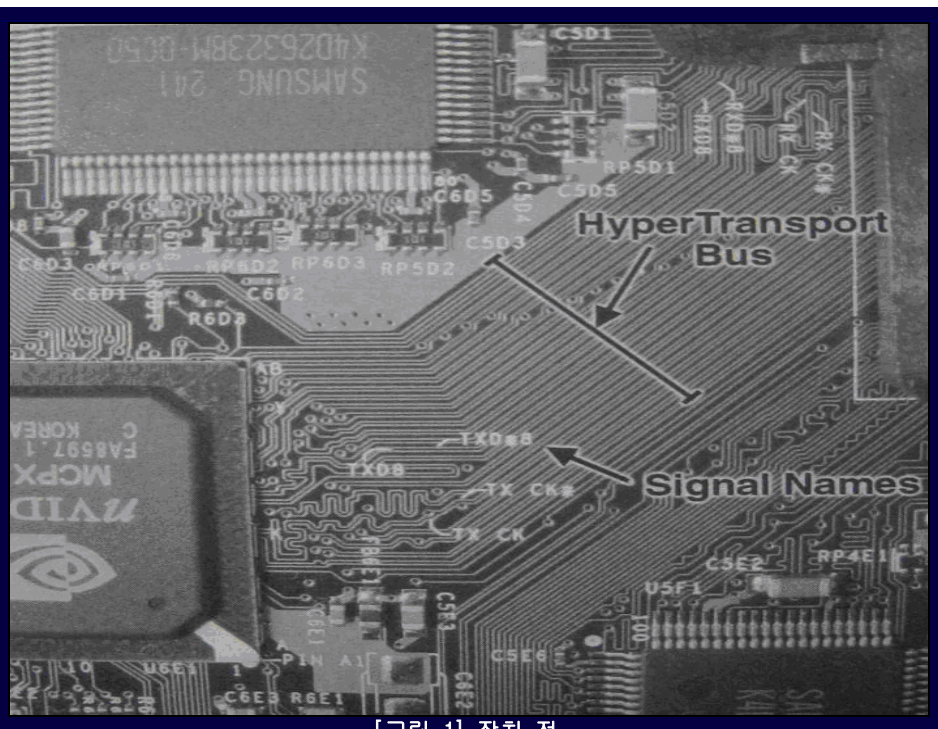
Art of Hooking

제 작 : AmesianX
배 포 : powerhacker.net

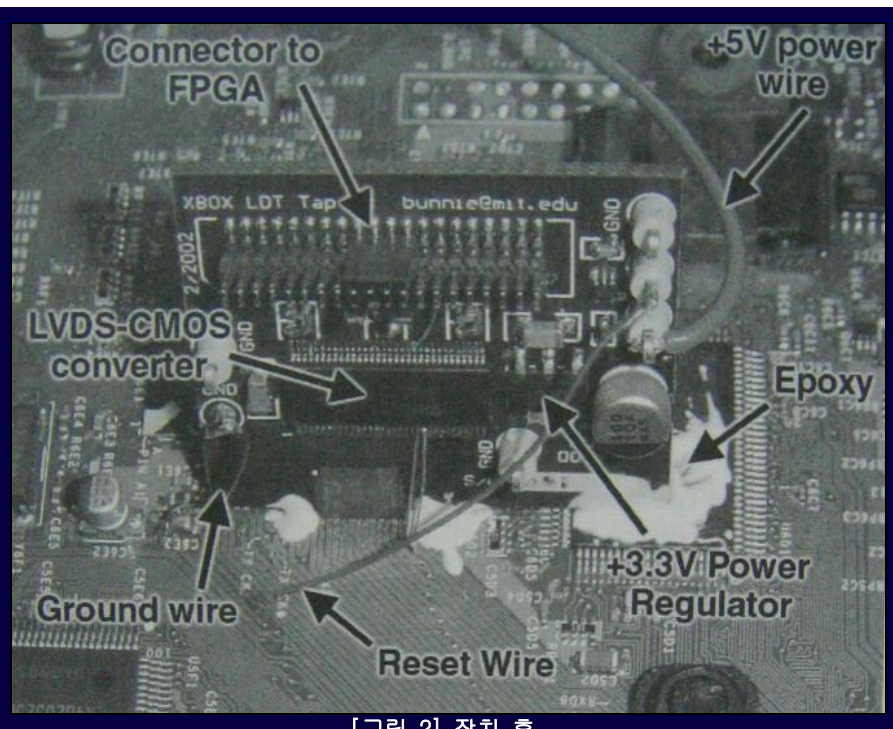
이 문서의 목적은 리버스 엔지니어링에서 디버깅과 디컴파일 다음으로 중요하다고 할 수 있는 후킹 기술을 습득함으로써 소스가 없어도 모든 실행 가능한 프로그램들을 대상으로 제어(추가/삭제/변조 및 제어) 할 수 있는 방법을 설명하는데 있다. 현 시점의 특성상 윈도우즈 플랫폼에 국한되어 설명할 것이나 일부 기술은 플랫폼 독립적인 구현이 가능하다. 무엇보다도 본 문서는 후킹이라는 기술을 실제로 써먹는 것을 보여줄 것이며 후킹기술 중에서도 가장 궁극의 후킹기술을 설명하는데 주력할 것이다.

후킹의 목적

먼저 후킹을 논하기 전에 후킹은 무엇이고 왜 하는가 정도의 목표 의식은 갖어야 할 것이다. 후킹이란 어떤 코드의 실행을 가로채어 제어하는 것을 말한다. 이러한 가로채기와 제어가 필요한 이유는 프로그램의 동작을 우리의 뜻대로 조종할 수 있다는 것이다. 왜 가로채고 제어를 해야 하는지 먼저 리버스 엔지니어링의 시초가 된 하드웨어 쪽을 보면 알 수 있다. 다음의 그림은 "Hacking the Xbox - An Introduction to Reverse Engineering" 이라는 책에서 발췌한 것이다. 이 책의 저자는 MIT 에 다니는 공대생으로 Xbox 를 처음 해킹한 사람이며 자신이 운영하는 홈페이지에 직접 Xbox 의 해킹된 ROM 이미지를 올려놓았다가 MS 에 항의를 받기도 했다고 한다. 이 저자는 직접 자신의 책에서 Xbox 를 해킹하는 방법을 설명하면서 다음과 같은 두 개의 그림을 실었는데 우리가 공부하려는 가로채기를 하드웨어적으로 보여주고 있다.



[그림 1] 장치 전



[그림 2] 장치 후

위에서 왼쪽 그림은 HyperTransport 버스를 나타내고 바로 오른쪽 그림은 이 버스 위에 직접 만든 칩을 Epoxy 로 접착시켜 연결한 장면이다. 저자는 위와 같은 방법으로 Xbox 의 데이타 버스에 지나가는 코드를 캡춰하여 로깅을 하는 것을 보여주고 있다. 필자가 하드웨어 지식이 전무하기 때문에 자세히는 모르겠지만 아마도 이런식으로 ROM 이미지를 빼내거나 제어를 하는 것으로 판단된다. 우리는 하드웨어가 아닌 소프트웨어적인 관점으로 접근할 것이기 때문에 위와 같은 작업을 소프트웨어에서의 후킹과 같은 작업이라고 생각할 수 있다. 즉, 우리는 하드웨어가 아닌 소프트웨어 프로그램을 대상으로 제어를 가로채려는 것이다.

후킹으로 할 수 있는 일들

우리는 중고등학교처럼 목표없이 쓰레기 같은 주입식 공부를 답습할 필요는 없다. 교과서로 할 수 있는 일들이 뭔지 알아야 한다. 필자의 경우 고등학교에서 벡터를 배울때 3D 모델을 제어할 수 있다는 것을 보여줬다면 다시 정석책을 뒤져보며 후회하지는 않았을 것이다. 이 문서도 마찬가지로 내용만을 따지고 보면 기존의 책과 문서들에서 설명한 후킹기술 그 이상도 이하도 아니지만 자신만의 리버스 프로젝트로 가는 도구로써 활용한다면 분명 도움이 될 것이고 기술을 위한 기술을 원한다면 이 문서는 아무런 도움도 줄 수 없다.

후킹으로 할 수 있는 일은 리버스 엔지니어링의 결과를 직접적으로 표현하는 것이다. 너무 추상적인 말이 되지 않기 위해서 다시 설명하자면 리버스 엔지니어링 기술을 통하여 프로그램을 분석한 다음 그 분석한 내용을 기반으로 프로그램에 변조를 가하기 위해서 후킹을 사용한다. 동시에 위의 그림에서 보인 것처럼 리버스 엔지니어링 분석기술로써 활용되기도 한다. 하지만 이것이 후킹으로 할 수 있는 일들의 전부라고 하기에는 뭔가 모자라 보인다.

우리들은 오늘의 최신 기술이 내일의 쓰레기가 되버리는 현실속에서 살고 있다.
시간이 지나고 플랫폼이 바뀌어도 훌륭한 프로그램이 끝까지 남을 수 있는데는 한계가 있다. 이를테면 도스 프로그램들은 이미 사장되었고 윈도우즈 버전으로 다시 나오기도 하지만 도스에서 쓰던만큼 심플한 매력을 주지 못하는 안타까운 경우가 종종 있다. 마찬가지로 윈도우즈에서 명작으로 평가되는 게임이 더이상 개발사가 유지보수를 안하는 경우도 있다. 예를들면 최신 그래픽카드가 nVidia 계열인데 3DFX 의 voodoo 라는 구형 3D그래픽 카드만 지원해서 최신 3D가속기능을 제공받지 못할 수 도 있다. 만약 이런 프로그램들의 소스코드가 공개되어 있다면 인터넷에서 활동하는 공개 개발자들이 무상으로 유지보수를 할 수 있을 것이다. 그러나 소스가 공개되지 않으면 거의 불가능에 가까운 작업을 리버스 엔지니어링이라는 역공학 기술로 가능하게 한다. 도스용 프로그램처럼 윈도우즈와 기반구조가 틀린 경우에는 에뮬레이터를 만들기도하며 분석을 통해서 프로그램을 윈도우즈용으로 다시 만들어 내기도 한다. 그렇지만 기반구조가 서로 다른 플랫폼이 아니라면 소스가 없어도 후킹만으로 프로그램을 유지보수 하는 것이 가능하다. 즉, 같은 계열 운영체제에서는 변조가 가능하다. 도스에서 윈도우즈나, 리눅스에서 윈도우즈 같은 경우 리버싱에 의한 소스코드 재현에 의존해야 하지만 도스에서 도스, 윈도우즈에서 윈도우즈, 리눅스에서 리눅스로 어떤 프로그램을 변조할 경우는 후킹만으로도 충분하다.

마지막으로 좀 더 설명하자면 후킹은 불가능을 가능하게 한다. 후킹기술만 볼때는 불가능해 보이는 것을 가능하게 하는 무서운 능력을 가지고 있는데 온라인 게임의 순간이동장치나 맵핵, 인터넷으로 매번 접하는 유틸리티의 크랙(시리얼키, 복사방지 및 하드웨어 프로텍션 등등을 깨버린 파일), PSP 처럼 임베디드 시스템에서 커널을 직접 패치하여 게임 디스크 대신 덤프(DUMP)된 게임을 돌리게 하는 등등 불가능해 보이는 것들이 후킹기술을 사용하면 가능한 작업들이다. 언급한 내용들을 보니 거의 90% 이상이 불법적인 인상이 풍기는데 누가 합법과 불법을 컴퓨터위에 얹어 놓았는가? 그건 이윤을 챙기기 위한 인간이 꾸며낸 잣대이다. 역사적으로 볼때 불가능을 가능하게 하는 막강한 기술들은 가진자의 기득권이 유린 당할 수 있다는 판단하에 외면당하고 불법과 이단이라는 딱지가 붙어왔다. 해커들은 기술적으로 불가능한 것을 극복하는데만 온 힘을 다하는데 배후세력(?)들은 장삿속을 챙기는데 기술을 계산때리고 있다. 반면 해커들은 무척 단순해서 모든 플랫폼을 대상으로 자신이 작동시키고 싶은 프로그램을 원하는대로 작동시킬때 얻는 희열을 맛보려고 해킹에 몰두한다. 이런 이유때문에 어떤 플랫폼이던지 후킹기술이 단골손님처럼 등장한다.

후킹의 의미

후킹으로 할 수 있는 일들은 사실상 리버스 엔지니어링이라는 역공학 분석기술이 밑바탕 되어야 가능한 것이기 때문에 후킹기술 자체는 엔지니어링에 속한다고 할 수 있다. 의사들이 연구를 통해 습득한 의학 지식들이 리버스 엔지니어링과 같다면 그들이 환자에게 행하는 수술 자체는 후킹과 같다. 의사들은 환자의 배를 갈라서 암 덩어리를 제거하기도하고 인체에서 가장 중요한 심장을 갈아 끼워넣는 엽기적인 짓도 한다. 이것이 후킹과 비교해 다를바가 하나도 없다. 단적인 예로 수술을 실수해서 환자가 죽는 것이 의학에만 있는 것이 아니다. 리버스 엔지니어링에서도 후킹을 실수할 경우 프로그램이 죽어버린다. 만약 의사들이 심장의 면역반응을 어떻게 체크하고 피와 산소, 마취제는 얼마나 줘야하는지 등등 환자가 죽지 않도록 유지하면서 심장을 바꿀 수 있는 자세한 수술법과 의학기기들을 그대로 전수해 준다면 우리가 취미로 의학을 공부하는 사람들이라고 가정할 때 수술을 과연 하지못할까? (물론, 면허가 있고 나름대로 열심히 공부했다는 가정하에 질문해본다..) 의학도들도 결국엔 수술을 하지 않는가? 필자는 후킹이라는 수술을 할 때 프로그램이 죽지 않도록 주의할 조건들과 수술에 성공할 수 있는 방법들을 설명할 것이다.(성공은 각자의 몫이다. 아마 여럿 죽여야 할 것이다. -_-)

후킹의 종류

후킹의 종류는 다음과 같이 API 후킹, 메시지 후킹, 코드(또는 API) 인터셉트 후킹 등으로 나누어진다. 각각의 후킹 방식들은 장단점이나 한계점이라는 것이 있기 때문에 사용하는 용도에 따라서 선택해야 한다. 이 중에서 우리는 코드(API) 인터셉트 후킹을 다룰 것이다. 이 코드 인터셉트 후킹은 리버스 엔지니어로써 꼭 갖춰야 할 기본 기술임에는 틀림없어 보이지만 이 기본기술을 익히기 위해 요구되는 제반지식 또한 만만치 않다. 주로 해외에서 국제적으로 활동하는 크래커(<1>)들은 이 기술을 밥먹듯이 구사하는 것으로 보인다. 아마도 금전적 풍요로움을 제공해 주기 때문일까.

<1> 양심상 해커라고는 말하지 못 할 것 같다. 그들은 프로그램을 크랙하고 변조하는 것이 주특기이고 바이너리 조작은 그들 세계에서는 아주 일상인 것처럼 보이지만 해커와 달리 창조적인 면은 보이지 않는 것 같다. 뛰어난 기술과 머리로 무장하고 있지만 프로그램의 보안장치와 같은 것을 풀기위해 크랙이나 후킹을 한다. 반면, 해커들은 뭔가 새로운 것을 만들기 위해 길목을 트는 역할로써 후킹을 사용한다. 이것이 가장 큰 차이점이라고 생각된다. 뒤에서 설명할 D2Hackit 같은 프로그램의 경우에도 돈이 목적이었다면 소스를 공개하지 않았을 것이며 공개적으로 여러명과 같이 개발하지도 않았을 것이다.
다른 예로써 PSP 라는 휴대용 게임기에서 펌웨어를 에뮬레이팅 해주는 프로그램이 있는데 후킹기술로 커널을 패치하여 추가기능을 구현하고 있다. 물론 처음 성공한 해커가 다른 사람들이 좀 더 나은 개발로 가는데 사용하도록 소스를 공개하고 있다. 필자는 해커와 크래커에대한 용어의 의미를 나름대로 정의하고 쓰는데 의식있는 개발자(기술공유와 창조적 가치의 생산)와 아닌 개발자(기술폐쇄적이며 소모적 가치의 생산)로 보고있다. (개발을 못하는 해커와 크래커는 존재하지 않는다고 본다) 우리가 백오리피스를 만든 사람들을 크래커라고 했었던가? 해커라고 불렀을 것이다. 그들은 누군가 더 나은 것을 만들 수 있도록 소스를 공개하고 기술을 공유했기에 그렇게 불려도 이상할 것이 없다고 본다. 바로 창조적 가치를 생산했기 때문이 아닐까.

API 후킹의 방식과 한계

API 후킹은 타켓 프로그램이 공유라이브러리(DLL)의 익스포트(Export)된 함수들을 호출하기 위해서 참조하는 임포트(Import) 테이블을 조작하여 후킹하는 것이다. 그러나 이 방식은 API 라는 함수적 단위만 후킹이 가능하여 프로그램의 원하는 코드만 가로채는 것은 좀 더 머리를 굴려야 할 것이다.

메시지 후킹의 방식과 한계

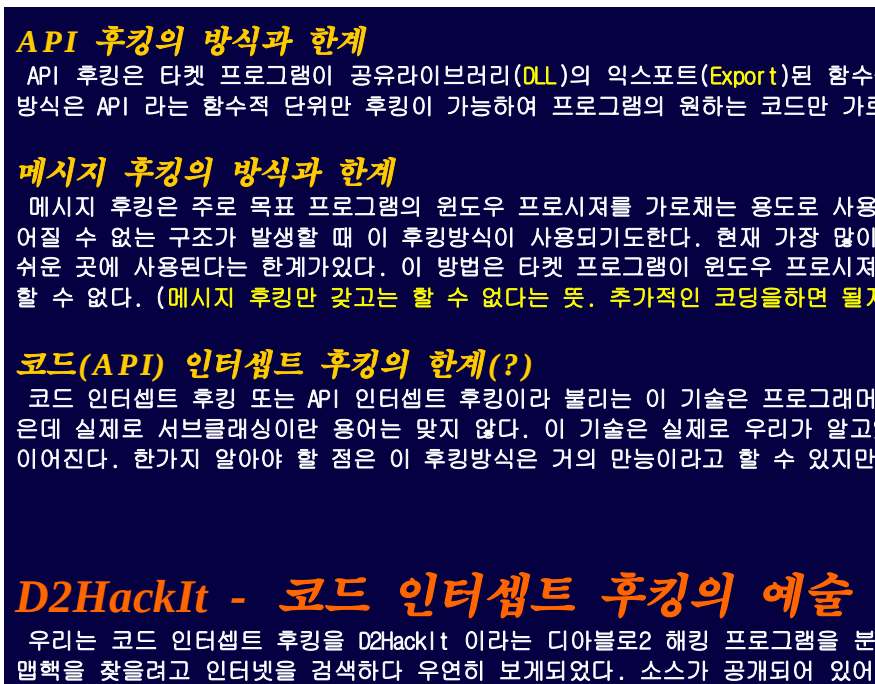
메시지 후킹은 주로 목표 프로그램의 윈도우 프로시져를 가로채는 용도로 사용하는데 이 방법을 응용하여 프로그래밍시 하위 윈도우와 상위 윈도우가 서로 메시지 통신이 이루어질 수 없는 구조가 발생할 때 이 후킹방식이 사용되기도한다. 현재 가장 많이 사용되고 있는 손쉬운 후킹방법이기 때문에 누구나 한번쯤은 접해봤을 것이다. 그러나 쉬운만큼 쉬운 곳에 사용된다는 한계가있다. 이 방법은 타켓 프로그램이 윈도우 프로시져가 없을 경우에는 후킹할 수 없고 또한 타켓 프로그램 속의 원하는 위치에 코드를 삽입하는 것도 할 수 없다. (메시지 후킹만 갖고는 할 수 없다는 뜻. 추가적인 코딩을하면 될지 모르지만 그건 메시지 후킹영역이라고 할 수 없다.)

코드(API) 인터셉트 후킹의 한계(?)

코드 인터셉트 후킹 또는 API 인터셉트 후킹이라 불리는 이 기술은 프로그래머의 특성상 서브클래싱이라고 부르기도 한다. 서브클래싱과 흡사하기 때문에 그렇게 부르는 것 같은데 실제로 서브클래싱이란 용어는 맞지 않다. 이 기술은 실제로 우리가 알고있는 코드패치 기술이다. 이 후킹방식은 본격적으로 다룰 내용이므로 자세한 설명은 다음에 계속 이어진다. 한가지 알아야 할 점은 이 후킹방식은 거의 만능이라고 할 수 있지만 후킹코드를 제작하는 사람의 프로그래밍 능력에 따라 좌우된다는 것이 한계라면 한계일 것이다.

D2HackIt - 코드 인터셉트 후킹의 예술

우리는 코드 인터셉트 후킹을 D2HackIt 이라는 디아블로2 해킹 프로그램을 분석함으로써 습득할 것이다. 이 D2HackIt 은 필자가 2001년 경 디아블로2에 빠져 허우적 될 당시 맵핵을 찾을려고 인터넷을 검색하다 우연히 보게되었다. 소스가 공개되어 있어서 뜯어봤는데 필자는 그만 경탄을 금치 못했다. 그 이유는 소스 자체가 바로 예술 그 자체였기 때문이다. D2HackIt 의 소스를 보게되면 먼저 채팅창을 가로채서 게임진행 중에도 얼마든지 채팅창을 통해 레지스터 내용을 출력하며 디버깅 정보를 얻게 해놓고 게임내부의 함수들을 자유자재로 후킹하여 사용하는 것을 환상적으로 보여주고 있다. 필자는 이 소스를 분석하면서 범용적인 사용이 가능함을 직감하였지만 응용의 실패는 계속되었다. 그 후에도 군입대라는 시련 속에서 야간 근무 때마다 전산실 구석탱이에 짱박혀서 계속 삽질을 하였으나 여전히 실패였다. 결국 제대 후 첫 직장에서 리버싱 프로젝트를 수행하면서 성공하게 되었다. 모 게임회사의 까다로운 보고서 리뷰를 통과하기 위해서는 반복적인 디버깅만이 살길이었는데 아무리 찔러보고 싶은 곳을 발견해도 조작해 볼 수 있는데는 한계가 있었다. 예를들면 버퍼나 내부변수의 조작은 포인터를 갖고와서 조작하거나 스택 값을 바꿔야 하는데 완벽하게 조작하고 다시 원래 프로그램에 아무일도 없었던 것처럼 넘겨주는 조작을 디버거로 한다는 것은 불가능해 보였다. 바이너리에서 간단한 점프루틴을 직접 뜯어고쳐 취약점이 발생하는 것은 이미 보고서로 리포팅한 상태였고 다시 그런 횡재를 바란다는 것은 무리였다. 필자의 머리가 x86 계열 두뇌를 갖고 있다면 모를까 점프루틴만 쫓아가며 새벽까지 삽질하다 멍한 눈으로 퇴근할 수는 없는 노릇아닌가.. 뭔가 분석을 진행하는 중에 얻은 리버싱 데이타만 갖고 직관적인 수정이 가능해야만 했다. 어쩔 수 없이 이 한계를 극복하기 위해서 다시 꺼내든 카드가 D2HackIt 이었고 필연적으로 극복해야할 과제였다. 한동안 안풀리던 수학문제가 어느날 갑자기 쉽게 풀리듯 D2HackIt 을 범용적으로 사용할 때 겪었던 오류들이 풀리며 필자가 착각했던 부분을 알게되었고 결국 D2HackIt을 응용하면 원하는 모든 프로그램을 개조할 수 있다는 것을 알았다. 이미 현 시점은 D2HackIt 이 많이 알려져서 WoW(월드오브워크래프트)의 순간이동 핵(HACK) 같은 프로그램들이 나오기도 했는데 소스를 보면 D2HackIt 을 모태로 하고 있다. 이 외에도 D2HackIt 소스가 공개된 것에 힘입어 이를 응용한 몇몇 핵(HACK) 프로그램과 유니버셜 패처(Patcher)같은 것들이 출현하기도 하였다.



[그림 3] 채팅창을 개조한 모습

위에서 보는 그림은 디아블로2 의 게임대기 룸(ROOM)의 화면인데 제일 상단에 광고가 나와야 할 부분에 커맨드 입력이되고 메인 채팅창에는 스크립트를 불러들여서 사용하고 있다. 원래 게임안에서 채팅창에 명령어를 입력하면 레지스터가 출력되는 장면이나 DLL 이 로딩되는 장면을 넣을려고 했으나 국내와 해외모두 D2HackIt 과 관련된 프로그래밍 접근법에 대한 정보가 전무하다시피 한 것 같다.(원론적인 내용을 제외하고..)

이 문서를 작성하던 중 업데이트가 거의 안되던 코드브레이커즈 저널이 업데이트를 해서 둘러 봤더니 눈에 띄는 기사가 하나 있었다. 바로 필자가 설명하는 내용 중 일부 기본 기술에 대해 도움을 얻을 수 있을만한 기사였다. 이 문서를 읽기 전에 한번 읽어보는 것도 좋을 것이다. 물론, 필자는 생계문제로 시간이 넉넉치 않아 다운로드만 받고 읽어보지는 못해서 어떤 문서인지 설명해 줄 수 없다. 코드브레이커즈의 문서는 다루는 주제만으로도 쉽게 다룰수 없는 최고급기술이 올라오기 때문에 의심할 여지없이 한번 모든 문서를 읽어보기 바란다. 리버싱을 공부하는 사람들은 이 사이트의 모든 문서를 읽어도 시간이 아깝지 않을 것이다.

<www.codebrakers-jounal.com>
CodeBreakers Magazine Vol.1, No.2(2006) - "Guide on how to play with processes memory, writing loaders, and Oraculumns" by Shub Nigurrath

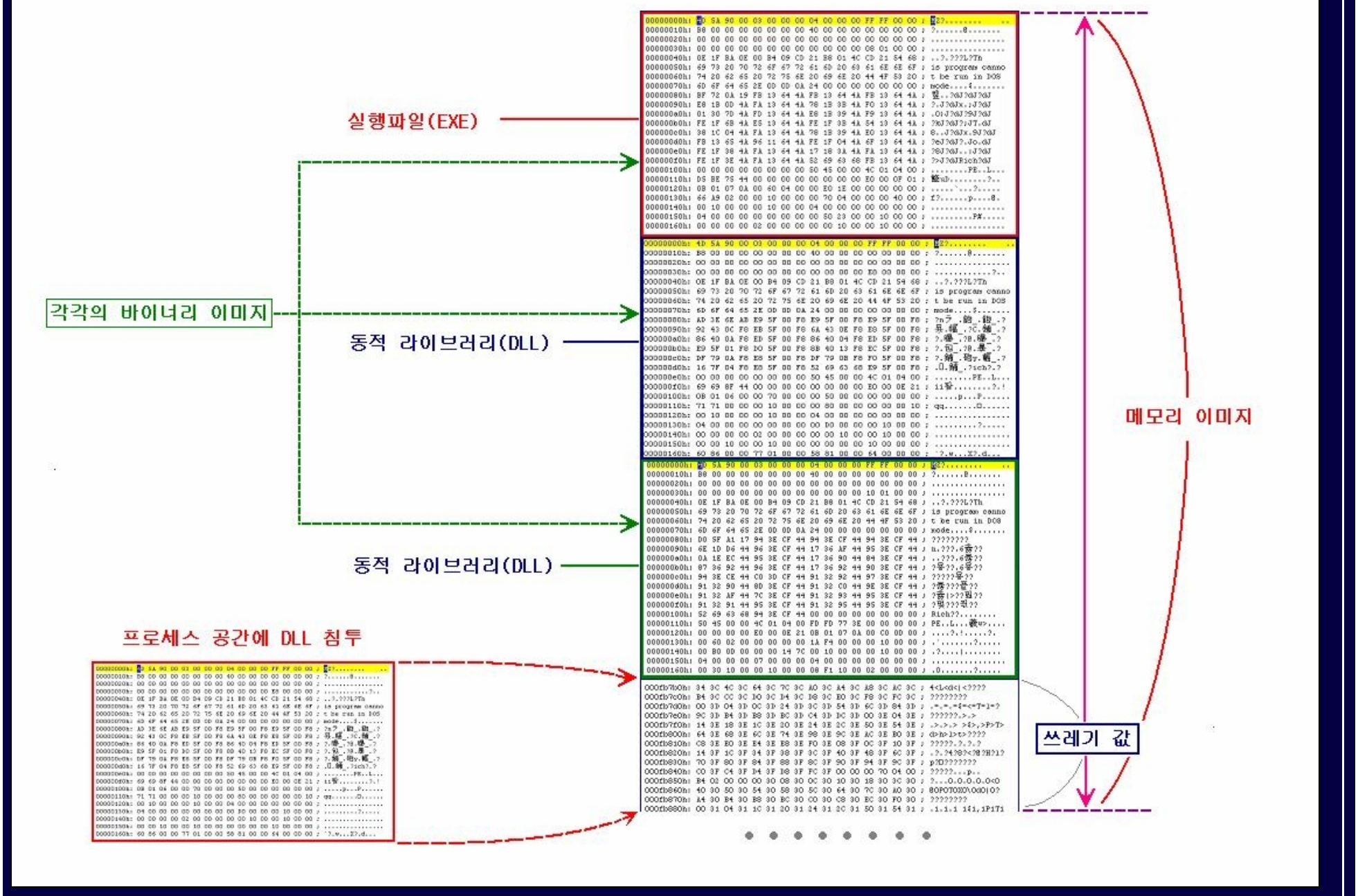
그렇다면 지금부터 인터넷에서 떠돌고 있는 D2HackIt 0.57 버전을 다운로드 받아서 VC++ 로 열어놓고 준비해두길 바란다. 간단한 소스 분석에 들어갈 것이다.

코드 인터셉트 후킹 시작

우리는 코드 인터셉트 후킹을 하기 위해서 타켓 프로그램의 메모리 공간으로 접근할 수 있는 방법이 필요한데 가장 많이 사용되는 DLL 인젝션(Injection)을 사용할 것이다. 이 때 중요한 선택으로써 타켓 프로그램의 실행가능한 파일을 수정해서 DLL 을 삽입해놓고 실행될때 자동으로 로딩되도록 할 것인지 아니면 타켓 프로그램이 이미 실행 된 후 DLL 을 동적으로 인젝션시킬 것인가를 정해야 한다. 그 이유는 만약 후자를 선택해서 DLL 을 타켓 프로그램으로 인젝션시키려면 로더(Loader)라는 것을 만들어야 하기 때문이다. D2HackIt 에는 이미 D2Loader 소스가 있기 때문에 이 소스를 수정해서 사용하면 될 것이나 필자는 이 로더를 사용하지 않고 Detours 라는 후킹 라이브러리가 제공하는 유틸리티 중 Withdll 과 Setdll 을 이용할 것이다. 이 Setdll 유틸리티는 아예 타켓 실행파일의 임포트 테이블을 직접 수정해서 DLL 정보를 박아버린다. 그렇기 때문에 실행가능한 파일들(EXE/DLL 등등)은 백업해 두지 않으면 변조되므로 주의해야 한다. 이 방법으로 직접 바이너리에 DLL 정보를 박아버리면 타켓 프로그램을 실행시킬때 항상 우리가 만든 DLL 을 로딩하게 된다. 이 외에도 Withdll 을 사용할 것인데 Withdll 은 타켓 프로그램의 시작시 해당 프로그램의 프로세스공간에 DLL 을 인젝션한다. 이제 DLL 을 타켓 프로그램의 메모리 공간으로 로딩시키는 방법을 정했으니 침투한 메모리 공간에서 무엇을 할 것인지를 정해야 한다.

우리가 타켓 프로그램에 삽입하게되는 DLL 파일은 아마도 D2HackIt.dll 이 될 것이다. 프로그램이 실행되면 먼저 임포트 테이블의 라이브러리(DLL)를 순서대로 하나씩 로딩하게 되는데 DLL 이 로딩 될 때는 DllMain 이라는 엔트리(Entry) 함수부터 호출이 된다. 즉, 우리가 DLL 프로그래밍을 할 때 프로그래밍 도입부(Entry Point)였던 DllMain 부터 코드가 시작될 것이라는 의미다. 여기서 순서대로라고 말한 이유는 중요한 주의사항인데 먼저 로딩된 D2HackIt.dll 이 아직 로딩되지 않은 DLL 의 함수를 호출하게 되면 프로그램이 박살난다. 이런 로딩 순서를 지키기 위해서 때로는 EXE 파일에 D2HackIt.dll 을 박지 못하는 경우도 생길 수 있으며 EXE 파일 이외에 같은 디렉토리의 다른 DLL 파일에 박아야 하는 경우도 생길 수 있다.(DLL 도 PE 구조이기 때문에 DLL 정보를 추가로 박을 수 있다.) 이렇게 해서 나중에 로딩되는 DLL 에 새로만든 D2HackIt.dll 을 박으면 로딩순서 때문에 프로그램이 박살나지는 않을 것이다.

들어가기 전에 먼저 DLL 이 프로세스 공간에 들어가는 것을 그림으로 나타낸 것을 보고 시작하자.



이제부터 간단한 소스분석을 시작할 것이다. 소스분석은 D2HackIt 소스를 직접 보면서 필자가 인도하는대로 따라오면 될 것이다. 그러나 소스를 분석하는 것에 너무 집착하지 말아야 할 것이다. 그 이유는 나중에 보면알겠지만 소스의 절반 이상이 디아블로2에만 맞춰져 있는 코드이기 때문에 필요없으므로 삭제할 것이기 때문이다. 오히려 나중에 필자가 소스를 재작성한 곳을 집중적으로 분석해야 할 것이다.

