은 굉장히 단순합니다.

실제로 하나씩 Step 실행 해 가면서 동작을 확인해 봅시다. CALL 명령 호출을 Step 실행하면(F7 키) ESP 가 0012FFC0h 가 됩니다. 스택 포인터가 변화 한다는 것은 스택이 사용되고 있다는 것입니다. OllyDbg 오른쪽 밑의 스택 윈도우를 보면 스택의 가장 위에 0040100Ah 가 들어 있을 것입니다. 이것은 CALL 명령이 호출한 서브 루틴으로부터 RET 명령으로 복귀할 때에, 어디로 복귀할 것인지를 저장하고 있는 것입니다(0040100Ah 는 CALL 명령의 다음 명령).

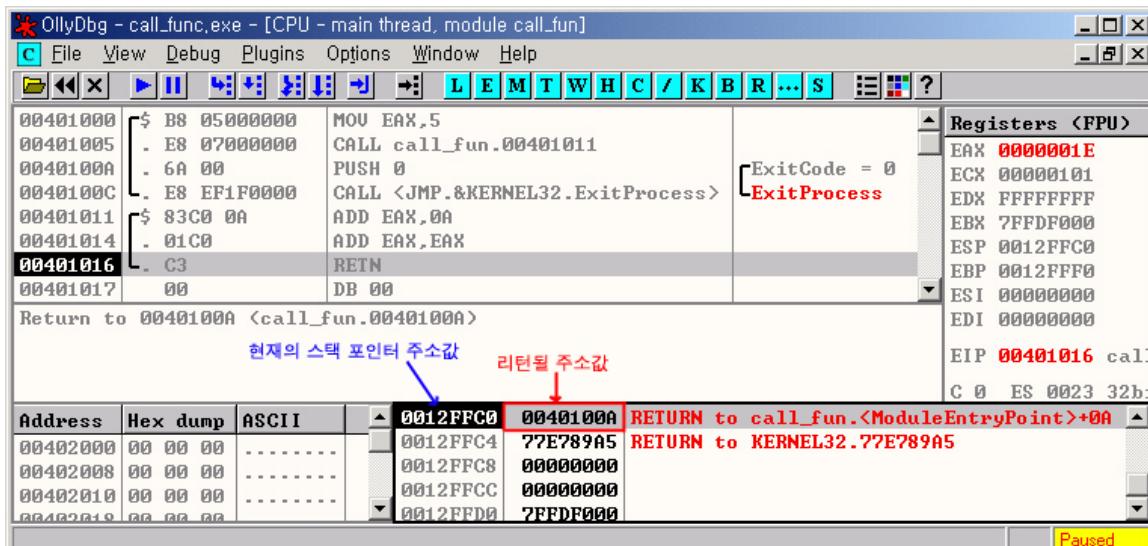


그림 33. ESP 와 RET 주소

즉, 2 번째 줄의 CALL 명령은 다르게 쓰자면 다음과 같이 할 수 있습니다.

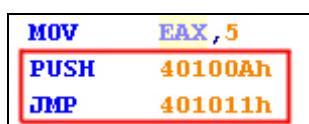


그림 33. CALL 명령의 치환

크랙에 필요한 어셈블리어 기초

마지막으로 7번째 줄 【RETN】 명령을 F7 키를 쳐서 진행해 봅시다. ESP는 원 상태인 0012FFC4h으로 변경되고, 프로그램 제어도 00401100Ah로 건넵니다.

이처럼 조건 점프 명령은 입력된 시리얼 키가 맞는지, 프로그램을 보호하기 위한 부분들이 제대로 실행되었는지 등을 체크하기 위해 쓰여지고 있습니다. 크랙과정에서 가장 중요한 명령군이라고 보아도 무방합니다.

○ NOP(NO oPeration)명령

NOP 명령은 아무 동작도 하지 않는 명령이며 CPU 내부 상태에 어떤 영향도 주지 않습니다. NOP 명령은 다양한 곳에서 사용되며, 응용 부분이 많기 때문에 반드시 알아 두어야 합니다. 예를 들면, 분기 명령이나 산술연산 명령을 수행하지 않도록 하기 위해 사용되기도 합니다. NOP 명령 사용 예제는 실전 크랙과정에서 소개할 예정입니다.

마치며

여기까지 설명한 내용이 최소한 알아 두어야 할 어셈블리 명령입니다. 또한 간단히 설명하기 위해 무리하게 이상한 표현을 쓰고 있는 부분도 있습니다. 그러나 처음부터 완벽하게 어셈블리 언어를 이해하여 크래킹을 배우기보다는 실천을 통하여 배우는 쪽이 월등히 효율적이며 즐거울 것입니다.

다소 모르는 것이 있어도 자꾸 다음 페이지로 넘어가면서 실제 리버싱/크랙으로부터 지식을 얻도록 노력하십시오. 이러한 반복과정을 통해 자신이 부족한 부분과 좀더 연구가 필요한 부분을 깨달을 것이며, 다른 참고서적이나 좋은 글들을 찾게 될 것입니다.

제 3 장 Windows 크랙 기본편

3.3 소프트웨어 크랙 기초(Software Crack Basic)

들어가며

쉐어웨어(Shareware)나 패키지 소프트웨어(Package software)의 대부분은 소프트웨어 개발 비용을 회수하거나 무단 복제를 금지하기 위해 시리얼 넘버에 의한 보호 방법을 채택하고 있습니다. 또한 요즘 같이 온라인 게임에서는 해킹 방지를 위해 게임보안 솔루션을 도입하여 게임을 보호하고 있습니다. 더 나아가서는 하드웨어 보호장치를 통한 소프트웨어 보호를 시도하고 있습니다. 그러나 그 보호 방법 대다수는 굉장히 취약합니다. 개인의 스킬에 따라 달라지기도 하지만 수분에서 수시간만에 크랙이 가능합니다. 현재도 Microsoft 를 비롯한 많은 기업에서 자사 소프트웨어를 대상으로 공개적인 크랙을 해달라고 요청하기도 합니다. 국내에서도 개발사가 크래커에 대하여 공개 크랙을 제시한 예가 몇 있었지만, 그러한 공개 도전에 비해서 Protection 레벨이 결코 높지 않았습니다. 단순히 자사 소프트웨어를 홍보하기 위한 수단들이 대부분 이었으므로 실제 크래커도 김빠지는 일이라고 생각됩니다.

크래커로부터 자신의 소프트웨어를 보호하기 위한 길은 크래커의 크랙 기법을 아는 것이 가장 최선의 방법입니다. 그러기 위해서는 자신이 크랙을 연습해 보는 것이 제일 효과적입니다. 상용 소프트웨어에서 채택하고 있는 Protection 방법들은 실제 연습 크랙 프로그램과 많이 다르지만 소프트웨어 크랙 기초를 다지기 위한 것이라면 전혀 문제가 되지 않습니다. 차후 다뤄질 크랙 중, 고급 과정에서는 몇몇 상용 소프트웨어와 유사한 공개 프로그램을 가지고 진행할 예정입니다. 그래서 이번 기초 과정에서는 필자가 작성한 crackme 라고 불리는 소프트웨어 크랙 연습용 프로그램을 이용하여 몇 개의 전형적인 Protection 을 실제로 크랙해 보기로 하겠습니다. 프로그램 개발자라면 이러한 크랙 과정에서 더욱 견고한 Protection 으로 나아가기 위한 힌트를 얻길 바랍니다.

보호 형태	지식 기반	고정키 방식	ID/패스워드	등록된 사용자 및 패스워드 입력(Online 포함)
		교환키 방식	등록 번호	허용된 등록번호 입력(Serial Number)
소유 기반	외부 매체 방식	등록 증명서	등록 증명서	인증파일이나 인증서로 허가 (인증키를 담고 있는 파일이나 인증서 기반)
		등록 매체	하드웨어 외부 매체에 저장된 Key 로 허가 (USB Disk, Serial Lock, Mobile, OTS)	
	고정 매체 방식	등록 컴퓨터	등록 컴퓨터	등록 허용된 컴퓨터에서만 허가 (HDD Serial Number, MAC Address, IP 등)
		생체 인식	등록된 사용자의 생체 정보로 허가 (지문, 흉채, 정맥, 안면 등)	

표. 소프트웨어 보호 형태

소프트웨어 크랙 시작

본 문서에서는 첨부파일#2에 있는 [CrackMe0x01.exe]을 사용하여 설명하도록 하겠습니다. 이 crackme에 설정되어 있는 Protection은 매우 단순하기 때문에 어려운 지식을 필요로 하지 않고 쉽게 크랙이 가능합니다.

먼저 크랙 대상이 되는 CrackMe0x01.exe를 실행해 봅시다(그림 1). 시리얼 넘버의 입력을 요청하기 때문에, 에디트 박스에 적당한 시리얼 넘버를 입력한 후에 [등록] 버튼을 누르면, 로또(Lotto)에 당첨될 정도로 운이 좋은 사람이 아닌 이상 [시리얼 넘버가 틀립니다]라고 표시될 것입니다(그림 2). 이러한 구조의 시리얼 넘버 입력은 인증 protection으로서 가장 기본적인 방법입니다. 크래커의 입장에서는 힌트가 되는 단어가 몇 개나 포함되어 있어서 그 힌트를 시작으로 크랙의 단서를 찾아낼 수 있습니다.



그림 1. CrackMe 실행 화면



그림 2. 에러 메시지 화면

OllyDbg를 이용한 크래킹

소프트웨어 크랙에서 주로 이용되는 툴은 디버거입니다. 디버거는 역 어셈블러와 달리 프로그램을 한 단계씩 동작시켜 나가면서 실행 시킬 수 있기 때문에 중요한 곳에 브레이크 포인트를 설정 한다든지, 메모리나 레지스터의 내용을 조사한다든지, 제어의 흐름을 추적하는 것이 가능하다는 점에서 큰 이점이 있습니다.

여기서는 공개 디버거 중에 가장 많이 알려진 [OllyDbg]의 최종 버전 1.10을 이용합니다. 이것은 기본적으로 영문이지만, 다양한 OllyDbg Plugin이 적용되고 메뉴가 한글화된 패치도 공개되어 있습니다. 본 문서에서는 쉽게 따라 할 수 있도록 최대한 OllyDbg 자체 기능만을 가지고 설명해 나가도록 하겠습니다. 기본 자체 기능을 사용하여 디버깅하는 방법을 먼저 습득한 후 관련된 Plugin이 존재할 경우 추가적으로 설명하는 것이 디버깅의 원리와 예외 상황이 발생했을 경우 스스로 문제를 해결해 나갈 수 있는 더 낳은 방법이 될 것입니다.

소프트웨어 크랙 기초

그럼 OllyDbg 를 실행하여 메뉴로부터 [File]>[Open]를 선택하십시오. CrackMe0x01.exe 를 불러들이면 역 어셈블 코드 분석이 실시되어 그림 3 과 같은 화면이 나타납니다. 그럼 디버거부터 실행해 봅시다. 메뉴의 [Debug]→[Run]을 선택하거나 단축키 F9 키를 누르면 CrackMe0x01 이 실행됩니다.

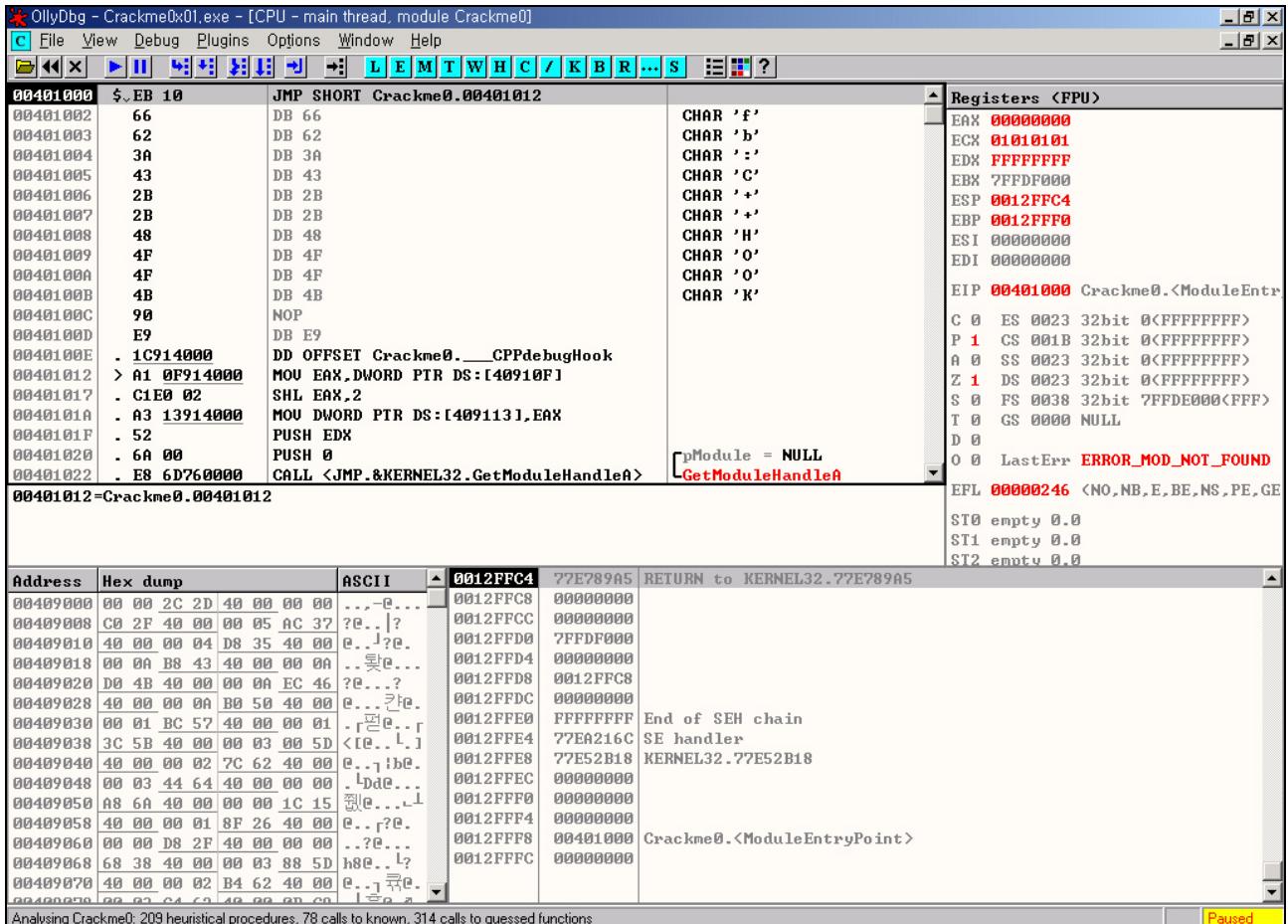


그림 3. OllyDbg 로 CrackMe0x01.exe 를 불러들인 화면

체크 루틴 찾기[Check routine search]

OllyDbg 윈도우의 왼쪽 위 메인 화면에는 CrackMe0x01.exe 의 역 어셈블 리스트가 표시되어 있는데, 출력되는 역 어셈블 리스트 전 코드를 훑어볼 필요는 없습니다. 리버스 엔지니어링이나 프로그램 분석을 하지 않는 이상 전체 중 극히 일부에 지나지 않는 시리얼 넘버 체크 루틴만 주목하면 됩니다. 그러기 위해서는 먼저 원하는 체크루틴 부분만 찾을 필요가 있습니다. 일반적으로 시리얼 넘버의 체크 흐름은 아래와 같습니다.

일반적인 시리얼 넘버의 체크 흐름

1. 사용자가 입력한 시리얼 넘버(혹은 문자열, 숫자+문자열)를 읽어 들인다
2. 내부적으로 시리얼 넘버가 맞는지 체크한다.
3. 체크 결과를 표시한다(대부분 메시지 창).

이번 크랙미는 1.번 상황으로 체크 루틴을 찾을 필요가 있습니다. 일반적인 시리얼 넘버는 에디트 박스에 숫자나 문자를 입력하는 것입니다. 그 에디트 박스에서 문자열을 수집하는 방법은 매우 한정되어 있기 때문에 크랙과정에서 큰 힌트가 됩니다.

구체적으로 말하면, 에디트 박스로부터 문자열을 수집하려면 Windows 가 제공하는 API 함수(Application Program Interface)를 이용할 필요가 있습니다. 그러나 문자열 수집 함수는 불과 3종류 밖에 되지 않습니다.

에디트 박스에서 문자열을 얻는 API 함수

[GetWindowText](#)

[GetDlgItemText](#)

[GetDlgItemInt](#)

3 번째 함수는 입력된 문자열을 수치로 변환하는 함수입니다. 예를 들어 에디트 박스에 “1234567”이라는 문자열을 입력하면, 이 문자열은 10 진수로 입력한 수치로 보고 변환을 실시합니다. 이 경우 “12D687h⁴⁵”라는 정수치가 획득됩니다. (숫자 이외의 문자열이 포함되는 경우는 에러를 발생 시킵니다)

3 개의 함수를 필자의 경험에 근거하여 사용 빈도 순으로 나열하면 아래와 같습니다.

`GetDlgItemInt < GetDlgItemText < GetWindowText`

이 3 종류 말고도 인수에 문자열(의 포인터)을 지정할 때에는 함수명 끝에 “A”나 “W”가 붙는 경우가 있습니다. “A”라면 ANSI 코드 형식, “W”라면 Unicode 형식 문자열에 대응한 함수입니다. 이 판별은 통상 컴파일러가 수행하기 때문에 개발자 대상의 API 참조에 있어서는 API 함수명에 “A”나 “W”는 붙지 않습니다.

이러한 문자열 획득 API 함수가 호출되는 순간을 파악할 수 있다면, 체크 루틴 부분을 매우 쉽게 지정할 수 있을 것입니다. 체크 루틴 설정은 디버거 기능인 [브레이크 포인트]를 이용하여 설정할 수 있습니다. 프로그램 실행 중에 임의 명령 코드에 브레이크 포인트를 설정하면, 그 명령 코드가 실행되기 직전에 프로그램이 정지(브레이크)하여, 디버거로 제어가 옮겨가는 방식입니다. 브레이크 포인트 설정 자체는 매우 간단합니다. 역어셈블리 리스트에 표시되어 있는 화면에서 브레이크 포인트를 설정하고 싶은 명령행의 2 번째 열을 더블클릭(그림 4)하는 것뿐입니다. 단축키로는 브레이크 포인트를 설정하고 싶은 라인에서 F2 키를 누르면 됩니다.

⁴⁵ “h”는 어셈블리 언어에서 이용되는 16 진수 값 임을 나타내는 기호입니다. C 언어 Java 의 경우 맨 앞에 “0x”를 붙이는 것으로 표현합니다.

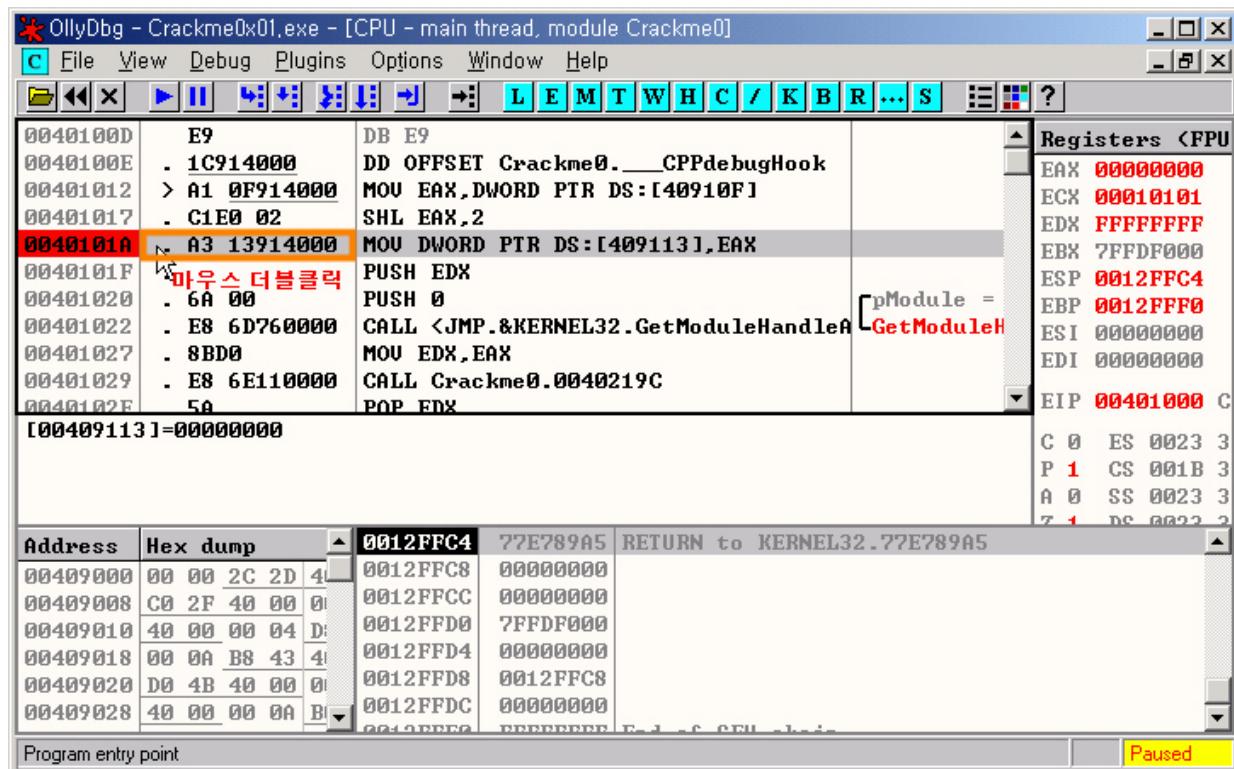


그림 4. 명령행 2 번째 열을 더블 클릭

이번에는 앞에서 기술한 3 종류의 API 함수에 대하여 브레이크 포인트를 설정하기 위해, 미리 에디트 박스에 적당한 시리얼 값을 입력해서 테스트 해봅시다(그림 5). 브레이크 포인트를 설정하고 나서 정확히 원하는 곳에 브레이크 되었는지 확인해야 하기 때문입니다.

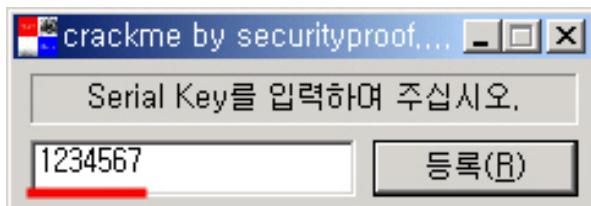


그림 5. 임의 시리얼 값 “1234567” 입력 테스트

OllyDbg는 디버깅중인 파일이 사용하고 있는 API 함수들을 확인 할 수 있습니다. 역 어셈블 원도우에서 마우스 오른쪽 클릭>[Search for]>[Name(label) in current module]으로 선택하거나(그림 6-1), Ctrl+N키를 눌러 모듈 목록을 표시할 수 있습니다(그림 6-2). 이 중에서 3 개의 API 함수가 존재하는지 찾아 봅시다. 이 원도우에 포커스를 맞춘 상태에서, 키보드에서 "getw"라고 타이핑하면 GetWindowTextA를 찾게 되므로, GetWindowTextA를 선택한 상태에서 마우스 오른쪽 클릭>[Set breakpoint on every reference]를 선택하면 GetWindowTextA(이것을 호출하고 있는 명령)⁴⁶로 브레이크 포인트가 설정됩니다. 브레이크 포인트가 설정되면 역어셈블 창에서 브레이크 포인트가 설정된 명령어 라인 주소에 붉은색으로 마킹 됩니다.

⁴⁶ API 함수에 직접 설정하고 싶은 경우는, 마우스 오른쪽 클릭>[Conditional breakpoint on import]를 선택

소프트웨어 크래치

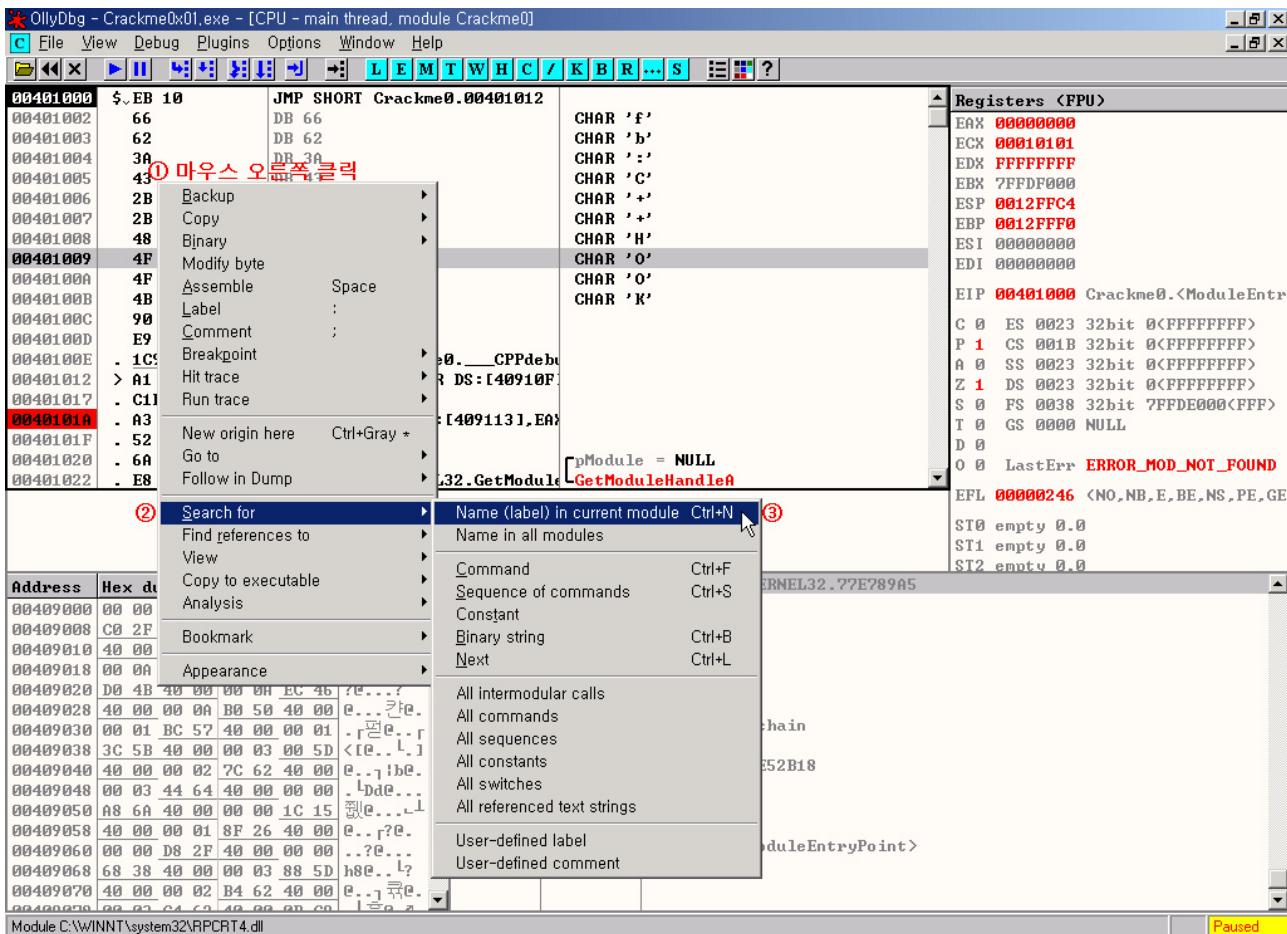


그림 6-1. 현재 외부 함수 검색

Names in Crackme0				
Address	Section	Type	<Known>	Name
0040E0D8	.idata	Import	<Known>	KERNEL32.CloseHandle
0040911C	.data	Export		_CPPdebugHook
0040E0DC	.idata	Import	<Known>	KERNEL32.CreateFileA
0040E19C	.idata	Import	<Known>	USER32.DialogBoxParamA
0040E1A0	.idata	Import	<Known>	USER32.EndDialog
0040E1A4	.idata	Import	<Known>	USER32.EnumThreadWindows
0040E0E0	.idata	Import	<Known>	KERNEL32.ExitProcess
0040E0E4	.idata	Import	<Known>	KERNEL32.GetACP
0040E0EC	.idata	Import	<Known>	KERNEL32.GetCommandLineA
0040E0E8	.idata	Import	<Known>	KERNEL32.GetCPInfo
0040E0F0	.idata	Import	<Known>	KERNEL32.GetCurrentThreadId
0040E1A8	.idata	Import	<Known>	USER32.GetDlgItem
0040E0F4	.idata	Import	<Known>	KERNEL32.GetEnvironmentStrings
00401059	.text	Export		_GetExceptDLLinfo
0040E0F8	.idata	Import	<Known>	KERNEL32.GetFileType
0040E0FC	.idata	Import	<Known>	KERNEL32.GetLastError
0040E100	.idata	Import	<Known>	KERNEL32.GetLocalTime
0040E104	.idata	Import	<Known>	KERNEL32.GetModuleFileNameA
0040E108	.idata	Import	<Known>	KERNEL32.GetModuleHandleA
0040E10C	.idata	Import	<Known>	KERNEL32.GetOEMCP

그림 6-2. 현재 외부 함수 리스트(CTRL+N)

여기서 설정된 브레이크 포인트는 메뉴의 [View]>[Breakpoint]를 선택하든지, 단축키 [Alt+B]키를 누르면 목록이 표시됩니다(그림 7). 이것으로 브레이크 포인트 설정은 끝났습니다. 그러면 브레이크 포인트 리스트

소프트웨어 크랙 기초

화면(Alt+B)에서 설정된 브레이크 명령에 마우스 오른쪽 버튼>[Follow in Disassembler]를 선택하거나 [Enter]키를 누르면, 브레이크된 어드레스 “004011C5h”로 이동합니다(그림 8). 이 시점에서는 아직 이 API 함수가 실행되지 않습니다.

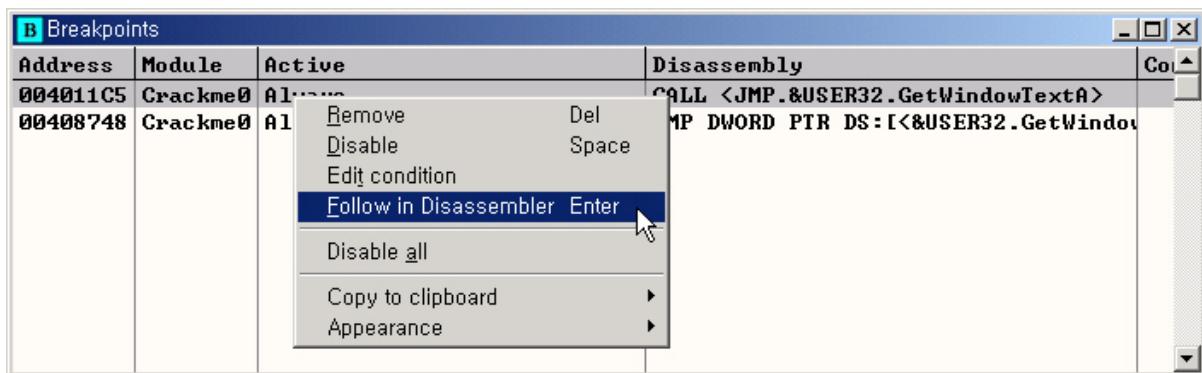


그림 7. 브레이크 포인트 리스트(ALT+B)

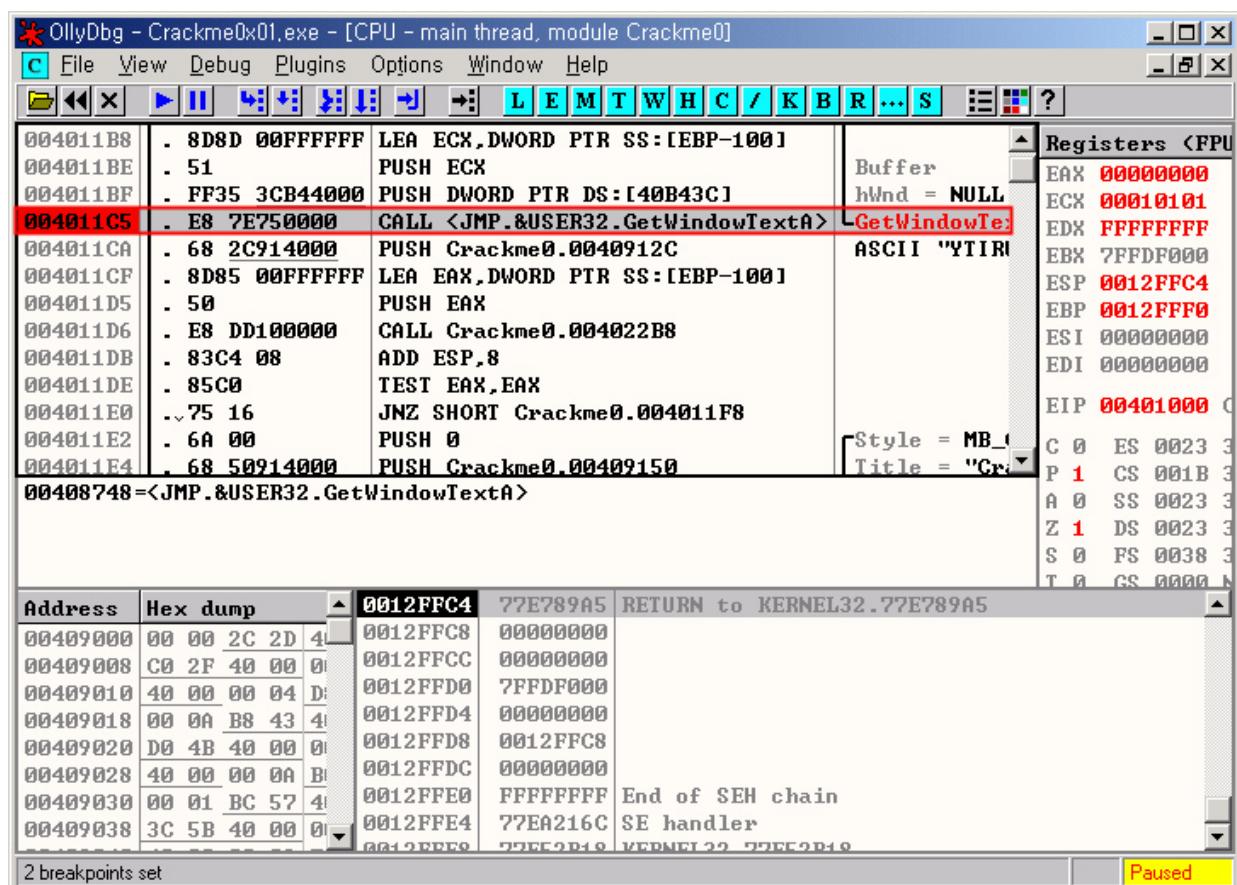


그림 8. 브레이크 포인트 설정 주소로 이동(주소 “004011C5h”)

체크 루틴 트레이스(Check routine trace)

브레이크 포인트가 설정된 앞쪽에는 체크 루틴이 존재할 것으로 한 개의 명령식 순차적으로 실행하여 자세히 조사할 시작합니다. 그림 8에서 체크 루틴과 체크 결과의 처리가 보이고 있는데, 실제로 트레이스를 시행해 봅시다.

① 브레이크한 부분에서 F9 키(Run)을 누르면 시리얼 등록창이 나타나면서 GetWindowTextA가 실행됩니다. ② 적당히 아무 값이나 입력하고(필자는 ‘1234567’입력)나서 등록 버튼을 누르면 어드레스 00408748h(CPU: 004011C5h)에서 프로그램이 브레이크 됩니다.

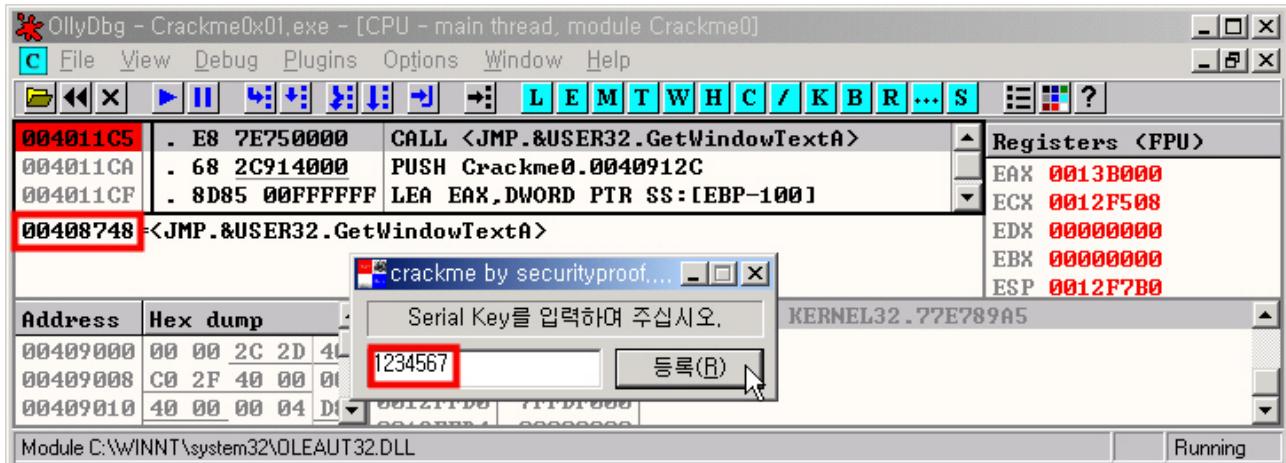


그림 9. Breakpoint 설정된 상태에서 실행(Run)

③ 여기에서 F8 키를 눌러 순차 실행(Step 실행)하면 00408748h 주소로 이동하게 되며 스택 포인트의 다음 주소는 004011CAh를 가리키고 있습니다.

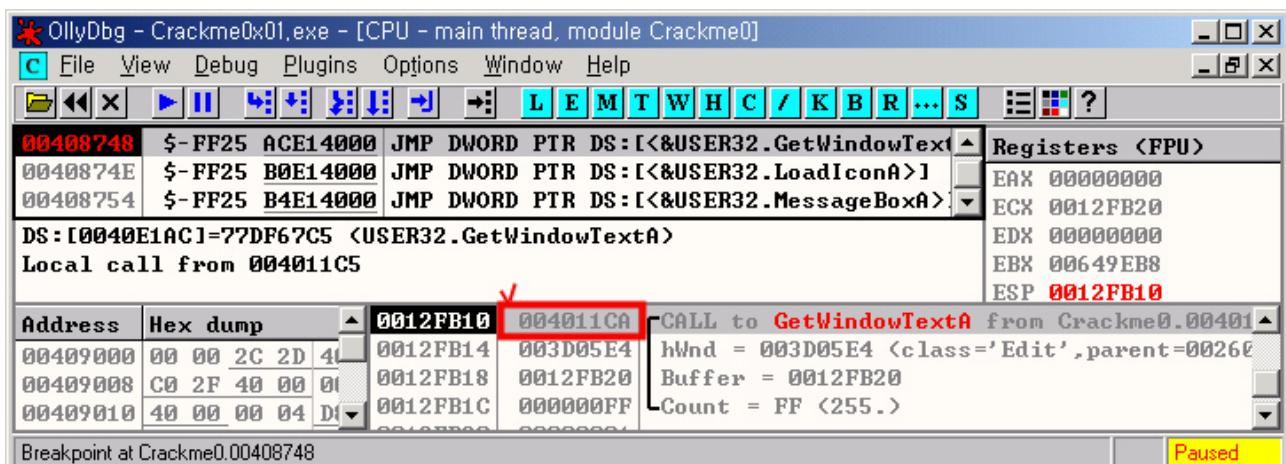


그림 10. 순차실행

④ 여기에서 다시 F9 키를 눌러 진행하면 004011CAh로 리턴 됩니다. ⑤ F8 키를 3 번 누르면 어드레스 004011D6 까지 진행됩니다. 이 CALL 명령은 GetWindowTextA 의 호출에서도 사용되고 있는 것처럼 고급언어에서

소프트웨어 크랙 기초

말하는 함수호출에 해당합니다. 인수를 포함하는 함수의 경우 호출할 때 지정되는 가인수(임시저장 값)는 스택을 경유하여 값을 주고받습니다. 따라서 인수를 포함하는 함수가 호출될 경우 CALL 명령 직전에 스택에 값을 쓰는(Push 하는) PUSH 명령이 이용됩니다. 또한 스택에 Push 된 값은 스택 어드레스 감소 방향으로 쌓여갑니다. OllyDbg 의 화면 오른쪽 밑 스택 윈도우를 봅시다(그림 11).

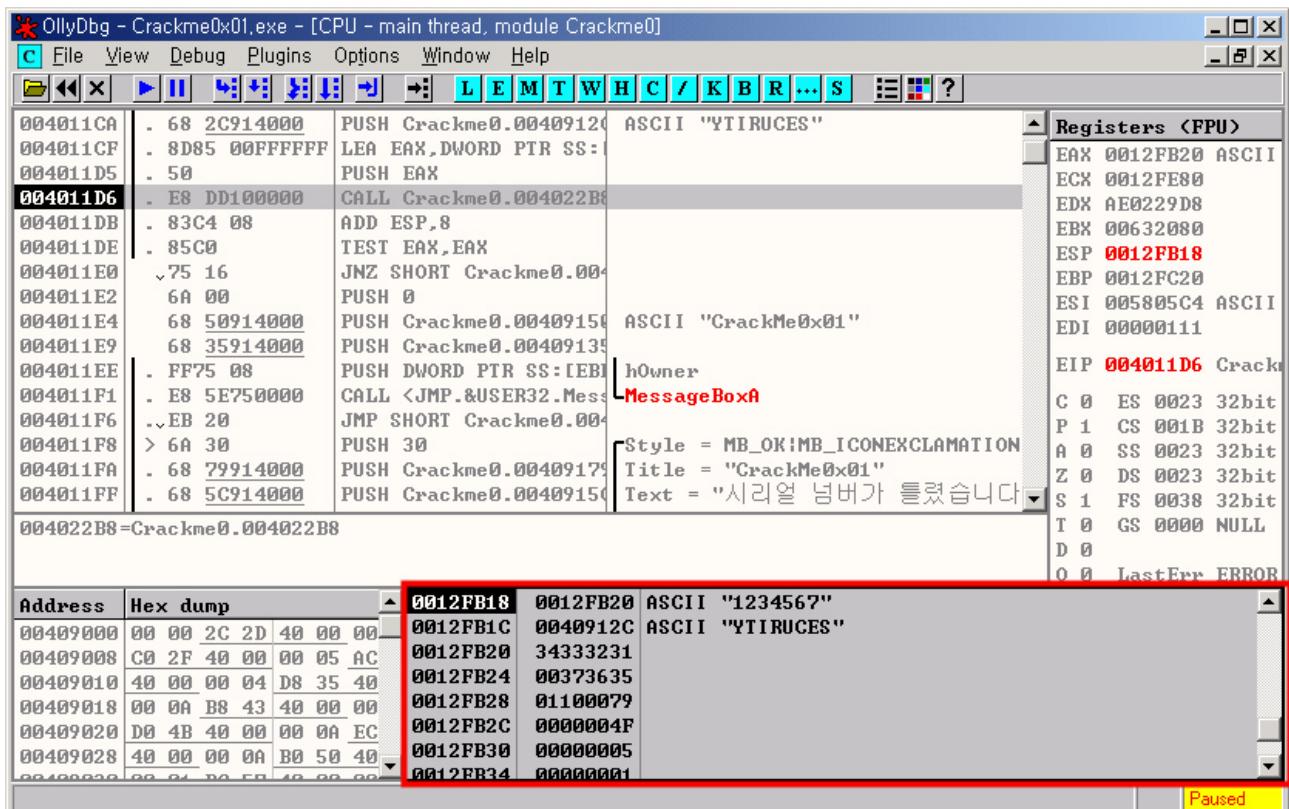


그림 11. 스택 윈도우(표시된 주소는 각자의 컴퓨터 환경에 따라 달라짐)

차후 DLL, LIB 와 같이 외부 프로그램의 함수를 이용할 경우, 함수의 구조체 형식 및 인수들을 알아내기 위해서는 스택창의 변화를 잘 파악해야 합니다.

스택에는 필자가 입력한 등록 번호 “1234567”이 보관되어 있는 메모리 주소와 문자열 “YTIRUCES”의 메모리 주소가 보관되어 있습니다. 이 2 개의 가인수로 CALL 명령 내에서 어떤 처리가 행해지고 있는 것으로, 문자열의 비교를 시행하고 있을 것이라는 것은 쉽게 추측할 수 있습니다. 이것으로부터 올바른 등록 번호는 “YTIRUCES”(“SECURITY” 역순)이라고 추측할 수 있습니다.

원래대로라면 CALL 명령으로 호출되어 있는 코드도 트레이스 해야 하지만, 처리 로직이 추측되기 때문에 CALL 부분은 트레이스 하지 않습니다. 여기에서 F8 키를 누르면 CALL 내부 명령이 실행됩니다. 함수의 return value 는 통상 EAX 레지스터에 보존되지만, 레지스터 윈도우를 보면 EAX 레지스터 값은 0xFFFFFD8 입니다. 이 값은 문자열 비교결과로 틀림 없습니다.

이 비교결과로 분기처리가 수행됩니다. 그러면 메시지 표시 처리 부분을 트레이스해 봅시다.

소프트웨어 크랙 기초

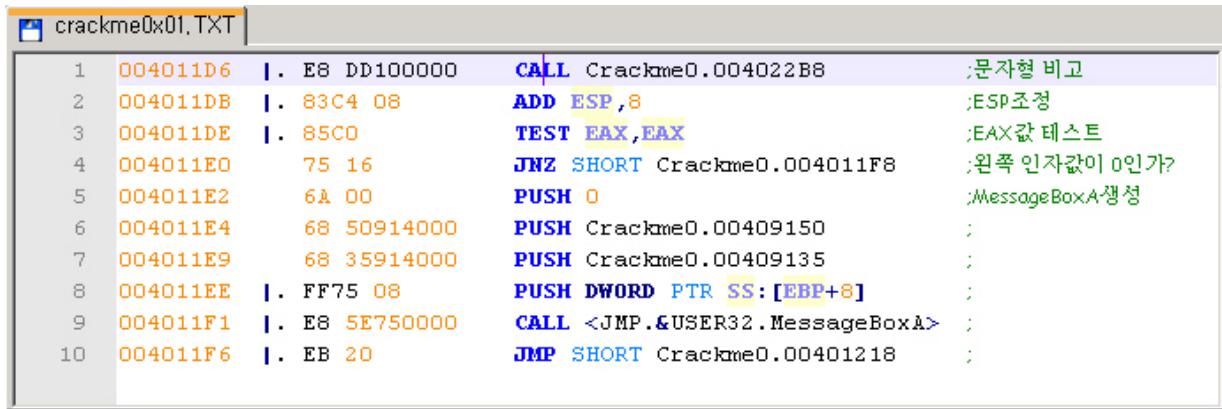


그림 12. 메시지 트레이스 처리

CALL 명령 직후의 ADD 명령은 ESP 레지스터(스택 포인터)를 Clear 합니다. 함수 인자는 2 개(사용자 입력 값, 원래 키 값)이므로 8 바이트가 됩니다. 여기서는 이 명령에 크게 신경 쓸 필요는 없습니다. 다음 TEST,JNZ(또는 JZ) 명령은 기본 명령으로 대상이 되는 레지스터가 제로인가 제로 이외인가를 조사하여 분기를 행할지 아닐지를 결정하는 것입니다. 즉, Jnz 는 왼쪽 인자 값이 0 이 아니라면 주소 004011F8h 로 점프하라는 것입니다. 최초의 TEST 명령에서는 양 연산수의 AND 연산 결과를 플래그에만 반영시킵니다. 왜냐하면 함수의 리턴 값이 EAX 레지스터에 저장되어 있고, 그 값이 0인지(올바른 키 값이 들어 왔는지) 확인하는 것이기 때문입니다.

여기서는 EAX 레지스터 값이 제로가 아니기 때문에 제로플래그는 어드레스 004011F8 로 점프합니다. EAX 레지스터 값이 제로라면 점프 하지 않고 등록성공 메시지 박스가 표시됩니다(그림 13).

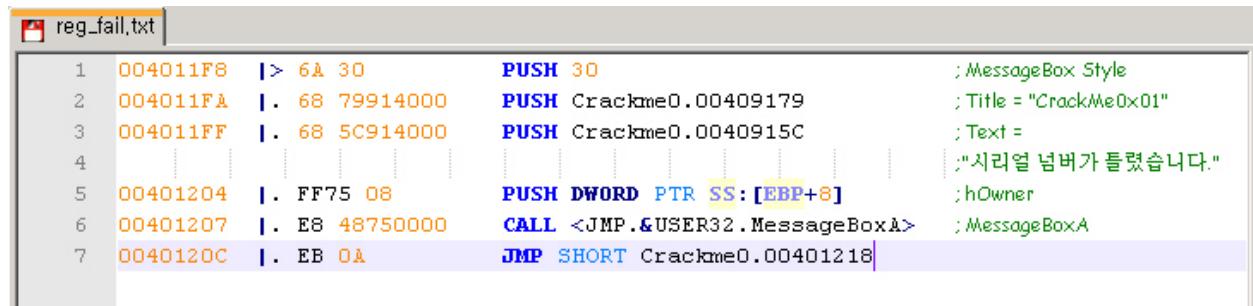


그림 13. 등록 실패 메시지 처리

올바른 등록 시리얼 넘버인 “YTIRUCES”를 입력하게 되면 등록 성공 메시지를 볼 수 있게 됩니다. 처리 부분은 아래와 같습니다.

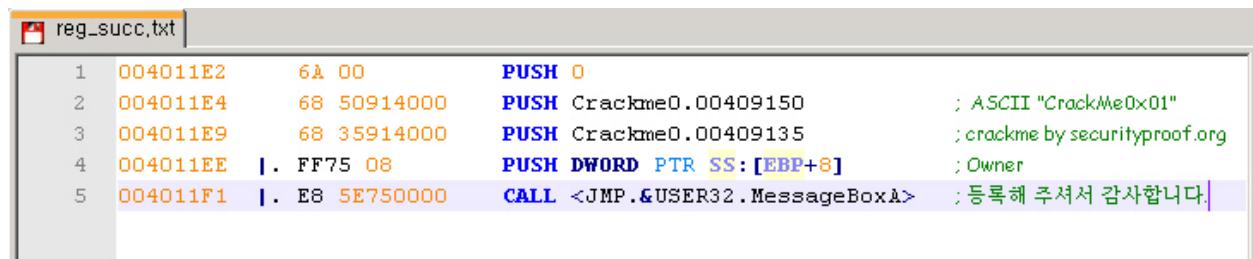


그림 14. 등록 성공 메시지 처리

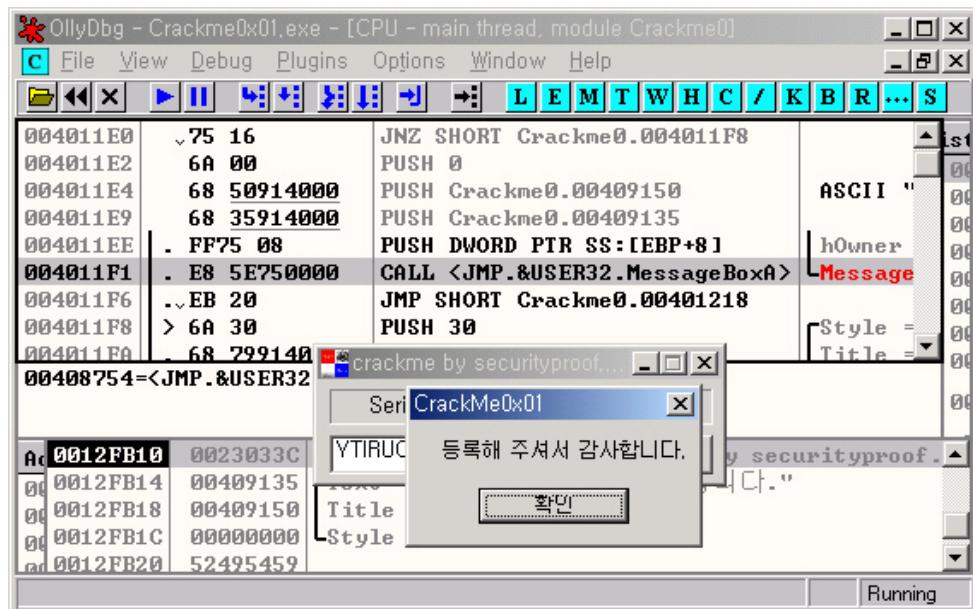


그림 15. 최종 등록 성공 화면

마지며...

이번 장에서는 크랙단서로 에디트 박스로부터 문자열을 수집하는 API 함수에 대해 알아 보았습니다. Import 된 API 함수를 자세히 살펴보는 것은 크랙 수행에 매우 중요합니다. Protection 기술 향상을 원한다면 API 함수를 일체 사용하지 않는 것이 바람직합니다. 그러나 Windows라는 거대하고 복잡한 OS 상에서 동작하는 어플리케이션의 경우라면 API 함수 사용은 불가피하기 때문에 Protection에 관계된 하나의 한계라고 말할 수 있을 것 같습니다.

또한 이번에 예로 든 Protection은 정말 간단한 것이었지만 쉐어웨어의 10%정도가 채용하고 있는 것입니다. 이 쪽의 Protection의 경우 디버거를 사용할 것도 없이 바이너리 에디터만으로도 크랙할 수 있는 것이 대부분입니다. CrackMe0x01.exe를 바이너리 에디터로 열어서 여러 메시지 [시리얼 넘버가 틀렸습니다.]를 문자열 검색하면, 오프셋 어드레스 00007F5C에서 찾게 되지만, 이 부근을 잘 보면 올바른 등록 번호가 보입니다.

이와 같은 Protection 예제는 첨부파일의 CrackMe0x01-01.exe, CrackMe0x01-02.exe, CrackMe0x01-03.exe 정도만 더 실습해 본 후 몇몇 쉐어웨어를 직접 크랙해 보는 것으로 충분할 것입니다.

예제 크랙미를 모두 풀어 보면 알겠지만, 프로그램의 초기화된 변수 값이나 사전 정의된 모든 값들은 PE 파일 섹션내의 데이터 영역(.data, .rdata, .bss 등)에 저장하고 있다는 것입니다. 그러므로 손쉽게 텍스트 편집기(노트패드 등)에서도 등록 키 값을 볼 수 있는 것입니다.

이번 Crackme0x01.exe 프로그램에서 올바른 크랙키를 얻는 것은 위에서 제시한 방법 이외에도 무수히 많이 있습니다. 바이너리 스트링 검색 프로그램을 이용하여(OllyDbg 스트링 검색, Bintext 등) 원하는 등록 번호를 얻을 수도 있고, 아예 등록 루틴을 임의적으로 수정하여 무조건 통과 시키거나 운영체제의 특정 메모리 값을 치환 시켜 등록 루틴을 우회 할 수도 있습니다. 더 나아가 원래 소스코드를 복원하여 동일한 카피본을 생성할 수도 있을 것입니다. 본 문서에서 이러한 방법들을 차근차근 하나씩 배워나갈 것입니다.

소프트웨어 크랙 기초

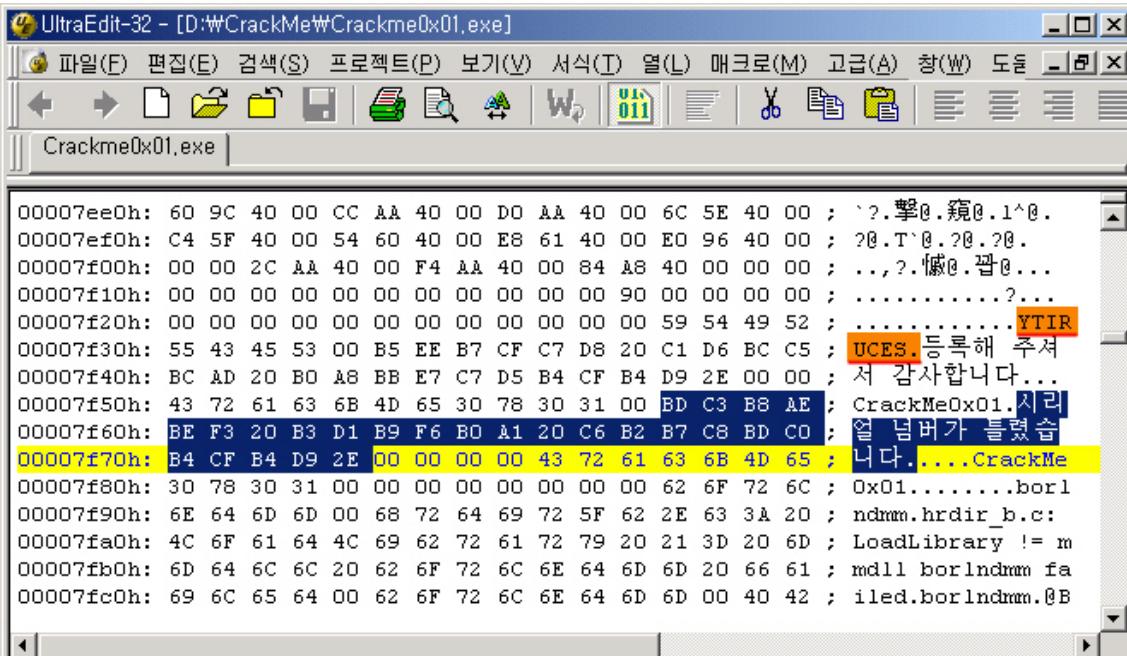


그림 16. 바이너리 에디터로 등록번호 찾기

Address	Disassembly	Text string
00401000	jmp short C <Initial CPU selection>	
004011CA	push Crackm	ASCII "YTIRUCES"
004011E4	push Crackm	ASCII "CrackMe0x01"
004011E9	push Crackm	ASCII "등록해 주셔서 감사합니다."
004011FA	push Crackm	ASCII "CrackMe0x01"
004011FF	push Crackm	ASCII "시리얼 넘버가 틀렸습니다."
004012C0	mov dword p	ASCII "MZP"
004013A8	push Crackm	ASCII "borlndmm"
004013BA	push Crackm	ASCII "hrmdir_b.c: LoadLibrary != mmdll borlndmm failed"
004013E6	push Crackm	ASCII "borlndmm"
004013FA	push Crackm	ASCII "@Borlndmm@SysGetMem\$qqpri"
00401407	push Crackm	ASCII "@Borlndmm@SysFreeMem\$qqrpv"
00401414	push Crackm	ASCII "@Borlndmm@SysReallocMem\$qqrpvi"
0040254F	push Crackm	ASCII "xxtype.cpp"
00402554	push Crackm	ASCII "VIC_STPHC/bacon-NtMain()

그림 17. OllyDbg 문자열 검색

키 검색 이외에 어셈블리 편집을 통해 올바른 키 값이 아니더라도 등록을 우회할 수도 있습니다. 아래는 등록 키 값을 비교하는 구문을 무조건 점프 시키는 JMP 구문으로 변경한 상태입니다. 이러한 경우 원본 바이너리 파일의 변경이 필요합니다. 혹은 **【TEST EAX, EAX】** 구문을 **【XOR EAX,EAX】**로 변경하고 올바른 키 값을 입력하지 않으면 될 것입니다. 그 외에도 수십 가지의 등록 루틴을 우회할 수 있는 방법들이 존재하므로 여기서 일일이 설명하지는 않겠습니다.

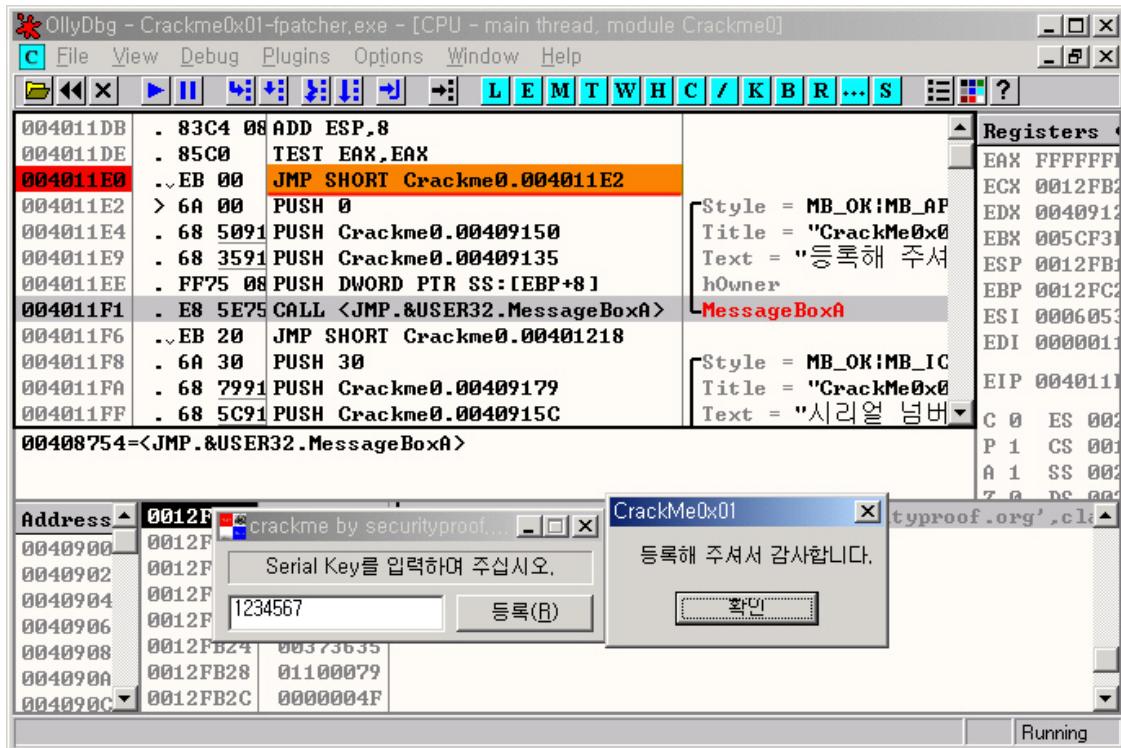


그림 17. 비교구문 치환(JNZ → JMP)

아래는 파일이 실행될 때 특정 메모리 값을 임의적으로 변경 시켜 등록 루틴을 우회하는 프로그램입니다(inline patch). 원본 바이너리 파일을 직접 변경하는 것이 아니라 운영체제의 메모리를 변경하는 형태입니다.

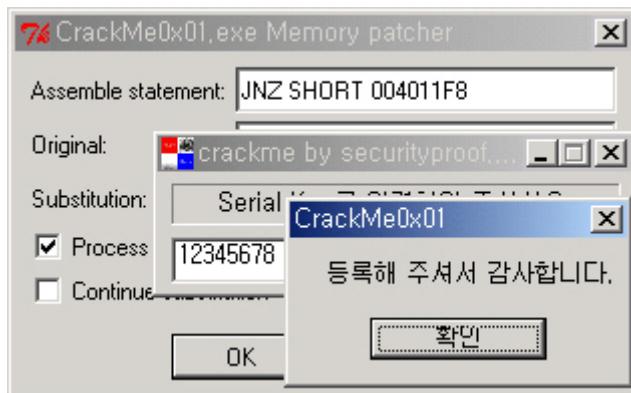


그림 18. Memory patcher

제 3 장 Windows 크랙 기본편

3.4 OllyDbg Plugin 소개(OllyDbg plug-in introduce)

들어가며

OllyDbg는 공개된 상태 만으로도 대단히 사용하기 쉽고 우수한 디버거로 알려져 있습니다. 그러나 추가적으로 플러그인⁴⁷을 통해 다양한 디버깅 작업을 손쉽게 수행할 수 있으며, 빠른 코드 분석을 가능 하도록 만들어 줍니다. 본격적으로 크랙 및 리버싱을 하기 전에 OllyDbg 플러그인 중에서 잘 알려진 것을 소개하고 간단한 사용방법을 알아 보도록 하겠습니다.

plug-in 설정(Plug-in setup)

OllyDbg 플러그인은 수십여 종류가 존재합니다. 한 달에 몇 개씩 새로 만들어 지고 업데이트 되고 있기 때문에 본인이 주기적으로 리버싱 포럼을 모니터링 하고 사용해 보는 방법 밖에는 없습니다. 또한 공개되지 않은 양질의 플러그인까지 포함하면 너무 많기 때문에 이 문서에서는 주로 이용되는 몇몇 플러그인을 설명하도록 하겠습니다.

OllyDbg 는 3-1 장 “소프트웨어 크랙에 이용되는 툴”的 “디버거(Debugger)”부분에서 앞서 설명하였습니다. 이미 다운로드 받아 한번쯤은 실행해 보았을 것입니다(3-2 장, 3-3 장을 눈으로만 읽었다면 지금 본인의 컴퓨터에 설치해 주시기 바랍니다). 처음 다운로드 받게 되면 9 개의 파일로 이루어져 있으며, “OLLYDBG.EXE” 최초 실행 시 “ollydbg.ini” 파일이 생성 됩니다. 생성된 “ollydbg.ini”파일은 OllyDbg 의 다양한 환경 설정 내용을 수정/추가 할 수 있습니다. 앞으로 많은 플러그인을 다운로드 받아 사용해야 하기 때문에 별도 디렉터리를 생성하여 관리하는 것이 좋습니다.

플러그인 관리를 위해 OllyDbg 루트에 “plugins”라는 디렉터리를 새로 생성 시킨 후 “ollydbg.ini”파일의 [History] 섹션 부분을 아래 그림과 같이 수정 합니다. 플러그인 디렉터리 설정이 완료되면 OllyDbg 를 실행시킬 때마다 플러그인들을 자동으로 불러오게 됩니다. 플러그인의 로딩 성공/실패는 OllyDbg 메뉴 → View → Log(단축키 ALT+L)를 통해 확인할 수 있습니다.

⁴⁷ OllyDbg Plug-in: <http://woodmann.net/ollystuph/index.php>

OllyDbg 플러그인 소개

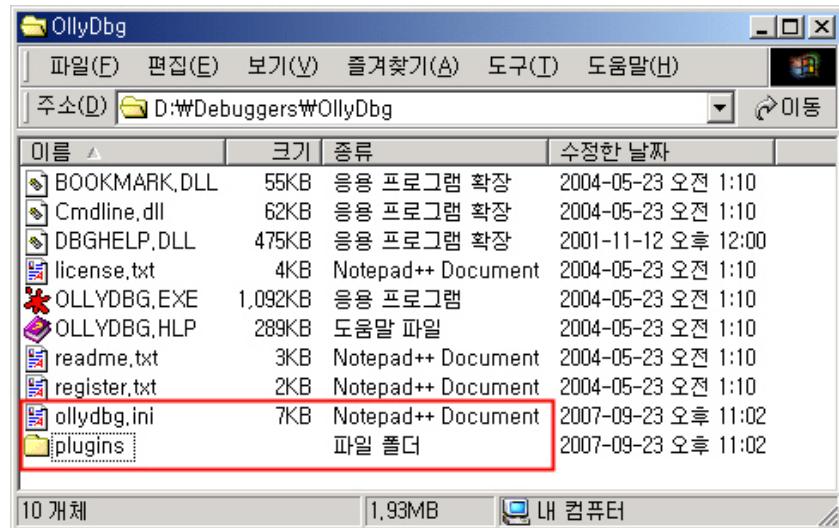


그림 1. 플러그인 디렉터리 생성

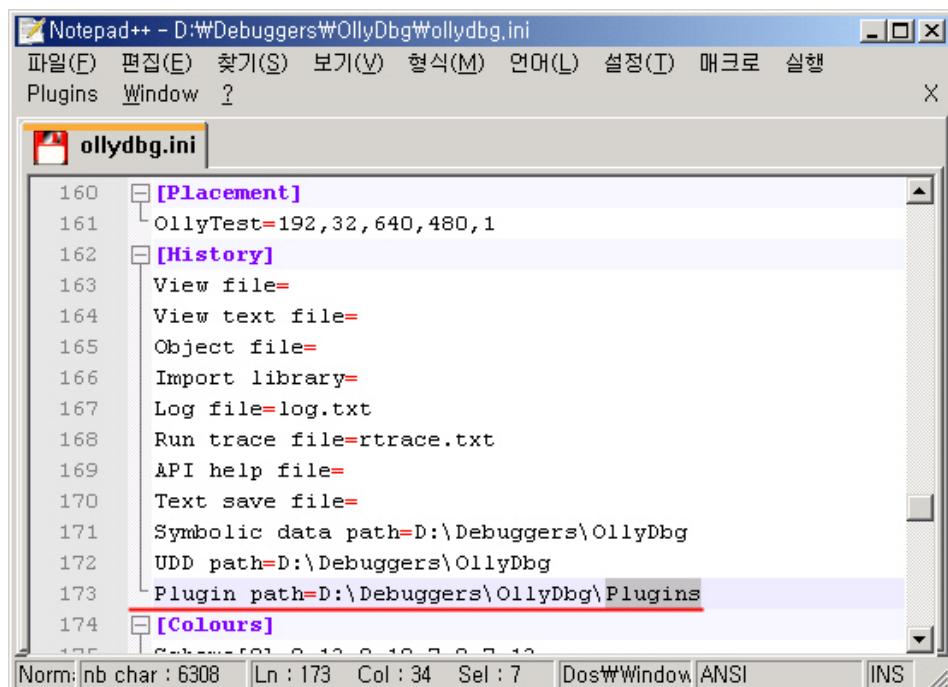


그림 2. 플러그인 디렉터리 설정

혹은 OllyDbg 메뉴를 통해 플러그인 디렉터리를 등록할 수 있습니다. OllyDbg 를 실행 시킨 후 “메뉴의 Options → Appearance → Directories” 템에서 플러그인 디렉터리를 찾아주면 됩니다.

OllyDbg 플러그인 소개

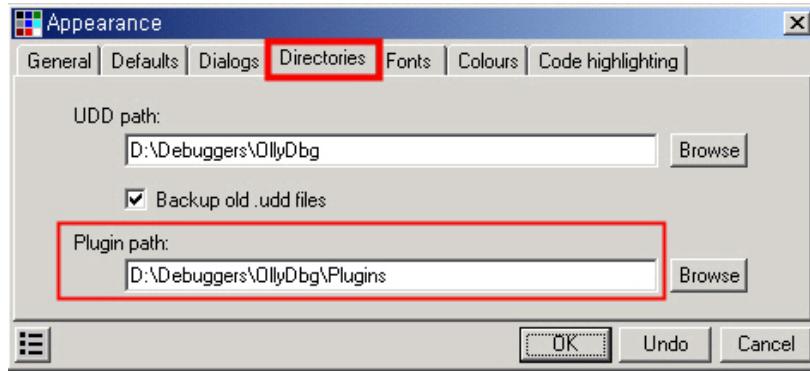


그림. 플러그인 디렉터리 등록

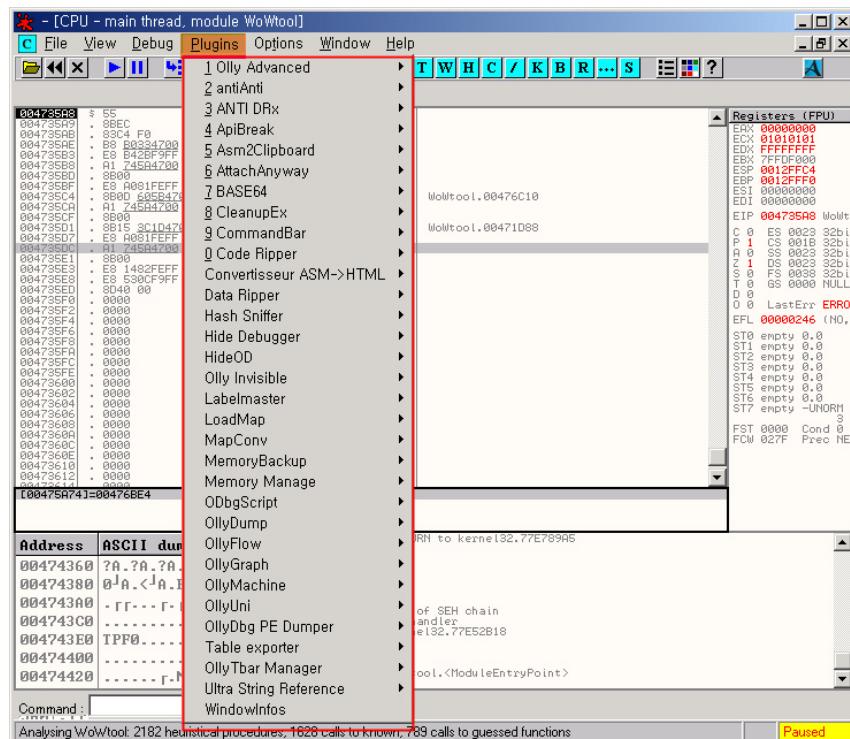


그림. OllyDbg 플러그인 메뉴

OllyDbg 플러그인을 수집해 놓은 대표적인 사이트는 아래와 같습니다. 링크가 많이 깨지고 웹 서버가 자주 다운되기 때문에 외국의 유명한 리버싱 포럼을 통해 주기적으로 업데이트 정보를 수집하시면 됩니다.

OllyDbg Plug-in download: http://www.openrce.org/downloads/browse/OllyDbg_Plugins

OllyDbg Plug-in download: <http://tuts4you.com/ollyplugin/index.php>

OllyDbg Plug-in download: <http://woodmann.net/ollystuph/index.php>

OllyDbg Plug-in download: <http://www.pedyi.com/tools/Debuggers/ollydbg/plugin.htm>

Command Bar plug-in

OllyDbg 플러그인 소개

OllyDbg v.1.08 의 Command line 플러그인을 개선 한 것으로, OllyDbg 메인 윈도우 하단에 배치하여 빠른 명령어 입력이 가능합니다. SoftICE 를 모방하여 하단에 작은 Bar 형태로 만들었습니다. 명령어 입력시에 자동 완성 기능과 명령어가 완성되면 인수를 표시하는 입력 보조기능을 지원하고 있습니다. 또한 추가적으로 여러 명령어를 한번에 실행할 수 있는 매크로 기능도 지원합니다. 소스코드가 공개되어 있기 때문에 다양한 버전들이 존재합니다.

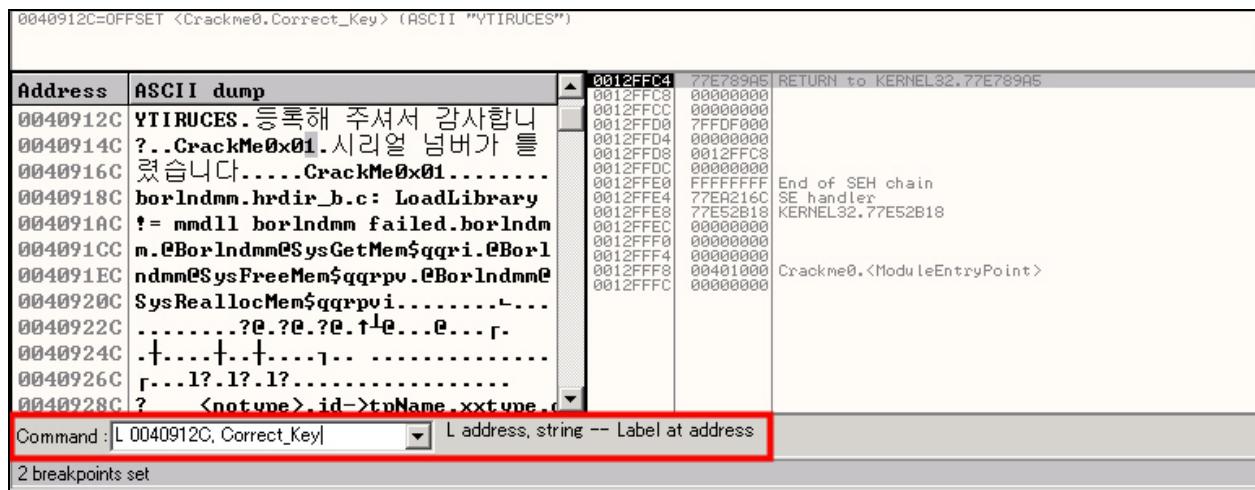


그림. Command Bar 플러그인에 의한 명령어 입력

※ 관련 파일: CmdBar.dll, CmdBar.ini

OllyDump plug-in

디버거(Debuggee) 프로세스 메모리를 덤프하여 파일로 만들어 주는 플러그인입니다. 소스파일도 공개되어 있으므로 원하는 형태로 재 컴파일이 가능합니다. 자체적으로 IAT(Import Address Table)를 Rebuild 기능이 있지만, 혹시 Import 함수들이 Rebuild 되지 않는다면 다른 Rebuild 툴을 이용하여 완전한 PE 파일로 만들 수 있습니다. IAT 는 윈도우 API 가 Call 되는 주소의 테이블이 담긴 곳으로, 실행파일 보호기(Protector)나 패커(Packer)로 바이너리를 조작하였을 경우 IAT 영역이 깨지게 됩니다. 따라서 IAT 영역은 프로텍션 전 후로 PE 헤더 정보가 달라지게 됩니다. OllyDbg 로 디버깅을 실행하면 메모리에 로드된 이후 코드 영역은 프로텍션 전과 동일한 값을 얻을 수 있지만 PE 헤더 영역은 복구 되지 않게 됩니다. 이러한 IAT 테이블을 복구해주는 것을 IAT Rebuild 라고 부르며 OllyDump 플러그인에서는 2 가지 방법으로 복구할 수 있습니다. 그러나 OllyDump 에서 IAT Rebuild 가능한 패커의 종류가 많지 않아 다른 툴(ImportREC, YodaPE)을 주로 이용합니다.

OllyDbg 플러그인 소개

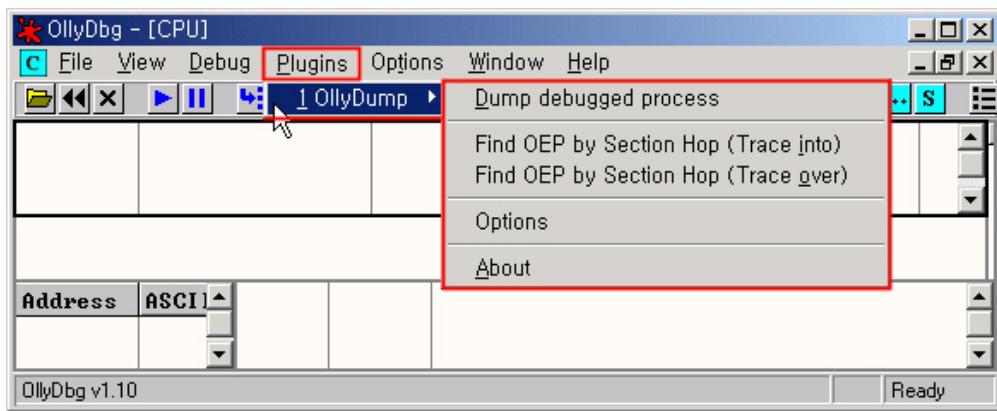


그림. OllyDump 플러그인

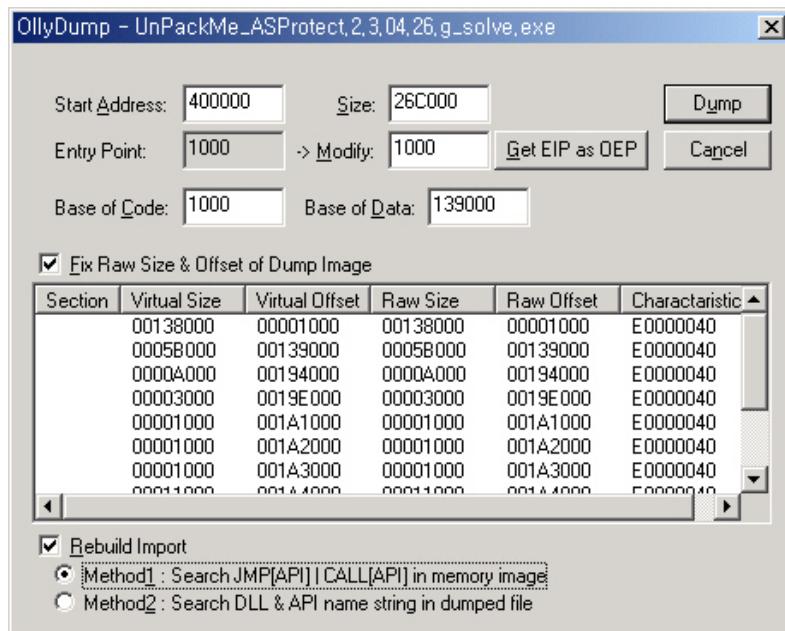


그림. OllyDump 의 메인 화면

간단한 리빌드(Rebuild) 기능과 몇몇 패커(Packer)의 IAT(Import Address Table) 트레이스도 지원하고 있습니다.