Entry Point

```
//////////////////////////////////////////////////////////////////////
// DllMain.cpp
// ------------------------------------------------------------------
// Default Dll entrypoint.
//
// <thohell@home.se>
//////////////////////////////////////////////////////////////////////
#define THIS_IS_SERVER
#include "..WD2HackIt.h"

// Global structures
SERVERINFO                  *si;
PRIVATESERVERINFO           *psi;
FUNCTIONENTRYPOINTS         *fep;
PRIVATEFUNCTIONENTRYPOINTS  *pfep;
THISGAMESTRUCT              *thisgame;

LinkedList ClientList;

BOOL APIENTRY DllMain( HANDLE hModule, DWORD  ul_reason_for_call, LPVOID lpReserved)
{
    BOOL hResult = TRUE;
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
            // initiate server
            hResult = ServerStart(hModule);
            break;

        case DLL_PROCESS_DETACH:
            hResult = ServerStop();
            break;
    }
    return hResult;
}
```

[그림 4] D2HackIt 의 DllMain.cpp 파일

BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)
타켓 프로그램에 의해 DLL 이 로딩될 때 DllMain 이 호출되어 도입부(Entry Point) 역할을 하게된다. 로딩될 때는 DLL_PROCESS_ATTACH 가 인자로 넘어오고 타켓 프로그램이 종료할 때 DLL_PROCESS_DETACH 가 넘어오게 된다. DLL 이 로딩될 때와 해제될때 각각 ServerStart 와 ServerStop 이 실행되는 것을 위의 그림4 를 통해 알 수 있다. 일차적으로 우리는 흐름을 따라갈 것이므로 더이상 여기서 얻을 정보가 없기 때문에 바로 ServerStart 라는 함수로 넘어가보자.

ServerStartStop

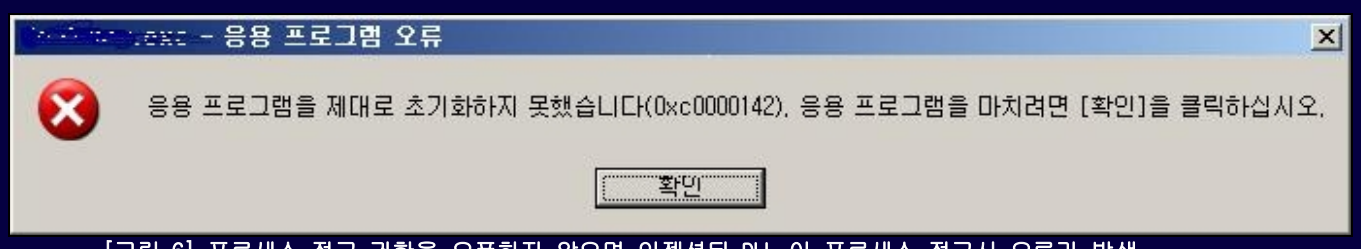
```
//////////////////////////////////////////////////////////////////////
// ServerStartStop.cpp
// ------------------------------------------------------------------
// Initialize/destroy server.
//
// <thohell@home.se>
//////////////////////////////////////////////////////////////////////
#define THIS_IS_SERVER
#include "..WD2HackIt.h"

// These are the dll's we want to force-load to get them in memory.
//char* NeededDlls[] = { "D2Common.dll", "D2Game.dll", "D2Multi.dll", "D2Client.dll", NULL };
char* NeededDlls[] = { "D2Common.dll", "D2Game.dll", "D2Client.dll", NULL };
//////////////////////////////////////////////////////////////////////
// ServerStart()
// ------------------------------------------------------------------
// Responsible for setting up the server.
//////////////////////////////////////////////////////////////////////
BOOL PRIVATE ServerStart(HANDLE hModule)
{
    // Temporary string
    LPSTR t=NULL;

    ////////////////////////////////////////////////////////////////////
    // Before anything else, create the global structures we use in
    // the hack. Make sure we delete these in ServerStop.
    ////////////////////////////////////////////////////////////////////
    si =    new SERVERINFO;
    psi =   new PRIVATESERVERINFO;
    fep =   new FUNCTIONENTRYPOINTS;
    pfep=   new PRIVATEFUNCTIONENTRYPOINTS;

    thisgame=new THISGAMESTRUCT;
    thisgame->player=NULL;
```

[그림 5] ServerStartStop.cpp

자.. 소스가 길기 때문에 위의 그림5 의 소스를 아래와 같이 적어서 주석을 달았다. 그냥 부담없이 읽어보자.

```
#define THIS_IS_SERVER
#include "..WD2HackIt.h"

// These are the dll's we want to force-load to get them in memory.
// 강제로 로딩하기 위한 DLL 목록을 넣어준다.
char* NeededDlls[] = { "D2Common.dll", "D2Game.dll", "D2Client.dll", NULL };

BOOL PRIVATE ServerStart(HANDLE hModule)
{
    LPSTR t=NULL;
    // 후킹한 정보들을 담을 구조체들을 초기화 한다. (주의: 우리는 디아블로2를 후킹하는 것이 목적이 아니기에 디아블로2만의 정보를 담는 구조체는
    // 의미가 없다.)
```

```
    si = new SERVERINFO;
    psi = new PRIVATESERVERINFO;
    // fep 구조체는 한번 유심히 봐야 한다. fep 구조체는 함수포인터들의 집합을 저장하는 구조체이다.
    fep = new FUNCTIONENTRYPOINTS;
    pfep= new PRIVATEFUNCTIONENTRYPOINTS;

    thisgame = new THISGAMESTRUCT;
    thisgame->player=NULL;

    // 위의 NeededDlls 에 있는 DLL 들을 여기서 LoadLibrary 로 강제 로딩을 시킨다. 이렇게 함으로써 우리는 타켓 프로그램이 갖고 있는 다른 DLL
    // 들까지도 패치시킬 수 있다.
    for (int i=0; NeededDlls[i] != NULL; i++) LoadLibrary(NeededDlls[i]);

    // D2HackIt 의 버전 셋팅(의미 없음)
    si->Version=__SERVERVERSION__;

    // D2HackIt 이 잘 만들어진 이유는 플러그인 구조를 취하고 있어서 누구나 역공학으로 디아블로2 의 핵심 루틴을 찾아서 자신만의 기능을 만들고
    // 플러그인으로 제작할 수 있다는 점인데 이 부분이 그 플러그인을 위한 디렉토리를 문자열을 뽑아내는 부분이다.
    t=new char[_MAX_PATH];
    if (!GetModuleFileName((HINSTANCE)hModule, t, _MAX_PATH)) {
        MessageBox(NULL, "Unable to get PluginPath!", "D2Hackit Error!", MB_ICONERROR); return FALSE;
    }

    // GetModuleFileName 으로 대략 C:WD2HackItWD2HackIt.dll 과 같은 내용이 t 버퍼에 들어왔다면 오른쪽부터 검색해서 첫번째로 역슬래쉬가
    // 나오는 곳까지 짜른다.
    int p=strlen(t);
    while (p)
    {
        if (t[p] == '\\') { t[p] = 0; p=0; }
        else
            p--;
    }

    // 이렇게 얻은 플러그인 디렉토리명을 서버정보 구조체 si->PluginDirectory 에 복사한다.(우리가 하려는 작업에 큰 의미는 없음)
    si->PluginDirectory=new char[strlen(t)+1];
    strcpy((LPSTR)si->PluginDirectory, t);
    // 의미없음
    psi->DontShowErrors=FALSE;

    // 앞에서 얻은 디렉토리명과 D2HackIt.ini 를 합치면 대략 C:WD2HackItWD2HackIt.ini 가 되며 psi->IniFile 구조체에 복사한다.
    sprintf(t, "%s\\WD2HackIt.ini", t);
    psi->IniFile=new char[strlen(t)+1];
    strcpy((LPSTR)psi->IniFile, t);
    delete t;

    // Get the process ID and the process handle
    // 현재 프로세스 ID 를 얻어서 접근권한을 풀어버린다.
    psi->pid = GetCurrentProcessId();
    psi->hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, psi->pid);
    if (!psi->hProcess) { MessageBox(NULL, "Can't get Diablo II's process handle.", "D2Hackit Error!", MB_ICONERROR); return FALSE;}
```



[그림 6] 프로세스 접근 권한을 오픈하지 않으면 인젝션된 DLL 이 프로세스 접근시 오류가 발생

```
    // Get build date/time
    // 빌드한 시간을 기록하는 것으로 우리 목적에는 별 의미가 없다.
    strcpy(psi->BuildDate, __DATE__);
    strcpy(psi->BuildTime, __TIME__);

    // 여기서 주의 깊게 봐야할 점은 이름으로도 알 수 있듯이 GamePrint 어쩌구 함수들을 펑션포인터(Function Pointer:함수포인터)들에 셋팅을 하고
    // 있다는 것이다.
    fep->GamePrintString=&GamePrintString;
    fep->GamePrintInfo=&GamePrintInfo;
    fep->GamePrintVerbose=&GamePrintVerbose;
    fep->GamePrintError=&GamePrintError;
    fep->GetHackProfileString=&GetHackProfileString;


// 자세한 설명을 덧 붙이자면 다음의 그림을 보면 fnGamePrintString 과 같은 함수포인터 타입의 변수들 모음을 fep 라는 구조체로 잡고 있는 것을 알
// 수 있다. 프로그래밍시 쉽게 관리할 목적으로 구조체로 지정해 놓은 것이다. GamePrintString 함수는 실제로 GamePrintFunctions.cpp 라는 파일에
// 구현되어 있으며 fep 구조체의 GamePrintString 이라는 함수포인터 변수에 주소를 넣어두고 모든 것을 fep-> 로써 구조체만으로 함수들을 호출하는
// 형태를 취하고 있다. 제작자는 쉽게 관리하기 위한 목적이지만 분석자 입장에서는 구조체를 잘 익혀놓고 분석해야만 코드를 이해할 수 있으므로 까칠
// 한 감이 없잖아 있다.
```

```
//////////////////////////////////////////////////////////////////////
// FUNCTIONENTRYPOINTS *fep;
// ------------------------------------------------------------------
// Pointers to functions inside the game code and helper functions
// inside D2HackIt. This is exposed to the clients.
//////////////////////////////////////////////////////////////////////
typedef BOOL    (__cdecl *fnGamePrintString)(LPCSTR buf);
typedef BOOL    (__cdecl *fnGamePrintInfo)(LPCSTR buf);
typedef BOOL    (__cdecl *fnGamePrintError)(LPCSTR buf);
typedef BOOL    (__cdecl *fnGamePrintVerbose)(LPCSTR buf);
typedef LPSTR   (__cdecl *fnGetHackProfileString)(LPCSTR lpHackName, LPCSTR lpSectionName, LPCSTR lpKeyName);
typedef BOOL    (__cdecl *fnGameSendPacketToServer)(LPBYTE buf, DWORD len);
typedef BOOL    (__cdecl *fnGameCommandLine)(char* buf);
typedef BOOL    (__cdecl *fnIntercept)(int instruction, DWORD lpSource, DWORD lpDest, int len);
typedef BOOL    (__cdecl *fnGetFingerprint)(LPCSTR szHackName, LPCSTR szFingerprintName, FINGERPRINTSTRUCT &fps);
typedef BOOL    (__cdecl *fnSetHackProfileString)(LPCSTR lpHackName, LPCSTR lpSectionName, LPCSTR lpKeyName, LPCSTR lpValue);
typedef BOOL    (__cdecl *fnGameSendPacketToGame)(LPBYTE buf, DWORD len);
typedef PTHISGAMESTRUCT (__cdecl *fnGetThisgameStruct)(void);


typedef struct functionentrypoints_t
{
    fnGamePrintString               GamePrintString;
    fnGamePrintInfo                 GamePrintError;
    fnGamePrintError                GamePrintInfo;
    fnGamePrintVerbose              GamePrintVerbose;
    fnGetHackProfileString          GetHackProfileString;
    fnSetHackProfileString          SetHackProfileString;
    fnGameSendPacketToServer        GameSendPacketToServer;
    fnGameCommandLine               GameCommandLine;
    fnIntercept                     Intercept;
    fnGetFingerprint                GetFingerprint;
    fnGetThisgameStruct             GetThisgameStruct;
    fnGameSendPacketToGame          GameSendPacketToGame;
} FUNCTIONENTRYPOINTS;
extern FUNCTIONENTRYPOINTS      *fep;
```

[그림 7] CommonStructs.h 헤더파일에 fep 구조체와 함수 포인터들

위의 구조체에서 GamePrintString 함수포인터 변수는 제작자가 작성한 GamePrintString 함수주소를 갖게되고 fep->GamePrintString=&GamePrintString 명령이 바로 함수주소를 함수포인터 변수에 할당하는 역할을 한다.

```
BOOL EXPORT GamePrintString(LPCSTR buf)
{
    // Make sure we *CAN* write to screen...
    if ((!psi->GamePrintStringLocation) || (!thisgame->player))
        return FALSE;

    // Make a string big enough to fit header+text
    char t[512];
    char *pt=t;
    DWORD Entrypoint=psi->GamePrintStringLocation;

    // Format the text so game will take it.
    sprintf (t, "%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%s%c",
        0xFF,0x01,0xFF,0xFF,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
        buf, 0x00);

    // The call
    __asm {
        mov ecx, pt
        mov eax, Entrypoint // Function to print text
        call eax
    };

    return TRUE;
}
```

[그림 8] GamePrintString 함수는 제작자가 리버싱을 통해 분석한 정보로 작성된 함수

위의 GamePrintString 함수는 디아블로2 채팅창으로 문자열을 출력하기위한 역할을 하는데 실제로 GamePrintStringLocation 변수가 디아블로2 를 역어셈블 했을 때 채팅창 코드부분의 주소를 갖게된다.(어떻게 GamePrintStringLocation 변수에 그런 주소가 들어가는지는 환경설정파일을 설명하면서 알게될 것이다.) 이미 눈치챘겠지만 우리는 디아블로2 를 후킹하고자 하는 것이 아니고 범용적으로 사용하려고 하는 것이기 때문에 제작자가 디아블로2 를 리버싱해서 찾아낸 주소들은 쓰레기와 같다. 그렇기 때문에 범용으로 사용하려면 제거가 필요하다. 이 문서의 후반부에서 제거하는 부분을 다루게 된다.

```
    // 다음은 프로세스 이미지에 관련된 함수를 셋팅하는 것으로 로딩된 바이너리 이미지의 베이스 주소와 총 크기를 구하기 위해서 작성된 함수의 주소를
    // 구조체에 할당한다. 일단 기본적으로 GetBaseAddress 함수가 베이스 이미지(실행파일) 주소를 리턴하고 GetImageSize 함수는 이미지(실행파일)
    // 총 크기를 리턴한다고만 알아두자.
    pfep->GetBaseAddress=&GetBaseAddress;
    pfep->GetImageSize=&GetImageSize;

    // 실제로 D2HackIt.ini 환경설정파일이 있는지 검사한다
    if (_access(psi->IniFile, 0))
    {
        LPSTR t=new char[strlen(psi->IniFile)+50];
        sprintf(t, "Unable to open ini-file:Wn%s", psi->IniFile);
        MessageBox(NULL, t, "D2Hackit Error!", MB_ICONERROR);
        Delete t;
        return FALSE;
    }

    // 바로 앞에서 GetHackProfileString 이라는 함수를 fep 구조체로 할당하는 것을 보았을 것이다. GetHackProfileString 함수는 D2HackIt.ini 라는
    // 파일에서 해당하는 섹션을 읽어들여서 값을 뽑아내는 역할을 한다. 다음의 설정은 단순히 디아블로2 채팅창에 출력될 프롬프트를 읽어들이는 것으로
    // 우리는 이 부분도 쓸모가 없다.
    t=fep->GetHackProfileString("D2HackIt", "Misc", "InfoPrompt");
    lstrcpyn(psi->InfoPrompt, (strlen(t)?t:DEFAULTINFOPROMPT),MAXPROMPTLENGTH-1);
    delete t;

    t=fep->GetHackProfileString("D2HackIt", "Misc", "ErrorPrompt");
    lstrcpyn(psi->ErrorPrompt, (strlen(t)?t:DEFAULTERRORPROMPT),MAXPROMPTLENGTH-1);
    delete t;
```

```
    t=fep->GetHackProfileString("D2HackIt", "Misc", "VerbosePrompt");
    lstrcpyn(psi->VerbosePrompt, (strlen(t)?t:DEFAULTVERBOSEPROMPT),MAXPROMPTLENGTH-1);

    t=fep->GetHackProfileString("D2HackIt", "Misc", "Verbose");
    if (!stricmp(t, "on"))
        psi->Verbose = TRUE;
    else
        psi->Verbose = FALSE;

    delete t;

// GetHackProfileString 함수를 간단히 보고 넘어가자. 특별한건 없고 D2HackIt.ini 파일을 읽어서 해당 섹션의 문자열을 리턴한다.

LPSTR EXPORT GetHackProfileString(LPCSTR lpHackName, LPCSTR lpSectionName, LPCSTR lpKeyName)
{
    LPSTR lpFileName = new char[strlen(si->PluginDirectory)+strlen(lpHackName)+6];
    sprintf (lpFileName, "%sWW%s.ini", si->PluginDirectory, lpHackName);
    LPCSTR lpDefault="";
    LPSTR  lpReturnedString;

    // Check if the file exists
    if (_access(lpFileName, 0))
    {
        char t[1024];
        sprintf(t, "Unable to open ini-file: ÿc4%s", lpFileName);
        fep->GamePrintError(t);
        delete lpFileName;
        return NULL;
    }

    // Try getting the data in 1924-byte increments
    DWORD  alloc=0;
    DWORD  allocStep = 1024;

    while (TRUE)
    {
        alloc+=allocStep;
        lpReturnedString = new char[alloc];
        if (GetPrivateProfileString(lpSectionName, lpKeyName, lpDefault, lpReturnedString, alloc, lpFileName) != strlen(lpReturnedString)+1)
            break;
        delete lpReturnedString;
    }
    delete lpFileName;
    return lpReturnedString;
}
```

[그림 9] IniFileHandlers.cpp 파일의 GetHackProfileString 함수

```
// 다음의 GetFingerprint 라는 함수는 중요한 부분으로 꼭 이해하고 넘어가야 하기 때문에 추가적인 작동설명은 따로 할 것이다.
// 여기서는 GetFingerprint 라는 함수의 사용과 작동만 주석으로 추가한다.
// GetFingerprint 함수는 이름으로 알 수 있듯이 핑거프린팅을 하는 함수인데 우리가 환경설정파일에 지정한 바이너리스트링을 갖고 메모리에 로딩된
// 실행파일이나 DLL 이미지를 대상으로 일치하는 위치(주소)를 찾아내는 역할을 한다. 우리가 마치 울트라에디트 같은 HEX 편집기로 실행파일을 열고
// OPCODE(명령코드) 바이너리스트링을 적어서 위치를 찾아내는 것과 같은데 단지 수동이 아니라 지정한 값을 지가 알아서 자동으로 찾는 것이 다르다.
// 이 함수의 첫번째 인자인 D2HackIt 은 D2HackIt.ini 파일명의 확장자를 뺀 D2HackIt 이고 두번째 인자인 GamePrintStringLocation 문자열은
// 설정파일의 FingerprintData 섹션에 있는 키 이름을 지정하는 것이다. 만약 GetFingerprint 함수가 메모리에 로딩된 바이너리 이미지로부터
// 코드주소를 찾아내면 그 위치를 구조체 fps 의 변수인 fps.AddressFound 에 셋팅해서 리턴한다.

FINGERPRINTSTRUCT fps;
if(!GetFingerprint("D2HackIt", "GamePrintStringLocation", fps))
{
    MessageBox(NULL, "Fingerprint information for 'GamePrintStringLocation'Wnmissing or corrupt!", "D2Hackit Error!", MB_ICONERROR);
    return FALSE;
}

// 찾은 주소 fps.AddressFound 를 또 다른 구조체인 psi 의 GamePrintStringLocation 변수로 할당하고 있다.
psi->GamePrintStringLocation=fps.AddressFound;
if (!psi->GamePrintStringLocation)
{ MessageBox(NULL, "Unable to find entrypoint for 'GamePrintStringLocation'!", "D2Hackit Error!", MB_ICONERROR); return FALSE; }

// Get playerinfo struct
if (!GetFingerprint("D2HackIt", "pPlayerInfoStruct", fps))
{ fep->GamePrintError("Fatal error! Exiting!"); return FALSE; }

// 다음의 코드를 보면 D2HackIt 의 기교를 엿 볼 수 있는데 꼭 타켓 프로그램의 명령어를 후킹하는 것이 아니더라도 캐릭터 정보가 저장된 메모리에서
// 해커가 직접만든 구조체로 데이타를 읽어들일 수도 있다. 물론, 이렇게 하기 위해서는 리버싱을 통해서 구조체의 분석이 이미 끝나있어야 할 것이다.
// 그 이유는 구조체의 전체크기와 각 내부변수의 크기를 알지 못하면 엉뚱한 값을 구조체로 갖고오기 때문이다.
// Messy pointers :)
thisgame->player=(PLAYERINFOSTRUCT*)*(DWORD*)(*(DWORD*)fps.AddressFound);

// Get gameinfo struct
// 게임정보 구조체를 읽는 부분이 있어야 하는데 소스가 잘 못 됐는지 플레이어 구조체 읽는 부분이랑 중복되어있다.
if (!GetFingerprint("D2HackIt", "pPlayerInfoStruct", fps))
{ fep->GamePrintError("Fatal error! Exiting!"); return FALSE; }

// 참고로 다음의 그림은 D2HackIt.ini 환결설정파일이다.
// 우리는 범용적 사용을 목표로 하므로 결국 위에서 설명한 부분역시 쓰잘데기 없다는 것을 잊지말자. 디아블로2에서 찾은 주소들은 아무짝에도 쓸모가
// 없다. 기교만 흡수하자.
```

```
[FingerprintData]

; This is now GrabIt! compatible.
GamePacketReceivedIntercept=D2Client.dll,7,13,8B5C2410xxxxxxxxxxxx8D145B8B0495
GamePacketReceivedIntercept2=D2Client.dll,7,18,85C9894C24xxOF8Fxxxxxxxx FFO5xxxxxxxx5F5E5D

; This lets us send packets even in single player games
GamePacketSentIntercept=D2Net.dll,5,0,!10005

; We hook this to set playerinfo when game starts/ends
GamePlayerInfoIntercept=D2Client.dll,6,10,E8xxxxxxxxE8xxxxxxxx8935xxxxxxxx5EC3

; Just a pointer to playerinfo, available any time
pPlayerInfoStruct=D2Client.dll,0,12,E8xxxxxxxxE8xxxxxxxx8935xxxxxxxx5EC3

; This is out entrypoint for displaying text
GamePrintStringLocation=D2Client.dll,0,0,81ECxxxxxxxx53558BE956578A45xx84C0

; This is the struct passed to us by the loader
; The modules searched for are actually: Game.exe, Diablo II.exe and D2Loader.exe
LoaderStruct=Game.exe,0,0,1d10abd1

; This is used by bind.d2h
GameKeyDownIntercept=D2Client.dll,5,5,8B770833D2xxxxxxxxxxxxC0668B41043BC6

; Thanks to Techwarrior for this info
GameSendPacketToGameLocation=d2net.dll,0,21,81EC0C010000538B1Dxxxxxxxx5556

; This lets us save&exit game safely (by Dan_Goon)
GameSaveAndExit=D2client.dll,0,0,33c9e8xxxxxxxxc705xxxxxxxxxxxxxxxxe8xxxxxxxxc705

; Can't find it in 1.10
GameSendMessageToChat=bnclient.dll,0,0,8BD181EC????????85D274??578BFA83C9FF
```

[그림 10] D2HackIt.ini 설정파일 일부내용 (한국인으로 보이는 Dan_Goon 아이디가 눈에 뜀)

```
// 아래의 GamePrintInfo 함수는 이미 앞에서 보았을 것이다. 이 함수는 디아블로2 바이너리(정확히 D2Client.dll)의 채팅창 관련 함수를 강제로
// 호출하고 있다. 그림8 을 보면 GamePrintString 함수가 결국 GamePrintStringLocation 을 호출하는데 GamePrintInfo 역시 이 출력함수와 연관되어
// 있다. 그렇기 때문에 fep->GamePrintInfo(t) 명령은 디아블로2 게임의 채팅창에 출력하는 함수이다. 역시 쓰잘데기 없는 내용인데 이렇게 설명
// 하는 이유는 D2HackIt 의 기교를 습득하기 위해서 이고 또한 나중에는 이 함수를 다른 함수로 교체할 것이기 때문에 미리 설명한는 것이다.
t=new char[128];
sprintf(t, "Starting D2HackIt! Mk2 version %d.%.2d (%s@%s)", LOWORD(si->Version), HIWORD(si->Version), psi->BuildDate,
        psi->BuildTime);
fep->GamePrintInfo(t);

// 이 부분은 D2 로더를 사용해서 D2HackIt.dll 을 타켓 프로그램속에 로딩시켰을때 로더의 정보를 얻어내는 부분이다. 그러므로 기교는 흡수하되
// 우리가 D2 로더를 사용하지 않으므로 제거할 것이다.
BOOL UsingD2Loader=FALSE;
psi->DontShowErrors=TRUE;
if (!GetFingerprint("D2HackIt", "LoaderStruct", fps))
{
    psi->DontShowErrors=FALSE;
    sprintf(fps.ModuleName, "Diablo II.exe");
    UsingD2Loader=TRUE;
    if ((fps.AddressFound=GetMemoryAddressFromPattern(fps.ModuleName, fps.FingerPrint, fps.Offset)) < 0x100)
    {
        sprintf(fps.ModuleName, "D2Loader.exe");
        if ((fps.AddressFound=GetMemoryAddressFromPattern(fps.ModuleName, fps.FingerPrint, fps.Offset)) < 0x100)
        {
            //fep->GamePrintError("Unable to find loader data in 'Game.exe', 'Diablo II.exe' or 'D2Loader.exe'!");
            //fep->GamePrintError("Fatal error! Exiting!"); return FALSE;
            fps.AddressFound=0;
        }
    }
}

psi->loader = (LOADERDATA*)fps.AddressFound;

if (psi->loader)
{
    sprintf(t, "Loader version c4%d.%.2d c0Game is %sstarted with D2Loader.",
            LOWORD(psi->loader->LoaderVersion), HIWORD(psi->loader->LoaderVersion), (UsingD2Loader?"":"c4not c0"));
} else {
    sprintf(t, "D2Hackit was loaded without loader");
}
fep->GamePrintInfo(t);

// 채팅창에 GamePrintStringLocation 을 핑거프린팅으로 찾았다는 메시지 출력.. 역시 제거해야 하거나 교체해야 한다.
sprintf(t, "Found 'GamePrintStringLocation' at %.8x", psi->GamePrintStringLocation);
fep->GamePrintVerbose(t);

// 지금부터 이 ServerStart 함수 내에서 가장 중요한 부분이라고 할 수 있다.
// 설정파일에서 GamePacketReceivedIntercept 에 해당하는 바이너리스트링을 읽어들여 핑거프린트 함수로 주소를 찾으면
// psi->fps.GamePacketReceivedIntercept 변수에 저장한다.
if (!GetFingerprint("D2HackIt", "GamePacketReceivedIntercept", psi->fps.GamePacketReceivedIntercept))
{ fep->GamePrintError("Fatal error! Exiting!"); return FALSE; }
// 위에서 핑거프린트로 디아블로2의 패킷수신부 주소를 찾으면 해당 주소지점을 Intercept 함수로 가로채어 GamePacketReceivedInterceptSTUB 함수로
// 라우팅 시키는 코드이다. 실제적인 작동은 INST_CALL 에 해당하는 0xE8 1바이트와 4바이트 GamePacketReceivedInterceptSTUB 함수주소를 합친
// 총 5 바이트 크기의 명령을 AddressFound 주소지점에 덮어쓴다. 이때 덮어지기 전의 5 바이트 명령을 GamePacketReceivedInterceptSTUB 함수의
// 앞부분에 마련한 NOP 으로 채워진 빈 공간에 복사한다. 이때 PatchSize 는 덮어지는 위치(주소지점)의 명령어가 몇 바이트 명령어인지 지정한다.
// 자세한 설명은 함수와 그림으로 대체할 것이다.
Intercept(INST_CALL, psi->fps.GamePacketReceivedIntercept.AddressFound, (DWORD)&GamePacketReceivedInterceptSTUB,
          psi->fps.GamePacketReceivedIntercept.PatchSize);

// Get GamePacketSentIntercept
// 패킷송신부 핑거프린팅
if (!GetFingerprint("D2HackIt", "GamePacketSentIntercept", psi->fps.GamePacketSentIntercept))
{
    fep->GamePrintError("Fatal error! Exiting!");
    // 만약 패킷송신부를 찾다 실패하면 이전에 인터셉트한 부분을 다시 원상복귀 시켜줘야 하므로 위와 다르게 Intercept 함수에 넘겨지는 인자가
    // 반대로 되어있다.
    Intercept(INST_CALL, (DWORD)&GamePacketReceivedInterceptSTUB, psi->fps.GamePacketReceivedIntercept.AddressFound,
              psi->fps.GamePacketReceivedIntercept.PatchSize);
    return FALSE;
}
```

```
// 위와 마찬가지로 패킷송신부를 인터셉트한다.
Intercept(INST_CALL, psi->fps.GamePacketSentIntercept.AddressFound, (DWORD)&GamePacketSentInterceptSTUB,
          psi->fps.GamePacketSentIntercept.PatchSize);
// Get GamePlayerInfoIntercept
// 이하 GetFingerprint 와 Intercept 들도 마찬가지의 작업이다. 분석할 필요는 없다. 뒤에서 이 부분을 모두 삭제할 것이다.
if (!GetFingerprint("D2HackIt", "GamePlayerInfoIntercept", psi->fps.GamePlayerInfoIntercept))
{
    fep->GamePrintError("Fatal error! Exiting!");
    Intercept(INST_CALL, (DWORD)&GamePacketSentInterceptSTUB, psi->fps.GamePacketSentIntercept.AddressFound,
              psi->fps.GamePacketSentIntercept.PatchSize);
    Intercept(INST_CALL, (DWORD)&GamePacketReceivedInterceptSTUB, psi->fps.GamePacketReceivedIntercept.AddressFound,
              psi->fps.GamePacketReceivedIntercept.PatchSize);

    return FALSE;
}
Intercept(INST_CALL, psi->fps.GamePlayerInfoIntercept.AddressFound, (DWORD)&GamePlayerInfoInterceptSTUB,
          psi->fps.GamePlayerInfoIntercept.PatchSize);

// Get GameSendPacketToGameLocation
// Thanks to TechWarrior
if (!GetFingerprint("D2HackIt", "GameSendPacketToGameLocation", fps))
{
    fep->GamePrintError("Fatal error! Exiting!");
    Intercept(INST_CALL, (DWORD)&GamePacketSentInterceptSTUB, psi->fps.GamePacketSentIntercept.AddressFound,
              psi->fps.GamePacketSentIntercept.PatchSize);
    Intercept(INST_CALL, (DWORD)&GamePacketReceivedInterceptSTUB, psi->fps.GamePacketReceivedIntercept.AddressFound,
              psi->fps.GamePacketReceivedIntercept.PatchSize);
    Intercept(INST_CALL, (DWORD)&GamePlayerInfoInterceptSTUB, psi->fps.GamePlayerInfoIntercept.AddressFound,
              psi->fps.GamePlayerInfoIntercept.PatchSize);

    return FALSE;
}
psi->GameSendPacketToGameLocation=fps.AddressFound;

// 잘은 모르겠으나 클라이언트 정보를 주기적으로 뭔가 하려는 목적으로 쓰레드를 돌리는 것으로 파악된다. (아직까지는 우리가 하려는 것에 필요치
// 않다.)
// Start TickThread, We dont care about closing it later.
// It will be destroyed when unloading the dll.
DWORD dummy=0;
psi->TickThreadHandle = CreateThread(NULL,0,TickThread,(void*)&ClientList,THREAD_TERMINATE,&dummy);
psi->TickThreadActive=TRUE;

// 이하 디아블로2에 관련된 것이므로 필요없으므로 제거할 내용이므로 설명을 생략한다.
// Load any clients listed in Autorun
t=new char[1024];
t=fep->GetHackProfileString("D2HackIt", "Misc", "Autoload");

if (strlen(t))
{
    char* command[2];
    command[0]=".load";
    char *p;
    p=t;

    command[1] = p;
    while (*p != 0) {
        if (*p == ',') {
            *(p++) = 0;
            GameCommandLineLoad(command,2);
            while (*p == ' ') p++;
            if (*p != 0) command[1] = p;
        } else
            p++;
    }
    GameCommandLineLoad(command,2);
}

    }

    delete t;

    fep->GamePrintInfo("D2HackIt! Mk2 Loaded! Type .c4.help0 for help on commands.");

    return TRUE;
}

// 다음은 프로그램이 종료하면서 DLL 이 DETACH(언로드)될 때 실행되는 ServerStop 함수이다.

BOOL PRIVATE ServerStop(void)
{
    // Kill our tickthread
    // 위에서 생성한 틱(Tick) 쓰레드를 죽이는 짓인데 우리는 쓰레드를 만들지 않을것이므로 죽이지도 않을 것이다.
    TerminateThread(psi->TickThreadHandle, 1234);
    CloseHandle(psi->TickThreadHandle);

    // Unload any loaded clients
    // 클라이언트 정보를 해제하는데 우리는 이 부분이 필요치 않다. 디아블로2를 분석해야 뭔짓을 하는지 알 것 같다.
    while (ClientList.GetItemCount())
    {
        LinkedItem *li=ClientList.GetLastItem();
        CLIENTINFOSTRUCT *cds=(CLIENTINFOSTRUCT*)li->lpData;
        char t[32];
        sprintf(t, "unload %s", cds->Name);
        GameCommandLine(t);
    }

    // Un-patch intercept locations
    // 이 부분이 ServerStop 함수에서 가장 중요하다. 프로그램이 시작되고 Intercept 가 되었을 것이므로 다시 복구해줘야 한다. 그렇지 않으면 프로그램
    // 종료시 뻑난다. 어차피 프로그램 뻑나는데 익숙해져 있다면 굳이 언패치를 가할 필요는 없지만 디아블로2 핵킷을 만든 사람들처럼 배포할 목적이라면
    // 언패치를 해야 사용자들이 돌을 안던질 것이다.
```

```
    Intercept(INST_CALL, (DWORD)&GamePlayerInfoInterceptSTUB, psi->fps.GamePlayerInfoIntercept.AddressFound,
              psi->fps.GamePlayerInfoIntercept.PatchSize);
    Intercept(INST_CALL, (DWORD)&GamePacketSentInterceptSTUB, psi->fps.GamePacketSentIntercept.AddressFound,
              psi->fps.GamePacketSentIntercept.PatchSize);
    Intercept(INST_CALL, (DWORD)&GamePacketReceivedInterceptSTUB, psi->fps.GamePacketReceivedIntercept.AddressFound,
              psi->fps.GamePacketReceivedIntercept.PatchSize);

    // Release dll's that we loaded upon entry.
    // 강제로 로딩했던 DLL 들을 해제시켜준다.
    for (int i=0; NeededDlls[i] != NULL; i++) FreeLibrary (GetModuleHandle(NeededDlls[i]));

    // 언로드 되었다고 디아블로2 채팅창에 출력
    fep->GamePrintInfo("D2HackIt! Mk2 Unloaded.");
    // 메모리를 할당한 변수들을 제거
    delete (LPSTR)si->PluginDirectory, (LPSTR)psi->IniFile, thisgame, pfep, fep, psi, si;

    return TRUE;
}
```

이제 ServerStartStop.cpp 파일을 대략적으로 훑어 봤는데 감이 잡힐 것이다. 우리가 실질적으로 필요한 루틴은 사실상 따져보면 GetFingerprint 함수와 Intercept 함수 밖에 없다. 독자는 이 두 함수를 먼저 분석을 한 뒤에 다음 부분으로 넘어가길 바란다. 또한 나머지 함수들이나 구조들도 독자가 시간적여유가 된다면 분석을 해보길 추천한다. 그래야만 이해의 깊이가 넓어지며 추후에라도 버그를 잡는다던지 기능을 확장한다던지 할 수 있는 이해능력이 생길 것이다.

실전 Lesson-1 : 온라인 게임 채팅창후킹

이제부터 우리는 실전에서 이 기술을 배울 것이다. 후킹대상을 선정하기위해서 고민을 많이 했는데 무엇보다 가장 많이 접하는 온라인 게임이 좋을 것 같았다. 필자가 온라인 게임을 대상으로 삼은 것은 넓은 의미로 봤을때 뭔가 그럴듯한 프로그램을 대상으로 실제로 후킹을 써먹을 수 있다는 확신을 갖게해주기 위함이며 좁은 의미로 본다면 온라인 게임의 채팅창을 가로채서 후킹기술을 설명하는데 그 목적이 있다. 최소한 이 정도를 해내면 앞으로 Windows 에서 후킹에 관해서는 두려울 것이 없을 것이다.

다음의 그림11 은 필자가 이 문서를 쓰기위해 일부러 시간을 투자하여 국내 최신 온라인게임을 리버싱한 데이타이다. (게임이름은 문제의 소지가 될 수 있으므로 거론하지 않으며 뒤에서도 삭제하였다. 관심있으면 찾아보길...) 우리는 이 온라인 게임의 채팅창을 후킹하여 기술을 습득할 것이므로 게임을 미리 리버싱해서 각 루틴들의 데이타베이스를 그림11 과 같이 최대한 많이 구축해 놓는 것이 필수이다.

그렇다면 어디서부터 시작해야 하는가?
이 세상의 모든 만물은 시작점(EntryPoint:도입부)이 있고 뿌리(root)란 것이 있기 마련이다. 고로 어디서부터 시작해야 할지 개념을 잡는 것이 가장 중요하다. 우리는 먼저 게임을 리버싱한 후 리버싱 데이타베이스를 참고로 후킹을 시도할 것이므로 리버싱 도중 채팅창루틴을 찾아내는 것이 가장 중요하다고 할 수 있다. 채팅창루틴 찾기가 후킹을 하기위한 시작점이 되는 것이다.(채팅창루틴도 못 찾으면 후킹을 어디에 걸 수 있겠는가? 지금 설명하는 후킹방법은 API 후킹이 아니다!!) 채팅창루틴을 찾았다면 후킹을 걸어서 채팅창의 역할을 개조할 것이고 게임 플레이 시 입력한 내용을 가로채고 자신만의 채팅커맨드를 만들어 볼 것이다. 혹시 채팅창에 대해서 모르는 독자가 있지는 않겠지만 그래도 굳이 설명하자면 채팅창은 첫번째로 입력창이 있어서 입력하는 글자가 보여지고 두번째로는 엔터를 쳤을때 입력한 글자가 뿌려지는 출력창으로 구성된다. 입력부와 출력부의 두가지가 하나의 채팅창이 되는 것이다. 또한, 채팅창에 입력한 채팅 메시지는 결국 서버로 전송되도록 구성된다. 이러한 구성의 채팅창을 후킹하여 자신만의 채팅커맨드를 만들게 된다면 채팅창 출력부분에 레지스터 상태를 출력시키는 것도 할 수 있다. 예를들어 #debug 라고 입력하면 채팅창 출력부분에 레지스터 상태를 출력시키도록 만들 수 있는 것이다. 이때 주의할 점은 서버로 채팅 메시지가 전송되기 때문에 주변의 게임을 플레이하는 유저들이 모두 다 보게 될 것이다. (이건 아니잖아.. 정상적인 유저들을 방해하는 건 우리의 목적이 아니었으니까..) 이러한 문제점을 극복하면서 채팅창을 순조롭게 후킹하게 되면 게임을 하면서 채팅창으로 디버깅을 할 수 있는 환경이 마련된다. 디버깅 뿐만 아니라 채팅창에 명령어를 입력하면서 동적으로 게임의 루틴을 호출하거나 데이타를 변경하거나 또는 추가 후킹 플러그인을 로딩해서 확장할 수 도 있다.

자.. 좀 지겨울지도 모르지만 끝까지 참고서 Lesson-1 을 마스터하길 바란다. 뒷부분에서는 그동안 많은(?) 후킹문서가 존재했지만 한번도 제대로 다루지 않았던 후킹시 치명적 결함을 피하기 위한 스택정리 방법 등의 추가정보를 얻게 될 것이다.

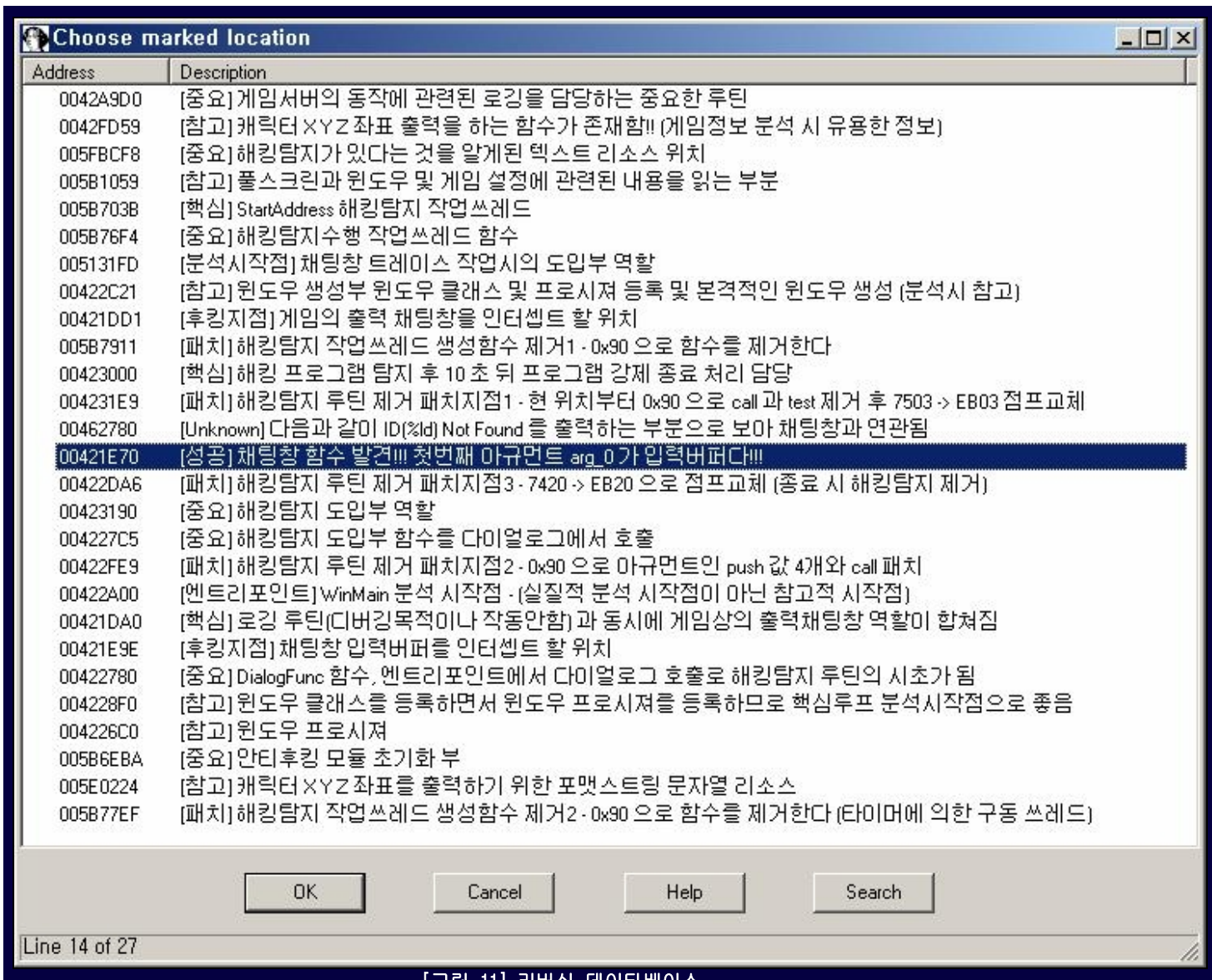
복병이 나타났다?!

먼저 후킹에 본격적으로 돌입하기도 전에 복병이 나타나서 방해공작을 하는 바람에 이 복병부터 때려잡아 놓고 후킹에 들어갈 것이다. 이 바닥에서 프로그래밍을 하거나 리버싱을 하는 사람들은 복병의 무서움을 잘 아실거라 생각한다. 꼭 잘 나가다가 발목을 붙잡고 안놓는 문제점들이 생기게 된다. 그렇다면 이 게임에서 복병은 어떤 놈일까? 필자가 이 게임을 후킹을 하면서 제일먼저 맞닥드린 복병 중 첫번째는 바로 해킹방지 장치다. 이 해킹방지 장치 때문에 처음부터 디버깅이 안되는 증상을 보였다.(소프트아이스 및 기타 디버그모드 탐지함) 또한 PE 바이너리의 무결성(해쉬값 이용함)을 검사하고 메모리의 악성 프로그램을 감시하며 제일 심각한 건 런타임코드 영역이 변조되는 것과 API 후킹을 탐지하여 10초 후 프로그램을 강제종료 시켜버린다.(기타 프락시체크와 스피드핵 탐지도 하고 있었음) 이미 알고있는 것처럼 우리가 하려는 작업은 런타임코드 영역을 바꾸게 되기 때문에 분명히 이 게임을 실행하고 DLL 을 인젝션하면 그 순간 프로그램이 종료될 것이다. 이 게임의 경우 다른 회사와 달리 별도의 해킹방지 솔루션이 모듈로써 탑재된 것이 아니라 자체 보안팀에서 개발한것 같은데 10초의 여유를 두고 종료시켜버리는 독특한 방어방식이 돋보였다(?)
필자가 작업에 걸린 소요시간은 약 3일간 리버싱하였으며 실제적인 시간으로 따진다면 약 이틀정도 걸렸다. 이 중 해킹탐지 보안루틴은 약 4~5시간만에 무장해제 시킬 수 있었다.<2> 참 어이없게도 두 군데의 작업쓰레드 생성을 막으니 해킹탐지 10초 후 프로그램의 자동종료를 막고 필자의 DLL 을 런타임때 인젝션하는것이 가능했다. 참고로 IDA Pro 4.9 버전을 사용하였으며 디버거를 사용할 필요도 없이 역어셈블된 내용만으로 가능한 작업이었다.

보안루틴을 제거한 후에는 채팅창 후킹함수가 예상치 못한 오류를 발생할 때마다 잠깐씩 올리디버거(OllyDbg)로 오류가 난 지점을 확인하는 정도로 디버거를 사용하였으며 올리디버거를 켜놓고 무작정 트레이싱(Tracing:추적)하는 작업은 하지 않았다. 올리디버거는 심볼이 약해서 IDA 보다 읽기 어려운 단점이 있고 반면 IDA 로 디버깅하는 것은 PC 가 아무리 빨라도 엄청난 인내심을 견뎌야 겨우 화면이 뜰 정도이기 때문에 디버거에 목숨걸고 작업하지는 않았다.
(필자는 디버거를 최소한으로 사용하는 것이 가장 좋은 리버싱 방법이라고 생각한다. 왜? 요즘처럼 이른바 원칩이라 불리는 CPU 가 탑재되는 전기밥솥 속에서 디버거를 돌릴 수 있다면 디버거만 파겠지만 언제나 좋은환경 속에서 리버싱을 할 거라고 판단하면 큰 오산이며 발전가능성이 없다고 생각하기 때문이다. 하드웨어 해킹까지 뛰어난 사람이 아니라면 디버거에 의존도를 크게 둘 때 분명히 한계가 빨리 찾아올 것이다. 하드웨어 해킹을 배우지 않고 다양한 시스템과 맞상대하려면 어셈블리어와 아키텍쳐 분석에 강해지는 수밖에 없다고 생각한다.)
결국 보안루틴의 경우 보안 프로그래머의 프로그래밍 상식이 전부 읽혔기 때문에 어이없이 무너졌는데 아무리 보안 프로그래머라고 하더라도 정석적으로 개발을 시도할 것이므로 100명을 데려다 놓고 프로그래밍을 시키면 100명이 전부 쓰레드를 사용한 탐지루틴을 작성할 것이다. 실시간 해킹탐지에 쓰레드를 사용하지 않는다면 무엇으로 실시간 체크가 가능하겠는가? 그러므로 보안 프로그래머는 일반 프로그래머와 똑같이 생각해서 전부 해커에게 읽혀버리는 치명적 실수를 범하면 안된다.
<아마 중국애들은 1시간 내외면 가능한 작업이 아닐까...>

<2> 필자는 다른 게임에서도 디아블로2 처럼 적용이 된다는 것을 실례로 보이고 싶었을 뿐인데 의외로 보안루틴이 걸려있어서 오기가 발동해서 해제했지만 어떤 악의적인 목적은 없다는 것을 밝힌다. 이 문서에서 설명하고자 하는 것이 후킹기술을 실제로 사용하는 것을 보이기 위함이기 때문에 그럴듯한 대상이 필요했을 뿐이다. 리버싱은 매우 고통스럽고도 지겨운 작업이기 때문에 누군가 이를 악용해서 뭔가 더 하고자 한다면 분명 그 답례를 치르게 될 것이다. (원형탈모 등등... 지금 보는 예보다 한보 더 나가서 게임의 로직을 바꾸는 것이 그리 만만한 작업이라고 생각하면 오산이라는 소리다...
수치바꾸기 따위는 언급하지 않는다.)

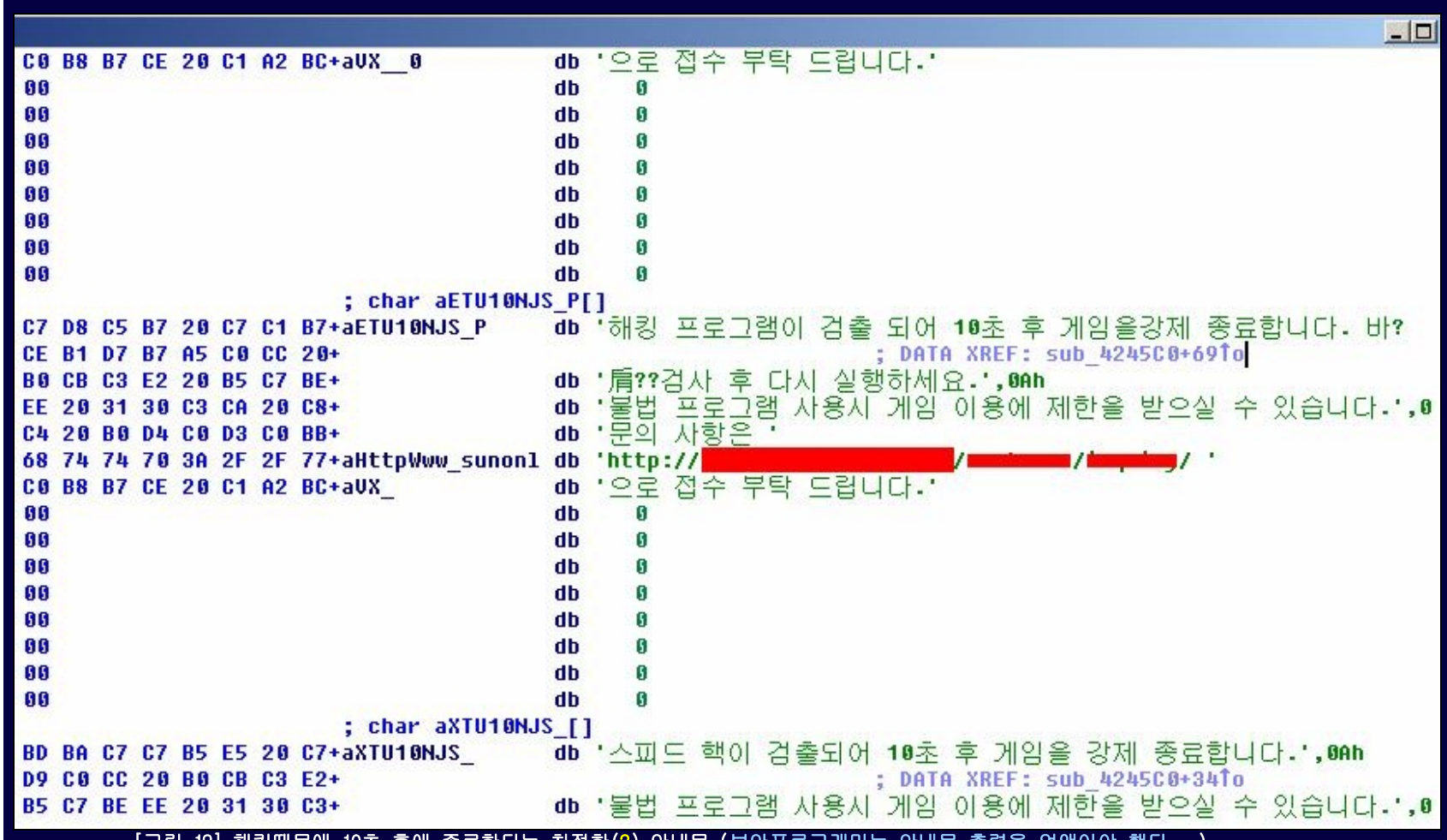
자.. 그럼 복병인 보안장치를 해제하는 것을 시작으로 실전에 들어가보자..



[그림 11] 리버싱 데이타베이스

위의 그림11은 필자가 약 3일동안 채집한(?) 리버싱 데이타베이스이며 IDA Pro 4.9 버전으로 작성되었다. 주소가 다 보이고 있지만 이 게임은 오픈베타이기 때문에 계속 바이너리가 업데이트 되어 지금은 위의 주소와 완전 딴판이 되었을 것이다. 먼저 게임을 실행시키고 위에서 찾은 리버싱 데이타베이스를 기준으로 후킹을 시도해 봤으나 게임 메인화면에 들어가자 해킹이 탐지되어 10초후에 프로그램을 종료한다는 메시지가 출력되며 로그인 하기도 전에 프로그램이 종료되는 현상이 나타났다. 10초가 아니라 10분이면 채팅창 가로채는 것만 간단히 보여주고 게임을 종료하는데 5분이면 충분하니까 보안루틴까지 건드리지 않아도 널널한 시간인데 10초는 너무도 짧았다. 그래서 일단 보안루틴부터 해제해야할 필요가 있었으며 분석작업을 하여 스트링에서 다음과 같은 부분을 찾을 수 있었다.

(참고: 여기서 보이는 한글은 원래 IDA 에서는 깨져서 보일 것이다. 직접 문자열 범위를 선택해서 키보드 'A' 키를 눌러서 유니코드 스트링으로 수동변환을 해야한다. IDA 는 영문자로 시작되는 문자열 중간에 한글이 섞여 있을 경우 이 전체적인 문자열을 유니코드로 인식하지 못해서 완벽하게 한글로 자동변환하지 못하는 문제가 있다. - 자동변환 플러그인을 새로 만들면 될 것 같기도 하지만 필요한 사람이 만들어 써야할 것 같다 -)



[그림 12] 해킹때문에 10초 후에 종료한다는 친절한(?) 안내문 (보안프로그래머는 안내문 출력을 없앴어야 했다...)

위의 화면은 IDA 로 타켓이 된 온라인 게임을 역어셈블시킨 화면이다. 그리고 그림12 는 이 바이너리의 데이타영역의 리소스(데이타)들을 뿌려주고 있다. 실제 실행되는 게임코드들은 데이타 영역에 초기화된 스트링을 사용하는데 자세한 것은 실행파일 구조와 컴파일쪽을 공부하면 배울 수 있으므로 여기서는 생략한다. 중요한 것은 스트링이 저장된 데이타 영역에서 힌트를 얻어서 리버싱 도입부가 될 지점을 찾아야 한다는 것이다. 일단 위의 화면에서는 "해킹 프로그램이 검출되어 10초 후 게임을 강제 종료합니다" 라는 문자열을 sub_4245C0+69 루틴에서 참조하고 있다는 것을 찾아낸 화면이다. 참조하는 루틴으로 이동하면 아래와 같이 해킹탐지에 따른 각각의 출력메시지를 끌어다가 버퍼에 담는 루틴이 존재하는 것을 볼 수 있다.

```
8D 4C 24 04                  lea     ecx, [esp+808h+var_804]
68 58 EA 5E 00               push    offset aRBTBTU10 ; "게임 실행에 영향을 줄 수 있는 프로세스(
51                           push    ecx             ; char *
E8 10 36 18 00               call    _sprintf
56                           push    esi
8D 54 24 10                  lea     edx, [esp+814h+var_804]
52                           push    edx
6A 03                        push    3
EB 97                        jmp     short loc_424607
                 ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴(

                 loc_424670:                            ; CODE XREF: sub_4245C0+29↑j
                                                        ; DATA XREF: .text:004246D8↓o
8D 44 24 04                  lea     eax, [esp+808h+var_804]
68 68 E9 5E 00               push    offset aRBTER10 ; "게임 실행에 영향을 줄 수 있는 프로그램?
50                           push    eax             ; char *
E8 F7 35 18 00               call    _sprintf
56                           push    esi
8D 4C 24 10                  lea     ecx, [esp+814h+var_804]
51                           push    ecx
6A 04                        push    4
E9 7B FF FF FF               jmp     loc_424607
                 ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴(

                 loc_42468C:                            ; CODE XREF: sub_4245C0+29↑j
                                                        ; DATA XREF: .text:004246DC↓o
8D 54 24 04                  lea     edx, [esp+808h+var_804]
68 90 E8 5E 00               push    offset aEBTU10NJS_ ; "게임 데이타 조작시도가 검출되어 10초
52                           push    edx             ; char *
E8 DB 35 18 00               call    _sprintf
56                           push    esi
8D 44 24 10                  lea     eax, [esp+814h+var_804]
50                           push    eax
6A 05                        push    5
E9 5F FF FF FF               jmp     loc_424607
                 ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴(

                 loc_4246A8:                            ; CODE XREF: sub_4245C0+29↑j
                                                        ; DATA XREF: .text:004246E0↓o
8B 4E 04                     mov     ecx, [esi+4]
51                           push    ecx
E8 B2 08 1A 00               call    sub_5C4F63
8B 8C 24 08 08 00 00         mov     ecx, [esp+80Ch+var_4]
83 C4 04                     add     esp, 4
5E                           pop     esi
E8 3E 36 18 00               call    sub_5A7(
81 C4 04 08 00 00            add     esp, 80(
C3                           retn
                 sub_4245C0  endp

                 ; 컴컴컴컴컴컴컴컴컴컴컴컴컴
F0 45 42 00      off_4246C8  dd offset loc_42
25 46 42 00                  dd offset loc_42
```

```
; 栢栢栢栢栢栢栢?S U B R O U T I N E 栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢

sub_5C4F63      proc near               ; CODE XREF: sub_4245C0+EC↑p
                                        ; StartAddress+70D↑p

arg_0           = dword ptr  4

                mov     eax, [esp+arg_0]
                cdq
                push    0Dh
                pop     ecx
                idiv    ecx
                mov     eax, edx
                retn
sub_5C4F63      endp
```

[그림 13] StartAddress 를 보라.

위의 화면에서 프로시져의 마지막부분에 노란색으로 칠해진 sub_5C4F63 함수를 보자. 이 함수를 들여다보면 CODE XREF 부분에 StartAddress 가 보일 것이다. 경험많은 리버스 엔지니어라면 보자마자 쓰레드라고 판단할 것이다. 위 화면에서 loc_4246A8 루틴에서 sub_5C4F63 함수가 호출되고 있고 함수 안에서도 쓰레드로 참조되고 있다는 것은 결국 쓰레드로 돌렸다는 힌트이다. 위에서 어쨌든 빨간색으로 표시한 참조위치인 StartAddress+70 으로 가서 마우스 가운데 버튼을 아래위로 살살 보듬어가면서 재밌는 메시지들을 발견해보자.

```
                 loc_5C48FD:                            ; CODE XREF: StartAddress+320↑j
F6 85 8C FA FF FF 02         test    byte ptr [ebp+var_574], 2
0F 84 CC 00 00 00            jz      loc_5C49D6
56                           push    esi             ; char
68 50 FA 60 00               push    offset aStep4   ; "   # STEP 4++"
53                           push    ebx             ; int
53                           push    ebx             ; int
E8 93 0A 00 00               call    sub_5C53AA
68 18 FA 60 00               push    offset aE_0     ; "[*] 런타임 프로세스로부터 악성프로그램?
E8 9C 25 FE FF               call    nullsub_4
88 9D C0 FC FF FF            mov     byte ptr [ebp+var_340], bl
6A 41                        push    41h
```

[그림 14] 내가 인젝션 할 DLL 이 악성프로그램이라고? 컴파일러라도 장착했나보지 아님 인공지능?

참조하고 있는 위치에는 쓰레드 메인루틴이 있고 악성 프로그램 탐지에 관련된 함수들을 호출하는 관련 루틴이 하나둘씩 나타나고 있음을 알 수 있다.
내용을 좀 더 위로 올려보면?

```
                        loc_5C483F:                             ; CODE XREF: StartAddress+278↑j
56                                  push    esi                 ; char
68 E4 FA 60 00                      push    offset aStep2_0 ; "  # STEP 2--"
53                                  push    ebx                 ; int
53                                  push    ebx                 ; int
E8 5E 0B 00 00                      call    sub_5C53AA
83 C4 10                            add     esp, 10h

                        loc_5C484F:                             ; CODE XREF: StartAddress+217↑j
F6 85 8D FA FF FF 01                test    byte ptr [ebp+var_574+1], 1
0F 84 A1 00 00 00                   jz      loc_5C48FD
56                                  push    esi                 ; char
68 D4 FA 60 00                      push    offset aStep3   ; "  # STEP 3++"
53                                  push    ebx                 ; int
53                                  push    ebx                 ; int
E8 41 0B 00 00                      call    sub_5C53AA
68 98 FA 60 00                      push    offset aXKLSI   ; "[*] 런타임 코드 영역의 무결성이 훼손되?
E8 4A 26 FE FF                      call    nullsub_4
83 C4 14                            add     esp, 14h
FF 15 E8 C0 5E 00                   call    ds:__imp_GetCurrentProcessId
50                                  push    eax
E8 22 2A 00 00                      call    sub_5C72A4
```

[그림 15] 런타임 코드 영역이 순결을 체크하고 있는듯 하다. 요즘이 어떤 세상인데..

마찬가지로 런타임 코드 영역의 무결성 체크를 하는 함수도 호출하고 있는 듯하다.. 스텝 별로 처리를 쓰레드 안에 넣어놓은 것으로 알 수 있다.
좀 더 올려서 쓰레드 프로시져의 시작점으로 가보자!

```
                        ; Attributes: bp-based frame

                        ; DWORD __stdcall StartAddress(LPVOID)
                        StartAddress    proc near               ; DATA XREF: sub_5C4D3A+35↓o

                        var_5F4         = dword ptr -5F4h
                        var_5F0         = dword ptr -5F0h
                        var_5EC         = dword ptr -5ECh
                        var_5E8         = dword ptr -5E8h
                        var_5E4         = dword ptr -5E4h
```

[그림 16] 프로시져의 시작부에 StartAddress(LPVOID) 와 어디서 참조하고 있는지가 중요하다

누차 언급하는 것이지만 StartAddress(LPVOID) 만으로 쓰레드라는 것을 알 수 있다. 왜냐면 역공학 이전에 정공학 즉, 프로그래밍을 공부하면서 패턴을 알게되기 때문이다. 그리고 수많은 프로그램을 역어셈블 해보면 하나같이 쓰레드는 위와 같이 StartAddress 가 존재한다. 진짜 숙련된 리버스 엔지니어라면 처음부터 크로스 레퍼런스를(XREF) 따라가지 않고 다음과 같이 텍스트 검색으로 StartAddress 를 찾았을지도 모른다. 어쨌든 이제 그림16 에서 보는 것처럼 이 쓰레드를 참조하는 놈이 어디있는지 그곳으로 추적해보자.. DATA XREF: sub_5C4D3A+35 위치로 가면된다.

Occurences of: Start

Address	Instruction	
.text:005C4B26	loc_5C4B26:	; CODE XREF: StartAddress+5DE↑j
.text:005C4B43	loc_5C4B43:	; CODE XREF: StartAddress+5D5↑j
.text:005C4BA5	loc_5C4BA5:	; CODE XREF: StartAddress+566↑j
.text:005C4BC2	loc_5C4BC2:	; CODE XREF: StartAddress+610↑j
.text:005C4BD5	loc_5C4BD5:	; CODE XREF: StartAddress+582↑j
.text:005C4CE2	loc_5C4CE2:	; CODE XREF: StartAddress+71B↑j
.text:005C4CFD	StartAddress endp	
.text:005C4D6F	68 36 45 5C 00	push offset StartAddress ; lpStartAddress
.text:005C4EDA	68 5C FE 60 00	push offset aCarmorkitStart ; "CArmorKit::Start"
.text:005C4EEF	68 28 FE 60 00	push offset aAkStarted ; "ak started >> "
.text:005C4F0E	68 5C FE 60 00	push offset aCarmorkitStart ; "CArmorKit::Start"
.text:005C4F63		; StartAddress+70D↑p
.text:005C50BC	; START OF FUNCTION CHUNK FOR sub_5C524A	
.text:005C6031	; START OF FUNCTION CHUNK FOR sub_5D2BFA	
.text:005C645B	sub_5C645B proc near	; CODE XREF: StartAddress+5C2↑p
.text:005C6EEE	sub_5C6EEE proc near	; CODE XREF: StartAddress+4CD↑p
.text:005C70FB	sub_5C70FB proc near	; CODE XREF: StartAddress+410↑p
.text:005C72A4	sub_5C72A4 proc near	; CODE XREF: StartAddress+347↑p
.text:005C76DF	sub_5C76DF proc near	; CODE XREF: StartAddress+26D↑p
.text:005C77E5	sub_5C77E5 proc near	; CODE XREF: StartAddress+13E↑p

Line 242 of 308

[그림 17] 처음부터 lpStartAddress 로 텍스트를 검색할 수 도 있다.