지원 가능한 패커로는 ASProtect 1.2x, PELOCK1.06, ASPack 2.12, UPX 1.24, PECompact 1.84, Petite 2.2, tElock 0.96/0.98bl, PeX0.99, WWPack 1.20, NeoLite 2.0, PE Diminisher 0.1, PEPack 1.0, PKLITE32 1.1 등이 있습니다.

OllyDump v3.00.110 이하 버전에서는 몇몇 버그가 존재하므로 최신 버전으로 다운로드 받아 사용하기 바랍니다.

※ 관련파일: OllyDump.dll, OllyDump.ini

OllyDbg PE Dumper

Gigapede 가 작성한 OllyDump 를 완전히 새로 작성한 플러그인으로 OllyDump 와 마찬가지로 프로세스의 메모리를 덤프 하는 플러그인 입니다. Anti-Dump, Anti-Protection, Anti-Debug 상태에서도 전체 프로세스를 대상으로 하여 덤프할 수 있습니다. 그러나 현재 존재하는 메모리 페이지들을 기반으로 저장하기 때문에, 메모리 영역에 존재하지

OllyDbg 플러그인 소개

않거나 할당되지 않은 공간이 존재한다면 다른 덤프 툴을 이용하여 다시 리빌드 해야만 합니다(OllyDump 도 동일).

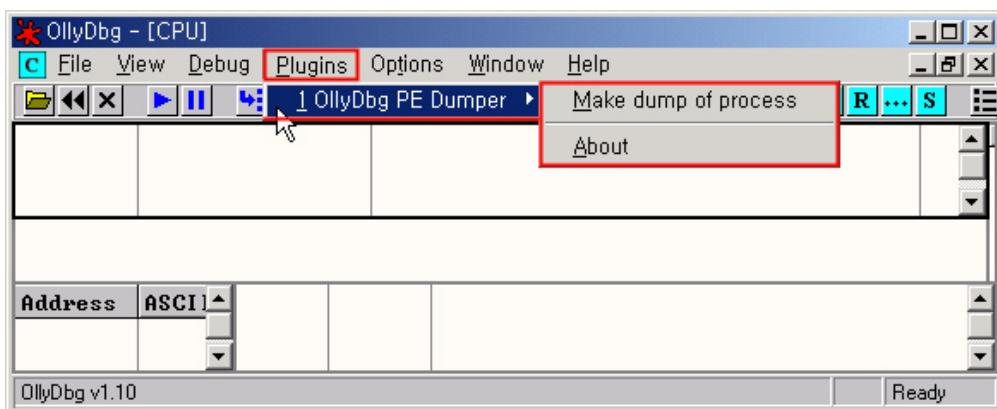
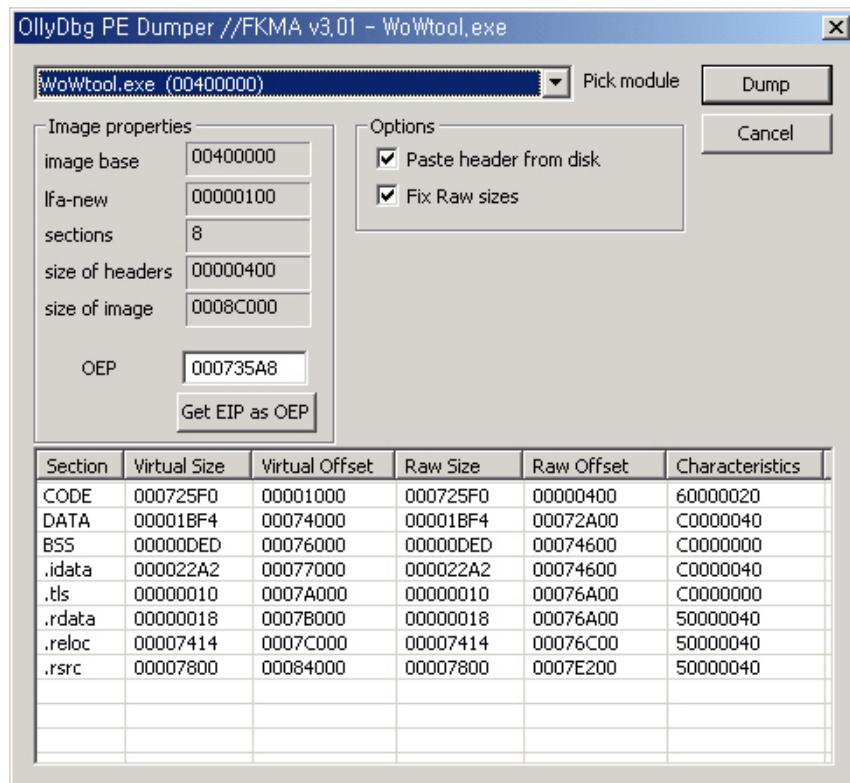


그림. PE Dumper 플러그인



Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
CODE	000725F0	00001000	000725F0	00000400	60000020
DATA	00001BF4	00074000	00001BF4	00072A00	C0000040
BSS	00000DED	00076000	00000DED	00074600	C0000000
.idata	000022A2	00077000	000022A2	00074600	C0000040
.tls	00000010	0007A000	00000010	00076A00	C0000000
.rdata	00000018	0007B000	00000018	00076A00	50000040
.reloc	00007414	0007C000	00007414	00076C00	50000040
.rsrc	00007800	00084000	00007800	0007E200	50000040

그림. OllyDbg PE Dumper 의 메인 화면

※ 관련 파일: pedumper.dll

CleanupEx plug-in

디버그 세션 파일을 편리하게 삭제할 수 있도록 만든 플러그인으로 모든 세션파일(*.udd) 및 백업 파일(*.bak)을 삭제합니다.

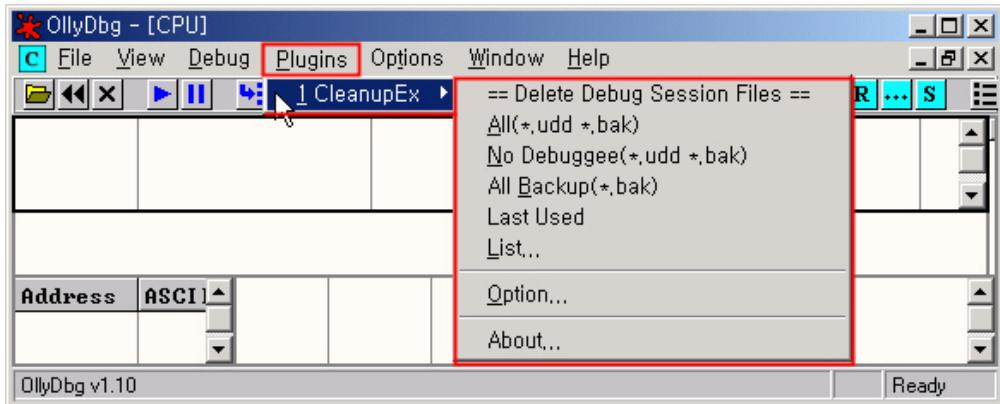


그림. CleanupEx 플러그인

또한 세션 파일 리스트(List...)에서 직접 하나씩 확인 하면서 삭제할 수 있습니다.

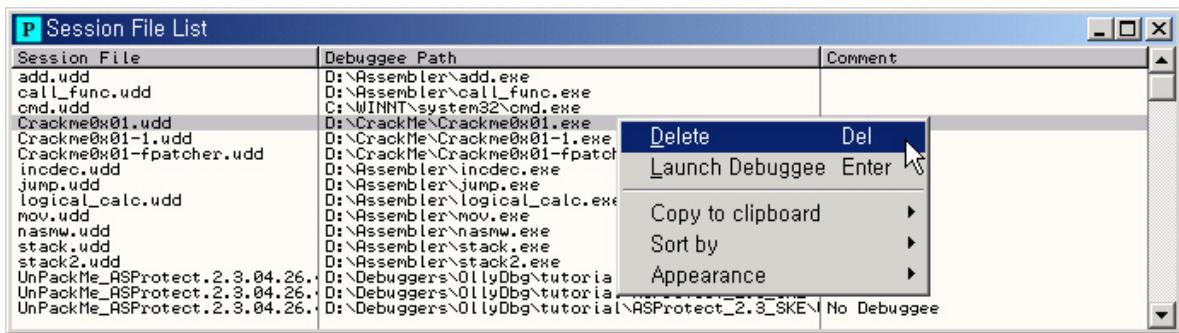


그림. 세션 파일 목록 삭제

* 관련파일: CleanupEx.dll, CleanupEx.ini

Ultra String Reference plug-in

OllyDbg 는 기본적으로 영문을 사용하고 있기 때문에 2 바이트 문자인 한글, 중국어, 일어를 지원하지 않고 있습니다. 특히 참조문자열 검색시 한글 문자를 검색할 수 없습니다(일부 검색되는 단어도 있음). 이러한 문제를 해결하는 것이 Ultra String Reference 플러그인입니다. 바이너리에서 ASCII 이나 Unicode 를 검색할 수 있는 플러그인입니다. 또한 검색된 문자열이 사용되고 있는 곳에 Break point 를 설정할 수 있습니다. 제작자가 주로 중국 온라인 게임을 리버싱 하기 위해 만들었다고 합니다. OllyDbg Font 를 적절히 설정해 주면 한국어, 중국어, 일본어 모두 검색 가능합니다.

OllyDbg 플러그인 소개

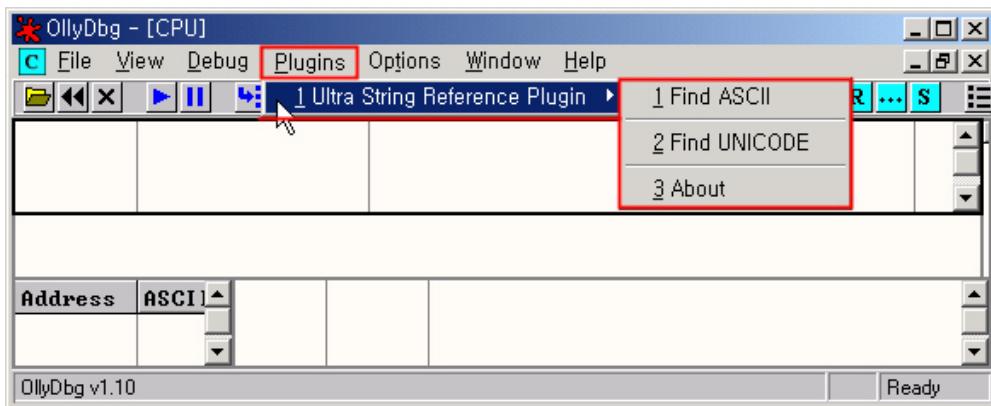


그림. Ultra String Reference 플러그인

Address	Disassembly	Text String
004DFE2A	MOV EDX, dumped_.004E0334	HP:
004DFE68	MOV EDX, dumped_.004E0354	饥饿:
004DFEBB	MOV EDX, dumped_.004E0364	火攻击力增加:
004DFF12	MOV EDX, dumped_.004E037C	火防御力增加:
004DFF69	MOV EDX, dumped_.004E0394	冰攻击力增加:
004DFFC0	MOV EDX, dumped_.004E03AC	冰防御力增加:
004E0013	MOV EDX, dumped_.004E03C4	电攻击力增加:
004E0064	MOV EDX, dumped_.004E03DC	电防御力增加:
004E00C0	MOV EDX, dumped_.004E03F4	攻击力增加:
004E011F	MOV EDX, dumped_.004E0408	防御力增加:
004E017E	MOV EDX, dumped_.004E041C	魔法攻击力增加:
004E01DD	MOV EDX, dumped_.004E0434	追
004E023D	PUSH dumped_.004E0444	获得经验值
004E0256	PUSH dumped_.004E0458	%增加
004E071B	MOV ECX, dumped_.004E0794	交易物品列表
004E0720	MOV EDX, dumped_.004E07AC	交易物品列表petadd
004E0986	MOV EDX, dumped_.004E09F0	交易对象:
004E09A9	MOV ECX, dumped_.004E09FC	交易
004E09AE	MOV EDX, dumped_.004E0A04	请求交易的玩家距离太远或者不存在.

그림. Ultra String Reference 를 사용하여 문자열 검색

* 관련 파일: ustrref.dll

FindCrypt plug-in

프로그램 내부에서 사용된 암호화 알고리즘을 검색하여 주는 플러그인입니다. 프로그램을 분석하는 동안 어떤 암호화 알고리즘이 어떤 주소에서 사용되고 있는지 알고 있다면 디버깅 작업에 매우 유용할 것입니다. 거의 모든 암호화 알고리즘은 미리 정의된 식별자(Magic Constants)를 사용하고 있으므로, 프로그램 내에서 이 식별자를 찾아 알려주는 것입니다. 원래는 IDA 플러그인 이었지만 OllyDbg 용으로 포팅 되었습니다. 지원하는 검색 알고리즘은 아래와 같습니다.

OllyDbg 플러그인 소개

[Blowfish, Camellia, CAST, CAST256, CRC32, DES, GOST, HAVAL, MARS, MD2, MD4, MD5, PKCS_MD2, PKCS_MD5, PKCS_RIPEMD160, PKCS_SHA256, PKCS_SHA384, PKCS_SHA512, PKCS_Tiger, RawDES, RC2, RC5, RC6, Rijndael, SAFER, SHA-1, SHA-256, SHA-512, SHARK, SKIPJACK, Square, Tiger, Twofish, WAKE, Whirlpool, zlib(압축)]

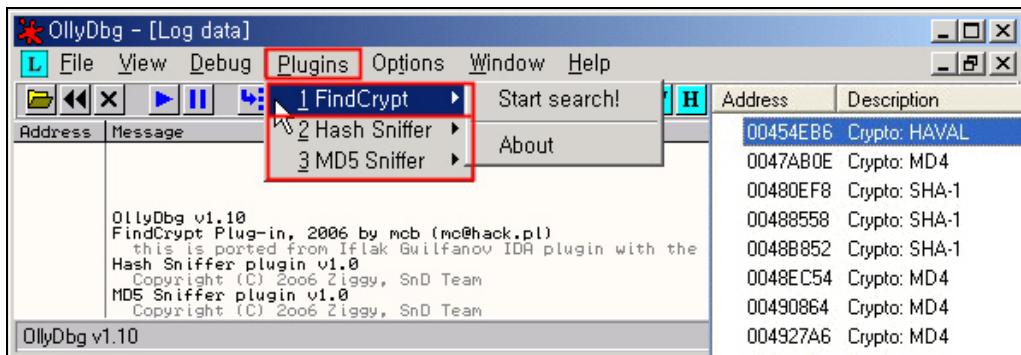


그림. FindCrypt 플러그인

※ 관련 파일: findcrypt.dll

Push Tracer plug-in

PushTracer 플러그인은 현재 모듈의 모든 PUSH [address] 명령어 위치를 검색해서 알려 줍니다. 또한 실시간으로 실행되고 있는 프로세스의 모든 Push 값을 모니터링 할 수 있으며 어떠한 주소에서 Push 되고 있는지 즉시 확인할 수 있습니다. 주로 Win32 어플리케이션을 분석 할 때에 매우 유용합니다.

OllyDbg 플러그인 소개

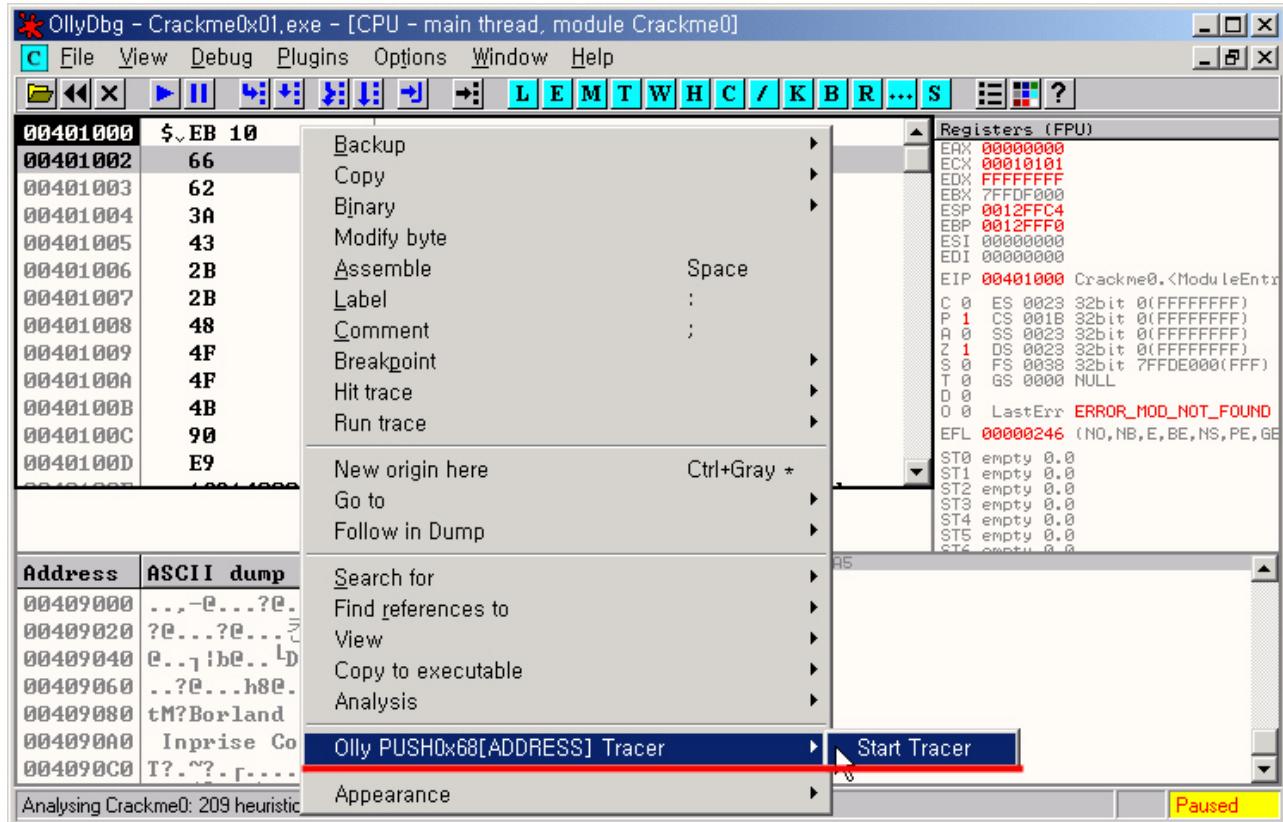


그림. Push Tracer 플러그인(CPU 화면에서 마우스 오른쪽 클릭 후 실행)

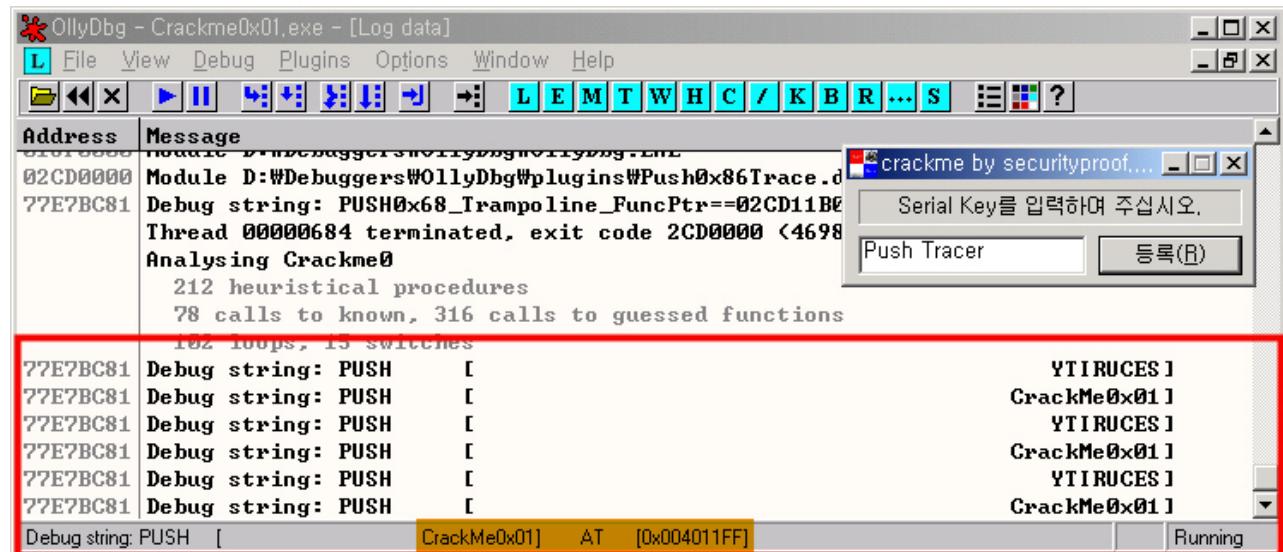


그림. Push Tracer로 실시간 덤프

Hide Caption plug-in

OllyDbg 의 MDI 원도우 캡션 Bar 를 표시 하지 않도록 합니다. 캡션을 숨기고 싶은 원도우만을 표시하여 배치를 정한 후 스냅샷을 찍는 것으로 지정할 수 있습니다. 캡션 바를 Enable/Disable 하여 고정된 화면에서 움직이지 못하도록 할 수 있습니다.

Labeler plug-in

Labeler 플러그인은 라벨 작성을 지원하는 플러그인입니다. 라벨이 붙은 메모리는 코드 중에서 참조될 때, 그 라벨은 심볼로서 표시되기 때문에 코드의 가독성이 좋아집니다. 특히 데이터 영역의 라벨 작성에 중점을 두고 있습니다. 배열이나 구조체 및 구조체 배열 등의 데이터 형태가 발견되었을 때 이 플러그인을 사용하면 간단하게 라벨을 붙일 수 있습니다.

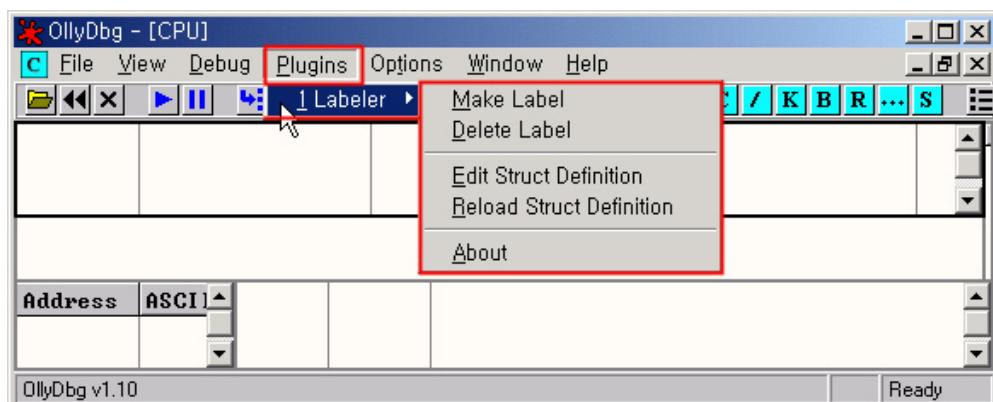


그림. Labeler 플러그인

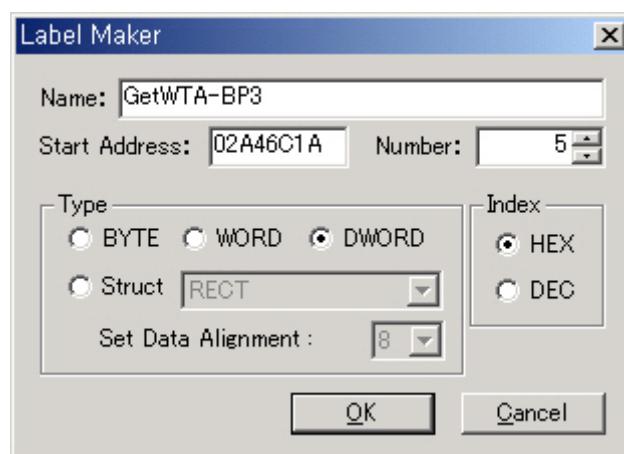


그림. Make Label 메인 화면

OllyDbg 플러그인 소개

※ 관련 파일: Labeler.dll, Labeler.def, Labeler.ini

LabelMaster plug-in

사용자 정의 라벨을 export 및 import 하는 플러그인입니다. 디버깅 중에 작성된 라벨이나 주석을 파일로 저장하거나 미리 정의한 라벨 및 주석을 불러와 적용할 수 있습니다.

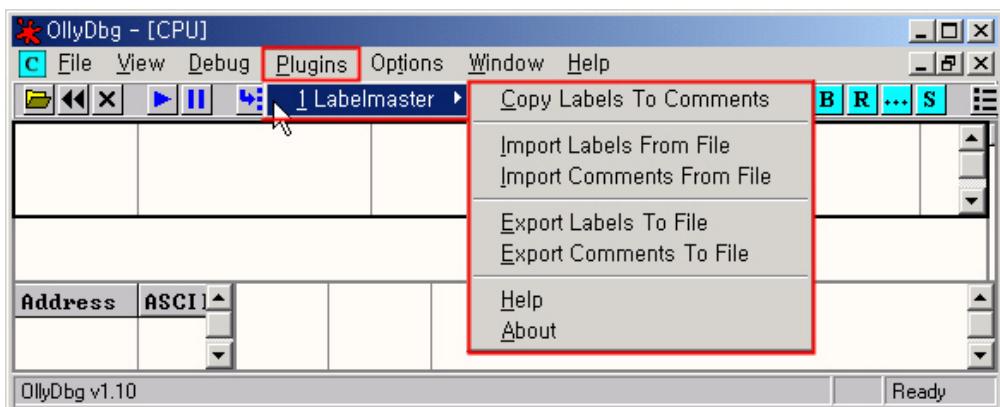


그림. Label Master 플러그인

※ 관련 파일: Labelmaster.dll

Debugger Hide plug-in

현재 프로세스가 사용자 모드 디버거(OllyDbg, WinDbg)에 의해 디버깅 중인지 확인하는 매우 간단한 Windows API 함수가 있습니다. 이들 IsDebuggerPresent(), CheckRemoteDebuggerPresent()는 프로세스 환경 블록(Process Environment Block)위에서 현재 프로세스가 사용자 모드 디버거에 의해 실행될 때 디버깅을 탐지합니다. 탐지루틴도 매우 간단하기 때문에 이를 매우 쉽게 회피할 수 있습니다. 물론 사용자 모드 디버깅을 탐지할 수 있는 개선된 몇몇 코드도 공개되어 있지만 완전한 탐지는 불가능 합니다.

시스템 위에서 커널 모드 디버거를 탐지할 때 사용되는 Native API 함수인 NtQuerySystemInformation 은 추가적으로 많은 정보를 요청할 수 있습니다.

```
NTSTATUS WINAPI ZwQuerySystemInformation(
    SYSTEM_INFORMATION_CLASS SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);
```

OllyDbg 플러그인 소개

SystemInformationClass 는 다양한 형태의 구조체 정보로 이루어져 있으며 디버깅 체크 구조체는 아래와 같습니다.

```
typedef struct __SYSTEM_KERNEL_DEBUGGER_INFORMATION {  
    BOOLEAN DebuggerEnabled;  
    BOOLEAN DebuggerNotPresent;  
} SYSTEM_KERNEL_DEBUGGER_INFORMATION, *PSYSTEM_KERNEL_DEBUGGER_INFORMATION;
```

커널 디버거가 시스템 위에서 동작 중인지 확인하기 위해서는 DebuggerEnabled 를 체크 합니다. 그러나 이러한 구조를 사용해서는 SoftICE 와 같은 디버거 종류는 탐지하지 못합니다. WinDBG 혹은 KD 와 같은 시리얼 접속 커널 디버거 만을 탐지합니다. SoftICE 와 같은 커널 드라이버를 통한 디버거는 별도의 탐지루틴이 존재합니다.

이 외에도 ProcessHeap(), NtGlobalFlag(), ZwQueryInformationProcess(), ZwSetInformation() 등을 이용하여 디버깅을 탐지할 수 있지만, 이런 탐지 방법을 우회하기 위한 다양한 플러그인이 또 존재합니다. 이런 디버깅 회피 플러그인은 다른 플러그인과 조합해서 사용하는 경우가 많습니다. 많이 사용되는 디버거 숨김 플러그인으로는 “IsDebug, hideod, AntiOlly, hidodbg, HideDebugger, Invisible, antianti, HideDbg, uhooker” 등이 있습니다.

최근에는 Windows API 나 Native API 함수를 사용하지 않고 인터프리터 언어로 작성한 허킹(Hooking) 플러그인들이 공개되어 있습니다.

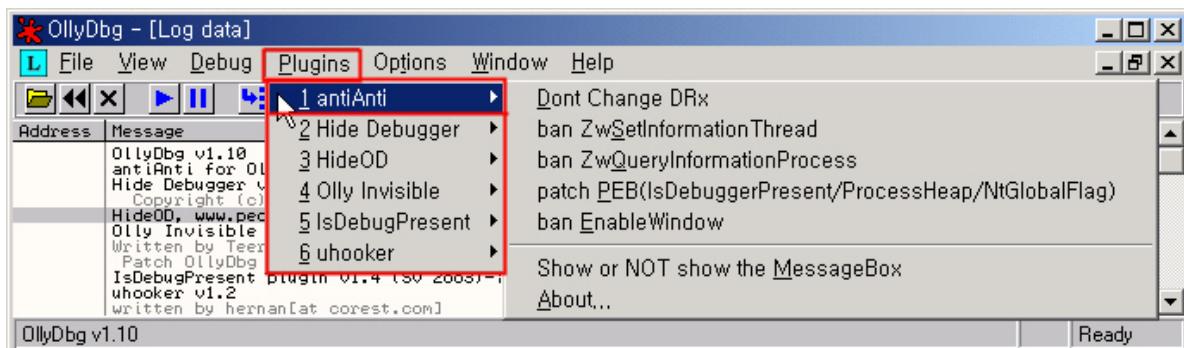


그림. 디버거 숨김 플러그인

OllyScript plug-in

이름 그대로 스크립트를 실행하는 플러그인입니다. OllyDbg 스크립트 언어는 어셈블리 언어와 비슷하며, OllyDbg 의 대부분 조작을 자동으로 수행할 수 있습니다. IDA 와 호환 가능한 리버싱 데이터의 추출이나 매뉴얼 언팩을 쉽게 할 수 있도록 다양한 패커의 OEP 를 찾는 스크립트 등이 이미 만들어져 있습니다. 주로 팩킹된 바이러스나 악성 프로그램 분석시 가장 많이 이용될 것입니다.

ollyDbg 플러그인 소개

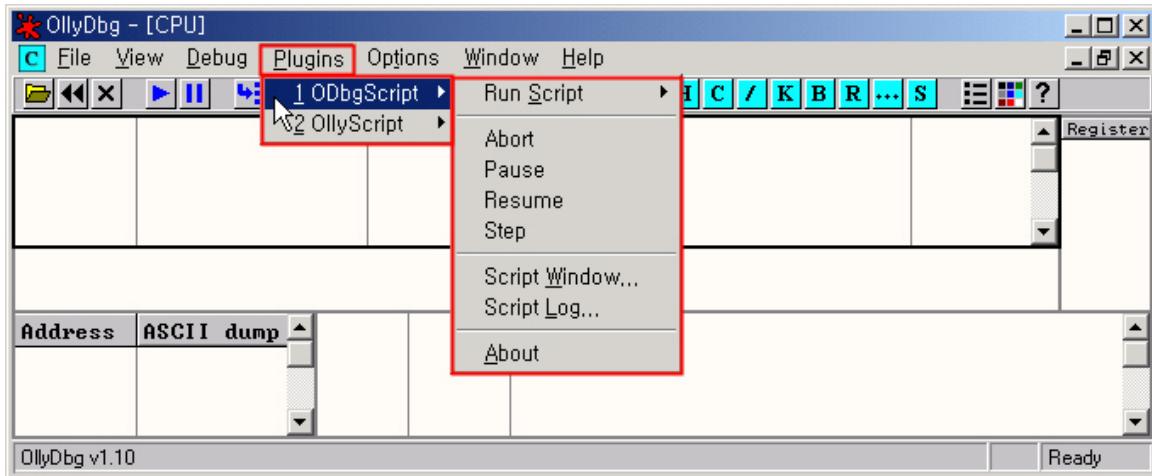


그림. ODbgScript 플러그인

* 관련 파일: ODbgScript.dll, OllyScript.dll, 그 외 스크립트 파일

아래 다운로드 사이트를 통해 확인할 수 있듯이 수십 종류의 스크립트들이 존재합니다.

<http://www.openrce.org/downloads/ollyscript/>

http://www.tuts4you.com/blogs/e107_files/downloads/scripts/

<http://www.pediy.com/tools/debuggers/ollydbg/script.htm>

OllyMachine plug-in

OllyMachine⁴⁸은 OllyScript와 동일하게 스크립트를 실행할 수 있는 플러그인입니다. OllyScript 제작자도 이 플러그인 작성에 도움을 준 것 같습니다. 어셈블리 언어를 손쉽게 스크립트 언어로 작성할 수 있는 특징이 있습니다. 본인의 입맛에 맞는 것을 사용하면 됩니다. 플러그인은 대부분 소스도 공개되어 있기 때문에 흥미 있는 분은 직접 수정 하여도 됩니다. 몇몇 플러그인은 본인 환경에 맞추기 위해 소스 코드를 수정 후 재 컴파일 해야 할 때도 있습니다.

⁴⁸ OllyMachine: <http://www.luocong.com/om/>

OllyDbg 플러그인 소개

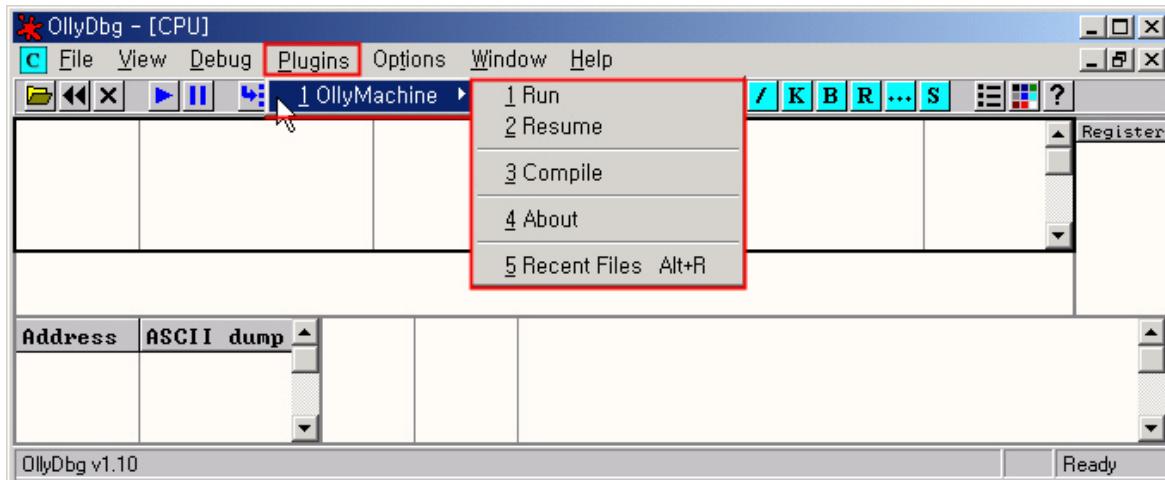


그림. OllyMachine 플러그인

※ 관련파일: OllyMachine.dll, OllyMachine.ini, 그 외 스크립트 파일

OllyDbg 스크립트를 편리하게 작성/편집하기 위한 스크립트 편집기도 존재합니다. 주로 이용되는 OllyDbg 스크립트 편집기는 OSEditor, Ollyscript Editor 등이 있습니다.

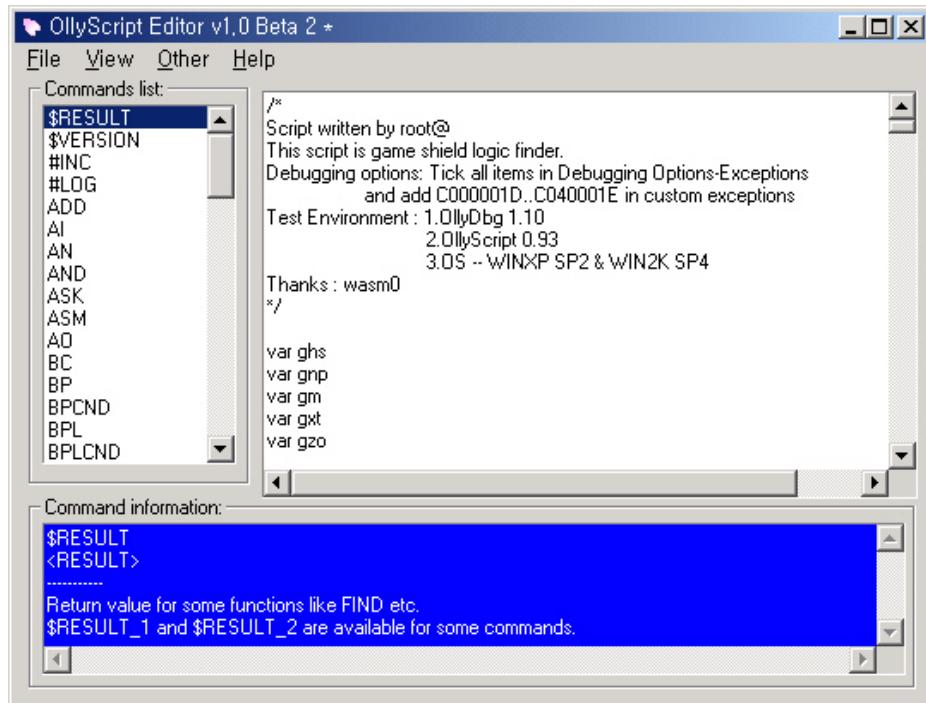


그림. 스크립트 편집기

Watch Man plug-in

OllyDbg 플러그인 소개

디버깅 중에 주목해야 할 변수/수식을 특정한 형태로 변환하기 위한 플러그인입니다. OllyDbg 에서는 스택 프레임에 의한 로컬 변수나 함수의 인수를 인식하는 경우 사용하기 쉽게 [local.0] [arg.0]과 같은 이름으로 치환할 수 있습니다. 이런 형태로 일련의 변수를 쉽게 Watch 변수로 바꿀 수 있도록 만들었습니다.

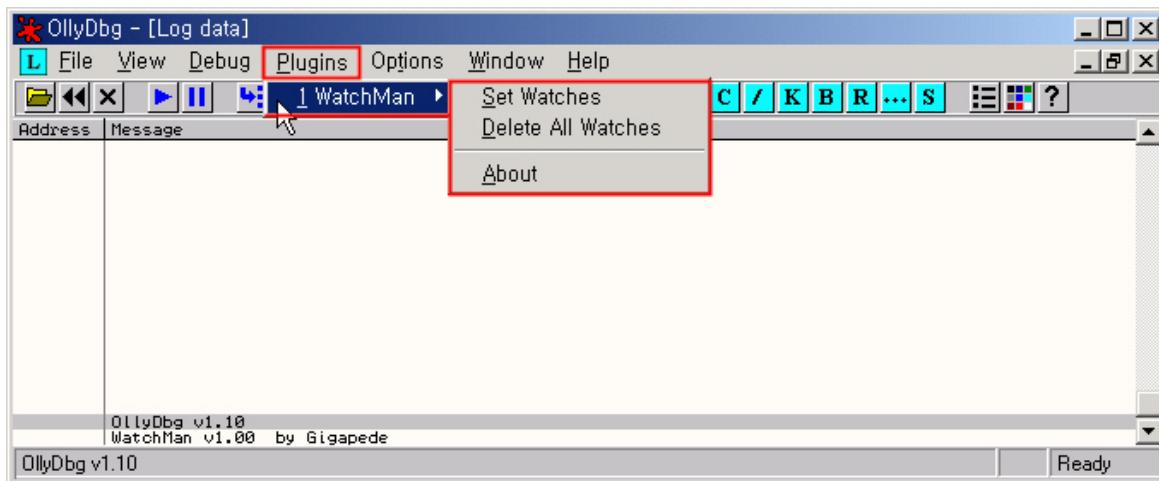


그림. WathMan 플러그인



그림. WathMan 플러그인 메인 화면

※ 관련파일: WatchMan.dll

Window Juggler plug-in

Windows Juggler⁴⁹는 원도우 정보 표시와 여러 가지 속성을 변경 하는 플러그인 입니다. 플러그인을 실행시킨 후 키보드 “Shift”키를 누른 채 정보 표시를 원하는 윈도우를 클릭하면 됩니다. 윈도우 핸들 정보를 가지고 Break

⁴⁹ Window Juggler: <http://esseemme.altervista.org>

OllyDbg 플러그인 소개

point를 설정할 때 주로 이용하거나 특정 윈도우 이벤트를 발생시켜 디버깅을 쉽게 할 수 있도록 도와 줍니다.

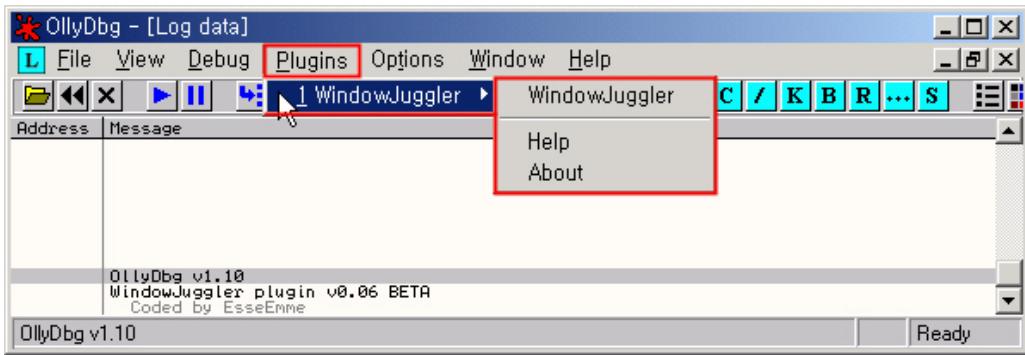


그림. WindowJuggler 플러그인

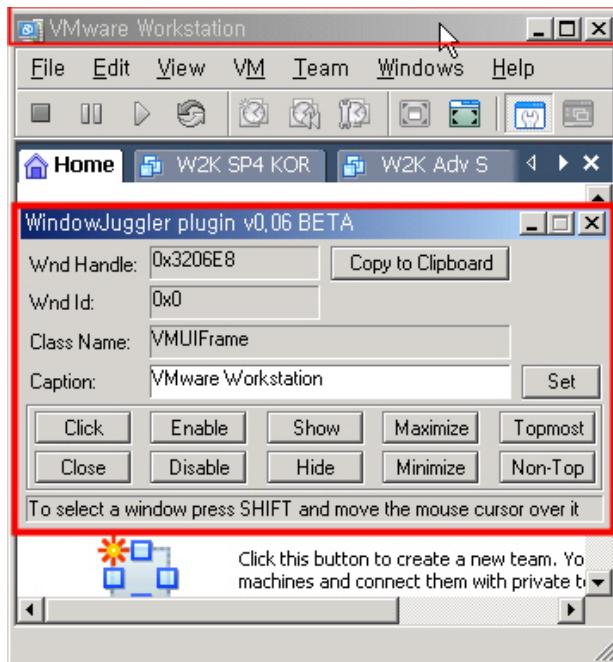


그림. Window Juggler 의 메인 화면(VMware 테스트)

OllyBonE Plug-in

일반적으로 패킹된 바이너리 파일은 PE Header에 Entry Point가 Decryptor stub로 이동된 후 프로그램의 시작 위치(OEP: address Of Entry Point)로 이동되어 실행됩니다. 즉 패킹된 상태가 풀리는 시점에서 메모리를 덤프 하면 패킹이전의 바이너리를 얻을 수 있습니다. OllyBonE는 이러한 패킹된 바이너리를 반자동으로 unpacking 해 주는 플러그인입니다. OllyBonE은 프로그램이 시작되는 OEP 위치에 Break point를 설정할 수 있어서 코드 분석이 가능하고 빠르게 unpacking 할 수 있도록 도와줍니다. OEP의 위치는 OllyDbg 왼쪽 하단에 “break-on-execute at [Address]”로 확인 할 수 있습니다.

OllyDbg 플러그인 소개

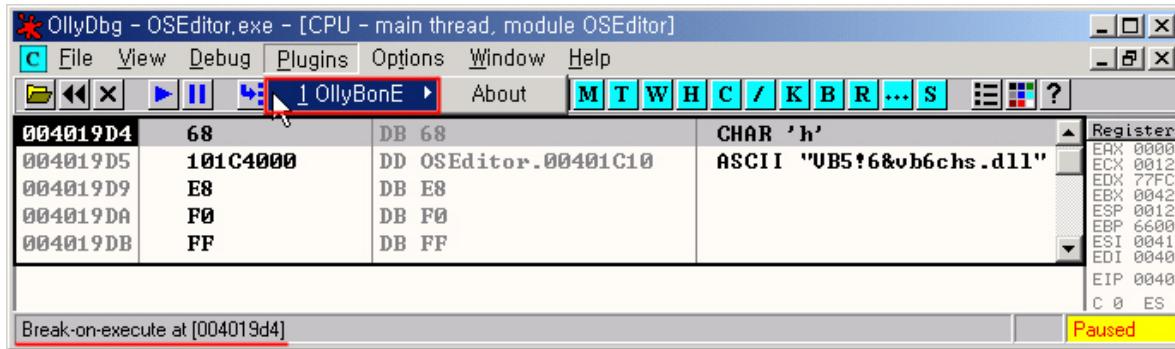


그림. OllyBonE 플러그인

※ 관련 파일: ollybone.dll(플러그인 디렉터리 복사), ollybone.sys(OllyDbg 루트 디렉터리 복사)

OllyFlow plug-in

OllyFlow는 IDA내 포함된 표준 그래픽 프로그램(wingraph32⁵⁰)의 도움을 얻어서 제어 분기 구조를 그래프 형식으로 보여주는 플러그인입니다. 크로스 레퍼런스 기능 참조나 특정 영역의 제어 분기 구조를 분석하기 쉽도록 Flow chart형식의 그림 파일로 보여 줍니다. IDA가 설치되어 있지 않으면 wingraph32 소스코드를 받아 컴파일 한 후 OllyFlow 플러그인과 연동해 주면 됩니다.

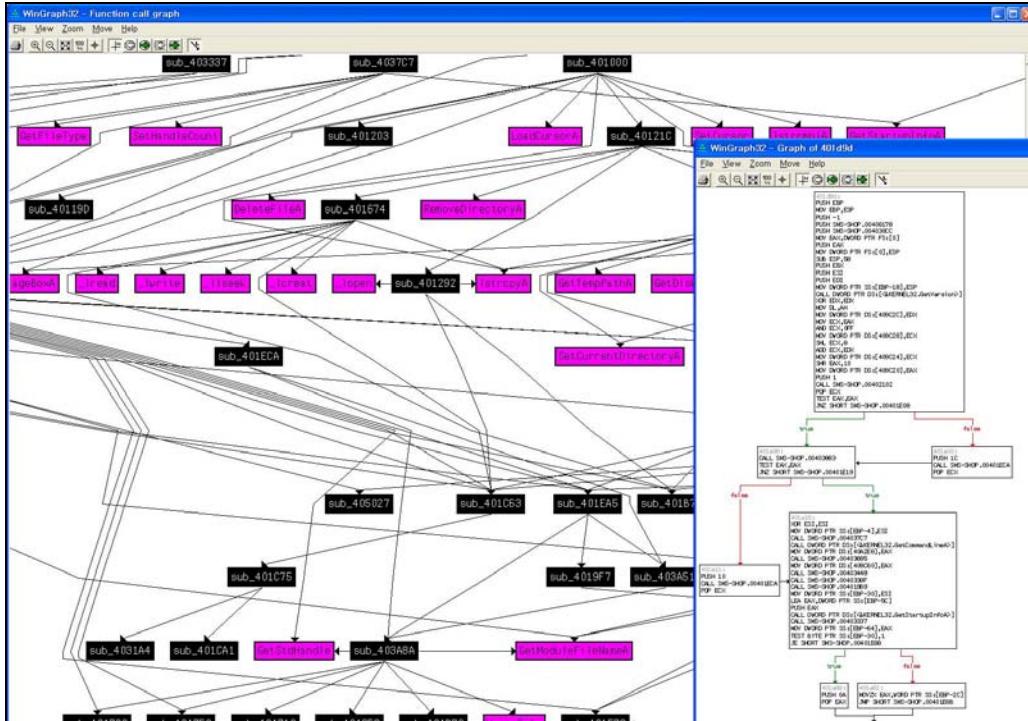


그림. OllyFlow 플러그인

⁵⁰ Wingraph32: http://www.datarescue.com/idabase/freefiles/wingraph32_src.zip

* 관련 파일: ollyflow.dll(플러그인 디렉터리 복사), wingraph32.exe (소스 컴파일 후 OllyDbg 루트에 복사)

마지며

여기서 소개하지 못한 다양한 플러그인이 있습니다. OllyDbg 단독으로 실행되는 플러그인도 있지만, 외부 프로그램과 연계되어 실행되는 플러그인도 존재합니다. 이처럼 디버깅을 손쉽게 도와주는 다양한 플러그인은 직접 사용해 보고 자기 것으로 만드는 것이 가장 좋습니다. 아래는 잘 알려진 플러그인 목록입니다.

【+BP-OLLY, AnalyzeThis, antiAnti, ANTI DRx, ApiBreak, Armadillo Process Detach, ASProtect, Asm2Clipboard, AttachAnyway, BASE64, Bookmarks, BreakOnLoad, Breakpoint Manager, CLBPlus!, CleanUp, CleanupEx, Code Ripper, Data Ripper, Command Bar, Command line, Convertisseur ASM->HTML, DebugPrivilege, Delphi Helper, dUP2, Evanescence, ExtraCopy, Extras, ForceAlloc, GODUP Plugin, Hash Sniffer, Heap Vis, HideDBG, Hide Debugger, HideOD, IsDebugPresent, LabelMaster, LoadMap, MapConv, MemoryBackup, MemoryManage, NonWrite, Olly Advanced, OEP Finder, OllyDbg PE Dumper, OllyDbgScript, OllyDump, OllyFlow, OllyGraph, OllyHelper, Olly Invisible, OllyMachine, OllyScript, OllyTbar Manager Gold, OllyTiper, OllyUni, PolyMorphic Breakpoint, RL!APIFinder, RL!Weasle, Table Exporter, Turbo Debug, Ultra String Reference, WindowInfos】

제 3 장 Windows 크랙 기본편

3.5 날짜 제한 Protection의 크랙

들어가며

쉐어웨어, 특히 상용 소프트웨어 체험판은 일정 기간이 지나면 더 이상 사용할 수 없도록 제한하고 있습니다. 이렇게 사용 기간을 제한 하는 것도 몇 종류가 있습니다. 최초 프로그램 실행 시점을 기준으로 30 일까지 사용할 수 있는 것이나, 지정된 년/월/일이 지나면 아예 실행되지 않도록 하는 것 등이 있습니다. 이런 보호 기법은 PC의 날짜를 변경하는 것으로 일단 계속 사용 할 수 있지만⁵¹, 윈도우 업데이트가 되지 않는 등 여러모로 불편하게 됩니다. 이번 장에서는 소프트웨어에서 날짜 제한 Protection의 체크 루틴 방법과 사용 기간을 무효화 시키는 방법에 대하여 알아 보도록 하겠습니다.

현재 시간 획득 방법

사용기한 체크를 수행하려면 현재 PC 시간을 알아 보아야 할 필요가 있으나, 이것도 윈도우 API 함수가 이용됩니다. 대표적으로 GetLocalTime()입니다. 세계 표준 시간(UTC: Universal Time Coordinated)을 획득하는 GetSystemTime()도 가끔 이용되지만 이 2종류만 외워 두면 문제 없을 것입니다.

C 언어에서는 표준 라이브러리 함수로서, 헤더 파일(time.h)에 날짜, 시간관리 함수가 정의되어 있습니다. 그 중의 time()을 이용하면, “API 함수 GetLocalTime 을 이용하지 않고도 시간을 확인 할 수 있다…” 라고 생각하는 것은 성급한 생각입니다. 표준 라이브러리 함수는 외부 호출이 아니라 실행 파일 내부에 함수의 원형(명령코드)이 포함됩니다. 그러므로 실제 time() 함수 내에서 GetLoacalTime 을 호출하고 있기 때문에, 외부 함수의 참조에서 해당 루틴이 사용되고 맙니다.

시간함수 예제(VC++)

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

using namespace std;
```

⁵¹ 현재 시간을 NTP(Network Time Protocol) 서버나 SNTP(Simple Network Time Protocol) 서버 등을 통해 확인하고 있는 경우라면 이 방법은 통용되지 않습니다. 그러나 Protection을 위해 네트워크 서버로 시간을 체크하는 경우는 거의 없습니다.

날짜 제한 Protection 크랙

```
int main()
{
    SYSTEMTIME st;
    GetSystemTime(&st);
    cout << "System time is(UTC) " << st.wYear << "/" << st.wMonth << "/" << st.wDayOfWeek << "/"
        << st.wDay << "/" << st.wHour << ":" << st.wMinute << ":" << st.wSecond << st.wMilliseconds << endl;

    GetLocalTime(&st);
    cout << "Local time is(PC) " << st.wYear << "/" << st.wMonth << "/" << st.wDayOfWeek << "/"
        << st.wDay << "/" << st.wHour << ":" << st.wMinute << ":" << st.wSecond << st.wMilliseconds << endl;

    return 0;
}
```

OllyDbg에서의 API 함수 확인

이번에는 날짜 제한 보호 방법을 채택한 CrackMe 【CrackMe02.exe】를 이용하여 설명하겠습니다. 정상적으로 실행했을 경우는 그림 1에서 보이는 윈도우가 표시됩니다. 2006년 8월 15일⁵² 이전이라면 정상적으로 실행이 가능합니다. 여기서 컴퓨터 날짜를 2006년 8월 16일로 변경하고 나서 다시 실행하면 그림 2와 같은 에러 메시지가 표시됩니다.

이번 Crackme는 【지정된 년/월/일이 지나면 실행되지 않도록 하는】 형태로, 보통 체험판 소프트웨어에서 정품 업데이트를 유도하기 위해 이용되고 있습니다. 그럼 OllyDbg를 이용하여 이러한 제한 설정을 풀어 보도록 하겠습니다.

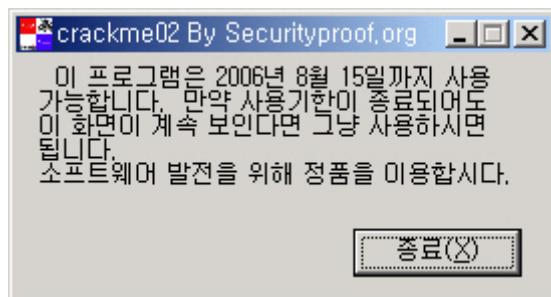


그림 1. 정상적인 실행의 경우

⁵² 대한민국 광복절 [光復節]

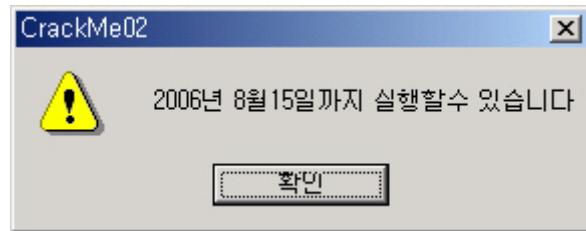


그림 2. 사용기한이 지난 경우

먼저 OllyDbg로부터 [CrackMe02.exe]를 불러옵니다. 불러온 직후는 Crackme가 바로 실행되지 않지만, 사용기한 체크는 실행직후에 행해지기 때문에, F9 키를 누르기 전에 브레이크 포인트를 설정해야 합니다.

【Ctrl+N】 키를 눌러 API 함수 목록을 표시한 후 “GetLocalTime”에 브레이크 포인트를 설정해 주십시오(마우스 오른쪽 클릭 → Set breakpoint on every reference 선택).

이것으로 기본적인 디버깅 준비는 갖추어졌기 때문에 F9 키(Run)를 눌러서 crackme를 실행 시킵니다. 그러면 곧바로 주소 004011F5h에서 멈추게(Break) 됩니다.

```

1 004011F4 | . 51      PUSH ECX          ; /pLocaltime
2 004011F5 | . E8 82740000 CALL <JMP.&KERNEL32.GetLocalTime>; 브레이크 포인트
3 004011FA | . 8D45 F0    LEA EAX,DWORD PTR SS:[EBP-10]

```

그림 3. Break Point 설정

GetLocalTime은 하나의 인자 값을 동반하는데, 이 인자는 시간 정보를 저장하는 SYSTEMTIME 구조체 영역의 포인터를 지정합니다. SYSTEMTIME 구조체의 구성은 아래와 같습니다.

SYSTEMTIME 구조체(C 언어의 헤더 파일에서)	
typedef struct _SYSTEMTIME {	
WORD wYear;	//년
WORD wMonth;	//월
WORD wDayOfWeek;	//요일(0이 일요일, 1이 월요일 … 6이 토요일)
WORD wDay;	//일
WORD wHour;	//시
WORD wMinute;	//분
WORD wSecond;	//초
WORD wMilliseconds;	//밀리초
}	SYSTEMTIME;

GetLocalTime을 실행하여 이 영역에 현재 일시가 들어가는 모양을 관찰해 봅시다. 스택 윈도우의 “pLocaltime”이라고 표시되어 있는 행의 값(그림 4), 여기에서는 주소 0012FC20h입니다. 이 값이 표시되어 있는

날짜 제한 Protection 크랙

곳에서 【마우스 오른쪽 클릭→Follow in Dump】를 하면 그 주소 영역이 덤프 되어 왼쪽 화면에 표시됩니다(그림 5).

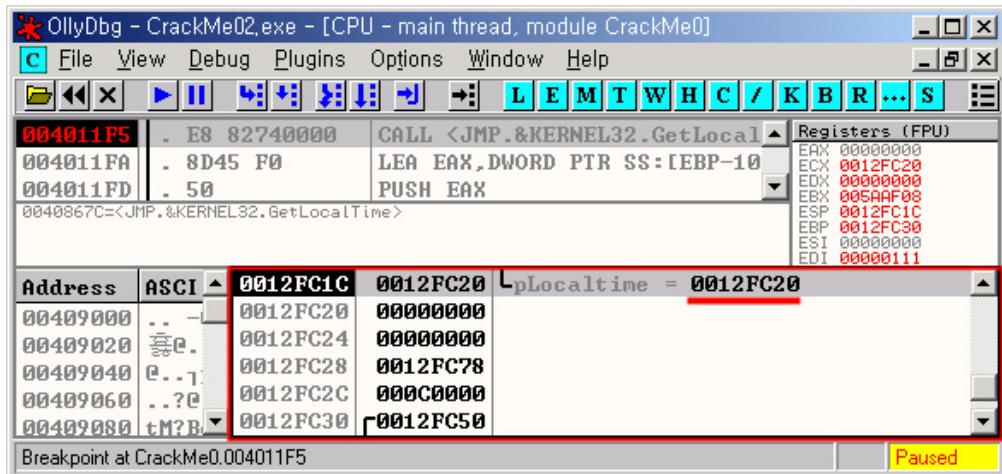


그림 4. pLocaltime(이 값은 컴퓨터 환경에 따라 달라짐)

Address	Hex dump	ASCII
0012FC20	00 00 00 00 00 00 00 00 78 FC 12 00 00 00 0C 00x?.....
0012FC30	50 FC 12 00 8F 15 E1 77 C2 03 09 00 11 01 00 00	P.??.?.?.
0012FC40	39 30 00 00 00 00 00 00 00 00 00 00 CD AB BA DC	90.....誠
0012FC50	8C FC 12 00 9C 27 E0 77 77 11 40 00 C2 03 09 00	90.?.?w@.?.
0012FC60	11 01 00 00 39 30 00 00 00 00 00 00 10 01 00 00	◀.90.....
0012FC70	11 01 00 00 C8 DB 5C 00 00 00 00 00 05 03 8B 02	◀.90.....
0012FC80	01 00 00 00 80 00 00 00 00 00 00 00 B4 FC 12 00	↑.....
0012FC90	CA C1 E0 77 C2 03 09 00 11 01 00 00 39 30 00 00	恰??.?.
0012FCA0	00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00[.....]
0012FCB0	00 00 00 00 10 FD 12 00 AF 1B F9 77 C4 FC 12 00+?.?.
0012FCC0	18 00 00 00 C8 DB 5C 00 11 01 00 00 39 30 00 00	↑...
0012FCD0	00 00 00 00 01 00 00 00 19 27 E0 77 0B 3A E0 77

그림 5. 해당 주소 덤프(이 값은 컴퓨터 환경에 따라 달라짐)

GetLocalTime을 실행하기 전에 불필요한 브레이크 포인트를 없애 주면, 예상치 못한 곳에서 브레이크 되는 일이 없어지기 때문에 프로그램 추적이 쉬워집니다. Alt+B키로 브레이크 포인트 목록을 표시하고, 어드레스 004011F5h 이외의 행에서 스페이스 키⁵³를 눌러서 3 번째 열 Active를 [Always]에서 [Disabled]로 바꾸면 (그림 6), 브레이크 포인트가 일시적으로 중지 됩니다.

Breakpoints			
Address	Module	Active	Disassembly
004011F5	CrackMe0	Always	CALL <JMP.&KERNEL32.GetLocalTime>
004058E0	CrackMe0	Disabled	CALL <JMP.&KERNEL32.GetLocalTime>
0040867C	CrackMe0	Disabled	JMP DWORD PTR DS:[<&KERNEL32.GetLocalTime>]

그림 6. 불필요한 브레이크 포인트 Disable

⁵³ 스페이스 키 대신에 DEL 키를 누르면 브레이크 포인트 자체가 없어집니다.

F8 키를 눌러서 GetLocalTime을 단계별(Step over)로 실행하면, 덤프윈도우에 표시된 영역에 현재 일시(日時)가 저장됩니다. 그러나 사람이 이해하기 어려운 16 진수 값으로 표시되기 때문에 표시방법을 변경합니다. 그림 7처럼 덤프 윈도우에서 【마우스 오른쪽 클릭→Short→Signed decimal】을 선택하면 10 진수 형태로 표시됩니다(그림 8)⁵⁴.

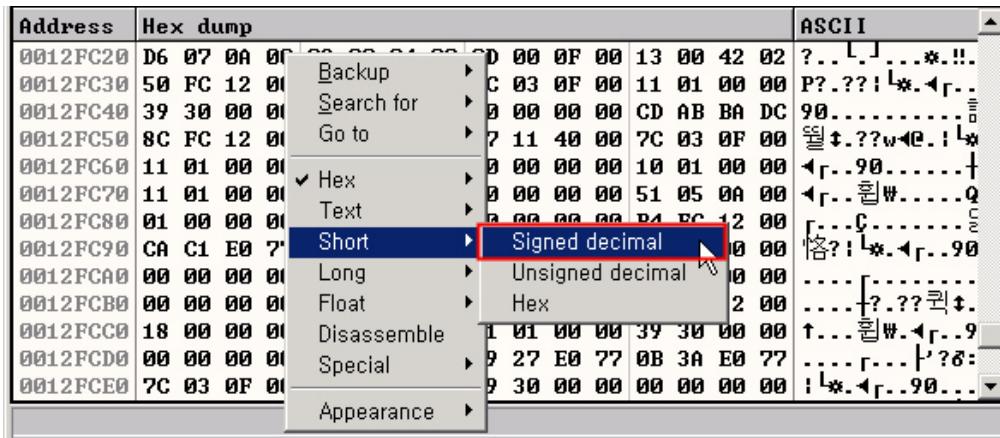


그림 7. 덤프 윈도우 표시 방법 변경(Signed decimal)

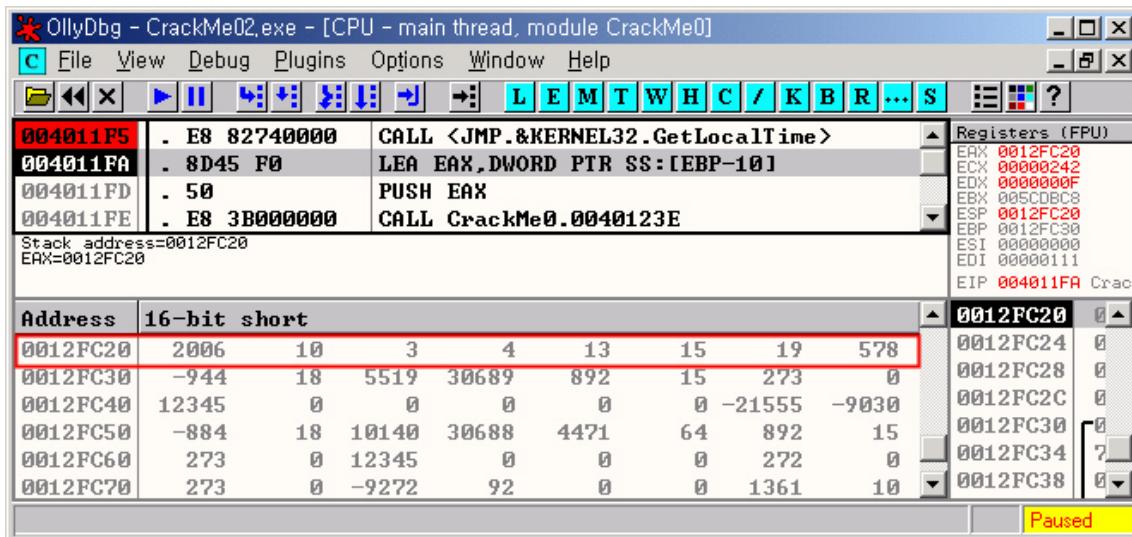


그림 8. 컴퓨터의 현재 시간 획득(Address는 컴퓨터 환경에 따라 달라짐)

이렇게 표시 된다면 수집된 현재 시간을 간단히 읽을 수 있습니다. 그림 8 과 앞에서 설명한 SYSTEMTIME 구조체를 대조해 보는 것으로,

2006년 10월 수요일 4일 13시 15분 19초 578미리초

⁵⁴ 기본 값으로 되돌릴 때는 【마우스 오른쪽 클릭→Hex→Hex/ASCII(16 bytes)】를 선택할 것.

인 것을 알 수 있습니다. 물론 이 일시는 문서 작성시의 것이므로, 실제로 여러분이 crackme 에 몰두할 때의 시간이 나올 것입니다.

체크 루틴 추적

GetLocalTime 함수 실행후의 코드는 아래와 같습니다.

```

1 004011FA | . 8D45 F0      LEA EAX, DWORD PTR SS:[EBP-10] ; 현재 위치
2 004011FD | . 50          PUSH EAX ; /Arg1
3 004011FE | . E8 3B000000 CALL CrackMe0.0040123E ; 여기에서 F7키를 누름
4 00401203 | . 59          POP ECX
5 00401204 | . 85C0        TEST EAX, EAX
6 00401206 | . 75 2E        JNZ SHORT CrackMe0.00401236 ; 메시지 박스 부분
7 00401208 | . 6A 30        PUSH 30 ; /Style = MB_OK|MB_ICONEXCLAM
8 0040120A | . 68 51914000 PUSH CrackMe0.00409151 ; |Title = "CrackMe02"
9 0040120F | . 68 2C914000 PUSH CrackMe0.0040912C ; |Text = "2006년 8월 15일 까지"
10           | .           ; 실행 할수 있습니다."
11 00401214 | . FF75 08      PUSH DWORD PTR SS:[EBP+8] ; |hOwner
12 00401217 | . E8 20750000 CALL <JMP.&USER32.MessageBoxA> ; \MessageBoxA

```

그림 9. GetLocalTime Traceing

경험상 CALL 명령 직후에 비교/분기명령이 있을 경우, CALL 명령으로 호출되어 있는 함수는 체크루틴일 가능성이 높기 때문에 CALL 명령 내부 함수까지 추적해야 합니다. 반대로 CALL 명령 직후에 비교/분기명령이 없을 경우, 체크 루틴과는 직접 관계 없는 함수일 가능성이 높기 때문에(물론 예외도 있음) 시간 절약을 위해 코드 추적은 하지 않습니다.

그림 9에서 어드레스 004011FEh에 CALL 명령이 있고, 그 직후에 함수의 반환값이 저장되는 EAX 레지스터값에 의해 분기가 행해지고 있는 것을 확인할 수 있습니다. 이 함수의 반환값이 제로가 아니라면 좋을 것 같지만, 이 함수 내부 코드는 추적할 필요가 있을 것 같습니다. F8 키(F7 키라도 가능)를 두번 눌러서 CALL 명령까지 진행한 후 F7 키를 눌러 주십시오.

F7/F8 키 둘 다 스텝 실행을 수행하지만, F7 키는 CALL 명령으로 호출되어 있는 함수 내부 추적까지 진행하는 “상세” 스텝 실행이기 때문에 주소 0040123Eh의 POP 명령으로 옮겨갑니다. 반면에 F8 키를 누르면 CALL 명령 내부로 추적하지 않고 바로 00401203h의 【POP ECX】 명령을 실행합니다.

여기까지 성공적으로 따라 왔다면 CALL 명령이 호출되어 있는 체크루틴으로 추적해 갈 수 있습니다.

```

1 0040123E /$ 55      PUSH EBP
2 0040123F |. 8BEC      MOV EBP,ESP
3 00401241 |. 8B45 08    MOV DWORD PTR SS:[EBP+8]
4 00401244 |. 66:8138 D607 CMP WORD PTR DS:[EAX],7D6 ;현재연도≥2006
5 00401249 |. 73 07      JNB SHORT CrackMe0.00401252 ;True이면 00401252h로
6 0040124B |. B8 01000000  MOV EAX,1                  ;False이면 계속 사용
7 00401250 |. 5D          POP EBP                 ;반환값으로 1을 리턴
8 00401251 |. C3          RETN                   ;사용기한내 리턴

```

그림 10. 현재연도 체크 부분(1)

CMP 명령으로 GetLocalTime에서 확인한 현재연도와 사용 만료연도 7D6h(2006)가 비교되어, 다음의 JNB(Jump if Not Below)명령에 의해 [현재연도≥2006]이라면 어드레스 00401252h로 점프(체크 계속)합니다. 조건을 만족시키지 못한 경우라면 2006년보다 이전해 이므로 계속 사용 가능하다는 판단을 내립니다. 그러므로 EAX 레지스터에 반환 값으로 1이 들어가고, RETN 명령으로 CALL 명령을 호출한 다음 어드레스인 00401203h로 리턴됩니다.

```

1 00401252 |> 66:8138 D607  CMP WORD PTR DS:[EAX],7D6 ;현재연도≤2006
2 00401257 |. 76 04      JBE SHORT CrackMe0.0040125D ;True이면 0040125Dh로 점프
3 00401259 |. 33C0      XOR EAX,EAX           ;2006년을 넘겼으면
4 0040125B |. 5D          POP EBP             ;반환값을 0으로
5 0040125C |. C3          RETN                ;사용기간 만료

```

그림 11. 현재연도 체크 부분(2)

이것도 마찬가지로 CMP 명령으로 현재연도와 사용 만료 연도 2006년이 비교되어, JBE(Jump if Below or Equal)명령에 의해 [현재연도≤2006]⁵⁵이라면 체크를 계속 진행합니다. 만약 2006년을 넘겼으면 사용기한 만료로 판단하여 반환값 0을 되돌려줍니다.

```

1 0040125D |> 66:8378 02 08  CMP WORD PTR DS:[EAX+2],8 ;현재월≤8
2 00401262 |. 73 07      JNB SHORT CrackMe0.0040126B ;True이면 0040126Bh로
3 00401264 |. B8 01000000  MOV EAX,1                  ;8월 이전이면
4 00401269 |. 5D          POP EBP                 ;반환값을 1으로
5 0040126A |. C3          RETN                   ;사용기간내 리턴

```

그림 12. 현재 월 체크 부분(1)

현재연도가 2006년이라면 다음은 현재월의 체크입니다. 기본 루틴은 연도 체크와 다를 바 없습니다.

⁵⁵ 현재연도≥2006이 아니면 이 비교명령은 실행되지 않기 때문에, [현재연도==2006]로 바꾸어 해석해도 문제없습니다.

날짜 제한 Protection 크랙

[현재월 \geq 8]이라면 JNB 명령에 의해 점프하여 체크 계속, 8월보다 이전이라면 사용기한 내로 간주되어 반환값 1을 되돌려 줍니다.

The screenshot shows assembly code from the file Crackme02_0040126Bh.txt. The code checks if the current month is greater than or equal to 8. If true, it jumps to address 00401276. Otherwise, it XORs EAX with EAX (clearing it), pops EBP, and returns. The comments explain the logic: if the month is less than or equal to 8, the value is returned; if it's greater, it goes to address 00401276.

Line	Address	OpCode	Description
1	0040126B	JNB	66:8378 02 08 CMP WORD PTR DS:[EAX+2],8 ;현재월<=8
2	00401270	JBE	76 04 JBE SHORT CrackMe0.00401276 ;True이면 00401276로
3	00401272	XOR	33C0 XOR EAX,EAX ;8월을 넘겼으면
4	00401274	POP	5D POP EBP ;반환값 0
5	00401275	RETN	C3 RETN ;사용기간 만료 리턴

그림 13. 현재 월 체크 부분(2)

[현재월 \leq 8]이라면 ⁵⁶ 체크가 계속되고, 현재월이 8 월을 넘겼으면 사용기한 만료로 간주 되어 반환값 0 을 되돌려줍니다.

The screenshot shows assembly code from the file Crackme02_00401276h.txt. It checks if the current day is greater than 15. If true, it jumps to address 00401284. Otherwise, it moves the value 1 to EAX, pops EBP, and returns. The comments explain the logic: if the day is less than or equal to 15, the value is returned; if it's greater, it goes to address 00401284.

Line	Address	OpCode	Description
1	00401276	CMP	66:8378 06 OF CMP WORD PTR DS:[EAX+6],OF ;현재일 > 15
2	0040127B	JA	77 07 JA SHORT CrackMe0.00401284 ;True면 00401284h로
3	0040127D	MOV	B8 01000000 MOV EAX,1 ;15일 까지는
4	00401282	POP	5D POP EBP ;반환값 1
5	00401283	RETN	C3 RETN ;사용기한내
6	00401284	XOR	33C0 XOR EAX,EAX ;15일을 넘겼으면
7	00401286	POP	5D POP EBP ;반환값 0
8	00401287	RETN	C3 RETN ;사용기한 만료

그림 14. 현재 일 체크 부분

2006 년 8 월이라면 현재일 까지 체크 합니다. [현재일 $>$ 15]이라면 JA(Jump if Above) 명령에 의해 점프한 후, 사용기한 만료로 간주되어 반환값 0 을 되돌려줍니다. 현재일이 15 일까지라면 사용기한 이내로서 반환값 1 을 되돌려 준다는 것을 알 수 있습니다.

이 일련의 체크 흐름을 고급언어로 표현하면 다음과 같이 됩니다.

체크 루틴을 고급언어로 표현
if (현재연도 < 2006) { return 사용기한내; } else if (현재연도 > 2006) { return 사용기간만료; }

⁵⁶ 이것도 마찬가지로 [현재월 \geq 8]이 아니면 이 비교명령은 실행되지 않기 때문에, [현재월==8]과 바꾸어 해석해도 문제 없습니다.

날짜 제한 Protection 크랙

```
if (현재월 < 8) {  
    return 사용기한내;  
}  
else if (현재월 > 8) {  
    return 사용기간만료;  
}  
  
if(현재일 <= 15) {  
    return 사용기한내;  
}  
else {  
    return 사용기간만료;  
}
```

크랙 패치 작성

사용기한 체크 부분을 앞에서 상세히 설명 하였듯이 반환값을 0 이외의 값으로 만들어 주면 프로그램을 계속 사용할 수 있습니다. 그것도 제일 처음의 년도 체크 부분만을 무조건 참으로 만들면 그 이후의 월/일 체크는 하지 않습니다. 그럼 첫번째 연도 체크 부분을 변경하여 사용기한 체크 부분을 무력화 시키도록 하겠습니다.

그림 10의 화면을 다시 한번 주목해 주십시오.

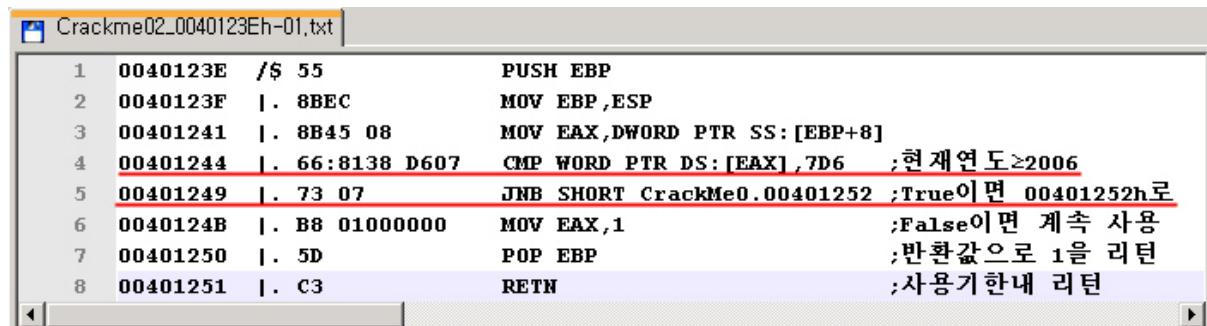


그림 10. 현재연도 체크 부분(1)

명령어 변경을 수행할 경우는 비교/분기명령에 주목하기 바랍니다. 여기서는 00401249h주소의 JNB 명령으로 점프하지 않으면 반환값 1이 되돌아오기 때문에, 일반적으로 [현재연도≥2006]를 [현재연도≥65535]⁵⁷으로 바꿔 줍니다. 이 방법을 염밀히 말하면 [사용기한 무효화]가 아니라 [사용기한 연장]이라고 불러야 하겠지만, 65535년에도 계속 프로그램을 사용하지는 않을 것이므로 특별히 문제는 없을 것입니다.

⁵⁷ 65535는 16비트로 나타낼 수 있는 최대치이며, 이 숫자를 사용한 이유는 CMP 명령이 16비트의 WORD형끼리의 비교이기 때문입니다.

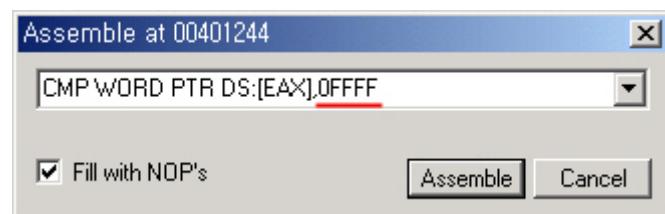


그림 15. 사용기한 65535 년으로 변경

어ドレス 00401244h 행의 3 번째 열을 마우스로 더블 클릭하면 어셈블리 명령을 수정할 수 있기 때문에, 비교대상의 연도 값 7D5(2006년)를 0FFFF(65535년)로 바꿔 적습니다(그림 15). 변경하고 난 후 전체적인 첫 번째 연도 체크 부분은 아래와 같습니다.