텍스트로 검색을 하던 역으로 추적하던 결국 다음의 부분에 당도할 수 있다. 여기서는 참조 부분을 유심히 보자.

```
                        ; 栢栢栢栢栢栢栢?S U B R O U T I N E 栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢栢
                        ; Attributes: bp-based frame

                        ; int __fastcall sub_5C4D3A(LPVOID lpParameter)
                        sub_5C4D3A      proc near               ; CODE XREF: fptc+15↓p
                                                                ; sub_5C4EB3+9A↓p

                        ThreadId        = dword ptr -94h
                        var_90          = dword ptr -90h
                        var_8C          = dword ptr -8Ch
                        var_88          = dword ptr -88h
                        var_84          = byte ptr -84h
                        var_4           = dword ptr -4

55                                      push    ebp
8D 6C 24 88                             lea     ebp, [esp-78h]
81 EC 94 00 00 00                       sub     esp, 94h
A1 84 36 62 00                          mov     eax, dword_623684
56                                      push    esi
57                                      push    edi
33 FF                                   xor     edi, edi
8B F1                                   mov     esi, ecx
47                                      inc     edi
39 BE 24 02 00 00                       cmp     [esi+224h], edi
89 45 74                                mov     [ebp+74h], eax
89 BE 24 02 00 00                       mov     [esi+224h], edi
74 45                                   jz      short loc_5C4DA7
FF 86 28 02 00 00                       inc     dword ptr [esi+228h]
8D 45 E4                                lea     eax, [ebp-1Ch]
50                                      push    eax             ; lpThreadId
6A 00                                   push    0               ; dwCreationFlags
56                                      push    esi             ; lpParameter
68 36 45 5C 00                          push    offset StartAddress ; lpStartAddress
6A 00                                   push    0               ; dwStackSize
6A 00                                   push    0               ; lpThreadAttributes
FF 15 FC C1 5E 00                       call    ds:CreateThread
8B F0                                   mov     esi, eax
83 FE FF                                cmp     esi, 0FFFFFFFFh
75 19                                   jnz     short loc_5C4D9E
68 54 FD 60 00                          push    offset aCarmorkitDoonc ; "CArmorKit::DoOnce"
68 24 FD 60 00                          push    offset aSCreatethreadG ; "%s: CreateThread() 함수 호출에 실패했습"...
57                                      push    edi             ; int
E8 6D 03 00 00                          call    sub_5C5102
50                                      push    eax
E8 22 21 FE FF                          call    nullsub_4
83 C4 10                                add     esp, 10h
```

[그림 18] 실제 작업쓰레드 StartAddress 를 생성하는 함수

자.. 이제 그림18 을 보면 중간지점에 CreateThread 로 쓰레드를 생성하는 부분을 볼 수 있다. 이때 쓰레드로 작동시킬 함수로써 그림16 에서 봤던 StartAddress 함수를 인자로 넘겨주고 있다. 바로 우리가 역추적한 함수이다. 또하나 흥미있는 점은 아머킷이라고 보이는데 아마도 이게 실제의 보안루틴이름이자 클래스일 것이며 쓰레드 생성루틴을 통해서 작동될 것이라는 추측이 가능하다. 그리고 StartAddress 쓰레드가 제대로 생성되었는지 아닌지 체크하는 부분이 바로 이어지고 있다는 것을 "CreateThread() 함수호출에 실패했습니다" 라는 문자열을 보고 알 수 있다. 이쯤되면 이 쓰레드 생성루틴을 아예 없애면 보안루틴이 해제되지 않겠냐는 생각을 하게될 것이다. 딩동댕! 맞았다! "그렇다면 어느 부분에서 쓰레드생성을 어떤식으로 막아버리는게 가장 좋을 것인가?" 라는 판단을 해야한다. 한마디로 어딜 고칠것인가라는 질문인 것이다. 여기서는 위 그림18 의 함수를 호출하는 곳을 찾아서 수정할 것인데 그 이유는 결국 그림18 에서 보는 int __fastcall sub_5C4D3A(LPVOID lpParameter) 함수가 쓰레드를 생성하는 CreateThread 호출을 포함하고 있기 때문에 StartAddress 를 고치는 것보다 좀더 포괄적으로 수정되기 때문이다. (아마 프로그래머이고 쓰레드를 생성해본 경험이 있다면 위 그림18 의 함수가 Static 함수라는 것을 알 것이다...) 그런데 이러한 쓰레드 제거작업을 하기전에 일단 전제를 걸어야 한다. 위 그림18 의 sub_5C4D3A(LPVOID lpParameter) 함수가 게임의 다른 로직에 영향을 미치지 않는다는 전제하에 호출하는 곳으로 가서 제거를 해야한다. 다시 말하면 sub_5C4D3A(LPVOID lpParameter) 함수가 해킹방지 쓰레드만 돌리느냐 아니면 게임의 그래픽이나 사운드나 기타 게임운영에 관련된 루틴들도 가지를 뻗치고 있느냐를 생각해야 한다는 것이다. 여기서는 단지 보안루틴만 쓰레드로 돌릴것이라는 가정을 하고 이 함수를 호출하는 곳을 따라가 제거할 것이다. 위의 sub_5C4D3A(LPVOID lpParameter) 함수의 오른쪽에 CODE XREF 라고 된 곳을 보면 두군데서 이 함수를 호출하고 있는 것을 볼 수 있다.
(참고사항: 원래 IDA Pro 의 기본 셋팅이 크로스 레퍼런스가 2개만 출력되도록 되어 있기 때문에 2군데에서 호출되고 있다고 생각하면 오산이다. 3군데 이상에서 호출하고 있어도 2개만 표시되기 때문이다. 그래서 필자는 IDA Pro 의 셋팅을 크로스 레퍼런스가 5 개까지 표시하도록 해놓았다. 그래서 위의 두군데에서 호출하고 있는 CODE XREF 표시는 진짜 두군데에서만 호출하는 것으로 믿어도 좋다.)

이제 보안방지 쓰레드를 생성하려고 호출하는 두개의 지점이 어떻게 호출하고 있는지 직접 찾아가는 것이다.
위의 그림18 에서 빨간색 동그라미로 표시한 부분을 보라. 호출하고 있는 곳이 두군데 밖에 안되기 때문에 슬렁슬렁 둘다 훑어보자.

```
                        ; TIMECALLBACK fptc
                        fptc            proc near               ; DATA XREF: sub_5C4E43+32↓o

                        lpParameter     = dword ptr  0Ch

68 78 FD 60 00                          push    offset aDobytimerproc ; "DoByTimerProc"
68 68 FD 60 00                          push    offset aSTrace  ; "%s >> TRACE\n"
E8 88 20 FE FF                          call    nullsub_4
59                                      pop     ecx
59                                      pop     ecx
8B 4C 24 0C                             mov     ecx, [esp+lpParameter] ; lpParameter
E8 FA FE FF FF                          call    sub_5C4D3A
C2 14 00                                retn    14h
                        fptc            endp
```

[그림 19] 두군데 XREF 중 첫번째

자.. 두군데 참조 중에서 첫번째로 추적을 들어가면 그림19 의 루틴이 나타난다. 중간 부분에 call sub_5C4D3A 와 같이 그림18 의 함수를 호출하고 있다. 대략적인 추측으로 그림19 의함수는 타이머 프로시저로부터 호출되어 쓰레드가 생성되는 것 같았고 게임틱(Tick) 과 연관되어 있는 것으로 추정되었다. 어쨌든 일단 제거지점 1순위로 눈도장을 찍어놓았다.

```
8B CE                           mov     ecx, esi
E8 FA FE FF FF                  call    sub_5C4E43
EB 07                           jmp     short loc_5C4F52
                ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴

                loc_5C4F4B:                             ; CODE XREF: sub_5C4EB3+8A↑j
8B CE                           mov     ecx, esi        ; lpParameter
E8 E8 FD FF FF                  call    sub_5C4D3A

                loc_5C4F52:                             ; CODE XREF: sub_5C4EB3+8D↑j
                                                        ; sub_5C4EB3+96↑j
33 C0                           xor     eax, eax
40                              inc     eax
89 86 44 02 00 00               mov     [esi+244h], eax
A3 30 D6 65 00                  mov     dword_65D630, eax
```

[그림 20] 두군데 XREF 중 두번째

다시 그림18 의 함수로 돌아가서 두번째 참조지점으로 추적을 하면 위와 같은 부분으로 바로 갈 것이다. 그림20 의 중간에 call sub_5C4D3A 와 같이 그림18 의 함수를 호출한다. 이 부분은 그냥 대강 훑어봐도 일반적인 프로그래밍 흐름상의 호출패턴으로 구성된 무난한 어셈블리들을 보게 되는데 첫번째에서 봤던 타이머 쪽하고는 다른 모습으로써 일반적인 구조로 보인다는 뜻이다. 우리는 이미 타이머 쪽에서 첫번째 제거지점을 마음먹었으니까 마지막 두번째 제거지점을 그림20 에서 보는 call sub_5C4D3A 로 할 수 있다. 그렇지만 단순히 이렇게 한다는 것은 위험한 생각이다. 앞서 그림18 의 쓰레드 제거를 설명때 포괄적인 부분을 제거해야 한다고 말한 것을 잊으면 안된다. 즉, 우리는 달랑 call sub_5C4D3A 명령만 제거할 수 있지만 그렇게 하지않고 좀 더 추적해 들어가서 포괄적인 부분에서 제거시킬 것이다. 그러므로 그림20 의 함수부에서 다시 마우스로 살살 올려서 프로시저의 시작점으로 가보자.

```
                sub_5C4EB3      proc near               ; CODE XREF: sub_4243D0+67↑p
                                                        ; sub_424750+59↑p
56                              push    esi
57                              push    edi             ; char
33 FF                           xor     edi, edi
39 3D 30 D6 65 00               cmp     dword_65D630, edi
8B F1                           mov     esi, ecx
74 08                           jz      short loc_5C4EC9
33 C0                           xor     eax, eax
40                              inc     eax
```

[그림 21] 그림20 의 해당루틴에서 프로시져 시작점으로 간 화면 (함수시작 맨 처음이라고 보면 됨)

그림20 의 해당루틴에서 프로시져 시작점으로 가니 또다시 그림21 처럼 두갈래 길이 나타나고 있다. 참조를 두군데에서 하고 있는데 이 중 첫번째를 추적하면 QueryPerformanceCounter 함수를 보게 되고 이 부분은 쓰레드 생성부로 가는 것과는 해당 사항이 없었다. QueryPerformanceCounter 함수가 있는 부분은 게임틱과 연관이 깊기 때문에 매 게임 갱신주기에 어떤 작업을 하고 있는 경우가 태반이다. 대부분 FPS 를 출력하는데 사용되며 프레임과 프레임 사이의 처리를 삽입할때 사용하는 걸로 알고있다.(일인칭 3D 게임을 보면 초당 몇 프레임이라고 나오는 것을 본적이 있을 것이다.) 그렇다면 이 첫번째 추적위치가 그림20 의 call sub_5C4D3A 명령호출로 이어진다는 것인데 만약 이 명령호출을 방해하면 QueryPerformanceCounter 의 특성상 뭔가 바로 눈에 띄는 에러를 낼 것으로 판단했다. 그러나 실제로 필자가 그림20 의 call sub_5C4D3A 를 수정하며 테스트해본 결과 특이사항이 없었다. 게임의 3D 렌더링도 지장없이 잘만 작동되었다. 그렇기 때문에 첫번째 추적위치인 QueryPerformanceCounter 함수가 나타나는 지점은 무시해도 된다는 결론을 얻었다. 수많은 테스트 결과를 더 반복하고서 내린 결론이었다.

이제 그림21 의 참조부분에 있는 두군데 참조지점 중에서 나머지 두번째 호출지점인 sub_424750+59 으로 가본다. 재밌게도 그림22 와 같이 DialogFunc 함수영역으로 가게될 것이다. 처음 게임을 시작시킬 때 잠깐 화면에 보였다가 사라지는 다이얼로그 박스가 있었는데 바로 이 부분이었다는 것을 직감할 수 있었다. 계속추적하면 DialogFunc 는 실제로 WinMain 에서 호출되고 있으며 프로그램의 초반 도입부에서 실행되고 있었다. 프로그래밍을 해본 사람은 VC++ 로 다이얼로그 기반 프로그래밍 시의 구조와 흡사하다는 것을 바로 눈치 챌 것이다.

다음의 그림22 는 눈에 띄라고 브레이크 포인트를 걸어서 빨간색으로 처리했으나 실제로는 디버거를 작동시키지 못했다. 물론 프로세스를 Attach 시키면 디버거를 붙일 수 있을거라고 생각할 수 있으나 그 시점에서는 이미 이 게임의 해킹방지루틴이 작동하고 있어서 런타임 코드영역 보호 및 안티디버깅 루틴 등등의 방어루틴이 작동되고 있을 시점이다. 그래서인지 Exception 에러가 발생하면서 디버깅시 비정상종료나 메모리접근 에러가 난다. 그러므로 디버거를 띄울 생각은 버리고, 역어셈블 데이타만으로 분석을 하는 습관을 들여야 할 것이다. 그것이 장차 본인에게도 도움이 된다.
(IDA Pro 4.9 부터 익셉션 오류를 무시하고 계속해서 프로그램을 디버깅하는 기능이 추가되었다. 그렇지만 본인은 익셉션 발생까지 무시하면서 뒤가 구려진 프로그램을 디버깅하고 싶지는 않았다. 오류가 나는 이유는 이미 어딘가 정보가 깨졌거나 잘못된 접근을 했다는 소린데 그런 상태에서 더이상 디버깅을 하는 것이 무슨 의미가있을까..)

```
                sub_424750      proc near               ; CODE XREF: DialogFunc+3C↑p
68 A8 ED 5E 00                  push    offset a1_0_2_1 ; "1.0.2.1"
68 C0 45 42 00                  push    offset sub_4245C0 ; int
B9 A0 6F 64 00                  mov     ecx, offset unk_646FA0 ; int
E8 2C FC 19 00                  call    sub_5C4390
83 F8 01                        cmp     eax, 1
75 49                           jnz     short loc_4247B2
68 9C ED 5E 00                  push    offset a[redacted]game_exe ; "[redacted]Game.exe"
B9 A0 6F 64 00                  mov     ecx, offset unk_646FA0
E8 B5 FB 19 00                  call    sub_5C432D
68 FF 01 00 00                  push    1FFh
B9 A0 6F 64 00                  mov     ecx, offset unk_646FA0
E8 9A FB 19 00                  call    sub_5C4321
68 10 27 00 00                  push    2710h
6A 02                           push    2
B9 A0 6F 64 00                  mov     ecx, offset unk_646FA0
E8 E7 FB 19 00                  call    sub_5C437F
6A 00                           push    0
B9 A0 6F 64 00                  mov     ecx, offset unk_646FA0
E8 85 FD 19 00                  call    sub_5C4529
B9 A0 6F 64 00                  mov     ecx, offset unk_646FA0
E8 05 07 1A 00                  call    sub_5C4EB3
85 C0                           test    eax, eax
75 03                           jnz     short loc_4247B5

                loc_4247B2:                             ; CODE XREF: sub_424750+17↑j
32 C0                           xor     al, al
C3                              retn
                ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴

                loc_4247B5:                             ; CODE XREF: sub_424750+60↑j
FF 15 9C C0 5E 00               call    ds:GetTickCount
A3 80 6F 64 00                  mov     dword_646F80, eax
B8 9A F0 61 00                  mov     eax, offset byte_61F09A
EB 09                           jmp     short loc_4247D0
                ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴
8D A4 24 00 00 00 00 8B+        align 10h
```

[그림 22] 두번째 호출지점으로 역추적. 여러 함수들의 호출뒤에 쓰레드 생성함수 호출. (DialogFunc 를 보라)

결국 두번째 참조위치를 역추적해오면 위 그림22 에서 빤간색으로 표시한 부분까지 당도할 수 있다는 것이다. 바로 여기가 제거지점이 될 수 있다. 참조를 보자. CODE XREF 라고 된 부분에 DialogFunc+3C 라고 되어있는데 다이얼로그 함수영역에서 호출하고 있다는 것을 의미한다. 여기서 의문을 가질텐데 아예 다이얼로그를 호출하는 부분을 제거하면

되지않겠는가 라는 질문이다. 만약 그렇다면 한방에 끝날 것이다. 그렇다면 한번 해보라.. ㅎㅎ 그래서 리버싱이 어려운 것이지 달리 어려운게 아니다. 리버싱하면서 "바로 이 부분이야!" 라는 코드에 대한 감각과 신뢰가 없을 경우 너무 많은 변수가 생겨서 여기저기 CALL 과 점프명령을 쫓아가며 들쑤시다 볼 일 다보게 된다. 위에서 전제를 둔 것을 기억하는가? 다이얼로그 함수가 만약 게임이 실행되는 로직이나 함수에 영향을 미치지 않고 보안루틴만을 작동시키기 위해 호출되는 것이라면 달랑 그 부분을 제거하면 될 것이다. 그러나 시도해 본 바로는 게임이 실행조차 안됐다. 가정이 틀렸다는 소리다. 즉, 다이얼로그 함수 안에는 보안루틴 뿐만 아니라 기타 게임설정부분도 들어있다는 것이다. 그러므로 우리는 다이얼로그 함수 전체가 아닌 다이얼로그 함수 안에서 호출하는 부분에 존재하는 지점을 제거하는 것이다. 그 부분이 바로 위에서 빨간색으로 표시한 위치이고 바로 이 위치부터가 쓰레드 생성함수로 가는 길목이 되는 샘이다. 그렇기 때문에 쓰레드 생성으로 가는 길목 부분만 제거해야 프로그램의 동작에 방해를 주지 않는다. 쓰레드 자체가 실제로는 함수단위이고 달랑 보안루틴 함수를 작업쓰레드로 돌리는 구조이기 때문에 암 덩어리를 도려내듯 잘라버릴 수 있다. 의학에서도 환자의 몸을 수술하기전 엑스레이를 찍고 조직검사라는 걸 하지 않는가? 한마디로 우리는 지금 리버스 엔지니어링을 통하여 제거할 지점을 선정하는 조직검사를 한 것이나 다름없다. 우리가 도려낼 보안루틴이 마치 암세포가 여러조직 속에 깨알처럼 퍼진것과 같은 그런 구조라면 수정이 불가능할 수 있으므로 타진을 해보는 것이다. 필자는 의학의 '의'자도 모르지만 겉으로 보기에는 의학과 리버싱은 상당히 닮은 점이 많은것 같다. (주위에서 암세포가 조직속에 깨알처럼 퍼져서 수술이 안된다는 얘기를 들어본 기억이 있다. 보안루틴을 이 구조에 기반해서 만들 수 있다면? 개발해 볼 가치가 충분히 있다고 본다.)

이렇게 해서 우리는 첫번째 복병인 해킹방지 루틴을 제거하기위해 두군데의 수술지점을 찝어내었다. 실제로 역어셈블리 데이타만 봤기 때문에 정말 맞는지는 확인을 해봐야 한다. 만약 아니라면 그 짜증나는 작업을 처음부터 다시해야 한다. (필자는 거의 하루정도를 이 작업에만 메달렸었다. 연속적시간으로 따지면 프로그래밍시간까지 합쳐서 대략 6시간 이상을 움직이지 않고 작업했다고 할 수 있다...)

- 리버스 엔지니어링은 시간이 흐를수록 고통이다... - by AmesianX

방해루틴 제거하기

이미 앞에서 쓰레드 생성부/쓰레드 생성부로 가는 길목 두개의 제거지점을 찾아냈다. 두 지점을 제거하기만 하면 보안루틴은 해제될 것이다.
그렇다면 제거방법을 구체적으로 알아야 할 것이다.

앞서 설명한 것처럼 타이머 쪽의 쓰레드 생성함수 호출부분을 제거한다. 제거는 이미 알고있는대로 0x90 으로 OPCODE(기계어코드)를 바꿀 것이다.
아래 그림23 의 빨간색으로 표시한 박스의 맨 좌측에 해당 OPCODE 가 있는데 E8 FA FE FF FF 라고 되어있다.

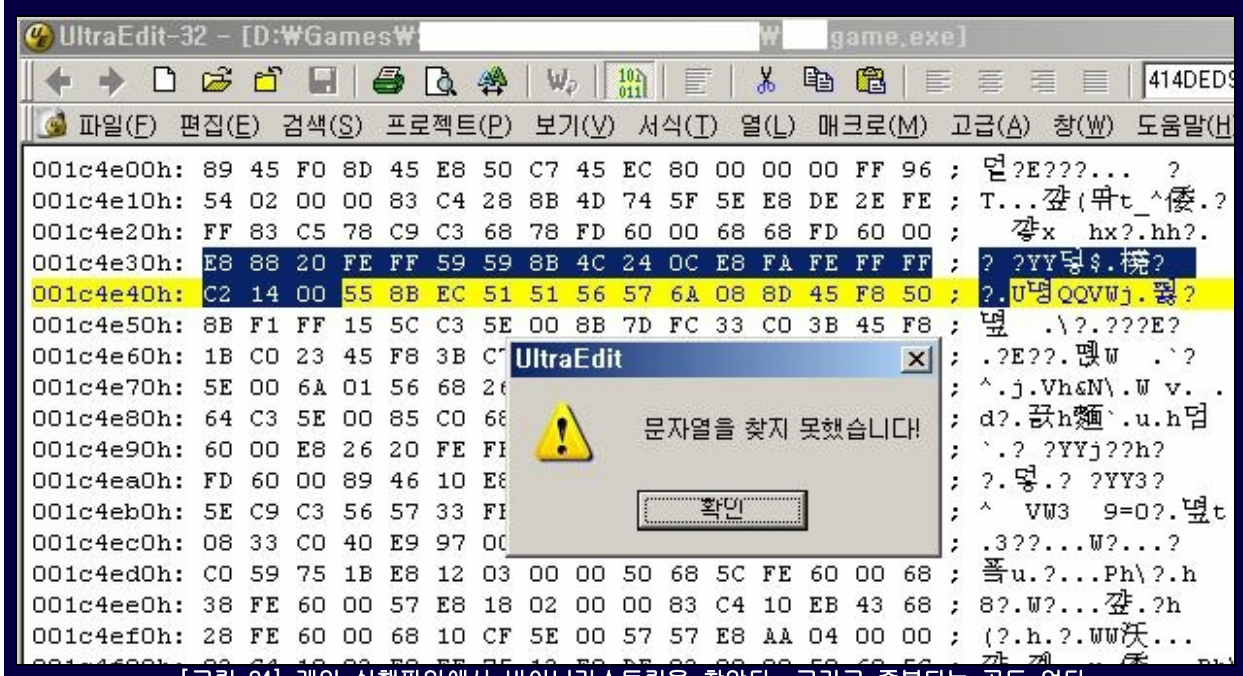
```
                        ; TIMECALLBACK fptc
                        fptc            proc near               ; DATA XREF: sub_5C4E43+32↓o

                        lpParameter     = dword ptr  0Ch

68 78 FD 60 00                          push    offset aDobytimerproc ; "DoByTimerProc"
68 68 FD 60 00                          push    offset aSTrace  ; "%s >> TRACE\n"
E8 88 20 FE FF                          call    nullsub_4
59                                      pop     ecx
59                                      pop     ecx
8B 4C 24 0C                             mov     ecx, [esp+lpParameter] ; lpParameter
E8 FA FE FF FF                          call    sub_5C4D3A
C2 14 00                                retn    14h
                        fptc            endp
```

[그림 23] 이미 앞에서 찾은 타이머 관련쪽 쓰레드 생성함수 호출부분

위 그림23 의 E8 FA FE FF FF 를 90 90 90 90 90 으로 지울것이다. 이미 우리가 알고있는 것처럼 0x90 은 NOP 코드로써 CPU 가 아무런 동작없이 EIP(인스트럭션 포인터)만 증가시킨다. 눈치빠른 독자라면 바이너리스트링을 만들것이라는 것을 알 것이다. 맞았다. 좌측의 OPCODE 를 일렬로 적어서 게임 실행파일을 울트라에디트 같은 HEX 편집기로 열고서 찾기를 할려고 한다. 이미 알고있는 내용이겠지만 다시 언급하자면 바이너리스트링을 만들때 적당한 범위로 적어야지 중복이 발생하지 않는다. 중복이란 위의 그림23 에서 보는 OPCODE 들이 실행파일 내에 다른 부분에도 중복 존재할 가능성이 있다는 뜻이다. 여기서 보는 루틴은 CALL 때문에 함수주소(오프셋)가 들어있으므로 중복될 가능성이 없지만 항상 바이너리스트링을 만들때는 만든 바이너리스트링이 파일 내에 한 부분만 일치하고 있다는 것을 확인해야 한다.
너무많이 적어도 곤욕이니까 적당히 call nullsub_4 부터 OPCODE 를 적어서 E88820FEFF59598B4C240CE8FAFEFFFC21400 라는 바이너리스트링을 만들었다.

(참고: 위의 그림23 의 루틴을 왜 호출지점에서 제거하지 않는가? 이유는 그림23의 루틴이 CALLBACK 함수라는 우려와 호출지점을 건드리면 분석범위가 커질것 같다는 필자의 경험에 의한 어떤 직감 때문이다. 이 문서를 쓰고있는 시점에서 다시 생각해보면 가능할 것 같다는 생각이 드는데. 사람은 참 간사하다. 화장실 들어가기 전에 다르고 나오고나서 다르다고 보안루틴 제거에 성공하니까 다른 방법을 써도 되겠단 생각이드니 얼마나 간사한가. 처음에 이 부분까지 추적해서 찾아내는데도 많은 어려움이 따랐고 실패도 많이 했는데 말이다.
잡설이지만.. 사람은 두가지 나쁜점이 있는데 하나는 남이 해결한건 자기도 열라 쉽게 할 수 있을거라는 생각. (잘하면 쉬워보이니까 일을 더 시킨다..) 또 다른 하나는 자기가 한번 빡시게 해결하고 나면 빡신 기억을 대부분 까먹는다. (정말 좋은 책은 독자가 격어보지 못한 빡신기억을 간접체험하도록 해줄수 있는 적절한 잡설이 포함된 책이라고 생각한다. 읽는데 방해가 될 수도 있겠지만 계속적인 질문을 던지면서 최대한 많은 참고내용을 전달하는 책이 필자의 경험상으로 볼때 남는것도 많았다.)

이제 만든 바이너리스트링을 갖고 실행파일에서 찾아보자. 필자는 HEX 편집기로 울트라에디트를 사용한다.



[그림 24] 게임 실행파일에서 바이너리스트링을 찾았다. 그리고 중복되는 곳도 없다.

그림24 에 보면 우리가 수정할려고 했던 E8 FA FE FF FF 가 보일 것이다. 90 90 90 90 90 으로 변경하면 된다. (바이너리 파일을 고칠꺼면 백업을 해둬야 한다)

이제 수정할 곳이 한 부분 더 남아있다.

```
E8 E7 FB 19 00                          call    sub_5C437F
6A 00                                   push    0
B9 A0 6F 64 00                          mov     ecx, offset unk_646FA0
E8 85 FD 19 00                          call    sub_5C4529
B9 A0 6F 64 00                          mov     ecx, offset unk_646FA0
E8 05 07 1A 00                          call    sub_5C4EB3
85 C0                                   test    eax, eax
75 03                                   jnz     short loc_4247B5

                        loc_4247B2:                             ; CODE XREF: sub_424750+17↑j
32 C0                                   xor     al, al
C3                                      retn
```

[그림 25] 그림22 의 색깔로 표시한 부분만 떼어낸 그림

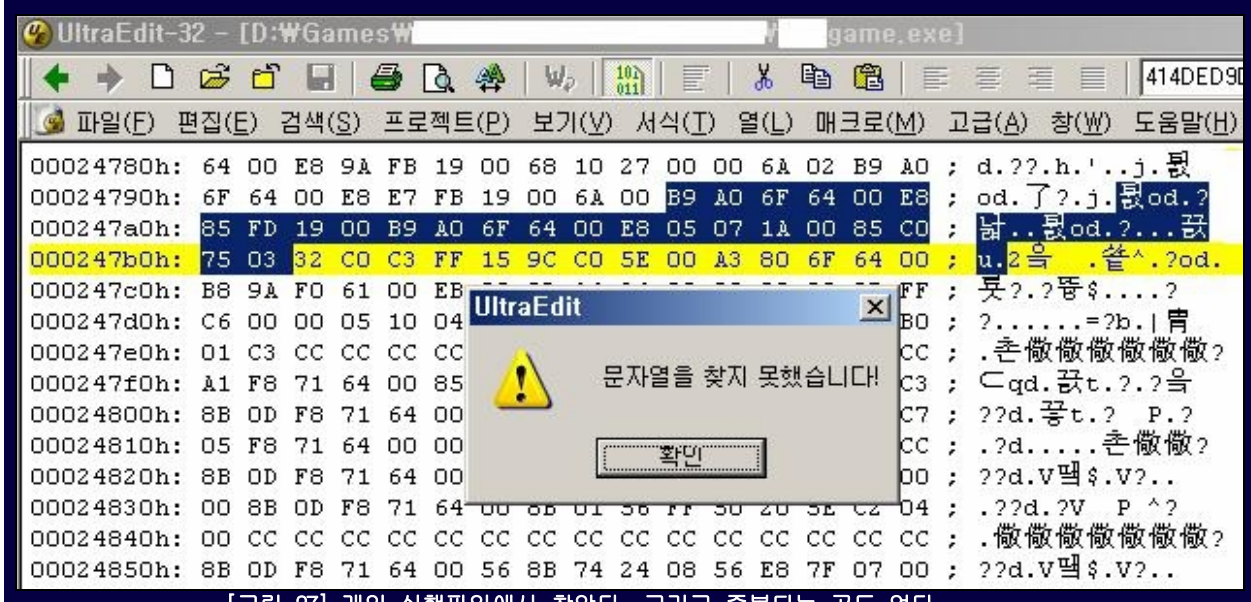
앞에서 찾은 그림22의 DialogFunc 에서 호출하고 있는 쓰레드 생성함수 호출도 제거해줘야 한다. 여기서는 쓰레드생성 함수를 호출하는 call 만 달랑 제거하면 안된다. 왜냐면 위에서 빨간색으로 표시된 호출부분 다음에 test eax, eax 가 쓰레드가 제대로 생성되었는지 반환값을 체크하고 있고 이 반환 값의 참/거짓 여부에 따라서 jnz short loc_4247B5 점프가 결정된다. jnz 는 점프 낫 제로(Jump Not Zero) 이므로 해석해보면 다음과 같다.

```
E8 E7 FB 19 00                          call    sub_5C437F
6A 00                                   push    0
B9 A0 6F 64 00                          mov     ecx, offset unk_646FA0
E8 85 FD 19 00                          call    sub_5C4529
B9 A0 6F 64 00                          mov     ecx, offset unk_646FA0
E8 05 07 1A 00                          call    sub_5C4EB3
85 C0                                   test    eax, eax
75 03                                   jnz     short loc_4247B5

                        loc_4247B2:                             ; CODE X
32 C0                                   xor     al, al
C3                                      retn
```

호출한다.
성공했나?
성공했으면 4247B5
루틴으로 가자!

[그림 26] 해석해보면 위와 같음

그렇기 때문에 우리는 call sub_5C4EB3 을 제거하고 밑에 test eax, eax 도 제거하고 jnz short loc_4247B5 는 jmp short loc_4247B5 로 수정하면 된다. 즉, 쓰레드 생성함수를 제거하고 체크하는 부분도 제거하고 성공했을때 조건분기하는 것을 무조건 분기인 jmp 명령으로 수정하면 된다. OPCODE 로는 그림26 의 빨간색 라인의 좌측부터 보이는 E8 05 07 1A 00 를 90 90 90 90 90 으로 교체하고 그 아래 85 C0 을 90 90 으로, 75 03 을 (75: jnz) EB 03 으로 (EB: jmp) 교체하면 된다. 결국 해당루틴의 부분을 다 지우고 75 03 조건분기 명령을 무조건 분기 EB 03 으로 바꾸는 작업을 하는 것이다. 이제 바이너리스트링을 만들어 보자. 적당히 mov ecx, offset unk_646FA0 명령부터 OPCODE 를 적어보자. 같은 명령이 두개 있으므로 헷갈리지 말고 첫번째부터 적어보자. 만들어진 바이너리스트링은 B9A06F6400E885FD1900B9A06F6400E805071A0085C07503 와 같이 할 수 있다.
마찬가지로 게임 실행파일을 열고 찾아서 확인까지 해본다.



[그림 27] 게임 실행파일에서 찾았다. 그리고 중복되는 곳도 없다.

자.. 위의 그림26 에서 빨간색으로 표시된 호출부분의 좌측 OPCODE 인 E8 05 07 1A 00 이 울트라에디트에서 보이는가? 앞서 설명한 것처럼 수정하면 된다. 혹시 모를수도 있다는 생각이 들어서 적어보면 E8 05 07 1A 00 부터 90 90 90 90 90 90 90 EB 03 으로 수정하면 될 것이다. 이렇게해서 실제로 게임 실행파일을 직접 HEX 에디터로 수정하면 해킹방지 루틴이 제거된다.

무슨 의미가 있는가? 우리는 왜 해킹방지 루틴을 제거했는가? 목적이 무엇인가?

간혹 기술을 추구하는 사람들이 목적을 잃고 기술을 위한 기술에 치우치는 것을 보곤하는데 참으로 안타깝다. 독자는 이 문서를 보면서 목적이 헷갈리지 않아야 된다.우리는 해킹방지 장치를 풀려고 한게 아니라 게임 채팅창을 후킹하려다가 해킹방지 장치가 공격해오는 바람에 정당방위의 일환으로써 놈을 잠재워 놨을 뿐이다. 계속해서 채팅창 후킹내용으로 넘어간다.

게임실행 바이너리를 직접 수정할 것인가? (예스? 노?)