```

Crackme02_0040123Eh-02.txt
1 0040123E /$ 55      PUSH EBP
2 0040123F | . 8BEC    MOV EBP,ESP
3 00401241 | . 8B45 08  MOV EAX,DWORD PTR SS:[EBP+8]
4 00401244 | . 66:8138 FFFF  CMP WORD PTR DS:[EAX],0FFF ;현재연도 ≥ 65535
5 00401249 | . 73 07    JNB SHORT CrackMe0.00401252 ;True이면 00401252h로
6 0040124B | . B8 01000000  MOV EAX,1           ;False이면 계속 사용
7 00401250 | . 5D       POP EBP            ;반환값으로 1을 리턴
8 00401251 | . C3       RETN              ;사용기한내 리턴

```

그림 16. 사용기한 변경

그러나 현시점에서는 메모리상에 전개된 명령 코드를 수정한 것에 지나지 않기 때문에, 실행 파일에 그 변경 부분을 반영시킬 필요가 있습니다. 역 어셈블 원도우상에서 【마우스 오른쪽 클릭→Copy to executable→All modifications】을 선택하면, 그림 17에 보이는 원도우가 표시됩니다. “Copy All”을 선택하면, 실행파일의 역 어셈블 리스트가 표시됩니다(그림 18). 이 원도우상에서 【마우스 오른쪽 클릭→Save file】을 선택하면 명령어 코드 수정이 완료된 실행 파일을 보존할 수 있습니다. 적당한 이름 “CrackMe02_patched.exe”로 저장합시다.

날짜 제한 Protection 크랙

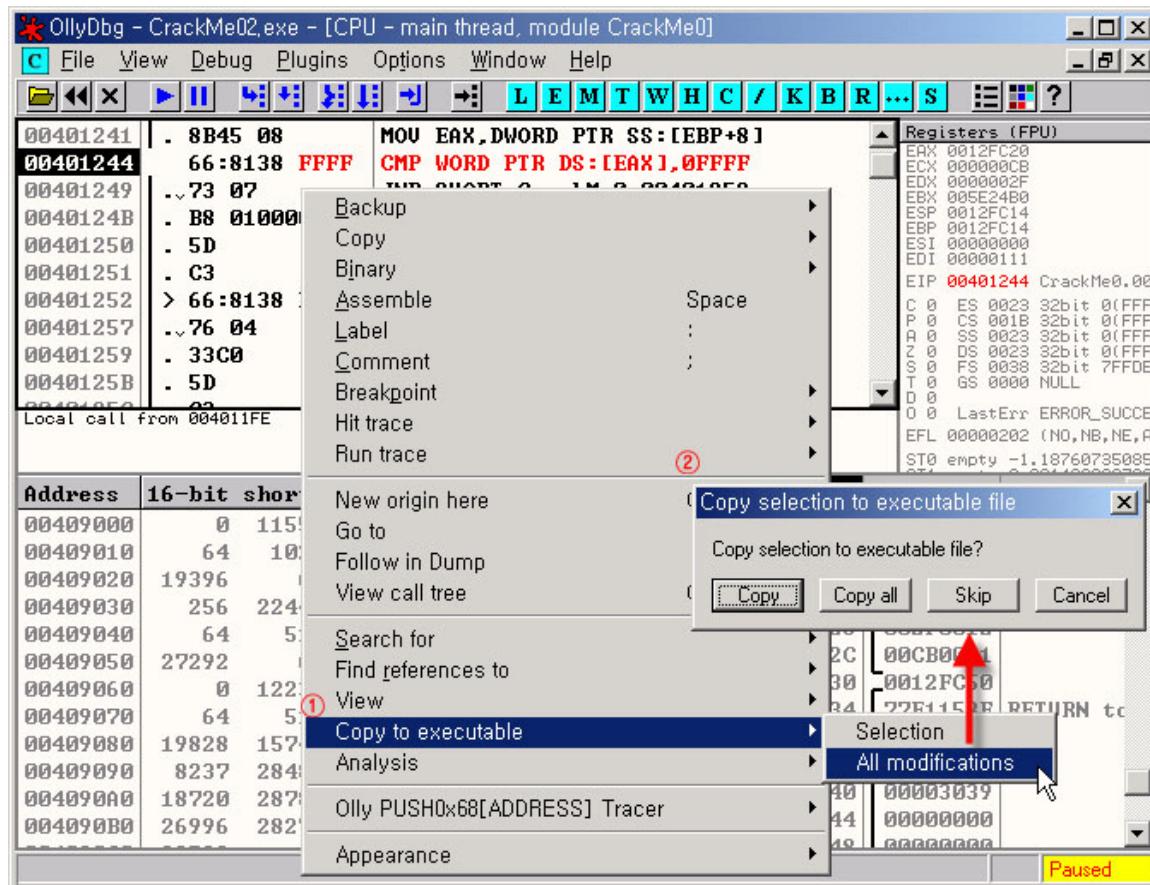


그림 17. 실행파일로 저장

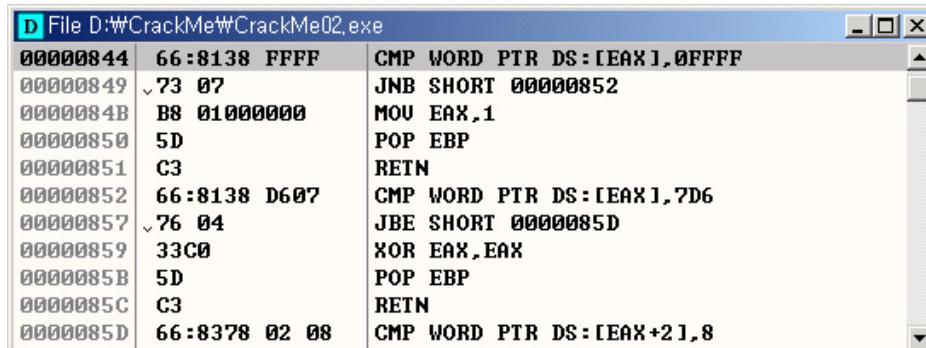


그림 18. 실행파일의 역 어셈블 리스트(메모리가 아닌)

만약 FCJ, ASK, BMT, Inline 형식으로 정보를 제공하려고 한다면, 아래와 같은 텍스트 형태로 제공될 수 있습니다.

```

FILENAME: Crackme02.exe
MD5: 4d18347575eb1474f46e7b60f6aaadae
EBP+0x8: 0x00401000h
EBP-0xC: 0x77EA216C
EBP-0x8: 0x77E52B18
00000847:D5 FF
00000848:07 FF

```

에러 메시지 참조하기

일반적으로 소프트웨어가 사용기한이 만료된 경우 정품 등록권유나 만료표시를 나타내는 메시지가 표시됩니다. 또한 가짜 등록 번호의 입력 때도 “정품 등록 번호가 다릅니다”라는 메시지가 표시됩니다. 이 메시지는 사용자에게 소프트웨어 사용에 대한 정보를 제공해 주고 있지만, 크랙이나 리버싱을 목적으로 하는 사람에게는 매우 유용한 정보가 될 수 있습니다.

OllyDbg의 역 어셈블 원도우로부터 【마우스 오른쪽 클릭→Search for→All referenced text strings】을 선택하거나, OllyDbg 플러그인 “Ultra String Reference”를 이용하여 문자열 검색을 하게 되면, 명령 코드가 참조하고 있는 전체 문자열이 표시됩니다(그림 19). 이 목록으로부터 문자열이 참조하고 있는 명령 코드로 점프할 수 있기 때문에, 그 부분을 시작으로 역추적하여 체크 루틴을 확인할 수 있습니다.

이러한 공격을 막으려면, 문자열을 암호화한 후 필요한 부분에서 복호화 하는 방법으로 대처할 수 있습니다. 또한 체험판과 정식 등록 버전판을 별도로 제작하여 체험판 크랙을 통해 정식 등록 버전으로 사용하지 못하도록 원천 봉쇄하는 방법도 사용할 수 있습니다.

Address	Disassembly	Text String
00401000	JMP SHORT CrackMe0.0040E	(Initial CPU selection)
0040120A	PUSH CrackMe0.00409151	CrackMe02
0040120F	PUSH CrackMe0.0040912C	2006년 8월15일까지 실행할수 있습니다
004013E8	PUSH CrackMe0.00409164	borlndmm
004013FA	PUSH CrackMe0.0040916D	hrdir_b.c: LoadLibrary != mmdll borlndmm failed
00401426	PUSH CrackMe0.0040919D	borlndmm
0040143A	PUSH CrackMe0.004091A6	@Borlndmm@SysGetMem\$qqpri
00401447	PUSH CrackMe0.004091BF	@Borlndmm@SysFreeMem\$qqrpv
00401454	PUSH CrackMe0.004091DA	@Borlndmm@SysReallocMem\$qqrpvi
00402543	PUSH CrackMe0.004092EA	xxtyle.cpp
00402548	PUSH CrackMe0.004092D3	IS_STRUCT(base->tpMask)
00402563	PUSH CrackMe0.0040930C	xxtyle.cpp
00402568	PUSH CrackMe0.004092F5	IS_STRUCT(dev->tpMask)
00402583	PUSH CrackMe0.0040933D	xxtyle.cpp

그림 19. Ultra String Reference 검색 결과

제 3 장 Windows 크랙 기본편

3.6 OllyDbg 플러그인 제작

들어가며

OllyDbg 는 앞서 설명한 플러그인들을 자체 제작하여 OllyDbg 프로그램 내부에서 자유롭게 사용할 수 있습니다. OllyDbg 는 공개된 버전만으로도 충분히 많은 기능을 수행할 수 있는 뛰어난 디버거이지만, 여기에 플러그인 제작 기능으로 툴을 확장할 수 있다는 점 때문에 오랜 기간 사용될 수 있는 것 같습니다. 그러나 많은 사람들은 누군가가 작성한 플러그인을 사용하는 데 그치고 있다고 생각됩니다. 플러그인 기능을 제작하는 것은 매우 어렵다고 생각할 수 있지만, 유용하다고 생각되는 기능을 가진 플러그인을 작성하여, 많은 OllyDbg 사용자에게 제공했으면 하는 바랍니다. 또한 본인이 다른 사람의 플러그인을 유용하게 사용했다면 이러한 기능을 제작해 준 사람에게 고마움을 표하는 것도 기본이라 생각됩니다.

플러그인 개발 도구(PDK: Plug-in Development Kit)

플러그인 작성을 위해서는 Plug-in Development Kit(PDK)라는 플러그인 작성 키트가 필요합니다. 이 PDK 안에는 플러그인에서 사용하는 함수의 원형(Prototype)이나 구조체의 헤더파일, 함수의 Import 라이브러리, 플러그인의 샘플 프로그램 등이 들어 있습니다. 공식 PDK는 Borland 나 Microsoft 의 C/C++ 컴파일러로 작성하여, C 언어를 대상으로 하고 있지만, 다른 사용자에 의해 아래와 같은 컴파일러/어셈블러로 포팅된 PDK가 만들어져 있습니다.

- Delphi6/7
- C++
- MASM32

컴파일 주의사항

OllyDbg 와 플러그인 사이의 통신을 원활하게 하려면 컴파일시 지켜야 할 주의사항이 있습니다.

- ① 모든 플러그인 Callback 함수는 서수(ordinal)가 아닌 이름으로 Export 할 것
- ② C++컴파일러 사용시 모든 callback 함수 위의 mangling 이름을 disable 할 것(extern “C”로 선언)
- ③ 모든 플러그인의 API 와 callback 함수 인자 전달은 표준 C 스타일로 할 것(cdecl로 선언)
- ④ “plugin.h”내 정의된 모든 OllyDbg 구조체를 BYTE 계열로 할 것(컴파일러 옵션: VC /Zpl, BCC -al)
- ⑤ 기본 문자형(character type)은 unsigned 로 할 것(컴파일러 옵션: VC /J, BCC -K)

콜백 함수(callback function)

OllyDbg 와 플러그인 사이를 연결하는 것이 콜백(callback) 함수입니다. OllyDbg 측에서 일어나는 여러 가지 이벤트에 대응하여 이 콜백 함수가 호출됩니다. 함수중에 ODBG_Plugindata 와 ODBG_Plugininit 의 2 개는 필수로, 나머지 함수는 옵션(필요할 때만 작성)으로 되어 있습니다. OllyDbg 는 지정된 플러그인 폴더의 DLL 을 검색하여 로드(load) 한 후 이 콜백 함수를 찾아서 플러그인을 설치합니다.

ODBG_Plugindata

필수 콜백 함수입니다. 이 함수가 없으면 플러그인은 설치되지 않습니다. 인수 shortname 은 NULL 을 포함하여 32 문자 이내의 플러그인 이름을 정의합니다. Shortname 에서 정의한 이름은 OllyDbg 의 메인 메뉴의 [Plugins]에 나타나는 메뉴 항목 이름으로 사용됩니다. 그렇기 때문에 10 자 내외로 가독성 있게 정의할 필요가 있습니다. 반환값으로는 plugin.h 내에서 정의되어 있는 PLUGIN_VERSION 값을 지정합니다. OllyDbg 는 이 값을 확인하여 버전이 맞지 않을 경우 플러그인이 설치되지 않습니다.

```
int ODBG_Plugindata(
    char shortname[32]      //플러그인 이름을 정의하는 32 문자(NULL 문자 포함) 문자열
);
```

ODBG_Plugininit

이것도 필수 콜백 함수입니다. 이 함수는 플러그인 초기화 처리를 수행합니다. 인수 ollydbgversion 에는 OllyDbg 의 버전이 들어 있어, 이것과 플러그인 버전을 비교하여 플러그인의 호환성을 체크합니다. 초기화가 성공하면 OllyDbg Log 윈도우에 플러그인명과 버전 등을 출력해 두면 좋을 것입니다. 처리가 성공하면 0 을 돌려주고, 실패하면 -1 을 되돌려주도록 합니다.

```
int ODBG_Plugininit(
    int ollydbgversion,      // OllyDbg 의 버전
    HWND hw,                // OllyDbg 의 메인 윈도우의 핸들
    DWORD *features         // 확장을 위해 예약(미사용)
);
```

3 번째 인수 features 는 확장을 위해 미리 예약 되어 있으나, OllyDbg v1.10 이 최종 버전으로 PDK 업데이트도 되지 않고 있으므로 현재는 사용되지 않습니다. 아마도 OllyDbg v2.0 에서는 사용될 것 같습니다.

ODBG_Pluginmainloop

OllyDbg 의 메인 루프를 통과할 때마다 이 함수가 호출되므로 주기적인 처리를 수행시키는데 사용됩니다. 디버거의 동작과는 관계없이 호출되므로 주의가 필요합니다. 이처럼 항상 처리되는 구조이기 때문에 일반적으로 디버거가 정지상태일 때나 플러인이 실행되도록 클릭 하였는지 등, 어떤 조건에서 처리가 되도록 하여야 합니다. 또한 조건판단을 매번 수행하기 때문에 전체적으로 처리속도에 많은 영향을 주게 됩니다. 그러므로 무턱대고 이 함수를 Export 하는 것은 피해야 합니다.

인수 debugevent 는 WaitForDebugEvent API 에서 얻어지는 DEBUG_EVENT 구조체의 포인터입니다. 이벤트가 없을 경우 NULL 이 들어갑니다.

```
void ODBG_Pluginmainloop(
    DEBUG_EVENT *debugevnt // WaitForDebugEvent 에 의해 설정되는 DEBUG_EVENT 구조체 포인터
);
```

ODBG_Pluginsaveudd

OllyDbg 는 디버그 세션 정보를 .udd 파일에 저장하고 있습니다. 현재 디버깅 세션에서 브레이크 포인트나 분석상태 등을 보존할 수 있으나, 플러그인으로 디버거 세션에 고유 정보를 저장하고 싶은 경우에는 이 콜백 함수를 Export 합니다. 실제로 데이터를 저장하려면 Pluginsaverecord 함수를 사용합니다.

```
void ODBG_Pluginsaveudd(
    t_module *pmod, // 모듈 기술자 t_module 구조체 포인터
    int ismainmodule // 디버거 메인 모듈인지 아닌지를 나타내는 플래그
);
```

ODBG_Pluginuddrecord

플러그인이 .udd 파일에 저장한 데이터를 받아오기 위해 Export 하는 콜백 함수입니다. OllyDbg 는 .udd 파일을 읽어 들어 인식할 수 없는 레코드가 존재하면, 이 함수를 호출합니다. 인수 tag 에 의해 그 레코드가 플러그인에 포함된 것인지를 확인하여, 포함된 경우는 데이터를 처리하여 1 을 돌려주고 속하지 않는 경우는 다른 플러그인에 레코드를 건네기 위해 0 을 되돌려 줍니다.

```
int ODBG_Pluginuddrecord(
    t_module *pmod, // 모듈 기술자 t_module 구조체에의 포인터
```

OllyDbg 플러그인 제작

```
int ismainmodule,           // 디버거 메인 모듈인지 아닌지를 나타내는 플래그
      ulong tag,           // 레코드가 속하는 플러그인 식별자
      ulong size,           // 데이터 사이즈
      void *data            // 레코드의 바이너리 데이터 포인터
);
```

ODBG_Pluginmenu

OllyDbg 의 메인 메뉴나 오른쪽 클릭 팝업 메뉴에 플러그인의 메뉴를 추가할 경우에 Export 할 콜백 함수입니다. 메뉴를 열 때마다 호출되기 때문에 프로그램 중에서 메뉴 항목의 변경이 가능합니다. 인수 item 에는 오른쪽 클릭 팝업 메뉴의 메인 윈도우에 해당하는 데이터(많게는 구조체의 포인터)가 들어 있어, 필요하다면 이것들을 확인하여 메뉴 항목을 변경하는 등의 처리를 수행합니다. 호출원에 대응하는 윈도우 코드와 item 의 캐스트형 관계를 표 1에 나타내었습니다.

origin value	호출원 윈도우	item 형 캐스트
PM_MAIN	OllyDbg 메인	NULL
PM_DUMP	Dump	(t_dump *)
PM_MODULES	Modules	(t_module *)
PM_MEMORY	Memory	(t_memory *)
PM_THREADS	Threads	(t_thread *)
PM_BREAKPOINTS	Breakpoints	(t_bpoint *)
PM_REFERENCES	References	(t_ref *)
PM_RTRACE	Run trace	(int *)
PM_WATCHES	Watches	1로 시작하는 인덱스
PM_WINDOWS	Windows	(t_window *)
PM_DISASM	CPU Disassembler	(t_dump)
PM_CPUTDUMP	CPU Dump	(t_dump)
PM_CPUSTACK	CPU Stack	(t_dump)
PM_CPUREGS	CPU Registers	(t_reg *)

표 1. 호출원 윈도우에 대응하는 윈도우 코드와 item 유형

메뉴를 설정한 경우는 1을 돌려주고 설정하지 않은 경우는 0을 되돌려 주도록 합니다.

```
int ODBG_Pluginmenu(
    in origin,           // 호출원 윈도우의 코드
```

OllyDbg 플러그인 제작

```
char data[4096],           // 메뉴 항목을 나타내는 문자열  
void *item                // 호출원 윈도우에 대응한 데이터의 포인터  
);
```

ODBG_Pluginaction

ODBG_Pluginmenu에서 작성된 메뉴가 실행되었을 때에 응답 처리를 수행할 Export 콜백 함수입니다. 인수 action에는 ODBG_Pluginmenu에서 지정한 메뉴 항목에서 처음에 기입한 식별자가 들어있기 때문에, 호출원 윈도우의 식별과 action 식별에 의해 실행하는 처리를 나누도록 코딩 합니다. 많은 플러그인이 메뉴와 액션 관계로 처리를 수행하게 될 것입니다.

```
void ODBG_Pluginaction(  
    int origin,           // 호출원 윈도우 코드  
    int action,            // ODBG_Pluginmenu에서 지정한 메뉴 항목 식별자  
    void *item             // 호출원 윈도우에 대응한 데이터 포인터  
);
```

ODBG_Pluginshortcut

단축키를 설정하고 싶은 경우에 Export 할 콜백 함수입니다. OllyDbg 표준 윈도우가 처리하지 않는 키 조합(단축키)이 눌려질 때마다 호출됩니다. 또한 이 함수는 통상 두 번 호출됩니다. 처음에는 OllyDbg 메인 윈도우로부터 글로벌 단축키가 됩니다. 두번째 호출은 키보드 포커스를 가지는 MDI 자 윈도우(pane 포함)로부터의 호출입니다.

단축키는 제한적인 리소스이며, 또한 다른 플러그인이나 기타 외부툴에 의해 새로운 단축키를 추가할 수 있기 때문에 주의가 필요합니다. 그래서 항상 설정된 단축키가 이미 존재하는지 검토할 것을 권장합니다. 단축키를 받아서 어떤 처리를 한 경우는 1을 아무것도 하지 않은 경우는 0을 되돌려주도록 합니다. 트레이싱 관련 플러그인이 아니라면 굳이 단축키를 설정하지 않아도 됩니다.

```
int ODBG_Pluginshortcut(  
    int origin,           // 호출원 윈도우 코드  
    int ctrl,              // Ctrl 키의 상태  
    int alt,               // Alt 키의 상태  
    int shift,              // Shift 키의 상태  
    int key,                // 가상 키 코드  
    void *item             // 호출원 윈도우에 대응한 데이터 포인터  
);
```

Ctrl, Alt, Shift 키의 상태는 눌러져 있으면 1, 눌러져 있지 않으면 0입니다.

ODBG_Pluginreset

사용자가 새로운 응용프로그램을 불러온다든지, 아니면 현재 응용프로그램을 재시작 했을 때 호출됩니다. 이 함수가 호출되면 플러그인 내부에서 사용하고 있는 변수나 데이터를 초기상태로 할 필요가 있습니다.

```
void ODBG_Pluginreset(void);
```

ODBG_Pluginclose

OllyDbg 를 종료하려고 했을 때 호출됩니다. 이 함수가 호출되면 플러그인이 작성한 윈도우는 모두 남아있습니다. 이 타이밍에서 플러그인의 인자값을 .ini 등으로 저장하는 것이 가장 좋습니다.

OllyDbg 를 안전하게 종료할 수 있는 경우는 반드시 0 을 리턴 하도록 해야 합니다. 0 이외의 값을 되돌리면 OllyDbg 종료 처리를 멈추게 됩니다. 이 경우 반드시 사용자에게 알림 창으로 이유를 나타내주고 판단을 하도록 해야 합니다.

```
int ODBG_Pluginclose(void);
```

ODBG_Plugindestroy

OllyDbg 가 종료할 때 호출됩니다. 이 시점에서는 이미 플러그인이 작성한 모든 MDI (Multiple Document Interface)윈도우는 파기됩니다. 플러그인이 할당한 리소스(메모리나 윈도우 클래스 등)는 모두 해제시킬 필요가 있습니다.

```
void ODBG_Plugindestroy(void);
```

ODBG_Paused

디버거가 일시 정지되어 OllyDbg 의 내부처리가 전부 끝났을 때 호출됩니다. 인수 reason에는 일시 정지가 되었을 때에 어떠한 처리를 한 후 곧 실행을 계속한다는 것을 지정 할 수 있습니다.

- PP_EVENT: 디버그 이벤트에 의한 정지
- PP_PAUSE: 사용자 요청에 의한 정지
- PP_TERMINATED: 어플리케이션 종료

```
int ODBG_Paused(
    int reason,           // 정지이유
    t_reg *reg           // 일시 정지한 원인이 된 스레드 레지스터 포인터 또는 NULL
);
```

ODBG_Pausedex

ODBG_Paused 와 같지만 보다 자세한 정보를 얻을 수 있습니다. ODBG_Paused 에서 나타난 핵심 정지 이유 코드에 추가적인 형태로 아래와 같은 값이 들어있어서 정지 이유를 더욱 자세히 알 수 있습니다.

- PP_BYPROGRAM: 프로그램에 의해 발생된 디버그 이벤트
- PP_INT3BREAK: INT3 브레이크 포인트
- PP_MEMBREAK: 메모리 브레이크 포인트
- PP_HWBREAK: 하드웨어 브레이크 포인트
- PP_SINGLESTEP: 싱글 스텝 트랩
- PP_EXCEPTION: 예외
- PP_ACCESS: 접근 위반
- PP_GUARDED: 보호된 페이지

```
int ODBG_Pausedex(
    int reason,           // 정지이유
    int extdata,          // 예약: 현재는 항상 0
    t_reg *reg,           // 정지한 원인이 된 스레드의 레지스터 포인터 또는 NULL
    DEBUG_EVENT *debugevent // 정지한 원인이 된 디버그 이벤트 구조체 포인터
);
```

ODBG_Paused 와 ODBG_Pausedex 가 양쪽 다 Export 되어 있었을 경우는 ODBG_Pausedex 만 불러 옵니다.

ODBG_Plugincmd

플러그인에 명령을 보내도록 지시된 조건부 기록 브레이크 포인트(Conditional logging breakpoint)에서 디버기(Debuggee)가 정지될 때마다 호출됩니다(OllyDbg에서 단축키 Shift+F4 키).

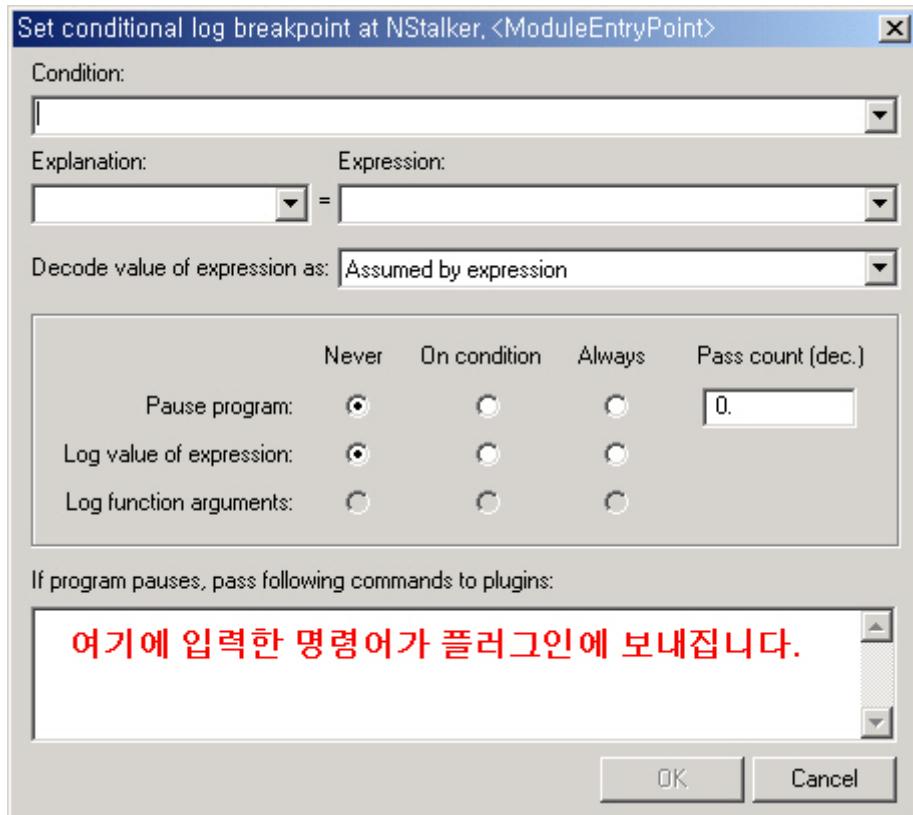


그림 1. 조건부 기록 브레이크 포인트 설정 다이얼로그

명령은 이 함수를 Export 하고 있는 전체 플러그인에 보내지기 때문에 플러그인 측에서 그 명령을 실행해야 할지 아닐지를 판단하지 않으면 안됩니다. 명령이 자신의 것이라고 인식한 경우는 남은 플러그인의 명령 송신을 정지시키기 위해서 1을 되돌려 줍니다. 그렇지 않을 경우 0을 되돌려주지 않으면 안됩니다.

```
int ODBG_Plugincmd(
    int reason,           // 정지한 이유: 현재는 PP_EVENT 만
    treg *reg,            // 정지한 원인이 된 스레드의 레지스터 포인터 또는 NULL
    char *cmd             // NULL로 끝나는 명령 문자열
);
```

플러그인 샘플(Plug-in sample)

플러그인 콜백 함수 전체를 예제로 한 샘플 프로그램을 작성했습니다. 이 샘플은 주로 각 콜백 함수가 호출될 때에 로그 윈도우에 그 뜻을 표시하는 단순한 것입니다.

아래 콜백 함수는 메뉴에서 테스트 표시를 온(On)으로 했을 경우에 로그윈도우 또는 메시지 박스를 출력하도록 하였습니다.

OllyDbg 플러그인 제작

- ODBG_Pluginmainloop
- ODBG_Pausedex
- ODBG_Pluginreset
- ODBG_Pluginclose (MessageBox 표시)
- ODBG_Plugindestroy (MessageBox 표시)

각각의 내용에 대해서는 커맨드를 참조해 주십시오. 또한 커맨드만으로는 잘 모른다고 생각되기 때문에, 실제로 동작시켜서 호출 타이밍 등을 확인해 보시기 바랍니다.

이 샘플은 Borland C++ Compiler 5.5에서 컴파일된 것을 전제로 작성했습니다. 명령 라인에서 컴파일 할 수 있는 환경이 되어 있다면 Make 를 사용할 수 있을 것입니다. Makefile 에는 몇몇 부분의 변경이 필요하므로 내용을 확인하여 사용해 주십시오. 컴파일 환경에 대해서는 레퍼런스나 Web 상의 정보 등을 참고해 주시기 바랍니다.

```
#include<windows.h>

// plugin SDK(PDK)의 헤더 파일
// PDK 패스를 적절하게 변경(절대 디렉터리로 변경)
#include "c:\bc55\Wolly_psdk\Wplugin.h"

// 버전 정보 표시용 매크로
#define PNAME "Sample" // 플러그인명
#define PVERS "0.01.112" // 플러그인 버전
#define ANAME "DkDn" // 제작자

HINSTANCE hinst; // DLL 인스턴스
HWND      hwnd; // OllyDbg 메인 윈도우 핸들

// 플러그인 테스트용 플래그
int PluginmainloopTest;
int PausedexTest;
int PlugincloseTest;
int PlugindestroyTest;
int PluginresetTest;

BOOL WINAPI DllEntryPoint(HINSTANCE hi, DWORD reason, LPVOID reserved)
{
```

OllyDbg 플러그인 제작

```
if(reason==DLL_PROCESS_ATTACH)

    hinst=hi;

    return 1;

}

//***** //

// 플러그인 존재를 알리는 콜백 함수(필수)

//***** //

extc int _export cdecl ODBG_Plugindata(char shortname[32])

{

    strcpy(shortname, PNAME);           // 플러그인 명을 OllyDbg 에 알린다

    return PLUGIN_VERSION;             // PDK 버전을 되돌려준다.

}

//***** //

// 플러그인 초기화 처리를 수행하는 콜백 함수(필수)

//***** //

extc int _export cdecl ODBG_Plugininit(int ollydbgversion, HWND hw, ulong *features)

{

    // PDK 버전과 OllyDbg 버전을 비교한다.

    // OllyDbg 의 버전이 낮으면 -1 을 되돌려주고 플러그인 설치를 중단한다.

    if(ollydbgversion<PLUGIN_VERSION) {

        return -1;

    }

    // 많은 곳에서 사용되는 OllyDbg 메인 윈도우의 핸들을 글로벌 변수로 정의한다.

    hwmain=hw;

    // 사용자의 초기화 처리를 여기에 적는다

    // 테스트용 플래그의 초기화

    // 사용자가 지정했을 때만 결과를 표시하도록 0 으로 초기화한다.

    PluginmainloopTest = 0;

    PausedexTest = 0;

    PlugincloseTest = 0;

    PlugindestroyTest = 0;

    PluginresetTest = 0;
```

```

// 플러그인에 관한 정보를 로그에 표시
Addtolist(0,0,PNAME" v%s", PVERS);

// 로그윈도우에 상황을 출력(OllyDbg 의 Alt+L 단축키)
Addtolist(0,1,"ODBG_Plugininit in Sample - Init OK!");

// OllyDbg 에 성공을 리턴
return 0;
}

//-----
// 디버그 Event Code 에 대응하는 Event 명 문자열
//-----

const char *DbgEvent[64] = {
    "Dummy",
    "Exception Debug Event",
    "Create Thread Debug Event",
    "Create Process Debug Event",
    "Exit Thread Debug Event",
    "Exit Process Debug Event",
    "Load DLL Debug Event",
    "Unload DLL Debug Event",
    "Output Debug String Debug Event",
    "Rip Event"
};

//*****
// OllyDbg 의 메인 루프를 통과할 때마다 호출되는 콜백 함수
//*****

extc void _export cdecl ODBG_Pluginmainloop(DEBUG_EVENT *debugevent)
{
    t_thread *pthread;
    int code;
    DWORD eip;
}

```

OllyDbg 플러그인 제작

```
// 테스트용 플래그가 Set 되거나 디버그 이벤트가 발생해 있으면
if(PluginmainloopTest && debugevent) {

    //현재의 인스트럭션 포인터(EIP 값)를 획득한다.
    pthread = Findthread(Getcputhreadid( ));

    if(pthread) {

        eip = pthread->reg.ip;

    }

    else {

        eip = 0;

    }

    // 디버그 이벤트 코드에서 디버그 이벤트의 종류를 조사하여, 로그 윈도우에 출력
    code = debugevent->dwDebugEventCode;
    Addtolist(eip,1,"ODBG_Pluginmainloop in Sample - code: %d %s", code, DbgEvent[code]);

}

}

//-----
// udd 파일에 저장할 데이터 식별자
//-----
#define TAG_SAMPLE 0x4C494645L

//*****
// 디버그 세션 파일(.udd)에 데이터를 저장하는 콜백 함수
//*****


extc void _export cdecl ODBG_Pluginsaveudd(t_module *pmod, int ismainmodule)
{

    t_thread *pthread;
    DWORD eip;

    // 메인 모듈의 udd 파일에만 저장한다.
    if(ismainmodule == 0) {

        return;
    }

    if(Getstatus() == STAT_STOPPED) {
        pthread = Findthread(Getcputhreadid());
```

OllyDbg 플러그인 제작

```
eip = pthread->reg.ip;

if(Pluginsaverecord(TAG_SAMPLE, sizeof(DWORD), &eip)) {
    Addtolist(0,1, "ODBG_Pluginsaveudd in Sample - current eip %08x is saved to udd file. Tag=%08x", eip,
TAG_SAMPLE);
}

else {
    Addtolist(0, 1, "ODBG_Pluginsaveude in Sample - saving to udd file failed");
}

}

//*****
// ODBG_Pluginsaveudd에서 저장된 레코드를 읽어내는 콜백 함수
//*****

extc int _export cdecl ODBG_Pluginuddrecord(t_module *pmod, int ismainmodule, ulong tag, ulong size, void
*data)
{
    DWORD eip;

    if(ismainmodule == 0) {           // 메인 파일에 저장되어 있는 것은 아닌가?
        return 0;
    }

    if(tag != TAG_SAMPLE) {          // 이 플러그인의 데이터가 아닌가?
        return 0;
    }

    eip = *(DWORD*)data;
    Addtolist(eip,1,"ODBG_Pluginuddrecord in Sample - Last session was closed at %08x" ,eip);
    return 1;
}

//*****
// 플러그인 메뉴를 작성하는 콜백 함수
//*****



extc int _export cdecl ODBG_Pluginmenu(int origin, char data[4096], void *item)
{
    switch(origin) {
```

OllyDbg 플러그인 제작

```
case PM_MAIN:
    // 메뉴 항목을 정하는 문자열을 data에 보관한다.
    // 메뉴의 표시상태를 알 수 있도록 행을 나누고 있으나 전체가 하나의 문자열
    strcpy(data, "ODBG_Pluginmainloop test{\"0 O&N,\" \"1 O&FF\" \"}"
           "ODBG_PausedException test{\"2 O&N,\" \"3 O&FF\" \"}"
           "ODBG_Pluginreset test{\"4 O&N,\" \"5 O&FF\" \"}"
           "ODBG_Pluginclose test{\"6 O&N,\" \"7 O&FF\" \"}"
           "ODBG_Plugindestroy test{\"8 O&N,\" \"9 O&FF\" \"}"
           "#63 \"PNAME\" v\"PVERS);"

    return 1;
default: return 0;
}

}

//*****
// ODBG_Pluginmenu에서 설정된 항목에 대한 액션을 실행하는 콜백 함수
//*****



extern void _export cdecl ODBG_Pluginaction(int origin, int action, void *item)
{
    switch(origin) {
        case PM_MAIN :
            switch (action){
                case 0:      // ODBG_Pluginmainloop 의 테스트플래그 ON
                    PluginmainloopTest = 1;
                    break;
                case 1:      // ODBG_Pluginmainloop 의 테스트플래그 OFF
                    PluginmainloopTest = 0;
                    break;
                case 2:      // ODBG_PausedException 의 테스트플래그 ON
                    PausedexTest = 1;
                    break;
                case 3:      // ODBG_PausedException 의 테스트플래그 OFF
                    PausedexTest = 0;
                    break;
                case 4:      // ODBG_Pluginreset 의 테스트플래그 ON
                    PluginresetTest = 1;
                    break;
            }
    }
}
```

OllyDbg 플러그인 제작

```
break;

case 5: // ODBG_Pluginreset 의 테스트플래그 OFF
    PluginresetTest = 0;
    break;

case 6: // ODBG_Pluginclose 의 테스트플래그 ON
    PlugincloseTest = 1;
    break;

case 7: // ODBG_Pluginclose 의 테스트플래그 OFF
    PlugincloseTest = 0;
    break;

case 8: // ODBG_Plugindestroy 의 테스트플래그 ON
    PlugindestroyTest = 1;
    break;

case 9: // ODBG_Plugindestroy 의 테스트플래그 OFF
    PlugindestroyTest = 0;
    break;

default: break;
}

break;

default: break;
}

}

//*****
// 콜백 함수의 인수에 설정되는 origin 의 변환문자열
// ODBG_Pluginshortcut 로 사용
//*****

const char *orgname[ ] = {
    "Main Window",                      // 0
    "d", "d", "d", "d", "d", "d", "d", "d", // 1~9 (더미)
    "Dump Window",                     // 10
    "Modules Window",                  // 11
    "Memory Window",                   // 12
    "Threads Window",                 // 13
    "Breakpoints Window",             // 14
    "References Window",              // 15
};
```

OllyDbg 플러그인 제작

```
"Run Trace Window",           //16
"Watch Window",              //17
"Windows Window",            //18
"d",           //19~30(더미)
"CPU Disassembler",          //31
"CPU Dump",                  //32
"CPU Stack",                  //33
"CPU Registers"             //34
};

//*****
// 단축키를 읽어 들이는 콜백 함수
// 단축키 내용을 확인하기 위해 로그윈도우에 출력
//*****

extc int _export ODBG_Pluginshortcut(int origin, int ctrl, int alt, int shift, int key, void *item)
{
    Addtolist(0, 1, "ODBG_Pluginshortcut in sample - Ctrl=%d Alt=%d Shift=%d Key=%X[%c] from %s", ctrl, alt,
shift, key, key, orgname[origin]);

// 1 을 반환하면 그 단축키는 사용되었다는 것을 나타내므로 다른 플러그인에서는 사용할 수 없습니다.
// 여기서는 테스트를 위해 0 을 되돌려 보내는 것으로 단축키를 처리하지 않은 것으로 합니다.
    return 0;
}

//*****
// 디버거(Debuggee)에서 새로운 플러그인을 읽어 들이기 위해 재시작 되었을 때 호출되는 콜백 함수.
//*****


extc void _export ODBG_Pluginreset(void)
{
    if(PluginresetTest == 1) {
        Addtolist(0,1,"ODBG_Pluginreset in Sample - New file loaded or current file restarted");
    }
}

//*****
// OllyDbg 가 닫혀지려고 할 때 호출되는 콜백 함수
```

OllyDbg 플러그인 제작

```
//*****  
extc int _export ODBG_Pluginclose(void)  
{  
    if(PlugincloseTest == 1) {  
        MessageBox(hwmain, "ODBG_Pluginclose in Sample is called", "Sample Plugin", MB_OK);  
    }  
    // 0 이외의 값을 돌려주면 OllyDbg 를 안전하게 닫을 수 없게 되어 OllyDbg 의 종료 시퀀스가 멈춘다.  
    return 0;  
}  
  
//*****  
// OllyDbg 의 종료처리가 끝나고 마지막 순간에 호출되는 콜백 함수  
//*****  
extc void _export ODBG_Plugindestroy(void)  
{  
    if(PlugindestroyTest == 1) {  
        // OllyDbg 의 메인 윈도우 핸들은 이미 파기되어 있어, 모 윈도우 핸들에 NULL 을 지정  
        MessageBox(NULL, "ODBG_Pluginclose in Sample is called", "Sample Plugin", MB_OK);  
    }  
}  
  
//*****  
// 디버거가 어떤 이유로 인해 일시 정지 상태가 되면 호출되는 콜백 함수  
//*****  
extc int _export ODBG_Pausedex(int reason, int extdata, t_reg *reg, DEBUG_EVENT *debugevent)  
{  
    char buf[255];  
    if(!PauseddexTest) {  
        return 0;  
    }  
  
    switch(reason&PP_MAIN) {  
        case PP_EVENT:  
            strcpy(buf, "Debugging event");  
            break;  
    }  
}
```

OllyDbg 플러그인 제작

```
case PP_PAUSE:
    strcpy(buf, "User request");
    break;

case PP_TERMINATED:
    strcpy(buf, "Terminated");
    break;

default: break;
}

if(reason & PP_BYPROGRAM) {
    strcat(buf, " - by program debug event");
}

if(reason & PP_INT3BREAK) {
    strcat(buf, " - int 3 break point");
}

if(reason & PP_MEMBREAK) {
    strcat(buf, " - memory break point");
}

if(reason & PP_HWBREAK) {
    strcat(buf, " - hardware break point");
}

if(reason & PP_SINGLESTEP) {
    strcat(buf, " - single step trap");
}

if(reason & PP_EXCEPTION) {
    strcat(buf, " - exception");
}

if(reason & PP_ACCESS) {
    strcat(buf, " - access violation");
}

if(reason & PP_GUARDED) {
    strcat(buf, " - guarded page");
}

Addtolist(0,1, "ODBG_PausedException in Sample - Paused reason: %s", buf);
return 0;
}
```

```

//*****
// 조건부 기록 브레이크 포인트에 command 가 설정되어 있다면 호출되는 콜백 함수
//*****

extc int _export ODBG_Plugincmd(int reason, t_reg *reg, char *cmd)
{
    // Command 로 “testcmd”를 받아 들인다.

    if(!strcmp(cmd, “testcmd”)) {
        Addtolist(0,1, “ODBG_Plugincmd in Sample – command % 2 is applied”);

        return 1;           // Command 를 받아들인 표시로 1 을 되돌려준다.

    }

    Addtolist(0,1, “ODBG_Plugincmd in Sample – %s is unrecognized command”);

    return 0;           // Command 는 타 플러그인 것으로 0 을 되돌려 다른 곳으로 넘긴다.

}

```

OllyDump 플러그인 소스(OllyDump Plug-in source code)

아래는 OllyDump 라는 플러그인 소스 코드로 디버기중인 프로세스 메모리를 덤프하는데 유용한 플러그인 입니다.
위에서 설명한 샘플 플러그인을 토대로 아래 실제 플러그인 소스에서 구현 방법을 확인하시기 바랍니다.
프로그램내의 중요한 부분은 커멘트를 달아 따로 설명하였습니다.

```

// OllyDump 2.21

// 목적: 디버기 프로세스 메모리 덤프

// 중요사항: 구조체의 Byte Alignment 를 사용하여 컴파일, unsigned char 를 사용하여야 함.

#define _WIN32_WINNT 0x0400          // IA-32 WINNT3.5 이상에서 동작 정의
#define _WIN32_WINNT 0x0500          // IA-32 Windows 2000 이상에서 동작 정의

#include <windows.h>
#include <commctrl.h>

#include <odbgWplugin.h>          // OllyDbg PDK 의 플러그인 헤더 파일 위치 지정
#include "resource.h"              // OllyDbg 에서 보여지는 리소스파일 정의

```

```

HINSTANCE hinst;           // DLL 인스턴스
HWND      hwndMain;        // 메인 OllyDbg 윈도우의 핸들
WNDPROC   SecLstDlgProcOrg; // 리스트 컨트롤내 섹션의 원래 윈도우 프로시저

char      strCurEIP[TEXTLEN];
char      szFileName[MAX_PATH]={0},szFile[MAX_PATH]={0},szWorkPath[MAX_PATH]={0};
BOOL     blFixSect,blRebuild;
LPBYTE   DbgePath,DbgeName,lpszSectName;
int      iRebMethod;

LRESULT CALLBACK MainDlgProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK SecLstDlgProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK SecEdtDlgProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK OptDlgProc(HWND, UINT, WPARAM, LPARAM);
BOOL   GetPEInfo(void);
BOOL   SaveDump(HWND);
BOOL   IsValidNumber(char *, int, int);
void   FreeSectInfo(void);
int   FindOEPbySectionHop(int);
DWORD GetCurrentEIP(void);

extern void  SearchImportData(void);
extern void  MakeIID(BYTE *pMemBase, DWORD dwNewSectSize);
extern BOOL  RebuildImport(char *szTargetFile);
extern WORD  GetApiNameOrdinal(char *libname, DWORD ApiAddress, char *ApiName);
extern DWORD rva2offset(DWORD dwRva);
extern DWORD offset2rva(DWORD dwOffset);
extern PIMAGE_SECTION_HEADER rva2section(DWORD dwRva);
extern BYTE RebuildITDeluxe(char *szTargetFile, BYTE byRebuildType);

#define NUM_DEC 1
#define NUM_HEX 2

#define ODP_TRACE_INTO 0
#define ODP_TRACE_OVER 1

```

```

#define PNAME    "OllyDump"           // 프로그램 이름 정의
#define PVERS    "v2.21.108"          // 프로그램 버전 정의
#define ANAME    "Gigapede"           // 프로그램 제작자 정의

char szODPath[MAX_PATH],szODIni[MAX_PATH],szPluginPath[MAX_PATH],szPluginIni[MAX_PATH],*pdest;
BOOL  SearchAnimation;
BOOL  SearchLog;
DWORD AnimationWait;

typedef struct {
    WORD  woNumOfSect;
    DWORD dwImageBase;
    DWORD dwSizeOfImage;
    DWORD dwAddrOfEP;
    DWORD dwBaseOfCode;
    DWORD dwBaseOfData;
} PEFILEINFO, *LPPEFILEINFO;

typedef struct {
    BYTE  byName[IMAGE_SIZEOF_SHORT_NAME];
    DWORD dwVSize;
    DWORD dwVOffset;
    DWORD dwRSize;
    DWORD dwROffset;
    DWORD dwCharacteristics;
} SECTIONINFO, *LPSSECTIONINFO;

PEFILEINFO    PEFileInfo,PEFileInfoWrk;
LPSSECTIONINFO lpSectInfo = NULL;
SECTIONINFO   SectInfoWrk;
BOOL TraceFlag = FALSE;

BOOL WINAPI DllEntryPoint(HINSTANCE hi,DWORD reason,LPVOID reserved) {
    if (reason==DLL_PROCESS_ATTACH)
        hinst=hi;                      // 플러그인 인스턴스 마킹
}

```

OllyDbg 플러그인 제작

```
return 1;                                // 리포트 성공
};

extc int _export cdecl ODBG_Plugindata(char shortname[32]) {
    // 필수 콜백 함수. 이 함수가 없으면 플러그인은 설치되지 않음.
    // 인수 shortname 은 NULL 를 포함하여 32 문자 이내의 플러그인 이름을 정의.

    strcpy(shortname,PNAME);      // 플러그인의 이름
    return PLUGIN_VERSION;
};

extc int _export cdecl ODBG_Plugininit(int ollydbgversion,HWND hw,ulong *features) {
    // ODBG_Plugindata 와 함께 필수 콜백 함수. 이 함수는 플러그인 초기화 처리를 수행함.
    // 인수 ollydbgversion 에는 OllyDbg 의 버전이 들어 있어, 이것과 플러그인 버전을 비교하여
    // 플러그인의 호환성을 체크.
    // 초기화가 성공하면 0 을 되돌려주고, 실패하면 -1 을 되돌려 줌.

    if(ollydbgversion<PLUGIN_VERSION) {
        return -1;
    }
    hwmain=hw;

    // 플러그인 옵션 및 사용내역 기록
    // 플러그인 패스\플러그인이름.ini 파일로 기록됨(eg. pluginWollydump.ini)
    // [OPTIONS]
    // Search Animation=1
    // Animation Wait=60
    // Search Log=1

    GetModuleFileName(NULL, szODPath, MAX_PATH);
    pdest = strrchr(szODPath, '\\');
    pdest[1] = 'W';
    wsprintf(szODIni,"%sollydbg.ini",szODPath);
    GetPrivateProfileString("History","Plugin path",szODPath,szPluginPath,sizeof(szPluginPath),szODIni);
    wsprintf(szPluginIni,"%s\%s.ini",szPluginPath,PNAME);
    SearchAnimation = GetPrivateProfileInt("OPTIONS", "Search Animation", 0, szPluginIni);
```

OllyDbg 플러그인 제작

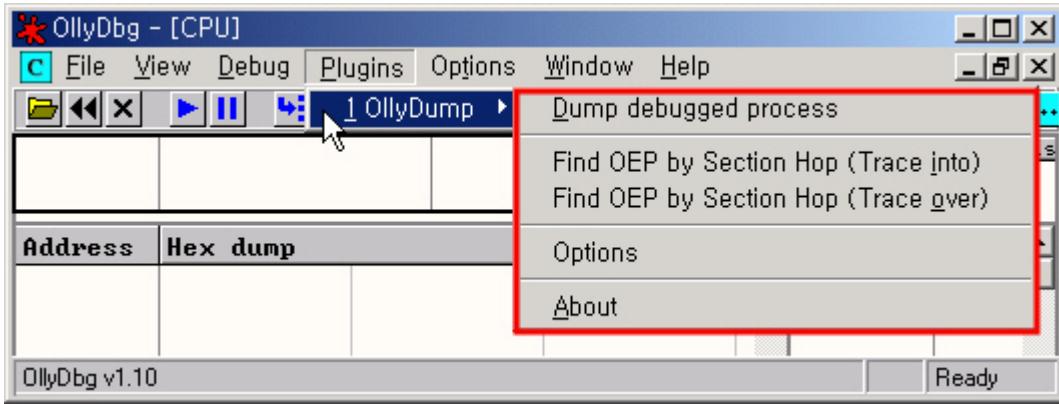
```
SearchLog      = GetPrivateProfileInt("OPTIONS", "Search Log"      , 0, szPluginInI);
AnimationWait   = GetPrivateProfileInt("OPTIONS", "Animation Wait"   , 60, szPluginInI);

Addtolist(0, 0,PNAME " " PVERS " by " ANAME);

return 0;

};

extc int _export cdecl ODBG_Pluginmenu(int origin,char data[4096],void *item) {
// OllyDbg 의 메인 메뉴나 오른쪽 클릭 팝업 메뉴에 플러그인 메뉴를 추가할 경우에 보여지는 메뉴 설정.
// 메뉴를 열 때마다 호출되기 때문에 프로그램 종에서 메뉴 항목의 변경 가능함.

/*

The screenshot shows the OllyDbg debugger window with the CPU tab selected. The Plugins menu is open, displaying a submenu with four items: 'Dump debugged process', 'Find OEP by Section Hop (Trace &into)', 'Find OEP by Section Hop (Trace &over)', and 'Options'. The 'About' item is also visible at the bottom of the submenu. A red box highlights the entire submenu area.
*/
switch (origin) {
case PM_MAIN:                                // 메인 윈도우의 플러그인 메뉴
    strcpy(data,
        "0 &Dump debugged process|"
        "1 Find OEP by Section Hop (Trace &into),"
        "2 Find OEP by Section Hop (Trace &over)|"
        "50 Options|"
        "63 &About"
    );
    return 1;
case PM_DISASM:
    if(Getstatus() == STAT_NONE) {
        return 0;
    }
    strcpy(data,"0 &Dump debugged process");
}
```

OllyDbg 플러그인 제작

```
return 1;

default:
    break;                                // 다른 윈도우는 종지
};

return 0;                                // 플러그인이 지원되지 않는 Windows
};

extc void _export cdecl ODBG_Pluginaction(int origin,int action,void *item) {
    // ODBG_Pluginmenu에서 작성된 메뉴가 선택 되었을 때에 응답 처리를 수행할 함수 부분.
    // 인수 action에는 ODBG_Pluginmenu에서 지정한 메뉴 항목에서 처음에 기입한 식별자가 들어있기 때문에,
    // 호출원 윈도우의 식별과 action 식별에 의해 실행하는 처리를 나누도록 코딩(Case 0, 1, 2, 50, 63).
    // 많은 플러그인이 메뉴와 액션 관계로 처리를 수행.

    int id;
    char buf[TEXTLEN];

    switch(origin) {
        case PM_MAIN:
        case PM_DISASM:
            switch (action) {
                case 0:                                // 프로세스 덤프 선택
                    if(Getstatus() == STAT_NONE) {
                        MessageBox(hemain,"No process to dump!!",PNAME,MB_OK);
                        return;
                    }
                    if(lpSectInfo) {
                        FreeSectInfo();
                    }
                    GetPEInfo();
                    id = DialogBox(hinst,MAKEINTRESOURCE(IDD_OLLYDUMP),hmain,(DLGPROC)MainDlgProc);
                    if(id == IDOK) {
                        SaveDump(hemain);
                    }
                    FreeSectInfo();
                    break;
            };
    };
}
```

OllyDbg 플러그인 제작

```
case 1:  
    if(Getstatus() == STAT_NONE) {  
        MessageBox(hemain, "No process to monitor!!", PNAME, MB_OK);  
        return;  
    }  
    FindOEPbySectionHop(0);  
    break;  
  
case 2:  
    if(Getstatus() == STAT_NONE) {  
        MessageBox(hemain, "No process to monitor!!", PNAME, MB_OK);  
        return;  
    }  
    FindOEPbySectionHop(1);  
    break;  
  
case 50: // OllyDump 옵션 설정 부분  
    id = DialogBox(hinst,MAKEINTRESOURCE(IDD_OPTIONS),hemain,(DLGPROC)OptDlgProc);  
    if(id == IDOK) {  
        wsprintf(buf, "%d", SearchAnimation);  
        WritePrivateProfileString("OPTIONS", "Search Animation", buf, szPluginInI);  
        wsprintf(buf, "%d", AnimationWait);  
        WritePrivateProfileString("OPTIONS", "Animation Wait" , buf, szPluginInI);  
        wsprintf(buf, "%d", SearchLog);  
        WritePrivateProfileString("OPTIONS", "Search Log" , buf, szPluginInI);  
    }  
    break;  
  
case 63: // 메뉴 아이템 “About”, 플러그인 정보를 화면에 출력  
    MessageBox(hemain,  
              PNAME" "PVERS  
              "Win  
              "by "ANAME" ",  
              "About "PNAME,  
              MB_OK|MB_ICONINFORMATION);  
    break;
```

OllyDbg 플러그인 제작

```
default:  
    break;  
}  
}  
}  
  
extc void _export cdecl ODBG_Pluginreset(void) {  
    // 사용자가 새로운 응용프로그램을 불러온다든지, 아니면 현재 응용프로그램을 재시작 했을 때 호출.  
    // 이 함수가 호출되면 플러그인 내부에서 사용하고 있는 변수나 데이터를 초기상태로 할 필요가 있음.  
  
    if(lpSectInfo) {  
        FreeSectInfo();  
        lpSectInfo = NULL;  
        Addtolist(0,-1,"==%s DEBUG== in ODBG_Pluginreset  Free allocated memory lpSectInfo",PNAME);  
    }  
    return;  
}  
  
extc int _export cdecl ODBG_Pluginclose(void) {  
    // OllyDbg 를 종료하려고 했을 때 호출. 이 함수가 호출되면 플러그인이 작성한 윈도우는 모두 남아있음.  
    // OllyDbg 를 종료할 수 있는 경우 반드시 0 을 리턴.  
  
    if(lpSectInfo) {  
        FreeSectInfo();  
        lpSectInfo = NULL;  
        Addtolist(0,-1,"==%s DEBUG== in ODBG_Pluginclose  Free allocated memory lpSectInfo",PNAME);  
    }  
    return 0;  
}  
  
extc void _export cdecl ODBG_Plugindestroy(void) {  
    // OllyDbg 가 종료할 때 호출. 모든 MDI (Multiple Document Interface) 윈도우는 파기.  
    // 플러그인이 할당한 리소스(메모리나 윈도우 클래스 등)는 모두 해제시켜야 함.  
  
    if(lpSectInfo) {
```

OllyDbg 플러그인 제작

```
FreeSectInfo();

IpSectInfo = NULL;

}

return;
}

extc void _export cdecl ODBG_Pluginmainloop(DEBUG_EVENT *debugevent) {

// OllyDbg 의 메인 루프를 통과할 때마다 수행되는 함수.

// 디버거의 동작과는 관계없이 호출되므로 주의.

t_thread *pthread;

t_status status;

if(TraceFlag) {

    status = Getstatus();

    if(status == STAT_STOPPED) {

        pthread = Findthread(Getcputhreadid());

        if(!pthread) {

            return;
        }

        if(pthread->reg.ip > PEFileInfo.dwImageBase && pthread->reg.ip < PEFileInfo.dwSizeOfImage) {

            if(PEFileInfo.dwAddrOfEP + PEFileInfo.dwImageBase != pthread->reg.ip) {

                Addtolist(0,-1,"EntryPoint      is      %X      and      Original      Entry      Point      may
be %X",PEFileInfo.dwAddrOfEP,pthread->reg.ip);

                TraceFlag = FALSE;
            }
        }
    }
}

BOOL GetPEInfo(void)

{

    int i;

    char msg[TEXTLEN];

    HANDLE hFile,hHeap;
```

OllyDbg 플러그인 제작

```
PIMAGE_DOS_HEADER idosh;
PIMAGE_NT_HEADERS ipeh;
PIMAGE_SECTION_HEADER isechn;
LPBYTE fbuf;
DWORD dwFsiz,dwRsiz;

DbgePath = (char*)Plugingetvalue(VAL_EXEFILENAME);
DbgeName = strrchr(DbgePath,'\\');
memset(szWorkPath,0,sizeof(szWorkPath));
strncpy(szWorkPath,DbgePath,(DbgeName-DbgePath));
DbgeName++;

// 디버거(Debuggee) 읽어오기
hFile = CreateFile(DbgePath,GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);
if(hFile == INVALID_HANDLE_VALUE) {
    wsprintf(msg,"Cannot Create File %s",DbgePath);
    MessageBox(hwmain,msg,PNAME,MB_OK);
    return FALSE;
}
dwFsiz = GetFileSize(hFile,NULL);
hHeap = HeapCreate(HEAP_NO_SERIALIZE,1,0);
fbuf = (char *)HeapAlloc(hHeap, 0, dwFsiz);
if(ReadFile(hFile,fbuf,dwFsiz,&dwRsiz,NULL) == 0) {
    MessageBox(hwmain,"Can't Read File ",PNAME" Error!",MB_OK|MB_ICONEXCLAMATION);
    CloseHandle(hFile);
    HeapFree(hHeap,HEAP_NO_SERIALIZE,fbuf);
    return FALSE;
}
CloseHandle(hFile);

// PE 헤더 정보 수집
idosh = (PIMAGE_DOS_HEADER)fbuf;
if(idosh->e_magic != IMAGE_DOS_SIGNATURE) {
    MessageBox(hwmain,"Bad DOS Signature!!",PNAME,MB_OK | MB_ICONEXCLAMATION);
```

OllyDbg 플러그인 제작

```
    HeapFree(hHeap,HEAP_NO_SERIALIZE,fbuf);

    return FALSE;
}

ipeh = (PIMAGE_NT_HEADERS)(fbuf + idosh->e_lfanew);

if(ipeh->Signature != IMAGE_NT_SIGNATURE) {

    MessageBox(hwmain,"Bad PE Signature!!",PNAME,MB_OK | MB_ICONEXCLAMATION);

    HeapFree(hHeap,HEAP_NO_SERIALIZE,fbuf);

    return FALSE;
}

PEFileInfo.woNumOfSect    = ipeh->FileHeader.NumberOfSections;
PEFileInfo.dwImageBase    = ipeh->OptionalHeader.ImageBase;
PEFileInfo.dwSizeOfImage  = ipeh->OptionalHeader.SizeOfImage;
PEFileInfo.dwBaseOfCode   = ipeh->OptionalHeader.BaseOfCode ;
PEFileInfo.dwBaseOfData   = ipeh->OptionalHeader.BaseOfData ;
PEFileInfo.dwAddrOfEP     = ipeh->OptionalHeader.AddressOfEntryPoint;

lpSectInfo = (LPSECTIONINFO)malloc(sizeof(SECTIONINFO)*(PEFileInfo.woNumOfSect+1));
ZeroMemory(lpSectInfo,sizeof(SECTIONINFO)*(PEFileInfo.woNumOfSect+1));
isech = IMAGE_FIRST_SECTION(ipeh);

for(i=0; i<(int)PEFileInfo.woNumOfSect; i++) {

    strcpy((lpSectInfo+i)->byName,(isech+i)->Name);

    (lpSectInfo+i)->dwVSize          = (isech+i)->Misc.VirtualSize;
    (lpSectInfo+i)->dwVOffset        = (isech+i)->VirtualAddress;
    (lpSectInfo+i)->dwRSize          = (isech+i)->SizeOfRawData;
    (lpSectInfo+i)->dwROffset        = (isech+i)->PointerToRawData;
    (lpSectInfo+i)->dwCharacteristics = (isech+i)->Characteristics;
}

HeapFree(hHeap,HEAP_NO_SERIALIZE,fbuf);

return TRUE;
}

void FreeSectInfo(void)
{
    if(lpSectInfo) {

        free(lpSectInfo);
    }
}
```

OllyDbg 플러그인 제작

```
lpSectInfo = NULL;
}

int FindOEPbySectionHop(int tracemode)
{
    int i;
    DWORD out0,out1,in0,in1,curEIP,curSectVA1,curSectVA2;
    t_reg reg;

    Deleteruntrace();
    TraceFlag = TRUE;
    // 버퍼내 섹션 정보를 비움
    if(lpSectInfo) {
        FreeSectInfo();
    }

    // PE 파일 헤더 값 수집
    GetPEInfo();
    curEIP = GetCurrentEIP();
    Addtolist(0,-1,"EP:%X           ImageBase:%X           SizeOfImage:%X           Current
EIP:%X",PEFileInfo.dwAddrOfEP,PEFileInfo.dwImageBase,PEFileInfo.dwSizeOfImage,curEIP);

    // 엔트리 포인트(Entry Point)에 속한 섹션 검색
    out0 = out1 = 0;
    for(i=0; i<PEFileInfo.wNumOfSect; i++) {
        curSectVA1 = lpSectInfo[i].dwVOffset + PEFileInfo.dwImageBase;
        curSectVA2 = curSectVA1 + lpSectInfo[i].dwVSize;
        if(curEIP >= curSectVA1 && curEIP < curSectVA2) {
            out0 = lpSectInfo[i].dwVOffset + PEFileInfo.dwImageBase;
            out1 = out0 + lpSectInfo[i].dwVSize - 1;
            break;
        }
    }
    if(out0 != 0 && out1 > out0) {
        Settracecondition(NULL,0,0,0,out0,out1);
        Addtolist(0,-1,"Current EIP(%08X) is in Section%02d  %08X - %08X",curEIP,i,curSectVA1,curSectVA2);
    }
}
```

OllyDbg 플러그인 제작

```
Addtolist(0,-1,"Trace Condition set out0:%X  out1:%X",out0,out1);
}

else {
    in0 = lpSectInfo[0].dwVOffset + PEFileInfo.dwImageBase;
    in1 = lpSectInfo[PEFileInfo.woNumOfSect-1].dwVOffset + lpSectInfo[PEFileInfo.woNumOfSect-1].dwVSize +
PEFileInfo.dwImageBase;
    Settracecondition(NULL,0,in0,in1,0,0);
    Addtolist(0,-1,"Current EIPW(%08XW) is out of Debuggee image",curEIP);
    Addtolist(0,-1,"Trace Condition set in0:%X  in1:%X",in0,in1);
}

Startruntrace(&reg);

switch(tracemode) { // 트레이스 모드 선택
case ODP_TRACE_INTO:
    Sendshortcut(PM_MAIN,0,WM_KEYDOWN,1,0,VK_F11); // Trace into
    break;
case ODP_TRACE_OVER:
    Sendshortcut(PM_MAIN,0,WM_KEYDOWN,1,0,VK_F12); // Trace over
    break;
}

return TRUE;
}

BOOL SaveDump(HWND hWnd) // 덤프 저장 부분
{
    int i;
    OPENFILENAME ofn;
    HANDLE hFile,hHeap;
    LPBYTE lpDumpData;
    DWORD dwFrom,dwSize,dwAccBytes;
    PIMAGE_DOS_HEADER idosh;
    PIMAGE_NT_HEADERS ipeh;
    PIMAGE_SECTION_HEADER iseh;

    dwFrom = PEFileInfo.dwImageBase;
    dwSize = PEFileInfo.dwSizeOfImage;
```

OllyDbg 플러그인 제작

```
hHeap = HeapCreate(HEAP_NO_SERIALIZE,1,0);
lpDumpData = HeapAlloc(hHeap,HEAP_NO_SERIALIZE | HEAP_ZERO_MEMORY,dwSize);
dwSize = Readmemory(lpDumpData,dwFrom,dwSize,MM_RESTORE);

idosh = (PIMAGE_DOS_HEADER)lpDumpData;
if(idosh->e_magic != IMAGE_DOS_SIGNATURE) {
    MessageBox(hwnd,"Bad DOS Signature!!",PNAME,MB_OK | MB_ICONEXCLAMATION);
    HeapFree(hHeap,HEAP_NO_SERIALIZE,lpDumpData);
    return FALSE;
}

ipeh = (PIMAGE_NT_HEADERS)(lpDumpData + idosh->e_lfanew);
if(ipeh->Signature != IMAGE_NT_SIGNATURE) {
    MessageBox(hwnd,"Bad PE Signature!!",PNAME,MB_OK | MB_ICONEXCLAMATION);
    HeapFree(hHeap,HEAP_NO_SERIALIZE,lpDumpData);
    return FALSE;
}

ipeh->FileHeader.NumberOfSections      = PEFileInfo.woNumOfSect;
ipeh->OptionalHeader.ImageBase        = PEFileInfo.dwImageBase;
ipeh->OptionalHeader.SizeOfImage     = PEFileInfo.dwSizeOfImage;
ipeh->OptionalHeader.BaseOfCode      = PEFileInfo.dwBaseOfCode;
ipeh->OptionalHeader.BaseOfData      = PEFileInfo.dwBaseOfData;
ipeh->OptionalHeader.AddressOfEntryPoint = PEFileInfo.dwAddrOfEP;

isech = IMAGE_FIRST_SECTION(ipeh);
if(bFixSect) {
    for(i=0; i<(int)PEFileInfo.woNumOfSect; i++) {
        strcpy((isech+i)->Name,(lpSectInfo+i)->byName);
        (isech+i)->Misc.VirtualSize = (lpSectInfo+i)->dwVSize;
        (isech+i)->VirtualAddress   = (lpSectInfo+i)->dwVOffset;
        (isech+i)->SizeOfRawData    = (lpSectInfo+i)->dwRSize;
        (isech+i)->PointerToRawData = (lpSectInfo+i)->dwROffset;
        (isech+i)->Characteristics = (lpSectInfo+i)->dwCharacteristics;
    }
}

// 파일 저장 윈도우 설정
memset(szFileName,0,sizeof(szFileName));
```

OllyDbg 플러그인 제작

```
memset(szFile,0,sizeof(szFile));

memset(&ofn, 0, sizeof(OPENFILENAME));

ofn.lStructSize      = sizeof(OPENFILENAME);

ofn.hwndOwner        = hWnd;

ofn.lpstrFilter       = "Executable file (*.exe)\\0*.exe\\0All files (*.*)\\0*.*\\0\\0";

ofn.lpstrFile         = szFileName;

ofn.lpstrFileTitle    = szFile;

ofn.nMaxFile          = MAX_PATH;

ofn.lpstrInitialDir   = szWorkPath;

ofn.Flags             = OFN_OVERWRITEPROMPT | OFN_HIDEREADONLY;

ofn.lpstrDefExt       = "exe";

ofn.lpstrTitle         = "Save Dump to File";



if(GetSaveFileName(&ofn)) {

    hFile   = CreateFile(szFileName,   GENERIC_READ | GENERIC_WRITE, 0, 0, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);

    if(hFile != INVALID_HANDLE_VALUE) {

        SetFilePointer(hFile, 0, 0, FILE_BEGIN);

        WriteFile(hFile, lpDumpData, dwSize, &dwAccBytes, NULL);

        CloseHandle(hFile);

    }

}

HeapFree(hHeap,HEAP_NO_SERIALIZE,lpDumpData);

Broadcast(WM_USER_CHALL,0,0);



if(blRebuild) {

    switch(iRebMethod) {

        case 1:

            RebuildImport(szFileName);

            break;

        case 2:

            RebuildITDeluxe(szFileName,1);

            break;

        default:

            break;

    }

}
```

```

    }

    return 0;
}

LRESULT CALLBACK OptDlgProc(HWND hDlgWnd, UINT msg, WPARAM wp, LPARAM lp)
{
    switch (msg) {

        case WM_INITDIALOG:
            SendMessage(GetDlgItem(hDlgWnd, IDC_CHK_ANIMATION),BM_SETCHECK,(WPARAM)SearchAnimation, 0L);
            SendMessage(GetDlgItem(hDlgWnd, IDC_CHK_SEARCHLOG),BM_SETCHECK,(WPARAM)SearchLog , 0L);
            SetDlgItemInt(hDlgWnd, IDC_EDT_ANIMWAIT, AnimationWait, FALSE);
            break;

        case WM_COMMAND:
            switch (LOWORD(wp)) {
                break;

                case IDOK:
                    SearchAnimation = ((IsDlgButtonChecked(hDlgWnd, IDC_CHK_ANIMATION) == BST_CHECKED) ? 1 : 0);
                    SearchLog      = ((IsDlgButtonChecked(hDlgWnd, IDC_CHK_SEARCHLOG) == BST_CHECKED) ? 1 : 0);
                    AnimationWait  = GetDlgItemInt(hDlgWnd, IDC_EDT_ANIMWAIT, NULL, FALSE);
                    EndDialog(hDlgWnd, IDOK);
                    break;

                case IDCANCEL:
                    EndDialog(hDlgWnd, IDCANCEL);
                    break;
            }
            break;

        default:
            return FALSE;
            break;
    }
    return TRUE;
}

```

```

LRESULT CALLBACK SecLstDlgProc(HWND hList, UINT msg, WPARAM wp, LPARAM lp)
// 리스트 컨트롤 섹션의 윈도우 프로시저

```

```

{

POINT pt;
LV_ITEM item;
LPBYTE stop;
HMENU hMenu,hSubMenu;
int    nItem,id;
char   buf[TEXTLEN];

switch (msg) {

case WM_RBUTTONDOWN:
    pt.x = LOWORD(lp);
    pt.y = HIWORD(lp);
    hMenu = LoadMenu(hinst, MAKEINTRESOURCE(IDM_SECTLIST));
    hSubMenu = GetSubMenu(hMenu, 0);
    ClientToScreen(hList, &pt);
    TrackPopupMenu(hSubMenu, TPM_LEFTALIGN, pt.x, pt.y, 0, hList, NULL);
    DestroyMenu(hMenu);
    break;

case WM_LBUTTONDOWNCLK:
    SendMessage(hList,WM_COMMAND,(WPARAM)IDM_EDITSECT,0);
    break;

case WM_COMMAND:
    switch (LOWORD(wp)) {
        case IDM_EDITSECT:
            nItem = ListView_GetNextItem(hList,(-1),LVNI_ALL|LVNI_SELECTED);

            item.mask      = LVIF_TEXT;
            item.cchTextMax = sizeof(buf);

            item.pszText  = buf;
            item.iItem    = nItem;
            item.iSubItem = 0;
            ListView_GetItem(hList, &item);
            wsprintf(SectInfoWrk.byName,"%s",buf);
    }
}
}

```

```
item.pszText = buf;
item.iItem = nItem;
item.iSubItem = 1;
ListView_GetItem(hList, &item);
SectInfoWrk.dwVSize = strtoul(buf,&stop,16);

item.pszText = buf;
item.iItem = nItem;
item.iSubItem = 2;
ListView_GetItem(hList, &item);
SectInfoWrk.dwVOffset = strtoul(buf,&stop,16);

item.pszText = buf;
item.iItem = nItem;
item.iSubItem = 3;
ListView_GetItem(hList, &item);
SectInfoWrk.dwRSize = strtoul(buf,&stop,16);

item.pszText = buf;
item.iItem = nItem;
item.iSubItem = 4;
ListView_GetItem(hList, &item);
SectInfoWrk.dwROffset = strtoul(buf,&stop,16);

item.pszText = buf;
item.iItem = nItem;
item.iSubItem = 5;
ListView_GetItem(hList, &item);
SectInfoWrk.dwCharacteristics = strtoul(buf,&stop,16);

id = DialogBox(hinst,MAKEINTRESOURCE(IDD_EDITSECT),hList,(DLGPROC)SecEdtDlgProc);
if(id == IDOK) {
    item.mask = LVIF_TEXT;
    item.cchTextMax = sizeof(buf);
```

ollyDbg 플러그인 제작

```
item.pszText = SectInfoWrk.byName;  
item.iItem = nItem;  
item.iSubItem = 0;  
ListView_SetItem(hList, &item);  
  
wsprintf(buf, "%08X", SectInfoWrk.dwVSize);  
item.pszText = buf;  
item.iItem = nItem;  
item.iSubItem = 1;  
ListView_SetItem(hList, &item);  
  
wsprintf(buf, "%08X", SectInfoWrk.dwVOffset);  
item.pszText = buf;  
item.iItem = nItem;  
item.iSubItem = 2;  
ListView_SetItem(hList, &item);  
  
wsprintf(buf, "%08X", SectInfoWrk.dwRSize);  
item.pszText = buf;  
item.iItem = nItem;  
item.iSubItem = 3;  
ListView_SetItem(hList, &item);  
  
wsprintf(buf, "%08X", SectInfoWrk.dwROffset);  
item.pszText = buf;  
item.iItem = nItem;  
item.iSubItem = 4;  
ListView_SetItem(hList, &item);  
  
wsprintf(buf, "%08X", SectInfoWrk.dwCharacteristics);  
item.pszText = buf;  
item.iItem = nItem;  
item.iSubItem = 5;  
ListView_SetItem(hList, &item);  
}
```

```

        break;

    default:
        break;
    }

default:
    break;
}

return (CallWindowProc((WNDPROC)SecLstDlgProcOrg, hList, msg, wp, lp));
}

LRESULT CALLBACK SecEdtDlgProc(HWND hDlgWnd, UINT msg, WPARAM wp, LPARAM lp)
// 에디트 컨트롤 셙션의 윈도우 프로시저
{
    int ichk;
    char buf[TEXTLEN];
    LPBYTE stop;

    switch (msg) {
    case WM_INITDIALOG:
        SetDlgItemText(hDlgWnd, IDC_SE_NAME, SectInfoWrk.byName);
        wsprintf(buf, "%08X", SectInfoWrk.dwVSize);
        SetDlgItemText(hDlgWnd, IDC_SE_VSIZE, buf);
        wsprintf(buf, "%08X", SectInfoWrk.dwVOffset);
        SetDlgItemText(hDlgWnd, IDC_SE_VOFFSET, buf);
        wsprintf(buf, "%08X", SectInfoWrk.dwRSize);
        SetDlgItemText(hDlgWnd, IDC_SE_RSIZE, buf);
        wsprintf(buf, "%08X", SectInfoWrk.dwROffset);
        SetDlgItemText(hDlgWnd, IDC_SE_ROFFSET, buf);
        wsprintf(buf, "%08X", SectInfoWrk.dwCharacteristics);
        SetDlgItemText(hDlgWnd, IDC_SE_CHAR, buf);

        ichk = (SectInfoWrk.dwCharacteristics & IMAGE_SCN_CNT_CODE) ? 1 : 0;
        SendMessage(GetDlgItem(hDlgWnd, IDC_SE_CONTCODE), BM_SETCHECK, (WPARAM)ichk, 0L);
        ichk = (SectInfoWrk.dwCharacteristics & IMAGE_SCN_CNT_INITIALIZED_DATA) ? 1 : 0;
        SendMessage(GetDlgItem(hDlgWnd, IDC_SE_CONTINI), BM_SETCHECK, (WPARAM)ichk, 0L);
    }
}

```

```

ichk = (SectInfoWrk.dwCharacteristics & IMAGE_SCN_CNT_UNINITIALIZED_DATA) ? 1 : 0;
SendMessage(GetDlgItem(hDlgWnd, IDC_SE_CONTUNINI),BM_SETCHECK,(WPARAM)ichk, 0L);

ichk = (SectInfoWrk.dwCharacteristics & IMAGE_SCN_MEM_SHARED) ? 1 : 0;
SendMessage(GetDlgItem(hDlgWnd, IDC_SE_SHARE),BM_SETCHECK,(WPARAM)ichk, 0L);
ichk = (SectInfoWrk.dwCharacteristics & IMAGE_SCN_MEM_EXECUTE) ? 1 : 0;
SendMessage(GetDlgItem(hDlgWnd, IDC_SE_EXEC),BM_SETCHECK,(WPARAM)ichk, 0L);
ichk = (SectInfoWrk.dwCharacteristics & IMAGE_SCN_MEM_READ) ? 1 : 0;
SendMessage(GetDlgItem(hDlgWnd, IDC_SE_READ),BM_SETCHECK,(WPARAM)ichk, 0L);
ichk = (SectInfoWrk.dwCharacteristics & IMAGE_SCN_MEM_WRITE) ? 1 : 0;
SendMessage(GetDlgItem(hDlgWnd, IDC_SE_WRITE),BM_SETCHECK,(WPARAM)ichk, 0L);

break;

case WM_COMMAND:
    switch (LOWORD(wp)) {
        case IDC_SE_CONTCODE:
            if(IsDlgButtonChecked(hDlgWnd, IDC_SE_CONTCODE) == BST_CHECKED) {
                SectInfoWrk.dwCharacteristics |= IMAGE_SCN_CNT_CODE;
            }
            else {
                SectInfoWrk.dwCharacteristics ^= IMAGE_SCN_CNT_CODE;
            }
            wsprintf(buf,"%08X",SectInfoWrk.dwCharacteristics);
            SetDlgItemText(hDlgWnd, IDC_SE_CHAR,buf);
            break;
        case IDC_SE_CONTINI:
            if(IsDlgButtonChecked(hDlgWnd, IDC_SE_CONTINI) == BST_CHECKED) {
                SectInfoWrk.dwCharacteristics |= IMAGE_SCN_CNT_INITIALIZED_DATA;
            }
            else {
                SectInfoWrk.dwCharacteristics ^= IMAGE_SCN_CNT_INITIALIZED_DATA;
            }
            wsprintf(buf,"%08X",SectInfoWrk.dwCharacteristics);
            SetDlgItemText(hDlgWnd, IDC_SE_CHAR,buf);
            break;
    }
}

```

```
case IDC_SE_CONTUNINI:
    if(IsDlgButtonChecked(hDlgWnd, IDC_SE_CONTUNINI) == BST_CHECKED) {
        SectInfoWrk.dwCharacteristics |= IMAGE_SCN_CNT_UNINITIALIZED_DATA;
    }
    else {
        SectInfoWrk.dwCharacteristics ^= IMAGE_SCN_CNT_UNINITIALIZED_DATA;
    }
    wsprintf(buf, "%08X", SectInfoWrk.dwCharacteristics);
    SetDlgItemText(hDlgWnd, IDC_SE_CHAR, buf);
    break;

case IDC_SE_SHARE:
    if(IsDlgButtonChecked(hDlgWnd, IDC_SE_SHARE) == BST_CHECKED) {
        SectInfoWrk.dwCharacteristics |= IMAGE_SCN_MEM_SHARED;
    }
    else {
        SectInfoWrk.dwCharacteristics ^= IMAGE_SCN_MEM_SHARED;
    }
    wsprintf(buf, "%08X", SectInfoWrk.dwCharacteristics);
    SetDlgItemText(hDlgWnd, IDC_SE_CHAR, buf);
    break;

case IDC_SE_EXEC:
    if(IsDlgButtonChecked(hDlgWnd, IDC_SE_EXEC) == BST_CHECKED) {
        SectInfoWrk.dwCharacteristics |= IMAGE_SCN_MEM_EXECUTE;
    }
    else {
        SectInfoWrk.dwCharacteristics ^= IMAGE_SCN_MEM_EXECUTE;
    }
    wsprintf(buf, "%08X", SectInfoWrk.dwCharacteristics);
    SetDlgItemText(hDlgWnd, IDC_SE_CHAR, buf);
    break;

case IDC_SE_READ:
    if(IsDlgButtonChecked(hDlgWnd, IDC_SE_READ) == BST_CHECKED) {
        SectInfoWrk.dwCharacteristics |= IMAGE_SCN_MEM_READ;
    }
    else {
        SectInfoWrk.dwCharacteristics ^= IMAGE_SCN_MEM_READ;
```

```
    }

    wsprintf(buf, "%08X", SectInfoWrk.dwCharacteristics);

    SetDlgItemText(hDlgWnd, IDC_SE_CHAR, buf);

    break;

case IDC_SE_WRITE:

    if(IsDlgButtonChecked(hDlgWnd, IDC_SE_WRITE) == BST_CHECKED) {

        SectInfoWrk.dwCharacteristics |= IMAGE_SCN_MEM_WRITE;

    }

    else {

        SectInfoWrk.dwCharacteristics ^= IMAGE_SCN_MEM_WRITE;

    }

    wsprintf(buf, "%08X", SectInfoWrk.dwCharacteristics);

    SetDlgItemText(hDlgWnd, IDC_SE_CHAR, buf);

    break;

case IDOK:

    GetDlgItemText(hDlgWnd, IDC_SE_NAME, SectInfoWrk.byName, sizeof(SectInfoWrk.byName));

    GetDlgItemText(hDlgWnd, IDC_SE_VSIZE, buf, sizeof(buf));

    SectInfoWrk.dwVSize = strtoul(buf, &stop, 16);

    GetDlgItemText(hDlgWnd, IDC_SE_VOFFSET, buf, sizeof(buf));

    SectInfoWrk.dwVOffset = strtoul(buf, &stop, 16);

    GetDlgItemText(hDlgWnd, IDC_SE_RSIZE, buf, sizeof(buf));

    SectInfoWrk.dwRSize = strtoul(buf, &stop, 16);

    GetDlgItemText(hDlgWnd, IDC_SE_ROFFSET, buf, sizeof(buf));

    SectInfoWrk.dwROffset = strtoul(buf, &stop, 16);

    GetDlgItemText(hDlgWnd, IDC_SE_CHAR, buf, sizeof(buf));

    SectInfoWrk.dwCharacteristics = strtoul(buf, &stop, 16);

    EndDialog(hDlgWnd, IDOK);

    break;

case IDCANCEL:

    EndDialog(hDlgWnd, IDCANCEL);

    break;

}

break;

default:

    return FALSE;
```

```

        break;
    }

    return TRUE;
}

LRESULT CALLBACK MainDlgProc(HWND hDlgWnd, UINT msg, WPARAM wp, LPARAM lp)
{
    // 메인 컨트롤 섹션의 프로시저

    {
        char *ListHeader[] = {"Section", "Virtual Size", "Virtual Offset", "Raw Size", "Raw Offset", "Characteristics"};
        const int ColX[] = { 50, 73, 73, 73, 73, 73 };
        82
    };

    RECT rect;
    UINT x,y,w,h,xMax,yMax;
    int i;
    char buf[TEXTLEN];
    LPBYTE stop;

    HWND hList;
    DWORD dwStyle;
    LV_COLUMN lvCol;
    LV_ITEM item;

    switch (msg) {
        case WM_INITDIALOG:
            SendMessage(GetDlgItem(hDlgWnd, IDC_FIXSECT), BM_SETCHECK, (WPARAM)1, 0L);
            bFixSect = TRUE;
            SendMessage(GetDlgItem(hDlgWnd, IDC_REBUILD), BM_SETCHECK, (WPARAM)1, 0L);
            bRebuild = TRUE;
            SendMessage(GetDlgItem(hDlgWnd, IDC_RDO_M1), BM_SETCHECK, (WPARAM)1, 0L);
            iRebMethod = 1;

            // 중앙 다이얼로그 창
            GetWindowRect(hDlgWnd, &rect);
            h = rect.bottom - rect.top;
            w = rect.right - rect.left;
    }
}

```

OllyDbg 플러그인 제작

```
xMax = GetSystemMetrics(SM_CXMAXIMIZED);

yMax = GetSystemMetrics(SM_CYMAXIMIZED);

x = xMax/2 - w/2;

y = yMax/2 - h;

MoveWindow(hDlgWnd,x,y,w,h,TRUE);

wsprintf(buf, "OllyDump - %s", DbgeName);

SetWindowText(hDlgWnd,buf);

wsprintf(strCurEIP, "%X", GetCurrentEIP() - PEFileInfo.dwImageBase);

SetDlgItemText(hDlgWnd, IDC_OEP, strCurEIP);

wsprintf(buf, "%X", PEFileInfo.dwImageBase);

SetDlgItemText(hDlgWnd, IDE_FROM, buf);

wsprintf(buf, "%X", PEFileInfo.dwSizeOfImage);

SetDlgItemText(hDlgWnd, IDE_SIZE, buf);

wsprintf(buf, "%X", PEFileInfo.dwAddrOfEP);

SetDlgItemText(hDlgWnd, IDC_EP, buf);

wsprintf(buf, "%X", PEFileInfo.dwBaseOfCode);

SetDlgItemText(hDlgWnd, IDC_BASECODE, buf);

wsprintf(buf, "%X", PEFileInfo.dwBaseOfData);

SetDlgItemText(hDlgWnd, IDC_BASEOFDATA, buf);

hList = GetDlgItem(hDlgWnd, IDC_SECTLIST);

dwStyle = ListView_GetExtendedListViewStyle(hList);

dwStyle |= LVS_EX_FULLROWSELECT | LVS_EX_GRIDLINES;

ListView_SetExtendedListViewStyle(hList, dwStyle);

for(i=0; i<sizeof(ListHeader)/sizeof(&ListHeader[0]); i++) {

    lvCol.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT | LVCF_SUBITEM;

    lvCol fmt = LVCFMT_LEFT;

    lvCol cx = ColX[i];
```

OllyDbg 플러그인 제작

```
lvCol.pszText = ListHeader[i];
lvCol.iSubItem = 0;
ListView_InsertColumn(hList, i, &lvCol);
}

item.mask = LVIF_TEXT;
item.cchTextMax = sizeof(buf);
for(i=0; i<(int)PEFileInfo.woNumOfSect; i++) {
    item.pszText = (IpSectInfo+i)->byName;
    item.iItem = i;
    item.iSubItem = 0;
    ListView_InsertItem(hList, &item);

    wsprintf(buf,"%08X", (IpSectInfo+i)->dwVSize);
    item.pszText = buf;
    item.iItem = i;
    item.iSubItem = 1;
    ListView_SetItem(hList, &item);

    wsprintf(buf,"%08X", (IpSectInfo+i)->dwVOffset);
    item.pszText = buf;
    item.iItem = i;
    item.iSubItem = 2;
    ListView_SetItem(hList, &item);

    wsprintf(buf,"%08X", (IpSectInfo+i)->dwVSize);
    item.pszText = buf;
    item.iItem = i;
    item.iSubItem = 3;
    ListView_SetItem(hList, &item);

    wsprintf(buf,"%08X", (IpSectInfo+i)->dwVOffset);
    item.pszText = buf;
    item.iItem = i;
    item.iSubItem = 4;
    ListView_SetItem(hList, &item);
```

```

wsprintf(buf, "%08X", (IpSectInfo+i)->dwCharacteristics);

item.pszText = buf;
item.iItem = i;
item.iSubItem = 5;
ListView_SetItem(hList, &item);

}

SecLstDlgProcOrg = (WNDPROC)SetWindowLong(hList, GWL_WNDPROC, (LONG)SecLstDlgProc);

return TRUE;

case WM_COMMAND:
switch (LOWORD(wp)) {
case IDC_FIXSECT:
blFixSect = (IsDlgButtonChecked(hDlgWnd, IDC_FIXSECT) == BST_CHECKED) ? TRUE : FALSE;

hList = GetDlgItem(hDlgWnd, IDC_SECTLIST);
if(blFixSect) {

item.mask = LVIF_TEXT;
item.cchTextMax = sizeof(buf);
for(i=0; i<(int)PEFileInfo.woNumOfSect; i++) {
wsprintf(buf, "%08X", (IpSectInfo+i)->dwVSize);

item.pszText = buf;
item.iItem = i;
item.iSubItem = 3;
ListView_SetItem(hList, &item);

wsprintf(buf, "%08X", (IpSectInfo+i)->dwVOffset);

item.pszText = buf;
item.iItem = i;
item.iSubItem = 4;
ListView_SetItem(hList, &item);
}
}
else {
item.mask = LVIF_TEXT;
}
}

```

```

item.cchTextMax = sizeof(buf);

for(i=0; i<(int)PEFileInfo.woNumOfSect; i++) {
    wsprintf(buf, "%08X", (lpSectInfo+i)->dwRSize);

    item.pszText = buf;
    item.iItem = i;
    item.iSubItem = 3;

    ListView_SetItem(hList, &item);

    wsprintf(buf, "%08X", (lpSectInfo+i)->dwROffset);

    item.pszText = buf;
    item.iItem = i;
    item.iSubItem = 4;

    ListView_SetItem(hList, &item);
}

}

break;

case IDC_REBUILD:
    blRebuild = (IsDlgButtonChecked(hDlgWnd, IDC_REBUILD) == BST_CHECKED) ? TRUE : FALSE;
    break;

case IDC_GETEIP:
    wsprintf(strCurEIP, "%X", GetCurrentEIP() - PEFileInfo.dwImageBase);

    SetDlgItemText(hDlgWnd, IDC_OEP, strCurEIP);
    break;

case IDOK:
    // 텍스트 길이를 체크하고 수집
    if(GetWindowTextLength(GetDlgItem(hDlgWnd, IDE_FROM)) == 0) {
        MessageBox(hDlgWnd, "Please specify start address of dumping.", PNAME, MB_OK | MB_ICONINFORMATION);
        break;
    }
    else {
        GetDlgItemText(hDlgWnd, IDE_FROM, buf, sizeof(buf));
        if(!IsValidNumber(buf, strlen(buf), NUM_HEX)) {
            goto NUM_INVALID;
        }
    }
}

```

```
        else {
            PEFileInfo.dwImageBase = strtoul(buf,&stop,16);
        }
    }

    if(GetWindowTextLength(GetDlgItem(hDlgWnd,IDE_SIZE)) == 0) {
        MessageBox(hDlgWnd,"Please specify dump size.",PNAME,MB_OK | MB_ICONINFORMATION);
        break;
    }
    else {
        GetDlgItemText(hDlgWnd,IDE_SIZE,buf,sizeof(buf));
        if(!IsValidNumber(buf,strlen(buf),NUM_HEX)) {
            goto NUM_INVALID;
        }
        else {
            PEFileInfo.dwSizeOfImage = strtoul(buf,&stop,16);
        }
    }

    if(GetWindowTextLength(GetDlgItem(hDlgWnd, IDC_OEP)) > 0) {
        GetDlgItemText(hDlgWnd, IDC_OEP,buf,sizeof(buf));
        if(!IsValidNumber(buf,strlen(buf),NUM_HEX)) {
            goto NUM_INVALID;
        }
        else {
            PEFileInfo.dwAddrOfEP = strtoul(buf,&stop,16);
        }
    }

    if(GetWindowTextLength(GetDlgItem(hDlgWnd, IDC_BASEOFCODE)) > 0) {
        GetDlgItemText(hDlgWnd, IDC_BASEOFCODE,buf,sizeof(buf));
        if(!IsValidNumber(buf,strlen(buf),NUM_HEX)) {
            goto NUM_INVALID;
        }
        else {
            PEFileInfo.dwBaseOfCode = strtoul(buf,&stop,16);
        }
    }
}
```

```
        }

    }

if(GetWindowTextLength(GetDlgItem(hDlgWnd, IDC_BASEOFDATA)) > 0) {
    GetDlgItemText(hDlgWnd, IDC_BASEOFDATA, buf, sizeof(buf));
    if(!isValidNumber(buf, strlen(buf), NUM_HEX)) {
        goto NUM_INVALID;
    }
    else {
        PEFileInfo.dwBaseOfData = strtoul(buf, &stop, 16);
    }
}

hList = GetDlgItem(hDlgWnd, IDC_SECTLIST);
item.mask = LVIF_TEXT;
item.cchTextMax = sizeof(buf);
for(i=0; i<(int)PEFileInfo woNumOfSect; i++) {
    item.pszText = (IpSectInfo+i)->byName;
    item.iItem = i;
    item.iSubItem = 0;
    ListView_GetItem(hList, &item);

    item.pszText = buf;
    item.iItem = i;
    item.iSubItem = 1;
    ListView_GetItem(hList, &item);
    (IpSectInfo+i)->dwVSize = strtoul(buf, &stop, 16);

    item.pszText = buf;
    item.iItem = i;
    item.iSubItem = 2;
    ListView_GetItem(hList, &item);
    (IpSectInfo+i)->dwVOffset = strtoul(buf, &stop, 16);

    item.pszText = buf;
    item.iItem = i;
```

OllyDbg 플러그인 제작

```
item.iSubItem = 3;
ListView_GetItem(hList, &item);
(lpSectInfo+i)->dwRSize = strtoul(buf,&stop,16);

item.pszText = buf;
item.iltem = i;
item.iSubItem = 4;
ListView_GetItem(hList, &item);
(lpSectInfo+i)->dwROffset = strtoul(buf,&stop,16);

item.pszText = buf;
item.iltem = i;
item.iSubItem = 5;
ListView_GetItem(hList, &item);
(lpSectInfo+i)->dwCharacteristics = strtoul(buf,&stop,16);
}

if(IsDlgButtonChecked(hDlgWnd, IDC_RDO_M1) == BST_CHECKED) {
    iRebMethod = 1;
}
if(IsDlgButtonChecked(hDlgWnd, IDC_RDO_M2) == BST_CHECKED) {
    iRebMethod = 2;
}

EndDialog(hDlgWnd, IDOK);
break;

NUM_INVALID:
    MessageBox(hDlgWnd, "Invalid Number!!",PNAME,MB_OK | MB_ICONINFORMATION);
    return FALSE;

case IDCANCEL:
    EndDialog(hDlgWnd, IDCANCEL);
    break;

default:
```

```

        return FALSE;
    }

    break;

default:
    return FALSE;
}

return TRUE;
}

DWORD GetCurrentEIP(void)
{
    t_thread* t2; // t_thread

    t2=Findthread(Getcputhreadid());
    return t2->reg.ip;
}

BOOL IsValidNumber(char *numstr, int size, int mode)
{
    int i;
    char *s;

    s = numstr;
    if(*s == '-' || *s == '+') {
        s++;
        size--;
    }

    for(i=0; i<size; i++) {
        switch(mode) {
        case NUM_DEC:
            if(*(s+i) < '0' || *(s+i) > '9') {
                return FALSE;
            }
            break;
        case NUM_HEX:
            if(!( (*s+i) >= '0' && *(s+i) <= '9')

```

ollyDbg 플러그인 제작

```
    || (*(s+i) >= 'A' && *(s+i) <= 'F')
    || (*(s+i) >= 'a' && *(s+i) <= 'f') )) {
    return FALSE;
}

break;

default:
break;

}

return TRUE;
}
```

제 3 장 Windows 크랙 기본편

3.7 아이디/패스워드 기반 프로텍션 크랙 및 Key generator 제작

들어가며

원도우 응용프로그램에는 다양한 Protection 이 존재합니다. 프로그램 사용을 허가할 때 단일 시리얼 넘버 외에 사용자 이름과 패스워드를 동시에 입력하는 것도 있습니다. 개별적인 사용자들에게 프로그램 사용 허가 여부를 결정할 수 있기 때문에 많은 프로그램에서 사용되고 있습니다. 이러한 Protection 크랙은 올바른 시리얼 넘버를 결정하는 체크 루틴을 리버싱 한 후 인증을 통과할 수 있는 사용자 ID/Password 를 유추할 수 있습니다. 더 나아가 임의적으로 시리얼 넘버를 생성하는 키 제너레이터(Key Generator)를 작성할 수 있습니다.

인증루틴 확인

첨부파일 “CrackMe03.exe”를 OllyDbg 로 불러 들이고 난 후 메뉴에서 [Debug]>[Run]을 클릭하여 실행 합니다(단축키 F9). 여기에서는 테스트를 위해 사용자 ID 는 “security”, 시리얼 넘버는 “87654321”을 임시적으로 입력하였습니다(그림 1). ([등록]버튼은 아직 누르지 않습니다.)

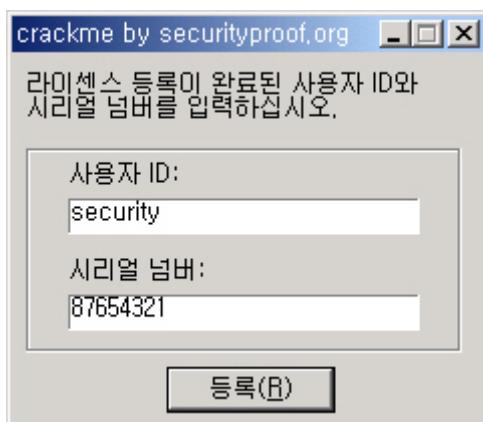


그림 1. CrackMe03.exe 메인 화면

등록 준비가 완료 되었다면 바로 브레이크 포인트를 설정합니다. 에디트 박스로부터 문자열을 획득하는 윈도우 API 함수는 3 가지 종류가 있다고 이미 설명하였습니다(3-3 장). Ctrl+N 키로 API 함수 목록을 표시한 후 문자열 획득 API 를 찾으면 GetDlgItemTextA 를 찾을 수 있으므로 그 함수에 브레이크 포인트를 설치 합니다 (GetDlgItemTextA에서 마우스 오른쪽 클릭 → Set breakpoint on every reference 선택).

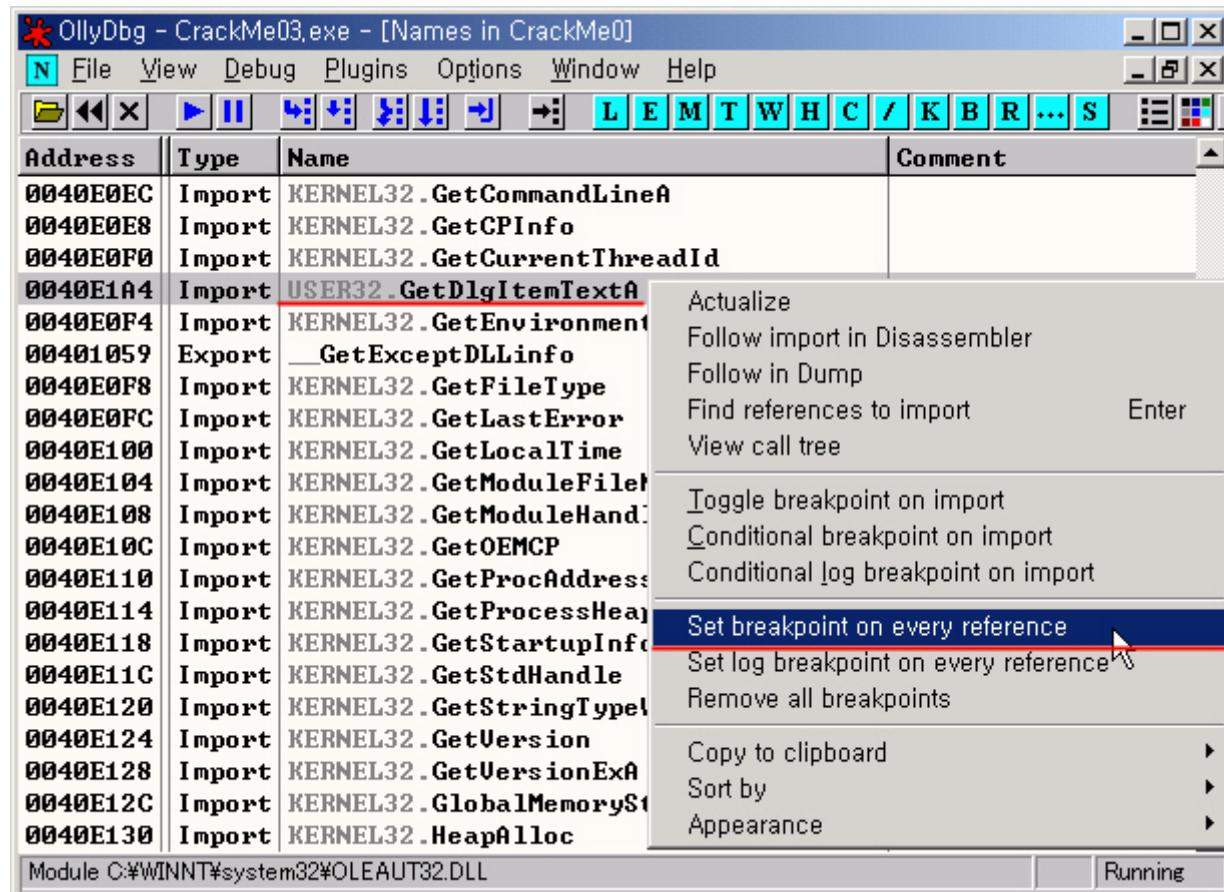


그림 2. GetDlgItemTextA에 브레이크 포인트 설정

브레이크 포인트 설정이 완료되고 CrackMe03.exe 메인 화면에서 [등록] 버튼을 누르면 주소 004011FAh [CALL <JMP.&USER32.GetDlgItemTextA>]에서 프로그램이 잠시 멈추게 됩니다. 이 API 함수의 브레이크 포인트 설정 지점에서 사용자 ID와 시리얼 넘버 어느 쪽이 획득 되는지 확인합니다. 대부분 사용자 입력을 받아들이는 순서대로 변수(버퍼)에 저장하게 됩니다. 리버싱/크랙을 빠르게 하려면 “내가 프로그램을 작성했다면 이러한 부분은 이렇게 작성했을 것이다”라는 가정도 많은 도움이 됩니다. 스택에 Push 된 값을 참조하면(그림 3), 이 경우 위에서 세번째 Buffer가 가르키는 주소 0012FB74h에서 획득한 문자열이 보관됩니다. 스택 윈도우의 세번째를 활성 상태로 하여, 마우스 오른쪽 클릭->[Follow in Dump]를 선택하면 왼쪽 밑 덤프윈도우에 보관된 영역이 표시됩니다.

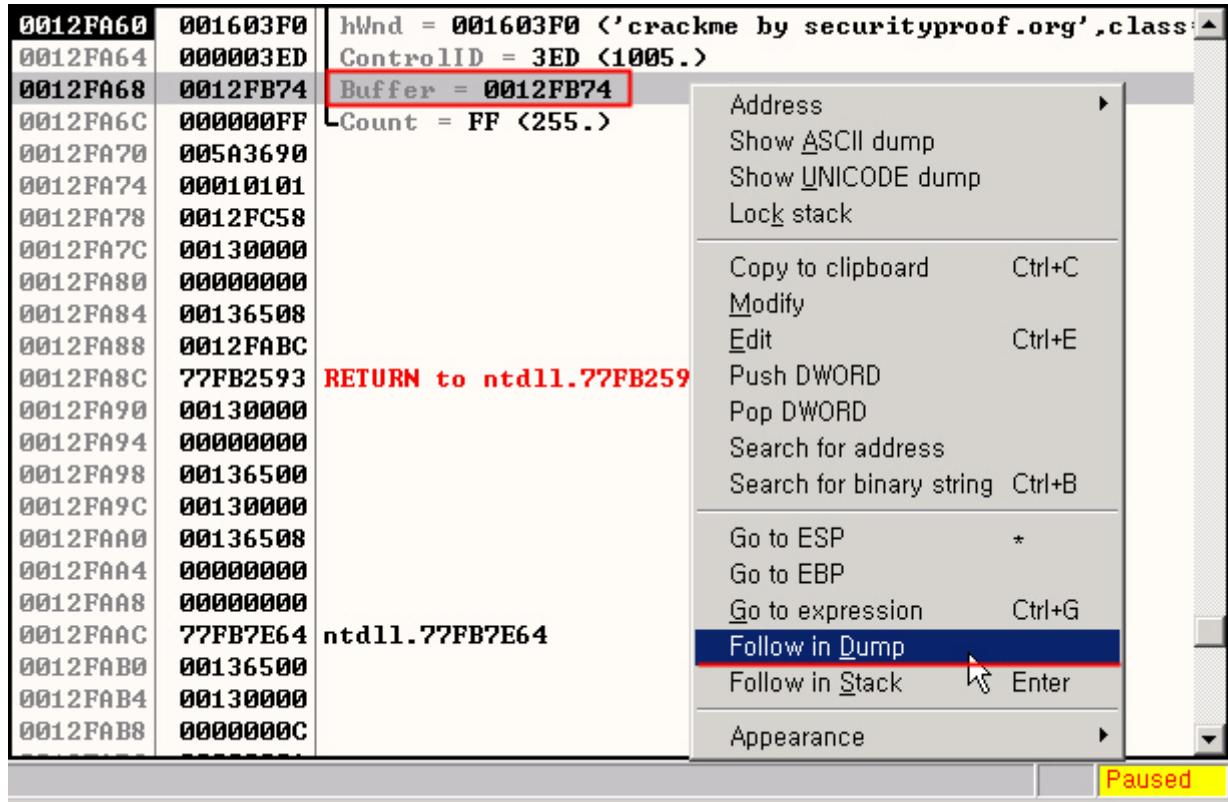


그림 3. GetDlgItemTextA 의 저장 버퍼 확인

단계별 진행을 몇 번 하면(F8 키를 15 번 누름), GetDlgItemTextA 실행이 완료되는 시점에서 덤프 윈도우에 사용자 ID로 입력한 "security" 문자열을 확인할 수 있습니다(그림 4).

Address	Hex dump	ASCII
0012FB74	73 65 63 75 72 69 74 79 00 00 10 01 4F 00 00 00	security .+r...
0012FB84	05 00 00 00 01 00 00 00 0D 00 00 00 79 00 10 01	...r.....y.+r
0012FB94	01 00 00 00 01 00 00 00 0F 00 00 00 28 FC 12 00	...r...*...<?.
0012FBA4	04 4C DF 77 EC 06 01 18 04 00 00 00 79 00 10 01	L??r↑...y.+r
0012FBBA4	49 00 5A 00 33 7E E0 77 EC 06 01 18 E8 FB 12 00	I.Z.3~??↑燎t.
0012FBC4	00 00 00 00 60 59 13 00 00 02 00 00 00 E0 FD 7F`Y!!..1...省△
0012FBD4	41 21 F4 77 5B 00 10 01 03 00 00 00 03 00 00 00	A?/?L.+rL...L...
0012FBE4	4F 00 00 00 04 00 00 00 04 00 00 00 00 00 00 00	N i i n

그림 4. 버퍼에 저장된 문자열 확인(사용자 ID).

문자열 확인이 끝났다면 그림 5의 코드를 봅시다. GetDlgItemTextA의 반환 값으로서 입력한 사용자 ID의 문자열 길이가 리턴(return)되고, 그 직후에 CMP 명령과 JNZ 명령으로 8 문자 이외라면 등록 실패가 되는 것을 알 수 있습니다. 처음부터 8 문자의 사용자 ID(security)를 입력하였기 때문에, 여기에서는 분기하지 않습니다. 만약 8자 이상의 문자를 넣고 싶다면 이 체크 부분을 수정할 수 있지만, 차후 시리얼 넘버의 생성 부분도 수정해야 되기 때문에 부득이한 경우가 아니라면 필자의 설명을 순서대로 따라가 주시기 바랍니다.

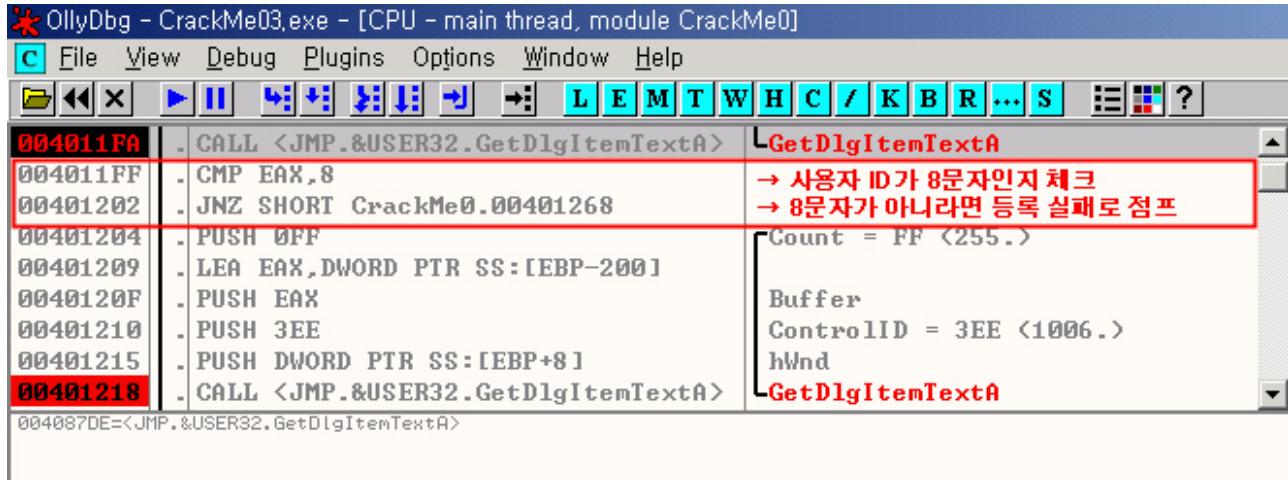


그림 5. 사용자 ID 길이 체크

다음은 시리얼 넘버의 획득입니다(그림 6). 사용자 ID 체크부분과 거의 같은 코드인 것을 알 수 있습니다. 시리얼 넘버도 8 자 이외는 등록 실패로 점프됩니다. 여기서 시리얼 넘버가 저장되는 buffer 주소는 0012FA74h 입니다. 이것으로 사용자가 입력한 ID 와 시리얼 번호 모두 획득되었습니다. 모든 조건을 만족하였다면, 이제부터 체크 루틴의 본체입니다.

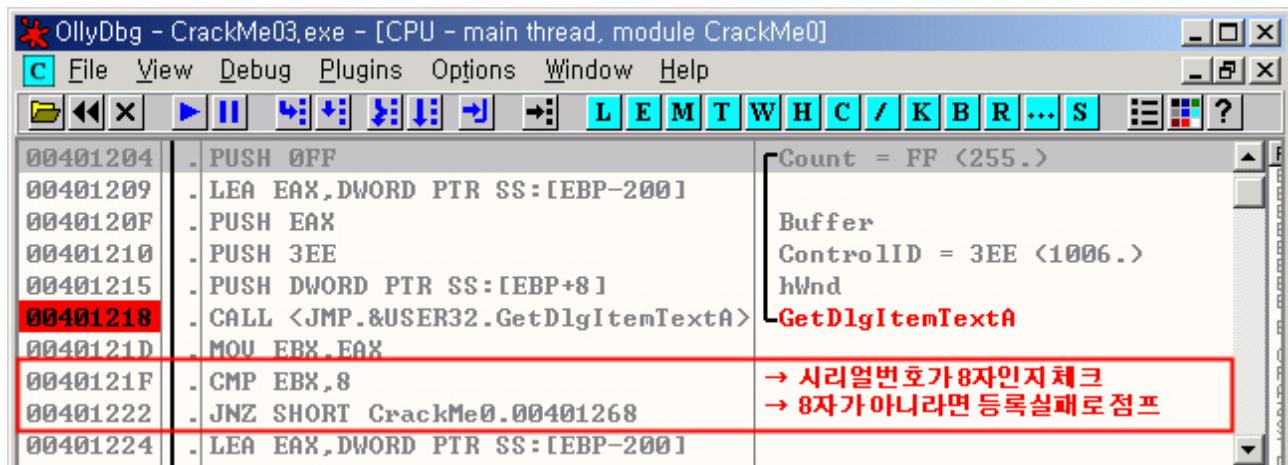


그림 6. 시리얼 넘버 길이 체크

CALL 명령 직후에 비교 분기 명령은 없습니다. 이전 장에서 “CALL 명령 직후에 비교/분기명령이 없을 경우, 체크 루틴과 직접 관계 없는 함수일 가능성이 높기 때문에 시간 절약을 위해 코드 추적은 하지 않습니다.”고 설명했지만, CALL 명령 직전에 입력 시리얼의 어드레스가 Push 되고 있기 때문에 전처리일 가능성이 높습니다. CALL 명령 호출 지점의 루틴 트레이스는 일단 뒤로하고, F8 키로 단계별 실행을 수행하여 반환값을 관찰합시다. EAX 레지스터에 반환(return value) 주소 값으로 05397FB1h⁵⁸ 가 저장되었습니다. 화면 오른쪽 위의 레지스터 윈도우에서 EAX 레지스터 숫자부분을 더블 클릭하면, 레지스터 값을 표시하는 작은 윈도우가 열립니다(그림 7).

⁵⁸ 반환되는 EAX 주소값은 컴퓨터에 따라 달라집니다.



그림 7. EAX 레지스터 화면(Unsigned 값을 확인!)

부호없는(Unsigned)값을 보면, EAX 레지스터값(5397FB1h)은 10 진수로 87654321 인 것을 알 수 있고, 이 값은 입력한 시리얼 넘버 “87654321”를 수치로 변환한 것과 동일합니다. 즉 입력 시리얼 값을 다르게 넣더라도 Signed 와 Unsigned 가 동일한 정수값을 갖고 있기 때문에, 문자열을 정수 값으로 변환하여 처리하고 있다는 것을 알 수 있습니다.

마지막으로 CALL 명령이 하나 더 있지만(0040123Ah), 그 직후에 비교 · 분기 명령(00401242h)이 있습니다. 그 다음에는 등록 성공 · 실패 메시지 표시 처리(00401244h, 00401268h)가 기다리고 있으므로 체크루틴(00401242h)이 중요한 부분이라는 것을 알게 됩니다. 하나 전의 CALL 명령은 입력 시리얼 넘버를 정수 값으로 변환하고 있었지만, 이 부분은 사용자 ID 를 기초로 해서 어떤 값을 생성하고 있는 것 같습니다. 일단 내부 트레이스는 뒤로 미루고, F8 키로 스텝 실행하여 CALL 명령의 반환값을 확인합시다. 사용자 ID 로 입력한 “security”에 대응하는 시리얼 키 값만 확인하기 위해서입니다.

00401231	. MOV EBX,EAX	
00401233	. LEA EAX,DWORD PTR SS:[EBP-100]	Arg1 → 입력한 사용자 ID값의 주소 CrackMe0.00401150 → ID로부터 키 생성
00401239	. PUSH EAX	
0040123A	. CALL CrackMe0.00401150	
0040123F	. POP ECX	
00401240	. CMP EBX,EAX	→ 입력 시리얼 키와 생성된 시리얼 키 값 비교
00401242	. JNZ SHORT CrackMe0.00401268	Style = MB_OK MB_APPLMODAL CrackMe0x03
00401244	. PUSH 0	
00401246	. PUSH CrackMe0.00409147	

그림 8. 사용자 ID 에 따른 시리얼 키 값 비교 부분

0040123Ah 주소에서 F8 키를 입력하고 난 후 반환되는 EAX 값을 확인하면, 입력한 사용자 ID “security”에 대응하는 시리얼 넘버는 0121060Ch 를 10 진수로 변환한 “18941452”인 것을 쉽게 알 수 있습니다. 즉, 00401240h 주소에서 입력한 시리얼 키 값 【EBX】에 “05397FB1h(87654321)”로 저장】과 생성된 시리얼 키 값 【EAX】에 “0121060Ch(18941452)”로 저장】이 같은지 확인하여 다르면 등록실패로, 같으면 등록성공으로 점프하게 됩니다.

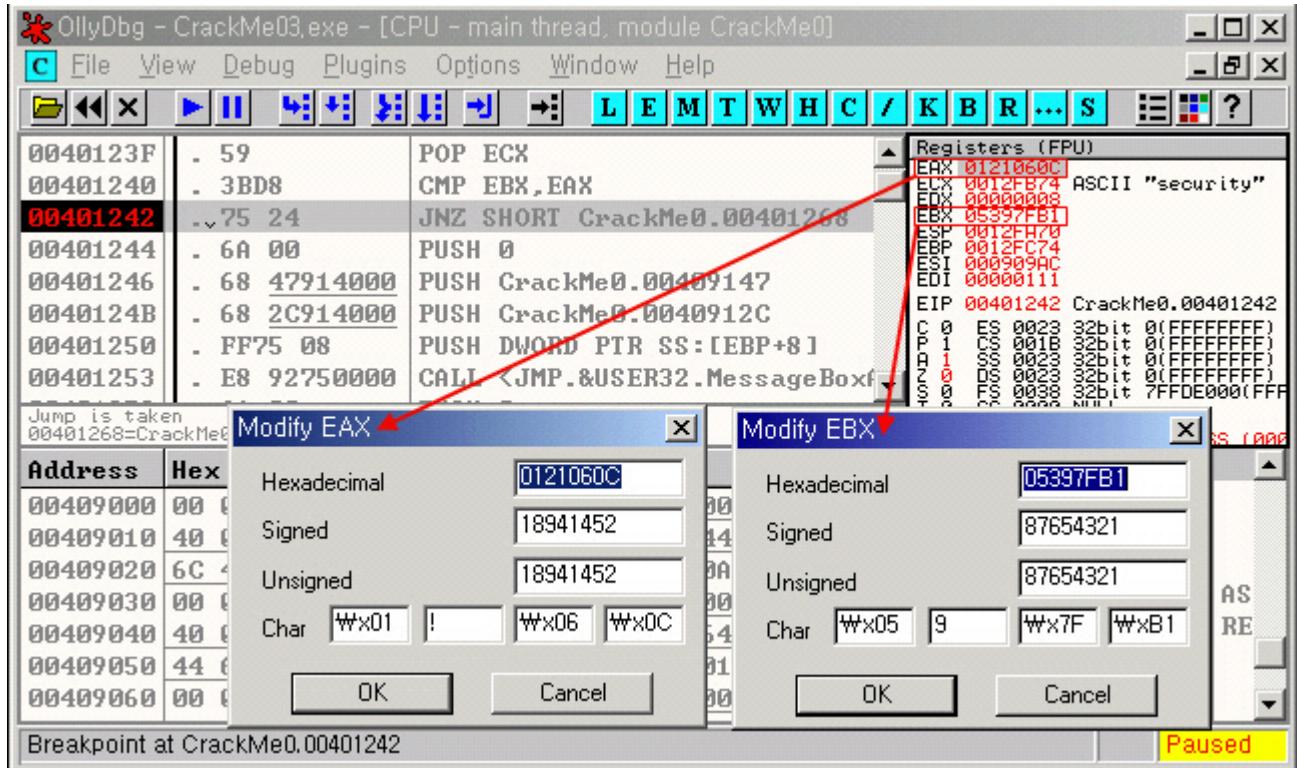


그림 9. 입력한 키 값과 생성된 키 값의 비교

윈도우에서 Crackme03.exe 를 다시 실행하여 사용자 ID 로 “security”를 시리얼 넘버로 “18941452”를 입력하면 등록성공 화면을 볼 수 있습니다.



그림 10. 등록성공 화면

Key generator 제작

CrackMe03.exe 는 입력한 사용자 ID 에 따라 유효한 시리얼 넘버를 생성하는 형태라는 것을 알았습니다. 앞에서 설명한 흐름을 다시 한번 생각해 보면, 유효한 시리얼 넘버만을 알아내기 위해서는 “키 생성 알고리즘”까지 분석할 필요가 없었습니다. 단순히 트레이스만으로 입력한 사용자 ID 에 대응된 유효한 시리얼 넘버를 확인 할 수

아이디/패스워드 기반 Protection 크랙

있습니다.

그러면 사용자 ID 만 입력하면 바로 유효한 시리얼 넘버가 생성되는 키 제너레이터(Key generator)를 작성해 보겠습니다. 키 제너레이터를 만들기 위해서는 트레이스를 뒤로 미룬 “사용자 ID 에서 유효한 시리얼 넘버를 생성하는” 부분을 분석해야 할 필요가 있습니다. 이 함수는 어ドレス 00401150h 에서 시작하고 있다는 것을 이미 앞에서 살펴 보았습니다. 그 흐름을 따라가 봅시다.

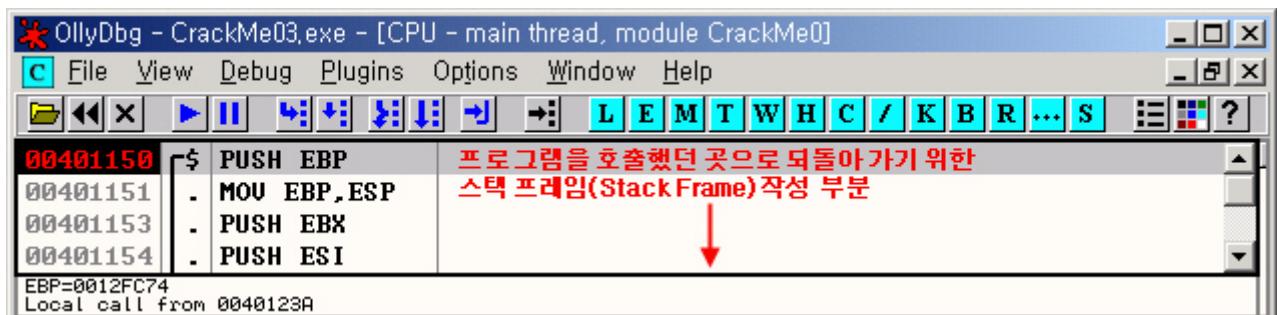


그림 11. 스택 프레임 작성 부분

스텝 실행하여(F7 키) 00401150h 주소까지 이동 합니다. 유효한 시리얼 넘버 생성을 마치고 해당 함수를 호출했던 원래의 위치로 되돌아갈 정보를 담고 있는 스택 프레임(Stack Frame)은 모든 Call Back 함수의 첫 부분에 반드시 나오게 됩니다. 00401150h ~ 00401154h 까지는 스택프레임 작성 부분이므로 특별히 신경 쓸 필요는 없습니다.

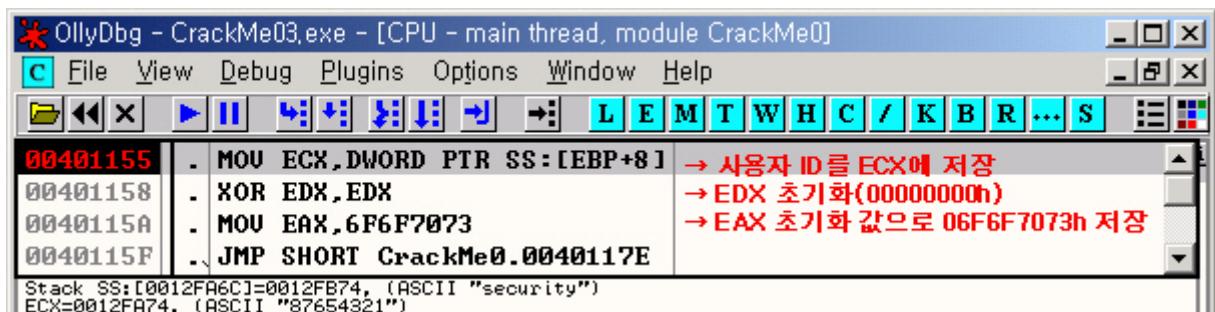


그림 12. 사용자 ID 를 ECX 에 저장

필자가 입력한 사용자 ID 와 시리얼 넘버는 현재 EAX(security), ECX(87654321)에 저장되어 있을 것입니다. 그림 12 에서 보듯이 00401155h 에서 ECX 레지스터는 사용자 ID 가 들어갑니다. 00401158h 에서는 EDX 레지스터 값은 제로로 초기화되고, EAX 레지스터에는 초기값으로서 06F6F7073h 가 저장됩니다. 그 뒤 0040115Fh 번지에서 JMP 명령에 의해 0040117Eh 번지로 점프하게 됩니다.

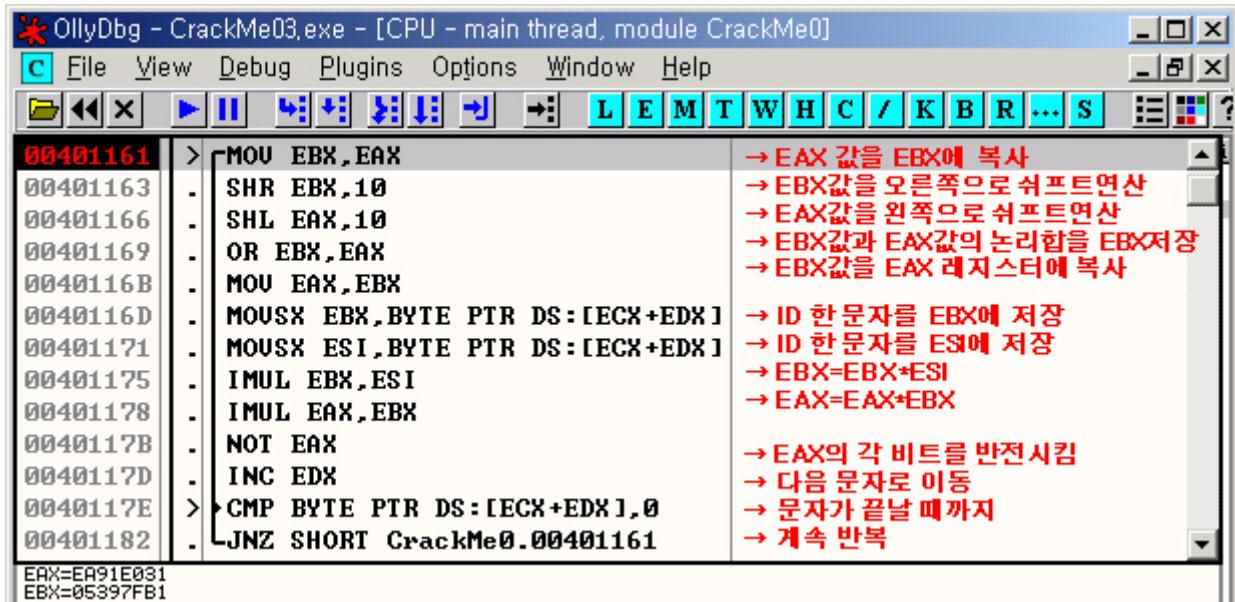


그림 13. 시리얼 넘버 생성 부분

여기가 시리얼 넘버 생성의 중요 부분입니다. 역 어셈블 리스트에 익숙하지 않을 때는 당황할 수도 있겠지만, 조금씩 설명하도록 하겠습니다. 먼저 0040115Fh의 JMP 명령에 의해 주소 0040117E부터 실행됩니다.

루프 중단 체크		
0040117E	> 803C11 00	CMP BYTE PTR DS:[ECX+EDX],0 ; 문자가 끝날 때까지
00401182	.^75 DD	JNZ SHORT CrackMe0.00401161 ; 계속 반복

이 시점에서는 ECX 레지스터에 사용자 ID의 첫번째 어드레스가, EDX 레지스터에는 제로가 저장되어 있기 때문에, 【BYTE PTR DS:[ECX+EDX】】는 사용자 ID의 첫 번째 문자를 나타냅니다. 이 값이 제로 이외라면(=문자열 끝이 아니라면) JNZ 명령으로 점프합니다.

EAX 레지스터의 상위 16비트와 하위 16비트 교체		
00401161	> 8BD8	MOV EBX,EAX ; EAX 값을 EBX에 복사
00401163	. C1EB 10	SHR EBX,10 ; EBX 값을 오른쪽으로 쉬프트 연산(16Bit)
00401166	. C1E0 10	SHL EAX,10 ; EAX 값을 왼쪽으로 쉬프트 연산(16Bit)
00401169	. 0BD8	OR EBX,EAX ; EBX 값과 EAX 값의 논리합을 EBX에 저장
0040116B	. 8BC3	MOV EAX,EBX ; EBX 값을 EAX 레지스터에 복사

오른 쪽에 주석을 넣었지만, 이들 조작은 “EAX 레지스터 상위 16비트와 하위 16비트의 교체”입니다. 이해를 돋기 위해 12345678h 값을 가지고 명령어마다 레지스터 값의 변화를 아래와 같이 나타내었습니다.

아이디/패스워드 기반 Protection 크랙

오른쪽 주석의 레지스터 값은 원쪽 명령 실행후의 결과[이해를 돋기 위한 예]	
MOV EBX,EAX ; EAX=12345678h	EBX=12345678h
SHR EBX,10 ; EAX=12345678h	EBX=0001234h
SHL EAX,10 ; EAX=56780000h	EBX=00001234h
OR EBX,EAX ; EAX=56780000h	EBX=56781234h
MOV EAX,EBX ; EAX=56781234h	EBX=56781234h

위에서 보듯이, EAX 레지스터의 상위 16 비트와 하위 16 비트가 교체되는 모습을 알 수 있습니다.

사용자 ID에서 문자를 수집하여 곱셈연산	
0040116D . 0FBE1C11	MOVsx EBX,BYTE PTR DS:[ECX+EDX] ; ID 한문자를 EBX에 복사
00401171 . 0FBE3411	MOVsx ESI,BYTE PTR DS:[ECX+EDX] ; ID 한문자를 ESI에 복사
00401175 . 0FAFDE	IMUL EBX,ESI ; EBX = EBX*ESI
00401178 . 0FAFC3	IMUL EAX,EBX ; EAX = EAX*EBX

0040116Dh ~ 00401171h 주소의 【BYTE PTR DS:[ECX+EDX】는 사용자 ID의 첫 번째 문자이며, 이것을 EBX 레지스터와 ESI 레지스터에 복사하고 있습니다. 단 MOV 명령은 EWORD 형의 레지스터에 BYTE 값을 지정할 수 없습니다. 그래서 확장된 MOV 명령을 사용하여야 하며, 부호에 신경을 써야할 경우 MOVSX 를, 부호 없는 경우는 MOVZX 이용합니다. C 언어에서 말하는 캐스트 연산이 시행되고 있습니다. 00401175h ~ 00401178h 주소의 명령은 IMUL 에 의한 정수간 곱셈이 수행되고 있습니다.

EDX 레지스터에 1을 더한 후 다음 문자를 가리키도록 한다.	
0040117B . F7D0	NOT EAX ; EAX의 각 비트를 반전 시킵
0040117D . 42	INC EDX ; 다음 문자를 가리키도록 맞춤
0040117E > 803C11 00	CMP BYTE PTR DS:[ECX+EDX],0 ; 문자가 끝날때까지
00401182 .^75 DD	JNZ SHORT CrackMe0.00401161 ; 계속 반복함.

0040117Bh 주소의 NOT 명령은 EAX 레지스터값의 각 비트를 반전 시킵니다. 0040117Dh 주소의 INC 명령은 EDX 레지스터 값에 1을 더하는 것입니다. 이 명령을 통해 다음 0040117Eh 주소의 【BYTE PTR DS:[ECX+EDX】값이 사용자 ID의 다음 문자를 가리키도록 되기 때문에, 루프를 반복할 때마다 다음 문자가 참조됩니다(s→e→c→u→r→i→t→y). 8 문자 전부가 참조되면 다음은 NULL 문자가 참조 되어 루프에서 종료됩니다.

반환값 설정과 함수 리턴	
00401184 . 25 FFFFFFF03	AND EAX,3FFFFFF ; 하위 21 byte 추출
00401189 . 5E	POP ESI ; ESI 값 추출
0040118A . 5B	POP EBX ; EBX 값 추출

아이디/패스워드 기본 Protection 크랙

0040118B . 5D	POP EBP	; EBP 값 추출
0040118C . C3	RETN	; 함수를 호출했던 곳으로 복귀

AND 명령에 의해 EAX 레지스터 값을 마스킹 처리 한 후 하위 21 비트⁵⁹만 추출합니다(반환값). 각 레지스터 값을 추출 한 후, RETN 명령으로 CALL 명령을 호출했던 곳으로 복귀하고 있습니다. 이 반환값을 8 자리수의 문자열로 고친 것이 입력한 사용자 ID에 대응하는 시리얼 넘버가 됩니다.

EAX 레지스터 마스킹				
연산자	Hex 값	2 진수 값	10 진수 값	
AND	D921 060C 3FF FFFF	11011001001000010000011000001100 11111111111111111111111111111111	3642820108 67108863	
	121 060C	01001000010000011000001100	18941452	
21Bit 추출	121 060C	1001000010000011000001100		18941452

이 일련의 처리를 키 제너레이터 프로그램으로 구현해 보겠습니다. 여기서는 키 제너레이터 작성에 C 언어, C#, Delphi 를 이용하였습니다. 완성한 키 제너레이터 소스 코드는 아래와 같습니다. 역 어셈블 코드와 비교해 가면서 각자 자신있는 프로그래밍 소스 코드를 보면서 역 어셈블 리스트의 어느 부분에 대응하고 있는지 살펴 보시기 바랍니다. 본 문서에서는 프로그래밍 작성 방법에 대해서 설명하지 않습니다. 각자 전문도서나 해당 프로그래밍 웹 사이트를 참조하시기 바랍니다. 또한 키 제너레이터 소스코드내에는 예외적인 상황에 대하여 코딩하지는 않았습니다. 위에서 논의된 키 제네레이터 부분만을 간결하게 표현한 것입니다.

CrackMe03 키 제너레이터(C)
#include <stdio.h> int main(void) { char szUserID[9]; unsigned long dwSerial; int i; printf("사용자 ID (8 문자):"); scanf("%8s", szUserID); dwSerial = 0x6F6F7073; for (i = 0; i < 8; i++) {

⁵⁹ 3FFFFFFh 를 이진수로 고치면 1이 21개 늘어서는 값이 됩니다.

아이디/패스워드 가비t Protection 크랙

```
dwSerial = ((dwSerial << 0x10 )|(dwSerial >>0x10);

dwSerial *= szUserID[i] * szUserID[i];

dwSerial =~ dwSerial;

}

dwSerial &=0x03FFFFFF;

printf("시리얼 넘버: %08lu",dwSerial);

return 0;

}
```

CrackMe03 키 제너레이터(C#)

```
using System;

namespace KeygenCS

{

    class keygen03

    {

        [STAThread]

        static void Main(string[] args)

        {

            string szUserID;

            UInt32 dwSerial;

            int i;

            Console.WriteLine("사용자 ID (8 문자):");

            szUserID = Console.ReadLine();

            dwSerial = 0x6F6F7073;

            for (i = 0; i < szUserID.Length ; i ++)

            {

                dwSerial = ((dwSerial << 0x10 )|(dwSerial >>0x10);

                dwSerial = dwSerial * ((UInt32)(szUserID[i]) * (UInt32)(szUserID[i]));

                dwSerial =~ dwSerial;

            }

            dwSerial &= 0x03FFFFFF;

            Console.WriteLine("시리얼 넘버: {0}",dwSerial.ToString());

        }

    }

}
```

아이디/패스워드 기반 Protection 크랙

```
    }  
}
```

CrackMe03 키 제너레이터(Delphi)

```
function KeyFormMain1.fn_keygen(key_source: string): LongWord;  
  
var  
  dwSerial : LongWord;  
  i : Integer;  
  
begin  
  dwSerial := $6F6F7073;  
  i := 1;  
  
  while i <= Length(key_source) do  
  begin  
    dwSerial := ((dwSerial shl $10) or (dwSerial shr $10));  
    dwSerial := dwSerial * ord(key_source[i]) * ord(key_source[i]);  
    inc(i);  
    dwSerial := Not dwSerial;  
  end;  
  
  dwSerial := dwSerial and $3FFFFFF;  
  
  Result := dwSerial;  
end;
```

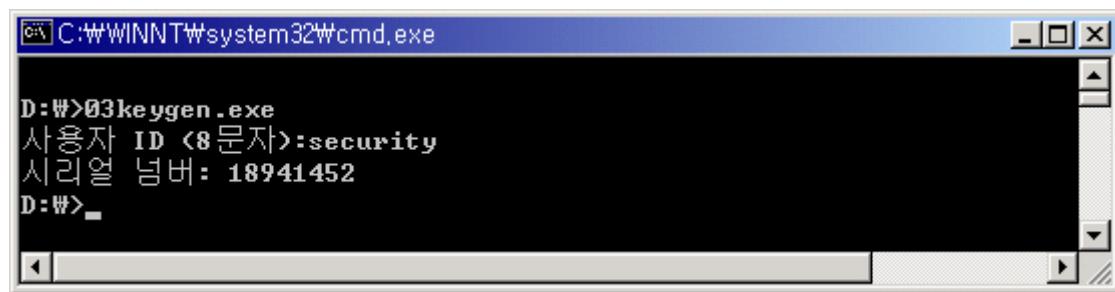


그림 14. 키 제너레이터로 확인(C 버전)



그림 15. 키 제너레이터로 확인(Delphi 버전)

소스코드를 보시면 알 듯이 각 언어의 형(type)의 범위만 다르고, 흐름은 모두 동일하다는 것을 알 수 있습니다.
자신의 생각을 표현할 수 있는 프로그래밍 언어 한가지는 반드시 습득하시길 바랍니다.

마지며

이번에 CrackMe 에 사용한 Protection 루틴은 내부에서 시리얼 넘버를 계산하고 있는 것으로 레지스터를 참조하면 사용자 ID 에 대응한 유효한 시리얼 넘버를 도출할 수 있습니다. 키 제너레이터 작성은 보더라도 핵심 루틴은 3 줄에 불과하고 번거로운 계산을 행할 필요도 없습니다. 산출된 루틴을 그대로 이식하면 되기 때문에 프로텍션으로서는 매우 취약합니다.

계산을 필요로 하는 복잡한 루틴을 짜면 그만큼 시간을 벌 수 있기 때문에 어느 정도 효과는 기대할 수 있습니다. 그러나 프로그래머 입장에서 그러한 복잡한 루틴을 생각한다는 것도 매우 성가신 작업이고, 어중간한 대처로는 전혀 의미가 없을 것입니다. 결국 레지스터나 메모리상에 유효한 시리얼 넘버를 직접 확인하지 못하도록 Protection 루틴을 작성 해야 합니다. 그러기 위해서는 컴파일러가 어떤 기계어 코드를 생성하는지에 대해서도 알 필요가 있을 것입니다.

제 3 장 Windows 크랙 기본편

3.8 크랙에 필요한 어셈블리어 이해(Getting acquainted with assembler)

들어가며

어셈블리 언어를 부분적으로 알고 있다면 디스어셈블한 코드를 한행씩 이해하는 것은 가능 합니다. 그러나 프로그램은 몇백 몇천줄 이상의 코드 덩어리가 결합되어 하나의 처리를 수행하는 경우가 많습니다. 매번 어셈블리 레퍼런스를 참조하는 것만으로는 바로 이해하지 못합니다. 같은 코드라도 다른 명령이나 다른 알고리즘으로 표현되어 있는 경우가 많고, 다양한 함수를 어셈블리언어에서는 몇줄에 걸쳐서 표현하기 때문입니다. 그래서 전체적인 어셈블리 언어를 연속적으로 빠르게 읽어 내려갈 수 있는 능력이 필요합니다.

예를 들어 EAX 레지스터 값을 제로로 하려면 [MOV EAX,0]뿐만이 아니라, [XOR EAX,EAX]라는 코드를 사용할 수도 있습니다. 이런 배경에는 코드 최적화(code Optimization)나 코드 난독화(code obfuscation), 컴파일러의 특성, 프로그래머의 취향 등이 있습니다. 최적화 패턴은 어느 정도 정해진 것이 대부분이나 난독화가 되면 조금 귀찮은 경우가 많고 경험에 의해 읽어서 이해하는 능력을 키워나갈 수 밖에 없습니다. 프로그래머의 입장에서 코드 난독화를 만드는 경우는 대다수 리버스 엔지니어링 시도를 자연시키기 위한 것이기 때문입니다. 최근 웹 사이트로부터 감염되는 바이러스들의 일부가 텍스트 형태의 스크립트(Javascript, VBscript)언어를 난독화⁶⁰ 시켜서 분석을 어렵게 만드는 사례도 좋은 예라고 할 수 있습니다. 다양한 프로그래밍 소스코드를 난독화 시켜주는 툴을 일컬어 “Source Code Obfuscator”라고 부릅니다. 2005년부터 급속히 증가되고 있는 polymorphic/metamorphic 기술을 접속시킨 소프트웨어들은 많은 리버서들에게 새로운 도전과제를 주고 있습니다.

이번 장에서는 최적화 코드의 형태와 하나의 결과를 여러 다른 명령을 사용하여 표현하는 예를 통해서 어셈블리 언어를 깊이있게 이해 하려고 합니다. 여기에서 설명하는 내용을 아는 것 만으로도 초보 리버서는 지금보다 몇배나 빨리 디스어셈블 코드를 해독할 수 있을 것입니다.

ADD 와 SUB

ADD 와 SUB 를 각각 사용하여 동일한 값을 계산해 보도록 하겠습니다. 먼저 EAX 에 128 을 더하는 방법을 생각해 보겠습니다. 그냥 생각하면 덧셈 명령인 ADD 를 사용합니다.

```
add eax, 128
```

⁶⁰ Javascript Obfuscator: <http://www.javascript-source.com/>

크랙에 필요한 어셈블리어 이해

이것을 SUB 명령을 사용하여 아래와 같이 동일하게 표현 할 수 있습니다.

```
sub eax, -128
```

128 을 더한다는 것은 -128 을 빼는 것과 같습니다. 4 칙연산의 기본입니다.

```
x + 128 = x - (-128)
```

두개의 명령어가 같은 결과를 얻게 된다는 것을 실제 확인해 보도록 하겠습니다. 텍스트 편집기를 이용하여 아래와 같이 어셈블리어를 작성한 후 add_sub.s라는 파일로 저장합니다.

```
1 section .bss
2 buf resb 10
3 section .text
4     global start
5 start:
6     mov eax,0
7     add eax,128
8     mov eax,0
9     sub eax,-128
10
11    push 0
12    extern ExitProcess
13    call ExitProcess
14
```

그림 1. add_sub.s 어셈블리 파일

작성이 완료되면 저장된 add_sub.s 파일을 오브젝트 파일로 변환하여 주기 위해 명령어 창에서 아래와 같이 실행합니다.

```
D:\WAssembler>nasmw.exe -fwin32 -O3 add_sub.s ➔ -O(Optimize branch offsets)
```

오브젝트 파일(incdec.obj) 생성이 완료되면 Windows 라이브러리와 연결하여 실행파일로 만들어 줍니다.

```
D:\WAssembler>golink.exe -entry start add_sub.obj kernel32.dll
```

```
C:\WINNT\system32\cmd.exe
D:\Assembler>nasmw.exe -fwin32 -O3 add_sub.s
D:\Assembler>golink.exe -entry start add_sub.obj kernel32.dll
GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk
Output file: add_sub.exe
Format: win32 size: 1,536 bytes
D:\Assembler>
```

그림 2. 실행파일 만들기

모든 과정이 성공적으로 끝났다면, OllyDbg 로 add_sub.exe 파일을 불러 옵시다. OllyDbg CPU 화면에서 주소 00401005h 부분과 0040100Fh 부분을 주목해 주십시오.

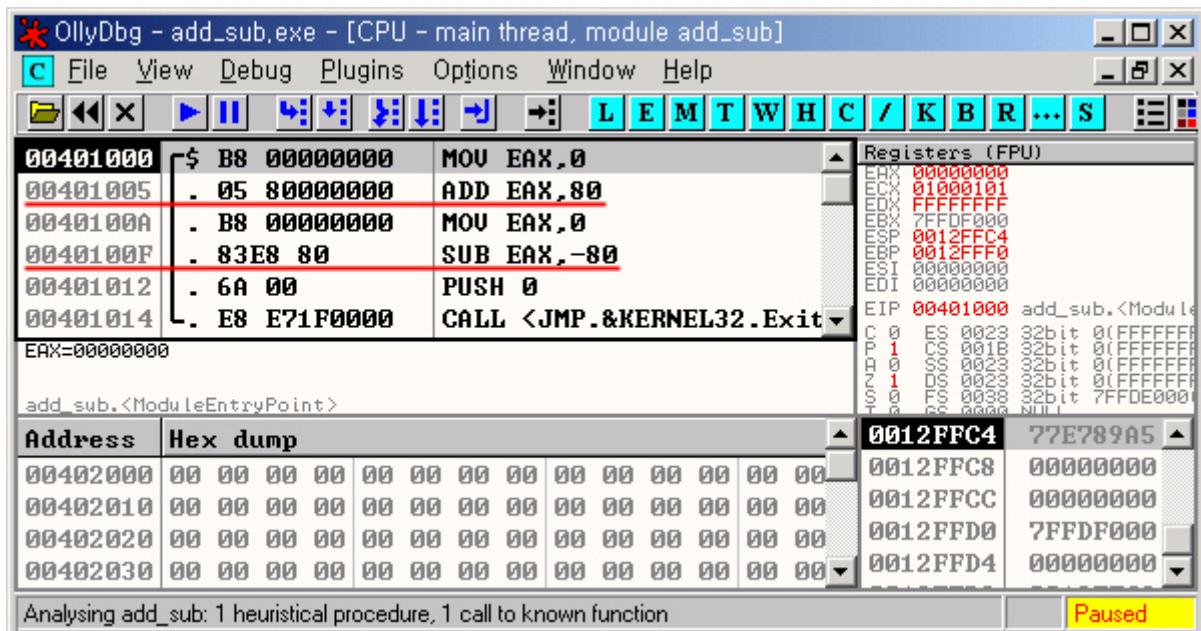


그림 3. OllyDbg 로 add_sub.exe 파일 불러오기

OllyDbg 의 「Hex dump」 부분이 각각 6 바이트(05 80000000h)와 3 바이트(83E8 80h)인 것을 알 수 있습니다. 이것은 기계어로 표현할 때 나타내는 부분이므로, ADD 를 사용한 명령보다 SUB 를 사용한 명령 쪽이 3 바이트 작다는 것입니다. 반 바이트로 같은 연산을 할 수 있으므로 코드 최적화를 할 수 있습니다. F7 키로 스텝실행 하면 00401005h 주소의 【ADD EAX,80】 연산과 0040100Fh 주소의 【SUB EAX,-80】 연산 결과 값은 모두 동일한 080h(128)이라는 것을 알 수 있습니다. 0 에서 128 까지의 덧셈 명령은 0 에서 -128 까지 뺄셈명령으로 치환할 수 있다는 의미입니다. 동일한 결과를 표현하는데 있어 메모리 사이즈를 줄일 수 있는 방법이 있다는 것입니다.

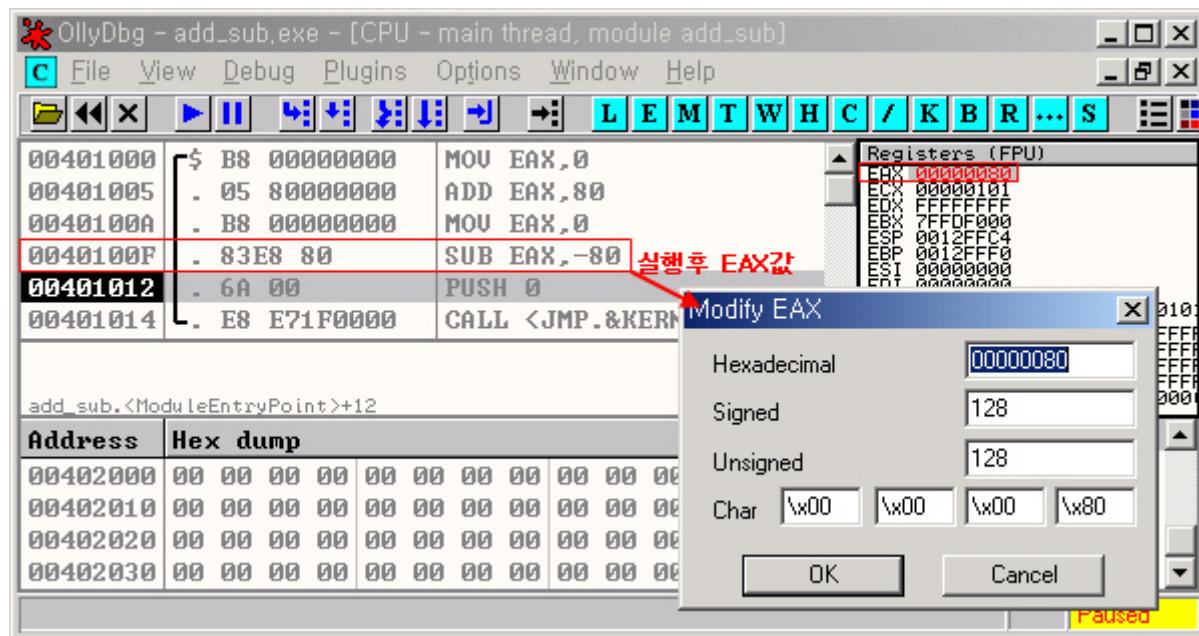


그림 4. 0040100Fh 주소 실행 후 EAX 값의 변화

레지스터 값 0으로 초기화

프로그램에서는 초기 값으로 0이라는 수치를 자주 사용합니다. 해당 변수에 임의의 쓰레기 값이 들어가지 않도록 미리 방지하기 위함입니다. 이처럼 단순히 변수를 0으로(레지스터 값을 0으로) 하는 것만 해도 여러가지 방법이 있습니다. 일반적으로 생각하면 아래와 같이 MOV 명령을 사용한 코드가 떠오를 것입니다.

```
mov eax,00000000h
```

수치를 직접 사용하는 명령은 시간이 걸리기 때문에 많은 컴파일러에서는 아래와 같은 명령을 생성합니다.

```
xor eax,eax
```

XOR 명령에는 같은 값의 비트면 0이 되는 특성이 있으므로 컴퓨터 입장에서는 더 빠르게 동작할 수 있습니다. 또한 x라는 값에서 x라는 값을 감산하면 0이 되기 때문에, SUB 명령을 사용하여 아래와 같은 코드를 사용할 수 있습니다.

```
sub eax,eax
```

실제 동일한 결과 값이 나오는지 아래와 같은 어셈블리어를 만들어 실행해 봅시다.

크랙에 필요한 어셈블리어 이해

```
 xor_sub.s

1 section .bss
2 buf resb 10
3 section .text
4     global start
5 start:
6     mov eax,12345678h
7     mov eax,0
8     mov eax,12345678h
9     xor eax,eax
10    mov eax,12345678h
11    sub eax,eax

12
13    push 0
14    extern ExitProcess
15    call ExitProcess
16
```

그림 5. xor_sub.s 어셈블리 파일

오브젝트 파일 생성과 실행파일 생성을 마친 후(위의 add_sub.s 와 동일과정) OllyDbg 로 add_sub.exe 파일을 불러옵시다(그림 6).

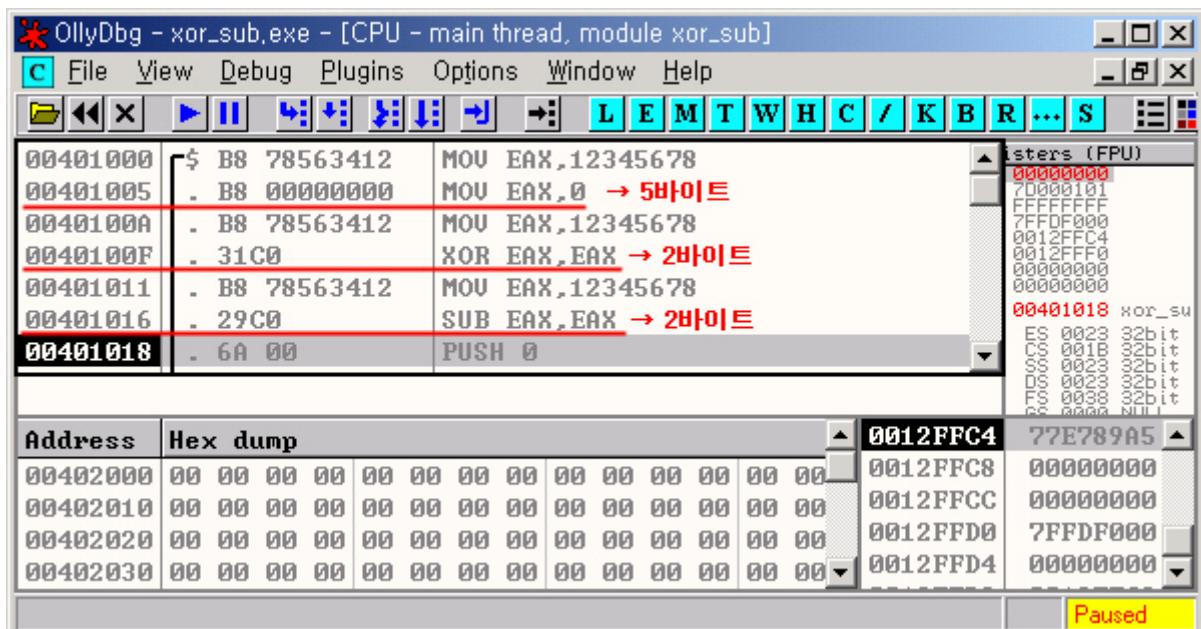


그림 6. OllyDbg 로 xor_sub.exe 파일을 불러온 화면

00401005h 주소의 MOV 명령에서는 수치 0 을 직접 사용해서 EAX 값을 초기화 시켰고, 5 바이트 길이가 사용되었습니다. 0040100Fh 주소의 XOR 명령과 00401016h 주소 SUB 명령 둘다 2 바이트만 사용하여 EAX 값을 초기화 시켰으므로 MOV 명령보다 처리 속도가 빨라 집니다. F7 키로 단계별 실행을 수행해보면 모두다 EAX 값이 0 으로 초기화 되는 것을 확인할 수 있습니다.

레지스터 값 -1로 초기화

-1이라는 수치도 잘 사용됩니다. 레지스터 값을 -1로 하는 방법을 봅시다. 아래 4개의 예를 들겠습니다.

```
mov eax,ffffffffffh      ;5 bytes  
  
xor eax,eax      ;2 bytes  
dec eax          ;1 bytes  
  
sub eax,eax      ;2bytes  
dec eax          ;1 byte  
  
stc              ;1byte  
sbb  eax,eax     ;2bytes
```

위의 3개는 레지스터를 0으로 만드는 방법으로 기본적으로 모두 동일합니다. 2번째와 3번째는 0으로 한 뒤 DEC 명령으로 -1로 만드는 것입니다.

4 번째 경우는 매우 특수합니다. 컴파일러가 생성하지 않는 경우입니다. 플래그 제어 명령군의 하나인 STC 명령에서 CF(캐리 플러그)를 세트(1)하고 있습니다. 다음에 SBB 명령으로 제 2 연산자(오른쪽의 EAX)와 CF(캐리 플러그)를 가산하여, 그 결과를 제 1 연산자(왼쪽 EAX)에서 뺀 후 제 1 연산자에 결과값을 보관합니다. 즉, STC 명령에서 CF가 1이 되어 있기 때문에, [SBB EAX,EAX]는

$$EAX - (EAX + CF) = EAX - (EAX + 1) = EAX - EAX - 1 = -1$$

이라는 계산이 수행되고 있는 것입니다. 패커나 프로텍터 등으로 랜덤 코드를 작성하는 엔진이 붙어있는 것이라면 이같은 코드를 생성할 수 있습니다.

실제 확인하려면 아래와 같이 initial_-1.s 어셈블리 소스 파일을 컴파일하여 OllyDbg로 불러와 직접 확인하면 됩니다. 각각 4 가지 경우 EAX 값이 어떻게 -1이 되는지 단계별 실행하여 확인해 보시기 바랍니다.

The screenshot shows a Notepad window titled "initial_-1.s" containing assembly code. The code defines a global variable "buf" at offset 10 in the .bss section, and a function "start" at address 0. The "start" function initializes "buf" to FFFFFFFFh, then performs a series of operations (XOR, DEC, SUB, STC, SHB) to clear the byte, and finally pushes 0 onto the stack before calling the ExitProcess function twice.

```
1 section .bss
2 buf resb 10
3 section .text
4 global start
5 start:
6     mov eax,0ffffffffffh
7
8     xor eax,eax
9     dec eax
10
11    sub eax,eax
12    dec eax
13
14    xor eax,eax
15    stc
16    shb eax,eax
17
18
19    push 0
20    extern ExitProcess
21    call ExitProcess
```

그림 7. “-1”로 초기화 어셈블리 소스 코드(initial_-1.s)

The screenshot shows a Command Prompt window with the title "C:\WINNT\system32\cmd.exe". It displays the commands used to assemble and link the source code. The user runs "nasmw.exe -fwin32 -O3 initial_-1.s" to generate an object file, and then "golink.exe -entry start "initial_-1.obj" kernel32.dll" to produce the final executable. The output shows GoLink version 0.26.4, copyright information, and the creation of "initial_-1.exe" with a size of 1,536 bytes.