자.. 복병제거까지 지나와서 후킹의 발판을 마련했다. 솔직히 여기까지는 누구나 다 아는 내용이고 IDA Pro 같은 역어셈블러와 울트라에디트 같은 HEX 편집이 가능한 유틸리티만 있으면 중고등학생 데려다놓고 1시간만 투자하면 다 따라할 수 있는 기초적인 내용 중에서도 가장 기초적인 내용이다. 그렇기 때문에 리버스 엔지니어라고 자부심을 갖고 싶다면 적어도 이런 방법은 테스트에만 쓰고 되도록이면 코드로 해결하려고 노력해야 할 것이다. 울트라에디트 같은 HEX 편집기로 저작권이 있는 파일을 수정하면 문제가 되는데 반면에 메모리를 패치하는 패처/로더 프로그램은 때에따라 법에 저촉되지 않는 것을 볼 수 있다.
해외에서는 이런 메모리 패치부류는 매우 자유롭게 논의되고 또, 공유하고 있으나 저작권이 있는 파일에 대한 수정과 업로드는 이들 공개커뮤니티에서 조차도 외면하는 경우를 종종 볼 수 있다. 글을 삭제한다던지 저작권이 있는 컨텐츠는 올리지 말라는 경고를 볼 수 있다. 공개적으로 크랙이나 해킹을 진행하는 곳에서 그런 현상을 보게 되면 참 어이없다고 생각할때가 있다. 하지만 그들은 직접적인 컨텐츠를 가공하는 것이아닌 코드를 통한 가공을 하기 때문에 법에 저촉된다는 생각을 갖고있지 않는것으로 보인다.(합법적으로 산 정품 운영체제가 제공하는 API 를 이용해서 내가 산 컴퓨터의 메모리를 제어한다는데 누가 뭐라고 딴지걸 수 있냐고 생각하는 것일까?) 바이너리 파일을 직접 수정하는 것에대한 문제가 이것만 있는 것은 아니다. 바이너리 파일을 보호하기 위해서 팩킹(Packing) 되어 있는 실행파일의 경우는 직접적인 파일수정을 가하기에는 많은 무리가있다.(이미 리버싱 커뮤니티에서 많이 논의되어온 언팩킹을 하면 가능하다.) 모든 것을 바이너리 수정으로 승부할 수 없는 또다른 이유는 제거나 변조는 어느정도 직접 HEX 에디터를 이용해서 가능하지만(위에서 보인 보안루틴제거) 그 외의 기능추가나 프로그램 내부로직을 제어할 수 있는 것은 한계가 따르기 마련이다.
좌우지간 여러가지 이유로 우리는 HEX 편집기로 할 작업을 프로그래밍으로 자동화 할수록 얻는게 더 많다. 그리고 그게 제일 깔끔한 방법이라고 생각한다.(프로그래밍을 좋아하는 사람이라면 깔끔한 방식과 그렇지 않은 방식에 대해서 자기만의 생각을 갖고 있을텐데 필자의 생각은 깔끔한건 범용적이고 모듈로써 떼었다 붙였다 할 수 있는 라이브러리 개념을 좋아한다. 대부분 필자와 같은 생각일 것이다. 잘 만들어놓은 후킹모듈은 나중에 실행파일에 붙였다 떼었다 하면서 하드에 저장된 원본 프로그램은 건드리지 않고도 원하는대로 제어할 수 있을 것이다.)

자.. 쓸데없는 내용을 잘 참아왔다. 기반이 마련됐으므로 이제 진짜 채팅창 후킹으로 넘어가보자.

위에서 보안루틴을 제거하는 것은 뒤에서 코드로 한꺼번에 추가할 것이다. 그보다도 여기서는 채팅창 후킹을 하기 위해서 채팅입력부분과 출력부분의 리버싱이 먼저 이루어져야 한다. 어느 지점의 코드를 후킹해야 할지 정해야 후킹을 할것 아니겠는가? 짐작하고 있겠지만 졸라 빡시게 입력창과 출력창을 찾아야 한다. 필자가 몇가지 노하우를 말해줄 수 있다면 리버싱 시에는 항상 가정을 하고 찾아야 해당 루틴을 금방 찾을 수 있다. 몇가지 가정을 해보면 채팅 입력창과 출력창은 각각 하나의 단위(?)함수로써 작동할 것이라는 가정, 입력버퍼와 출력버퍼는 각각 인수로써 넘어갈 것이라는 가정, 채팅 입력창에 입력을 한 다음에는 입력한 내용이 출력창에 뿌려지므로 입력창 함수와 출력창 함수는 필히 붙어있을 것이라는 가정을 미리 정해놓고 찾을 수록 더 빨리 찾을 수 있다. 이 가정을 하고 찾으면 마우스로 휘휘 저으면서 역어셈블 코드를 보더라도 "바로 이 지점인것 같

은데?" 라는 감이 올 것이다. 물론, 이때 확신이 서야 되는데 이노메 확신이 들기까지 계속 함수속을 헤메고 다니기 일쑤다. 결국 돌아오는 지점이 처음에 찾은 지점인 경우가 허다하다. 만일 확신하기 위해서 테스트 들어가는 시간을 없애고 감으로만 찾는다면 10 분 ~ 15 분 정도면 왠만한 루틴은 다 찾아낼 수 있다. 물론 테스트 작업을 거치지 않으므로 실패율은 따지지 않고 말하는 거지만 경험이 풍부한 프로그래머라면 거의 90% 이상 처음에 찾은 루틴이 대부분 맞아떨어질 것이다. 필자가 하는 말이 사실인지는 국내에 한글로 번역되어 나온 "소프트웨어 보안 코드 깨부수기: 원제 - EXPLOITING SOFTWARE: How to break code" 라는 책을 보면 알 수 있다. 리버스 엔지니어들은 10 분 ~ 15 분이면 필요한 루틴을 찾아낼 수 있다고 기술하고 있다.(세계적으로 유명한 사람이 지은 책으로 한글판이 나오기 한참 전부터 P2P 에서 영문판으로 PDF 가 나돌고 있음) 잡설이 좀 늘어졌는데 그렇다면 위에서 채팅창을 찾기위해서 몇가지 가정을 한 것은 무엇에 바탕을 둔 것인가?
정공학(일반 프로그래밍)에 의거한 것이다. "내가 이 프로그램을 만든다면 이 루틴은 이렇게 짤 것이다" 라는 생각이 역공학 분석시의 가정으로 작용하는 것이다. 해외의 리버싱 커뮤니티를 뒤지다 보면 지네들끼리 "이 코드는 이렇게 구성되어 있을 것이다" 라고 짤막하게 코드를 짜서 올리는 것을 종종 볼 수 있다. 그래서 리버스 엔지니어라면 최신 3D 게임(종합 프로그래밍 선물셋트)이나, 일반 프로그램(어플리케이션), 암호학(수학) 심지어는 임베디드와 운영체제(만능의 길)까지 깊은 프로그래밍 식견을 갖는 것이 실력을 좌지우지하게 된다. 즉, 프로그래밍을 잘해야 된다. 리버스 엔지니어링 잘하는 사람치고 프로그래밍 잘 못하는 사람은 보지 못했다.

(참고: 팁을 세가지 언급하자면 함수를 찾아가면서 인자로 넘어가는 것은 기본적으로 포인터라고 생각을 하라. 어셈블리는 기계어이다. 기계어는 메모리 주소를 갖고 장난치는걸 좋아하기 때문에 레지스터나 메모리를 대상으로 작업하고 이를 함수에 인자로 넘기게 되면 대부분은 주소값이다. 그렇기 때문에 함수를 후킹할때는 인자로 넘어가는 값을 잘 모른다면 그냥 포인터라고 생각하고 다룬다. 다른 팁은 요즘은 유니코드가 대세이기 때문에 버퍼가 넘어갈 때 char* (캐릭터 형 포인터) 가 쓰일수도 있지만 반면 unsigned short int * (부호없는 2바이트 정수형 포인터) 가 쓰일 수도 있다. 한글이 2바이트 이기 때문에 부호없는 short int 인 2바이트 포인터로써 넘어가게 된다. 즉, 한글버퍼가 함수에 인자로 넘어가는 것을 눈치챌 수 있는 방법이다.
이 밖에 플러스 알파팁을 하나 보너스로 드리면 게임을 구성하는 이미지나 객체의 좌표는 대부분 실수로 처리된다. 예를들면 640.0, 480.0 의 값이 게임내에 정확히 한군데만 사용된다고 가정한다면 절대로 640 에 해당하는 16 진수나 480 에 해당하는 16 진수로 찾아낼 수 없다. 이런 경우가 가끔 생기게 되는데 이럴 경우에 640.0 에 해당하는 실수 값 0x44200000 과 480.0 에 해당하는 실수 값 0x43F00000 을 키워드로 찾으면 매칭되는 곳을 찾을 수 있다. 필자의 경우 463 이라는 좌표 값을 바꿔야 하는 일이 생긴적이 있었는데 이 값이 세 지점에 존재했었다. 그런데 아무리 463 에 해당하는 16 진수 0x1CF 를 수정해도 원하는 변경이 이루어지지 않았다. 이 때문에 거의 한달가까이 삽질을하여 463(0x1CF) 의 실수 값 463.0 을 수정해야 한다는 것을 알아내었다. 실제로 463.0 에 해당하는 실수 값 0x43E78000 수치가 사용되고 있는 지점은 정확히 바이너리의 한 곳 뿐이었다. 게임 프로그래밍을 해본 사람만이 안해본 사람보다 리버싱 하는데 유리한 내용의 예가 될 수 있다. 일반적으로 다이렉트X 관련 함수가 정수보다도 실수를 많이 다루는 것을 사전에 알면 쉽게 해결할 수 있다는 것이다. 이상 리버싱 할때의 자잘한 팁을 제공하였다.)

자.. 이제 필자가 찾은 채팅창 입력부분인데 솔직히 찾은 방법을 마땅히 설명할 길이 없어서 위에서 잡소리를 좀 많이 했다. 굳이 설명한다면 C 언어의 프린트 계열 함수에서 문자열 포맷스트링 인자로 취급되는 %s 로 텍스트 검색을 해서 하나씩 함수들을 뒤지다가 가장 깔끔하고 간결한 놈을 잡아서 후킹을 걸었는데 그게 맞아 떨어졌다. 게임에서 채팅창에 "하이롱" 이라고 입력을 하면

아메시안엑스 : 하이롱

처럼 출력이 되는데 잘 보면 %s : %s 의 문자열 포맷스트링이다. (설마 포맷스트링 출력을 모른다면 C/C++ 과 어셈블리 공부에 소홀히 했을 가능성이.. -_-;)
IDA 에서 %s 로 검색을 하면 %s : %s 가 텍스트 검색에 나타난다. (IDA 에서 ALT+T 단축키 후 %s 입력으로 검색) 검색된 %s 주위에서 함수들을 왔다리 갔다리 하면서 뒤지다가 찾는데 거의 이틀이 걸렸다. 크로스레퍼런스(참조관계)가 여기저기 뻗어있어서 휘저어 다니는게 인간이 할짓이 못되는 것 같다. 어떻게든 문서를 쓰기위해 기를쓰고 찾았지만 채팅창입력 위치를 찾기위한 명확한 설명방법이 없는것이 참 애석할 따름이다. (이 부분에 대해서는 어쩌면 당연한 것일지도 모르겠다. 명확한 방법이 없기 때문에 리버스 엔지니어링을 공부하는 것이다. 좀 더 명확한 분석방법이나 노하우를 연구하는 것이 리버스 엔지니어가 죽을때까지 해야할 공부이니까.. 본인이 노하우를 하나 공개하자면 IDA 라는 디스어셈블러와 REC 라는 이탈리아 사람이 만든 C 언어 디컴파일러를 동시에 사용하는데 C 언어 제어구문으로 디컴파일 된 바이너리를 빠르게 탐색하여 위치를 찾으면 IDA 로 해당 부분을 다시 분석하는 방법이다. 이 방법은 C 언어에 강한 사람한테는 엄청난 가독성과 정확성으로 바이너리를 분석할 수 있게해준다. 가독성이 떨어지는 디스어셈블리만으로 바이너리 속의 정확한 루틴을 찝어내는 것이 전문가들에게도 어려운일이기 때문에 이러한 자기만의 방법들이 있어왔다. 이처럼 실질적인 분석론이나 노하우를 연구하는 사람들이 리버스 엔지니어링 커뮤니티에서 중요한 활동을 하게된다. 물론 해외의 경우이고 국내의 리버스 엔지니어링 커뮤니티는 본질이 왜곡되어 흔히 말하는 언팩킹과 크랙미(프로그램하나 만들어놓고 크랙하는 예제형식의 공부시스템) 또는 PE 헤더 공부에 많이 집중되어가는 경향이 있다. - 사실 리버스 엔지니어링은 소스코드의 재구현이라는 거대하고도 방대한 목표가 있음에도 불구하고 이를 망각하고 언팩킹과 크랙미 같은 “기계적 공부하기”를 하는 것은 초보자에게는 바보가 되는 지름길이다. 국내에서 언팩킹에 목메고 있을때 해외에서는 범용 언팩커를 만들 생각을 하며 크랙을 배우고자 하는 사람들을 위해서 크랙미를 만들었다. 이런일을 해외 해커들이 한 이유는 그들이 처음시작하는 리버스엔지니어를 돕기위함이거나 리버싱을 방해하는 요소를 해결하기 위함이다. 즉, 이러한 것들이 리버싱의 최종적인 모습이 아니라는 것을 안다면 언팩킹과 크랙미 또는 PE 헤더 같은 공부를 열심히 하는 것이 이상한 일이 아니다. 그렇지않고 실력의 고저차를 떠나 이런 특정 부분만 잘한다고 리버싱 잘하는 것이라고 생각한다면 그 곳이 그 사람의 한계점이다. 그 이상의 기대는 무리이다. 실제로 해외의 해커들이 리버싱하는 모습을 볼 수 있는 커뮤니티를 찾아보길 추천한다. 해외의 경우는 실제로 프로그램을 분석하면서 내부에서 사용되는 데이타구조를 분석하여 가시화 시키고 그 데이타를 다루는 게임내 루틴들을 찾고 분석하여 최종적으로 그 바이너리를 재창조한 것과 같은 효과를 얻는 것이 리버싱의 최종목표이며 그것을 실력의 기준으로 삼는다. 이런 커뮤니티에서는 그냥 단순히 다른 사람이 활동하는 것을 지켜보기만 해도 본인의 스킬이 향상된다. 실질적인 리버스 엔지니어링을 배우려면 해외 해커들 중에서도 개발자 성향 내지는 하드웨어와 소프트웨어 리버싱을 다루는 사이트를 많이 돌아다니자. 임베디드 쪽도 도움이된다. 웹서핑 만큼은 누구에게도 지지않을 자신이 있다면 어렵지않게 신의 경지에 오른 실력자가 리버싱하는 코드를 볼 수 있을 것이다. 조심하자.. 자포자기 심정을 느낄 만큼 강력한 포스(Force)의 코드나 문서가 존재할지 모르니까.. 필자도 몇번이나 슬럼프에 빠졌었다..)

다음은 앞에서 언급했던 %s 와 %s : %s 같은 포맷 스트링 탐색으로 찾아낸 채팅창 입력부분이다. 실제로는 채팅창 출력부분을 먼저 찾았지만 설명상 채팅창 입력부분을 먼저 설명하겠다.

```
                        ; Attributes: bp-based frame

                        ; int __fastcall sub_422F40(int,int,int,char)
                        sub_422F40      proc near               ; CODE XREF: sub_488AC0+20D↓p
                                                                ; sub_4B3EE0+2D↓p
                                                                ; sub_4E3810+B↓p
                                                                ; sub_4E5D30+29↓p
                                                                ; sub_4E5DB0+20↓p ...

                        var_208         = byte ptr -208h
                        var_4           = dword ptr -4
                        arg_0           = dword ptr  8
                        arg_4           = byte ptr  0Ch

55                                      push    ebp
8B EC                                   mov     ebp, esp
83 E4 F8                                and     esp, 0FFFFFFF8h
81 EC 0C 02 00 00                       sub     esp, 20Ch
A1 84 36 62 00                          mov     eax, dword_623684
57                                      push    edi             ; char
89 84 24 0C 02 00 00                    mov     [esp+210h+var_4], eax
33 C0                                   xor     eax, eax
C6 44 24 08 00                          mov     [esp+210h+var_208], 0
B9 7F 00 00 00                          mov     ecx, 7Fh
8D 7C 24 09                             lea     edi, [esp+9]
F3 AB                                   rep stosd
66 AB                                   stosw
AA                                      stosb
8B 45 08                                mov     eax, [ebp+arg_0]
6A 00                                   push    0
50                                      push    eax
B9 D0 62 64 00                          mov     ecx, offset unk_6462D0
E8 92 74 14 00                          call    sub_56A410
84 C0                                   test    al, al
74 45                                   jz      short loc_422FC7
8A 45 0C                                mov     al, [ebp+arg_4]
84 C0                                   test    al, al
74 28                                   jz      short loc_422FB1
68 00 02 00 00                          push    200h
8D 4C 24 0C                             lea     ecx, [esp+214h+var_208]
51                                      push    ecx
68 86 11 01 00                          push    11186h
B9 80 A5 63 00                          mov     ecx, offset dword_63A580
E8 5E 11 04 00                          call    sub_464100
8D 54 24 08                             lea     edx, [esp+210h+var_208]
52                                      push    edx             ; char *
6A 04                                   push    4               ; char
E8 C2 FE FF FF                          call    ChatPrint
83 C4 08                                add     esp, 8
```

[그림 28] 채팅창 입력부분 arg_0(첫번째 인자) 가 입력버퍼다

위 그림28 에서 보이는 루틴에서 arg_0 즉, 다시말해서 [ebp+0x08] 위치에 채팅창에 입력한 "하이룽" 문자열이 저장된 주소를 담고 있다. 빨간색 네모박스로 처리한 것은 그 지점에 후킹을 걸게 될 것이기 때문에 표시해 놓았다. 자 그럼 밑부분에 ChatPrint 라고 된 부분이 보이는가? ChatPrint 함수는 채팅창출력 화면에 출력을 담당한다. 이 함수는 주소가 아니고 왜 이름으로 되어 있을까? 그건 필자가 미리 찾아서 이름을 지어놨기 때문에 주소로 안나오고 이름으로 나오고 있는 것이다. 다음의 그림을 보자.

```
                        ; int __cdecl ChatPrint(char,char *,char)
                        ChatPrint       proc near               ; CODE XREF: sub_407D60+75↑p
                                                                ; sub_407E20+72↑p
                                                                ; sub_407ED0+CD↑p
                                                                ; sub_40AA80+72↑p
                                                                ; sub_40C410+8D↑p ...

                        var_204         = byte ptr -204h
                        var_4           = dword ptr -4
                        arg_0           = byte ptr  4
                        arg_4           = dword ptr  8
                        arg_8           = byte ptr  0Ch

81 EC 04 02 00 00                       sub     esp, 204h
A1 84 36 62 00                          mov     eax, dword_623684
8B 8C 24 0C 02 00 00                    mov     ecx, [esp+204h+arg_4]
89 84 24 00 02 00 00                    mov     [esp+204h+var_4], eax
8D 84 24 10 02 00 00                    lea     eax, [esp+204h+arg_8]
50                                      push    eax             ; va_list
51                                      push    ecx             ; char *
8D 54 24 08                             lea     edx, [esp+20Ch+var_204]
68 00 02 00 00                          push    200h            ; size_t
52                                      push    edx             ; char *
E8 DC 59 18 00                          call    __vsnprintf
8B 8C 24 18 02 00 00                    mov     ecx, dword ptr [esp+214h+arg_0]
8D 44 24 10                             lea     eax, [esp+214h+var_204]
50                                      push    eax             ; unsigned __int8 *
51                                      push    ecx             ; char
E8 4D FF FF FF                          call    sub_422E00
8B 8C 24 18 02 00 00                    mov     ecx, [esp+21Ch+var_4]
E8 40 4E 18 00                          call    sub_5A7CFF
81 C4 1C 02 00 00                       add     esp, 21Ch
C3                                      retn
                        ChatPrint       endp
```

[그림 29] 채팅창 출력함수

그림29 의 루틴이 실제로 채팅창에 입력된 문자열을 채팅출력창에 뿌리는 일을 담당한다. 그럼 처음에 어떻게 찾았는가? 다음의 그림을 보면 알수 있다.

```
                        ; char aSOJT_[]
5B 25 73 5D 20 C4 F9 BD+aSOJT_          db '[%s] 퀘스트를 포기하셨습니다.'
BA C6 AE B8 A6 20 C6 F7+                                    ; DATA XREF: sub_4C5880+46C↑o
00                                      db    0
00                                      db    0
00                                      db    0
                        ; char aDCSBJC_[]
25 64 20 B0 E6 C7 E8 C4+aDCSBJC_        db '%d 경험치를 습득 하셨습니다.',0
A1 B8 A6 20 BD C0 B5 E6+                                    ; DATA XREF: sub_4C5880+2E7↑o
00 00 00                                align 4
                        ; char aSOB_[]
5B 25 73 5D 20 C4 F9 BD+aSOB_           db '[%s] 퀘스트가 자동 수락 되었습니다.'
BA C6 AE B0 A1 20 C0 DA+                                    ; DATA XREF: sub_4C5880+17C↑o
00                                      db    0
44 61 74 61 5C 49 6E 74+aDataInterfa_70 db 'Data₩Interface₩99_Event.iwz',0
65 72 66 61 63 65 5C 39+                                    ; DATA XREF: sub_4C6700+35↑o
44 61 74 61 5C 49 6E 74+aDataInterfa_69 db 'Data₩Interface₩99_1_Event_Itemselect.iwz',0
65 72 66 61 63 65 5C 39+                                    ; DATA XREF: sub_4C6700+15↑o
```

[그림 30] 경험치를 습득했다는 것은 게임을 플레이할 때 채팅출력창에 나오는 메시지이다.

보통 게임에서 경험치를 습득하면 채팅출력창에 뿌려진다. 위에서 보는 것처럼 경험치를 습득했다는 문자열이 데이타 영역에 박혀있고 이를 참조하고있는 루틴으로 이동하면 다음과 같은 부분이 나온다.

```
74 1D                   jz      short loc_4C5B7D
51                      push    ecx
8D 9E D4 00 00 00       lea     ebx, [esi+0D4h]
68 0C 94 5F 00          push    offset aDCSBJC_ ; "%d 경험치를 습득 하셨습니다."
53                      push    ebx             ; char
E8 04 21 0E 00          call    _sprintf
53                      push    ebx             ; char *
6A 04                   push    4               ; char
E8 F6 D2 F5 FF          call    ChatPrint
83 C4 14                add     esp, 14h
```

[그림 31] 문자열을 sprintf 함수를 통해 버퍼에 넣고 이 버퍼를 ChatPrint 함수의 인자로 넘긴다.

자.. 이제 감이 올 것이다. 경험치를 습득했다는 문자열 주소가 스택에 push 되고 바로 다음에 오는 함수가 채팅출력창이 될 것이라고 생각할 수 있지만 실제로는 sprintf 가 가장 많이 나타날 것이다. 미리 준비해둔 버퍼에 sprintf 함수로 경험치 습득 문자열을 할당한 뒤 버퍼주소를 인자로 넘기면서 다음명령의 함수를 호출한다. 대부분은 이때 실제 처리함수가 호출되며 이 실제 처리함수가 위에서 ChatPrint 라고 해놓은 채팅창출력 함수이다. (앞에서 말했지만 저렇게 보기좋게 ChatPrint 라고 되있지않고 주소로 나온다. 필자가 이름을 바꾼거다.)
이렇게 해서 필자는 채팅출력창도 찾았고 이름을 ChatPrint 라고 바꿔서 채팅입력창을 찾을때 한결 수월 할 수 있었다. 그림29 에서 함수의 이름을 바꾸면 IDA 는 자동으로 모든 참조부분의 이름을 바꾸기 때문에 그림28 에서도 자동으로 함수가 바뀌어 있는 것이다. 그림28 에는 아래가 잘려서 안나왔지만 입력버퍼의 조건에 따라서 즉, 비어있는지 아닌지에 따라서 ChatPrint 함수에 뿌려지기도 안뿌려지기도 하는 것이 채팅입력창 루틴이라는 확신을 좀 더 확고하게 할 수 있었기 때문에 분석시에 많은 도움이 되었다. 만일 그림28 의 채팅입력창 루틴이 엄청나게 복잡했다면 ChatPrint 라고 이름을 바꿔놓지 않았을때 일일이 함수를 들락날락하면서 어떤 함수인지 확인했을 것이다. 머리가 좋아서 루틴과 주소를 명확히 기억하는 사람이라면 해당사항이 없겠지만 필자는 머리가 나빠서 한번 뒤진 함수를 또 뒤지기때문이다. (실제로 채팅출력창이 제일 찾기 쉬웠기 때문에 먼저 찾아서 ChatPrint 로 이름을 바꾼후 포맷스트링 %s : %s 을 이용한 채팅입력창 찾기에 성공하였다.)

이제 채팅입력창과 그림29 의 채팅출력창을 후킹하기 위한 코드를 작성해야 한다.

```
                        ; Attributes: bp-based frame

                        ; int __fastcall sub_422F40(int,int,int,char)
                        sub_422F40      proc near               ; CODE XREF: sub_488AC0+20D↓p
                                                                ; sub_4B3EE0+2D↓p
                                                                ; sub_4E3810+B↓p
                                                                ; sub_4E5D30+29↓p
                                                                ; sub_4E5DB0+20↓p ...

                        var_208         = byte ptr -208h
                        var_4           = dword ptr -4
                        arg_0           = dword ptr  8
                        arg_4           = byte ptr  0Ch

55                                      push    ebp
8B EC                                   mov     ebp, esp
83 E4 F8                                and     esp, 0FFFFFFF8h
81 EC 0C 02 00 00                       sub     esp, 20Ch
A1 84 36 62 00                          mov     eax, dword_623684
57                                      push    edi             ; char
89 84 24 0C 02 00 00                    mov     [esp+210h+var_4], eax
33 C0                                   xor     eax, eax
C6 44 24 08 00                          mov     [esp+210h+var_208], 0
B9 7F 00 00 00                          mov     ecx, 7Fh
8D 7C 24 09                             lea     edi, [esp+9]
F3 AB                                   rep stosd
66 AB                                   stosw
AA                                      stosb
8B 45 08                                mov     eax, [ebp+arg_0]
6A 00                                   push    0
50                                      push    eax
B9 D0 62 64 00                          mov     ecx, offset unk_6462D0
E8 92 74 14 00                          call    sub_56A410
```

XREF 를 5개이상 출력하도록 했는데 "..." 이다. 호출지점으로 가서 첫번째 인수를 일일이 다 후킹해줄 것인가? 함수 하나만 후킹하면 참조하는 놈들은 다 적용될 것이다.

push ebp 이전 프레임포인터 저장
ebp 에 esp 를 할당(스택포인터:TOP)
and 연산은 모냐... -_-;
esp 에 0x20C 를 빼서 지역공간마련,
eax 에 메모리 공간을 넣어주고 이를 지역변수 var_4 에 준다. 그리고 edi 에 어떤 지점을 주고 ecx 에 7F 만큼 지정한 후에 stos 명령으로 메모리를 반복해서 복사하는데 이 지점안에서 5 바이트 코드를 가로챘을때 stos 명령이 이루어지기 전이라 문제가 생길지 모르므로 안전하게 stos 명령뒤에 5 바이트를 가로채기로 한다.

또한, 빨간색 박스부분을 후킹지점으로 선택한 이유는 [ebp+arg_0]를 eax 레지스터에 넣고 있으므로 이 부분을 가로채면 후킹함수에서 실행되기 때문에 eax 레지스터에 다른 주소값만 넣어주면 버퍼를 바꾸는데 편리하다는 이점이 있다.

[그림 32] 그림28 의 채팅입력창 루틴의 후킹지점 부가설명

위의 그림32 에서 채팅입력창의 후킹지점으로 눈도장 찍어버린 빨간 박스의 코드를 다음과 같이 후킹시킬 수 있다.

```
void __fastcall GameChatHooker(char *Chat)
{
    #ifdef _DEBUG
    DbgPrintf("GameChatHooker = %s₩n", Chat);
    #endif

    char MsgBuff[2048] = "님! 죄송한데요.. 뭐 좀 하나만 물어봐도 될까요?";

    __asm {
            pushad
            mov     eax, 0x00422E00
            lea     ebx, MsgBuff
            push    ebx
            push    0x05
            call    eax
            add     esp, 0x08
            popad
    }
}

void __declspec(naked) GameChatHooker_STUB()
{
    __asm {
        nop                                 // Make room for original code
        nop
        nop
        nop
        nop
        nop
        nop

//      .text:00422F6E 8B 45 08                 mov     eax, [ebp+arg_0]
//      .text:00422F71 6A 00                    push    0

        // 스택복구 처리
        pushad
        mov     edx, dword ptr [esp+0x20+0x04]
        mov     ebx, dword ptr [esp+0x20]

        mov     dword ptr [esp+0x20+0x04], ebx
        mov     dword ptr [esp+0x20], edx

        mov     ecx, eax
        CALL    GameChatHooker
        popad

        ret
    }
}
```

[그림 33] 후킹코드 스텁과 함수로 이루어짐

위 그림33 의 후킹함수의 호출관계를 보면 GameChatHooker_STUB 함수가 GameChatHooker 함수를 호출하는 관계다. 그러므로 독자는 GameChatHooker_STUB 함수를 어디서 작동시켜 주는지 궁금할 것이다. D2HackIt 의 ServerStartStop.cpp 파일을 열고 Intercept 라는 함수를 보라. 이미 앞에서 설명한 인터셉트가 메모리 상의 게임 바이너리에서 후킹할 지점의 코드를 가로채어준다.

Intercept(INST_CALL, psi->fps.GameChatHooker.AddressFound, (DWORD)&GameChatHooker_STUB, psi->fps.GameChatHooker.PatchSize);

Intercept 함수의 psi->fps.GameChatHooker.AddressFound 변수에 담겨있는 4바이트의 DWORD 값은 복사할 장소 즉, 후킹할 장소의 메모리 주소가 된다. 그림33 에서 GameChatHooker_STUB 함수를 호출하기 위해서는 CALL GameChatHooker_STUB 이라는 5 바이트 명령이 실행되어야 하는데 그림28 의 빨간박스로 된 코드의 주소지점인 psi->fps.GameChatHooker.AddressFound 에다가 복사하게 되는 것이다. 이때 INST_CALL (1바이트) + GameChatHooker_STUB 함수주소(4바이트) = 총 5 바이트의 호출(CALL) 명령이 복사된다. (PatchSize 는 조금 뒤에서 설명함)

(참고: 어셈블리어에서 0xE8 즉, CALL 명령은 5 바이트 명령으로 0xE8 명령코드와 호출할 함수의 주소(주의: 고정주소가 아니라 오프셋임!!) 로 이루어지는
5 바이트 명령이다.)

자, 여기서 psi->fps.GameChatHooker.AddressFound 의 실제 값은 다음의 fps 구조체변수만 넘기면 자동으로 셋팅되서 리턴된다.

```
if (!GetFingerprint("D2HackIt", "GameChatHooker", psi->fps.GameChatHooker))
{
    #ifdef _DEBUG
    DbgPrintf("GameChatHooker  Not  Found!");
    #endif

    return FALSE;
}
```

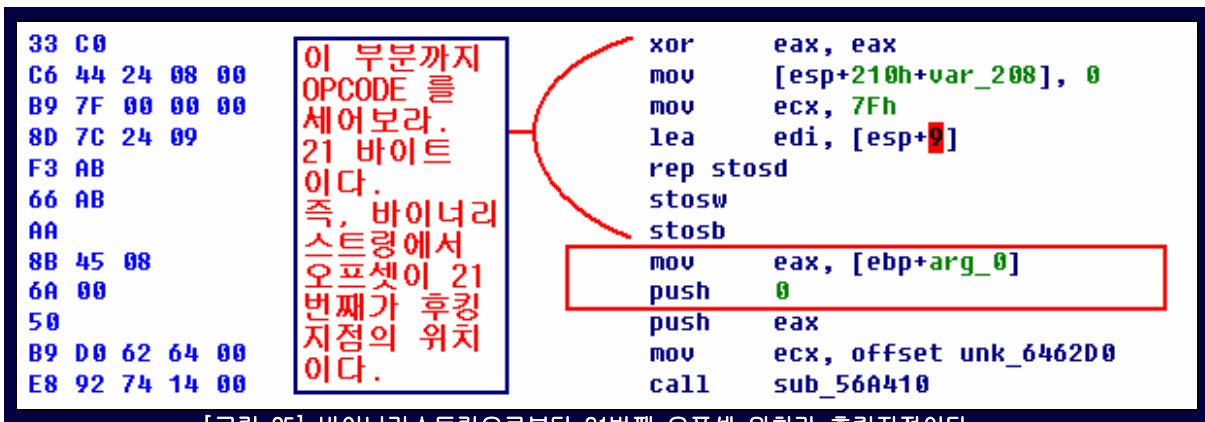
GetFingerprint 함수는 메모리의 실행파일 이미지를 검색하여 앞서 빨간색 박스로 표시했던 후킹지점 위치를 찾아서 주소를 돌려준다. 그 주소를 psi->fps.GameChatHooker.AddressFound 에 잘 저장해서 돌려주므로 중요한 놈이다. 그러면 GetFingerprint 함수는 뭘보고 찾는건가? 예상했겠지만 GetFingerprint 함수를 분석하면 환경설정파일 D2HackIt.ini 파일의 FingerprintData 섹션으로부터 정보를 얻어낸다.

```
1  [FingerprintData]
2  ; GameChatHooker
3  GameChatHooker=[redacted]game.exe,5,21,33C0C64424080CB97F0000008D7C2409F3AB66ABAA8B45086A00
4  SecurityRemoverTimer=[redacted]game.exe,5,11,E88820FEFF59598B4C240CE8FAFEFFFFC21400
5  SecurityRemoverDialog=[redacted]game.exe,9,15,B9A06F6400E885FD1900B9A06F6400E805071A0085C07503
```

GameChatHooker=실행파일.exe,패치하는 곳의 명령사이즈,바이너리스트링
때에 따라서는 DLL 파일도 가능하다

[그림 34] 환경설정 파일인 D2HackIt.ini 내용

앞서 보안루틴 해제를 설명하면서 바이너리스트링을 언급했던 것을 기억할 것이다. 방법이 같지만 프로그래밍을 통해 자동화 시키는 것이다.
그림32 에서 xor eax, eax 명령의 좌측에 있는 OPCODE 부터 적어보면 위의 GameChatHooker 항목과 같다.
이 33C0C64424080CB97F0000008D7C2409F3AB66ABAA8B45086A00 바이너리스트링으로 메모리에 로딩되어있는 실행파일에서 후킹지점을 대략적으로 찾은 후에 바이너리스트링의 시작점으로부터 지정된 오프셋(그림34 에 21로 지정되어있음) 위치를 통해 정확한 지점을 찾게된다. 다음의 그림으로 보면 간단할 것이다.