```
D:\Assembler>nasmw.exe -fwin32 -O3 initial_-1.s
D:\Assembler>golink.exe -entry start "initial_-1.obj" kernel32.dll
GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk
Output file: initial_-1.exe
Format: win32 size: 1,536 bytes
```

그림 8. 어셈블리 컴파일 및 링크

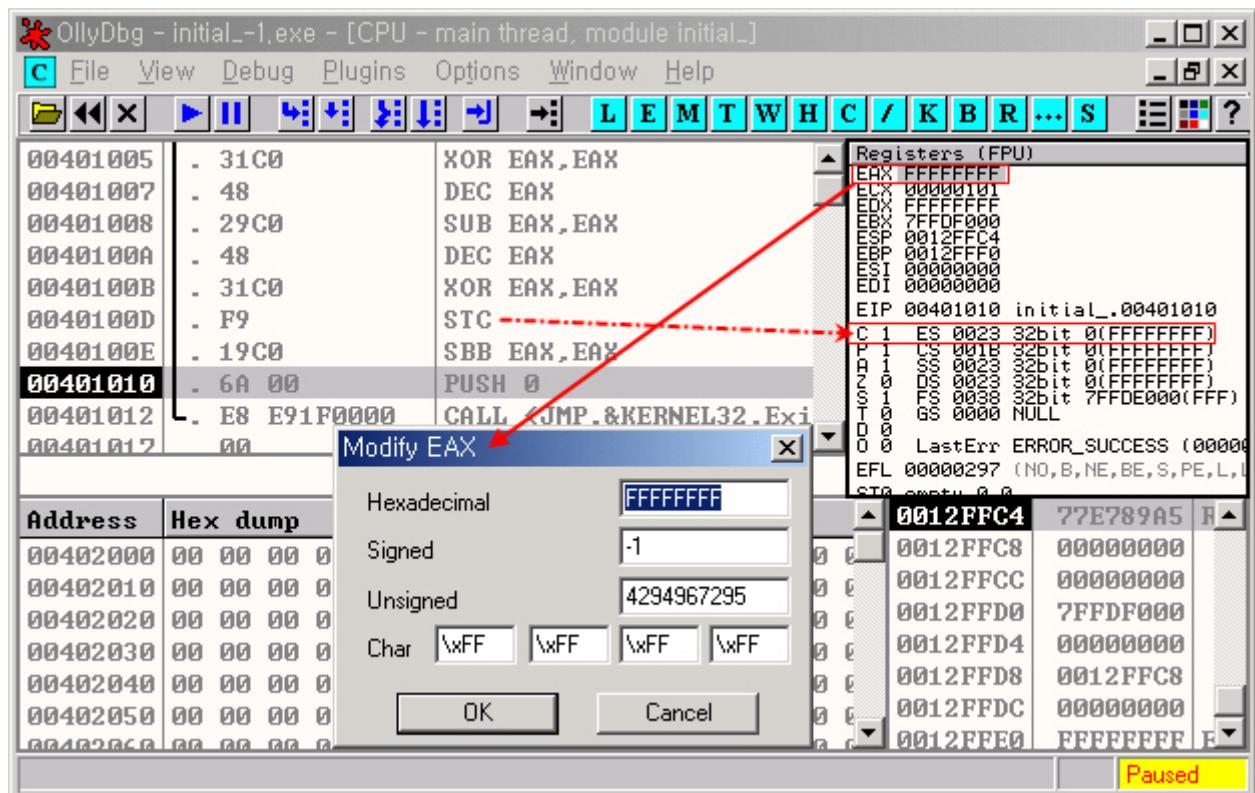


그림 9. OllyDbg로 불러온 후 실행된 화면

0 혹은 1을 체크하여 조건 점프(인증 성공)

크래킹에 있어서 조건 점프는 가장 중요 체크포인트입니다. 특히 조건에 매치하고 있는지 아닌지 조사하는데 EAX 가 0 인지 1 인지 비교하는 것이 가장 많이 나옵니다. 웹으로 인증하는 경우 인증 성공(Ture: “1 or 0”), 인증 실패(False: “-1”)가 패킷 데이터로 전달되는 경우도 많습니다. 이와 같이 0 혹은 1 을 체크하는 경우는 인증이 성공했는지 점검하는 경우에 주로 이용됩니다.

아래 3개 예시는 EAX가 0인지 아닌지를 조사한 후 0 일 때 _label_에 점프하는 명령입니다(만약 EAX 값이 1 인지 조사한 후 점프하는 명령은 JNE 가 사용).

```

cmp eax, 00000000h ; 5 bytes
je _label_           ; 2/6 bytes (short/near)

or eax,eax          ; 2 bytes
je _label_           ; 2/6 bytes (short/near)

test eax,eax        ; 2 bytes
je _label_           ; 2/6 bytes (short/near)

```

크랙에 필요한 어셈블리어 이해

첫번째 CMP 명령을 사용하여 eax 값이 0 인지 체크하는 것은 가장 자연스런 명령이라 할 수 있습니다. 어셈블리 언어의 기본 형태입니다.

두번째 OR 명령은 각 비트가 0 으로된 쌍이라면 0 이 되고, 그렇지 않을 경우는 1 이 됩니다. 결과적으로 0 이 되면 ZF(제로플러그)가 세트됩니다. TEST 명령은 OR 명령과 같은 동작을 하지만, 플래그에만 영향을 줍니다. 전부 EAX 가 0 일 때 ZF(Zero Flag)가 세트되기 때문에, 다음의 JE 명령으로 ZF 가 세트되어 있을 때는 _label_에 점프합니다.

JE 명령은 _label_까지의 거리가 가까운(1 바이트로 나타낼 수 있는) short 점프인지, 긴(1 바이트로 나타낼 수 없는) near 점프인지에 의해 기계어로 번역되었을 때 사이즈가 달라집니다. 각각 2 바이트와 6 바이트입니다.

EAX 가 0 인지 아닌지를 비교할 경우는 CMP 명령보다는 OR 명령이나 TEST 명령이 사용될 경우가 많습니다. 꼭 기억해 두십시오.

또한 조금 특수한 경우(컴파일러가 생성하지 않음), 다음과 같은 명령어로 처리되는 경우도 있습니다.

```
xchg eax,ecx      ; 1 bytes  
jecxz _label_      ; 2 bytes
```

이것은 JECXZ 명령이 CX 가 0 일 때 점프하는 것을 이용한 경우입니다. XCHG 명령으로 EAX 와 ECX 값이 교환되므로 사용시에 주의가 필요하지만, 사이즈만 보면 위의 3 개보다 작은 사이즈로 동일한 명령을 수행 할 수 있습니다. 컴파일러는 이렇게 명령을 생성하지는 않습니다. 그러나 바이러스 제작자나 리버싱을 어렵게하기 위해 어셈블리 언어로 작성된 프로그램에 나올 수가 있습니다. 안티 리버싱이나 프로텍터를 분석할 때 볼 수 있습니다.

-1(fffffffffh)을 체크하여 조건 점프(인증 실패)

앞서 설명했던 것과 같이, 이번에는 -1(0xffffffffh)인지 판별하여 조건 점프하는 예를 살펴 보겠습니다.

```
cmp eax,ffffffffh    ;5 bytes  
je _label_          ; 2/6 bytes  
  
inc eax             ; 1 byte  
je _label_          ; 2/6 bytes
```

첫번째 줄의 CMP 명령을 사용하는 것이 가장 일반적인 방법입니다. 쉽게 이해할 수 있을 것입니다. 두번째 INC 명령을 사용한 방법은 EAX 가 -1 일 때에 INC 명령에서 1 을 증가시키면 EAX 가 0 이 됩니다. 즉 ZF(제로플래그)가 세트(1) 된다는 것을 이용한 것입니다. 단, 이 처리에서는 EAX 레지스터 값이 1 이 증가 되어버리기 때문에, 필요하다면 DEC 명령으로 EAX 값을 되돌려 주어야 합니다. 그래도 DEC 명령은 1 바이트 명령이기 때문에 첫번째

크랙에 필요한 어셈블리어 이해

CMP 명령을 사용한 것보다 더 작은 사이즈로 원하는 결과를 수행 할 수 있습니다.

MOVZX 명령의 이해

MOVZX 명령을 다르게 바꾸어 보고, MOVZX 명령 처리에 대하여 생각해 보겠습니다. 아래 2 개의 서로 다른 코드는 동일한 결과를 수행할 수 있습니다.

```
xor eax,eax          ; 2 bytes  
mov ax,word [esi+xx]    ; 4 bytes  
  
movzx eax,word [esi+xx]  ; 4 bytes
```

MOVZX 명령은 AX 에 어드레스 [ESI+xx]에 있는 워드값을 복사하여 EAX 상위 16 비트를 0 으로 채우고 있습니다.
아래 코드는 워드값이 아닌 바이트 값으로 치환한 예제입니다.

```
xor eax,eax          ; 2 bytes  
mov al,bytes[esi+xx]     ; 3 bytes  
  
movzx eax,byte [esi+xx]   ; 4 bytes
```

MOVZX 명령은 AL 레지스터에 [ESI+xx]에 있는 바이트값을 복사하여 EAX 의 상위 24 비트를 0 으로 채우고 있습니다. MOVZX(MOVSX)명령은 사용하려는 레지스터나 메모리 비트의 크기를 항상 지정해야 된다는 점을 유의해 주십시오.

또 레지스터끼리 복사할 때에도 아래와 같은 방법을 사용할 수 있습니다.

```
xor eax,eax          ; 2 bytes  
mov ax,bx             ; 3 bytes  
  
movzx eax,bx          ; 3 bytes
```

MOVZX 명령은 사이즈 최적화에 자주 사용됩니다.

레지스터에 수치 입력

크랙에 필요한 어셈블리어 이해

MOV 명령은 수치를 레지스터에 대입하는 기본적인 명령입니다. 아마도 어셈블리 언어에서 가장 많이 사용될 것입니다.

```
mov eax,50h ; 5 bytes
```

속도는 느리지만 PUSH, POP 명령을 조합하여 수치를 레지스터에 넣을 수 있습니다.

```
push 50h ; 2 bytes  
pop eax ; 1 byte
```

PUSH, POP 명령을 사용하여 레지스터에 값을 대입할 경우 사이즈를 줄일 수 있습니다. Embed System 과 같이 작은 메모리 공간에서는 크기가 상당히 중요할 수 있습니다.

다음과 같이 PUSH 를 많이 하는 명령을 봅시다.

```
push 0 ; 2 bytes  
push 0 ; 2 bytes
```

수치를 직접 대입하여 사용하는 것은 사이즈가 커지기 때문에 레지스터를 사용하여 사이즈를 줄일 수 있습니다.

```
xor eax,eax ; 2 bytes  
push eax ; 1 byte  
push eax ; 1 byte
```

똑같은 것을 몇번이나 반복하고 있기 때문에 LOOP 명령을 사용하면 사이즈를 더 줄일 수 있습니다.

```
push 7 ; 2 bytes
```

크랙에 필요한 어셈블리어 이해

```
pop ecx          ; 1 byte  
_label_ : push 0      ; 2 bytes  
loop _label_      ; 2 bytes
```

LOOP 은 ECX 를 1 감소 시킨 후 ECX 가 제로가 아니라면 _label_에 점프하는 명령입니다. 결국 [push 0]이 7 번 반복되는 것이 됩니다.

여기에서 예제로 설명한 것은 매우 특수한 사례이지만, 하나의 명령을 수행하는데 있어서 다른 방법도 많이 존재한다는 것을 아는 것만으로도 어셈블리 언어를 공부하는데 많은 도움을 줄 것입니다. 또한 디스어셈블한 코드를 해독하는데 있어서도 플러스로 작용한다고 봅니다.

메모리간 대입

메모리에서 메모리로 대입하는데 있어서 MOV 명령과 PUSH, POP 명령을 사용했을 때 각각의 차이를 간단히 나타내었습니다.

```
push eax          ;1 byte  
mov eax,dword[ebp+xxx] ;6 bytes  
mov dword [ebp+xxx],eax ; 6 bytes  
pop eax          ; 1 byte  
  
push dword [ebp+xxx] ; 6 bytes  
pop dword[ebp+xxx] ; 6 bytes
```

IMUL 명령 이해

곱셈을 하는 명령어중에서 부호가 달린 수치를 곱셈하는 IMUL(Integer multiply)명령을 이해하기 위한 아래 3 개 코드를 보십시오.

```
mov ecx, 28h          ; 5 bytes  
mul ecx            ; 2 bytes  
  
push 28h            ; 2 bytes  
pop ecx            ; 1 byte  
mul ecx            ; 2 bytes
```

```
imul eax,eax,28h ; 3 bytes
```

첫번째와 두번째의 차이는 MOV 명령을 PUSH, POP 명령으로 하고 있을 뿐입니다. 세번째는 IMUL 명령을 사용하고 있습니다. IMUL 명령은 두번째 레지스터를 세번째 연산자로 곱하여, 첫번째 연산자의 레지스터에 저장합니다.

LODSx(LOAD String) 명령 이해

메모리로부터 레지스터에 데이터를 로드하는 LODSB(Load string byte) 명령 동작을 MOV 명령과 INC 명령으로 나타내면 첫번째 코드가 됩니다. AL 레지스터에 어드레스 【ESI】에서 바이트값을 추출하여 INC 명령으로 ESI 레지스터를 1 증가(byte)시키고 있습니다.

```
mov al,[esi] ;2bytes
inc esi ;1byte

lodsb ;1byte
```

LODSW(Load string word) 명령 동작은 다음과 같습니다. AX 레지스터에 어드레스 【ESI】에서 워드값을 꺼내어 ESI 레지스터를 2 증가(word)시킵니다.

```
mov ax,[esi] ;3bytes
inc esi ;1byte
inc esi ;1byte

lodsw ;2bytes
```

LODSD(Load string dword) 동작은 다음과 같습니다. EAX 레지스터에 어드레스 【ESI】로부터 더블 워드값을 꺼내어 ESI 레지스터를 4 증가(dword)시킵니다.

```
mov eax,[esi] ;2bytes
add esi,4 ;3byte

lodsd ;1byte
```

크랙에 필요한 어셈블리어 이해

또한 LODSD, STOSD(STOrd String Double)의 패턴은 다음과 같이 바꿔 쓸 수 있습니다.

```
mov eax,[esi]          ; 2 bytes  
add esi,4             ; 3 bytes  
mov dword [edi],eax   ; 2 bytes  
add edi,4             ; 3 bytes  
  
lodsd                ; 1 bytes  
stosd                ; 1 bytes
```

STOSD 명령은 EAX 레지스터 값을 어드레스 【EDI】 가 가리키는 위치에 저장하며, EDI 를 4 증가(dword)시킵니다.

결론적으로 【MOV DWORD [EDI],EAX】 → 【ADD EDI,4】 가 STOSD 와 같은 결과가 나오는 처리를 수행합니다.

문자열 끝 확인

문자열 처리 프로그램은 어셈블리 언어에서 보면 매우 번거롭습니다. 여기에서는 문자열 처리의 기본적인 예제를 살펴 보고, 문자열 끝을 찾는 다양한 방법중에서 대표적인 것들을 알아 보겠습니다.

프로그램에서 문자열을 표현할 경우, 대다수 문자열의 끝부분에 0 바이트 값을 추가하여 문자열 끝을 나타냅니다. 문자열 끝을 알면 전체 길이를 알 수 있습니다. 아래 프로그램에서는 asciiz 를 문자열의 시작 어드레스로, 0 은 문자열 끝을 나타내고 있습니다.

```
        mov esi,asciiz      ; 5 bytes  
s_check:  
        lodsb            ; 1 bytes  
        test al,al       ; 2 bytes  
        jne s_check      ; 2 bytes
```

먼저 ESI에 asciiz의 주소를 넣습니다. LODSB 명령으로 ESI가 가리키고 있는 어드레스에서 1바이트를 꺼내어 AL 레지스터에 대입하여 ESI 값을 1 만 증가시킵니다. TEST 명령에서 AL 에 0 이 들어있는지 아닌지를 조사하여, 0 이 아닐 때 JNE 명령으로 라벨 s_check 로 점프하여 같은 동작을 반복합니다.

```
        mov edi, asciiz    ; 5 bytes  
        xor al,al        ; 2 bytes  
s_check:  
        scasb           ; 1 byte
```

jne s_check	; 2 bytes
-------------	-----------

다음은 SCASB 를 사용하여 EDI 의 값과 AL 의 값을 비교하는 형태입니다.

문자열 끝을 찾는 방법은 여러가지 있습니다. 프로그램 토작에 따라 컴파일러에 따라 문자열 끝을 찾는 방법이 모두 다르므로 다양한 형태를 스스로 생각해 보는 것도 좋은 공부 방법입니다.

```

1 section .bss
2 string1 resb 10
3 buf resb 10
4 section .data
5 asciz db 'strsend',0
6 section .text
7 global start
8
9 start:
10    mov esi,asciiz ; 5 bytes
11 str_check:
12    lodsb ; 1 bytes
13    test al,al ; 2 bytes
14    jne str_check ; 2 bytes
15
16    mov edi,asciiz ; 5 bytes
17    xor al,al ; 2 bytes
18
19 str_check2:
20    scash ; 1 byte
21    jne str_check2 ; 2 byte
22
23    push 0
24    extern ExitProcess
25    call ExitProcess
26

```

그림 10. 문자열 끝 테스트 어셈블리 소스 파일(Strings_end.s)

```

C:\WINNT\system32\cmd.exe
D:\Assembler>nasmw.exe -fwin32 strings_end.s
D:\Assembler>golink.exe -entry start strings_end.obj kernel32.dll
GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk
Output file: strings_end.exe
Format: win32 size: 2,048 bytes
D:\Assembler>_

```

그림 11. 어셈블리 컴파일 및 링크

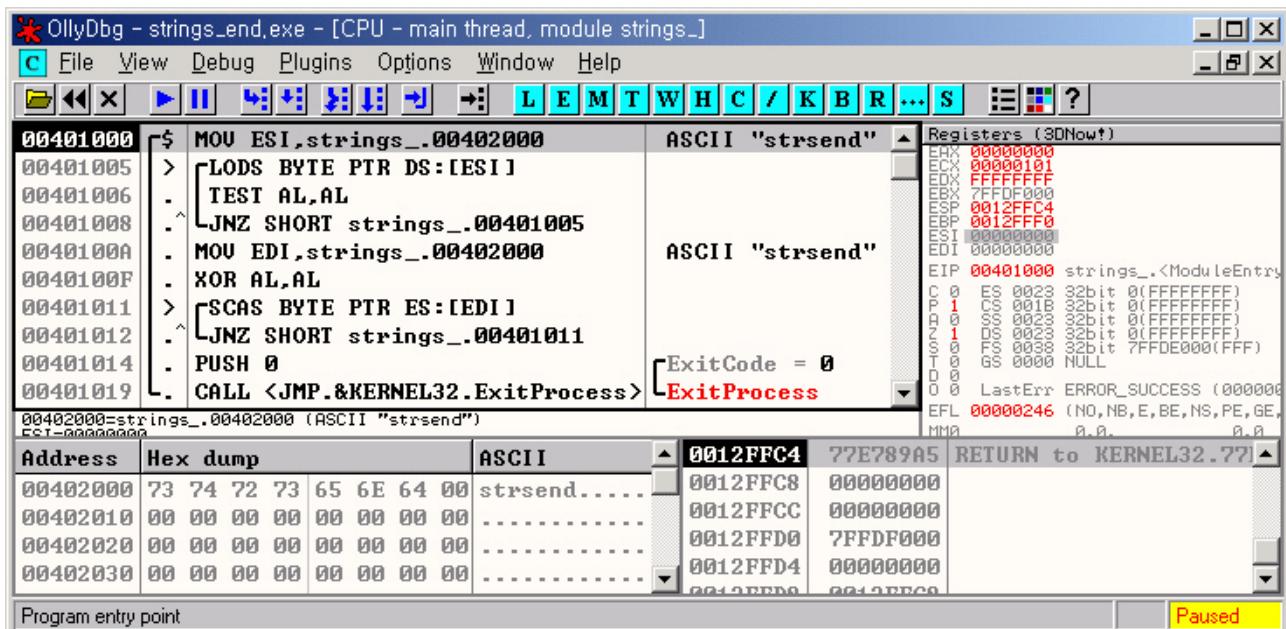


그림 12. OllyDbg 로 불러온 화면

EDX 초기화

EAX 가 080000000h 이하일 때 EDX 를 0 으로 초기화 하는데는 CDQ 명령을 사용할 수 있습니다.

```
cdq ; 1 byte
```

이것은 매우 특수한 예이지만 알아두면 좋습니다.

2의 n승 곱셈과 나눗셈

DIV(divide) 명령을 사용한 나눗셈 예제를 먼저 하나 보겠습니다. 1000h(4096)을 4 로 나누는 예제 입니다.

```
mov eax,1000h          ; 5 bytes
mov ecx,4              ; 2 bytes
xor edx,edx            ; 2 bytes
div ecx
```

위의 나눗셈은 아래와 같이 하면, 공간을 더 작게 차지하고 속도도 빨라집니다.

shr eax,2 ; 3 bytes

2의 n승 나눗셈은 SHR 명령으로 n번 이동 시키면 가능합니다. 2로 나누려면 [SHR EAX,1], 4로 나누려면 [SHR EAX,2], 8로 나누려면 [SHR EAX,3]이 됩니다. 이 명령은 자주 사용되므로 반드시 기억해 둡시다. 이해가 안되면 자신이 실제로 수치를 2진수로 적어보고, 최하위 비트를 잘라 버린 후, 10진수로 변환하여 확인하면 됩니다.

mov ecx,4 ; 5 bytes mul ecx ; 2 bytes
--

위의 곱셈은 나눗셈일 때와 반대로 비트를 이동 시켜서 아래와 같이 작성 할 수 있습니다.

shl eax,2 ; 3 bytes

2의 n승 곱셈은 SHL 명령으로 n회 이동시키면 가능합니다(SHL 명령에서 최하위 비트에 0이 들어감). 2배 하려면 [SHL EAX,1], 4배 하려면 [SHL EAX,2], 8배 하려면 [SHL EAX,3]이 됩니다. 이것도 자주 사용되므로 외워 두시면 좋습니다.

The screenshot shows a Notepad window with the file name 'shx_cmd.s'. The code is written in NASM assembly language:

```
1 section .bss
2 buf resb 10
3 section .text
4     global start
5
6 start:
7     mov eax,1000h ; 5 bytes
8     mov ecx,4      ; 2 bytes
9     xor edx,edx    ; 2 bytes
10    div ecx
11
12    mov eax,1000h
13    shr eax,2
14
15    mov eax,16
16    shr eax,3
17
18    mov eax,4
19    mov ecx,4      ; 5 bytes
20    mul ecx       ; 2 bytes
21
22    mov eax,4
23    shl eax,2      ; 3 bytes
24
25    push 0
26    extern ExitProcess
27    call ExitProcess
```

그림 13. 나눗셈과 곱셈 예제 소스(shx_cmd.s)

The screenshot shows a Command Prompt window with the title 'C:\WINNT\system32\cmd.exe'. The logs show the compilation and linking process:

```
C:\>Assembler>nasmw.exe -fwin32 shx_cmd.s
D:\Assembler>golink.exe -entry start shx_cmd.obj kernel32.dll
GoLink.Exe Version 0.26.4 - Copyright Jeremy Gordon 2002/6-JG@JGnet.co.uk
Output file: shx_cmd.exe
Format: win32 size: 1,536 bytes
D:\Assembler>
```

그림 14. 어셈블리 컴파일 및 링크

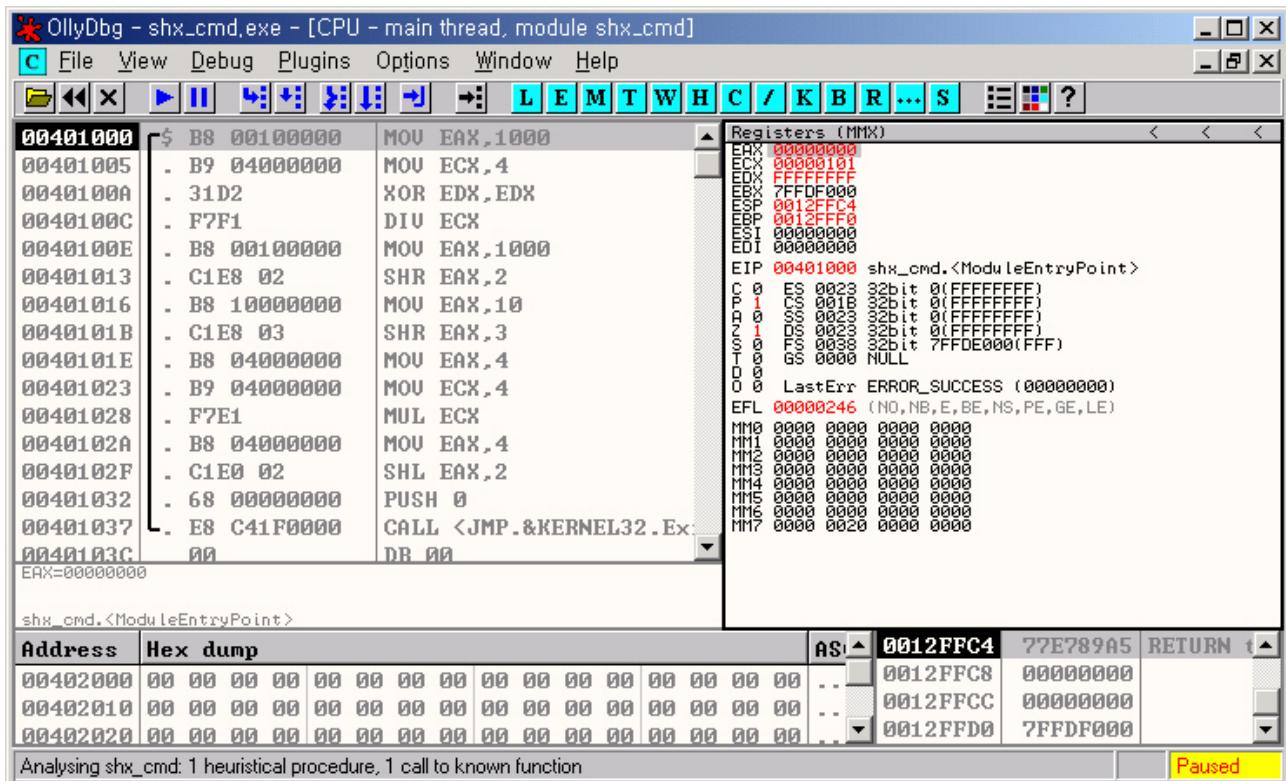


그림 15. OllyDbg 로 불러온 화면

루프(LOOP) 명령의 개요

일반적인 어셈블리 언어에서 루프 구문은 아래와 같습니다.

```
loop _label_ ; 2 bytes(SHORT)
```

위 LOOP 명령은 다음 조건점프와 동일합니다.

```
    dex ecx ; 1 byte
    jne_label_ ; 2 bytes(SHORT)
```

XCHG 명령의 개요

일반적으로 두 피연산자의 값을 서로 바꾸는 명령의 예는 아래와 같습니다.

```
push eax ; 1 byte
push ebx ; 1 byte
```

크랙에 필요한 어셈블리어 이해

```
pop eax          ; 1 byte  
pop ebx          ; 1 byte
```

위 명령은 아래의 레지스터 교환 XCHG 명령과 동일합니다.

```
xchg eax,ebx    ;2bytes
```

한쪽의 레지스터 값을 남겨둘 필요가 없다면 MOV 명령을 사용하는 편이 더 빨라집니다.

```
mov ecx, edx    2bytes
```

끝으로

컴파일러가 생성하는 코드는 최적화를 생각한 코드가 기본이므로 패턴이 어느정도 정해져 있습니다. 또한, 컴파일러 특성에 의해 컴파일러마다 특징적인 코드(StartUP 코드⁶¹, 변수 범위등)가 생성되는 형태가 많기 때문에, 몇개 크랙을 실제 해보면 쉽게 작성된 프로그램언어와 로직을 알아낼 수 있습니다. 숙달된 리버서라면 10 줄~20 줄 사이의 어셈블리코드가 표현하는 함수들은 자연스럽게 직독이 가능합니다. 그러나 더 나아가 랜덤한 코드를 생성하는 프로텍터(Linear, NonLinear, Multi layer encrypt, Random decryption등)나 처리된 소프트웨어(MtE, TPE, ACG, DPSPG, Tapion, Spi, DAME, APME, PEGD, SVKP등) 및 리버싱 지연을 위해 코드 난독화를 해놓은 경우, 디버깅 과정중에 이른바 절대로 컴파일러가 생성하지 않는 것 같은 의미없는 처리나 긴 명령군을 보게 됩니다(Garbage data injection, Garbage call function injection). 여기서부터는 온갖 방법들이 동원됩니다. 필자는 A4 용지를 길게 늘여 붙여서 프로그램에 실제 영향이 되는 코드들만 정렬해 나갑니다. 모니터를 세로로 뒤집어 디버거에서 일일이 라벨링하는 수고도 필요합니다. 이런 실제 리버스 엔지니어링 과정에서 보듯이, 이번 장에서 설명한 것은 아주 일부에 지나지 않습니다. 이것을 참고하여 자신이 여러가지를 생각해 보면 좋을 것입니다.

⁶¹ 사용자가 작성한 코드가 실행되기 이전에 프로그램 실행에 필요한 기본적인 초기화 작업을 담당하는 코드를 말합니다.

크래에 ~~필요한~~ 어셈블리에 의해

제 3 장 Windows 크랙 기본편

3.9 SoftICE Debugger

들어가며

최근 몇 년 사이 윈도우 디버거로 OllyDbg 가 주로 이용되고 있지만, Windows95/98 이 주류 운영체제로 사용되던 시대에는 주로 Compuware 사의 SoftICE 디버거를 사용하였습니다. 현재와는 달리 디버거의 종류도 적고, 또한 막강한 기능을 가진 것은 SoftICE 외에는 별로 없었습니다. 현재 SoftICE 는 Compuware 사의 DriverStudio/DevPartner 제품에 포함되어 있습니다.

Windows95 가 1995 년 릴리즈 되고서도 10 여년 경과한 현재에도 크랙에 필요한 커널 디버거로서 SoftICE 가 사용되고 있습니다. 1995 년부터 지금까지도 SoftICE 를 사용한 많은 크랙 Tutorial 문서들이 존재합니다. 그 Tutorial 에는 SoftICE 가 동작하고 있는 것을 전제로 기술되어 있는 것이 대부분으로, SoftICE 환경을 접하지 못한 분에게는 의미없는 것 밖에 안됩니다.

최근 인기 있는 OllyDbg 와 비교하면 Windows XP 이상에서 쓸수 있도록 설정하는 것은 어렵겠지만, 꼭 SoftICE 설치하여 인터넷상에 있는 Crack/Reversing Tutorial 을 따라해 보시기 바랍니다.

※ 이러한 막강한 커널 디버거를 만든 Compuware 사는 2006 년 4 월경 SoftICE(DriverStudio 포함) 개발을 중단한다고 선언했습니다.

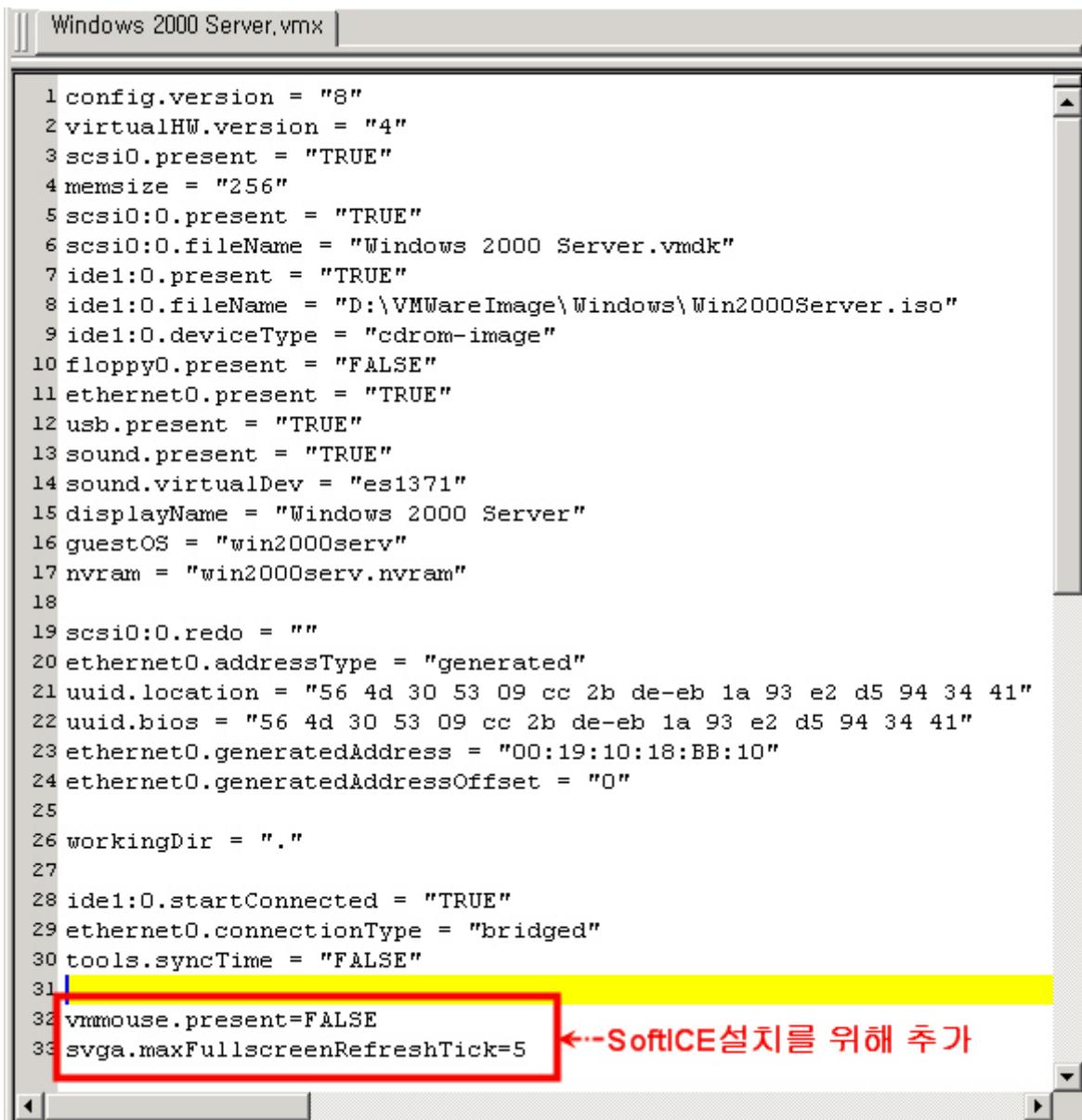
SoftICE 설치

먼저 SoftICE 를 설치하기 전에 시스템 백업을 해 두면 좋습니다. 최악의 경우 설치에 실패하면 운영체제가 동작되지 않기 때문입니다. 개인적으로는 Ghost, DriveImage, True Image, Maestro recovery 와 같은 디스크 전체를 이미지로 백업하고 복구 할 수 있는 유ти리티들을 추천합니다.

그러나 대부분 SoftICE 설치를 위해 별도의 가상 원도우인 VMWare 를 주로 이용합니다. VMWare 에 인스톨할 경우는 가상 머신 설정 파일(.vmx)을 열어 아래 두 행을 추가해 주십시오.

```
vmmouse.present=FALSE  
svga.maxFullscreenRefreshTick=5
```

※ SoftICE 플러그인들은 몇몇 버전에서만 동작됩니다. 최신 SoftICE 버전에서는 플러그인 대다수가 동작하지 않으므로, 본인의 환경에 적합한 SoftICE 를 설치하시기 바랍니다.



```

1 config.version = "8"
2 virtualHW.version = "4"
3 scsi0.present = "TRUE"
4 memsize = "256"
5 scsi0:0.present = "TRUE"
6 scsi0:0.fileName = "Windows 2000 Server.vmdk"
7 ide1:0.present = "TRUE"
8 ide1:0.fileName = "D:\VMWareImage\Windows\Win2000Server.iso"
9 ide1:0.deviceType = "cdrom-image"
10 floppy0.present = "FALSE"
11 ethernet0.present = "TRUE"
12 usb.present = "TRUE"
13 sound.present = "TRUE"
14 sound.virtualDev = "es1371"
15 displayName = "Windows 2000 Server"
16 guestOS = "win2000serv"
17 nram = "win2000serv.nram"
18
19 scsi0:0.redo = ""
20 ethernet0.addressType = "generated"
21 uuid.location = "56 4d 30 53 09 cc 2b de-eb 1a 93 e2 d5 94 34 41"
22 uuid.bios = "56 4d 30 53 09 cc 2b de-eb 1a 93 e2 d5 94 34 41"
23 ethernet0.generatedAddress = "00:19:10:18:BB:10"
24 ethernet0.generatedAddressOffset = "0"
25
26 workingDir = "."
27
28 ide1:0.startConnected = "TRUE"
29 ethernet0.connectionType = "bridged"
30 tools.syncTime = "FALSE"
31
32 vmmouse.present=FALSE
33 svga.maxFullscreenRefreshTick=5
  
```

--SoftICE설치를 위해 추가

그림 1. SoftICE 설치를 위한 VMWare 환경 설정파일 수정.

svga.maxFullscreenRefreshTick 은 0 이상의 값을 주면 되는데 값이 낮을 수록 SoftICE 의 키입력, 명령에 대한 결과가 화면에 보여지는 속도는 높아지나 시스템 성능에 영향을 주므로, 본인의 컴퓨터에서 값을 낮추면서 적절한 값을 찾아야 합니다. VMWare 사에서는 기본적으로 “5”를 권장하고 있습니다.

필자는 DriverStudio 3.2.0, VMWare 가상 윈도우에서 Windows 2000 Server SP4 운영체제에 설치할 것입니다.

먼저 설치 CD-ROM에서 autorun.exe를 실행 시키면 SETUP 화면을 볼 수 있습니다.

SoftICE 설치 및 환경 설정

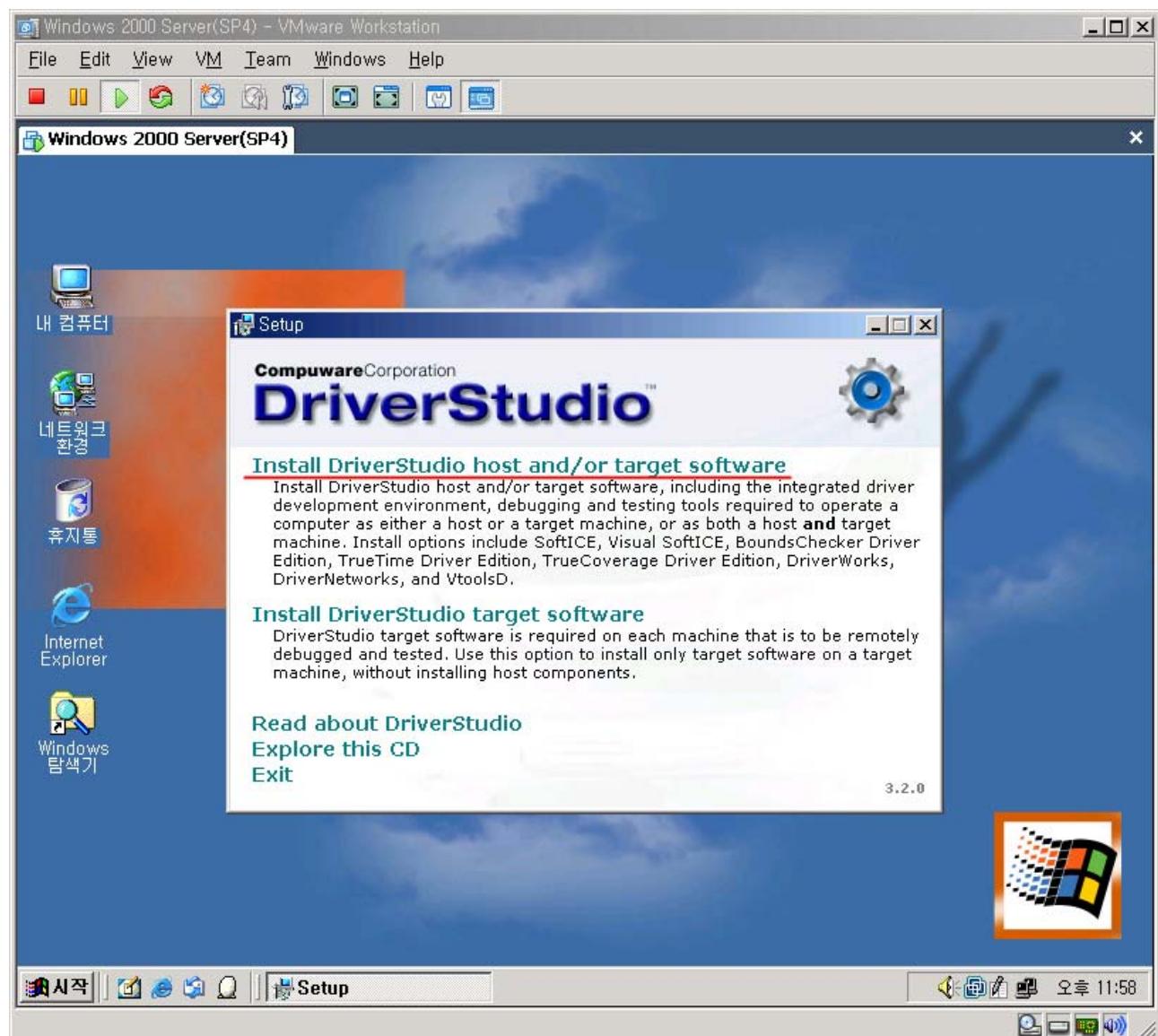


그림 2. SETUP 화면

"Install DriverStudio host and/or target software"를 선택하여 로컬 및 원격에서 양쪽다 디버깅 할 수 있도록 설치를 진행합니다.

라이센스 동의(License Agreement)를 수락하면 고객 정보 입력창이 나타납니다. 제품 구매 완료시 회신 받은 등록번호와 구매정보를 입력합니다.

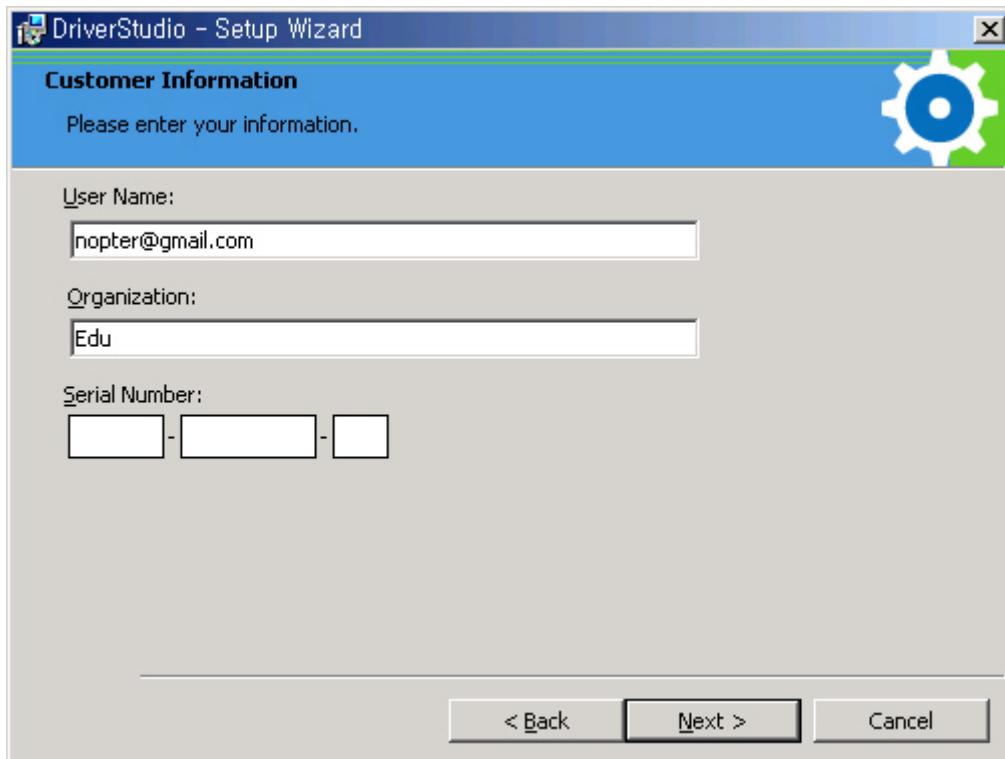


그림 3. 고객 정보 입력창

라이센스는 파일로 개별 등록하거나 라이센스 서버로부터 등록하여야 합니다. 주문 완료시 받았던 라이센스 파일을 선택하시면 됩니다.

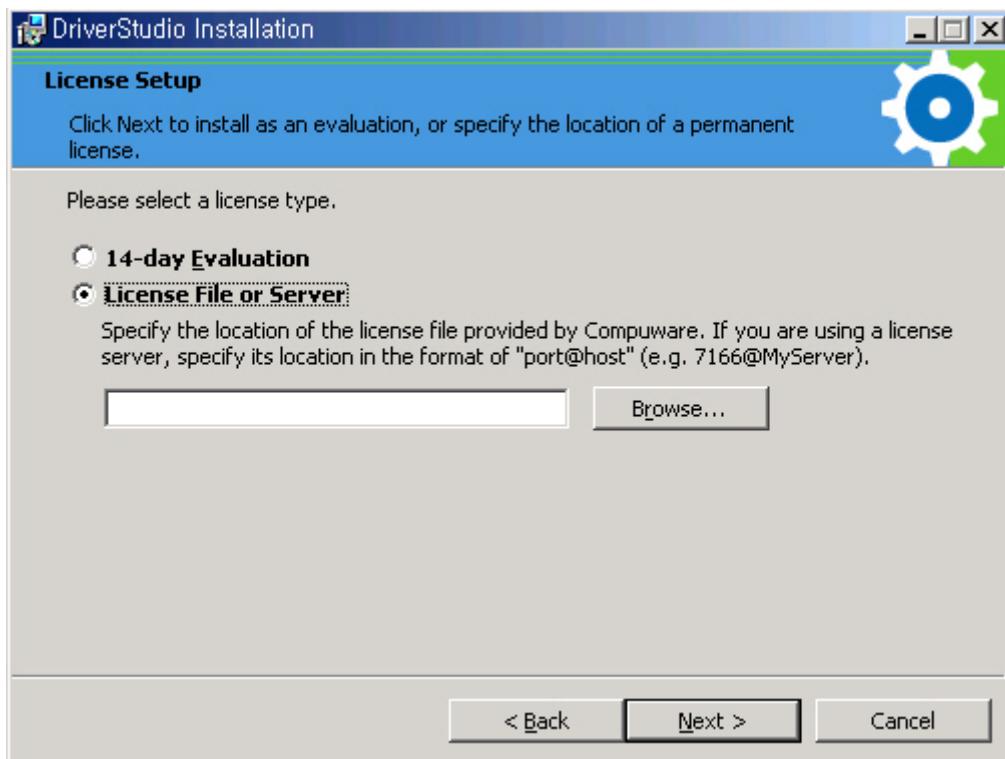


그림 4. 라이센스 등록

SoftICE 설치 및 환경 설정

설치 디렉터리를 적당히 지정하면 머신(Machine) 환경설정 화면이 나오게 됩니다. SoftICE는 원격에서 디버깅이 가능합니다. 로컬 호스트로 설치를 진행할지 아니면 원격 호스트로 진행할지 선택하지만, Host/Target 모두 디버깅과 분석을 위해 “Both host and target”을 선택하도록 하겠습니다.

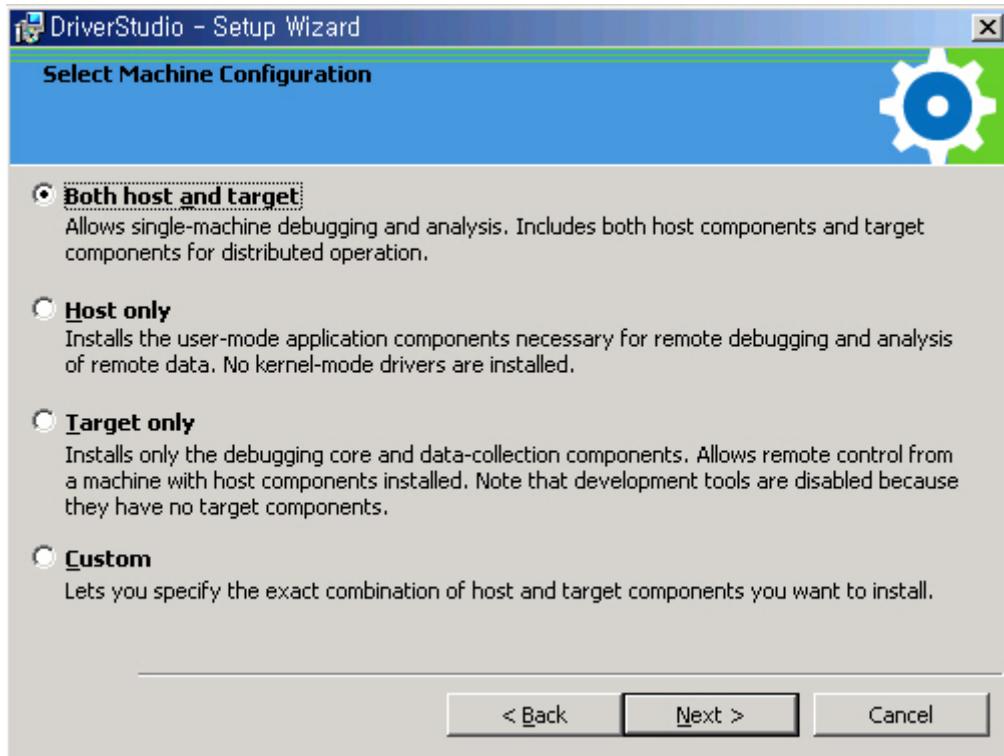


그림 5. Machine 환경 선택

기능 선택 화면에서 모든 기능을 선택한 후 다음 단계로 진행하여 최종 설치를 완료하면 됩니다. 각 기능들에 대해서는 차후 설명하도록 하겠습니다. 만약 개발환경을 사용하지 않고 디버깅 환경만 구축하려면 “Development tools” 항목은 미체크 하시면 됩니다.

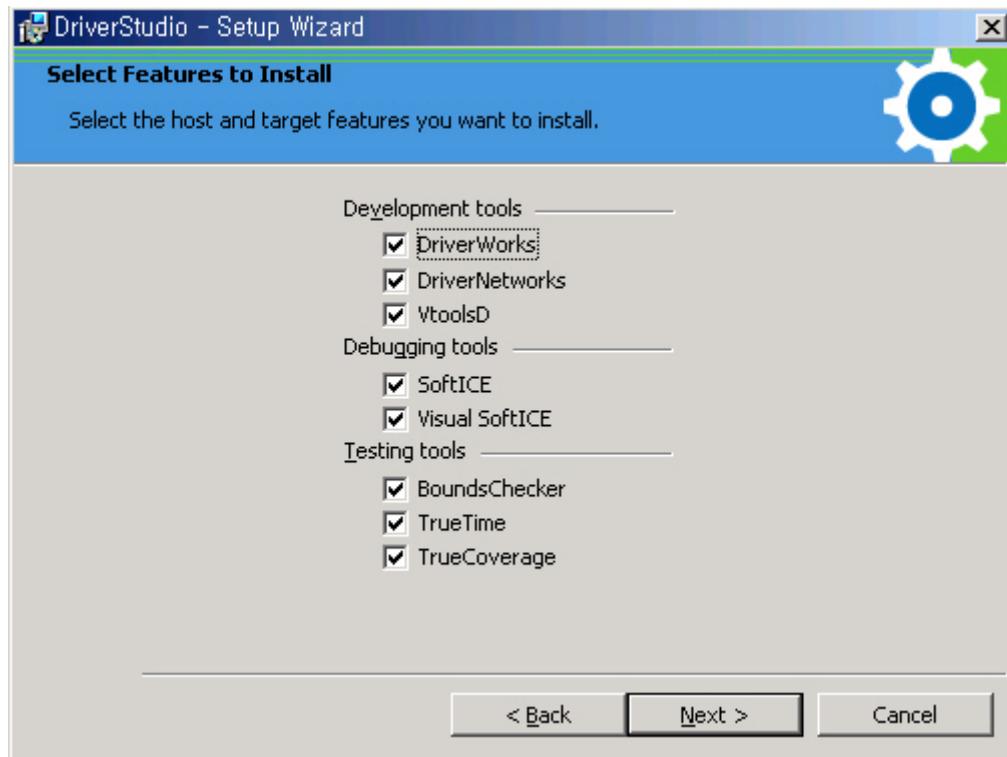


그림 6. 기능 선택 화면

VtoolsD 기능을 선택하였다면 SoftICE 가 디버깅 가능하도록 USER,MAK 를 생성하도록 합시다. VtoolsD 는 Windows Millennium Edition 이하 윈도우에서 가상 디바이스 드라이버 개발을 손쉽게 하기 위한 툴입니다. VtoolsD 보다는 더욱 지능화되고 사용하기 편한 DriverWorks 를 더 많이 사용합니다. 이러한 개발환경은 C/C++에서만 지원됩니다.

파일 복사가 모두 끝나면 내 컴퓨터에서 SoftICE 를 실행시키기 위한 환경설정 화면이 나오게 됩니다. “Control Panel\WStartup”항목은 아래 그림처럼 기본 설정상태를 유지해 주시기 바랍니다. 단지 Windows 95/98 에서 SoftICE 를 동작시키려면 【Boot】옵션으로 변경하셔야 합니다. Windows NT4 이상부터는 【Manual】로 선택하더라도 언제든지 SoftICE 를 동작 시킬 수 있습니다.

SoftICE 설치 및 환경 설정

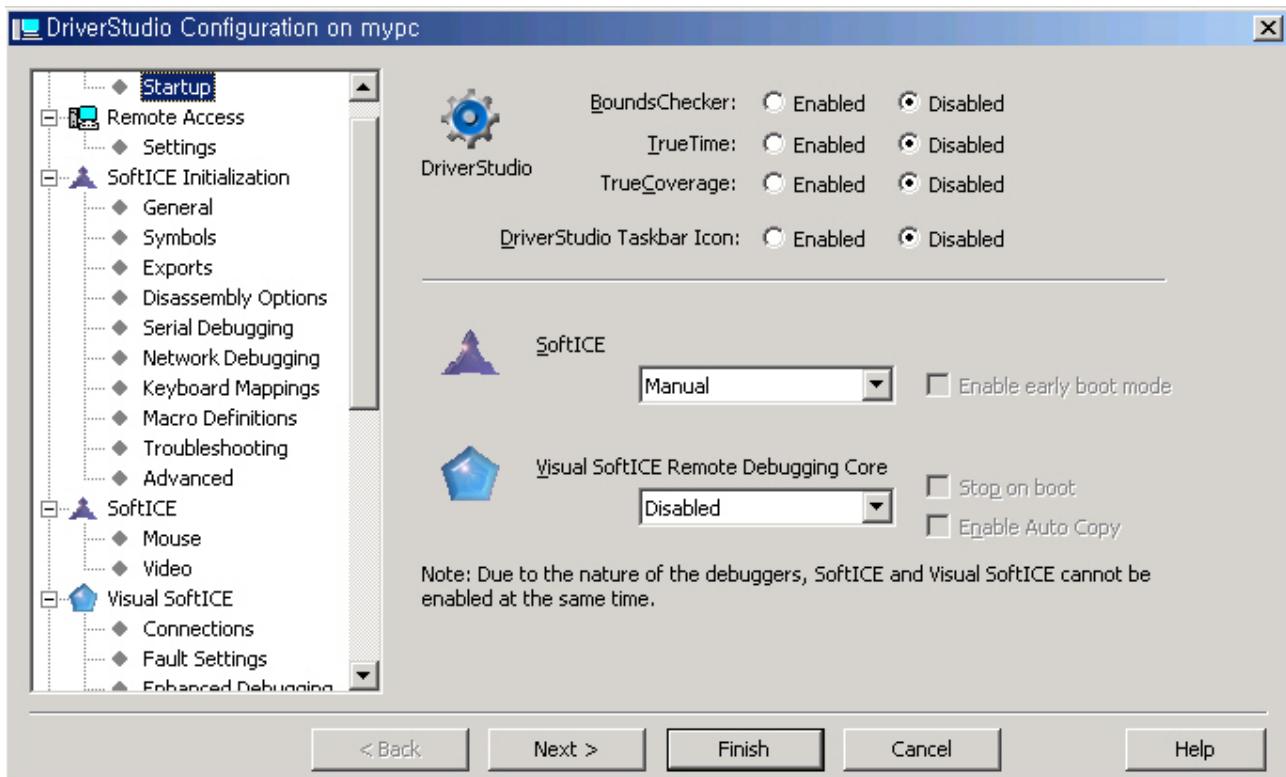


그림 7. Control Panel 의 Startup 환경 설정

“SoftICE Initialization\WGeneral”의 “Initialization strings”메뉴에서 SoftICE 를 사용하기 편리하도록 라인수와 폭을 미리 정의해두면 좋습니다(lines 120; width 160; wc 60;X;).

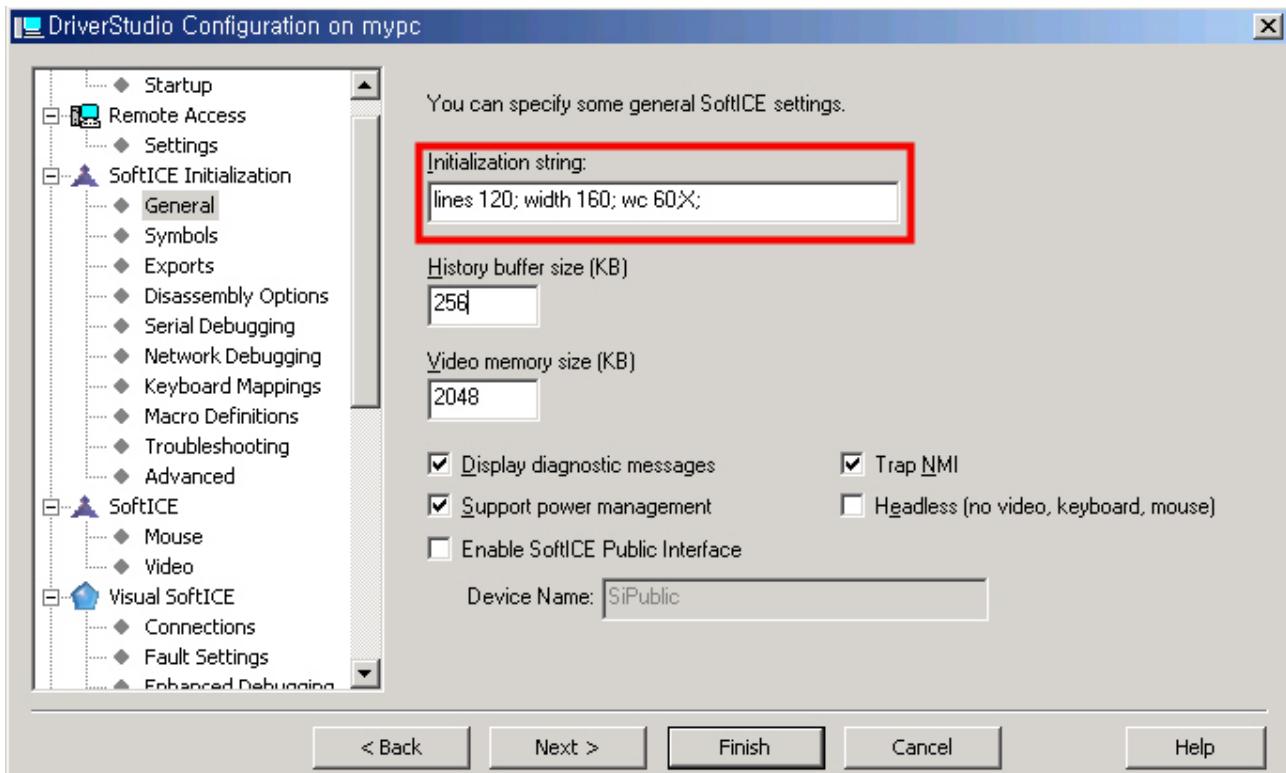


그림 8. General 옵션 설정

SoftICE 설치 및 환경 설정

“SoftICE InitializationWSymbols”의 “Symbol Retriever”에서 최신 심볼을 다운로드⁶² 받아 SoftICE 기동시 사용할 수 있도록 하면 디버깅시 더욱 유용한 정보를 획득할 수 있습니다. 설치 완료 후 Symbols Loader/Retriever를 통해서도 환경 설정을 할 수 있습니다.

“SoftICEWMouse”에서 마우스 종류를 설정합니다. [PS/2 compatible or USB]를 선택합니다. 또한 휠이나 두개 이상의 버튼을 가진 마우스를 사용하는 분은 아래의 [Enhanced Mouse]를 체크 해 주십시오. 만약 마우스 사용으로 문제가 발생될 경우는 [None]을 선택해 주십시오.

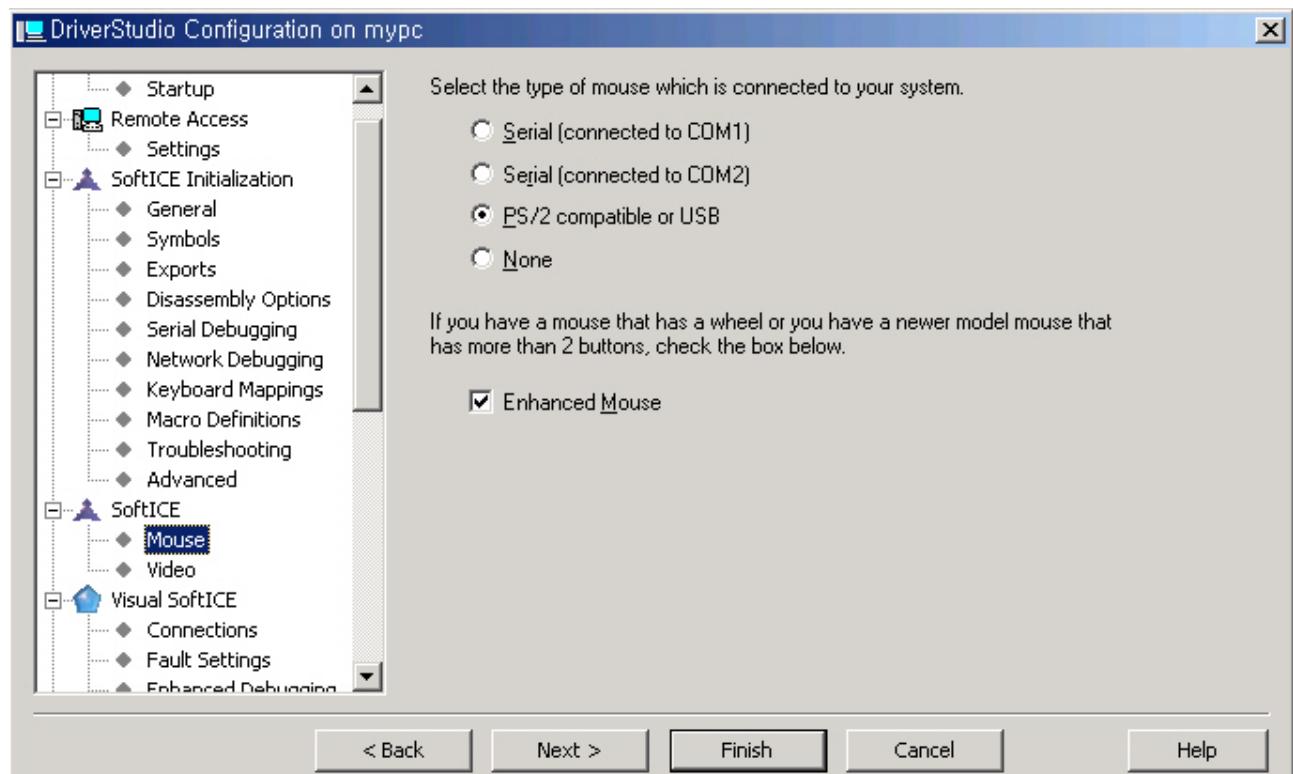


그림 9. 마우스 설정

“SoftICEWVideo”메뉴는 SoftICE 가 동작하는데 중요한 부분입니다. 여기에서는 반드시 [Universal Video Driver (SoftICE appears in a “window”)]를 선택한 후 “Test”버튼을 눌러 비디오 카드 드라이버가 동작하는데 문제가 없는지 테스트하도록 합시다. 정상적인 경우라면 [SoftICE Universal Video Driver test completed successfully.]라는 메시지 박스가 표시됩니다.

⁶² Symbols Download: <http://msdl.microsoft.com/download/symbols>

SoftICE 설치 및 환경 설정

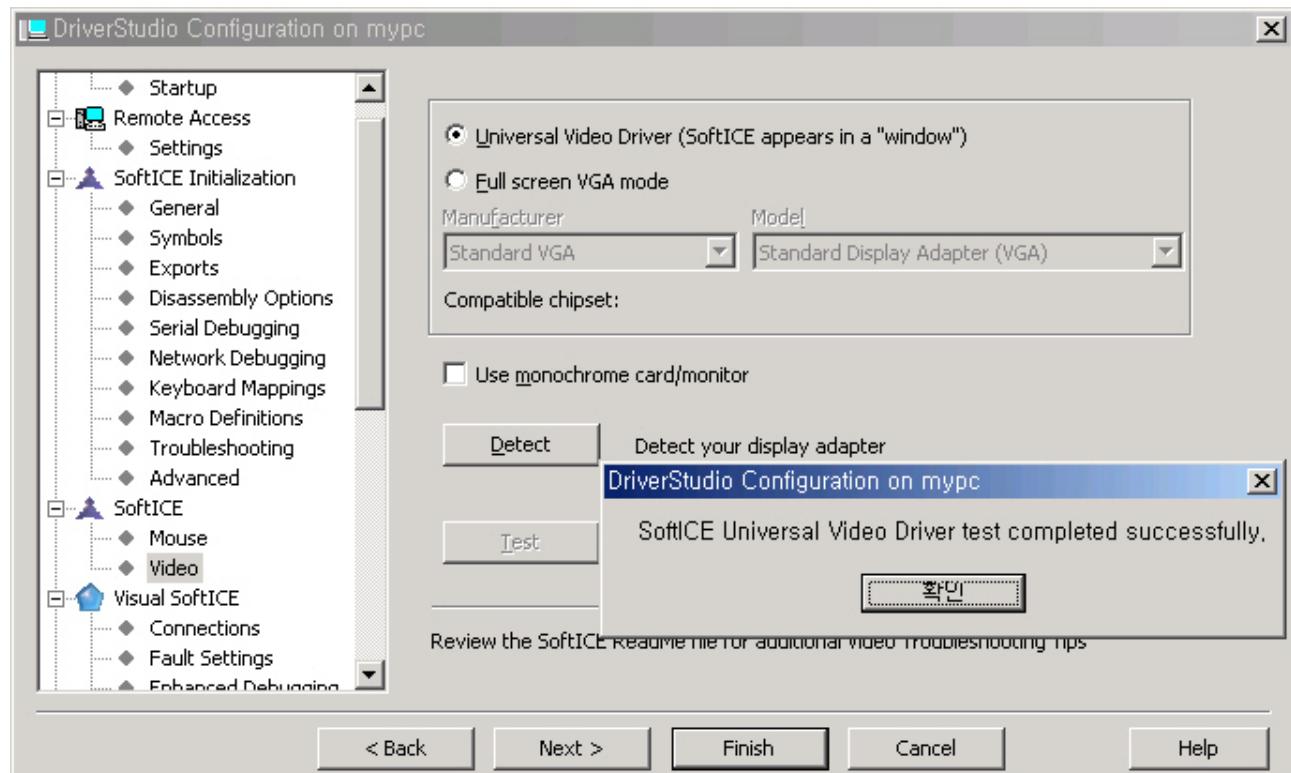


그림 10. 비디오 드라이버 테스트

이것으로 SoftICE 를 기본적으로 실행할 수 있는 환경설정을 모두 마쳤습니다. “Finish”버튼을 눌러 시스템을 재부팅 합니다.

Startup 설정을 Manual 로 한 경우 SoftICE 서비스를 시작해 주거나 원도우 메뉴의 “Start SoftICE”를 클릭해주면 디버깅 상태로 진입할 수 있습니다.

SoftICE 설치 및 환경 설정

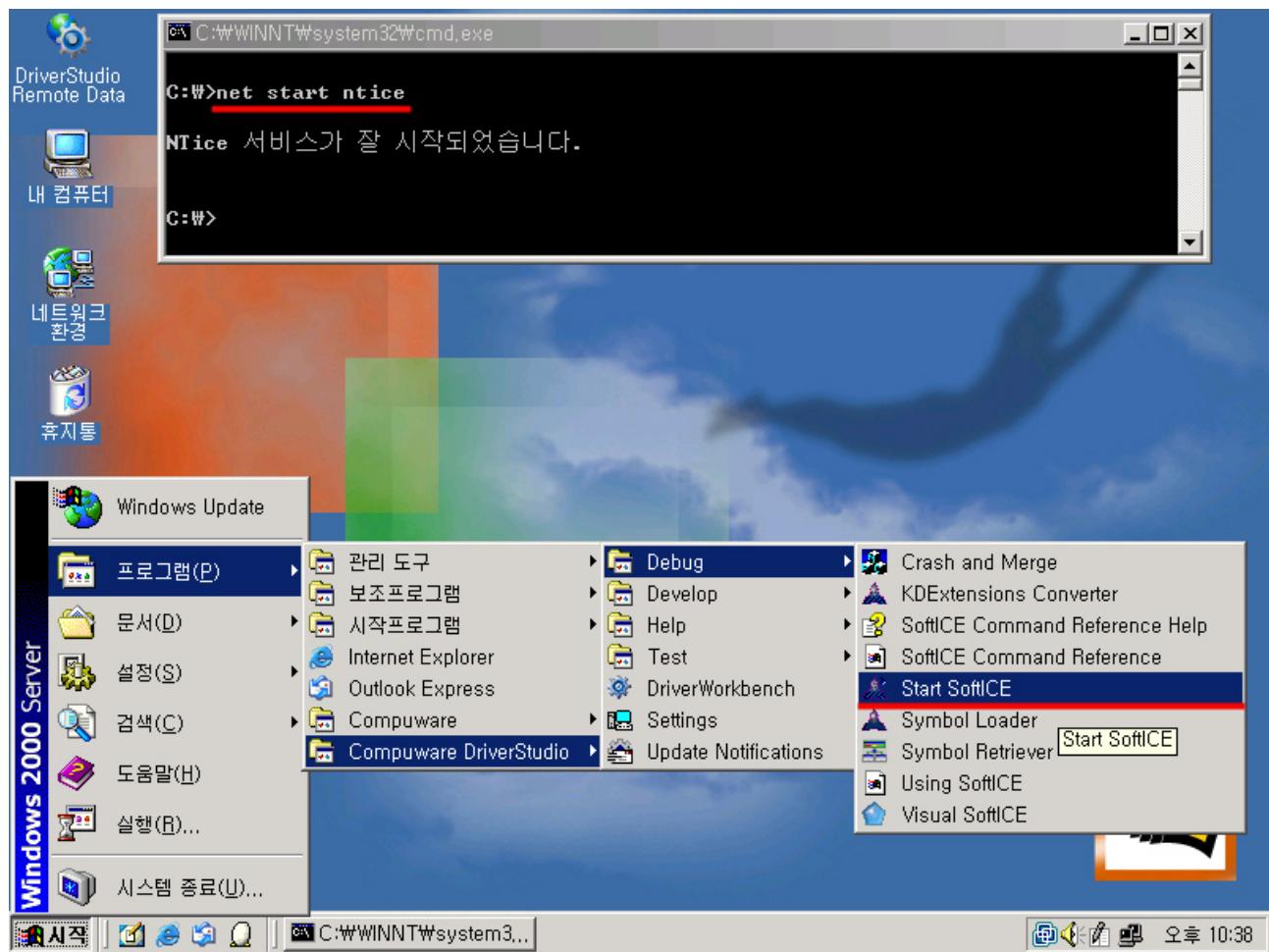


그림 11. SoftICE 서비스 시작

SoftICE 환경 설정 파일인 "winice.dat"를 통해 세부적인 설정을 할 수 있습니다. SoftICE 보조 프로그램인 [Symbol Loader]를 사용하여 GUI를 통한 설정도 가능하지만, 재 사용 가능한 winice.dat 파일을 직접 편집하는 것이 좋습니다. Winice.dat 파일은 Windows NT 계열인 경우 [%SYSTEMROOT%\system32\drivers\winice.dat]에 Windows 9x 계열은 [%SYSTEMROOT%\system\winice.dat]에 저장되어 있습니다. 파일을 텍스트 편집기로 열어 보면 설치과정중에 설정했던 내용들이 들어 있습니다.

행번호	Winice.dat 내용
1	NMI=ON
2	VERBOSE=ON
3	
4	HST=256
5	DRAWSIZE=2048
6	INIT="lines 120; width 160; wc 80;"
7	INIT="SET ORIGIN 600 120; WATCH ES:EDI; WATCH DS:ESI; FAULT OFF;"
8	INIT="CODE ON; WR 3; WF F; WW 2; WD 8; WC 33; SET BreakInShareMods ON; X;"

SoftICE 설치 및 환경 설정

9	SYM=512
10	
11	DISASSEMBLYHINTS=ON
12	LOWERCASE=OFF
13	CODEMODE=OFF
14	
15	SELECTORS=ON
16	CHECKSTRINGS=ON
17	AUTOCONNECT=OFF
18	NETSUPPORT=OFF
19	HOSTNAME=MYPC
20	F1="h;"
21	F2="^wr;"
22	F3="^src;"
23	F4="^rs;"
24	F5="^x;"
25	F6="^ec;"
26	F7="^here;"
27	F8="^t;"
28	F9="^bpX;"
29	F10="^p;"
30	F11="^G *SS:ESP;"
31	F12="^p ret;"
32	SF3="^format;"
33	AF1="^wr;"
34	AF2="^wd;"
35	AF3="^wc;"
36	AF4="^ww;"
37	AF5="CLS;"
38	AF11="^dd dataaddr->0;"
39	AF12="^dd dataaddr->4;"
40	CF1="altscr off; lines 60; wc 32; wd 8;"
41	CF2="^wr;^wd;^wc;"
42	MACROS=32
43	
44	MOUSE=ON

SoftICE 설치 및 환경 설정

	45 ECHOKEYS=OFF
	46 NOLEDS=OFF
	47 NOPAGE=OFF
	48 PENTIUM=ON
	49 THREADP=ON
	50 SIWVIDRANGE=ON
	51 MENU=Copy , NMPD_COPY , 0
	52 MENU=Paste , NMPD_PASTE , 0
	53 MENU=Copy&Paste , NMPD_COPYANDPASTE , 0
	54 MENU=Display , NMPD_DISPLAY , 0
	55 MENU=Un-Assemble , NMPD_UNASSEMBLE , 0
	56 MENU=What , NMPD_WHAT , 0
	57 MENU=Prev , NMPD_PREV , 0
	58 MENU=Reip , r eip %cp% , 0
	59 MENU>Add Watch , watch %cp% , 0
	60 MENU=Break On Text , bpx %cp% , 0
	61 MENU=Name , name %cp% , 4
	62 ; WINICE.DAT
	63 ; (SystemRoot\Windows\System32\WINICE.DAT)
	64 ; for use with SoftICE for Windows NT (versions 3.0 and greater)
	65 ;
	66 ; ***** Examples of export symbols that can be included *****
	67 ; Change the path to the appropriate drive and directory
	68 EXP=C:\Windows\System32\version.dll
	69 EXP=\Windows\System32\hal.dll
	70 EXP=\Windows\System32\ntoskrnl.exe
	71 EXP=\Windows\System32\ntdll.dll
	72 EXP=\Windows\System32\kernel32.dll
	73 EXP=\Windows\System32\user32.dll
	74 EXP=\Windows\System32\csrssrv.dll
	75 EXP=\Windows\System32\basesrv.dll
	76 EXP=\Windows\System32\winsrv.dll

설명
○ 4 행: HST=4096

SoftICE 설치 및 환경 설정

Command 윈도우의 히스토리 버퍼 사이즈를 KB 로 지정합니다. 명령창에 입력한 명령의 결과나 브레이크한 함수 어ドレス 등이 표시됩니다. 기본적으로 256KB 로 되어 있지만 여유를 가지고 4096KB 로 지정해 두면 좋습니다.

○ 5 행: DRAWSIZE=2048

비디오 카드 탑재 메모리를 KB 로 지정합니다. 비디오 카드 탑재 메모리가 작으면 큰 화면으로 표시할 수 없습니다. 또 비디오 카드 탑재 메모리가 충분히 있어도 SoftICE 는 비디오 카드 탑재 메모리를 자동 인식하지 않기 때문에 DRAWSIZE 로 지정하지 않으면 큰 화면으로 표시할 수 없습니다.

○ 6 행: INIT="lines 120; width 160; wc 80;"

화면에 나타낼 라인수와 폭, 문자갯수를 정의합니다. 위의 설정은 1020*768 화면에 맞추어져 있습니다. 본인의 컴퓨터 환경에 맞춰 변경할 수 있습니다.

○ 7 행: INIT="SET ORIGIN 600 120; WATCH ES:EDI; WATCH DS:ESI; FAULT OFF;"

○ 8 행: INIT="CODE ON; WR 3; WF F; WW 2; WD 8; WC 33; SET BreakInShareMods ON; X;"

SoftICE 실행시에 초기화 하고 싶은 명령을 기술합니다. 6,7,8 행으로 나누어져 있지만 보기 쉽게 나눈 것 이므로, 한행으로 기술해도 문제 없습니다. 중요한곳은 [X;]와 [FAULT OFF;]입니다.

먼저 [FAULT OFF;]는 일반 보호모드 예외나 페이지 예외가 일어나도 SoftICE 를 팝업하지 않도록 지정합니다. 이 옵션을 지정하지 않으면 SoftICE 는 무언가 에러가 일어난 것을 감지하여 자동으로 팝업합니다. [X;]는 SoftICE 화면에서 제외 됩니다. 이것은 INIT 행 마지막에 기술해야 합니다. [X;] 이후에 명령어를 기술하면 다음 팝업 후에 명령이 실행됩니다.

[SET ORIGIN 600 120;]은 화면의 어떤 위치에 SoftICE 를 팝업 시킬지 지정합니다. 이 옵션을 설정하지 않으면 화면 중앙에 SoftICE 가 팝업 합니다. 만약 DriverStudio2.7 이후 버전의 SoftICE 를 설치 한 분은 [SET BreakInShareMods on]을 INIT 행에 추가하십시오.

○ 20 행 ~ 41 행

이 행들은 키보드 단축키를 정의하고 있습니다. 미리 정의된 단축키를 변경하지 않습니다. 익숙해질 때까지는 변경하지 않는 것이 좋습니다.

○ 42 행: MACROS=32

SoftICE 에서는 매우 편리하게 매크로를 정의할 수 있습니다. 이 행에서는 정의할 수 있는 매크로 갯수를 지정합니다. 샘플에서는 [32]로 되어 있으므로 32 개의 매크로를 정의할 수 있습니다. 매크로에 대해서는 차후에 소개하겠습니다.

○ 44 행: MOUSE=ON

마우스를 사용할 것인지 정의하고 있습니다. 만약 마우스를 사용해서 오류가 발생될 경우는

SoftICE 설치 및 환경 설정

[MOUSE=OFF]로 마우스를 사용하지 않도록 합니다.

○ 51 행 ~ 61 행

마우스를 사용했을 때 오른쪽 클릭 메뉴를 정의하고 있습니다. 오른쪽 클릭 메뉴는 사용할 일이 별로 없기 때문에 기본적으로 변경할 필요는 없습니다.

○ 68 행 ~ 76 행

DLL 등으로 정의되어 있는 API에 브레이크 포인트를 설정하려면 SoftICE에 관련 DLL을 로드할 필요가 있습니다. 이 행에서 로드할 DLL을 지정합니다.

예를 들어 GetWindowText는 KERNEL32.DLL에 정의되어 있기 때문에 KERNEL32.DLL을 로드하지 않으면 GetWindowText 함수에 브레이크 포인트를 설정할 수 없습니다. DLL 로드는 필요할 때 심볼 로더에서 간단히 로드할 수 있지만, 사용빈도가 높은 API가 정의되어 있는 DLL을 매번 수동으로 로드하는 것은 시간 낭비이기 때문에 자동적으로 로드되도록 해야 합니다. API가 어느 DLL을 사용하고 있는지 확인하려면 Dependency Walker⁶³와 같은 Dependency checker나 exeScope와 같은 Resource Viewer를 이용하면 됩니다.

WindowsNT 계열을 사용하고 있는 분은 이것으로 기본적인 설정을 완료할 수 있습니다.

그러나 Windows 9x 계열에서 SoftICE를 구동하려면 몇 가지 설정을 추가적으로 해야 합니다. 시스템 부팅에 중요한 config.sys와 Autoexec.bat를 변경하기 때문에 변경 전에 백업해 두십시오.

먼저 config.sys를 아래와 같이 편집하십시오.

행번호	Config.sys 내용
1	[menu]
2	menuitem SOFTICE,Load SoftICE Debugger Behind Windows
3	menuitem NORM,Normal Mode
4	menudefault NORM,9
5	
6	[SOFTICE]
7	SoftICE 동작에 필요한 드라이버
8	
9	[NORM]
10	윈도우 기본 부팅시에 필요한 드라이버(SoftICE 제외)
11	

⁶³ Dependency Walker: <http://www.dependencywalker.com/>

SoftICE 설치 및 환경 설정

12	[common]
13	공통 드라이버

설명
<p>○ 1 행 ~ 4 행</p> <p>부팅시 SoftICE 를 실행할지 아니면 SoftICE 를 실행시키지 않을지 선택할 수 있도록 메뉴를 만들고 있습니다. 이 샘플에서는 4 행에서 [menudefault NORM]으로 정의하고 있으므로 기본적으로 SoftICE 가 동작하지 않은 상태에서 Windows 가 부팅하도록 되어 있습니다. 처음부터 SoftICE 를 동작 시키려면 [menudefault SOFTICE]로 하면 됩니다.</p>
<p>○ 6 행 ~ 8 행</p> <p>여기에서 SoftICE 가 동작할 때 필요한 드라이버를 기술합니다. 이번에 처음으로 설치한 분에게는 어느것이 필요한 것인지 모를 것이므로 인스톨러가 추가한 것을 이 [SOFTICE] 부분에 전부 넣습니다.</p>
<p>○ 9 행 ~ 11 행</p> <p>SoftICE 를 동작 시키지 않고 일반적인 윈도우로 부팅할 때 필요한 드라이버를 기술합니다. MS-DOS 환경이라면 이 행에 여러가지 적게 되지만, Windows 9x 라면 아무것도 적지 않아도 문제 없습니다.</p>
<p>○ 12 행째~13 행째</p> <p>SoftICE 동작시나 일반적인 운영체제 부팅시 둘다 필요하게 되는 드라이버를 기술합니다.</p>

다음으로 Autoexec.bat 을 편집합니다. 물론 편집 작업전에 백업해 두십시오. 이 파일에 대해서는 특별히 설명 할 것은 없습니다.

행번호	Autoexec.bat 내용
1	goto %config%
2	
3	:SOFTICE
4	C:\NUMEGA\SOFTICE\WINICE.EXE
5	
6	goto common
7	:NORM
8	
9	:goto common
10	:common

SoftICE 설치 및 환경 설정

설명

○ 4 행: C:\NUMEGA\SOFTICE\WINICE.EXE

이 행은 WINICE.EXE(SoftICE)의 위치를 지정하고 있지만, MS-DOS 의 8.3 형식으로 기술해야 합니다. SoftICE 를 설치한 올바른 위치로 변경해 주십시오. config.sys 와 autoexec.bat 를 편집한 뒤에 재 부팅하면 부팅시에 메뉴가 표시되어 SoftICE 동작 유무를 선택할 수 있습니다.

SoftICE 매크로

SoftICE 를 사용하고 있으면 자주 사용하는 명령어(예, bpx GetDlgItemTextA 나 bpx GetWindowTextA 등)를 치는 것이 매우 번거로울 수 있습니다. 이렇게 자주 사용하는 명령을 단축 명령어로 등록할 수 있습니다. 아래와 같이 Winice.dat 에 정의하면 됩니다..