[그림 35] 바이너리스트링으로부터 21번째 오프셋 위치가 후킹지점이다.

그림34 를 보면 GameChatHooker 항목의 "5,21,바이너리스트링" 에서 5 는 위의 그림에서 보는 것처럼 빨간색 박스의 후킹지점에 해당하는 코드가 5 바이트라는 것을 의미하고 앞서 Intercept 함수의 마지막 인수였던 psi->fps.GameChatHooker.PatchSize 가 이 값을 갖게된다. 21 은 위의 그림 설명처럼 바이너리스트링 시작점으로부터 21번째 떨어져있다는 오프셋을 지정한 것이다. 이 21 값은 GetFingerprint 함수 내부적으로 계산하여 사용된다.
이제 감이 왔겠지만 GetFingerprint 함수가 울트라에디트(-_-;) 역할을 대행하는 것을 알 수 있다. (뒤에서 더 설명하겠지만 울트라에디트가 해주지 못하는 기능까지도 GetFingerprint 함수로 할 수 있다.)

자 이제 정리를 해보면 환경설정파일을 이용해서 GetFingerprint 함수가 정확한 후킹지점을 찾아내고 그 지점을 Intercept 함수를 통해 우리의 함수를 호출하도록 코드를 덮어버린다.

[그림 36] 기존의 5 바이트는 우리의 함수를 호출하도록 덮어진다.

바로 채팅입력창 루틴에 위에서 작성한 GameChatHooker_STUB 함수호출이 박힘으로써 우리가 만든 코드로 빠져드는 것이다. (휴~ 문서쓰기 정말 힘들다.)
GameChatHooker_STUB 이 런칭되는 것을 설명하기 위해서 긴 설명을 한 것 같다. 말 안해도 알겠지만 우리의 함수가 실행되려면 채팅입력을 시도 해야지만 실행될 것이다. (너무 당연한 얘긴가?)
독자는 여기서 몇가지 의문을 가져야 한다. 그림36 처럼 원래의 5 바이트 코드가 다른 코드로 변조되었는데 원래의 코드가 어디로 갔는 가? 원래의 실행코드가 변조되었는데 과연 제대로 잘 작동할 것인가? 라는 의문정도는 가져야 할 것이다. 5 바이트 코드가 어디로 갔는지는 Intercept 함수를 분석하면 그 이유를 알 수 있는데 앞서 채팅입력창 루틴의 빨간박스로 지정한 후킹지점의 코드는 GameChatHooker_STUB 함수의 내부에 마련된 빈공간에 복사 되어 GameChatHooker_STUB 함수가 실행될때 그 곳에서 실행된다.

자.. 포토샵보다 깔삼한 MSPaint 로 작성된 필자의 그림노트를 보면서 어떻게 돌아가는지 훑어보라..

채팅창입력루틴이 후킹된 모습

```
C6 44 24 08 00          mov     [esp+210h+var_208], 0
B9 7F 00 00 00          mov     ecx, 7Fh
8D 7C 24 09             lea     edi, [esp+9]
F3 AB                   rep stosd
66 AB                   stosw
AA                      stosb
E8 xx xx xx xx          call GameChatHooker_STUB
50                      push    eax
B9 D0 62 64 00          mov     ecx, offset unk_6462D0
E8 92 74 14 00          call    sub_56A410
84 C0                   test    al, al
```

명령마다 스택포인터가 움직이는 것을 가상으로 따져보는 것이지 실제 함수의 작동과 연관지어 생각하지 말자!! ESP 가 4 증가(-) 하면서 리턴어드레스(돌아갈 위치)를 저장한다는 개념이며 ESP 가 어디까지 왔나를 따져서 이동된 곳을 보자는 개념이다!

스택을 조심하라!! 매우중요!!

위의 채팅입력창의 원본 5 바이트 코드를 가로챘다. 흐름이 바뀐다... 경고!! 경고!!

우리의 함수가 실행되기 전까지는 원래 게임코드이므로 신경쓸 필요가 없으나 우리코드가 삽입된 부분부터 스택이 어떻게 변하는지 유심히 따져봐야 한다.
call GameChatHooker_STUB 함수가 실행되면 스택에 리턴주소가 저장된다!!

그렇다면 함수 실행이 끝나면 안전하게 리턴될까? 천만의 말씀 만만의 콩떡이라고 절대 그렇지 않다. call 호출이 일어나고 스택에 리턴주소를 저장한다는 건 함수실행이 모두 끝날때 그 리턴주소를 스택에서 정확히 빼낼 수 있다는 말인데 오른쪽 파란박스에서 원본코드가 복사된 것을 보면 push 명령이 실행되어 스택에 4바이트를 증가시킨다.(스택은 마이너스로 커짐) 아래부분을 보면 pushad 와 popad 로 둘러싸인 것은 신경쓸 필요 없다! 왜? pushad 는 8개의 중요 레지스터를 스택에 백업하고 popad 는 다시 복구하는 명령이기 때문에 스택 포인터는 결국 제자리로 돌아와서 뒤틀릴 수 없다. 그렇다면 스택을 다루는 명령이 짝이 안맞게 된다는 것을 눈치챘을 것이다. 어셈블리의 기본중에서 push 와 pop 은 짝을 이루어야 한다고 가르치고 있다. 그런데 오른쪽 파란박스를 보면 복사된 원본코드의 push 가 맨 마지막의 ret 와 짝을 이루게 된다. 즉, 리턴주소라고 믿고 스택에 저장된 0x00000000 을 뽑아서 점프하는 바람에 EIP 가 0x00000000 이 되고 엑세스 바이올레이션 에러가 나면서 프로그램이 박살날 것이다. 그렇기 때문에 스택복구를 하는 것이다. 오른쪽의 주석을 보면 스택복구 처리라고 되어 있는데 원본코드가 스택에 0 을 저장하기 때문에 0 과 리턴 어드레스의 자리를 바꿔서 저장하는 처리를 하고 있다.

```
void __declspec(naked) GameChatHooker_STUB()
{
    __asm {
        nop                 // Make room for original code
        nop
        nop
        nop
        nop
        nop
        nop

        // 스택복구 처리
        pushad
        mov     edx, dword ptr [esp+0x20+0x04]
        mov     ebx, dword ptr [esp+0x20]

        mov     dword ptr [esp+0x20+0x04], ebx
        mov     dword ptr [esp+0x20], edx

        mov     ecx, eax
        CALL    GameChatHooker
        popad

        ret
    }
}
```

```
8B 45 08                mov     eax, [ebp+arg_0]
6A 00                   push    0
```

개념상 호출과 동시에 리턴어드레스가 여기에 저장됨!

원본코드는 이미 Intercept 함수에 의해서 이 위치에 실제로 복사되어 있는 상태이다!!

push 0 가 ESP 를 4 증가시킴

ESP

개념상으로 ESP 가 이 위치까지 와있다고 생각하라!!

8개의 레지스터가 스택을 0x20(32바이트) 만큼 증가시킨다. 개념상 ESP 위치를 보라. [esp+0x20] = ESP 이다. [esp+0x20+0x04] = ESP+0x04 이고 리턴어드레스 위치 이다. 이 둘을 바꾸면 스택이 복구된다.

fastcall 호출규약을 보라! ecx 가 GameChatHooker 함수의 Chat 인수로 취해진다. 위의 함수에서 ecx 는 eax 를 넘겨받는데 eax 는 복사된 원본코드에 의해서 arg_0 를 담고 있다!!

자.. 이렇게 해서 결국 GameChatHooker 함수로 채팅창에 입력한 문자열이 넘어오게 된다. 이를 테스트로 뿌려보기 위해서 DbgPrintf 라는 함수로 출력한다. 출력되는 결과는 DebugView 라는 외부 프로그램이 로그를 잡아서 보여준다.(DebugView 의 사용은 인터넷에서 참고요망)

그 다음에는 MsgBuff 라는 버퍼를 하나 마련해서 "님! 죄송한데요.. 뭐 좀 하나만 물어봐도 될까요?" 라는 문자열을 담아 놓는다. 그리고 우리가 만든함수가 아닌 실제 게임에서 사용하고 있는 0x00422E00 함수를 호출하면서 인수로써 MsgBuff 를 넘겨준다. 그러면 게임을 실행해서 채팅입력부에 아무런 문자나 입력할때 게임 화면의 상단과 채팅창출력부에 "님! 죄송한데요.. 어쩌구" 의 문자가 출력되는 것을 볼 수 있다. 나이스!!

여기서 주의할 점!! 필자가 거의 이틀을 헤맸던 부분인데 두개의 인자를 넘겨줬기 때문에 스택을 0x08 만큼 다시 복구 시켜야 한다. add esp, 0x08 을 call 뒤에서 복구해주지 않으면 스택명령의 짝이 뒤틀려서 엉뚱한 곳으로 리턴하므로 프로그램이 박살날 것이다. (0x05 는 0x00422E00 함수를 임의적으로 후킹을 함으로써 뭐가 넘어 가는지 정탐하면서 뽑아낸 숫자다.)

```
void __fastcall GameChatHooker(char *Chat)
{
    #ifdef _DEBUG
    DbgPrintf("GameChatHooker = %s\n", Chat);
    #endif

    char MsgBuff[2048] = "님! 죄송한데요.. 뭐 좀 하나만 물어봐도 될까요?";

    __asm {
            pushad
            mov     eax, 0x00422E00
            lea     ebx, MsgBuff
            push    ebx
            push    0x05
            call    eax
            add     esp, 0x08
            popad
    }
}
```

[그림 37] 후킹함수의 작동과 주의점 + 부가설명 (핵심내용!!)

자.. 너무 난잡한가? -_-; 필자가 겪은 문제와 해결책을 한꺼번에 설명하려다 보니까 너무많은 부가설명이 들어갔다. 하지만 모조리 습득해야 단 한번이라도 성공할 수 있으니 몇번을 되새김질 하더라도 이해해야 한다.

그림으로 설명하지 못한 부분의 설명

[먼저 스택복구는 무엇인가? 왜 디아블로2 를 후킹하는 D2HackIt 에는 스택복구가 없는데 필자의 소스에만 스택복구가 있는가?]
실제로 디아블로2 소스에는 스택복구가 없다. 왜 일까?
그건 디아블로2 를 리버싱해서 후킹한 사람들이 하나같이 스택관련 명령을 피해서 후킹했기 때문이다. 왜 피했는가? 실제로 필자가 디아블로2 의 후킹된 지점을 찾아보니 mov 명령이 있는 지점이었다. mov 명령 같은 것은 스택을 건드리지 않고 달랑 레지스터에 값만 할당한다. 즉, 디아블로2 를 후킹한 사람들은 레지스터만 건드리지 제어(흐름)에 영향을 미칠만한 코드를 후킹하지 않았다는 소리가 된다. 제어에 영향을 미치지 않는 코드란 무엇일까? 쉘코드 프로그래밍을 해본 경험이 있는가? 쉘코드 프로그래밍을 할 때 0x90(NOP) 코드 대신에 대체 명령을 찾아본 경험이 있다면 이해할 수 있을 것이다. 0x90 은 제어에 아무런 영향을 끼치지 않으며 아무런 작동도 안하는 코드로 알려져있다. 즉, EIP 라고 불리는 명령포인터는 1 을 증가하지만 그 외에는 CPU 가 작동하는데 있어서 아무런 흐름에 영향을 미치지 않는다는 것이다. 많은 해커들이 워게임(War Game:해킹대회를 일컬음) 같은 곳에서 IDS(Intrusion Dectection System:침입탐지시스템) 의 0x90 값 필터링을 우회하려고 EIP 는 증가시키되 CPU 의 흐름은 변경되지 않으면서 레지스터 값 변경을 최소한도로 허용하는 코드를 연구해왔다. 디아블로2 를 후킹하던 사람들과 해커들 둘 간에 공통점이 있는데 알 수 있겠는가.. 둘 다 자신의 코드가 실행되기 전까지 CPU 에게 흐름을 보장받기를 원한다는 점이다.

바이너리들을 리버싱하면서 어셈블리 코드를 읽다보면 중요한 부분에서 스택과 연관이 없는 코드보다 있는게 더 많다는 것을 알 수 있을 것이다. 심지어 함수를 호출할 때 넘어가는 인자를 보라. 스택으로 전달하고 있을 것이다. 함수호출 관계에서 스택은 뺄 수 없으며 버퍼나 혹은 저장변수들이 기본적으로는 스택에 있다고 볼 수 있다. 그런데 스택과 연관있는 부분때문에 복구를 피하려고 후킹지점을 mov 와 같이 독립적인 명령이 있는 부분을 찾아야만 한다면 상당히 융통성도 없고 직관적으로 후킹을 하기도 힘들다. 왜냐면 쫌 후킹해 볼만한 지점이다 싶으면 스택과 연관되어 있고 스택과 연관되지 않은 지점 중에서 값을 가로챌 알맞은 부분을 찾다보면 5 바이트 길이가 안되서 후킹을 전혀 할 수 없는 상황도 발생한다.(현재 D2HackIt 은 5바이트보다 작은 코드를 후킹 할 수 없다. 왜? 인터셉트 방식이 CALL 을 삽입하는 것인데 CALL 은 5바이트 명령이기 때문이다. 앞의 그림35 를 보라. 빨간 박스안에 있는 하나의 명령크기가 5 보다 작기 때문에 두 라인에 걸쳐 두개의 명령을 하나로 보고 후킹지점을 삼은 것이다.) 그래서 필자가 처음 D2HackIt 을 범용적으로 사용하기 위해서 개조할 때 가장 최우선적인 과제로 삼은 것이 스택복구였다. 스택복구라는 가장 큰 문제점이 해결되고나니 바로바로 직관적인 후킹이 가능해졌고 기존에 걸려있던 모든 제한이 풀려버리는 느낌이었다. 이제는 원하는 루틴은 모든지 후킹이 가능하다. 심지어는 원본게임 바이너리의 간단한 함수 한개 정도는 아예 날려버리고(0x90 으로 지워버림) 필자의 후킹함수로 똑같이 코딩해서 그 기능을 대체시킬 수 도 있다.

개념보충 부분은 내용이 난잡하니 꼭 읽을 필요는 없음

[개념보충]
처음 D2HackIt 을 범용적으로 개조하면서 헤멘 부분이 후킹할 지점이 어디냐였다. 주위에서 프로그래밍을 좀 하는 사람들한테 물어보라. 코드 인터셉트 방식의 후킹을 약간 설명만하면 알아듣는 사람이 태반일 것이다. 물론, 그들 방식대로 이해한다. 코드인터셉트 후킹이란 말은 흔히쓰지 않는다. 그들은 하나같이 "아~ 함수를 가로채면 되지.." 라고 아주쉽게 대답할지도 모른다. 그러나 그들에게 당신이 후킹하려는 함수가 어떤 함수라는 것을 알려주고 어떻게 후킹해야 하는지 방법을 설명해달라고 해보라.. 아마도 십중팔구는 후킹하고 싶은 함수를 호출할때 가로채면 되지 않겠냐고 말 할 것이다. 이것이 바로 필자가 겪은 아픔의 발단이 되었다. 모르는 사람의 말은 듣지 말자!! 혼자서 파자!! 필자는 이 때문에 거의 2년 반의 세월을 흘려보냈다. 필자가 공부할때는 서점에서 후킹관련된 책을 찾을 수 도 없었다.(소프트아이스로 점프뛰기 같은 강좌는 언급하지 않겠다.. 필자가 원하는 정보가 아니니까.. 필자는 후킹으로 완벽한 제어를 하고 싶었기에.. -_-;) 주위에 프로그래밍을 잘 하는 사람이나 실력이 뛰어난 사람들한테 모두 물어봐도 같은 대답 뿐이었는데 후킹하려는 함수를 후킹하면 된다는 말 내지는 후킹하고 싶은 함수를 호출할때 가로채면 되지않겠냐 였다. 그래서 필자의 머리통엔 그렇게 박혀버렸었던 것이다. 즉, 개념탈피에 실패했던 것이다. 후킹하고싶은 함수를 후킹하면 되지 않겠냐라는 것은 너무 성의없는 대답이니까 무시하고, 후킹하고 싶은 함수를 호출할때 그 부분에서 가로채면 되지 않겠냐라는 대답에 대해서 언급하자면.. 열라 어이없는 말이니까 믿지 말자!! 필자는 간단한 타켓 프로그램을 먼저 리버싱해놓고 CALL TESTFUNC 같은 부분이 있으면 그 지점을 후킹지점으로 삼아서 시도를 계속해왔었다. 이미 예상되겠지만 실행하자마자 뻑나버리고 단 한번도 성공하지 못했고 그 이유조차 모르고 있었다. 그렇게 시간이 흐르고 또 흘렀다.. -_-; 지금 생각하면 졸라 멍청했다는 생각이 든다. 진작에 디아블로2 파일을 열어봤더라면.. D2HackIt 이 디아블로2 후킹 프로그램이기 때문에 디아블로2 의 바이너리 파일을 열어서 해당 후킹지점을 찾아서 봤으면 진작에 눈치챘을 것을 필자는 D2HackIt 의 소스를 보고는 쉽다고 판단하고 바로 범용으로 적용하려고 팔을 걷어붙인

게 잘못된 시발점이였다. 소스자체가 아주 환상적이라서 필자는 바로 소스를 수정했고 범용적으로 적용하고자 CALL TESTFUNC 같은 CALL 지점만 골라서 후킹을 시도하고 있었던 것이다. 그러다가 한번이라도 성공하면 그 때 부터 자세하게 분석해보자는 심산이었다.
결국 성공하게된 계기는 필자의 전 직업이 공격자였던지라 공격방법을 짜내고 짜내도 너무 방법이 없어서 지친 마음에 그냥 천천히 공부나 때리면서 기술극복이라도 해보자는 생각에 다시 D2HackIt 을 꺼내든 탓이었다. 그날따라 디아블로2 맵핵에 들어있는 DLL 파일이 유독 열어보고 싶어졌고 해당 후킹지점을 찾아 봤을때 아뿔싸! 하는 느낌이 들었다. 그때서야 개념을 잘못 잡았다는 실수를 인식했던 것이다. 그때 봤던 부분이 바로 CALL 이 아니라 7 바이트 mov 명령 이었던 걸로 기억한다. 그때까지 필자는 리버싱하면서 CALL TESTFUNC 라는 것을 찾았을 때 이 호출코드를 가로챘었는데 왜 그랬었는지 지금도 그때는 뭔가에 홀렸었던 것 같다. 기존개념을 다 깨버리고 D2HackIt 소스를 보니까 프로그램이 작살날 수 밖에 없었다는 것을 알 수 있었다.

필자가 주위 사람들에게 물어봐서 한 작업이 왜 실패했을까?
중요한 함수가 TESTFUNC 라고 했을때 호출지점 CALL TESTFUNC 를 가로챘다면?

```
////////////////////////////////////////////////////////////////////////
// GameChatHooker_STUB()
// --------------------------------------------------------------------
////////////////////////////////////////////////////////////////////////
void __declspec(naked) GameChatHooker_STUB()
{
    __asm {
        nop                                 // Make room for original code
        nop
        nop     CALL TESTFUNC 가 복사되어 실행됨
        nop
        nop
        nop
        nop
        nop
```

[그림 38] 실패할 수 밖에 없는 개념

위의 그림38 처럼 가상으로 가정을 하고 보자. 만약 CALL TESTFUNC 라는 부분을 후킹하게되면 그림37 에서 보는 것과 마찬가지 방식으로 후킹지점의 코드가 후킹하는함수 속으로 쏙 복사가 되서 실행된다. GameChatHooker_STUB 이 후킹할 지점의 CALL TESTFUNC 를 복사해서 내부에서 실행하는 것이다. 자.. 그런데 아마 후킹함수가 작동할때 쯤에는 바로 뻑나버릴 것이다. 그 이유는 TESTFUNC 함수가 호출될 때 인자를 취할텐데 push 로 받는다고 가정을 하면 리턴주소가 인자로 넘어갈 것이다.

─── 후킹작동 코드 ───
1) 게임코드의 정상적인 실행...
2) 후킹함수 호출을 만나게됨...
3) CALL GameChatHooker_STUB -> STACK[리턴번지 저장]
4) 후킹함수 안쪽으로 들어감...
4) CALL TESTFUNC -> TESTFUNC 함수가 인자를 취하면 위에서 저장된 STACK[리턴번지 저장] 요놈이 TESTFUNC 함수의 첫번째 인자가(뻑나는 이유) 되버린다. 인자를 취하지 않는 void 방식이라고 할 경우에도 중요한 점이 있다. 그 것은 CALL 이 실행되면 레지스터가 변경된다. 그림37 에서 보는 것처럼 pushad 와 popad 로 감싸지 않으면 레지스터는 뒤죽박죽이 되어 나중에 원래 게임실행코드로 돌아갈때 작살난다. 만약에 pushad 와 popad 로 둘러싸는 코드를 넣는다면?

```
__asm {
    pushad          // pushad 와 popad 안에 CALL TESTFUNC 가 들어가도록 미리 자리를 마련해 둔다.
    nop             <── CALL TESTFUNC 명령복사 (Intercept 함수가 복사하는 역할을 함)
    nop
    nop
    nop
    nop
    popad
}
```

위와 같이 되려면 Intercept 함수를 수정해야 된다. 즉, Intercept 함수가 후킹함수 주소(GameChatHooker_STUB 주소)의 첫부분에 복사를 하지 말고 pushad 명령만큼을 뒤로 건너띄고 복사하도록 코드를 수정해야한다. 이렇게 했다고 가정을 했을때도 또 문제가 있다. 위의 경우에는 TESTFUNC 함수가 무조건 인자를 취하지 않는 void 형 함수라야지만 가능한 얘기다. 만약에 CALL TESTFUNC 를 할때 TESTFUNC 함수가 인자를 취하는 놈이라면? pushad 명령뒤에 0x20 만큼 스택을 건너뛰고 원래의 함수인자를 뺄 수 있도록 스택보정을 해주는 스택복구 코드가 그림37과 똑같이 들어가야한다. 더 골때리게 되는건 Intercept 함수가 정확한 위치에 복사하기 위해서 동적으로 복사를 해줘야된다. TESTFUNC 함수에 넘어가는 인자들을 계산해서 때려주는 것도 그렇고 동적 Intercept 함수를 구현하는 것도 그렇고 한마디로 아주 효율성없는 뻘짓이 된다. 아마 이 글을 읽는 사람들도 머리가 꼬일 것이다. 아주 골때리는 현상이 나타난다는 소리다. 결국 그림37 과 같은 형태의 모습이 가장 최적화된 모습이라는 소리다. 처음에 필자가 착각을 했었던 것은 한마디로 개념이 없었기 때문이다. CALL 코드는 복사되면 안된다. 그렇다면.. 복사되서는 안되는 코드가 몇이 있을 것이다. CALL, JMP(점프계열 명령), RET(리턴명령) 등등 프로그램의 흐름을 바꾸는 명령이 복사되면 안된다. 생각해보라 JMP 가 포함된 지점을 후킹지점으로 삼는다면 이게 후킹함수 내부로 복사되서 엉뚱한 지점으로 강제로 점프하게 됨으로써 뻑나게 된다. 또한 조건문이 복사되서 함수 내부에서 플래그가 변경될 수도 있다. 이 경우 원본게임코드로 복귀했을때 조건이 바뀌어져 버려서 조건점프가 안될 수 도 있다는 점을 생각해야 한다.

[0x00422E00 은 도대체 어디서 튀어나온 놈일까?]
눈치빠른 사람들은 그림29 에서 ChatPrint 함수의 빨간박스를 아직 기억하고 있을까 모르겠다...

```
                        ; int __cdecl ChatPrint(char,char *,char)
                        ChatPrint       proc near               ; CODE XREF: sub_407D60+75↑p
                                                                ; sub_407E20+72↑p
                                                                ; sub_407ED0+CD↑p
                                                                ; sub_40AA80+72↑p
                                                                ; sub_40C410+8D↑p ...

                        var_204         = byte ptr -204h
                        var_4           = dword ptr -4
                        arg_0           = byte ptr  4
                        arg_4           = dword ptr  8
                        arg_8           = byte ptr  0Ch

81 EC 04 02 00 00                       sub     esp, 204h
A1 84 36 62 00                          mov     eax, dword_623684
8B 8C 24 0C 02 00 00                    mov     ecx, [esp+204h+arg_4]
89 84 24 00 02 00 00                    mov     [esp+204h+var_4], eax
8D 84 24 10 02 00 00                    lea     eax, [esp+204h+arg_8]
50                                      push    eax             ; va_list
51                                      push    ecx             ; char *
8D 54 24 08                             lea     edx, [esp+20Ch+var_204]
68 00 02 00 00                          push    200h            ; size_t
52                                      push    edx             ; char *
E8 DC 59 18 00                          call    __vsnprintf
8B 8C 24 18 02 00 00                    mov     ecx, dword ptr [esp+214h+arg_0]
8D 44 24 10                             lea     eax, [esp+214h+var_204]
50                                      push    eax             ; unsigned __int8 *
51                                      push    ecx             ; char
E8 4D FF FF FF                          call    sub_422E00
8B 8C 24 18 02 00 00                    mov     ecx, [esp+21Ch+var_4]
E8 40 4E 18 00                          call    sub_5A7CFF
81 C4 1C 02 00 00                       add     esp, 21Ch
C3                                      retn
                        ChatPrint       endp
```

[그림 39] 0x00422E00 이 보이는가?

sub_422E00 함수는 채팅창 출력함수에서 이미 보여주었던 부분인데 일부러 언급을 안했었다.
이 함수의 역할은 ChatPrint 함수내에서 실제 채팅창 출력을 담당하는 함수이다.
그림39 에서 파란색 박스부분이 바로 다음의 그림40 에서 인라인 어셈블리로 구현되어 있다.

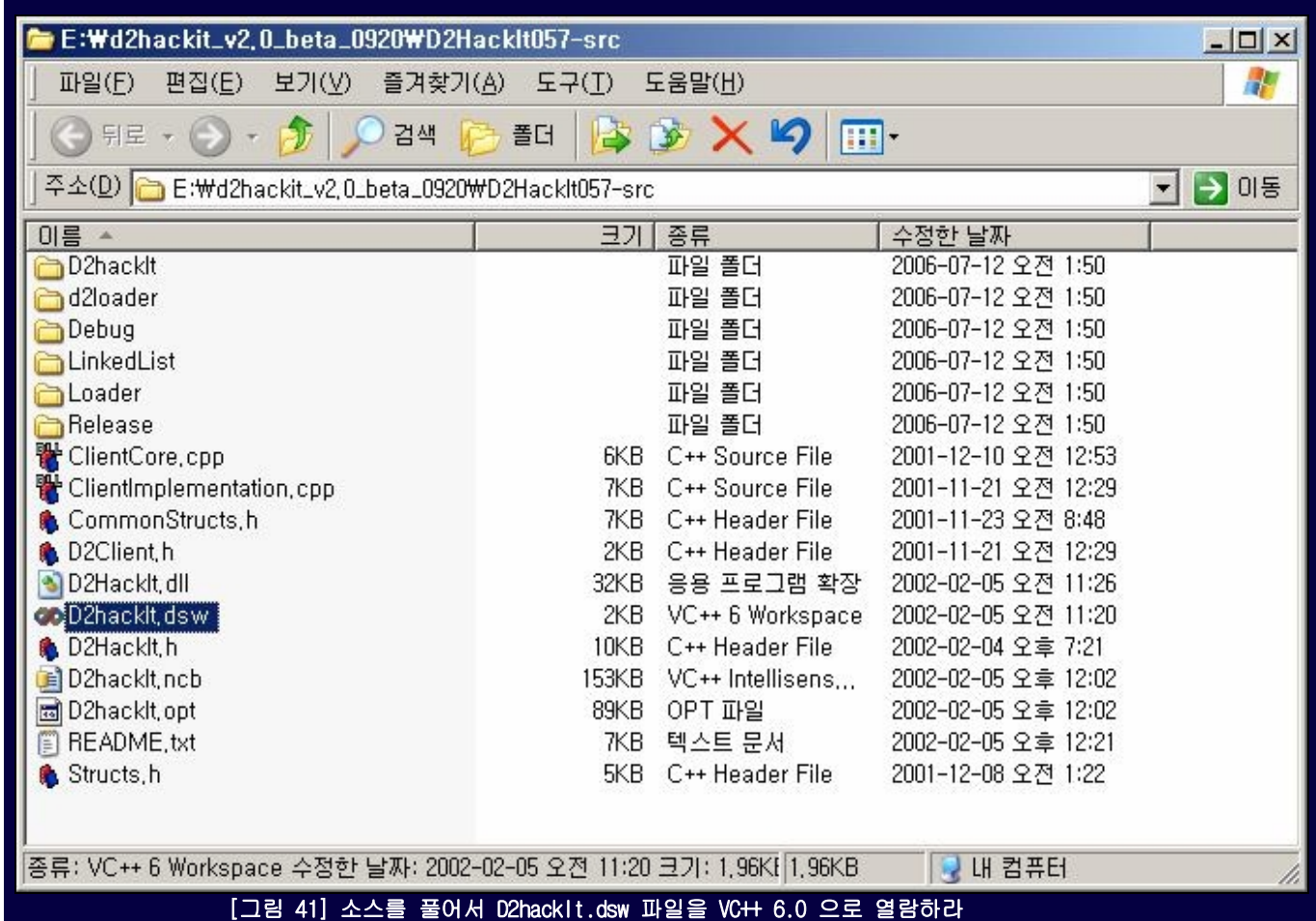
```
char MsgBuff[2048] = "님! 죄송한데요.. 뭐 좀 하나만 물어봐도 될까요?";

__asm {
        pushad
        mov     eax, 0x00422E00
        lea     ebx, MsgBuff
        push    ebx
        push    0x05
        call    eax
        add     esp, 0x08
        popad
}
```

[그림 40] 채팅출력창 함수를 강제로 호출

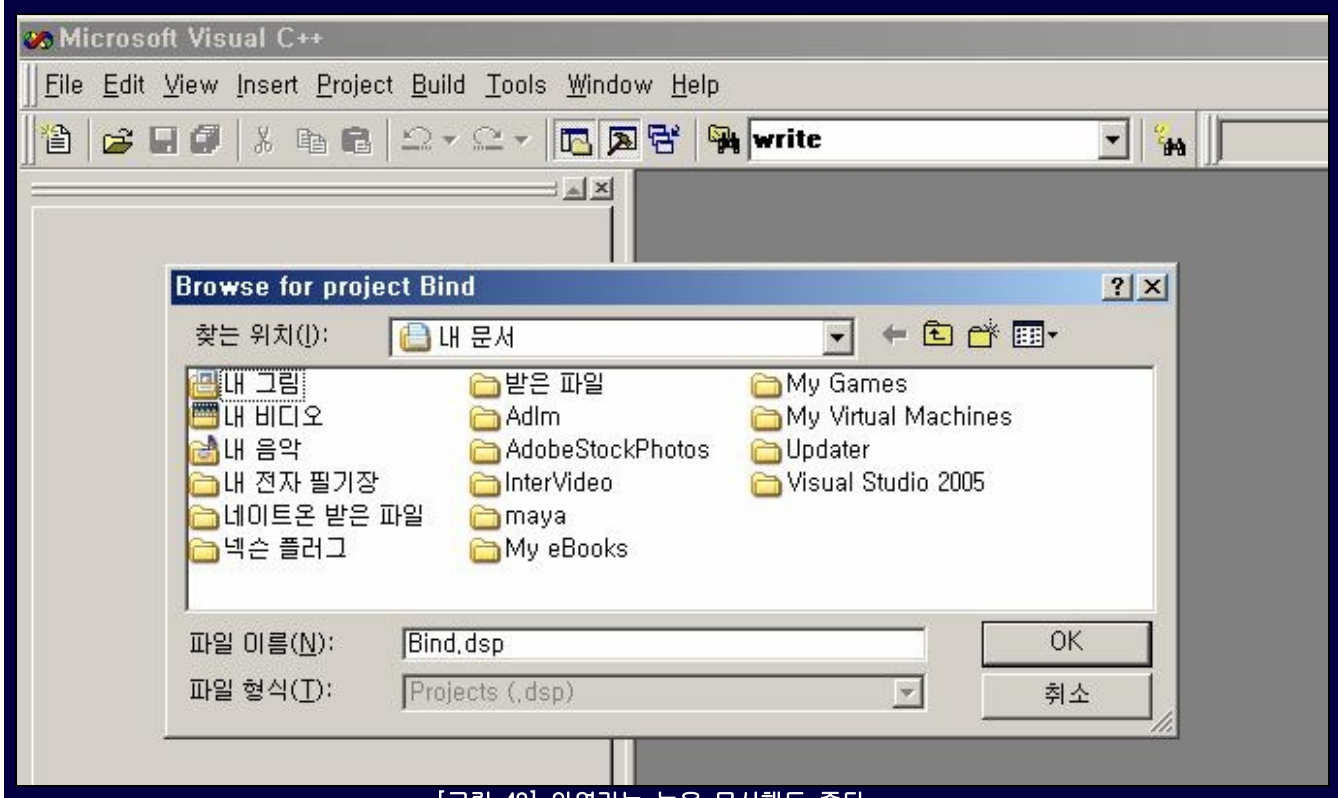
한가지 궁금할텐데 왜 ChatPrint 함수를 호출하지 않고 0x00422E00 을 호출했을까? 그림39 의 ChatPrint 함수는 vsnprintf 도 보이고 va_list 도 보이는 것이 꼭 그 구조가 전형적인 로깅 루틴처럼 생겼다. 실제로 필자가 리버싱할때 "분명히 이 부분인거 같은데 왜 로깅함수가 존재하는 걸까?" 하고 계속해서 헷갈리게 고생시킨 놈이였다. 그런데 계속 채팅창 출력함수가 불려야 하는 위치에서 사용되고 있었기 때문에 진짜 맞는지 테스트를 하기 위해서 통째로 ChatPrint 함수를 호출시켜 보았다. 그런데 문제는 ChatPrint 함수를 호출하려면 그림39 의 va_list 와 바로 다음 라인에 존재하는 ecx 레지스터(char* 형) 처럼 인자를 맞춰넣고 호출해야만 했다. 필자는 vsnprintf 함수를 이미 알고 있었기에 포맷스트링 형식을 맞춰서 넣어줘야 한다는 것을 알 수 있었다. 그러나 이를 맞추기가 영~ 고생이였다. 그래서 그냥 뒤쪽의 0x00422E00 함수를 호출해보니 정상적으로 채팅창에 출력이 되는 것이었다. 그래서 굳이 ChatPrint 함수 전체를 호출하지 않고 그 안에 있는 0x00422E00 함수를 직접 호출하기로 한 것이다.

코드작성

현재까지 설명한 것을 모두 코드로 옮겨 적어본다. 우리는 새로운 코드를 작성하지 않을 것이며 인터넷이 떠돌아다니는 원본 D2HackIt057-src.zip (D2HackIt 0.57버전) 소스코드를 가져다가 커스터마이징 할 것이다.

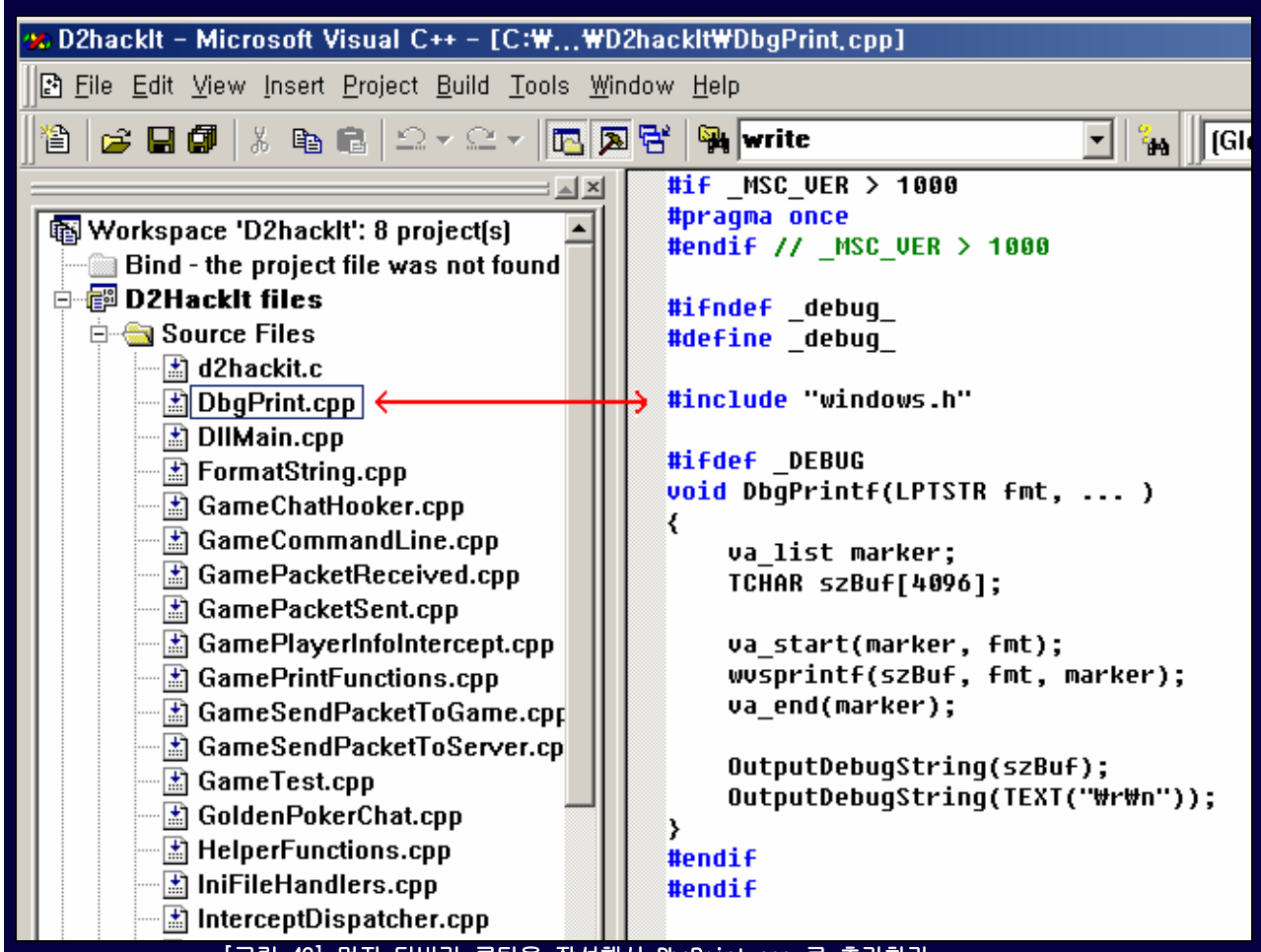


[그림 41] 소스를 풀어서 D2HackIt.dsw 파일을 VC++ 6.0 으로 열람하라

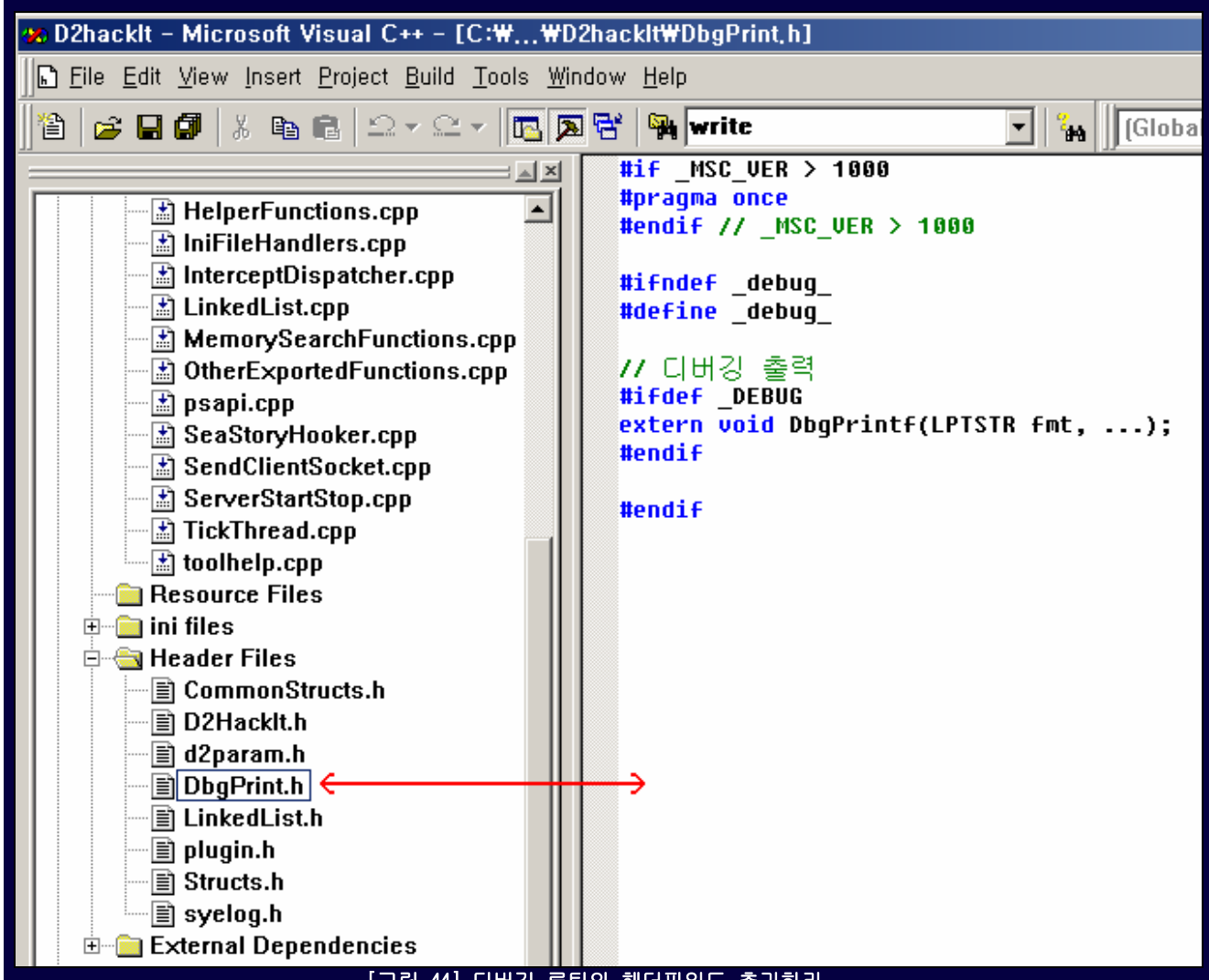


[그림 42] 안열리는 놈은 무시해도 좋다.

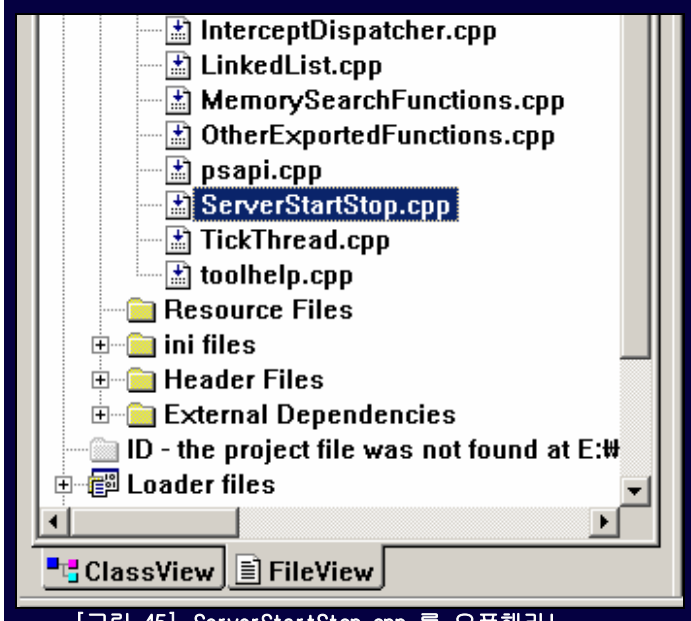
안열리는 프로젝트가 있다고 경고가 나오면 무시해라. 플러그인이다. 우리한텐 전혀 필요없다.



[그림 43] 먼저 디버깅 루틴을 작성해서 DbgPrint.cpp 로 추가하라



[그림 44] 디버깅 루틴의 헤더파일도 추가하라



[그림 45] ServerStartStop.cpp 를 오픈해라!

다음부터는 ServerStartStop.cpp 파일 내에서만 수정함

```
//////////////////////////////////////////////////////////////////////
// ServerStartStop.cpp
// ------------------------------------------------------------------
// Initialize/destroy server.
//
// <thohell@home.se>
//////////////////////////////////////////////////////////////////////
#define THIS_IS_SERVER
#include "..₩D2HackIt.h"

// These are the dll's we want to force-load to get them in memory.
//char* NeededDlls[] = { "D2Common.dll", "D2Game.dll", "D2Multi.dll", "D2Client.dll", NULL };
char* NeededDlls[] = { "D2Common.dll", "D2Game.dll", "D2Client.dll", NULL };
//////////////////////////////////////////////////////////////////////
// ServerStart()
// ------------------------------------------------------------------
```

[그림 46] NeededDlls 에 디아블로2 관련 dll 들을 제거하라!

```
#define THIS_IS_SERVER
#include "..₩D2HackIt.h"

// These are the dll's we want to force-load to get them in memory.
//char* NeededDlls[] = { "D2Common.dll", "D2Game.dll", "D2Multi.dll",
char* NeededDlls[] = { NULL };
//////////////////////////////////////////////////////////////////////
// ServerStart()
// ------------------------------------------------------------------
```

[그림 47] 그림46 을 NULL 로 바꾼 모습

```
    // Get plugin path
    t=new char[_MAX_PATH];
    if (!GetModuleFileName((HINSTANCE)hModule, t, _MAX_PATH))
        { MessageBox(NULL, "Unable to get PluginPath!", "D2Hackit Error!", MB_ICONERROR); return FALSE; }
    int p=strlen(t);
    while (p)
    {
        if (t[p] == '₩₩')
            { t[p] = 0; p=0;}
        else
            p--;
    }
```

[그림 48] MessageBox 를 DbgPrintf 함수로 바꿔라 (왜? 에러가 나도 프로그램의 블록킹을 없애기 위해서.. 그냥..)

```
    // Get plugin path
    t=new char[_MAX_PATH];
    if (!GetModuleFileName((HINSTANCE)hModule, t, _MAX_PATH))
    {
        #ifdef _DEBUG
        DbgPrintf("Unable to get PluginPath!");
        #endif

        return FALSE;
    }
    int p=strlen(t);
    while (p)
    {
        if (t[p] == '₩₩')
            { t[p] = 0; p=0;}
        else
            p--;
    }
```

[그림 49] 그림48 의 MessageBox 를 DbgPrintf 함수로 대체한 화면

```
    // Get the process ID and the process handle
    psi->pid = GetCurrentProcessId();
    psi->hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, psi->pid);
        if (!psi->hProcess) { MessageBox(NULL, "Can't get Diablo II's process handle.", "D2Hackit Error!", MB_ICONERROR); return FAL


    // Get build date/time
    strcpy(psi->BuildDate, __DATE__);
    strcpy(psi->BuildTime, __TIME__);

    ////////////////////////////////////////////////////////////////////////
    // Build initial callbacks in the FUNCTIONENTRYPOINTS structure.
    ////////////////////////////////////////////////////////////////////////
//  fep->GetMemoryAddressFromPattern=&GetMemoryAddressFromPattern;
    fep->GamePrintString=&GamePrintString;
    fep->GamePrintInfo=&GamePrintInfo;
    fep->GamePrintVerbose=&GamePrintVerbose;
    fep->GamePrintError=&GamePrintError;
    fep->GetHackProfileString=&GetHackProfileString;
```

[그림 50] 불필요한 부분을 주석처리하고 MessageBox 를 DbgPrintf 함수로 바꿔라

```
    // Get the process ID and the process handle
    psi->pid = GetCurrentProcessId();
    psi->hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, psi->pid);
    if (!psi->hProcess) {
        #ifdef _DEBUG
        DbgPrintf("Can't get Game's process handle.");
        #endif

        return FALSE;
    }


    // Get build date/time
/*
    strcpy(psi->BuildDate, __DATE__);
    strcpy(psi->BuildTime, __TIME__);
*/

    ////////////////////////////////////////////////////////////////////////
    // Build initial callbacks in the FUNCTIONENTRYPOINTS structure.
    ////////////////////////////////////////////////////////////////////////
//  fep->GetMemoryAddressFromPattern=&GetMemoryAddressFromPattern;
//  fep->GamePrintString=&GamePrintString;
//  fep->GamePrintInfo=&GamePrintInfo;
//  fep->GamePrintVerbose=&GamePrintVerbose;
//  fep->GamePrintError=&GamePrintError;
    fep->GetHackProfileString=&GetHackProfileString;
```

[그림 51] 그림50 을 수정한 화면

```
 ////////////////////////////////////////////////////////////////////////
 // Check if D2HackIi.ini exisits
 ////////////////////////////////////////////////////////////////////////
 if (_access(psi->IniFile, 0))
 {
     LPSTR t=new char[strlen(psi->IniFile)+50];
     sprintf(t, "Unable to open ini-file:\n%s", psi->IniFile);
     MessageBox(NULL, t, "D2Hackit Error!", MB_ICONERROR);
     delete t;
     return FALSE;
 }
```

[그림 52] MessageBox 를 DbgPrintf 로 바꾼다.

```
////////////////////////////////////////////////////////////////////////
// Check if D2HackIi.ini exisits
////////////////////////////////////////////////////////////////////////
if (_access(psi->IniFile, 0))
{
    LPSTR t=new char[strlen(psi->IniFile)+50];
    sprintf(t, "Unable to open ini-file:\n%s", psi->IniFile);
    #ifdef _DEBUG
    DbgPrintf(t);
    #endif

    delete t;
    return FALSE;
}
```

[그림 53] 그림52 를 수정한 모습

자 이제 그림53 의 바로 다음라인부터 return TRUE; 이전까지 모든 코드를 삭제하라. (ServerStart 함수 밖에까지 벗어나서 지우면 곤란하다.. ㅎㅎㅎ -_-;)
그리고 다음의 새로운 코드를 삽입한다. (잘 알아 볼 수 있도록 그림53 의 일부분이 포함되어 있으니까 추가할때 중복추가는 하지말자..)

```
//////////////////////////////////////////////////////////////////////
if (_access(psi->IniFile, 0))
{
    LPSTR t=new char[strlen(psi->IniFile)+50];
    sprintf(t, "Unable to open ini-file:\n%s", psi->IniFile);
    #ifdef _DEBUG
    DbgPrintf(t);
    #endif

    delete t;
    return FALSE;
}

if (!GetFingerprint("D2HackIt", "GameChatHooker", psi->fps.GameChatHooker))
{
    #ifdef _DEBUG
    DbgPrintf("GameChatHooker Not Found!");
    #endif

    return FALSE;
}

if (!GetFingerprint("D2HackIt", "SecurityRemoverTimer", psi->fps.SecurityRemoverTimer))
{
    #ifdef _DEBUG
    DbgPrintf("SecurityRemoverTimer Not Found!");
    #endif

    return FALSE;
}

if (!GetFingerprint("D2HackIt", "SecurityRemoverDialog", psi->fps.SecurityRemoverDialog))
{
    #ifdef _DEBUG
    DbgPrintf("SecurityRemoverDialog Not Found!");
    #endif

    return FALSE;
}

// "E88820FEFF59598B4C240CE8FAFEFFFC21400"
//                       "9090909090"
// "B9A06F6400E885FD1900B9A06F6400E805071A0085C07503"
//                                "9090909090909090EB03"

unsigned char lpSource1[] = "\x90\x90\x90\x90\x90\x90";
unsigned char lpSource2[] = "\x90\x90\x90\x90\x90\x90\x90\x90\xEB\x03";

DWORD lpDest1 = psi->fps.SecurityRemoverTimer.AddressFound;
DWORD lpDest2 = psi->fps.SecurityRemoverDialog.AddressFound;

int len1 = 0x05;
int len2 = 0x09;

DWORD oldSourceProt,oldDestProt=0;

VirtualProtect((void*)lpSource1,len1,PAGE_EXECUTE_READWRITE,&oldSourceProt);
    VirtualProtect((void*)lpDest1,len1,PAGE_EXECUTE_READWRITE,&oldDestProt);
        memcpy((void*)lpDest1,(void*)&lpSource1,len1);
    VirtualProtect((void*)lpDest1,len1,oldDestProt,&oldDestProt);
VirtualProtect((void*)lpSource1,len1,oldSourceProt,&oldSourceProt);

VirtualProtect((void*)lpSource2,len2,PAGE_EXECUTE_READWRITE,&oldSourceProt);
    VirtualProtect((void*)lpDest2,len2,PAGE_EXECUTE_READWRITE,&oldDestProt);
        memcpy((void*)lpDest2,(void*)&lpSource2,len2);
    VirtualProtect((void*)lpDest2,len2,oldDestProt,&oldDestProt);
VirtualProtect((void*)lpSource2,len2,oldSourceProt,&oldSourceProt);

Intercept(INST_CALL, psi->fps.GameChatHooker.AddressFound, (DWORD)&GameChatHooker_STUB, psi->fps.GameChatHooker.PatchSize);

return TRUE;
}

//////////////////////////////////////////////////////////////////////
// ServerStop()
// ---------------------------------------------------------------
// Responsible for stopping the server.
//////////////////////////////////////////////////////////////////////
BOOL PRIVATE ServerStop(void)
{
```

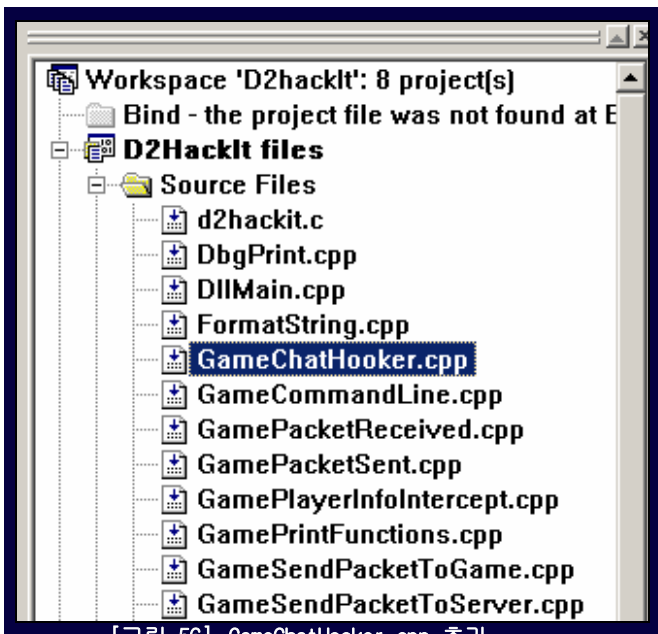
[그림 54] 필요없는 코드를 싹 지워버리고 새로 삽입한 코드

위와 같이 ServerStart 함수에 삽입했으면 이제 ServerStop 함수도 수정해야한다.

```
//////////////////////////////////////////////////////////////////////
// ServerStop()
// ---------------------------------------------------------------
// Responsible for stopping the server.
//////////////////////////////////////////////////////////////////////
BOOL PRIVATE ServerStop(void)
{
    // Un-patch intercept locations
    Intercept(INST_CALL, (DWORD)&GameChatHooker_STUB, psi->fps.GameChatHooker.AddressFound, psi->fps.GameChatHooker.PatchSize);

    // Release dll's that we loaded upon entry.
    for (int i=0; NeededDlls[i] != NULL; i++) FreeLibrary (GetModuleHandle(NeededDlls[i]));

    delete (LPSTR)si->PluginDirectory, (LPSTR)psi->IniFile, thisgame, pfep, fep, psi, si;

    return TRUE;
}
```

[그림 55] ServerStop 함수를 위와 같이 바꿔준다. (불필요한 루틴을 제거함)

이제 ServerStartStop.cpp 파일의 수정은 끝났다.
다음은 실질적인 후킹루틴으로써 GameChatHooker.cpp 파일명으로 작성될 것이다. VC++ 의 프로젝트에 추가하라.



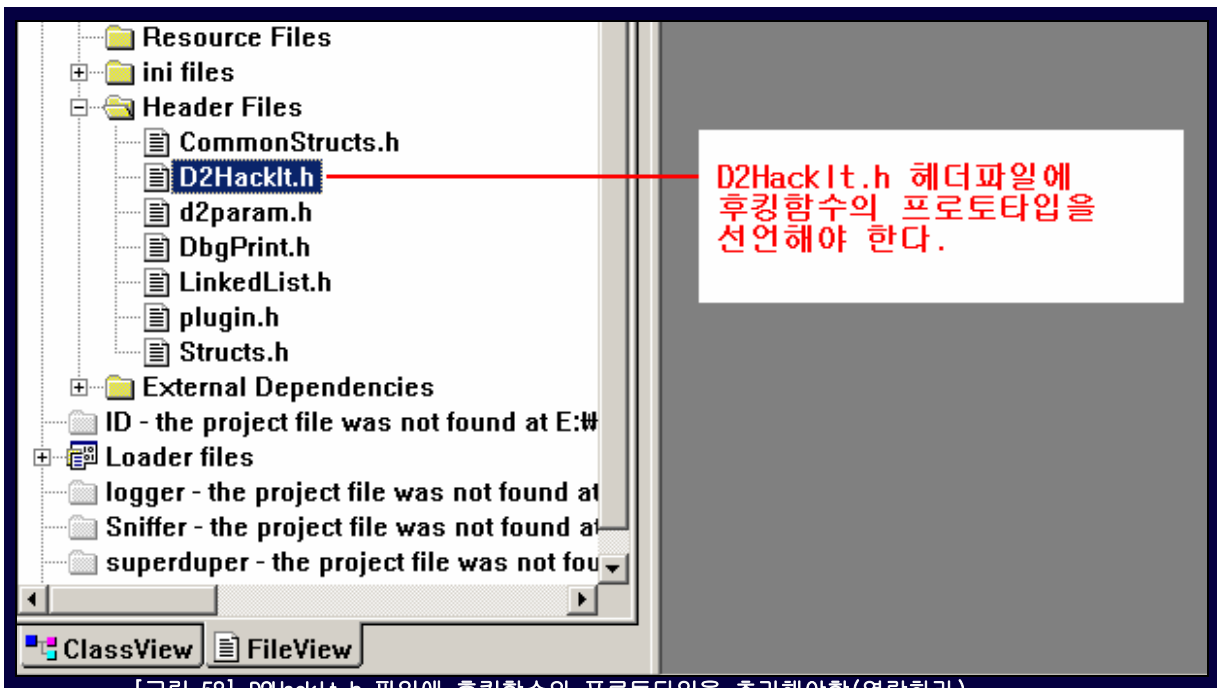
[그림 56] GameChatHooker.cpp 추가

GameChatHooker.cpp 파일의 내용은 다음 그림과 같이 작성해서 위 그림56 처럼 프로젝트에 추가하면 된다.

```
/*
    Programming : Reversing by AmesianX in powerhacker.net
*/

//////////////////////////////////////////////////////////////////////////
#define THIS_IS_SERVER
#include "..₩D2HackIt.h"
#include "DbgPrint.h"
//////////////////////////////////////////////////////////////////////////
// GameChatHooker()
//////////////////////////////////////////////////////////////////////////
void __fastcall GameChatHooker(char *Chat)
{
    #ifdef _DEBUG
    DbgPrintf("GameChatHooker = %s₩n", Chat);
    #endif

    char MsgBuff[2048] = "님! 죄송한데요.. 뭐 좀 하나만 물어봐도 될까요?";

    __asm {
            pushad
            mov     eax, 0x00422E00
            lea     ebx, MsgBuff
            push    ebx
            push    0x05
            call    eax
            add     esp, 0x08
            popad
    }
}

//////////////////////////////////////////////////////////////////////////
// GameChatHooker_STUB()
// ---------------------------------------------------------------------
//////////////////////////////////////////////////////////////////////////
void __declspec(naked) GameChatHooker_STUB()
{
    __asm {
        nop                                 // Make room for original code
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop

//      .text:00422F6E 8B 45 08                 mov     eax, [ebp+arg_0]
//      .text:00422F71 6A 00                    push    0

        // 스택복구 처리
        pushad
        mov     edx, dword ptr [esp+0x20+0x04]
        mov     ebx, dword ptr [esp+0x20]

        mov     dword ptr [esp+0x20+0x04], ebx
        mov     dword ptr [esp+0x20], edx

        mov     ecx, eax
        CALL    GameChatHooker
        popad

        ret
    }
}
```

[그림 57] GameChatHooker.cpp 파일내용

자.. 이제 굵직굵직한 작업들은 거의 끝났다. 나머지는 졸라 짜증나는 작업만 남았다.



[그림 58] D2HackIt.h 파일에 후킹함수의 프로토타입을 추가해야함(열람하라)

```
// GamePrintFunctions.cpp
BOOL    EXPORT GamePrintInfo(LPCSTR buf);
BOOL    EXPORT GamePrintError(LPCSTR buf);
BOOL    EXPORT GamePrintVerbose(LPCSTR buf);
BOOL    EXPORT GamePrintString(LPCSTR buf);

// GameChatHooker.cpp
void __fastcall GameChatHooker(char *Chat);
void GameChatHooker_STUB();

// GamePacketReceived.cpp
DWORD __fastcall GamePacketReceivedIntercept(BYTE* aPacket, DWORD aLength);
void GamePacketReceivedInterceptSTUB();

// GamePacketSent.cpp
DWORD __fastcall GamePacketSentIntercept(BYTE* aPacket, DWORD aLength);
void GamePacketSentInterceptSTUB();

// GameSendPacketToServer.cpp
BOOL EXPORT GameSendPacketToServer(LPBYTE buf, DWORD len);
```

프로토타입 추가

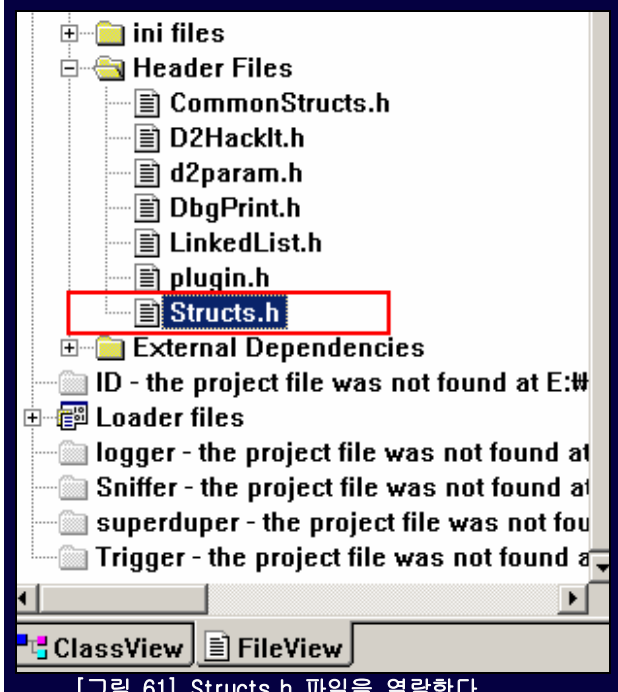
[그림 59] D2HackIt.h 파일을 열람해서 프로토타입을 추가한 모습

```
if (!GetFingerprint("D2HackIt", "GameChatHooker", psi->fps.GameChatHooker))
{
    #ifdef _DEBUG
    DbgPrintf("GameChatHooker Not Found!");
    #endif

    return FALSE;
}
```

fps 구조체에 GameChatHooker
변수를 추가해준적이 없으므로
컴파일 에러가 날 것이다.

[그림 60] fps 구조체에 핑거프린트 된 주소를 담을 변수를 만들어 줘야한다.

GetFingerprint 함수를 사용하면서 구조체를 지정해주는데 선언한 적이 없으므로 컴파일 에러가 난다.
선언은 Structs.h 파일에 구조체가 선언되어 있으므로 그 곳에 추가하면 된다.

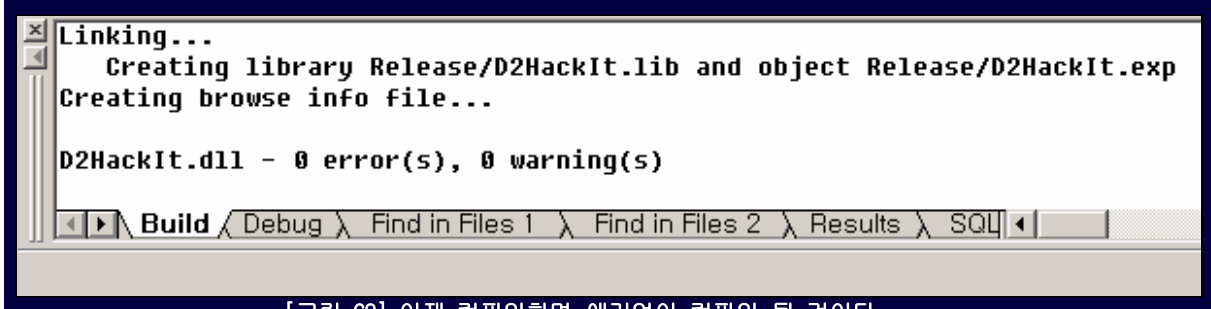


[그림 61] Structs.h 파일을 열람한다

```
    // Fingerprinted information
    DWORD           GameSocketLocation;
    DWORD           GamePrintStringLocation;
    DWORD           GameKeyDownIntercept;
    DWORD           pPlayerInfoStruct;
    DWORD           GameSendPacketToGameLocation;
    struct {
        FINGERPRINTSTRUCT   GamePacketReceivedIntercept;
        FINGERPRINTSTRUCT   GamePacketSentIntercept;
        FINGERPRINTSTRUCT   GamePlayerInfoIntercept;
        FINGERPRINTSTRUCT   GameChatHooker;
        FINGERPRINTSTRUCT   SecurityRemoverTimer;
        FINGERPRINTSTRUCT   SecurityRemoverDialog;
    } fps;
} PRIVATESERVERINFO;
extern PRIVATESERVERINFO            *psi;
```

핑거프린트 구조체 변수 세개를 추가해줌

[그림 62] FINGERPRINTSTRUCT 구조체 세개를 추가해준다.