```
MACRO GETWTA="bpx GetWindowTextA"
```

이것으로 GETWTA 라고 치는 것 만으로도 GetWindowTextA 에 브레이크 포인트를 설정할 수 있습니다. 입력한 문자열을 획득하는 API를 실행한 후 어떤 문자열을 획득했는지 확인하는 것이 일반적이라고 생각합니다.

매크로에 의해 간단히 브레이크 포인트를 설정할 수 있도록 되었지만, 브레이크한 후에 획득된 문자열을 확인하는 것도 번거롭기 때문에, 이것도 매크로를 사용하여 브레이크 한 후에 자동적으로 획득한 문자열이 표시되도록 해보겠습니다.

bpx, bpm, bpr 등의 브레이크 포인트를 설정하는 명령에는 [DO] 옵션이 있어 [DO] 옵션에 기술하면 브레이크한 직후에 실행합니다. 이것을 매크로와 조합하여 사용합니다. 아래 내용을 Winice.dat 에 정의 하면 됩니다. DO 옵션에 두개 이상의 명령을 쓸 경우, 미리 매크로로 등록하여 호출할 필요가 있기 때문에 아래와 같은 형태로 정의하여야 합니다.

```
MACRO ESP8G="db @(esp+8);G @SS:ESP;? EAX"  
MACRO GETWTA="bpx GetWindowTextA do W"ESP8GW""
```

간단히 설명하면, [ESP8G]로 세개의 명령을 정의하고 있습니다. [db @(esp+8)]로 데이터 원도우에 바이트 형식으로WindowTextA 의 텍스트용 버퍼 어드레스를 표시합니다. [G @SS:ESP]에서 평션 키 F11 을 눌러 API 에서 ret 하여 [? EAX]에서 취득한 문자열의 문자수를 표시하고 있습니다. [GETWTA]에서는 GetWindowTextA 에 브레이크 포인트를 설정하여 브레이크되면 매크로 [ESP8G]를 실행합니다.

또 다른 매크로를 보도록 하겠습니다.

```
MACRO vb5s="s 0 1 bfffffff 56 57 8b 7c 24 10 8b 74 24 0c 8b 4c 24 14 33 c0 f3 66 a7"
```

SoftICE 설치 및 환경 설정

```
MACRO vb6s="s 0 1 bfffffff 8B C2 D1 E8 89 45 10 8B 7D 0C 8B 7D 0C 8B 75 08 8B 4D 10 33 C0 F3 66 A7"
```

이 매크로는 매우 유명한 것으로 VB theory 라고 불리고 있습니다. VB theory 에서는 [s] 커맨드를 사용하여 특정 바이트열을 검색하지만, 검색 바이트수가 많기 때문에 외우기가 곤란합니다. 이같은 VB theory 와 같이 외울 수 없거나, 타입하는 문자수가 많을 때는 매크로가 반드시 필요할 것입니다.

SoftICE 기능 추가

SoftICE 체크 회피

SoftICE 는 매우 유명한 디버거이기 때문에 SoftICE 가 실행되고 있으면 시스템이 디버깅중이라는 알림창이 표시되거나, 종료하거나, 때로는 아무런 경고 표시도 하지 않고 전혀 실행이 안되는, 이를바 디버거 체크 루틴을 갖춘 소프트웨어가 늘어났습니다. 이같은 SoftICE 검출 기능을 갖춘 소프트웨어를 크랙하려면 먼저, SoftICE 체크 루틴을 우회 할 필요가 있습니다. 시리얼 체크 루틴이나 특정 기능을 제거 하기 위해 디버거를 이용하려면 반드시 디버거 체크 루틴을 회피할 수 있어야 합니다.

수동으로 체크루틴을 우회 할 수 있지만, 이것은 크랙 및 리버싱할 소프트웨어마다 작업이 발생하기 때문에 매우 귀찮습니다. 현재 SoftICE 체크를 회피하는 툴이 몇 개 공개되어 있으므로 그것을 사용하면 됩니다.

○ FrogsICE(<http://protools.cjb.net/>)IceGhost

Frog's Print+씨가 작성한 가장 유명한 SoftICE 체크회피 툴입니다. 첫번째 특징은 동적으로 SoftICE 를 숨길 수 있는 매우 우수한 툴입니다. 그러나 2001년 이후로 업데이트가 되지 않아 최신의 버전 SoftICE에는 동작하지 않고 Windows 95, 98, Me 에서만 사용할 수 있습니다.

사용법은 앱축 파일을 해제하면 서브 폴더가 3 개 생깁니다. 그 중에서 자신의 운영체제 환경에 맞는 [frogsice.vxd]를 [Fuploaded.exe]와 같은 폴더에 저장하고 [Fuploader.exe]를 실행 시킵니다. 실행되면 트레이에 상주하기 때문에 작업표시줄의 트레이 아이콘을 오른쪽 클릭하여 메뉴 화면을 열어, Enable/Disable 하는 기능이나 옵션을 선택할 수 있습니다. 옵션을 변경할 경우 반드시 FrogsICE 가 Disable 되어 있는 것을 확인해 주십시오. FrogsICE 가 disable 되어 있는 경우는 메뉴 화면의 [Activation]에서 [Disable FrogsICE] 버튼이 회색으로 되어 있습니다.

또한 FrogsICE 가 정지 되어 있는 경우 작업 표시줄의 FrogsICE 아이콘에 빨간 엑스마크가 붙어 있습니다. 젤 처음에는 어떤 기능을 Enable 하면 좋을지 모르기 때문에 메뉴 화면의 [Quick options]에서 [Protection]에 대상이 되는 보호 방법을 선택하십시오. 어떤 기능을 유효로 할 것인지 설정이 끝나면 메뉴 화면의 [Activation]에서 [Enable FrogsICE] 버튼을 클릭합니다. 이것으로 SoftICE 을 숨길 수 있습니다.

○ NT ALL(<http://protools.cjb.net/>)

SoftICE 설치 및 환경 설정

FrogsICE 의 WindowsNT 판으로 많은 사이트에 FrogsICE 의 클론이라고 소개되어 있지만 FrogsICE 와 비교하여 기능이 매우 적습니다. NT ALL 도 FrogsICE 와 같이 동적으로 SoftICE 를 숨길 수는 있지만, 오랜 기간 업데이트되지 않았기 때문에 최신 SoftICE 에서는 동작하지 않습니다.

○ IceExt⁶⁴

SoftICE 확장 플러그인으로 현재까지도 업데이트가 이루어지고 있습니다. 특정 메모리 공간의 브레이크 포인트 설정이나 메모리 덤프 기능, 스크린덤프 기능, 파일을 메모리로 로딩할 수 있는 기능등 유용한 많은 기능들이 있습니다. 소스도 공개되어 있기 때문에 자신만의 기능을 추가할 수 있습니다.

※ 주요 명령어

- !BPR – break point on memory range;
- !CP – set current codepage (866 or 1251);
- !DUMP – dump memory to disk;
- !DUMPSCREEN – dump SoftICE® screen to disk in RAW format;
- !EB – patch byte in memory (restores PTE after patching);
- !FONT – set screen font;
- !IF – conditional commands execution (useful for macros');
- !LASTBRANCH – display MSR LastBranch info;
- !LOADFILE – load disk file to memory;
- !PROTECT – turn SoftICE® antidection on/off;
- !HELP – display help;
- !SUSPEND – suspend current thread execution (only for ring-3 threads);
- !RESUME – resume thread that was suspended before;
- !TETRIS – tetris game.
- and so on...

○ IceDump⁶⁵

메모리, 프로세스를 덤프하거나 로드/언로드할 때 사용됩니다. Sdc(ScreenDump Converte)툴을 이용하면 화면을 텍스트파일, 그림파일, html 파일 등으로 변환할 수 있는 기능도 지원합니다. 또한 자체적으로 디버깅 회피 기능과 Anti Winice 탐지 기능을 갖추고 있습니다.

○ IceX⁶⁶

⁶⁴ iceExt: <http://stenri.pisem.net/>, <http://sourceforge.net/projects/iceext/>

⁶⁵ IceDump: <http://programmerstools.org/node/94>

⁶⁶ IceX: <http://programmerstools.org/node/89>

SoftICE 설치 및 환경 설정

SoftICE에서 사용되는 C 형태와 비슷한 스크립트 언어입니다. IO 제어, 메모리 접근, 키보드 제어, 매크로 작성등 많은 부가적인 기능을 수행할 수 있는 스크립트를 작성 할 수 있습니다.

메모리 덤프

최근 많은 소프트웨어에서 사용되고 있는 패커(Packer)는 실행 파일 자체가 압축, 암호화되어있기 때문에 원래 바이너리로 복원하려면 한번 실행시킨 후 메모리에 로딩된 정상적인 상태의 코드를 덤프할 필요가 있습니다.

SoftICE 에도 기본적으로 덤프하는 기능이 있지만, 사용하기 어렵기 때문에 IceDump(<http://protools.cjb.net/>)을 사용한 덤프 기능을 주로 이용합니다. Windows9x 와 WindowsNT 계열에서 설치 방법이 다릅니다.

○ WindowsNT 의 경우

압축 파일을 풀면 [wnt] 폴더가 생깁니다. 서브 폴더에 SoftICE 버전 별로 폴더가 생기기 때문에, 사용하고 있는 SoftICE 와 같은 버전의 폴더에 있는 [icedump]와 [nitzd.exe]를 [%SystemRoot%\system32\drivers]에 복사하여 [nticedump.bat]을 Command 창에서 실행하면 설치가 완료됩니다. IceDump 의 기능은 운영체제를 재부팅 하기 전까지 유효합니다. 한번 실행하면 중지 할 수 없다는 점을 주의해 주십시오. 또한 NASM 이 미리 설치되어야 합니다.

○ Windows9x 계열의 경우

압축 파일을 풀면 [w9x] 폴더가 생성됩니다. 사용하고 있는 SoftICE 와 같은 버전의 [icedump.exe]를 DOS 프롬프트에서 실행하면 설치가 완료됩니다. WindowsNT 계열과 달리 IceDump 를 사용하지 않으려면 [icedump u]를 DOS 프롬프트에서 실행해 주면 됩니다.

그외 내장된 몇몇 플러그인 중에서 화면 덤프 컨버터(sdc.exe)를 이용하면 덤프된 파일을 텍스트나 그림파일 등으로 변경할 수 있습니다.

마치며

이번에 소개한 SoftICE 플러그인이나 툴은 공개된 것의 일부에 지나지 않습니다. 인터넷에서 SoftICE 관련 툴이 많이 공개되어있기 때문에 Google(<http://www.google.com/>)을 통한 검색이나 리버싱 포럼등에서 [softice]를 키워드로 검색해 보십시오. 보다 좋은 플러그인과 툴들을 발견할 수 있을 것입니다.

※ 추가적인 SoftICE 플러그인은 아래 사이트를 참고하시길 바랍니다.

Softice extention: <http://programmerstools.org/taxonomy/term/37/>

제 4 장 Windows 크랙 응용편

4.1 P-Code 크랙편

들어가며

Visual Basic6.0 의 Native code⁶⁷ 실행 파일 분석은 OllyDbg 등의 디버거를 이용할 수 있지만, P-Code⁶⁸로 출력된 실행 파일의 분석은 WKTVBDE, ExDec, P32Dasm, VBDecom 등을 사용하여야 한다.

P-Code 의 Trace 는 Native code 와는 달리 스택에 PUSH/POP 이 기술되지 않고 수행되기 때문에, 스택의 상태를 주의 깊게 관찰하는 것이 매우 중요하다. 스택 주소는 동작환경에 따라 변화하므로 다시 트레이스(trace)한 경우도 같은 주소가 사용된다는 보증도 없어서 매우 번거롭고 어렵다. 그렇지만 점차 익숙해 지면 어렵지 않다.

아래 표는 Visual Basic 프로그램을 Crack 하거나 Reversing 하기 위한 대표적인 툴 목록이다.

VB 버전	프로그램명	설명	URL
VB 3/4	VBDIS	P-Code Disassembler	Expired
VB 5/6	ExDec	P-Code Disassembler	http://t4c.fbi.cz/dec/exdec.zip http://www.pediy.com/tools/Decompilers/VB_pcode/EXDEC/exdec818.zip
	P32Dasm	P-Code Decompiler	http://t4c.ic.cz/forum/attachment.php?attachmentid=363 http://programmerstools.org/node/127
	wkt-vbdebugger	P-Code Debugger (Loader v4.2)	http://vbdebug.cjb.net/ [Linked out] http://programmerstools.org/node/81 http://t4c.fbi.cz/dec/wktdebug.exe
	VBReformer	Native Decompiler	http://www.decompiler-vb.net/ http://t4c.fbi.cz/dec/vbreformer.zip
	SemiVBDecompiler	P-code/Native Decompiler	http://t4c.fbi.cz/dec/semi.zip http://www.semivbdecompiler.com/
	VBDE	Native Decompiler	http://t4c.fbi.cz/dec/vbde.zip http://programmerstools.org/node/129

⁶⁷ Native Code: CPU 가 직접 이해할 수 있는 코드.

⁶⁸ P-Code: Pseudo Code (의사 코드)의 약어. 의사코드는 CPU 가 직접 이해할 수 없는 중간언어.

Visual Basic P-Code 크랙 2

	VB RezQ	P-code/Native Decompiler	http://www.vbrezq.com/ http://t4c.fbi.cz/dec/vbrq.zip
	RaceeX		http://t4c.fbi.cz/dec/race.zip
	VBEeditor II		http://t4c.fbi.cz/dec/vbeditor.zip

표 1. Visual Basic Crack/Reserving tools

소프트웨어 크랙을 시작하기 전에

본 문서의 CrackMe 는 Visual Basic 6.0(SP6)의 P-Code 모드로 컴파일 하였다. 만약에 동등한 개발환경이 아니라면 Visual Basic 6.0(SP6)의 런타임 라이브러리[<http://support.microsoft.com/kb/290887>]를 다운받아 설치하여야 한다. 또한 예제로 설명한 CrackMe 의 모든 스택주소 값은 P-Code 의 특성상 모두 달라 질 수 밖에 없다. 그러므로 본인의 컴퓨터에서도 매번 실행할 때마다 달라지므로 분석시 주의하길 바란다.

WTKVBDE 설치

WTKVBDE 란, Visual Basic 의 P-Code 모드에서 컴파일된 실행 프로그램을 분석하기 위한 디버거이다(그림 1). Visual Basic 5.0 이후부터는 네이티브 코드(Native Code)도 지원하기 때문에 Visual Basic 프로그램 전부가 P-Code 라고 할 수는 없다. 현재 WTKVBDE 공식 배포 웹사이트는 폐쇄되었으므로 표 1 의 다운로드 사이트나 Google 검색을 통해 다운받으면 된다.

→Visual Basic P-Code Debugger (<http://vbdebug.cjb.net/>, <http://kickme.to/wkt>)



그림 1. WTKVBDE Loader

Visual Basic P-Code 크랙 2

다운 받은 [vbdebug14e.exe]를 더블클릭 한 후 설치관리자가 진행되면 아래 그림과 같은 에러 메시지가 표시된다(그림 2).

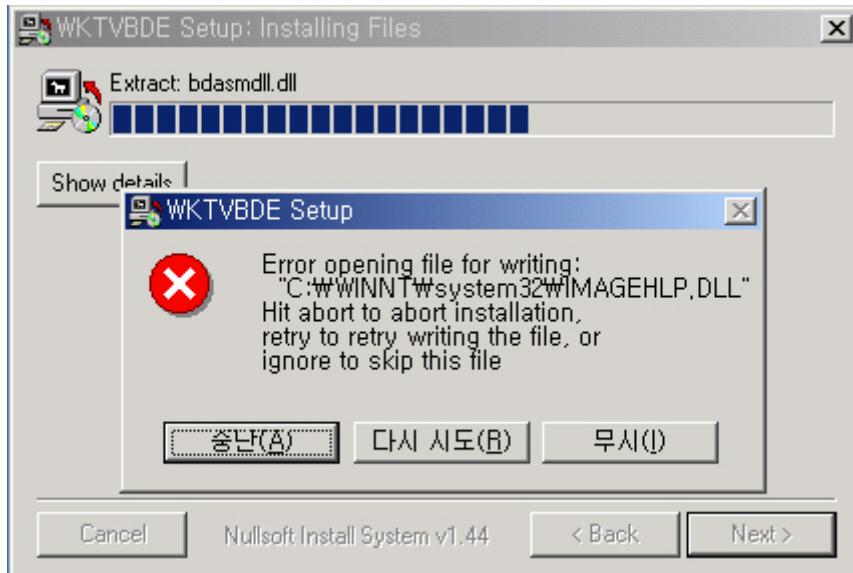


그림 2. 에러 메시지 화면

이런 경우 [Ignore]를 선택하여 인스톨을 계속 진행한다. 또한 WTKVBDE로 VB 프로그램을 실행했을 때에 [Can't load WKTVBDE.DLL…]와 같은 메시지가 표시되면(그림 3), WKTVBDE를 설치한 폴더 아래에 있는 [WKTVBDE.DLL]을 Windows 시스템 폴더에 복사하거나, WKTVBDE.EXE 단축 아이콘(바로 가기)을 생성하여 시작 위치에 WKTVBDE가 설치되어 있는 폴더를 지정한다.



그림 3. DLL 로드 에러 메시지 화면

P-Code 디컴파일러

WKTVBDE로 분석된 P-Code는 한정된 부분만을 표시하여 주기 때문에 전체적인 프로그램 로직을 살펴보는 것은 불가능하다. 이런 단점을 채우기 위해서 디컴파일러를 사용하여야 한다.

본 문서에서 사용하는 P-Code 디컴파일러는 다수 공개되어 있는 툴 중에서 P32Dasm, Exdec 툴을 선택하였다(그림 4).

Visual Basic P-Code 크랙 2

이 P32Dasm는 GUI 조작으로 P-Code 실행 파일을 디컴파일 할 수 있다. 또한 디컴파일된 결과를 RTF⁶⁹ 포맷으로 저장 할 수 있다. 그러나 배포 사이트가 폐쇄되었기 때문에, 다운로드가 불가능하므로 표 1 을 참조하여 다운로드 후 설치하길 바란다. 또한 파일 오프셋으로 주소를 표시하기 때문에 메뉴의 RVA 툴을 이용하여 다시 계산해 주어야 한다.

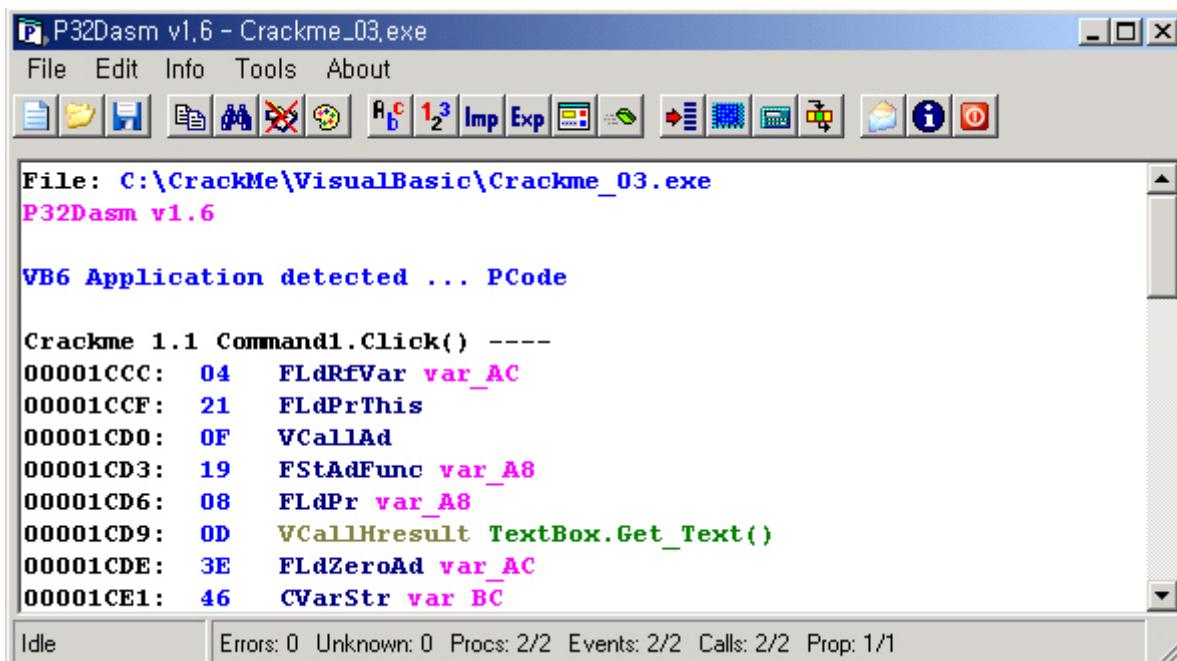


그림 4. P32Dasm(P-Code 디컴파일러)

⁶⁹ 워드패드 파일로 하이라이트 기능과 상세 주석처리가 되어 있음. 줄 바꿈 코드는 [LF] 코드뿐이다.

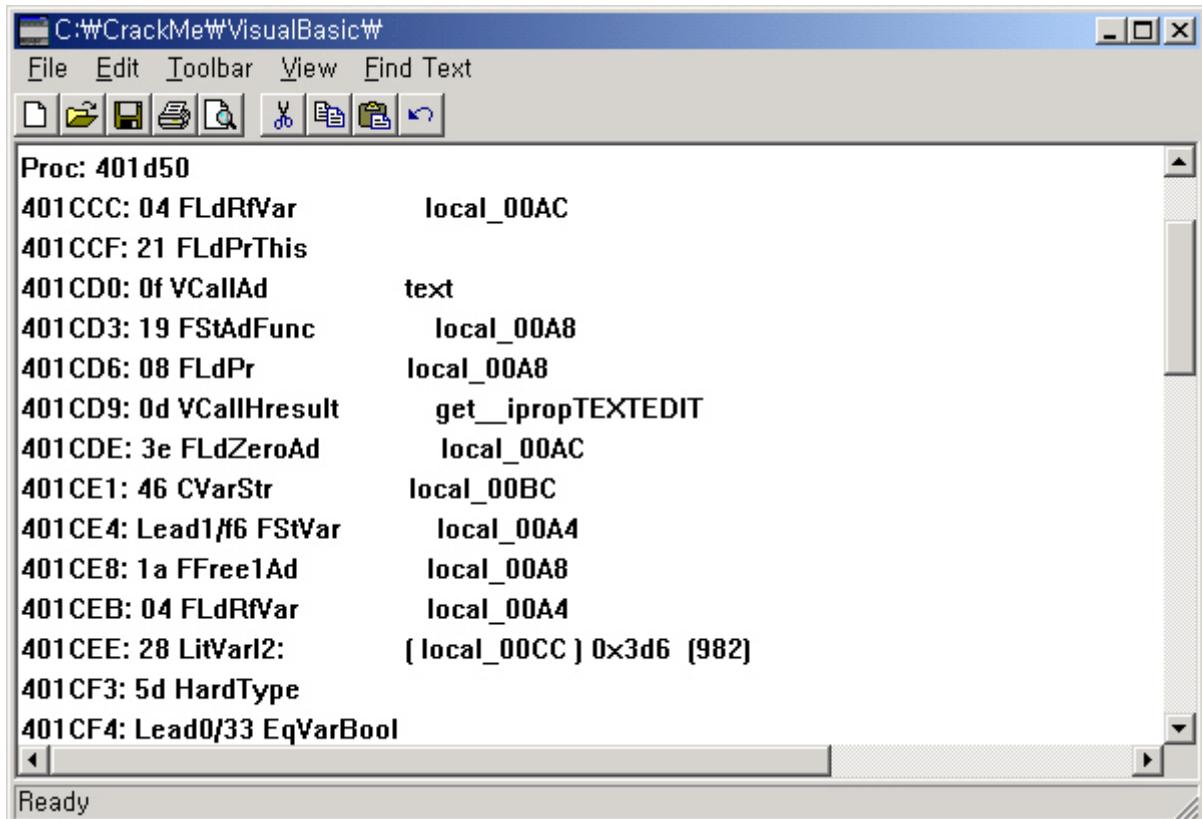


그림 5. Exdec(P-Code 디컴파일러)

WTKVBDE 사용방법

WTKVBDE 디버거를 실행한 후 P-Code 형식의 파일을 읽어 들어 메뉴의 (Action->Run)을 수행하면 WTKVBDE 윈도우가 표시된다. 이 윈도우가 메인 화면이다(그림 6).

Visual Basic P-Code 크래프트

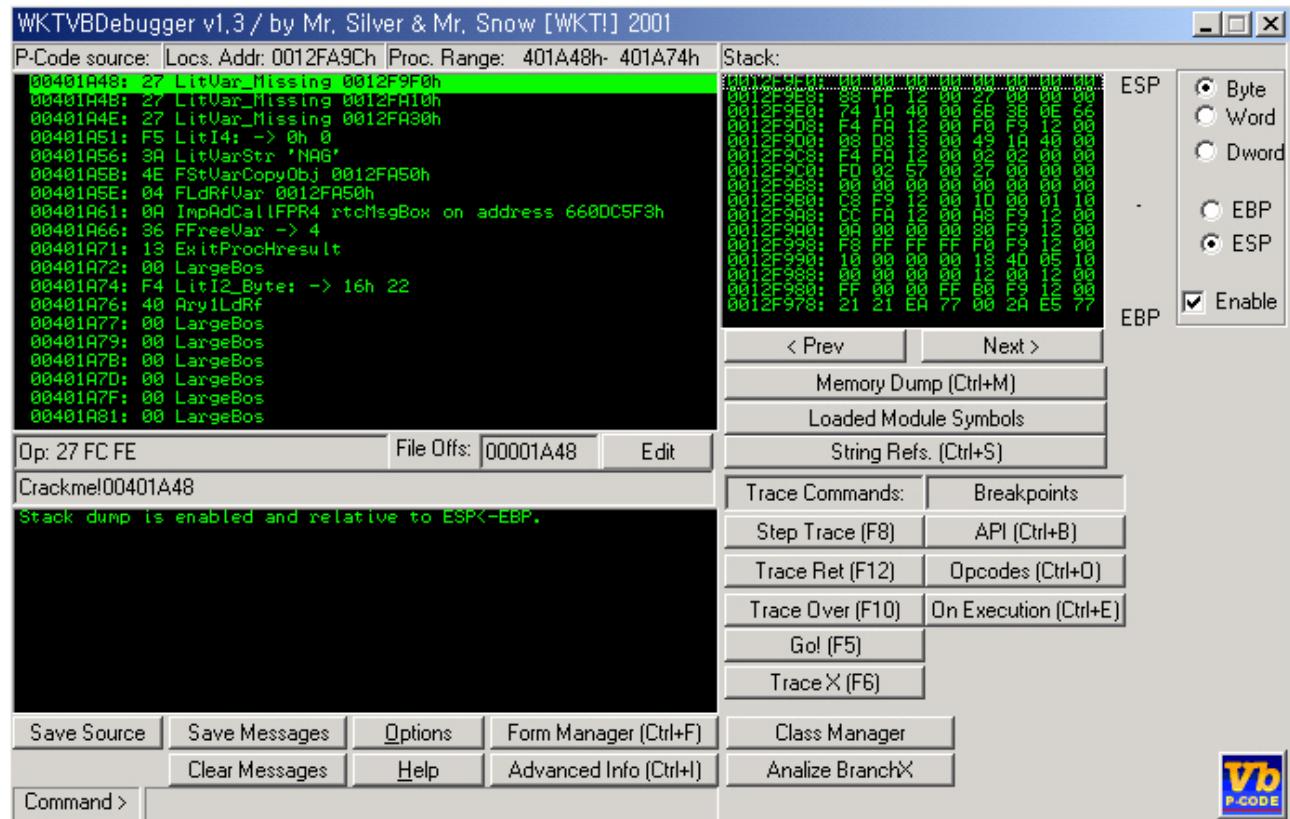


그림 6. WTKVBDE 메인 윈도우

P-Code 리스트 표시 부분

액 캄파일된 P-Code 가 표시되는 부분이다. 반전 표시되어 있는 행이 지금부터 실행되는 P-Code이며(①), 스택 어드레스(②)에는 프로그램 실행에 중요한 정보가 표시된다. 또한 브레이크 포인트 설정도 가능하다(그림 7).

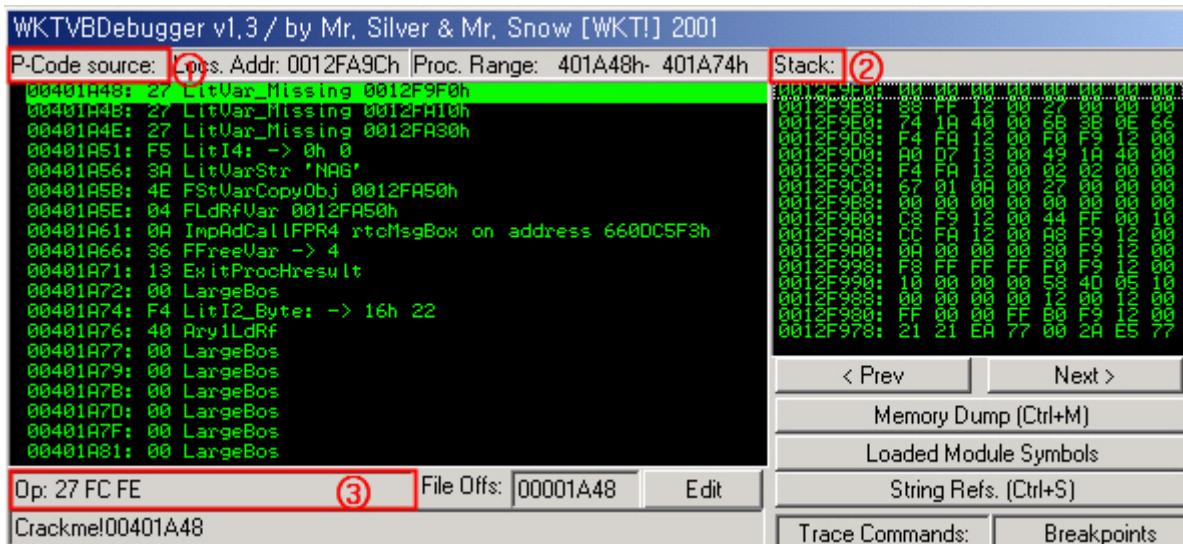


그림 7. P-Code 리스트 표시 부분

Visual Basic P-Code 크랙 2

[Op: 27 FC FE]라고 표시되어 있는 부분(③)은 현재의 연산 코드이며, [File Offs:]부분에는 파일의 Offset 값이 표시된다. [Edit] 버튼을 클릭하면 [Memory viewer and editor] 윈도우가 표시되어 연산 코드의 편집이 가능하다.

P-Code 는 어셈블러와는 달리 레지스터를 사용하지 않는다. 연산, 비교, 서브루틴 콜 등은 모두 스택을 경유하여 진행된다. 따라서 트레이스 할 경우, 스택 화면은 매우 중요한 정보 표시부분이 될 것이다(②). [<Prev] [>Next] 버튼으로 스택 정보를 페이지 단위로 이동 할 수 있고, [Byte][Word][Dword]의 라디오버튼을 변경하면 표시형식이 1 바이트, 2 바이트, 4 바이트 단위로 변경된다.

메시지 정보 부분

브레이크 포인트에서 정지한 메시지나 VB 의 함수 결과 등 보조적인 정보가 표시된다(그림 8).

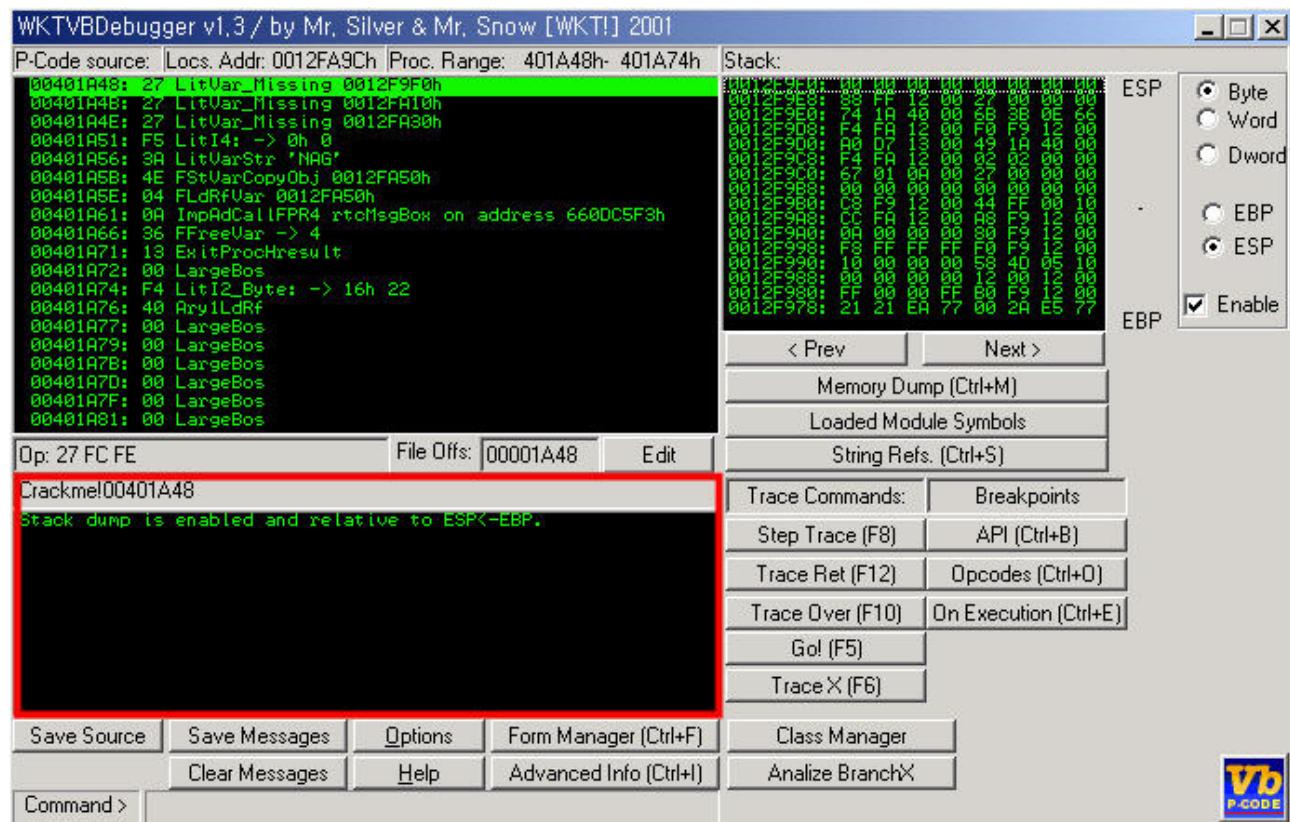


그림 8. 메시지 표시 부분

각종 버튼 그림은 아래와 같다(그림 9).

Visual Basic P-Code 크랙 2

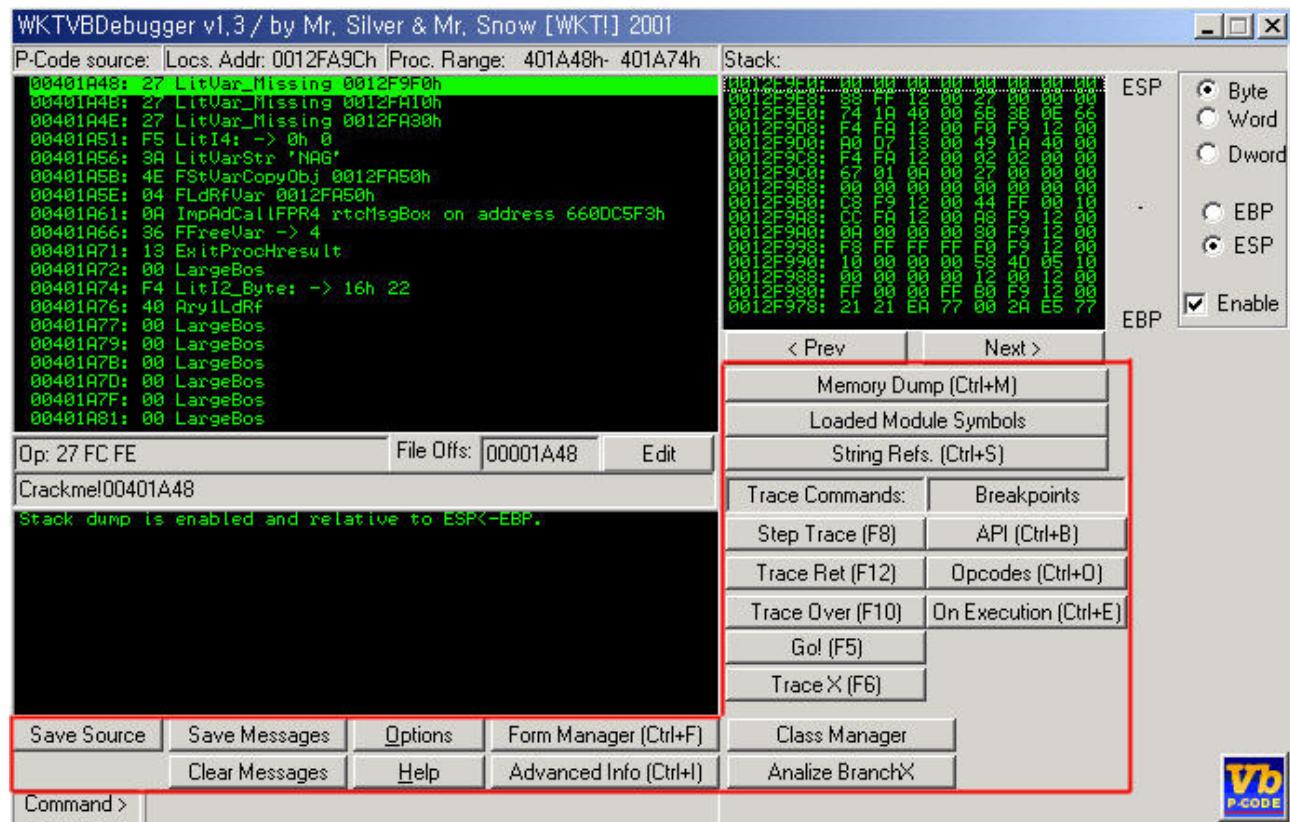


그림 9. WTKVBDE 버튼들

■ 트레이스 명령 부분 버튼

○ [Step Trace(F8)]

한 스텝씩 실행한다. 콜 명령시에는 콜 내부로 이동한다.

○ [Trace Ret(F12)]

리턴 명령을 만날 때 까지 실행한다.

○ [Trace Over(F10)]

한 스텝씩 실행한다. 콜 명령을 하나의 명령으로 실행한다.

○ [Go!(F5)]

실행(Run) 명령이다.

○ [Trace X(F6)]

X 행을 실행 한다. 파일로그에 실행할 행수를 입력한다(그림 10).

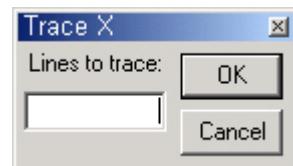


그림 10. TraceX 파일로그

■ 브레이크 포인트 부분

- [API(Ctrl+B)]

VM DLL에서 Export된 함수가 표시되어 있는 디아일로그를 사용하여, 함수 콜 부분에 브레이크 포인트를 설정할 수 있다(그림 11).

Exported VM Functions		
BP	Ord	Name
66006B5C	0137	BASIC_CLASS_AddRef
660C9FD3	0139	BASIC_CLASS_GetIDsOfNames
660CA06F	013A	BASIC_CLASS_Invoke
66006B90	0136	BASIC_CLASS_QueryInterface
6600E713	0138	BASIC_CLASS_Release
660653E8	0140	BASIC_DISPINTERFACE_GetTICount
660C96F1	0141	BASIC_DISPINTERFACE_GetTypeInfo
660D4CFD	005A	CopyRecord
660EA734	076C	CreateExprSrvObj
660C624E	0035	DLLGetDocumentation
6609FBBE	0036	DllCanUnloadNow
6600A019	0057	DllFunctionCall
6609FB40	0037	DllGetClassObject
660C5F63	0006	DllRegisterServer
660C6141	0007	DllUnregisterServer
660C9ABA	0154	EVENT_SINK2_AddRef

그림 11. Export 함수 목록

화면 줄 앞의 램프와 같은 것이 브레이크 포인트 상태를 나타낸다. 회색이 브레이크 포인트 미 설정, 녹색이 브레이크 포인트 설정(Enable), 노란색이면 브레이크 포인트 중지(Disable)를 나타낸다. [Save To File] 버튼을 클릭하면 함수 목록을 텍스트 파일로 저장할 수 있다.

- [Opcodes(Ctrl+0)]

VM의 Opcode 전부가 표시되어 있는 디아일로그를 사용하여, Opcode가 사용되어 있는 부분에 브레이크 포인트를 설정할 수 있다(그림 12).

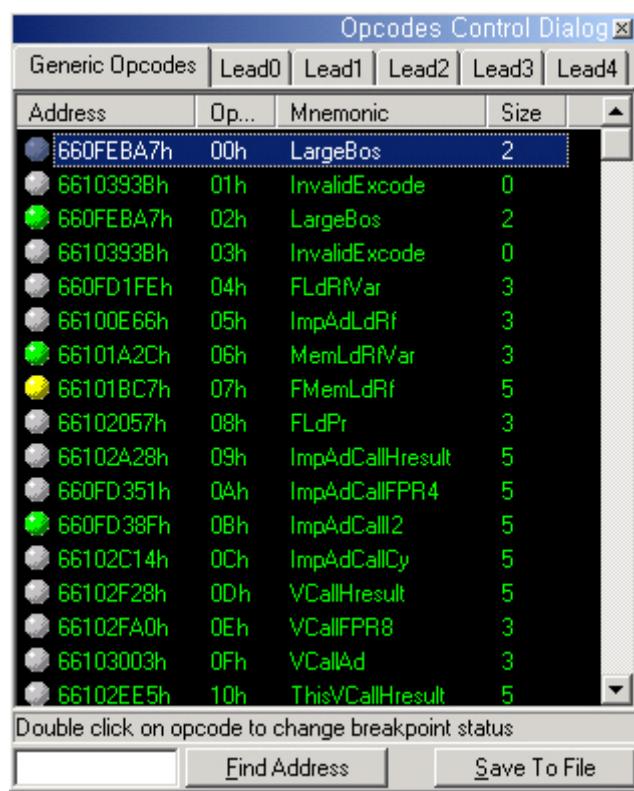
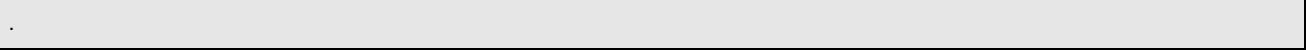


그림 12. Opcode 목록

화면 줄 앞의 램프와 같은 것이 브레이크 포인트의 상태를 나타낸다. 회색이 브레이크 포인트 미 설정, 녹색이 브레이크 포인트 설정(Enable), 노란색이면 브레이크 포인트 중지(Disable)를 나타낸다. [Save To File] 버튼을 클릭하면 선택중인 탭 내에 표시되어 있는 Opcode 전부를 텍스트 파일로 저장할 수 있다. 또한 이 디아일로그에는 연산 코드([Opcode])와 명령사이즈([Size])도 표시된다.

위의 탭 [Lead0] [Lead1] [Lead2] [Lead3] [Lead4]는, [General Opcodes] 탭의 FBh(Lead0), FCh(Lead1), FDh(Lead2), FEh(Lead3), FFh(Lead4) 코드를 사용할 때, 아래와 같은 연산자 Lead 숫자에 대응된 탭 명령 해석으로 변환된다.

Lead 사용시 예			
Generic Opcodes 의 [00h]에 각 Lead 명령을 추가할 경우, 아래와 같이 해석되는 명령이 변환된다.			
	수식		해석되는 명령
GenericOpcodes	없음		LargeBos
Lead0	FBh		InvalidExcode
Lead1	FCh		CStrR8
Lead2	FDh		FstVarCopy
Lead3	FEh		ThisVCallUI1
Lead4	FFh		FnStrComp3Var



○ [On Execution(Ctrl+E)]

임의 어드레스에 브레이크 포인트를 설정하거나, 설정된 브레이크 포인트를 관리한다(그림 13).



그림 13. BPX 디아일로그

■ 기타 버튼

○ [Save Source]

P-Code 리스트 화면에 표시되어 있는 P-Code 리스트를 텍스트 파일로 저장한다.

○ [Save Messages]

메시지 정보부분에 표시하고 있는 메시지를 텍스트 파일로 저장한다.

○ [Clear Messages]

메시지 정보부분의 메시지를 클리어 한다.

○ [Form Manager(Ctrl+F)]

폼과 폼에서 사용하고 있는 컨트롤을 표시하는 디아일로그이다(그림 14).



그림 14. Form Managerダイアログ

또한 사용된 컨트롤 버튼을 클릭하면, 그 컨트롤 정보를 표시하며 [BPX] 버튼을 클릭하면, 해당 컨트롤 처리시에 브레이크 포인트를 설정할 수 있다.

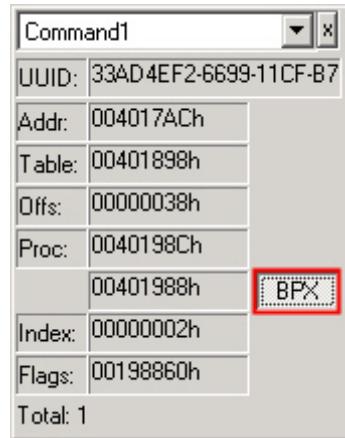


그림 15. 브레이크 포인트 설정

○ [Class Manager]

Form Manager과 같은 기능으로, 폼 대신에 클래스를 관리한다.

○ [Analize BranchX]

Visual Basic P-Code 크랙 2

실행도중 명령 처리부분에서 사용되고 있는 분기계열 명령 목록을 표시한다(그림 16).

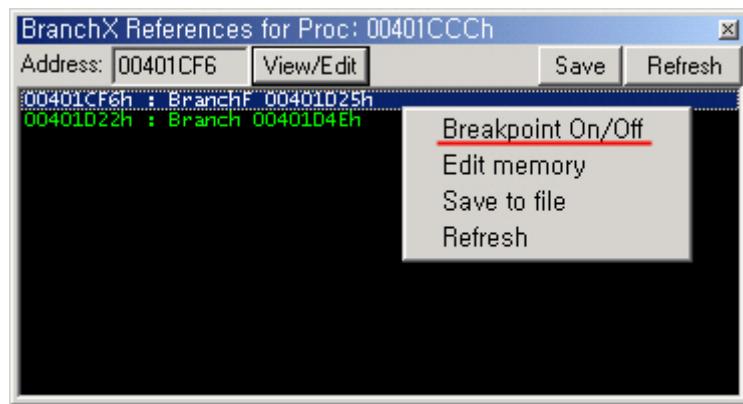


그림 16. Branch 목록 디아일로그

마우스 오른쪽 클릭 메뉴를 사용하면, 점프 명령 부분에 브레이크 포인트를 설정할 수 있다.

○ [Memory Dump(Ctrl+M)]

메모리 내용을 표시하는 화면이다(그림 17). 또한 메모리 편집 기능이 있어서, 변경하고 싶은 부분의 행을 더블 클릭하면 수정할 수 있다.

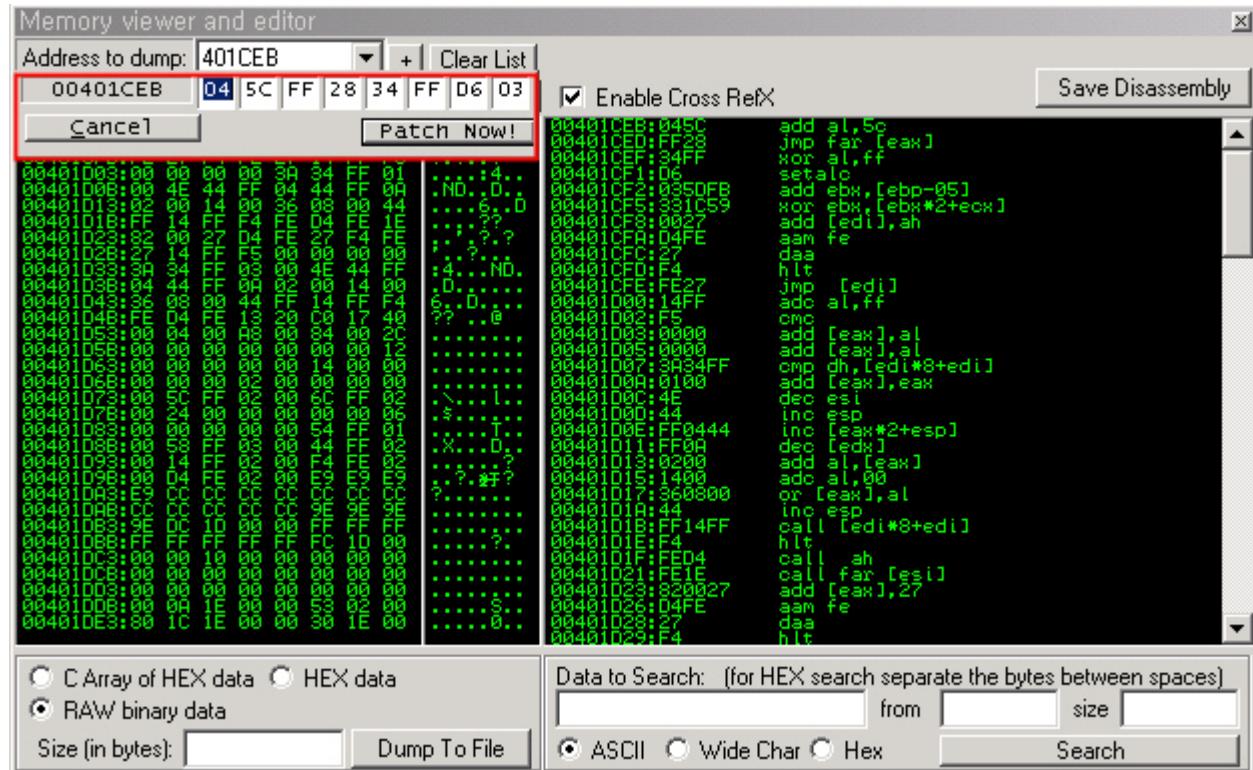


그림 17. 메모리 Viewer 화면

○ [String Refs.(Ctrl+S)]

Visual Basic P-Code 크랙 2

참조하고 있는 문자열을 표시하는ダイ얼로그이다(그림 18). 마우스 오른쪽 메뉴를 사용하여 문자열 변경을 할 수 있다.

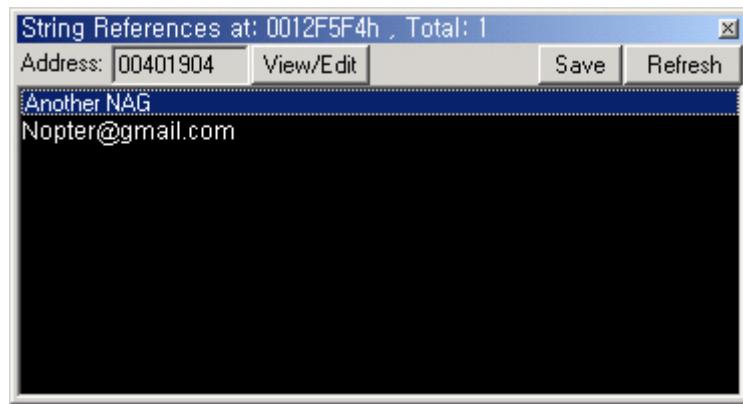


그림 18. 참조 문자열 목록

상세한 사용방법은 아래 Visual Basic Crack Me 를 예제로 사용법을 익혀 나가면 될 것이다.

제 4 장 Windows 크랙 응용편

4.2 P-Code 크랙편(Nag 표시창 없애기)

WTKVBDE 프로그램을 이용하여 다양한 Visual Basic P-Code 프로그램들을 크랙하는 예제를 살펴보도록 하겠습니다. 첫번째로 Shareware 프로그램이나 일부 기능제한 프로그램에서 사용되는 Nag 창을 나타나지 않도록 하는 예제입니다.

Nag 표시창 없애기

첨부파일 [VBCrackMe1.exe]를 실행하면 아래와 같은 메인 윈도우 화면이 보인 후 2 초 후에 Nag⁷⁰ 창이 보여진다.

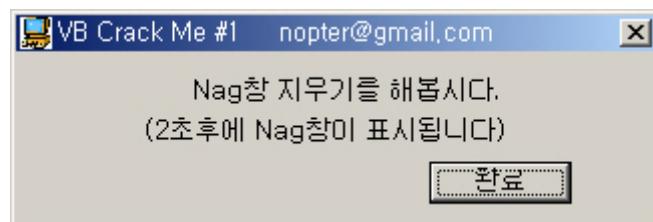


그림 1. VBCrackMe1.exe 실행 화면

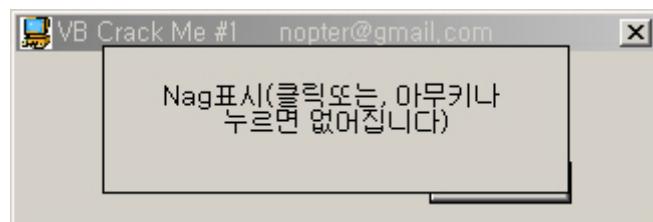


그림 2. Nag 표시 화면

그럼 [WTKVBDE]를 실행한 후 메뉴의 [File]>[Open…]에서 [VBCrackMe1.exe]를 불러 들인다(그림 3).

⁷⁰ Nag: Shareware 프로그램에서 정식 등록, 기능제한을 위해 프로그램 최상단에 알림창이 뜨는 것을 말한다.

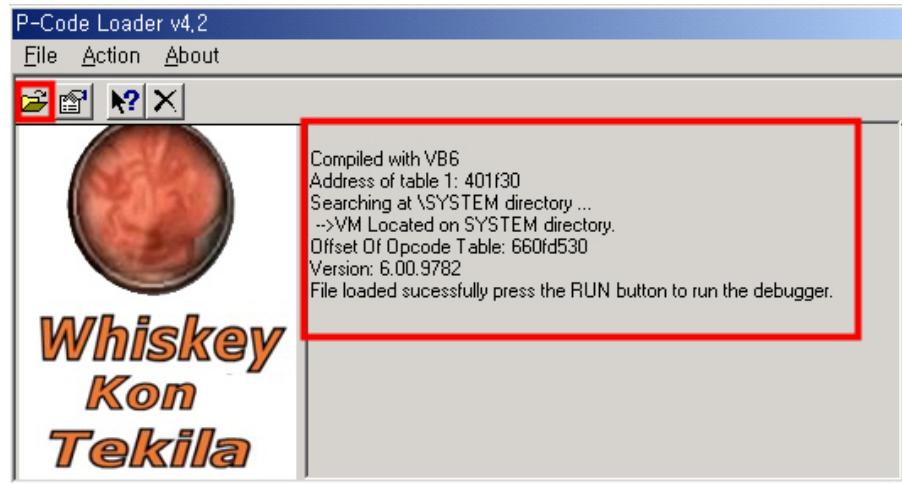


그림 3. VBCrackMe1.exe 불러옴

정상적으로 불러오면, 윈도우 우측 화면에 컴파일한 Visual Basic 버전 및 실행 파일과 관련된 정보가 표시된다. 불러온 파일을 실행시키면([Action]>[Run] 현재 운영체제 버전과 실행 프로그램의 정보를 알려주는ダイ얼로그 윈도우가 차례대로 표시된다(그림 4,, 그림 5).

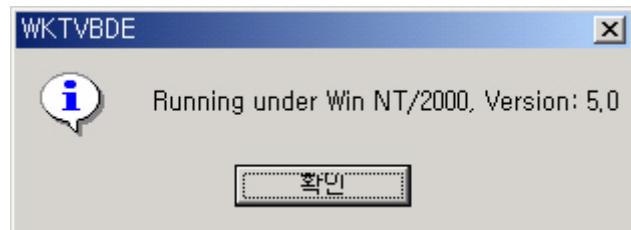


그림 4. 현재 실행하고 있는 운영체제 버전



그림 5. 실행 프로그램 정보

만약 위의ダイ얼로그를 5초안에 확인하지 않으면, 여러 메시지가 표시되거나 WTKVBDE 가 강제 종료해 버린다. 그런 경우 메뉴의 [Action]>[Run]을 선택하여 실행하면 다시 WTKVBDE 윈도우가 표시된다(그림 6).

Visual Basic P-Code 크래프트

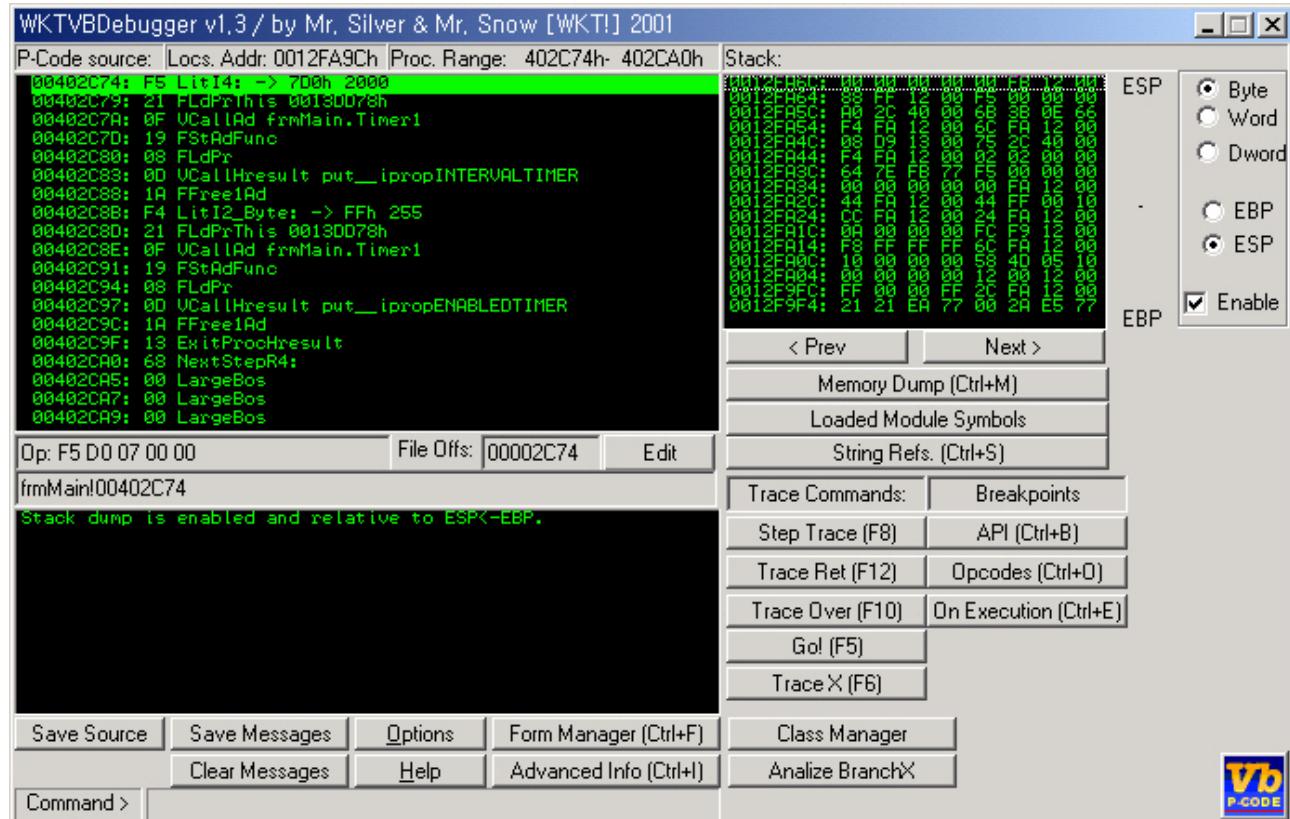


그림 6. WTKVBDE 윈도우

Visual Basic 프로그램의 Form Load 이벤트⁷¹ 시작부분에서 프로그램이 정지되어 있는 상태이다. WTKVBDE의 P-Code 리스트 표시부분은 아래와 같이 표시된다.

```

00402C74: F5 LitI4: -> 7D0h 2000 ; 4byte 값->스택
00402C79: 21 FLdPrThis 0013DD78h
00402C7A: 0F VCallAd frmMain.Timer1
00402C7D: 19 FStAdFunc
00402C80: 08 FLdPr
00402C83: 0D VCallHresult put__ipropINTERVALTIMER ; Interval 값 세트
00402C88: 1A FFree1Ad
00402C8B: F4 LitI2_Byte: -> FFh 255 ; 2Byte 값->스택
00402C8D: 21 FLdPrThis 0013DD78h
00402C8E: 0F VCallAd frmMain.Timer1
00402C91: 19 FStAdFunc
00402C94: 08 FLdPr
00402C97: 0D VCallHresult put__ipropENABLEDTIMER ; Enabled 값 세트

```

⁷¹ Form Load 이벤트: VB에서 디자인한 윈도우 또는 다이얼로그를 폼이라고 부른다. 하나의 폼이 로드되기 직전에 발생하는 이벤트이며, 윈도우 또는 다이얼로그가 표시되기 전의 처리가 기술되어 있다.

Visual Basic P-Code 크랙 2

```
00402C9C: 1A FFree1Ad  
00402C9F: 13 ExitProcHRESULT
```

; 처리 끝

Form Load 이벤트 처리부분에서는 Visual Basic Timer 컨트롤의 Property를 설정하고 있다.

먼저 [00402C74⁷²: F5 LitI4: -> 7D0h 2000]에서 2000의 값을 스택에 저장한 후 [00402C83: 0D VCallHRESULT put__ipropINTERVALTIMER]에서 인수로 설정 된다. Timer 컨트롤의 Interval값은 밀리초로 지정하기 때문에(2000 → 2초), 품이 로드 되고 나서 2초 후에 Timer 이벤트가 발생하게 된다.

즉 [VBCrackMe1.exe]를 직접 실행해 보면, 메인 윈도우가 표시된 후 2초 후에 Nag 윈도우 창이 표시되었던 것이다. 따라서 메인 윈도우가 표시 되고 2초 후의 Nag 윈도우 타이밍 제어는 Timer 컨트롤을 사용하고 있다는 것을 알 수 있다. 이 후의 처리 [00402C97: 0D VCallHRESULT put__ipropENABLEDTIMER]에서는, Timer 컨트롤을 유효하게 하는 Property를 설정하고 있다. 이 때 인수로서 [00402C8B: F4 LitI2_Byte: -> FFh 255]에서 255(-1) 값을 스택에 저장(쌓아 올리고)한다. 여기에서 값을 설정하고 있는 Property의 속성은, Boolean형⁷³이므로 true값을 설정하고 있는 것이다.

이 단계에서 어떤 부분을 변경하면 Nag 창을 지울 수 있는지 알 수 있다. 즉 Timer 컨트롤이 이벤트가 발생하지 않도록 하면 되는 것이다. 정확히 [00402C83: 0D VCallHRESULT put__ipropINTERVALTIMER]에서 Timer를 유효하게 하고 있는 부분이 존재한다.

여기에서 설정하고 있는 true를 false로 하면, Timer 컨트롤은 Disable되어 Nag 창이 나타나지 않게 된다. 먼저, 인수를 스택에 저장하고 있는 [00402C8B: F4 LitI2_Byte: -> FFh 255]에 브레이크 포인트를 설정 한다.

[On Execution(Ctrl+E)] 버튼을 클릭하여 브레이크 포인트 설정 윈도우를 표시한다(그림 7).



⁷² 컴퓨터 환경에 따라 주소는 달라질 수 있다.

⁷³ Boolean형은, True와 False만 갖는 형식으로 크기는 2바이트다. Visual Basic에서는, true값이 =0(제로)가 사용되고 있다.

그림 7. 브레이크 포인트 설정 윈도우

[Add] 버튼 옆의 텍스트박스에 “00402C8B”를 입력하여 [Add] 버튼을 클릭하거나, 메인 화면에서 브레이크 포인트를 설정할 라인에서 더블 클릭하면 된다(그림 8).



그림 8. 브레이크 포인트 설정

위의 그림과 같이 브레이크 포인트가 추가되면, [Go!(F5)] 버튼을 클릭하여 실행한다. 그러면 브레이크 포인트가 설정된 부분에서 프로그램이 정지한다(그림 9).

Visual Basic P-Code 크래프트

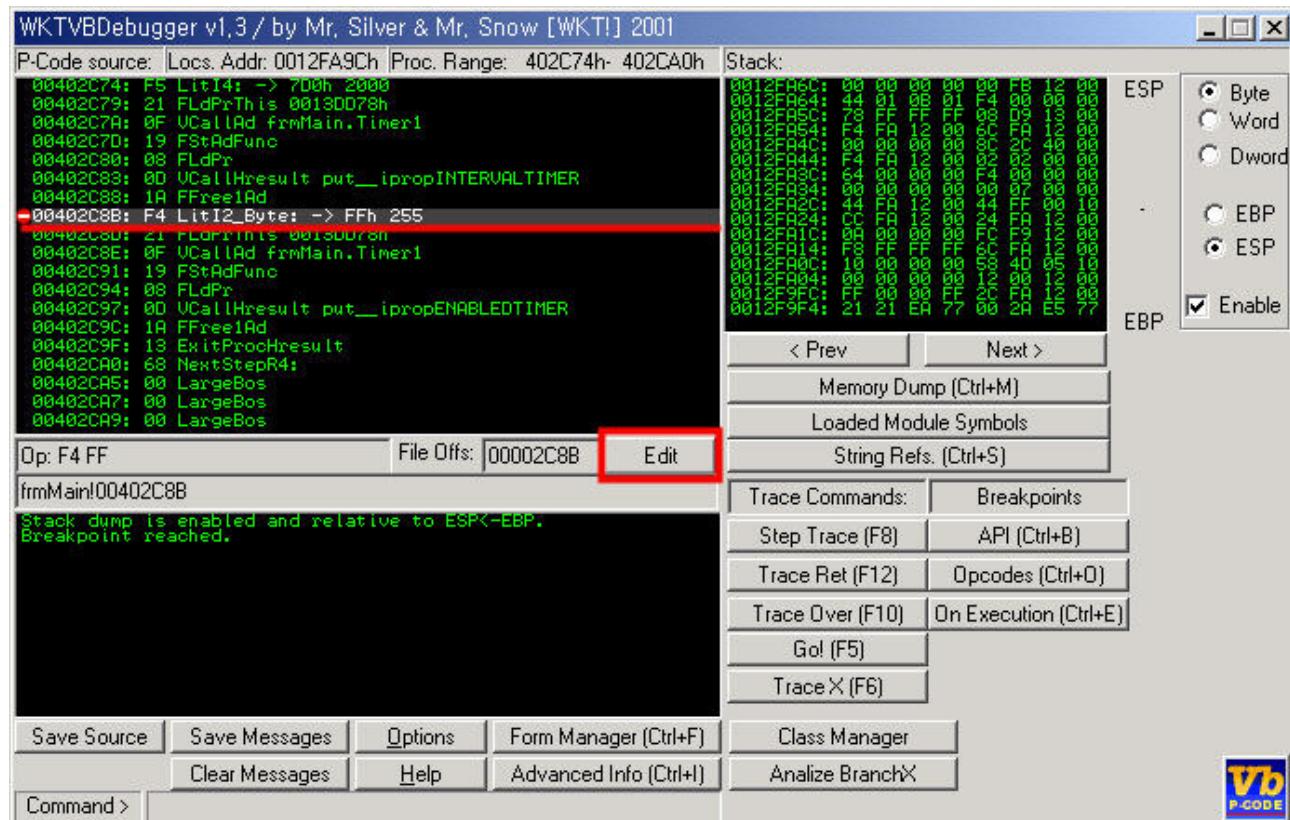


그림 9. 00402C8B에서 브레이크

여기에서 스택에 저장한 값을 true에서 false로 변경한다. [Edit] 버튼을 클릭하면, [Memory Viewer and editor] 윈도우가 표시된다(그림 10).

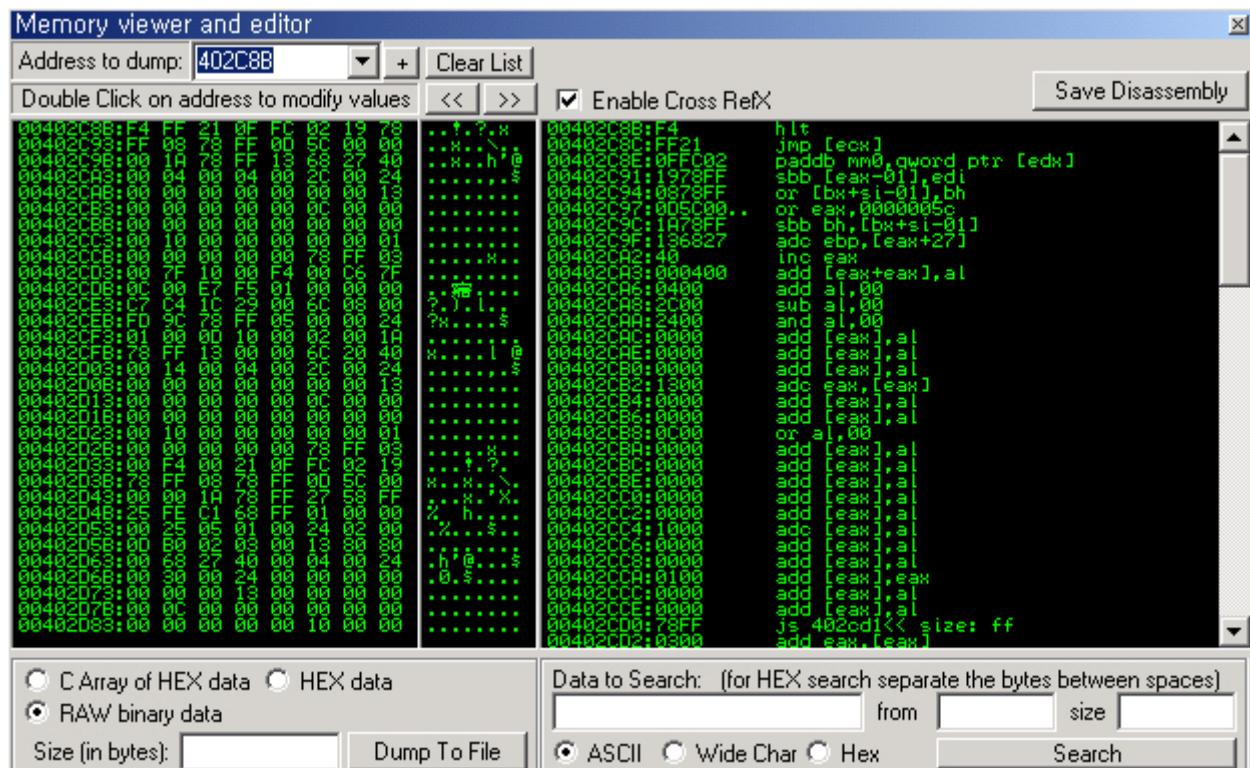


그림 10. Memory viewer and editor 윈도우

그리고, 어드레스 402C5F 로부터의 값이 표시되어 있는 행을 더블 클릭하면, 메모리 편집 화면이 나타난다(그림 11).

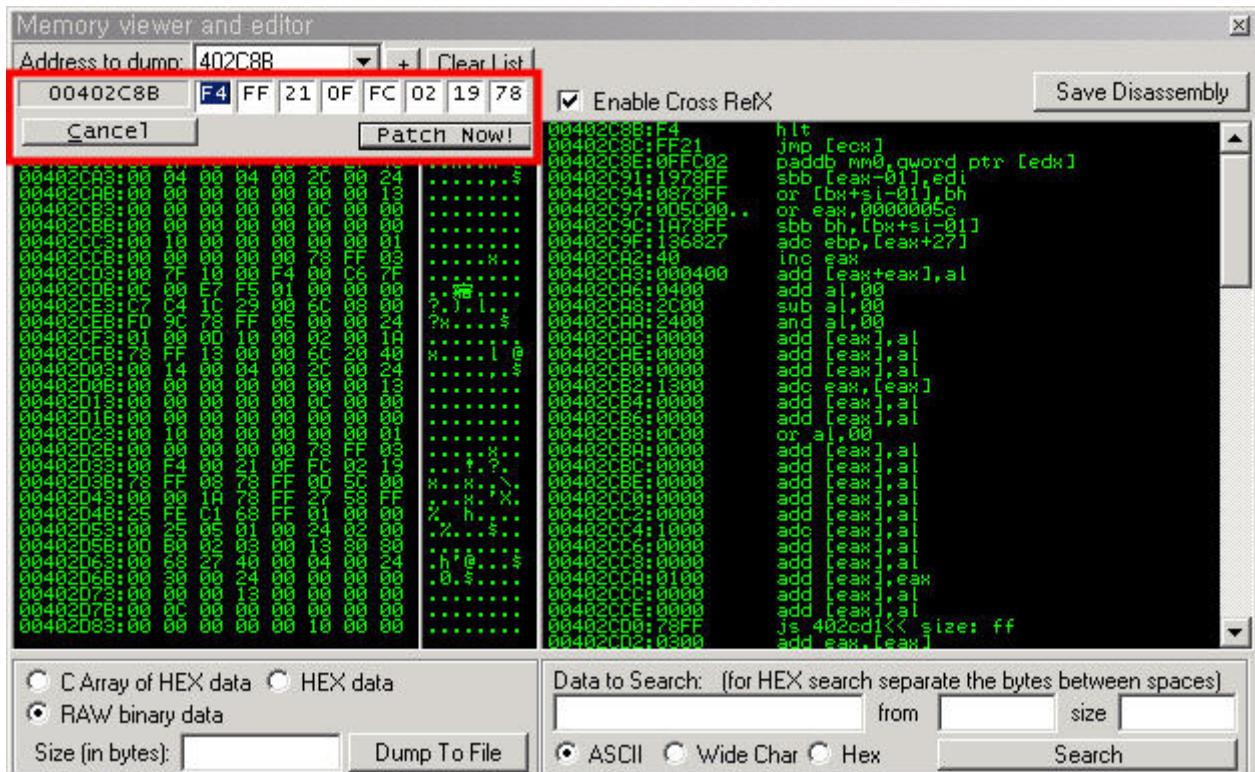


그림 11. 메모리 내용 변경

[F4] 옆의 [FF]를 [00(제로)]로 변경한 후 [Patch Now!] 버튼을 클릭하면 반영되어, P-Code 리스트 표시도 [00402C8B: F4 LitI2_Byte: -> 0h 0]로 바뀐다.



그림 12. 메모리 편집(FF -> 00)

여기에서 파일을 바꾸기 위하여 파일 Offset 값을 구합니다. [Edit] 버튼의 왼쪽에 [File Offs]라고 표시되어 있다(그림 13).

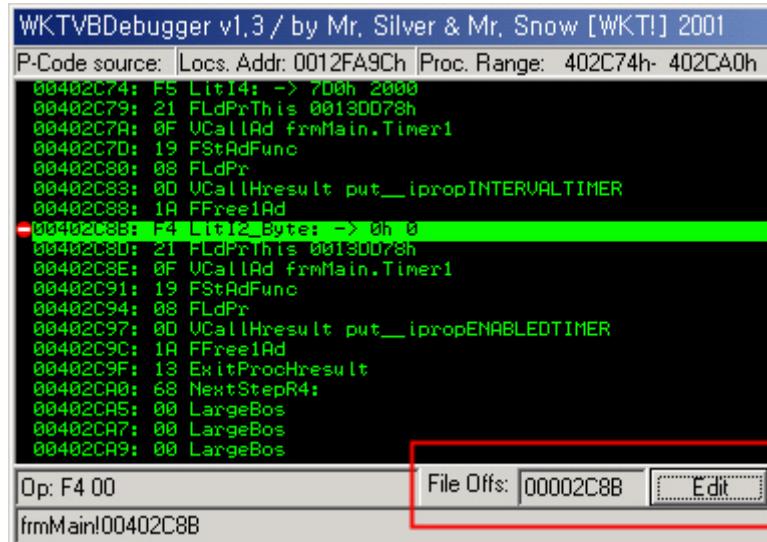


그림 13. 파일 오프셋 표시부분

지금, 표시되어 있는 Offset 값은 어드레스 00402C8B에 대응하고 있다. 따라서 다음 파일 Offset은 현재 Offset값+1인 00402C8D가 된다. 파일 Offset은 PE 파일의 RVA Offset과는 개념이 아예 다르다. [Go!(F5)] 버튼을 클릭하여 실행하면 Nag 표시가 나오지 않는 것을 확인할 수 있다.

Interval 속성 값 변경 패치	
00402C74: F5 LitI4: -> 7D0h 2000	; 4byte 값->스택
00402C79: 21 FLdPrThis 0013DD78h	
00402C7A: 0F VCallAd frmMain.Timer1	
00402C7D: 19 FStAdFunc	
00402C80: 08 FLdPr	
00402C83: 0D VCallHresult put__ipropINTERVALTIMER	; Interval 값 세트
00402C88: 1A FFree1Ad	
00402C8B: F4 LitI2_Byte: -> 0h 0	; 2Byte 값->스택 (Interval 을 0[False]으로)
00402C8D: 21 FLdPrThis 0013DD78h	
00402C8E: 0F VCallAd frmMain.Timer1	
00402C91: 19 FStAdFunc	
00402C94: 08 FLdPr	
00402C97: 0D VCallHresult put__ipropENABLEDTIMER	; Enabled 값 세트
00402C9C: 1A FFree1Ad	
00402C9F: 13 ExitProcHresult	; 처리 끝

그러나 Timer의 Enable Property를 설정하지 않을 수도 있으므로 다른 패치도 생각해 보겠다. 변경할 수 있는 부분은 Interval 값을 설정하고 있는 부분밖에 없다. 즉, WKTVBDE에서 [VBCrackMe1.exe]를 다시 불러 들인 후

Visual Basic P-Code 크랙 2

[Go!(F5)] 버튼을 클릭하여 실행한다.

Form_Load 이벤트 처리의 첫 부분에서 프로그램이 정지한다. 이번에는 처음 [00402C74: F5 LitI4: -> 7D0h 2000]에서 스택에 저장하고 있는 값을 2000에서 0(제로)으로 변경한다. [Edit]버튼을 클릭하여 [Memory viewer and editor] 윈도우에서 [F5]의 바로 다음 값 [D0]와 [07]을 [0(제로)]으로 변경한다(07D0h → 2,000).



그림 14. 메모리 편집(D007 → 0000)

그러면 [00402C74: F5 LitI4: -> 0h 0]으로 바뀌며, 변경 완료된 패치 이미지는 아래와 같다.

Enable 속성 값 변경 패치	
00402C74: F5 LitI4: -> 0h 0	; 4byte 값->스택(2 초를 0 으로)
00402C79: 21 FLdPrThis 0013DD78h	
00402C7A: 0F VCallAd frmMain.Timer1	
00402C7D: 19 FStAdFunc	
00402C80: 08 FLdPr	
00402C83: 0D VCallHresult put__ipropINTERVALTIMER	; Interval 값 세트
00402C88: 1A FFree1Ad	
00402C8B: F4 LitI2_Byte: -> FFh 255	; 2Byte 값->스택
00402C8D: 21 FLdPrThis 0013DD78h	
00402C8E: 0F VCallAd frmMain.Timer1	
00402C91: 19 FStAdFunc	
00402C94: 08 FLdPr	
00402C97: 0D VCallHresult put__ipropENABLEDTIMER	; Enabled 값 세트
00402C9C: 1A FFree1Ad	
00402C9F: 13 ExitProcHresult	; 처리 끝

그러나 매번 Nag 창을 없애기 위해서 디버깅할 수 없으므로, 직접 바이너리를 패치 하여야 한다. 위의 패치코드를 바이너리 편집기를 통해 수정하면 손쉽게 Nag 창이 없어진 실행파일을 만들 수 있다. 아래는 “Enable 속성 값 변경 패치” 부분을 수정하는 예제이다.

① Hex 편집기로 VBCrackMe1.exe 를 불러온다.

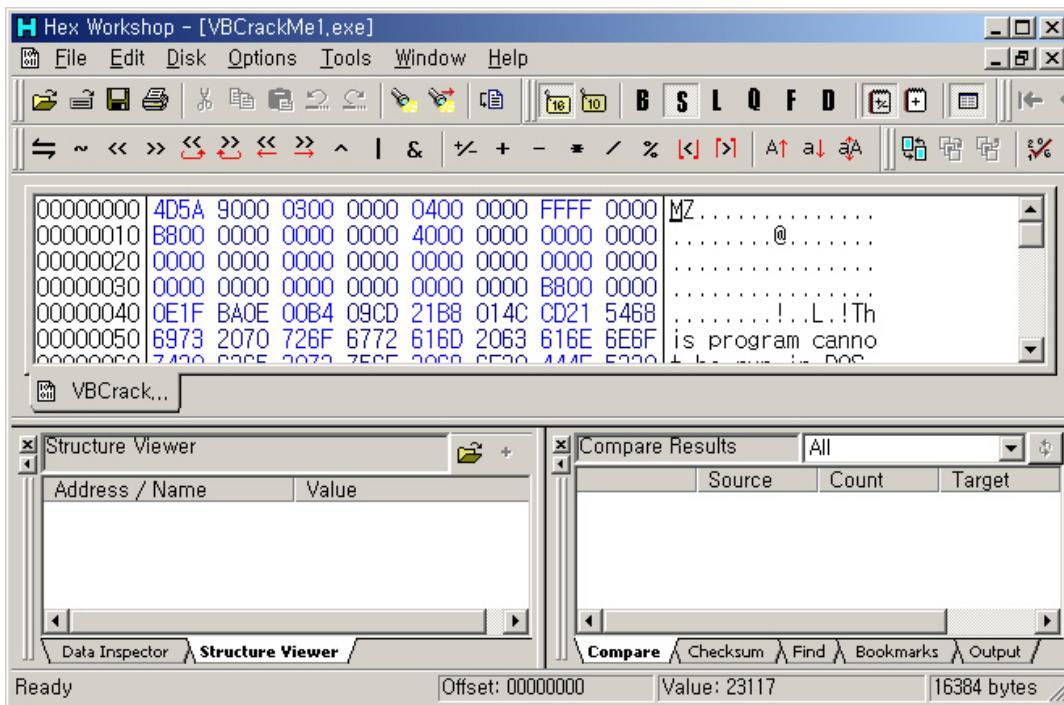


그림 15. Hex 편집기로 VBCrackMe1.exe 불러오기

② Enable 속성 변경 패치를 하기 위해, 00402C74 주소의 명령 코드값인 “F5D0070000210FFC”를 Hex 값으로 검색한다.

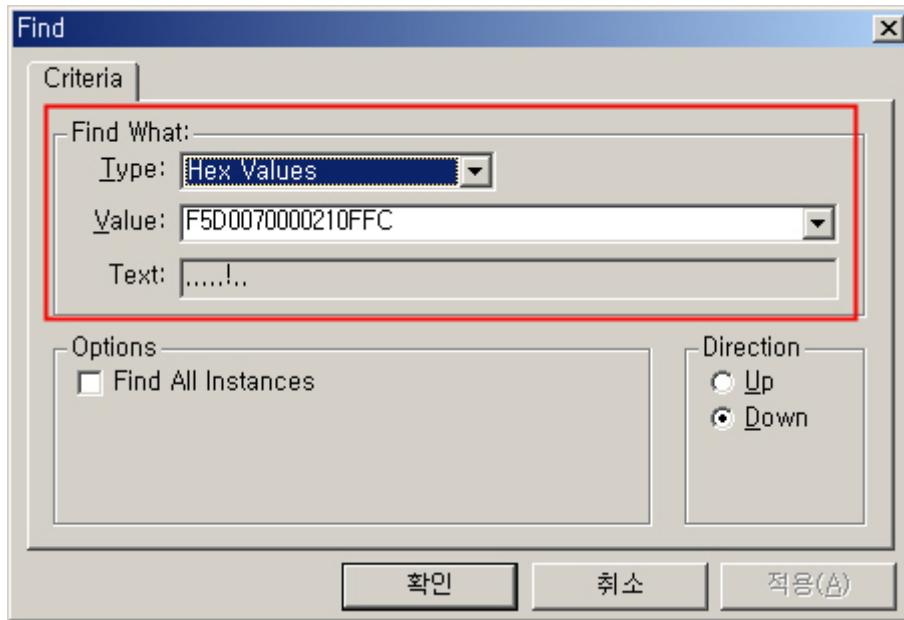


그림 16. Enable 속성 값 검색

③ 검색된 결과값 “F5D007”을 “F50000”로 변경한 후 저장하면 Nag 창이 사라진 바이너리 파일을 최종적으로 만들 수 있다(Hex Workshop은 수정할 부분을 마킹한 후 CTRL+Insert 키로 수정하여야 함).

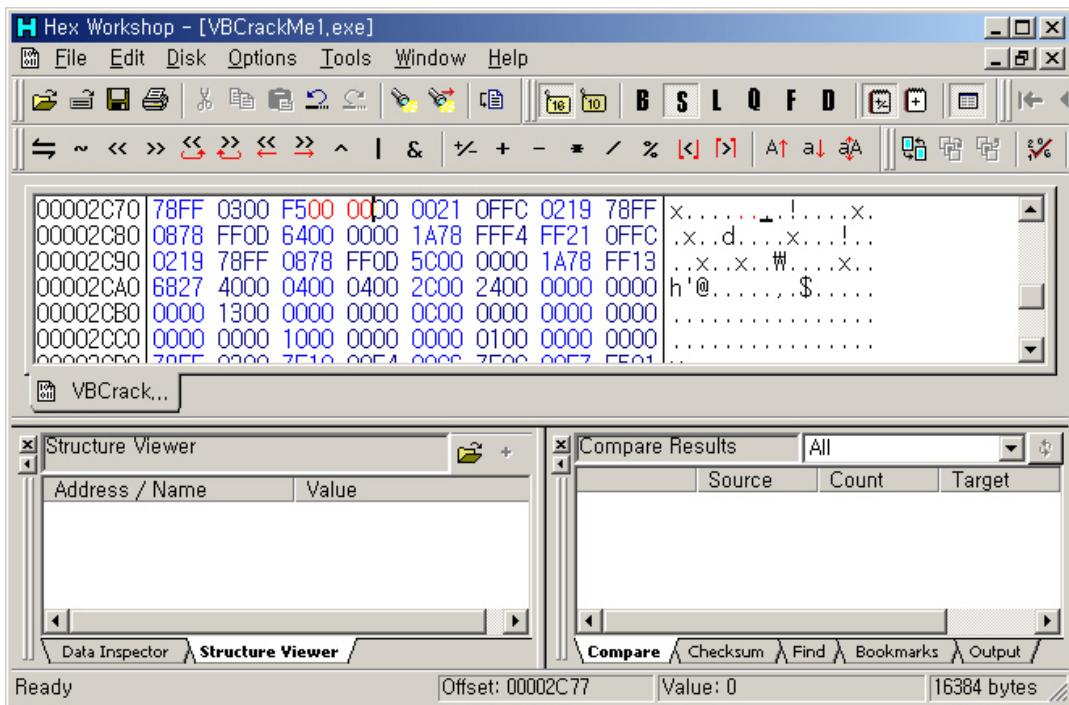


그림 17. F5D007 => F50000 수정

마지며...

[VBCrackMe1.exe]는 Visual Basic에서 사용되는 Timer 컨트롤을 이용하여 시간제어를 수행하고 있다. Timer 컨트롤을 사용하지 않고 시간 제어를 수행하는 경우는 Windows API의 [GetTickCount]를 사용해도 가능하다. 단, Timer 컨트롤도 [GetTickCount]를 사용하고 있기 때문에, Timer 컨트롤과 [GetTickCount] 양쪽을 동시에 사용하는 것은 불가능하다. GetTickCount 함수(or timeGettime 함수)는 게임해킹의 스피드핵을 구현할 때에도 자주 이용된다. 보다 자세한 내용은 “Game Reversing” 부분에서 설명하도록 하겠다.

Visual Basic 프로그램은 일부 Decompile이 가능하기 때문에, 원본 소스코드를 복원할 수 있다. 위에서 설명한 VBCrackMe1.exe 파일을 VB Decompiler⁷⁴라는 프로그램을 이용하면 Visual Basic으로 컴파일한 EXE, OCX, DLL의 소스 코드를 확인할 수 있다.

⁷⁴ VBDecompiler: www.vb-decompiler.org

Visual Basic P-Code 크랙 2

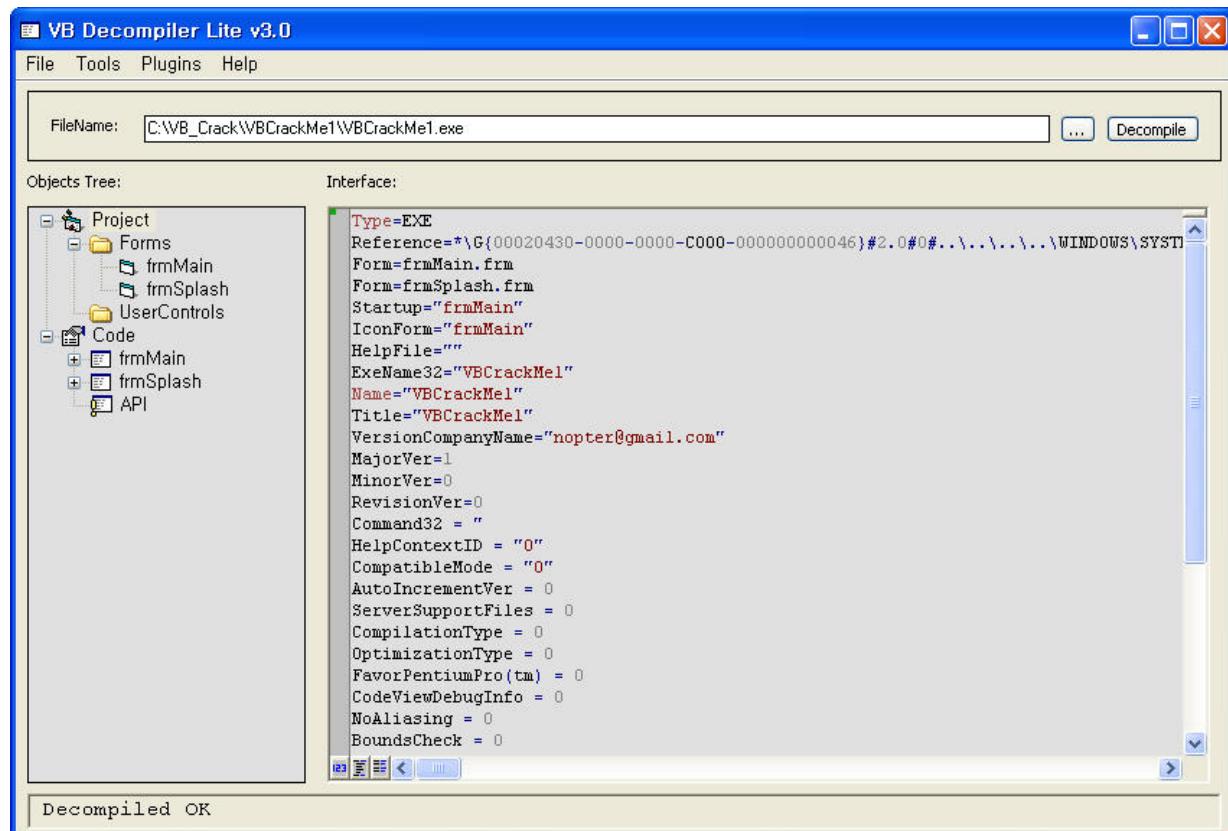


그림 18. VB Decompiler

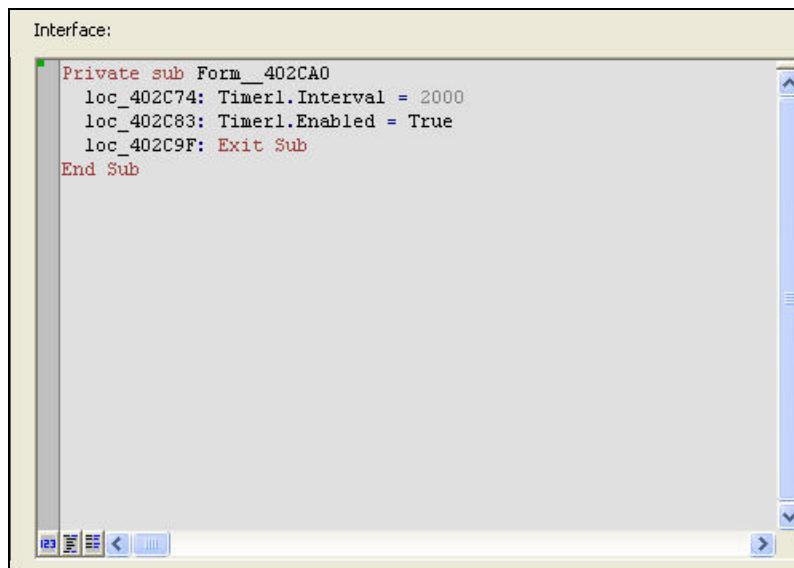


그림 19. frmMain.Form Decompile

제 4 장 Windows 크랙 응용편

4.3 P-Code 크랙편(사용기한 없애기)

WTKVBDE 프로그램을 이용하여 다양한 Visual Basic P-Code 프로그램들을 크랙하는 예제를 살펴보도록 하겠다. 이번에는 일정 기간만 프로그램을 사용할 수 있는 Shareware 프로그램을 크랙하는 예제이다.

사용기한 없애기

첨부파일 [VBCrackMe2.exe]을 직접 실행하면 사용기한과 사용할 수 있는 일수가 표시된 후 [VB Crack Me #2] 윈도우가 표시된다(그림 1, 그림 2). 만약 2006년 12월 31일이 지났다면 사용기간이 만료되었다는 알림창을 띄우며 프로그램을 더 이상 사용할 수 없다. 정상적인 실행 상태를 확인하려면 컴퓨터의 날짜를 2006 년 12 월 31 일 이전으로 변경한 후 첨부파일 [VBCrackMe2.exe]를 실행하면 된다.

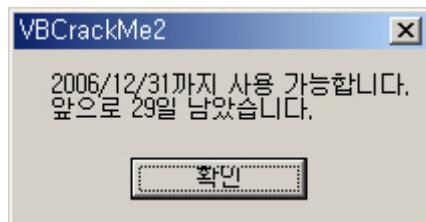


그림 1. 사용기한을 표시하는ダイ얼로그

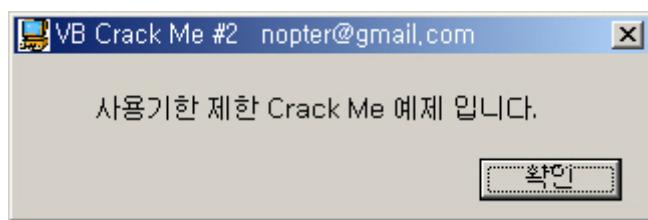


그림 2. VB Crack Me #2

본격적인 크랙을 위해 [WKTVBDE]를 실행한다. 메뉴의 [File]>[Open…]에서 [VBCrackMe2.exe]를 읽어 들인 후, 메뉴의 [Action]>[Run]으로 실행시킨다.

불러온 disassemble 코드는 Textedit 컨트롤 문자를 획득하기 위해 여러가지 처리를 수행하고 있다는 것을 알 수 있다. 먼저 WKTVBDE의 P-Code 리스트 표시에서 비교구문을 시작하는 라인(화면 마지막 라인) [00402E28: 04 FLdRfVar 0012FA18h]⁷⁵를 더블 클릭하여 브레이크 포인트를 설정한다(그림 3).

⁷⁵ 컴퓨터 환경에 따라 주소가 달라 질 수 있다.

```
00402DE4: 04 FLdRfVar 0012FA58h
00402DE7: 21 FLdPrThis 0013DD78h
00402DE8: 0F VCallAd frmMain.Text1
00402DEB: 19 FStAdFunc
00402DEE: 08 FLdPr
00402DF1: 0D VCallIResult get__ipropTEXTEDIT
00402DF6: 04 FLdRfVar 0012FA48h
00402DF9: 0A ImpAdCallIFPR4 rtcGetPresentDate on address 660D1893h
00402DFE: F5 LitI4: -> 1h 1
00402E03: F5 LitI4: -> 1h 1
00402E08: 3A LitVarStr 'yyyy/mm/dd'
00402E0D: 4E FStVarCopyObj 0012FA28h
00402E10: 04 FLdRfVar 0012FA28h
00402E13: 04 FLdRfVar 0012FA48h
00402E16: 04 FLdRfVar 0012FA18h
00402E19: 0A ImpAdCallIFPR4 rtcVarFromFormatVar on address 660F58ABh
00402E1E: F5 LitI4: -> 1h 1
00402E23: F5 LitI4: -> 1h 1
00402E28: 04 FLdRfVar 0012FA18h ; Breakpoint set
```

Visual Basic P-Code 크래시

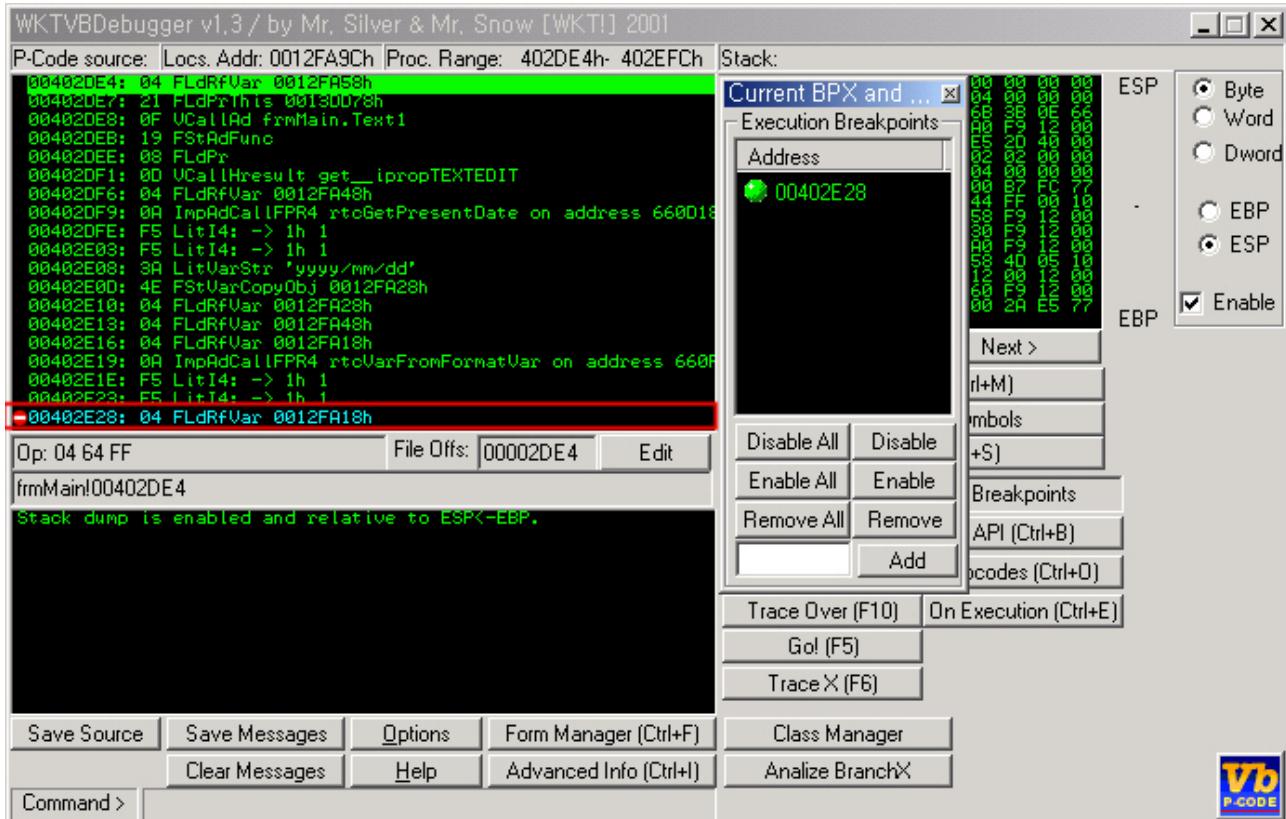


그림 3. 브레이크 포인트 설정

[Go!(F5)] 버튼을 클릭하여 실행하면 설정한 브레이크 포인트에서 프로그램이 멈춘다(그림 4).



그림 4. 00402E28에서 브레이크

여기에서는, P-Code 리스트에 표시되어 있는 0012FA18h⁷⁶의 주소값을 스택에 저장하려고 하고 있다. [Memory

⁷⁶ 주소값 0012FA18h는 스택 영역의 주소를 나타내며, 이 스택영역은 실행하고 있는 운영체제 환경에 의해 변화될 수 있다.

Visual Basic P-Code 크랙 3

Dump(Ctrl+M)] 버튼을 클릭하면 [Memory viewer and editor] 윈도우가 표시되며, [Address to dump:] 란에 “0012FA18”를 입력하면 내용이 표시된다.

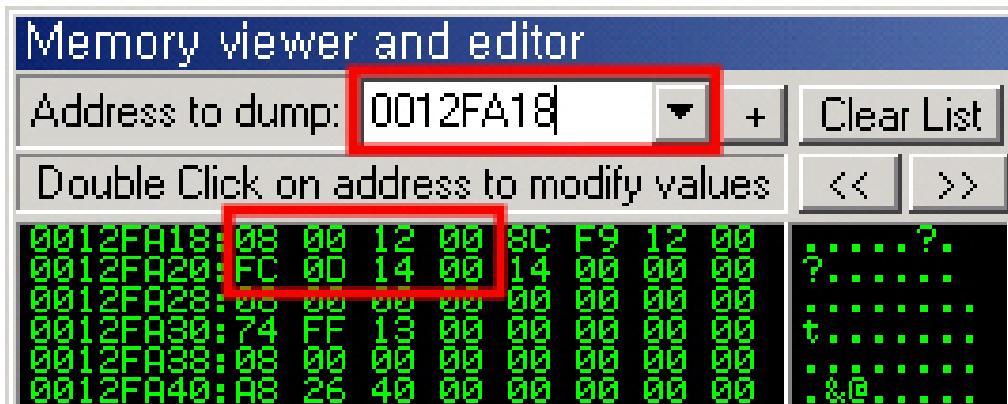


그림 5. 0012FA18h 주소 메모리 덤프

처음 1 바이트가 08h 로 되어 있는 것을 알 수 있다. 일단 정수 값으로 표시하고 있는 것처럼 보이지만, 다음 오프셋 주소가 가리키는 0x00140DFC 의 내용을 보면 정확한 변수형을 알 수 있다. 해당 메모리에 저장된 값은 오늘의 년/월/일 문자열이 Unicode 형식으로 들어 있다(그림 6). 따라서 0x12FA18 주소가 나타내고 있는 데이터 속성은 Visual Basic에서 variable type(※참고 1)중 문자열 변수라는 것을 알 수 있다.

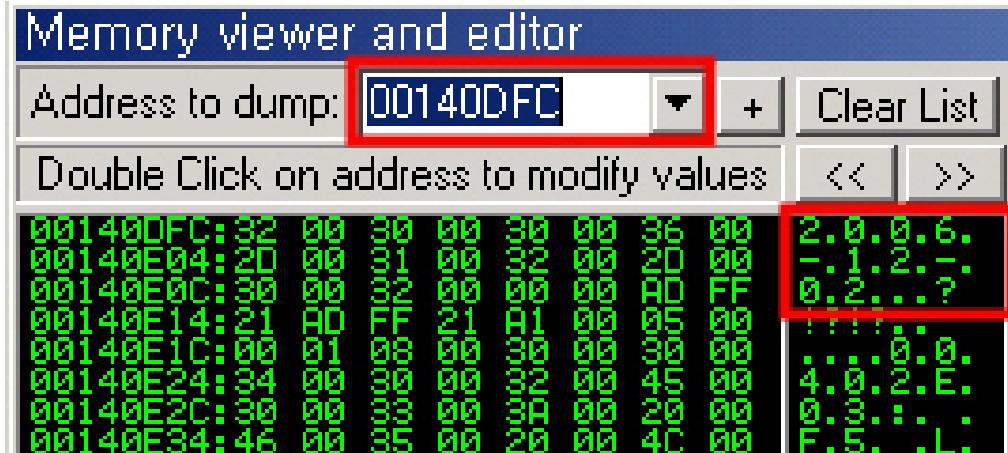


그림 6. 현재 년/월/일

※참고 1) Variant 는 정수값, 문자열 등 Visual Basic에서 사용하는 모든 데이터를 취급하는 변수이다.
Variable type의 구성은 아래와 같다.

MEM	+0	+1	+2	+3	+4	+5	+6	+7
00	08	00	xx	xx	xx	xx	xx	xx
08	FC	0D	14	00	xx	xx	xx	xx

Variant 는 16 바이트로 구성되며, 선두 2 바이트에 데이터형 코드(아래 표의 16 진수 값), 하위 8 바이트에 데이터 값이 들어간다. 단, 문자열은 문자열의 주소 값이 들어간다.

16 진수 값	설명	기타
0	Empty 값	초기화가 안된 값
1	Null 값	
2	Integer	정수
3	Long	실수
4	Single	부호없는 정수
5	Double	
6	currency	화폐, 통화 단위
7	Date	날짜
8	String	문자열
9	Object	
10	Error Value	에러 값
11	Boolean	True, False
12	Variant	Variant 형 배열에만 사용
13	비 OLE automation object	
14	10 진수형	
17	바이트형(Byte)	
36	User Define 형을 포함한 Variant 형	사용자 정의
8192	array	배열

표. Data Type 코드

다음에 [00402E34: 04 F1dRfVar 0012F9F8h]에 브레이크 포인트를 설정하고 [Go!(F5)] 버튼을 클릭하여 실행시킨다(그림 7).

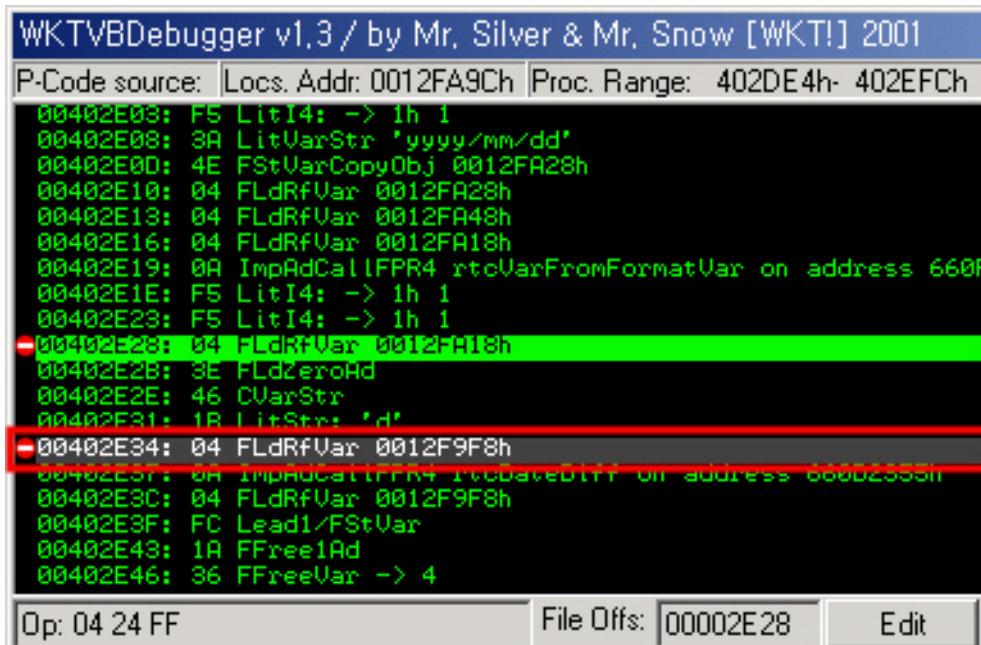


그림 7. 비교구문

브레이크 포인트를 설정한 부분에서 실행이 멈추면, 0x12F9F8의 내용도 참조해 보면 모두 0(제로)이다. 아마 밑의 행[00402E37: 0A ImpAdCallFPR4 rtcDateDiff on address 660D2355h] 결과가 들어갈 영역이 되는 것 같다.



그림 8. 0x0012F9F8 메모리 덤프

rtcDateDiff는 두 개의 날짜 간격을 구하는 함수로, 처음 일자는 좀 전에 확인한 오늘 일자가 되고, 두 번째 일자는 WKTVBDE 윈도우의 스택값 표시부분의 첫번째 행 하위 4 바이트 주소 값이 된다(그림 9).

Visual Basic P-Code 크랙 3

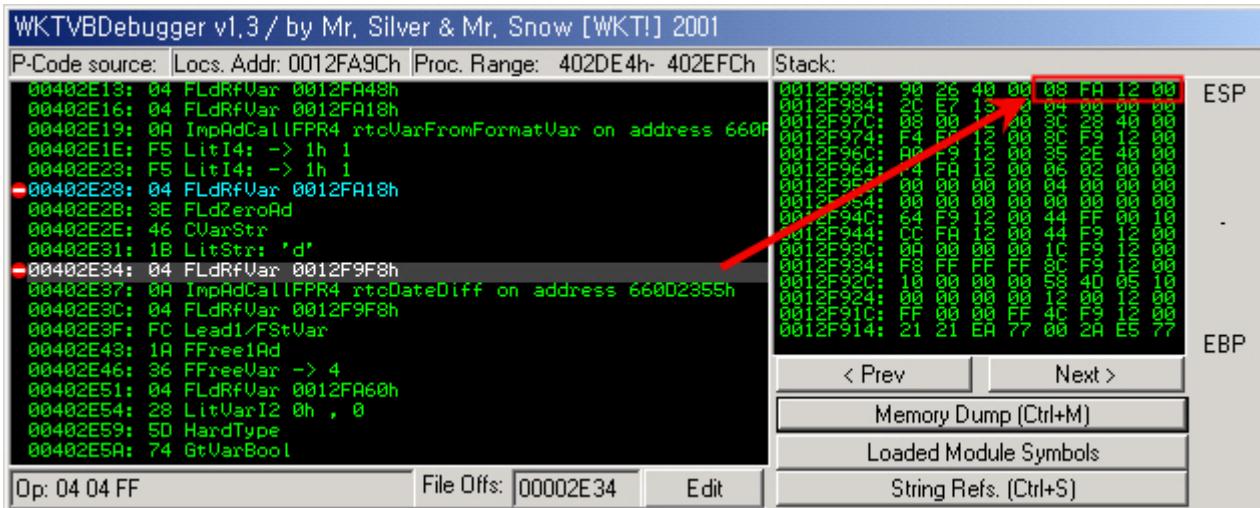


그림 9. 스택 표시부분

0x12FA08의 내용은 아래와 같이 Variant형 문자열 변수로, 문자열을 참조하면 사용기한의 날짜로 되어 있다.

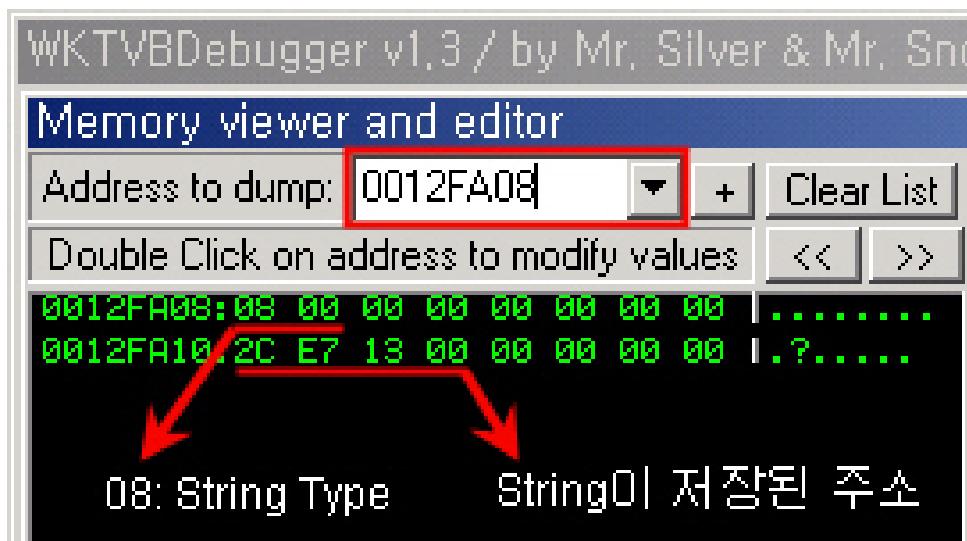


그림 10. 0x0012FA08 메모리 덤프



그림 11. 0x0013E72C 메모리 덤프

여기에서 사용기한까지의 날짜를 구하고 있다. 그러면 [Step Trace(F8)] 버튼을 2 번 클릭해서 [00402E3C: 04 FLdRfVar 0012F9F8h]까지 진행한다. 날짜간격 값이 들어 있는 변수 주소값 0x12F9F8를 [Memory viewer and

Visual Basic P-Code 크랙 3

editor] 원도우에서 참조하면 실수(Long)형을 가지고 있는 Variant 형(0x00 03) 변수인 것을 알 수 있다. 변수에 저장된 값은 4 바이트로 “FFE3” 이므로 227(-29 일)이 들어있다.



그림 12. 0x0012F9F8(사용기한)

다음은 비교를 하고 있는 [00402E5A: 74 GtVarBool]에 브레이크 포인트를 설정한 후 [Go!(F5)] 버튼을 클릭하여 실행한다(그림 13).

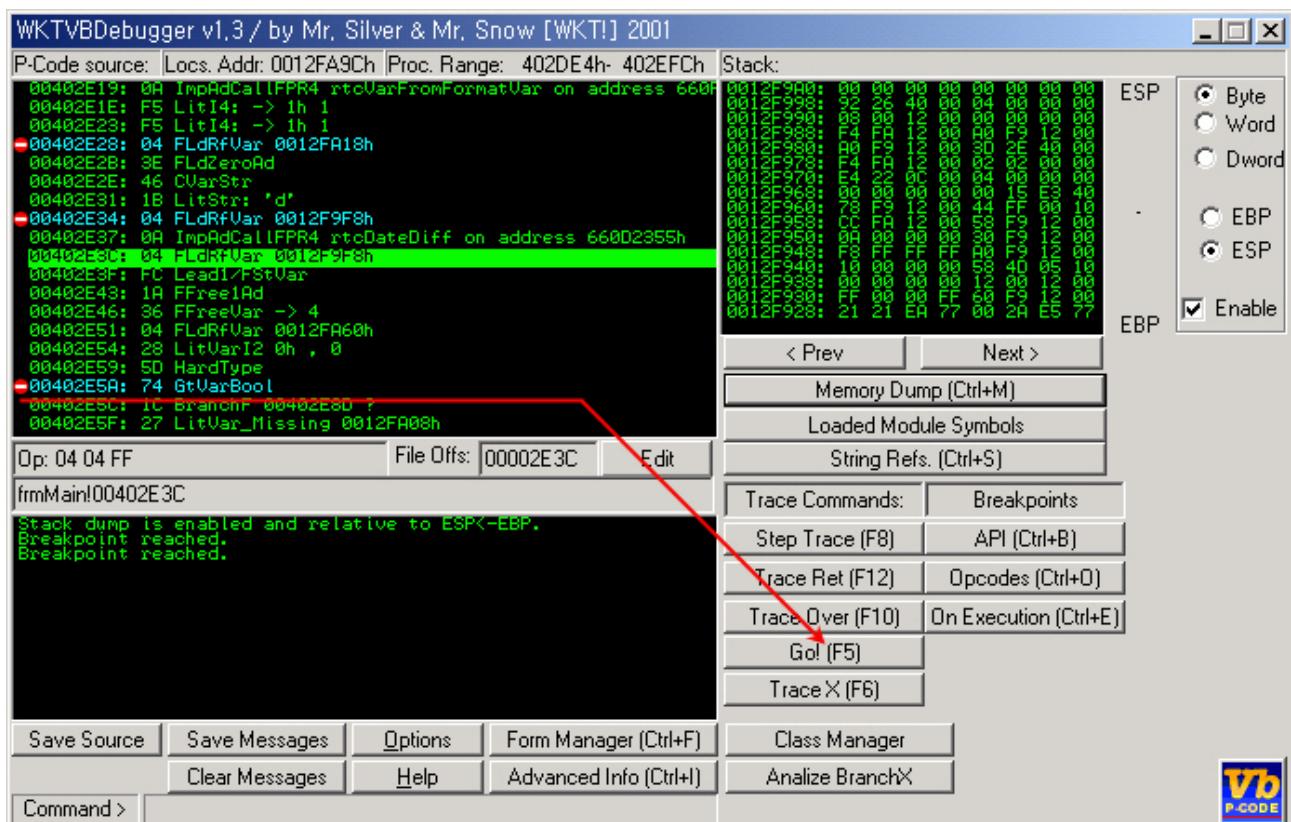


그림 13. 비교구문 브레이크 포인트 설정

비교대상 값은 WKTVBDE 윈도우의 스택 표시 부분의 첫번째 행 상위 4 바이트와 하위 4 바이트 변수가 된다(그림 14).

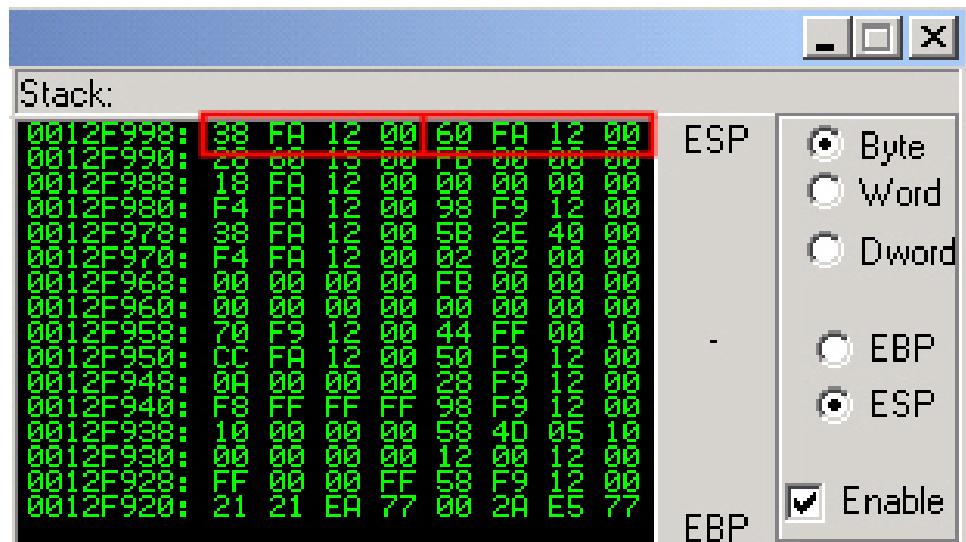


그림 14. 비교전의 스택 부분

첫번째 스택 주소(0x0012FA38) 메모리를 덤프하면 아래와 같다(그림 15).

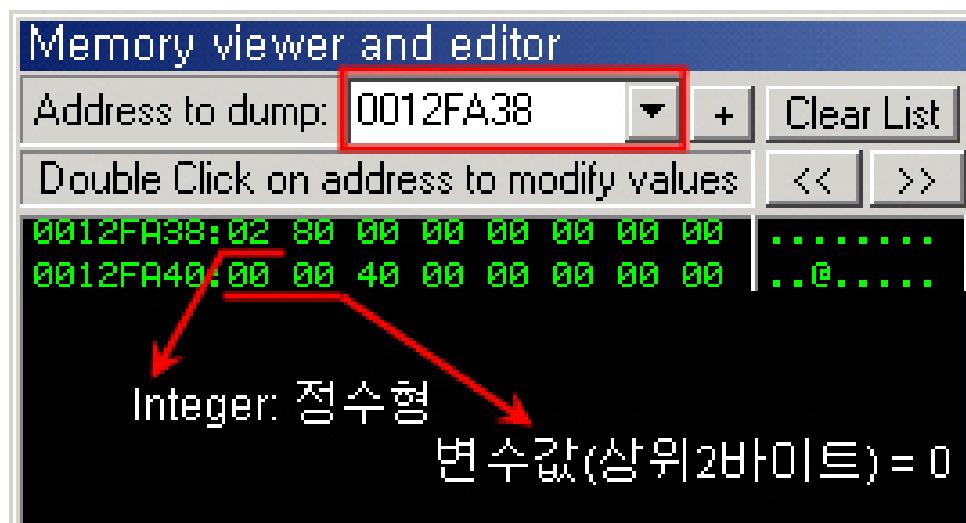


그림 15. 0x0012FA38 메모리 덤프

두번째 스택 주소(0x0012FA60) 메모리를 덤프하면 아래와 같다(그림 16).



그림 16. 0x0012FA60 메모리 덤프

여기에서 비교조건은 P-Code [GtVarBool]의 처음 2 문자(Gt: GreaterThan)를 비교[P2>P1]하는 것이다. 실제 숫자로 치환하면 [-227>0]⁷⁷의 비교를 수행한 후 결과를 Boolean형으로 되돌려 준다. [Step Trace(F8)] 버튼을 클릭하여 비교하면, 결과 값이 Boolean(00)값으로 스택에 저장 된다(그림 17). 결과적으로 0(제로=False)값이 들어간다.



그림 17. 비교 수행 후 Stack에 결과값 저장

다음 파일 Offset은 P-Code의 점프하는 명령 [00402E5C: 1C BranchF 00402E8D(Jump)]이다. 이 [BranchF]⁷⁸는 비교결과가 false일 때 점프한다.

이 시점에서 Window NT 계열 OS(※참고 2)를 사용하는 WKTVBDE에서는 P-Code 리스트 부분에, 사용기한이 지났다는 알림 문자열이 [00402E6D: 3A LitVarStr '사용불가능합니다.'][에 표시되어 있다. 실행이 멈춰있는

⁷⁷ 필자가 프로그램을 실행시킨 컴퓨터의 날짜는 2006년 12월 2일이다.

⁷⁸ BranchF 명령은, 비교결과가 false일 때 점프하는 명령이다. 이 명령과는 반대로 비교결과가 true일 때 점프하는 명령은 [BranchT]이다. 또 [Branch]는 무조건 점프하는 명령이다.

Visual Basic P-Code 크랙 3

분기부분이 사용기한을 넘기고 있는 분기구문이다.

여기서 패치 이미지를 생각하여 다음 점프할 곳을 찾게 되지만, WKTVBDE P-Code 리스트에는 점프할 주소까지 보이지 않는다. 그래서 디컴파일한 리스트를 필요로 하게 되는 것이다.

※참고 2: Visual Basic의 문자열은 Unicode 형식이 표준으로 되어 있다. Windows NT 계열 OS는 Unicode 형식을 지원하고 있기 때문에 표시 가능한 것이다. Unicode를 지원하지 않는 Windows95 계열의 OS 환경에서는 표시되지 않는다.

VBCrackMe2.exe 디컴파일 리스트

Proc: 402efc

402DE4: 04 FLdRfVar local_009C ; Form_Load 처리 부분

[... 중략 ...]

402E43: 1a FFree1Ad local_0098

402E46: 36 FFreeVar local_00AC local_00CC local_00EC local_00DC

402E51: 04 FLdRfVar local_0094

402E54: 28 LitVarI2: (local_00BC) 0x0 (0)

402E59: 5d HardType

402E5A: Lead0/74 GtVarBool ; '남은 날수 > 0' 비교

402E5C: 1c BranchF: 402E8D ; 비교 결과 분기 부분

[... 중략 ...]

402E7D: 36 FFreeVar

402E88: Lead1/c8 End

402E8A: 1e Branch: 402ef9

402E8D: 27 LitVar_Missing ; 기간내의 수행 시작부

402E90: 27 LitVar_Missing

[... 생략 ...]

402ED5: Lead0/ef ConcatVar

402ED9: 0a ImpAdCallFPR4:

402EDE: 32 FFreeStr

402EE7: 1a FFree1Ad	local_0098
402EEA: 36 FFreeVar	; 무조건 점프 ←
402EF9: 13 ExitProcResult	
[... 생략 ...]	

[402E5C: 1C BranchF 402E8D]를 무조건 점프로 변경한다. 점프주소는 사용 기한 이내일 경우 처리하는 마지막 부분 [402EEA: 36 FFreeVar]이 되며, 패치 이미지가 완성되었다면 점프할 주소를 계산한다. Visual Basic의 점프는 실행하고 있는 주소로부터 상대 Offset이 아니라, 실행부 처음으로부터의 상대 Offset이다.

점프할 곳의 주소 0x402EEA에서, From_Load 처리의 선두 어드레스 0x402DE4를 뺀 값 0x106이 점프할 때 사용하는 Offset 값이다. [Edit] 버튼을 클릭하여 [Memory viewer and editor] 윈도우에서 처음 3 바이트를 “1E 06 01”로 변경한다(그림 18). Branch 명령으로 바꾸기 위한 명령어는 1E이다.

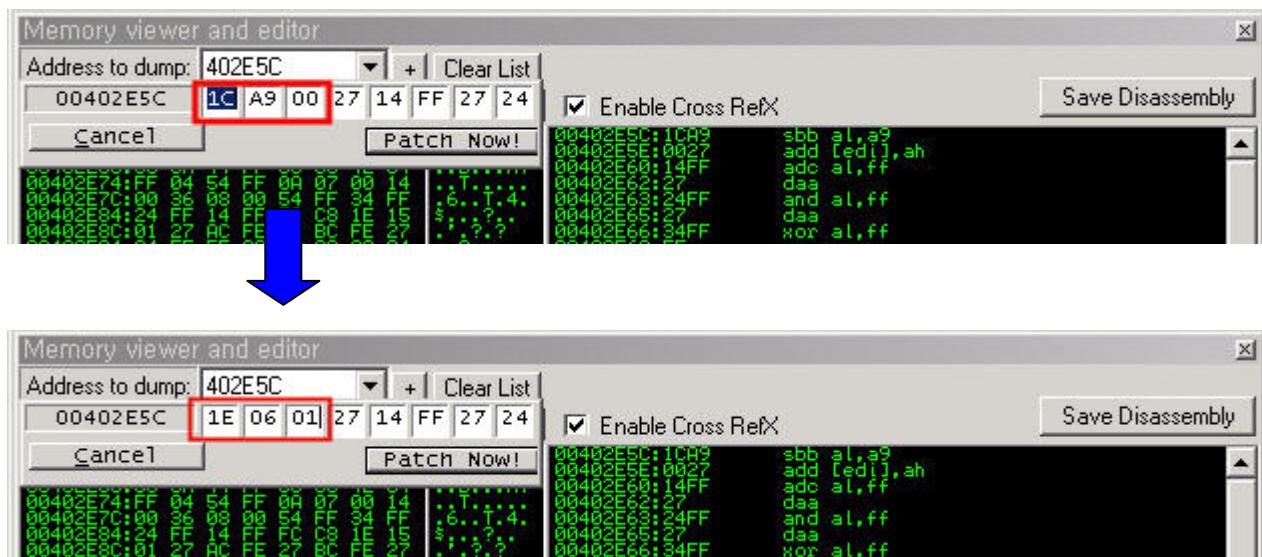


그림 18. 점프 명령의 변경

그리면 P-Code 리스트는 [00402E5C: 1E Branch 00402EEA (Jump)]으로 바뀐다. 실제로 패치한 파일 오프셋값은 표시되어 있는 0xE5C에서 3 바이트 변경된다.



그림 19. 파일 패치

그러면 [Go!(F5)] 버튼을 클릭하여 동작을 확인하도록 하자.

사용기간의 메시지가 표시되지 않게 되며, 최종 패치 이미지는 아래와 같이 된다.

주소	0x00402E5C
변경 전	1C A9 00
변경 후	1E 06 01

그러나 사용기한을 연장하기 위해 매번 디버깅할 수 없으므로, 직접 바이너리를 패치 하여야 한다. 위의 패치코드를 바이너리 편집기를 통해 수정하면 손쉽게 사용기한을 없앤 실행파일을 만들 수 있다. 바이너리 패치시 주의할 점은 동일한 명령 코드가 다른 곳에 있을 수 있으므로, 전 후 명령어(OPCODE)를 조합해서 검색하여야 한다.

① Hex 편집기로 VBCrackMe2.exe 를 불러온다.

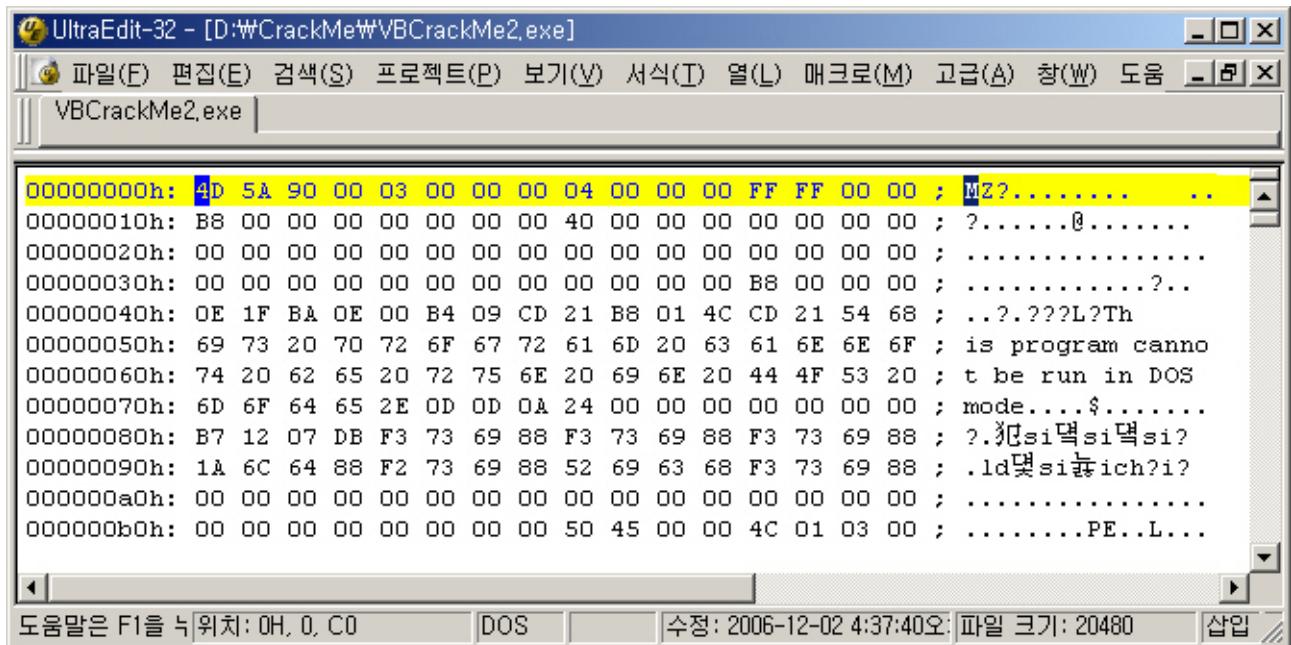


그림 20. Hex 편집기로 VBCrackMe1.exe 불러오기

② 사용기한 체크를 무조건 참(True)으로 만들어야 하므로, 00402E5C 주소의 명령 코드값인 “1CA900”를 Hex 값으로 검색한다.

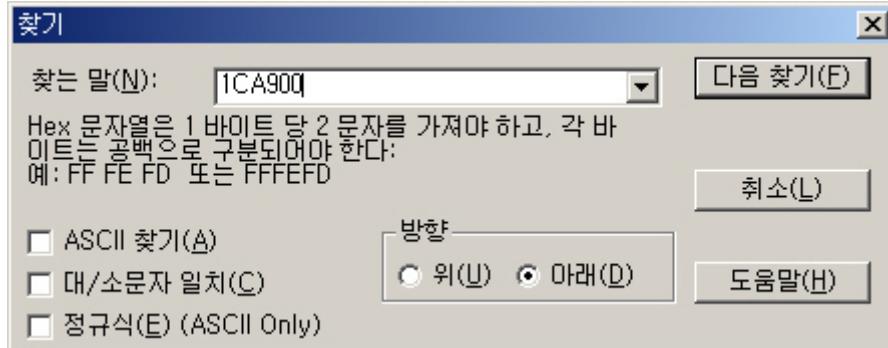


그림 21. 사용기한 체크 명령어 Hex 값 검색

③ 검색된 결과값의 “1CA900”을 “1E0601”로 변경한 후 저장하면 사용기한이 없어진 바이너리 파일을 최종적으로 만들 수 있다.

```

00002df0h: FF 0D A0 00 00 00 04 54 FF 0A 01 00 04 00 F5 01 ; .?...T ....?
00002e00h: 00 00 00 F5 01 00 00 00 3A 44 FF 02 00 4E 34 FF ; ...?...:D ..N4
00002e10h: 04 34 FF 04 54 FF 04 24 FF 0A 03 00 14 00 F5 01 ; .4 .T .$. ....?
00002e20h: 00 00 00 F5 01 00 00 00 04 24 FF 3E 64 FF 46 14 ; ...?...$ >d F.
00002e30h: FF 1B 04 00 04 04 FF 0A 05 00 18 00 04 04 FF FC ; ..... .... ??
00002e40h: F6 6C FF 1A 68 FF 36 08 00 54 FF 34 FF 14 FF 24 ; ? .h 6..T 4 . $?
00002e50h: FF 04 6C FF 28 44 FF 00 00 5D FB 74 1E 06 01 27 ; .1 (D ..]?)??
00002e60h: 14 FF 27 24 FF 27 34 FF F5 00 00 00 00 3A 44 FF ; . '$ '4 ?...:D
00002e70h: 06 00 4E 54 FF 04 54 FF 0A 07 00 14 00 36 08 00 ; ..NT .T .....6..
00002e80h: 54 FF 34 FF 24 FF 14 FF FC C8 1E 15 01 27 AC FE ; T 4 $ . 恍...??
00002e90h: 27 BC FE 27 04 FF F5 00 00 00 00 04 64 FF 21 OF ; '송'. ?...d !.
00002ea0h: FC 02 19 68 FF 08 68 FF OD A0 00 00 00 6C 64 FF ; ? .h .h .?..ld

```

도움말은 F1을 누르거나 위치: 2e50h, 11870, C0 | DOS | 수정: 2006-12-02 4:37:40오 | 파일 크기: 20480 | 삽입

그림 17. 1CA900 => 1E0601 수정

위의 크랙 패치 말고도 더 많은 방법들이 존재한다. 예를 들어 사용 기한을 2006년 12월 31일이 아니라 3006년 12월 31일로 변경하거나 날짜 간격을 무조건 참으로 수정해도 기간연장을 할 수 있다.

```

00001e20h: 24 05 00 46 6F 72 6D 31 00 26 00 27 00 35 2D 00 ; $.Form1.&.'5-.
00001e30h: 00 00 4A 01 00 00 84 12 00 00 B0 04 00 00 40 01 ; ...J...?..?..0.
00001e40h: 00 00 00 90 01 DC 7C 01 00 04 B1 BC B8 B2 46 02 ; ...??...굴림F.
00001e50h: FF 01 2B 00 00 00 01 05 00 54 65 78 74 31 00 02 ; .+.....Text1..
00001e60h: 04 00 00 48 03 BF 04 3B 01 09 00 0B 0A 00 33 30 ; ...H.?;.....30
00001e70h: 30 36 2F 31 32 2F 33 31 00 12 02 00 FF 03 27 00 ; 06/12/31.... .
00001e80h: 00 00 02 07 00 63 6D 64 51 75 69 74 00 04 01 04 ; .....cmdQuit....
00001e90h: 00 C8 AE CO CE 00 04 98 0D F6 02 29 04 3B 01 11 ; .확인...??.;..
00001ea0h: 01 00 13 FF FF 03 56 00 00 00 03 07 00 6C 62 6C ; ... .V.....lbl
00001eb0h: 4D 61 69 6E 00 01 01 23 00 BB E7 BF EB B1 E2 C7 ; Main...#.사용기?
00001ec0h: D1 20 C1 A6 C7 D1 20 43 72 61 63 6B 20 4D 65 20 ; ?제한 Crack Me
00001ed0h: BF B9 C1 A6 20 CO D4 B4 CF B4 D9 2E 00 05 58 02 ; 예제 입니다...x.


```

도움말은 F1을 누르거나 위치: 1e70h, 7790, C0 | DOS | 수정: 2006-12-02 4:37:40오 | 선택된 바이트: 9 | 삽입

그림 18. 사용기한 수정(2006/12/31 → 3006/12/31)

마지며...

이번 장에서는 간단한 사용기한 체크 프로그램을 크랙하는 예제를 살펴보았다. 최근의 Visual Basic 프로그램들은 Native Code로 컴파일하기 때문에 이러한 크랙 예제는 많이 없을 것이다. 그러나 Native Code라고 해도 사용기한 체크 로직은 별반 다르지 않기 때문에 P-Code 컴파일 크랙 예제를 먼저 습득한다면 보다 쉽게 접근할 수 있다.

제 4 장 Windows 크랙 응용편

4.4 P-Code 크랙편(인증 크랙)

이번에는 사용자 ID 및 Password 나 시리얼 키를 요구하는 프로그램의 인증 크랙을 살펴보도록 하겠다.

인증 크랙

첨부파일 [VBCrackMe3.exe]를 직접 실행하면 [VB Crack Me3]의 윈도우가 표시되고, 사용자 이름과 패스워드 입력 폼이 나타나게 된다(그림 1).



그림 1. VB Crack Me3

이번 장에서는 WKTVBDE 의 P-Code 목록에서 표시할 수 없는 부분이 많기 때문에, Exdec 또는 P32Dasm 디컴파일 리스트를 참조해 가면서 분석해 나가야 한다.

먼저 [WKTVBDE]를 실행하여 메뉴의 [File]>[Open…]에서 [VBCrackMe3.exe]를 불러온 후 메뉴의 [Action]>[Run]에서 실행시킨다. 앞의 예제에서는 WKTVBDE 윈도우가 표시되었으나 이번 예제는 P-Code 리스트에 아무것도 나타나지 않는다(그림 2).

Visual Basic P-Code 크랙 4

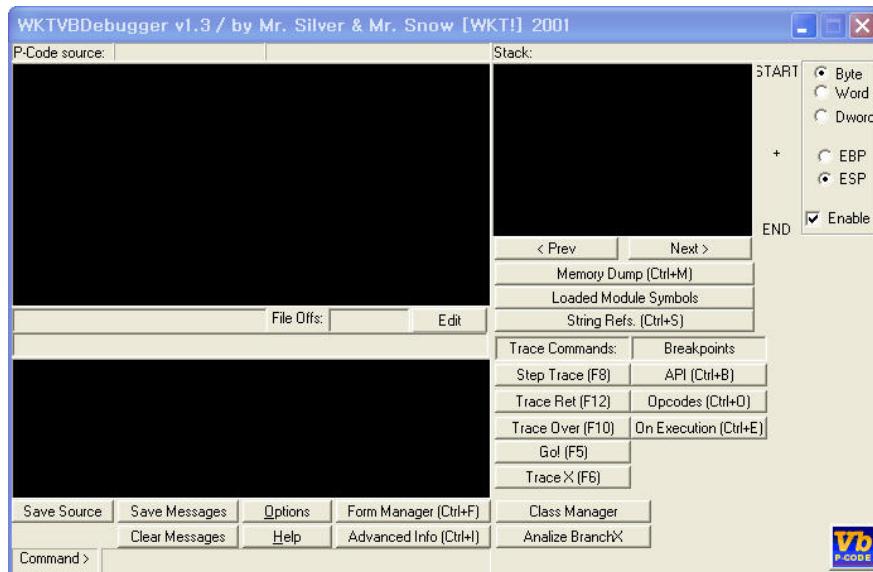


그림 2. 실행후의 WKTVBDE 윈도우

이것은 FormLoad 이벤트 처리가 없기 때문에 WKTVBDE 가 Break 하지 않고 실행된 상태이다. 아마도 “VB Crack Me #3” 등록화면이 바로 나타날 것이다. 그렇기 때문에 프로그램 시작시점에 브레이크 포인트를 설정하는 장소를 찾아야 한다. [Form Manager(Ctrl+F)] 버튼을 클릭하여 [Form Manager] 윈도우를 표시한다(그림 3).



그림 3. Form Manager 윈도우

위의 폼 리스트를 클릭하면 존재하는 폼이 표시된다(그림 4).

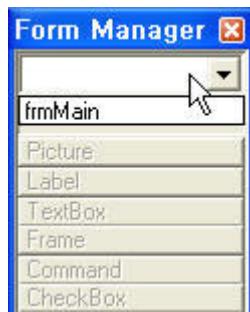


그림 4. 폼 리스트 표시

지금 실행중인 Crack Me 는 하나의 폼밖에 가지고 있지 않다는 것을 알 수 있으며, 표시되어 있는 [FrmMain]을 선택하면 아래와 같이 [Label][TextBox][Command] 3 개 버튼이 사용 가능하게 된다(그림 5).



그림 5. 폼 선택 후

이 버튼들은 폼 내에서 사용하고 있는 컨트롤을 나타내고 있다. [VB Crack Me #3] 윈도우의 [등록] 또는 [종료] 버튼은 [Command] 컨트롤이다. [Command] 버튼을 클릭하면, [cmdOK]와 [cmdQuit] 2 개를 선택할 수 있는 팝업 윈도우가 표시된다(그림 6).

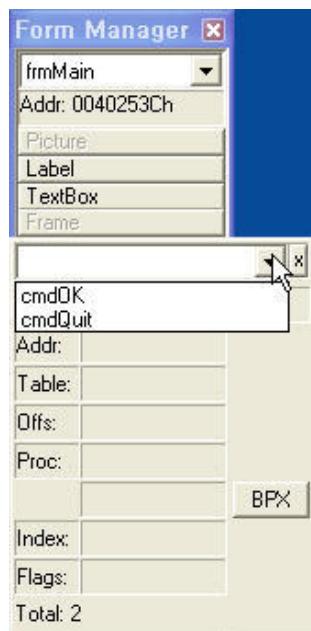


그림 6. Command 버튼 목록

[cmdOK]를 선택하면 여러 정보가 표시되고, [BPX] 버튼을 클릭하면 [등록] 버튼을 클릭했을 때의 처리 수행부분에 브레이크 포인트가 설정된다. Break Point 설정이 제대로 되었는지 확인하려면 “현재 브레이크 포인트 설정 메뉴(CTRL+E)”를 선택하여 00402E58에 녹색으로 설정되어 있는지 보면 된다(그림 7).

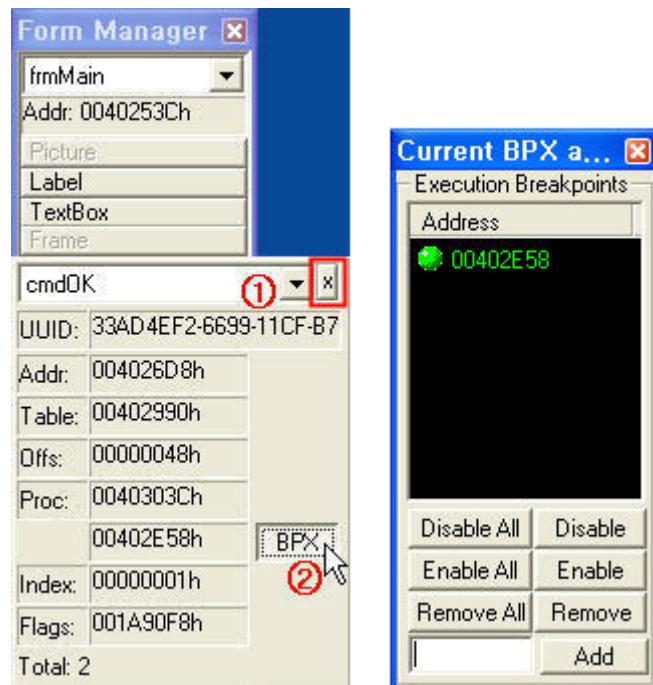


그림 7. BPX 설정 및 BPX 확인

[Go!(F5)] 버튼을 클릭하여, Name에 “nopter” Password에 “1234567890ABCDEF”을 입력하고 [등록] 버튼을 클릭하면, [00402E58: F4 LitI2_Byte:-> 0h 0]에서 프로그램이 멈추게 된다. 처음 키보드 입력시 WKTVBDE 가 브레이크 할 경우도 있는데, WKTVBDE 가 최초 실행 행에서 수행이 정지되는 경우라서 이런 경우 무시하고

Visual Basic P-Code 크랙 4

[Go!(F5)] 버튼을 클릭하여 브레이크 포인트까지 프로그램을 실행하도록 한다.

WKTVBDE 의 P-Code 리스트를 보면 [00402E61: 0F VCallAd frmMain.txtPass] 부근에서 입력한 Password 를 수집하고 있는 것 같다. 텍스트 박스 컨트롤로부터 문자열을 수집한 부근의 [00402E6F: 6C ILdRf 00000000h]에 브레이크 포인트를 설정하여 [Go!(F5)] 버튼을 클릭한다(그림 8).

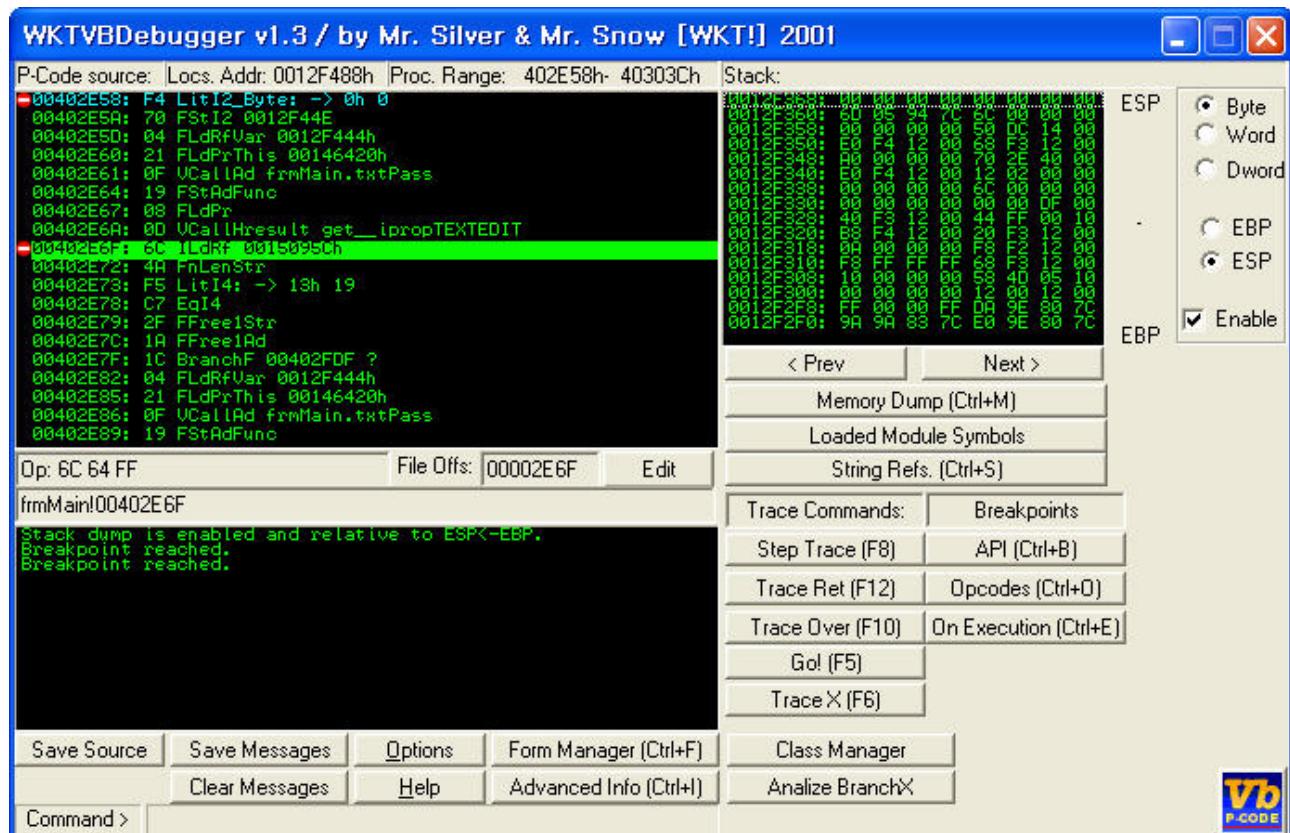


그림 8. 00402E6F에서 브레이크

조금 전까지 0(제로)이었던 어드레스 값이 0015095Ch 로 표시된다. 0x0015095C 메모리 내용을 [Memory viewer and editor] 윈도우에서 보면 아래와 같이 표시 된다(File Offs: 옆의 Edit 버튼 클릭). 이것은 입력한 패스워드 문자열이다.

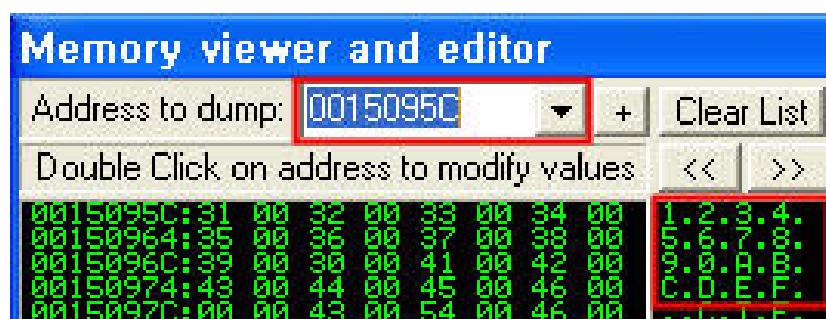


그림 9. 0015095Ch 메모리 덤프(패스워드 문자열)

다음 파일 Offset에서 실행되는 P-Code 부분은 Password 길이를 체크하고 있는 것 같다.

402E72: 4A FnLenStr	; Password 길이 취득
402E73: F5 LitI4: -> 13h 19	; 정수 19를 스택에 저장
402E78: C7 EqI4	; 정수 값(4 바이트)이 같은가?
402E79: 2F FFree1Str	
402E7C: 1A FFree1Ad	
402E7F: 1C BranchF: 00402FDF	; 비교 결과 False면 점프

디버거 코드를 보면 Password 전체 길이가 19 문자인 것을 알 수 있다. 비교결과 후 점프할 구문이 있는 0x00402E7F 까지 [Step Trace(F8)] 버튼을 클릭하여 진행한다.

[Memory viewer and editor] 윈도우를 사용하여 비교결과를 false에서 true로 변경하여 처리가 진행되도록 하거나 VBCrackMe3.exe 프로그램을 다시 실행⁷⁹하여 패스워드를 19 자로 입력하여 진행하면, [402E7F: 1C BranchF: 00402FDF]의 결과가 Jump에서 No Jump로 변한다.

OPCode	명령어
1C	BranchF(JNE)
1D	BranchT(JE)
1E	Branch(JMP)

주소	0x00402E7F
변경 전	1C 87 01
변경 후	1D 87 01

True 상태로 진입한 P-Code 밑 쪽의 텍스트 박스 컨트롤에서 Password 문자열을 취득하여 Left 함수로 5 문자를 걸러내고 있다.

402E8F: 0D VCallHresult get__ipropTEXTEDIT	
402E94: F5 LitI4: -> 5h 5	; Left 함수를 이용하여 왼쪽 5개 문자 추출
402E99: 6C ILdRf 0015FA8Ch	; Password 가 저장된 시작 주소(Step 진행시 나타남)
402E9C: 0B ImpAdCallII2 rtcLeftCharBstr on address 660E625Eh	
402EA1: 31 FStStr 0015CDACCh to 0012F478h	; 추출된 5개 문자 저장

위 부분을 [Step Trace(F8)] 버튼을 클릭하여 [402EA1: 31 FStStr 0015CDACCh to 0012F478h]⁸⁰까지 진행하면,

⁷⁹ 다시 실행한 경우 첫 번째 문자를 입력할 때 Break 되므로, [Go!(F5)]로 버튼을 클릭하여 계속 진행한다.

⁸⁰ VBCrackMe3.exe 를 재 시작 하였다면 주소가 변경될 수 있다.

Visual Basic P-Code 크랙 4

Left 함수의 문자열을 왼쪽에 표시되어 있는 0015CDACH에 저장한다는 것을 확인할 수 있다(그림 10). 즉, 입력된 패스워드 12345 를 0015CDACH 주소에 저장한다.



그림 10. 패스워드 첫 5 자리 저장(0015CDACH)

여기에서 저장된 주소는 뒤에서 계속 참조하기 때문에 기억하길 바란다. 이 P-Code 이후도 비슷하게 텍스트 박스 컨트롤에서 Password 문자열의 중간 여섯번째 문자의 첫 번째 문자를 추출하고 있다. 계속해서 [FStStr] 명령의 결과 저장 주소를 바탕으로 분석을 진행해 나가도록 하겠다.

```
402EB1: 0D VCallHresult get__ipropTEXTEDIT
402EB6: 28 LitVarI2 1h , 1           ; 문자 추출 길이(1 개)
402EBB: F5 LitI4: -> 6h 6          ; 여섯번째 문자 추출 위치
402EC0: 6C ILdRf 0015FA8Ch         ; Password 가 저장된 시작 주소(Step 진행시 나타남)
402EC3: 0B ImpAdCallI2 rtcMidCharBstr on address 660E64A6h
402EC8: 31 FStStr 00145AB4h to 0012F474h ; 추출된 6 번째 문자 저장
```

위 부분을 [Step Trace(F8)] 버튼을 클릭하여 [402EC8: 31 FStStr]까지 진행하면, 두 번째 추출된 문자열을 저장하는 주소가 표시된다.

[402EC8: 31 FStStr 00145AB4h to 0012F474h]와 같이 표시되기 때문에 왼쪽에 표시되고 있는 00145AB4h 주소를 메모한다. 그 다음 수행할 부분은 Text Box로부터 입력된 Password 중 13 번째 문자의 첫 번째 문자를 추출하고 있다.

여기에서도 결과가 저장되는 곳까지 [Step Trace(F8)] 버튼을 실행시켜 표시된 주소를 메모한다.

```
402ED8: 0D VCallHresult get__ipropTEXTEDIT
402EDD: 28 LitVarI2 1h , 1           ; 문자 추출 길이(1 개)
402EE2: F5 LitI4: -> Dh 13          ; 13 번째 문자 추출 위치
402EE7: 6C ILdRf 0015FA8Ch         ; Password 가 저장된 시작 주소(Step 진행시 나타남)
402EEA: 0B ImpAdCallI2 rtcMidCharBstr on address 660E64A6h
402EEF: 31 FStStr 0015CD84h to 0012F470h ; 추출된 13 번째 문자 저장
```

Visual Basic P-Code 크랙 4

그 다음 수행할 부분은 Text Box로부터 입력된 Password 중 7 번째 문자에서 6 개 문자를 추출하는 과정으로 이어진다. 여기에서도 결과가 저장되는 곳까지 [Step Trace(F8)] 버튼으로 실행시켜 표시된 주소를 메모한다.

```
402EFF: 0D VCallHRESULT get__ipropTEXTEDIT
402F04: 28 LitVarI2 6h , 6 ; 문자 추출 길이(6 개)
402F09: F5 LitI4: -> 7h 7 ; 7 번째 문자 추출 위치
402F0E: 6C ILdRf 0016423Ch ; Password 가 저장된 시작 주소(Step 진행시 나타남)
402F11: 0B ImpAdCallI2 rtcMidCharBstr on address 660E64A6h
402F16: 31 FStStr 00160284h to 0012F46Ch ; 추출된 6 개 문자 저장
```

최종적으로 Text Box로부터 입력된 Password 중 14 번째 문자에서 6 개 문자를 추출하는 과정으로 이어진다. 반복된 작업으로 지루할지 모르지만, 결과적으로 추출된 문자를 저장하는 장소까지 [Step Trace(F8)] 버튼으로 실행시킨 후 표시된 주소를 메모하면 된다.

```
402F26: 0D VCallHRESULT get__ipropTEXTEDIT
402F2B: 28 LitVarI2 6h , 6 ; 문자 추출 길이(6 개)
402F30: F5 LitI4: -> Eh 14 ; 14 번째 문자 추출 위치
402F35: 6C ILdRf 0015D634h ; Password 가 저장된 시작 주소(Step 진행시 나타남)
402F38: 0B ImpAdCallI2 rtcMidCharBstr on address 660E64A6h
402F3D: 31 FStStr 001618CCh to 0012F468h ; 추출된 6 개 문자 저장
```

이것으로 Password 문자열 추출이 모두 끝났다. 여기에서 저장된 문자와 주소를 정리하면 아래와 같다. 정리된 주소 값은 스택 주소이므로 동작환경에 따라, 매번 실행할 때마다 변경된다.

```
0015CDACh ; 패스워드 첫 5 문자(12345)
00145AB4h ; 패스워드 6 번째 문자의 1 문자(6)
0015CD84h ; 패스워드 13 번째 문자의 1 문자(C)
00160284h ; 패스워드 7 번째 문자의 6 개 문자(789AB)
001618CCh ; 패스워드 14 번째 문자의 6 개 문자(DEF)
```

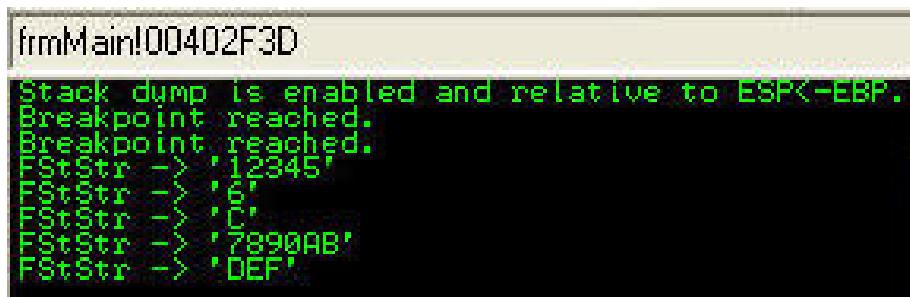


그림 11. 최종 저장된 FStStr

입력된 패스워드 추출과정이 끝나면 [ThisVCallHresult 00402D18->00402CF8]의 콜 명령이 이어진다. [402F4A:10 ThisVCallHresult 00402D18->00402CF8]까지 [Step Trace(F8)] 버튼으로 실행한다. 이 때 스택의 처음 4바이트 주소 값을 [Memory viewer and editor] 윈도우에서 참고하면, 아래와 같이 Password 처음 5문자가 저장된 주소가 들어 있다(그림 12).

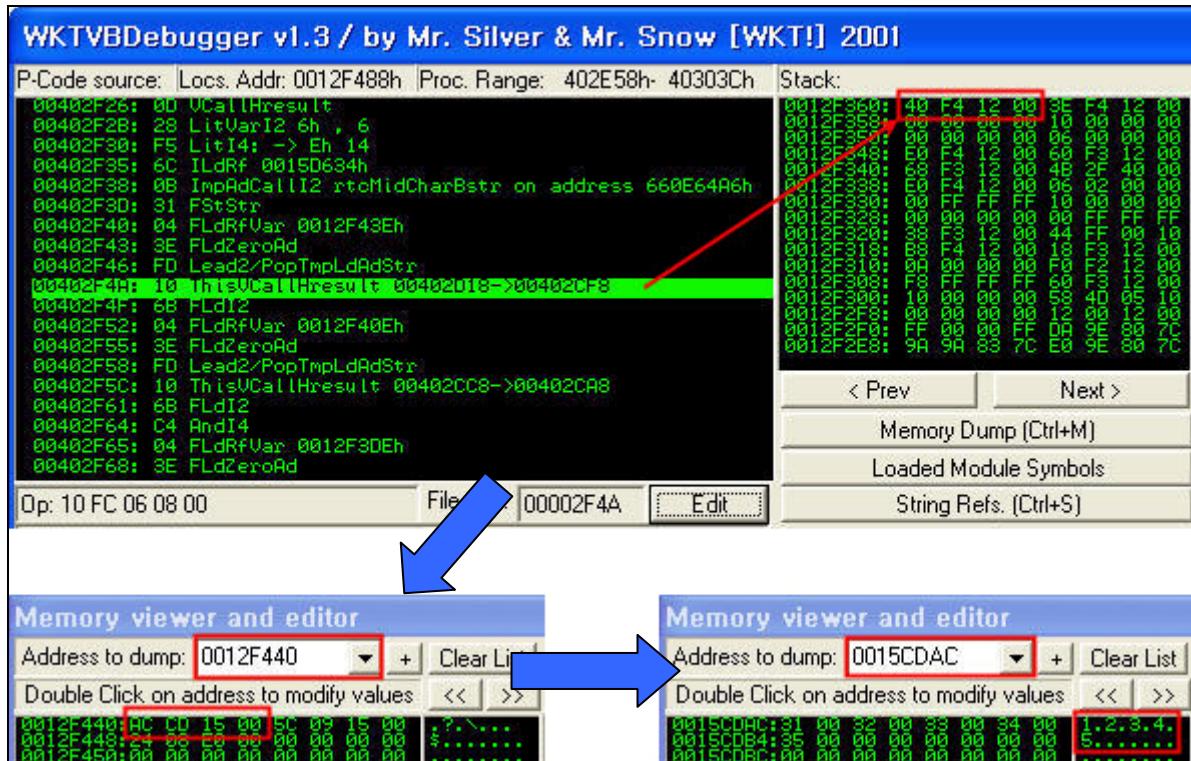


그림 12. 저장된 패스워드 주소

[Step Trace(F8)] 버튼을 클릭하면 콜 부분으로 들어간다.

402CF8 P-Code 부분(Call Function)	
402CF8: 80 ILdI4	
402CFB: 1B LitStr: VbCrk	; 문자정수
402CFE: 30 EqStr	; 문자열 비교
402D00: 1C BranchF 00402D0B	
402D03: F4 LitI2_Byte: -> FFh 255	; 문자열 일치 리턴 값
402D05: 70 FStI2	
402D08: 1E Branch: 402d10	
402D0B: F4 LitI2_Byte: -> 0h 0	; 문자열 불일치 리턴 값
402D0D: 70 FStI2	

402D10: 2F ExitProcCbHResult

위 부분은 불러온 인자값(12345)과 문자 정수 ‘VbCrk’를 비교하는 함수이다. [402D00: 1C BranchF: 402D0B]까지 [Step Trace(F8)] 버튼으로 실행시킨 후, 문자열을 무조건 참으로 만들기 위해 스택의 첫 4 바이트를 0(제로)에서 “FFFFFFFF”로 변경한다(Memory viewer and editor 를 이용하여 스택주소 0012F2C8h 값을 변경하면 됨). 그 뒤는 [Step Trace(F8)] 버튼을 클릭하여 콜에서 리턴 직후 [00402F4F: 6B FLdI2 FFFFFFFFh]까지 실행한다.

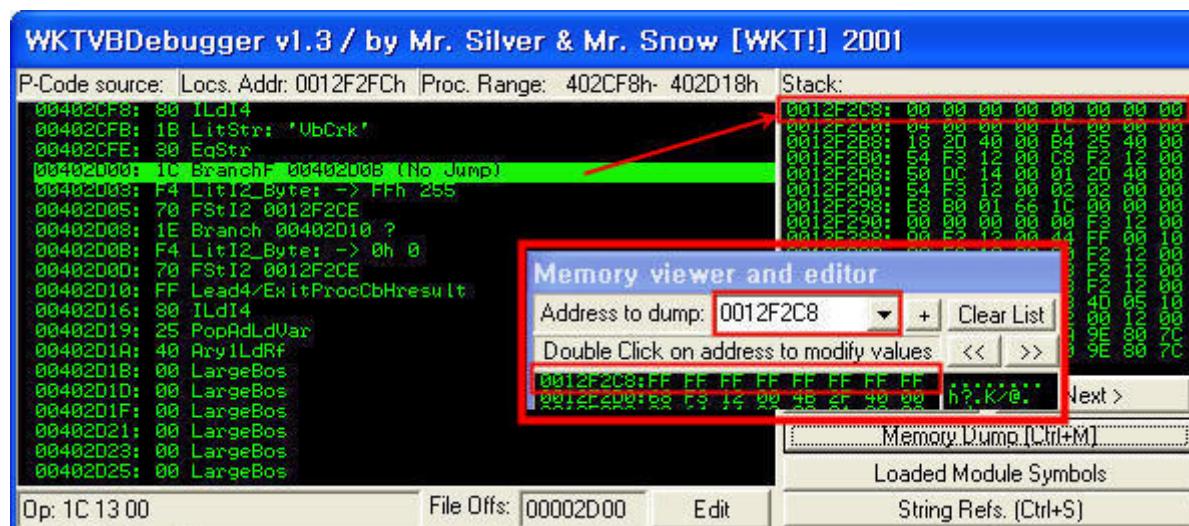


그림 13. 조건식 결과값 무조건 True로 변경

402F4F P-Code 부분	
402F4F: 6B FLdI2 FFFFFFFFh	; Call 리턴 값
402F52: 04 FLdRfVar 0012F40Eh	
402F55: 3E FLdZeroAd	
402F58: FD Lead2/PopTmpLdAdStr	; 여기에서 리턴 값 처리
402F5C: 10 ThisVCallHresult 00402CC8->00402CA8	

동일하게 [402F5C: 10 ThisVCallHresult 00402CC8->00402CA8]까지 실행시켜 스택의 처음 4 바이트 주소값을 참조 하자.

402F5C P-Code 부분의 스택 상태
0012F35C: 10 F4 12 00 0E F4 12 00

Password 6 번째의 1 개 문자를 저장한 주소가 들어 있다. [Step Trace(F8)] 버튼을 클릭하여 콜 함수 처리를 보자.

402CA8 P-Code 부분(Call Function)

Visual Basic P-Code 크랙 4

```
402CA8: 80 ILdI4 -> 145ab4h 1333940
402CAB: 1B LitStr: ‘-’ ; 문자정수
402CAE: 30 EqStr ; 문자열 비교
402CB0: 1C BranchF: 00402CBB
402CB3: F4 LitI2_Byte: -> FFh 255 ; 문자열 일치 리턴 값
402CB5: 70 FStI2 0012F2CA
402CB8: 1E Branch: 00402CC0
402CBB: F4 LitI2_Byte: -> 0h 0 ; 문자열 불일치 리턴 값
402CBD: 70 FStI2 0012F2CA
402CC0: FF Lead4/ExitProcCbHresult
```

좀 전의 Password 처음 5 문자를 체크하고 있는 처리와 비교하는 문자열 정수 값만 다르고 나머지는 같다.

[402CB0: 1C BranchF: 00402CBB]까지 [Step Trace(F8)]버튼으로 실행시킨 후 스택의 첫 4 바이트를 0(제로)에서 “FFFFFF”로 변경하고, 콜의 리턴 직후 [00402F01: 6B FLdI2 FFFFFFFF]까지 실행한다.

402F61 P-Code 부분

```
402F61: 6B FLdI2 FFFFFFFFh
402F64: C4 AndI4
402F65: 04 FLdRfVar 0012F3DEh
402F68: 3E FLdZeroAd
402F6B: FD Lead2/PopTmpLdAdStr
402F6F: 10 ThisVCallHresult 00402CC8->00402CA8
```

리턴값을 AND 연산하면 바로 콜 함수가 나타난다. 그러나 이번 콜 주소(00402CA8)는 Password 6 번째 문자의 1 문자일 때도 Call Function 을 수행하게 된다. 문자정수 ‘-(하이픈)’과 비교처리를 수행하고 있다. 단지 비교대상의 인수를 확인하는 것이다. [402F6F: 10 ThisVCallHresult 00402CC8->00402CA8]까지 실행시킨 후 스택의 첫 4 바이트 주소 값이 나타내는 부분을 참조해 보자.

402F6F P-Code 부분의 스택 상태

```
0012F35C: E0 F3 12 00 DE F3 12 00
```

Password 13 번째 문자의 1 개 문자 저장 주소가 들어 있다. 이번은 [TraceOver(F10)] 버튼을 클릭하여 콜 장소의 처리를 건너뛴다. 이 때 [00402F74: 6B FLd2 FFFF0000h]리턴 값은 FFFF0000h 로 표시되어 있다. 이것은 함수 리턴 값을 변경하지 않았기 때문에 False 가 반환 된 것이다. [StepTrace(F8)] 버튼을 한번 클릭하여 [00402F77: AndI4]로 연산할 때 스택 첫 4 바이트를 [00 00 FF FF]에서 [FF FF FF FF]로 변경한다. 그러면 WKVBDE 의 P-Code 리스트 표시도 바뀐다.

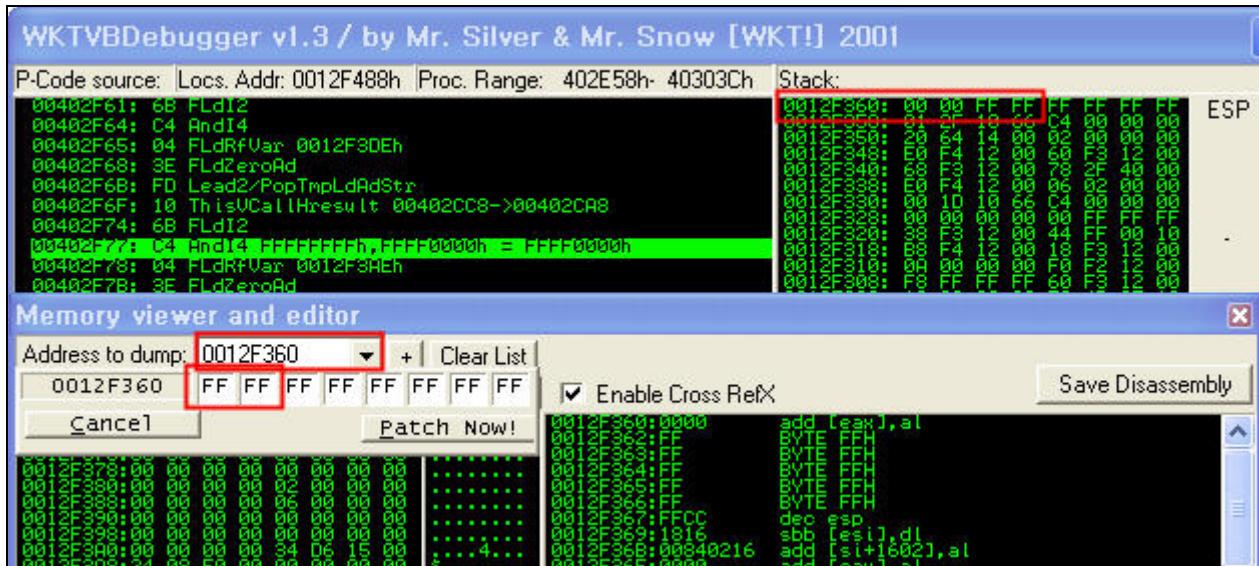


그림 14. 리턴 값 True로 Patch

이것으로 리턴값의 연산결과는 전부 True가 된다. 아직 리스트에는 Call Function 부분이 표시되어 있다. 계속해서 [00402F82: 10 ThisVCallHresult 00402C78->00402C58]까지 실행시켜 스택의 첫 4 바이트 주소값이 나타내는 영역을 표시한다.

402F82 P-Code 부분의 스택 상태

0012F35C: B0 F3 12 00 AE F3 12 00

Password 7 번째 6 개 문자 저장 주소가 들어 있다. [StepTrace(F8)]버튼을 클릭하여 콜 함수로 들어가자.

402C58 P-Code 부분(Call Function)

402C58: 80 ILdI4 -> 160284h 1442436	
402C5B: 1B LitStr: ‘2ntpri’	; 문자정수
402C5E: 30 EqStr	; 문자열 비교
402C60: 1C BranchF 00402C6B	
402C63: F4 LitI2_Byte: -> FFh 255	; 문자열 일치 리턴 값
402C65: 70 FStI2 0012F2CA	
402C68: 1E Branch 00402c70	
402C6B: F4 LitI2_Byte: -> 0h 0	; 문자열 불일치 리턴 값
402C6D: 70 FStI2 0012F2CA	
402C70: 2F ExitProcCbHresult	

00402C58h 부분도 저장된 인수와 문자정수 “2ntpri”를 비교하는 함수이다. [402C60: 1C BranchF 00402C6B]까지

Visual Basic P-Code 크랙 4

[Step Trace(F8)] 버튼으로 실행시킨 후 스택의 첫 4 바이트를 0(제로)에서 "FFFFFF"로 변경한다. 변경을 완료하고 다시 메인으로 리턴되면, 마지막 콜 함수인 [00402F95: 10 ThisVCallHresult 00402C28->00402C08]까지 실행한다. 여기에서도 스택의 첫 4 바이트 주소값이 나타내는 영역을 참조하자.

402F95 P-Code 부분의 스택 상태
0012F35C: 80 F3 12 00 7E F3 12 00

Password 14 번째 문자의 6 개 문자 저장 주소가 들어 있다. [Step Trace(F8)] 버튼을 클릭하여 콜 함수로 들어간다.

402C58 P-Code 부분(Call Function)
402C08: 80 ILdl4 -> 1618cch 1448140 402C0B: 1B LitStr: 'Nopter' ; 문자정수 402C0E: 30 EqStr ; 문자열 비교 402C10: 1C BranchF 00402C1B 402C13: F4 LitI2_Byte: -> FFh 255 ; 문자열 일치 리턴 값 402C15: 70 FStl2 402C18: 1E Branch 00402c20 402C1B: F4 LitI2_Byte: -> 0h 0 ; 문자열 불일치 리턴 값 402C1D: 70 FStl2 0012F2CA 402C20: FF Lead4/ExitProcCbHresult

위의 부분도 저장된 인수와 문자정수 'Nopter'를 비교하는 함수이다. [402C18: 1e Branch 00402c20]까지 [Step Trace(F8)] 버튼으로 실행시킨 후 스택의 첫 4 바이트를 0(제로)에서 "FFFFFF"로 변경한다. 변경을 완료하고 메인 함수로 리턴하여 [00402FE2: 1C BranchF 00403011 (No Jump)]까지 실행시키면, [등록해 주셔서 감사합니다.] 문자열이 보인다. 마지막으로 정리하면 Password 는 [VbCrk-2ntpri-Nopter]가 된다.

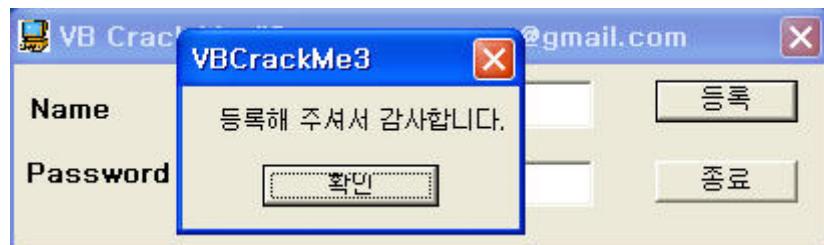
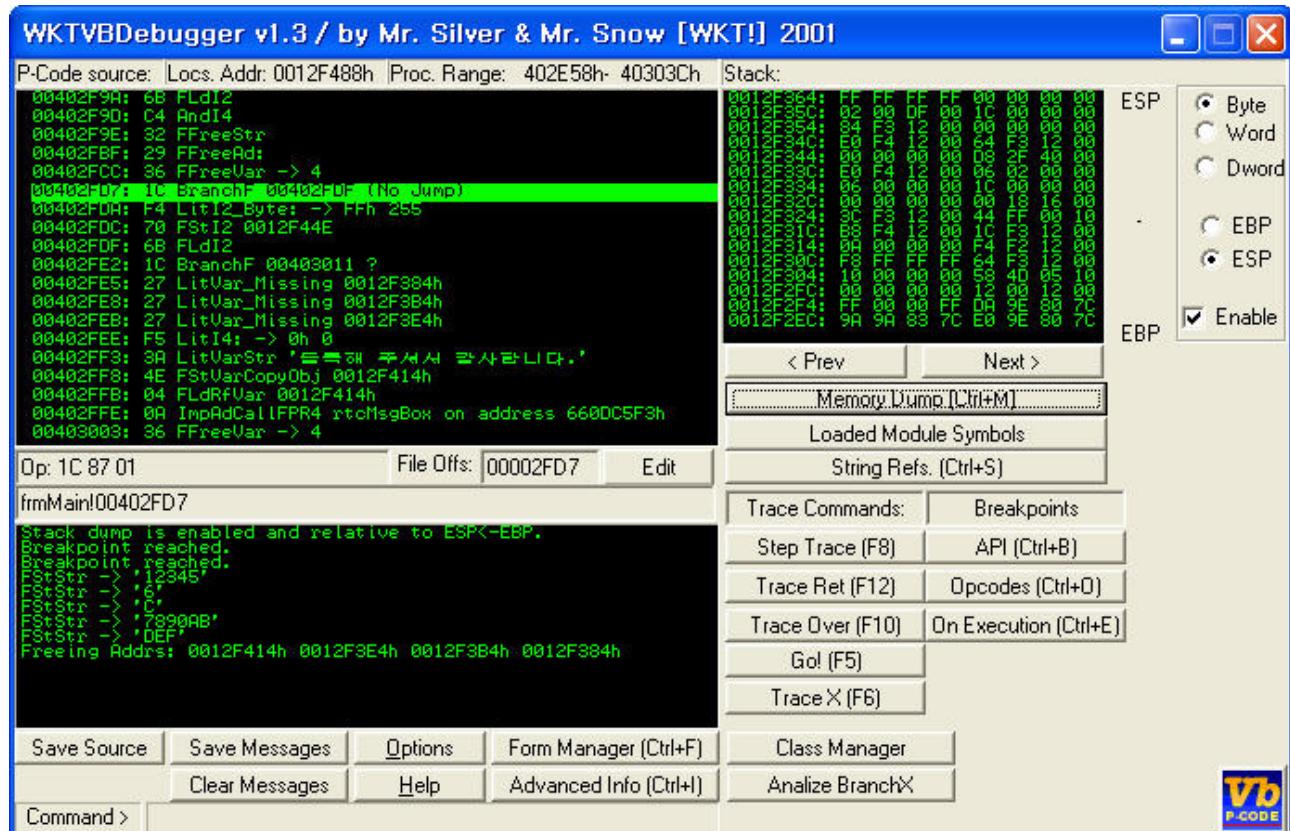


그림 15. 등록 완료 메시지

다른 방법은?

위의 디버깅 과정을 거치지 않고 손쉽게 시리얼 등록번호를 알아내는 방법이다. 윈도우 PE 파일이나 P-Code로 컴파일된 실행파일은 일반적으로 데이터 영역(.data)에 전역변수를 미리 담고 있다. 그러므로 바이너리 편집기나 disassembler 툴의 문자열 검색을 통해 등록번호를 쉽게 유추할 수 있다.

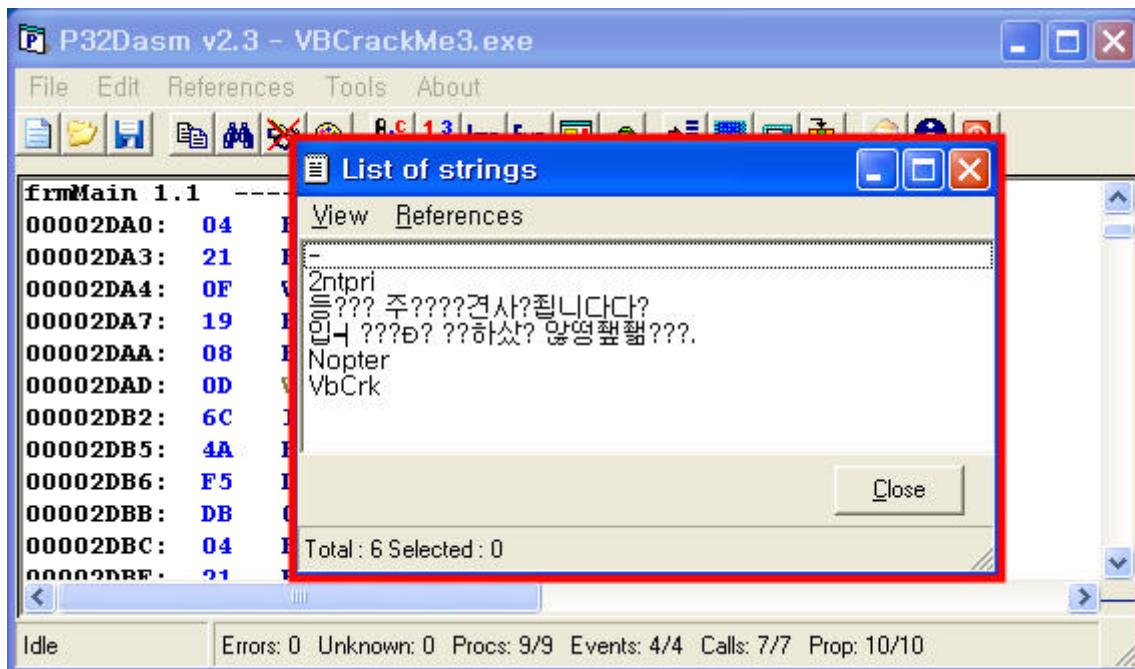


그림 16. 문자열 목록

마지며...

이번 장에서는 간단한 시리얼 체크 프로그램을 크랙하는 예제를 살펴보았다. 최근 프로그램들은 이렇게 바이너리 파일내에 단순하게 문자열을 비교하여 등록번호를 체크하는 예는 많지 않다. 그러나 시리얼 체크 로직은 별반 다르지 않기 때문에 이번 장에서 습득한 내용을 이해하였다면 보다 쉽게 접근할 수 있을 것이다.

Visual Basic.NET은 Visual Basic 6.0 소스를 단순히 포팅하기에는 어렵기 때문에 Visual Basic이라는 이름만 같고 전부 다른 것이라고 주장하는 사람도 있다. Object 부터 MSIL(Microsoft Intermediate Language)이라는 중간 코드(.NET 오브젝트 형식)로 출력되어, Native Code/P-Code와는 다르다. 또 Visual Basic 6.0 지원도 2005년도에 종료할 예정이었지만, 개발자들의 항의에 의해 3년 연장되었다. 이젠 P-Code로 컴파일된 소프트웨어를 찾기 어려울 것 같다.

제 4 장 Windows 크랙 응용편

4.5 JAVA 크랙편

들어가며

프로그래밍 언어중의 하나인 Java 는 등장할 당시 “Network, Dynamic, Run Anywhere”라는 키워드로 주목 받았다. 특정 플랫폼에 국한되지 않고, Web 브라우저상에서 애니메이션(Java-Applet) 등을 표현할 수 있는 것들이 처음에는 높게 평가되었다. 그러나 조금 지나지 않아 개발과 애니메이션 표현에 더 뛰어난 Flash 에 의해 시장을 많이 빼앗겨 버렸다. 또한, 로컬 PC 에서 실행되는 Java 어플리케이션은 가상 머신상에서 동작하는 특성 때문에 속도에서 뒤떨어질 수 밖에 없었고, 다양한 플랫폼에서 발생되는 문제, Java 프로그램 배포 등 예기치 못한 것들 때문에 기대만큼 성장을 보이지는 못했다.

그대신 SKT, KTF, LGT등을 대표로 하는 휴대단말기상의 어플리케이션이나, 서버 사이드 어플리케이션 분야에서는 큰 신장세를 보이고 있다. 또한 기업의 통합 업무 시스템 개발 분야에서도 Java는 인기 있으며⁸¹, 쉽게 배우고 응용할 수 있다는 장점도 있다.

이번 장에서는 Java 프로그램의 크랙 방법에 대해 알아 보도록 하자. Java 의 특징을 이해 한다면 일반적인 소프트웨어 크랙과는 접근 방법이 다르다는 것을 알 수 있다.

JAVA 이해

Java 프로그램은 [WORA: Write Once, Run Anywhere]⁸²를 이념으로 하고 있으며, 기본적으로 OS나 CPU에 의존하지 않고 동작한다(엄밀히 말하면 OS나 CPU에 의존적인 명령도 있다). Java 언어로 작성된 소스 코드를 컴파일하면, Class라 불리는 바이너리 파일이 만들어 진다. 이것은 Java 가상 머신(Java VM: Java Virtual Machine) 소프트웨어에서 동작하는 “바이트코드”다.

그러나 컴퓨터가 이해할 수 있는 언어는 기계어(Native Code)뿐이기 때문에, Java 프로그램을 실행할 때에는 Java 가상 머신이 플랫폼에 맞춰진 Native Code 로 변환하는 작업을 수행하고 나서 실행된다. 이것은(바이트코드) 각 플랫폼에 맞춘(Native Code 로 변환할 수 있는) Java 가상 머신만 있으면 플랫폼이 어떤 것이든 같은 바이트코드를 실행 할 수 있다는 것이다.

이러한 Java 특성 때문에 일반적인 소프트웨어 크랙과 차이가 나게 된다. Debugger, Disassembler라고 하는 툴은 CPU나 실행 파일 형식에 의존한다. 우리들이 잘 사용하는 디버거, 예를 들어 OllyDbg나 SoftICE 등은 x86 계열의 기계어를 해석할 수 있지만, Java 가상 머신상의 바이트코드를 전혀 해석할 수 없기 때문에 이런 툴은 거의 사용할

⁸¹ Visual 한 환경을 웹 브라우저만을 통해 제공하기 때문에 손쉽게 배포, 이용 가능하다는 큰 장점이 있다.

⁸² 코드를 한번 작성하면 어디에서도 실행된다

수가 없다⁸³.

이러한 문제에 대한 최선의 방법은 바이트 코드를 해석할 수 있는 툴을 사용하는 것이다. 소프트웨어 크랙은 정적 분석보다도 동적 분석이 더 효과적이라고 말할 수 있지만, OllyDbg 나 SoftICE 정도의 기능을 가진 바이트코드 분석용 디버거는 존재하지 않는 것 같다. 그러나 정적 분석 툴의 하나인 디컴파일러로는 실용적으로 쓸만한 것이 존재한다. 따라서 Java 프로그램의 크랙은 “디컴파일러로 복원한 소스코드를 분석” 하는 것으로 수행하기 때문에 크랙 특유의 기술, 지식은 거의 필요로 하지 않고 순수하게 Java 프로그래밍 지식이 요구된다.

DJ JAVA Decompiler

쓸만한 Java 디컴파일러는 여러 개 존재하지만, 여기에서는 디컴파일러 [jad]⁸⁴ 의 FrontEnd 툴인 [DJ Java Decompiler]⁸⁵를 사용하여 분석을 수행할 것이다. 그렇다고 해서 이 문서에서 Java 소스 코드 읽는 법(혹은 Java 프로그래밍 기술)에 대하여 설명하는 것은 너무 광범위한 주제이기 때문에, Java 프로그래밍의 기초적인 지식을 가지고 있다는 것을 전제로 설명해 나갈 것이다.

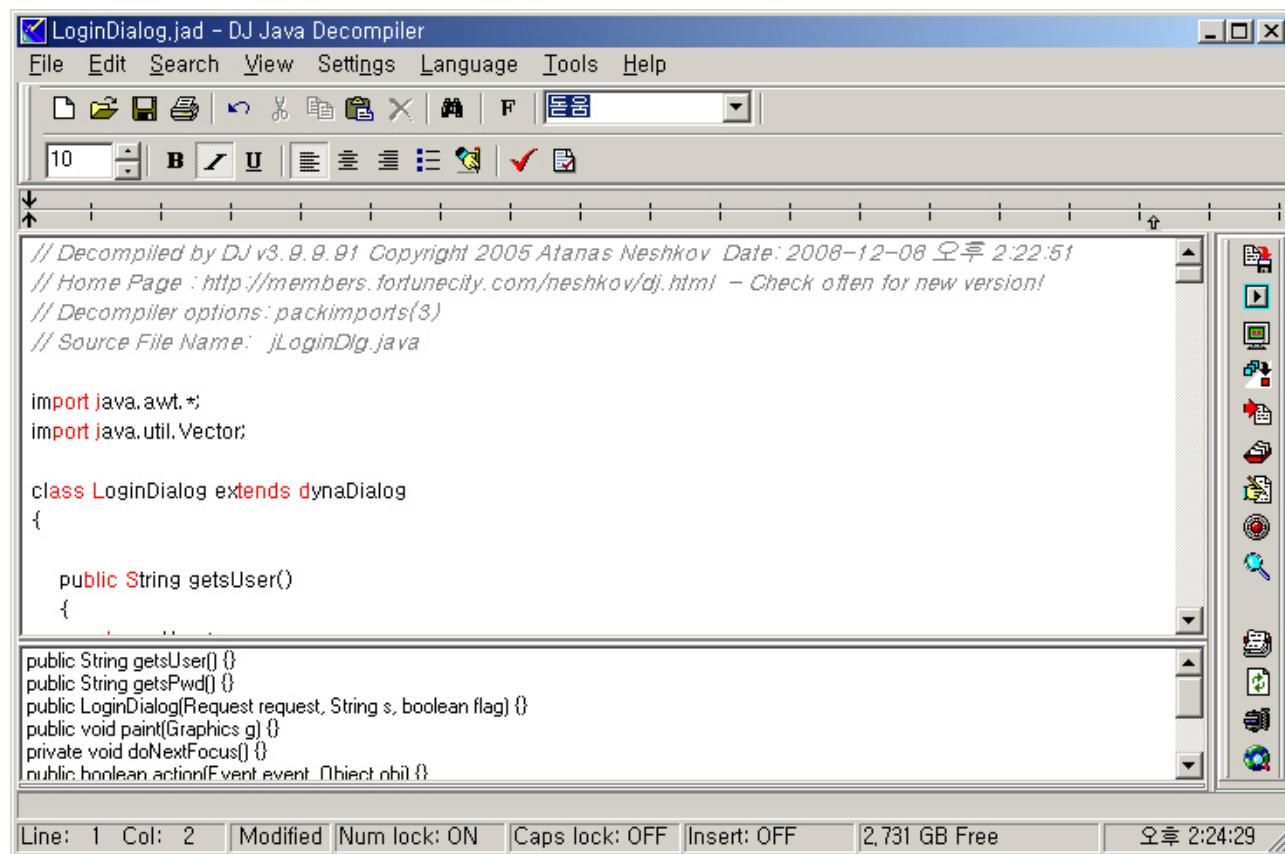


그림 1. DJ Java Decompiler 메인 화면

⁸³ 반드시 Java에 한정되지 않고, 중간 코드로 존재하는 형태는 OllyDbg나 SoftICE로 분석이 매우 힘들다.

⁸⁴ Jad(the fast JAvA Decompiler): <http://www.kpdus.com/jad.html>

⁸⁵ DJ: <http://members.fortunecity.com/neshkov/dj.html>

DJ JAVA Decompiler 설치

앞서 말했듯이 Java 디컴파일러 중에서도 잘 알려진 “DJ Java Decompiler”(이하 DJ로 생략)을 사용하여 설명을 할 것이다. Java로 작성한 간단한 JavaCrackMe⁸⁶를 목표로 하여 실제 디컴파일 해보도록 하겠다.

“DJ Java Decompiler”를 “<http://members.fortunecity.com/neshkov/dj.html>”에서 다운받아 설치한다. 설치과정이 완료된 후, DJ를 실행하면 문서편집기 같은 윈도우가 표시된다. 바로 Java Class 파일을 읽어 들어 들이고 싶지만 디플트 환경으로 분석시 불편한 사항이 있으므로 몇몇 설정 부분을 변경 해야 한다.

폰트 변경

먼저 화면에 표시되는 폰트를 변경한다. 기본 폰트는 한국어 표시가 안 되는 [Arial]이 선택되어 있기 때문에, 프로그램 코드 표시에는 적당하지 않다. 메뉴에서 [Setting]>[Configuration…]을 선택하면 환경 설정 윈도우가 열리므로(그림 2), [Appearance and Colors] 탭을 선택하여 [Change Default Font Setting] 버튼을 눌러 폰트를 변경 하도록 한다. 본인은 폰트는 MS 고딕, 사이즈 10으로 설정하였다.

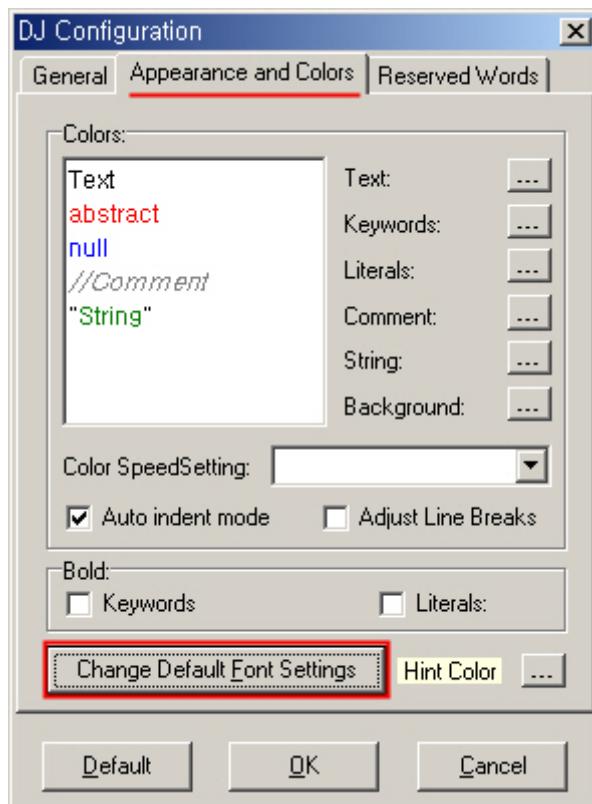


그림 2. 기본 폰트 변경

⁸⁶ CrackMe 를 실행시키려면, J2SE SDK(또는 JRE) 버전 1.4.2 이상이 설치되어 있어야 한다.

디컴파일 설정

메뉴에서 [Setting]>[Decompiler Setting]을 선택하여 디컴파일 설정 윈도우가 열리면 [Convert Unicode strings into ANSI strings]⁸⁷, [Output original line numbers as comments]에 체크 하도록 한다 이 체크를 통해 나중에 바이너리 패치 작성 단계에서 유용하게 사용되는 [원본 소스 코드 행 번호]⁸⁸가 출력될 수 있다.

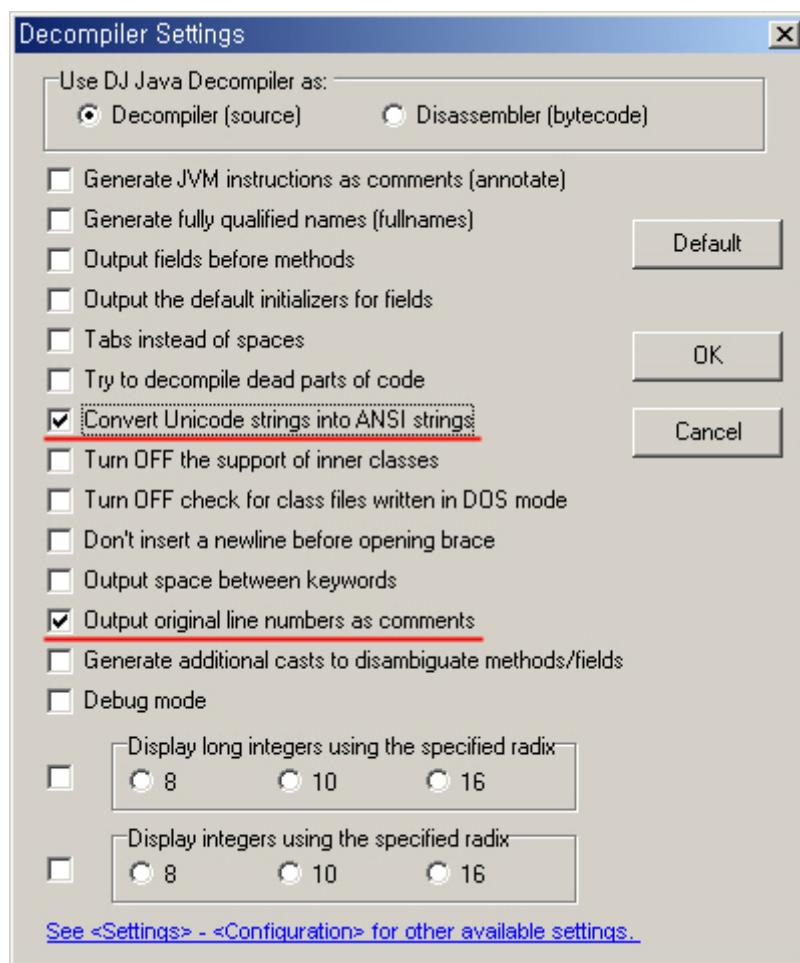


그림 3. Decompiler 설정

디컴파일을 통한 소스 복원

JavaCrackMe 를 실행하면 아래 그림과 같이 사용자 계정 및 시리얼 키 값을 입력하는 형태이다. 어떠한 형태인지 확인되었다면 실제 크랙에 도전해 보도록 하자.

⁸⁷ 체크를 하지 않으면, 문자열이 “\u30A1\u30A2\u30A3\u30A4\u30A5”와 같이 출력된다.

⁸⁸ 이 행 번호는 Java 에러 로그에서 사용되는 것이다.

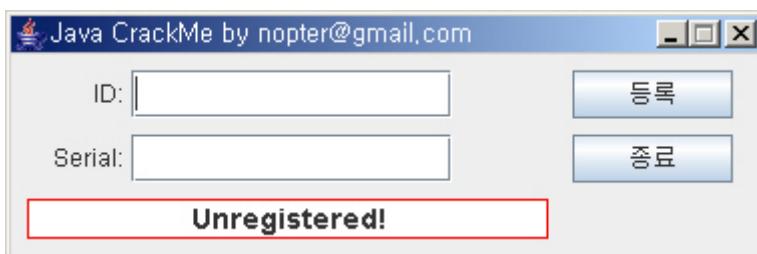
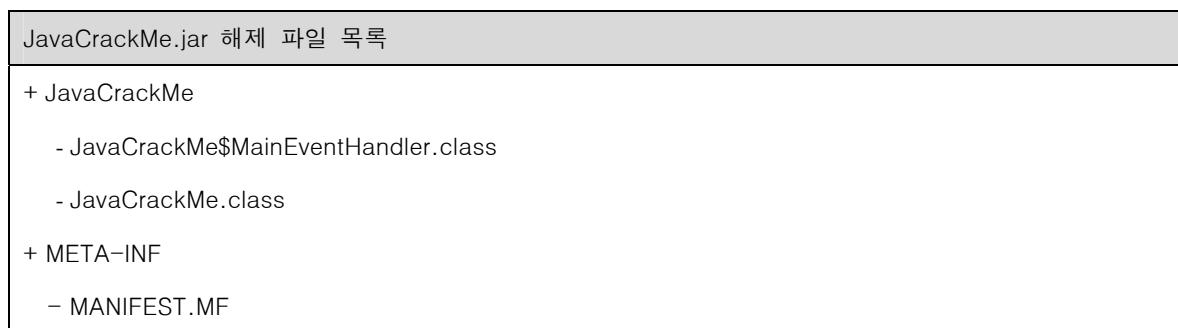


그림 4. Java Crack Me 실행 화면

Java 프로그램은 실행에 필요한 Class 파일을 JAR(JavaArchiver)라고 불리는 압축 포맷 형태로 제공한다. JavaCrackme.jar 도 예외는 아니기 때문에 먼저 압축을 해제한다. J2SE SDK(또는 JRE)가 설치되어 있으면 jar 커맨드로 해제할 수 있지만, jar는 실제 ZIP 포맷이기 때문에 확장자를 jar에서 zip으로 바꾸면 일반적인 압축 해제 프로그램에서도 해제할 수 있다.

JavaCrackMe.jar를 압축해제 하면 아래와 같은 파일들을 볼 수 있다.



DJ 가 정상적으로 설치되어 있다면 “.class” 확장자가 DJ 에 링크되어있기 때문에 더블클릭으로 바로 디컴파일을 수행할 수 있다. 또한, 읽어 들일 대상 Class 파일이 큰 경우는 특정문자열의 강조 표시를 수행할지 물어오는데(그림 5) 강조 표시 기능에 버그가 존재하기 때문에, 다른 텍스트 에디터 등으로 다시 읽을 것을 권장한다.

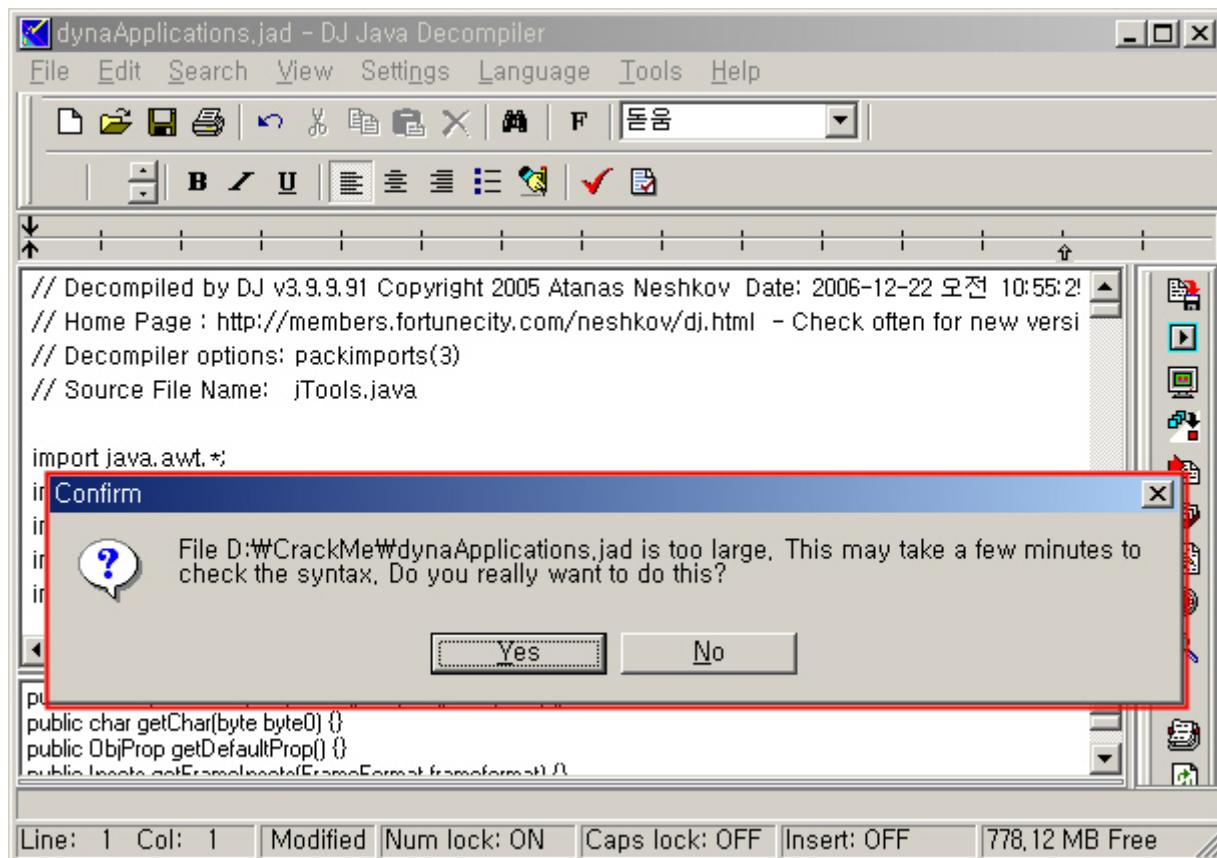


그림 5. 특정 문자열 강조 표시처리 확인

이번 JavaCrackMe는 2개의 CLASS 파일로 구성되어 있다. 단지 2번 더블 클릭 하는 것만으로 전체 소스 코드를 얻을 수 있지만, 실제 어플리케이션에서는 CLASS 수가 100 개를 넘는 경우도 있다.

복수의 CLASS파일을 일괄 디컴파일하려면, 메뉴에서 [Tools]>[Decompile more files]를 선택한다(그림 6). [Select File:]에는 디컴파일 대상이 되는 CLASS파일을 ⁸⁹, [Destination directory:]에는 복원 코드를 저장할 폴더를 지정해 준다. 화면 하단부에 체크박스 형태로 디컴파일 옵션을 설정 할 수 있기 때문에, [Convert Unicode strings into ANSI strings]와 [Output original line numbers as comments]에 체크 하여 [Decompile] 버튼을 누르면 디컴파일이 실행되어, 확장자 jar의 소스 코드가 생성된다.

⁸⁹ [서브디렉토리 이하 포함]과 같이 디렉토리 단위 지정이 안되기 때문에, CLASS 파일이 다수의 디렉토리로 나누어져 있는 경우는 디렉토리마다 파일을 지정해야 한다.

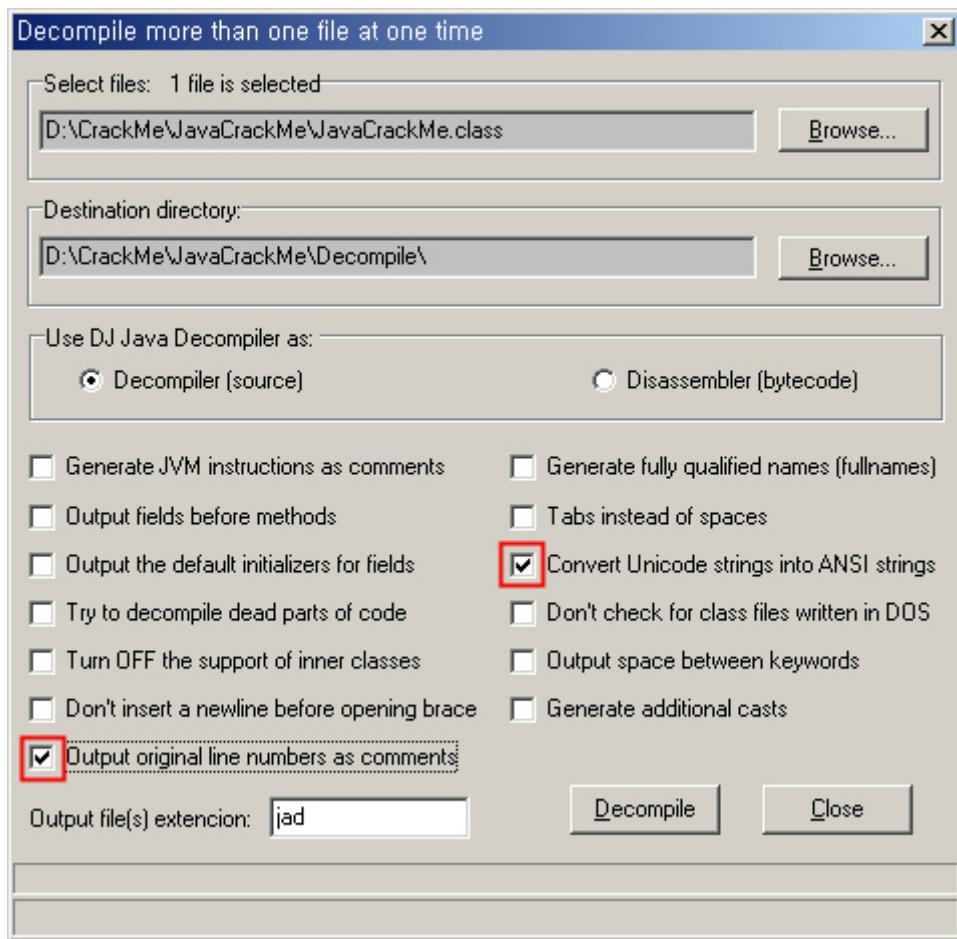


그림 6. Decompile more files 원도우

JAVA 소스 분석

소스 코드가 복원되면, 소스 코드를 읽어서 체크 루틴을 찾기만 하면 된다. 이미 이야기 했지만, 여기서부터 작업은 Java 코드를 분석하는 것이기 때문에 Java 프로그래밍 지식이 요구된다. 복원된 소스코드에 나타난 몇몇 문자열로 힌트를 짐작할 수는 있겠지만, 소스 코드를 읽을 수 없다면 한계에 다다를 것이다.

그러나 Java 프로그래밍은 본 문서의 취지에서 벗어나기 때문에 상세한 해설은 하지 않고, 체크 루틴을 호출하고 있는 부분만을 분석해 보도록 하겠다.

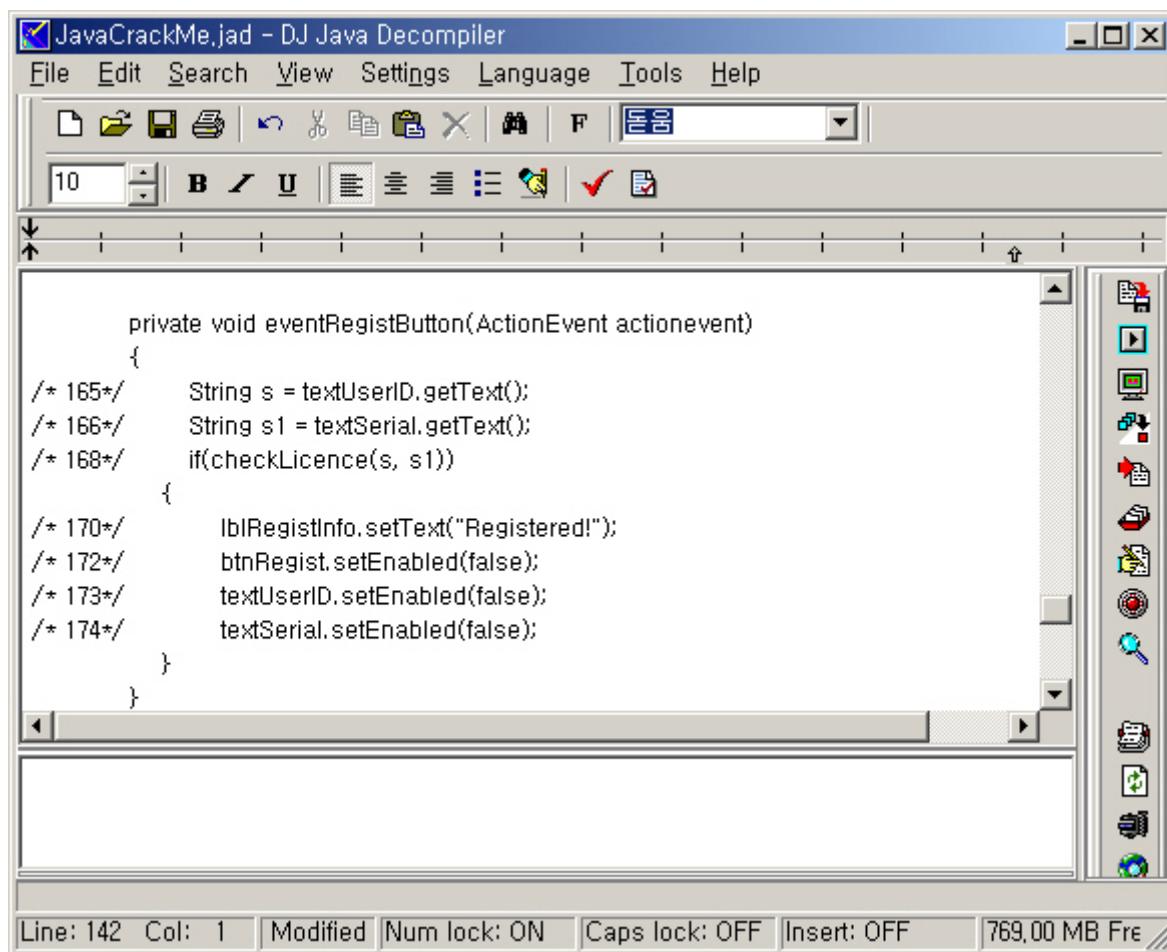


그림 7. JavaCrackMe.jad(Decompile 체크 루틴 부분)

JavaCrackMe.jad(Decompile 체크 루틴 부분)

```

private void eventRegistButton(ActionEvent actionevent)
{
/* 165*/     String s = textUserID.getText();
/* 166*/     String s1 = textSerial.getText();
/* 168*/     if(checkLicence(s, s1))
    {
/* 170*/         lblRegistInfo.setText("Registered!");
/* 172*/         btnRegist.setEnabled(false);
/* 173*/         textUserID.setEnabled(false);
/* 174*/         textSerial.setEnabled(false);
    }
}

```

168 행 checkLicence(s, s1) Method 이름에서 알 수 있듯이 프로그램 등록 부분을 체크하고 있다. 또한 직전에 textUserID, textSerial 오브젝트의 getText() 메소드가 호출되어 있는 것도 알 수 있다.

JAVA Application Crack

다음은 이 checkLicence(s, sl) Method 를 검사하여 Key Generator 를 작성 하면 된다. 그것이 귀찮으면, 소스 코드를 아래와 같이 수정하고 다시 컴파일 한 후 JAR 로 묶어줘도 좋다.

The screenshot shows the DJ Java Decompiler interface with the following Java code:

```
private void eventRegistButton(ActionEvent actionevent)
{
    if(true)
    {
        /* 170*/     lblRegistInfo.setText("Registered!");
        /* 172*/     btnRegist.setEnabled(false);
        /* 173*/     textUserID.setEnabled(false);
        /* 174*/     textSerial.setEnabled(false);
    }
}

public static void main(String args[])
{
```

The word "true" is highlighted in red underlined, indicating it is being edited. The status bar at the bottom shows: Line: 132 Col: 11 Modified Num lock: ON Caps lock: OFF Insert: OFF 768,57 MB Free.

그림 8. 프로그램 체크 루틴 수정

CLASS 파일 패치

위에서 본 것 같이 JAVA 는 소스 코드를 복원할 수 있기 때문에 크랙이 매우 쉽다고 할 수 있다. 그러나 Code 난독화나 재 컴파일 과정에서 문제가 발생하면 다른 방법을 찾아야 한다. 이런 경우 CLASS 파일의 역 어셈블 리스트를 얻은 후 CLASS 파일을 재 작성할 필요가 있다.

DJ 는 디컴파일만이 아니라 역어셈블도 가능하기 때문에 바로 역어셈블 리스트를 만들어 보자. 메뉴에서 [Setting]>[Decompiler Setting]을 선택한 후 화면 위쪽 라디오버튼을 [Decompiler(source)]에서 [Disassembler(bytecode)]로 변경하면 DJ 는 역어셈블러로 변환해 준다. 마찬가지로 [Output original line numbers as comments]와 [Debug mode]에 체크 한다. 이렇게 하면 역어셈블 리스트가 만들어 지며, 크랙 패치 작성에 필요한 각종 정보도 출력할 수 있다.

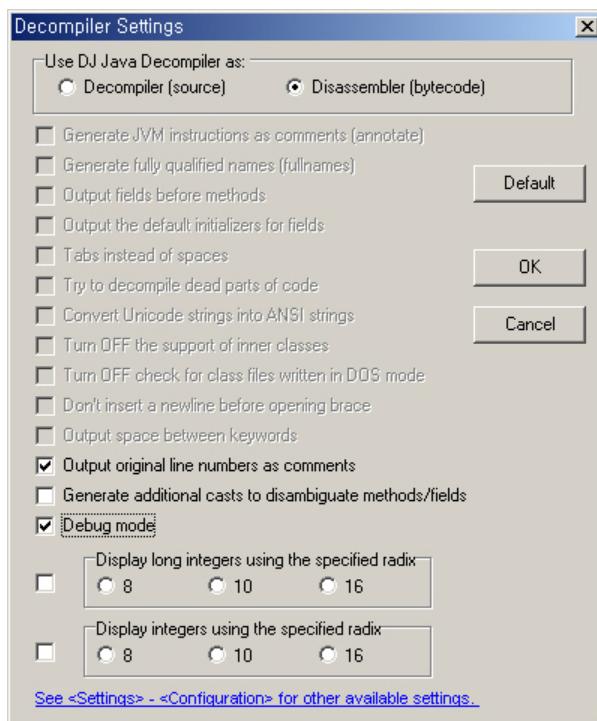


그림 9. Disassembler 설정 화면

환경설정을 끝내고 다시 JavaCrackMe.class 파일을 불러오면 Disassemble 코드로 출력된다. “Output original line numbers as comments”를 선택하였기 때문에 복원한 소스코드 라인 번호와 역어셈블 코드 번호가 동시에 출력된다. 라이센스를 체크하는 Method 부분을 다시 한번 보자.

```

private void eventRegistButton(ActionEvent actionevent)
{
/* 185*/ // 0 0:aload_0
/* 185*/ // 1 1:getfield #33 <Field JTextField txtUserID>
/* 185*/ // 2 4:invokevirtual #88 <Method String JTextField.getText()>
/* 185*/ // 3 7:astore_2
/* 188*/ // 4 8:aload_0
/* 188*/ // 5 9:getfield #37 <Field JTextField txtSerial>
/* 188*/ // 6 12:invokevirtual #88 <Method String JTextField.getText()>
/* 188*/ // 7 15:astore_3
/* 188*/ // 8 16:aload_0
/* 188*/ // 9 17:aload_2
/* 188*/ // 10 18:aload_3
/* 188*/ // 11 19:invokepecial #69 <Method boolean checkLicence(String, String)>
/* 188*/ // 12 22:ifeq 60
/* 170*/ // 13 25:aload_0
}

public MainEventHandler()
public void actionPerformed(ActionEvent actionevent)
public JavaCrackMe()

```

Line: 489 Col: 1 Num lock: ON Caps lock: OFF Insert: OFF 6,587 GB Free

그림 10. 역어셈블 코드(라이센스 체크 Method)

소스 코드에 손을 댄다면 488 행째가 적절하지만, 여기에 적혀져 있는 원래의 소스 코드 행 번호는 168 행이다. 더 자세히 말하면, 역어셈블 리스트의 485–489 행이 원본 소스코드 168 행과 대응된다. 여기서 보면 489 행의 “ifeq”라는 비교/분기 명령 같은 니모닉에 눈이 갈 거라 생각한다. 결론적으로 이 역어셈블 코드를 바꾸면 된다.

JAVA Byte Code

크랙 패치를 작성 하려면 CPU 명령에 대응한 바이트코드 지식이 필요하다. 여기에서는 최소한 필요하다고 생각되는 명령을 소개한다. 그 전에 CLASS 파일의 역어셈블 리스트를 보고 뭔가 이상하다고 생각했을 것이다. 레지스터에 해당하는 것이 존재하지 않는다. Java 가상 머신은 전체 데이터를 스택을 경유하여 주고받는 스택 머신을 모델로 하고 있기 때문에 x86 계열 CPU 명령 코드에 익숙한 사람이라면 조금 당황 할 수도 있겠다. 단지 여기에서는 어디까지나 크랙 패치 작성은 중점적으로 하고 있기 때문에 가장 중요하다고 생각되는 비교/분기 명령으로 한정하여 소개하겠다.

아무것도 하지 않는 명령

X86 계열의 NOP 과 동일하다.

바이트코드	니모닉
00	nop

표 1. NOP 명령

무조건 분기 명령

바이트코드	니모닉
A7 XX XX	goto XX XX
C8 XX XX XX XX	goto_w XX XX XX XX

표 2. 무조건 분기명령

x86 계열 jmp 명령과 동일하다. 바이트코드 중의 XX는 연산자에 해당하는 부분이지만, 여기에서는 목적지 주소가 상대값으로 들어 있다. 점프 오프셋 범위가 2 바이트인지 4 바이트인지는 goto 와 goto_w 로 나누어 사용되지만, goto_w 가 사용되는 것은 거의 없다고 본다.

Java 가상 머신의 상세한 점이나 타 바이트 코드를 알고싶은 경우는 아래 자료를 참고로 하면 좋을 것이다.

The Java TM Virtual Machine Specification Second Edition[영어]

(<http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSSpecTOC.doc.html>)

Int 분기 명령

x86 계열의 경우 “비교”와 “분기”는 각각 다른 명령이지만, JAVA 바이트코드는 int 형만 하나의 명령으로 비교와 분기를 수행할 수 있다. 여기에서 [int 형]이라고 제한된 표현으로 쓰고 있지만, boolean, byte, char, short 형의 값은 컴파일의 단계에서 int 형식으로 강제 형 변환되기 때문에 모두 동일하다. 또한 Java 는 x86 계열과 달리 전체 값이 부호가 있는 것으로 취급된다.

if?? 비교/분기 명령

Operand 스택 최상위 값인 int 형의 값(val1 이라 표현)과 0 을 비교/분기하는 명령이다. 명령은 6 종류가 있지만 각각 분기 성립 조건이 다르다.

바이트코드	니모닉	분기성립 조건
99 XX XX	ifeq	Vall == 0
9A XX XX	ifne	Vall != 0
9B XX XX	iflt	Vall < 0
9C XX XX	ifge	Vall >= 0
9D XX XX	ifgt	Vall > 0
9E XX XX	ifle	Vall <= 0

표 3. if??명령

icmp?? 비교/분기 명령

Operand 스택의 최상위 값인 int 형의 값(val1 이라 표현)과 2 번째에 쌓인 int 형의 값(val2 로 표현)을 비교하여 분기하는 명령이다.

바이트코드	니모닉	분기 성립 조건
9F XX XX	icmpeq	val1 ==0
A0 XX XX	icmpne	val1!=0
B1 XX XX	icmplt	val1 < 0
C2 XX XX	icmpge	val1 >=0
D3 XX XX	icmpgt	val1 > 0
E4 XX XX	icmple	val1 <=0

표 4. icmp?? 명령

원래 니모닉은 선두에 “if_”가 붙어서 “if_cmpeq”와 같이 되지만, DJ로 역어셈블한 경우는 “if_”가 붙지 않는다.

Long 형태 값의 비교 명령

long 형태 값의 비교명령은, 비교결과를 int 형태로 Operand 스택에 PUSH 만 하고 분기까지는 수행하지 않는다. 그래서 비교결과에 대하여 앞에서 말한 if?? 명령을 함께 사용한다.

바이트코드	니모닉
94	icmpeq

표 5. long 형태 값의 비교명령

비교에 의해서 Operand 스택에 PUSH 된 값은 아래 표 6 과 같다.

PUSH 되는 값	비교
1	val1 > val2
0	val1 == val2
-1	val1 < val2

표 6. PUSH 되는 비교결과

부동 소수점의 비교 명령

부동 소수점 사이의 비교명령은 long 형 비교명령과 같이 비교결과를 int 값으로 Operand 스택에 PUSH 만 하고 분기까지는 수행하지 않는다. 비교결과에 대하여 앞서 말한 if?? 명령이 함께 사용된다.

바이트코드	니모닉
95	fcmpl
96	fcmpg

표 7. fcmp?명령(float 형 비교)

바이트코드	니모닉
97	dcmpl
98	dcmpg

표 8. dcmp?명령(double 형 비교)

비교에 의해 Operand 스택에 PUSH 되는 값은 표 9 와 같다.

푸쉬되는 값	비교
1	val1 > val2
0	val1 == val2
-1	val1 < val2

표 9. PUSH 되는 비교결과

단, val1, val2 어느 한쪽이라도 숫자가 아닌 경우의 fcmpg, dcmpg 명령을 사용하면 1, fcmpl, dcmpl 명령이면 -1로 결과값이 리턴된다.

참고 비교 분기 명령

Operand 스택의 참고 값(Object)을 대상으로 한 비교/분기명령이다. ifnull/ifnonnull(참고값 null 판단/분기)는 참고값이 null인지 아닌지 판단하는 명령이다. 오브젝트 사이에서 대소 비교는 안되기 때문에 null인지 아닌지 두 가지 명령밖에 존재하지 않는다.

바이트코드	니모닉	분기 성립 조건
C6 XX XX	ifnull	val1 == null
C7 XX XX	ifnonnull	val1 != null

표 10. 참고값 null 판단/분기명령

바이트코드	니모닉	분기 성립 조건
A5 XX XX	if_acm	val1 == val2
A6 XX XX	if_acmp	val1 != val2

표 11. icmp??의 참고 값 비교/분기

이상의 명령만 파악해 두면 크랙 분석시 어려움은 없을 것이다.

크랙패치 작성

바이트코드 설명이 길었지만, 이제 크랙 패치 작성을 해보도록 하자. 위의 비교분기 명령어 표에서 역어셈블리스트로부터 489 행째의 ifeq 명령을 바꾸면 된다는 것은 이해할 수 있을 것이다. 이 명령이 CLASS 파일의 어느 부분에 배치되어있는지 조사해 보자.

역어셈블리스트의 470 행째에 [Code length: 61 bytes, Code offset: 4068]라고 있고, 이 4068(10 진 표기)은 eventRegistButton 메소드의 선두 오프셋이다. 그리고 바꿀 대상이 되는 489 행째의 [22:ifeq]의 22(이것도 10 진

JAVA Application Crack

표기)도 메소드 선두부터의 상대 오프셋 값이다. 따라서 ifeq 명령에 대응하는 오프셋 어드레스는 $4068+22=4090(0x0FFA)$ 이 된다.

바이너리 에디터로 CLASS 파일 “JavaCrackme.class”을 열어 오프셋 주소 0x0FFA로 건너뛰면, ifeq에 대응하는 바이너리는 “99 00 26”인 것을 알 수 있다(그림 11)⁹⁰. 이번에는 점프하지 않으면 되기 때문에 이 명령을 nop 명령으로 변형한다(그림 12). 이것으로 크랙은 완료된 것이다.

FILENAME JavaCrackMe.class	
00000FFA: 99 00 26 2A	[변경 전]
00000FFA: 99 00 26 00	[변경 후]

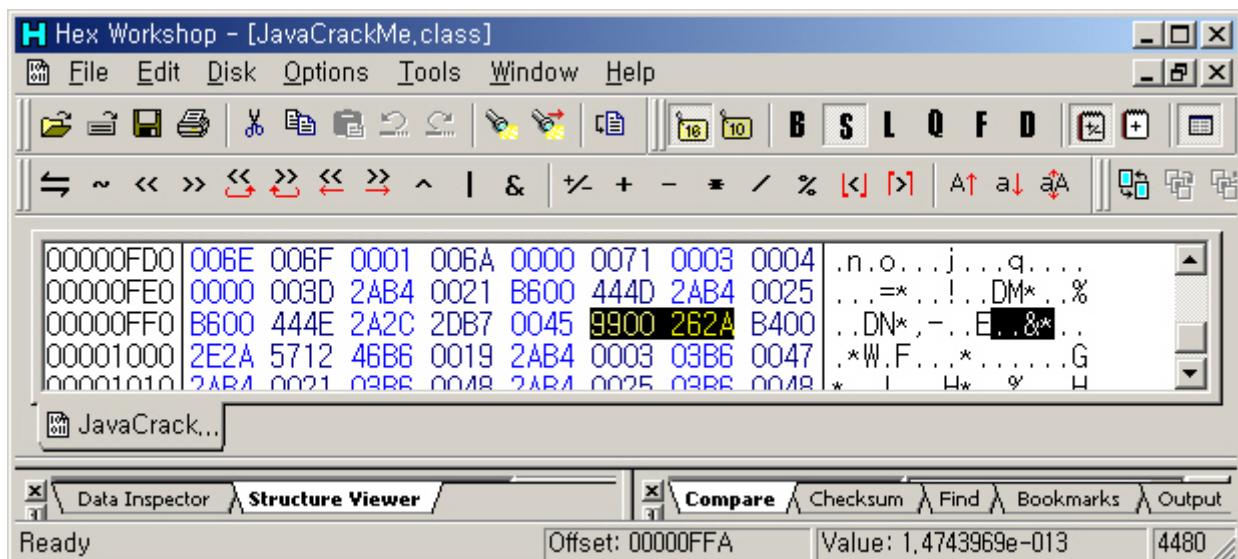


그림 11. 수정 전

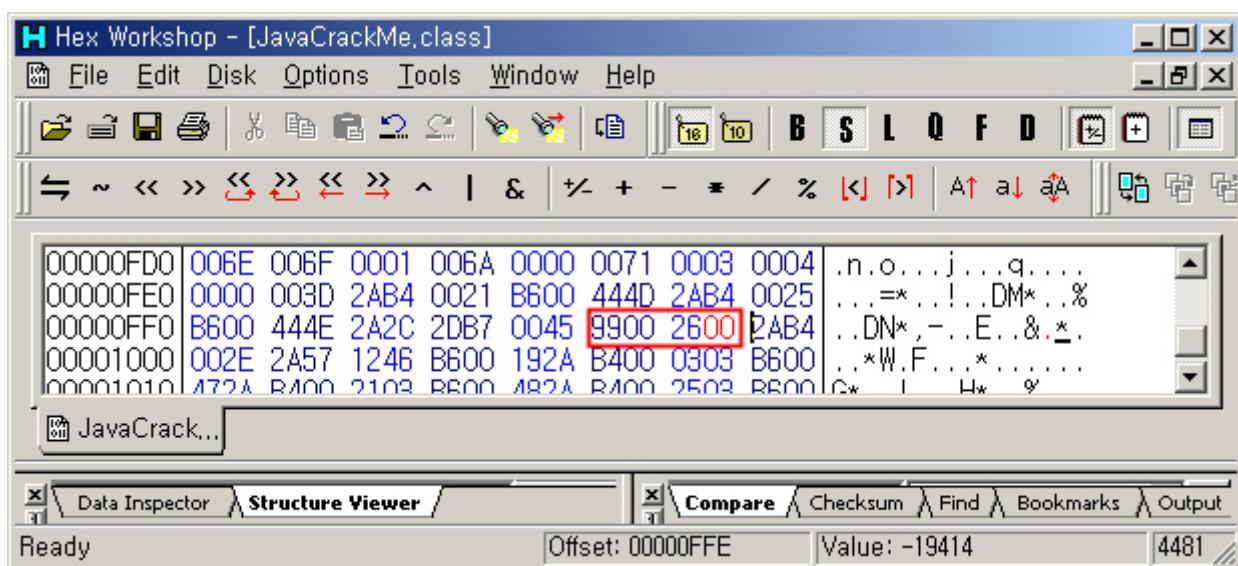


그림 12. 수정 후

⁹⁰ “00 26”이 건너뛸 주소를 나타내고 있지만, Java 가상 머신은 Big-Endian 형식이기 때문에 0x2600 이 아니라 0x26 이다.

Java 프로그램이 디컴파일 되기 쉬운 이유

왜 java 프로그램은 디컴파일되기 쉬울까? Java 는 Java 가상 머신을 사용하기 때문에 운영체제나 CPU 가 달라도 같은 프로그램을 동작시킬 수 있다는 것은 이미 설명하였다. 그러나 바이트 코드 구조도 각 플랫폼에 의존하지 않는 형태가 될 수 없고, Sun 의 javac 도 컴파일러가 통상적으로 수행해야 할 [최적화]를 거의 수행하지 않는다(※참고 2). 게다가 바이트 코드 명령 자체가 객체 지향으로 설계된 구조이기 때문에, 프로그램 구조가 복잡하게 되는 것도 없이 거의 원본 형태로 바이트코드에 반영되어 버린다. 원본 구조가 보존되는 이상 소스 코드 복원도 손쉽게 되는 것이다.

또한 CLASS 파일이라는 형태로 프로그램을 모듈화할 수 있는 구조에도 문제가 있다. 이것은 외부 참조를 전제로 하고 있는 것이라서 CLASS 파일에 함수명의 심볼 정보가 그대로 들어 있기도 하다. 디컴파일러는 이 CLASS 파일의 심볼 정보와 바이트코드를 일치시켜 가독성 높은 소스코드를 복원할 수 있는 것이다.

※참고 2)

최근 컴파일러는 최적화 기능이 매우 다양하게 존재하며, 특정 CPU 아키텍처에 맞춘 최적화도 아주 당연하게 수행하고 있다. 그러나 Java 프로그램의 최적화는 컴파일러가 아닌 Java 가상머신이 담당한다. Java 컴파일 단계에서는 동작하는 플랫폼을 확정하지 않기 때문에, 원본 코드의 구조를 붕괴해서라도 성능 향상을 위해 최적화를 수행하는 것은 알맞지 않다. 단, 정수를 넣거나 Dead 코드 제거 등 간단한 최적화는 수행한다.

끝마치며...

소프트웨어 리버스 엔지니어링은 원칙상 금지되어 있지만, 기존 프로그램을 해석하고, 자사제품보다 우위를 차지하기 위한 행위는 일상적으로 행해지고 있다. 그러나, 독자 기술이나 아이디어 등 비밀로 해야 할 지적 재산을 포함한 프로그램에게 있어서 디컴파일러 존재는 바로 위협이라고 말할 수 있다.

이런 위협을 피하기 위해 개발자도 디컴파일에 대응하는 [난독화], [패킹] 기술을 사용한다. 보다 자세한 내용은 다른 장에서 설명하도록 하겠다.

문서의 오탈자, 오류, 추가내용은

nopter@gmail.com