[그림 63] 이제 컴파일하면 에러없이 컴파일 될 것이다.

휴~ 졸라 힘들다... 이제 마무리 작업을 해야한다. 제일 빡신작업이 하나 남았는데 아주 중요한 작업이다.

```
                fps.PatchSize=atoi(&szReturnString[i+1]);
                break;
            }
        }
    }
    strcpy(fps.ModuleName, szReturnString);
    delete szReturnString;


    if ((fps.AddressFound=GetMemoryAddressFromPattern(fps.ModuleName, fps.FingerPrint, fps.Offset)) < 0x100)
    {
        if (psi->DontShowErrors)
            return FALSE;

        t=new char[256];
        sprintf(t, "Unable fo find location for '%s'.",
            szFingerprintName, szHackName);
        fep->GamePrintError(t);
        delete t;
        return FALSE;
    } else {
        t=new char[256];
        sprintf(t, "Found '%s' at %.8x",
            szFingerprintName, fps.AddressFound);
        fep->GamePrintVerbose(t);
        delete t;
        return TRUE;
    }
}
```

GamePrintError
GamePrintVerbose
이 GamePrint 계열 함수는 모두 디아블로2 의 채팅창함수를 후킹해서 메세지를 출력해주기 때문에 컴파일이 잘 됐다고 해도 이 함수들이 있지도 않은 디아블로2 채팅창함수를 호출해대는 바람에 프로그램이 개작살 나게 되었다.

fep->GamePrintVerbose(t) 를 DbgPrintf(t) 로 바꿔준다.
물론 제거할 수도 있지만 이왕이면 출력하는게 좋다. 그렇기 때문에 DbgPrintf 함수를 미리 만들어 놓은 것이다.

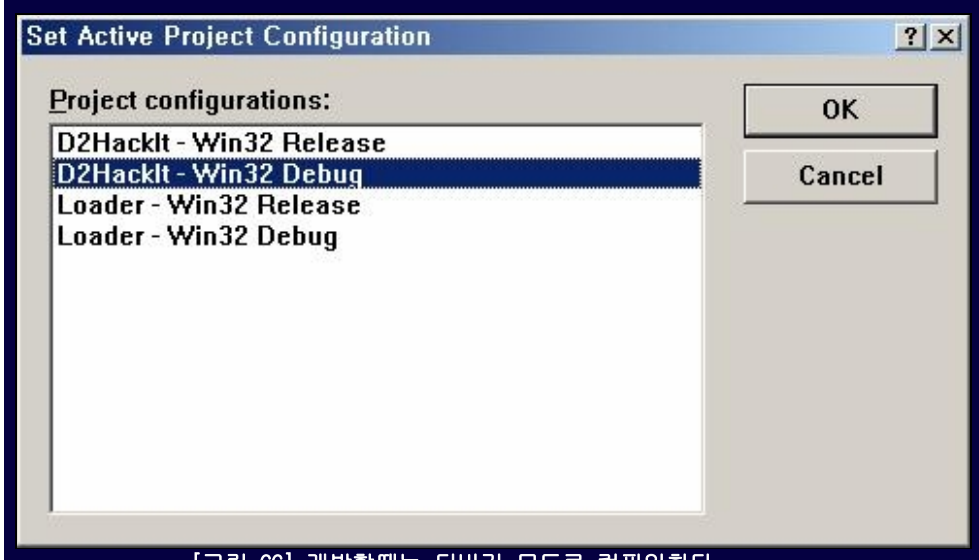
[그림 64] 디아블로2 에 최적화된 GamePrint 계열 함수를 교체하기

VC++ 의 파일검색 기능을 이용해서 GamePrint 로 찾아보라... 엄청많이 나온다. 대략 50 여 군데가 나온다. 필자는 선별해서 변경하는게 짜증나서 그냥 전부 DbgPrintf 함수로 바꿔버렸다. 똑똑한 사람은 아예 GamePrint 계열 함수를 뜯어 고치는 방법을 사용할 것이다. 필자는 그 코딩이 더 짜증나서 그냥 검색으로 해당지점을 모두 변경하였다. 심지어는 MessageBox 출력도 DbgPrintf 로 출력하게 모두 바꿨다. 왜냐면 게임을 실행시키고 후킹을 걸다가 빡나면 메시지 박스가 뜨면서 버벅대는 바람에 정신을 못차리는 현상이 간혹있었기 때문이다. 독자들도 출력관련 함수들을 모두 DbgPrintf 로 변경하는 것을 추천한다.

```
// D2HackIt.h
// ------------------------------------------------------------------
// Main incude file for D2HackIt.
//
// <thohell@home.se>
//////////////////////////////////////////////////////////////////////
#pragma once
#include <windows.h>
#include <iostream>
#include <io.h>
#include <stdio.h>
#include <tlhelp32.h>
#include "LinkedList₩LinkedList.h"
#include "DbgPrint.h"

#ifdef __cplusplus
extern "C" {            /* Assume C declarations for C++ */
#endif  /* __cplusplus */
```

[그림 65] D2HackIt.h 파일의 헤더 선언부에 DbgPrint.h 를 추가해야 된다.

출력함수를 DbgPrintf 로 변경하고 나면 몇몇놈이 선언이 안됐다고 에러를 낼텐데 D2HackIt.h 헤더파일에 디버그함수의 헤더파일을 추가해주면 에러가 안날 것이다. 자.. 이제 다 변경하고 컴파일 했는가? 디버깅 메시지를 보려면 VC++ 에서 디버깅모드로 컴파일 해야된다. DbgPrintf 함수를 사용할때 #ifdef _DEBUG 조건을 걸어놔서 디버깅 모드가 아니면 출력이 안될 것이다.



[그림 66] 개발할때는 디버깅 모드로 컴파일한다.

이제 D2HackIt 0.57 버전의 원본소스를 우리것으로 커스터마이징 하는 작업이 끝났다. 모든 코드작성이 끝났고 위에서 언급했었던 환경설정파일을 만들어줘야 한다. D2HackIt.ini 라는 파일을 만들어서 이미 앞에서 다 설명했었던 후킹할 지점에 관한 바이너리스트링 정보를 적는다. (유전자 식별정보같군.. 그런게 있나몰라.. -_-)

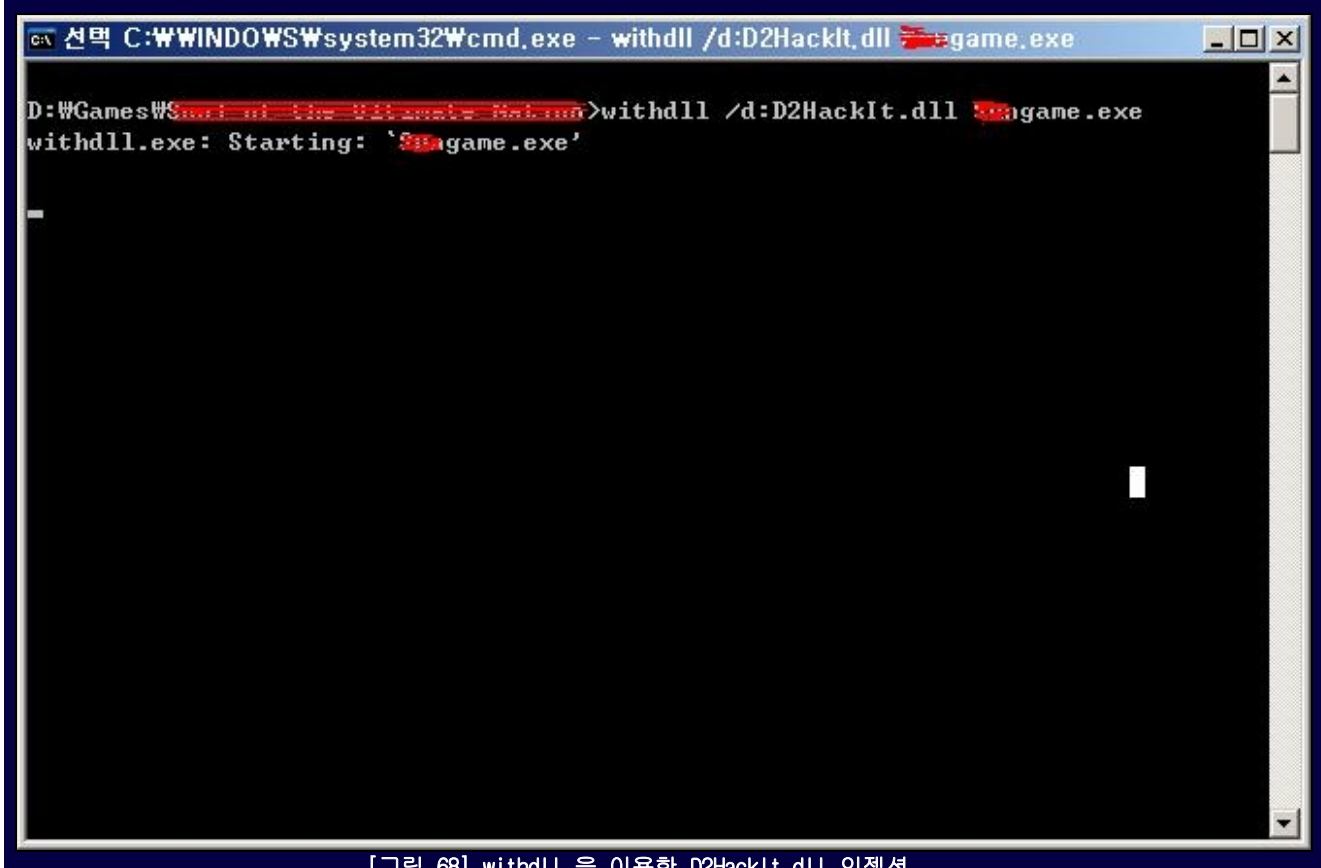
```
[FingerprintData]
; GameChatHooker
GameChatHooker=[redacted]game.exe,5,21,33C0C644240800B97F0000008D7C2409F3AB66ABAA8B45086A00
SecurityRemoverTimer=[redacted]game.exe,5,11,E88820FEFF59598B4C240CE8FAFEFFFFC21400
SecurityRemoverDialog=[redacted]game.exe,9,15,B9A06F6400E885FD1900B9A06F6400E805071A0085C07503
```

[그림 67] D2HackIt.ini 파일의 내용

(참고: SecurityRemover 부분의 패치사이즈는 실제로 코딩에서 아무런 의미가 없다. 왜? 코딩부분을 보면 Intercept 함수를 사용하지 않기 때문이다. 대신에 GetFingerprint 함수로 보안루틴을 패치할 지점을 찾기 위해서 오프셋과 바이너리스트링은 필요하다. 코드를 굳이 설명하지 않아도 울트라에디트로 수정하는 것과 완전히 같은 작업을 메모리를 대상으로 하고있다는 것을 알 것이다. 해당 코드조차 분석할 수 없다면 이 문서는 독자에게 아무런 지식도 전달할 수 없다. 정공학(프로그래밍)부터 관심을 가져야될 것이다.)

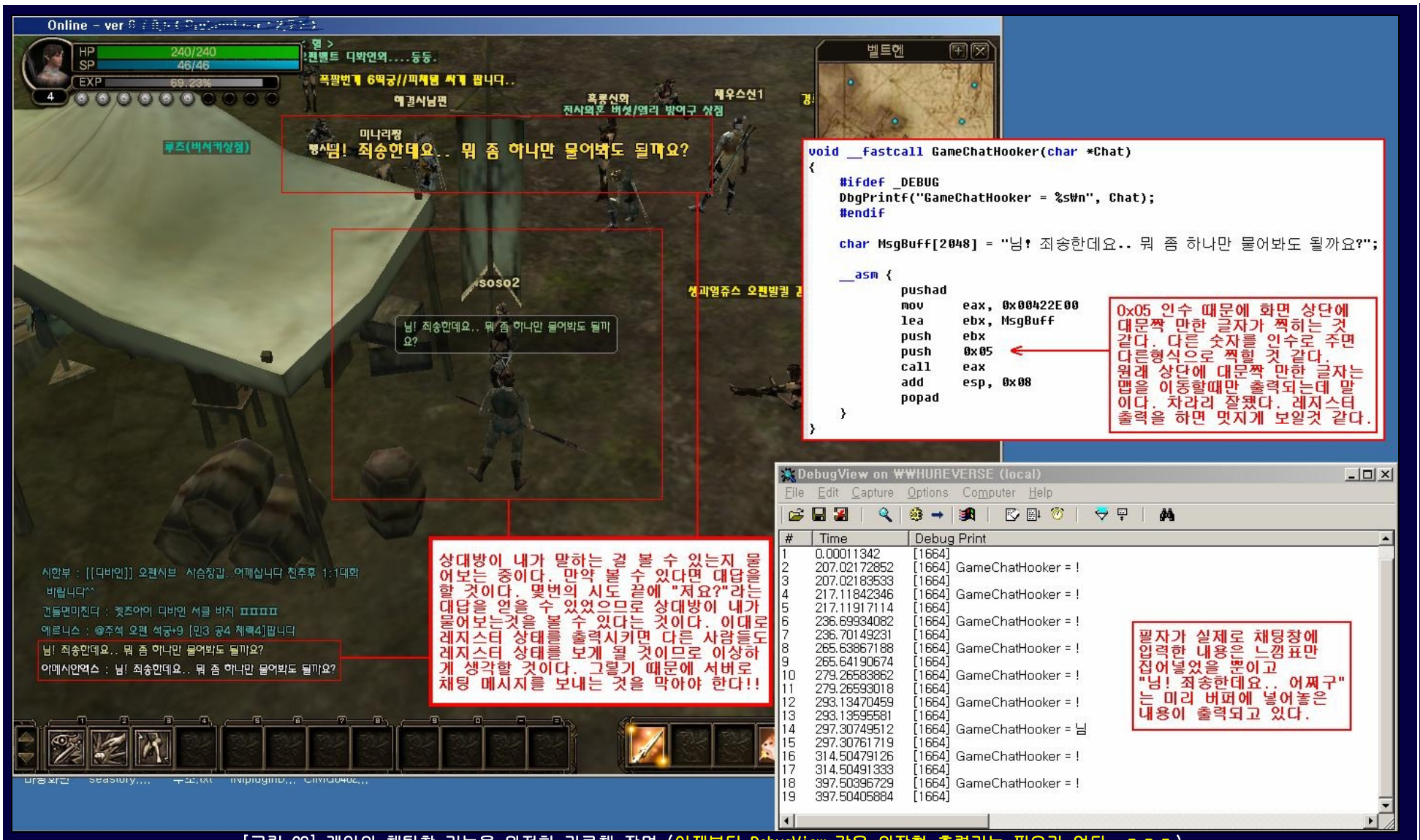
컴파일 완료된 D2HackIt.dll 파일은 Release 라는 디렉토리에 생성되어 있을 것이다. 이 DLL 파일과 환경설정파일인 D2HackIt.ini 를 후킹할 대상 게임의 실행파일이 있는 디렉토리에 복사해준다. 해당 게임에 인젝션 시켜서 실행할 것이다. 아직 우리는 로더를 만들지 않았기 때문에 로더역할을 해줄 놈을 골라야 하는데 필자는 유명한 후킹라이브러리인 Detours 에서 제공하고 있는 withdll.exe 유틸리티를 사용하였다. 이 놈은 소스까지 공개되어 있어서 공부하기에도 좋다. 필자가 시간이 된다면 추가 문서에서 D2Loader 를 분석해 볼 것이다. (참고: Detours 라이브러리의 샘플을 컴파일 시켜야 withdll.exe 를 얻을 수 있을 것이다.)

이제 withdll 유틸로 게임을 실행 시켜보자.



[그림 68] withdll 을 이용한 D2HackIt.dll 인젝션

withdll 은 소스를 자세히 훑어보진 않았지만 CreateProcess 로 실행시키는 구조인걸로 알고 있다. 어쨌든 로더는 다음에 생각하자 이것도 빡시니까..



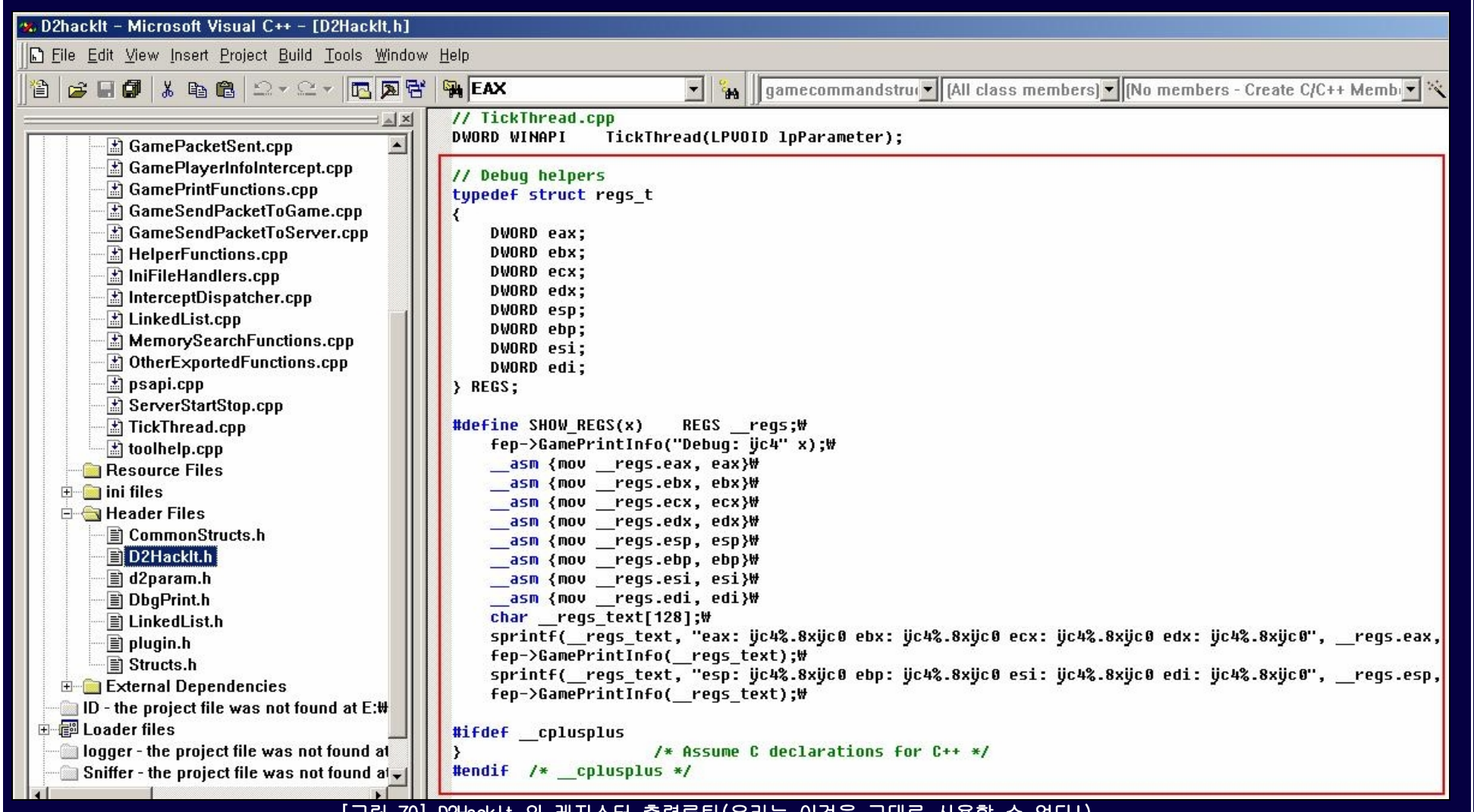
[그림 69] 게임의 채팅창 기능을 완전히 가로챈 장면 (이제부터 DebugView 같은 외장형 출력기는 필요가 없다. ㅎㅎㅎ)

자.. 이제 위의 그림69 에서 보는 것과 같이 완전히 채팅창의 기능을 제어할 수 있으므로 게임에 로그인 한 이상 DebugView 같은 외장형 디버그 출력기에 의존할 필요가 없고 직접 게임을 하면서 레지스터 상태출력, 패킷내용 출력 또는 스택상태를 덤프해서 찍거나 특정 버퍼내용을 출력해서 볼 수 있다. 이 얼마나 아름다운가... D2HackIt 이라는 디아블로2 해킹 키트(kit)가 처음 공개되었을때도 원 제작자는 디아블로2의 채팅창을 가로채서 출력함수로 이용하도록 하는 기본적인 기능만 넣고 배포하였다. 추 후에 다른 개발자들이 참여하면서 채팅창 후킹함수를 API 사용하듯 끌어다 사용해서 추가 플러그인 개발이 이루어졌고 스니퍼, 아이템 자동획득기, 캐릭터 좌표 출력기, 매직 아이템 성공확률 출력기 등등 오만가지가 개발되었다. 이렇게 개발된 후킹함수들이 결국에는 게임 API 와 마찬가지라서 D2HackIt 개발사이트에서는 이를 문서화하여 디아블로2 게임의 개발 SDK 처럼 공개하고있다. 하나의 게임이 코드 인터셉트 후킹을 통해서 완전히 재구성 될 수 있다는 것을 제대로 보여준 케이스다.

어쨌든.. 부가설명은 뒤에서 하기로 하고.. 일단 우리는 여기서 한가지를 해결하고 넘어가야 된다. 그것은 바로 우리가 채팅창에 레지스터 상태같은 디버깅 정보를 뿌릴 때 그 내용이 서버에 전송되지 않도록 해야 한다는 것이다. 그렇기 때문에 채팅창에 '#'문자로 시작되는 입력이 들어오면 화면에 레지스터 상태를 출력하고 버퍼를 바로 청소해서 서버에 빈 문자들이 전송되도록 수를 써야 한다는 것이다. 그래야 주위의 게임유저가 보지 않을 것 아닌가.. (정상적인 유저를 방해 하려는 목적은 아니니까..) 일단은 문제점이 있다는 것을 기억하고 채팅창에 레지스터 상태를 출력해보는 것부터 먼저 차근차근 해보도록 하자..!!

레지스터 상태 출력을 위한 소스수정

자.. 그렇다면 어떻게 수정을 해야지 레지스터 출력을 할 수 있을까?
앞에서 설명했지만 이미 우리는 채팅창에 출력을 할 수 있다. 그렇다면 레지스터 값만 뿌려주면 간단한 거라고 생각할 것이다. 그러나 실제로는 아주 간단하지는 않았다. 그 부분은 진행하면서 설명하기로 하고 일단은 여러분이 이미 생각한대로 레지스터 출력루틴을 만들어보자.



[그림 70] D2HackIt 의 레지스터 출력루틴(우리는 이것을 그대로 사용할 수 없다!)

위에서 빨간박스를 보라.. D2HackIt.h 파일의 내용인데 레지스터를 다루는 부분이 보이는가? 이미 D2HackIt 은 SHOW_REGS 라는 매크로를 만들어서 레지스터 상태를 출력해서 보는 기교를 부리고 있었다. 그러나 앞에서 이미 설명했던 부분을 다시 언급하는데, fep->GamePrintInfo 함수는 디아블로2의 채팅출력창 루틴을 호출한다. 즉, SHOW_REGS 매크로를 우리가 고대로 사용하면 프로그램이 개작살난다.(없는 함수주소를 호출해서 EIP 가 그리로 점프하기 때문에 뻑나는 것이다!!) 그렇다면 우리는 이미 이 게임(현재 대상게임)의 채팅출력창을 호출할 수 있으니까 함수만 바꿔주면 되지 않겠는가? 라고 생각할 것이다. 물론.. 그게 맞다.. 아주 정확하다..
그러나 세상은 그렇게 호락호락하지 않다. 쉽게 생각할 수도록 복병도 많은 법..

실제로 GamePrintInfo 함수는 여러단계의 함수로 구성되어 최종 마지막 함수에서 인라인어셈블리로 디아블로2 채팅출력창을 호출한다. 필자도 그렇게 비스므리하게 코딩을 해서 테스트를 했다. 그랬더니.. 프로그램이 정신 못차리고 뻑나버리는게 아닌가.. 아마도 버퍼가 여러단계의 함수 사이를 거치면서 주소안에 주소가 들어가는 오류를 범한것 같은데 그냥 귀찮아서 매크로부분의 코드를 아예 떼어다가 필자의 후킹루틴에 박아버렸다. 다음이 바로 그 소스이다.

```
/*
    Programming : Reversing by AmesianX in powerhacker.net
*/

//////////////////////////////////////////////////////////////////////////
#define THIS_IS_SERVER
#include "..₩D2HackIt.h"
#include "DbgPrint.h"
//////////////////////////////////////////////////////////////////////////
// GameChatHooker()
//////////////////////////////////////////////////////////////////////////
void __fastcall GameChatHooker(char *Chat)
{
    #ifdef _DEBUG
    DbgPrintf("GameChatHooker = %s₩n", Chat);
    #endif

    char __regs_text[2048];

    if(strcmp(Chat,"#debug") == 0)
    {
        REGS __regs;

        __asm {mov __regs.eax, eax}
        __asm {mov __regs.ebx, ebx}
        __asm {mov __regs.ecx, ecx}
        __asm {mov __regs.edx, edx}
        __asm {mov __regs.esp, esp}
        __asm {mov __regs.ebp, ebp}
        __asm {mov __regs.esi, esi}
        __asm {mov __regs.edi, edi}

        sprintf(__regs_text, "[AmesianX in ROOTClub]₩n₩nDEBUG REGISTER STATUS₩n₩nEAX: 0x%.8x EBX: 0x%.8x ECX: %.8x EDX: 0x%.8x₩n"
                             "ESP: 0x%.8x EBP: 0x%.8x ESI: 0x%.8x EDI: 0x%.8x", __regs.eax, __regs.ebx, __regs.ecx, __regs.edx,
                             __regs.esp, __regs.ebp, __regs.edi, __regs.edi);

        //  char MsgBuff[2048] = "님! 죄송한데요.. 뭐 좀 하나만 물어봐도 될까요?";


        __asm {
            pushad
//          [구 버전]
//          mov     eax, 0x00422E00
//          [신 버전:0x00422F20]
            mov     eax, 0x00422F20
            lea     ebx, __regs_text
            push    ebx
            push    0x05
            call    eax
            add     esp, 0x08
            popad
        }
    }
}
```

[그림 71] GameChatHooker.cpp 파일을 위와 같이 레지스터를 출력하도록 소스수정

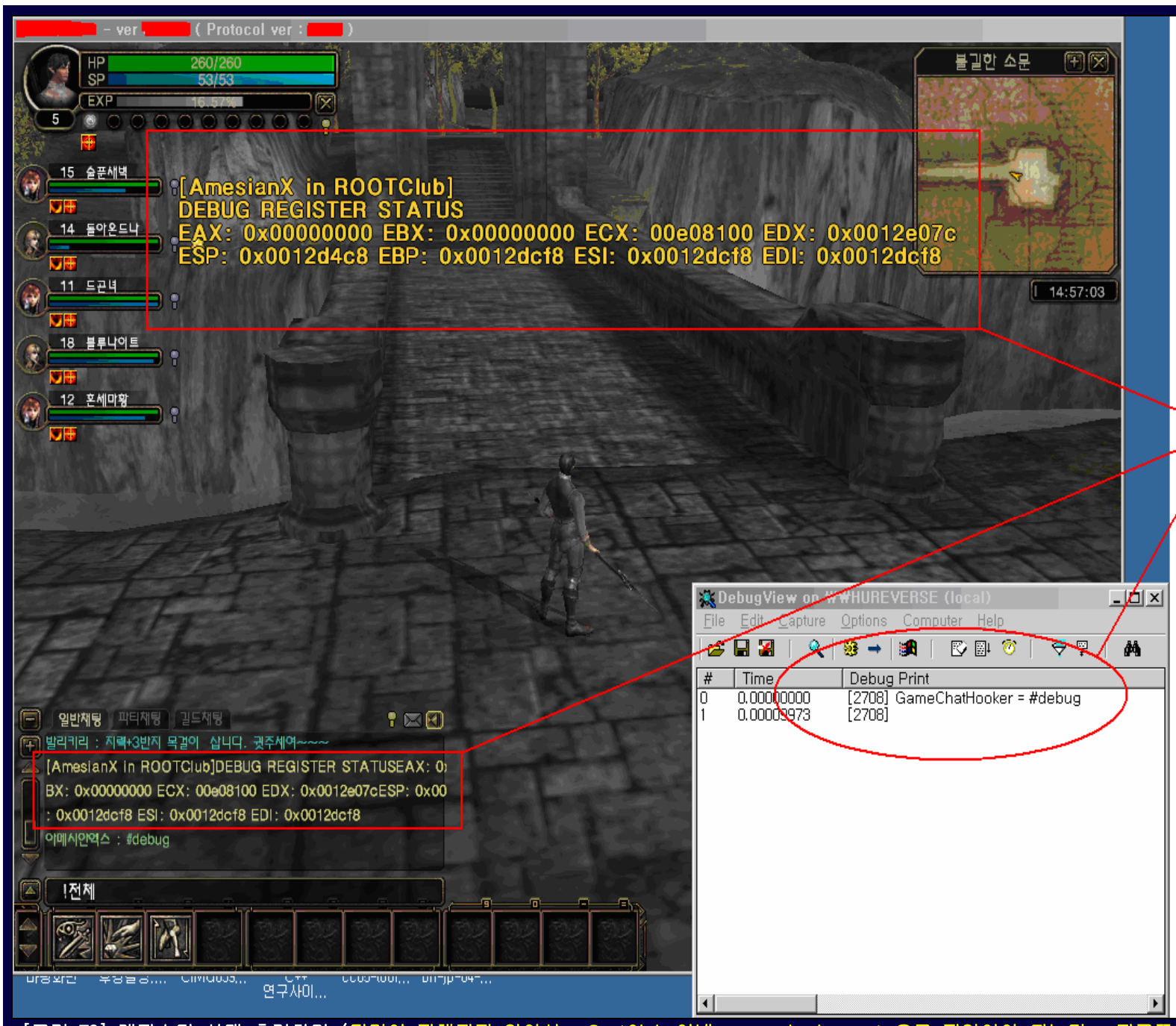
위의 소스를 읽어보면 GameChatHooker 함수만 나와 있는데 GameChatHooker_STUB 함수는 달라진게 없다. (문서작성이 오래걸리자 게임의 업데이트 버전이 나오면서 함수주소가 바뀐것을 제외하면 변한게 없음.) 소스를 위에서부터 대략한번 훑어보자. GameChatHooker 함수가 호출될때는 이미 GameChatHooker_STUB 함수가 후킹을 해서 버퍼를 넘겨준 상태이므로 Chat 변수에는 채팅입력창의 버퍼가 넘어온다.(호출규약:Calling Convention 을 공부하라) 이 버퍼를 DbgPrintf 함수로 찍어서 DebugView 라는 외장 프로그램으로 출력해 디버깅을 대비한다. 그 다음은 Chat 버퍼의 문자열이 "#debug" 가 입력되었는지 strcmp 함수로 스트링 비교를 한다. 만약(if 문을 보시오) strcmp(Chat,"#debug") == 0 이면 Chat 버퍼와 "#debug" 문자열이 일치하는 것이므로 우리의 커맨드가 버퍼에 들어왔다는 소리가 된다. 그래서 이 if 문장의 블럭({} 로 둘러싸인부분)안에서 실행되는 코드는 명령커맨드가 들어왔을 때에만 실행되는 것이다. 그래야만 선택적으로 레지스터를 출력하면서 정상적인 채팅도 할 수 있다. 이렇게 처리하지 않을 경우엔 어떻게 될까.. 위의 그림69 의 경우가 된다. "님! 죄송한데요.. 뭐 좀 하나만 물어봐도 될까요?" 라는 단순한 질문만 주구장창 출력되고 정상적인 채팅을 할 수 없다. if 문 블럭안의 코드를 계속 읽어보면 그림70 의 D2HackIt.h 에서 봤던 REGS 구조체를 하나 선언해주고 __asm {mov __regs.eax, eax} 와 같은 인라인어셈블리 명령을 사용해서 REGS 구조체 각각의 변수에다 현재의 레지스터 값을 담는다. 그리고 sprintf 함수를 사용해서 미리 마련한 __regs_text 버퍼에 채팅창화면으로 출력할 문자열들과 레지스터 상태를 조합해서 담는다. 그러면 __regs_text 버퍼에 레지스터 상태가 담긴 문자열이 들어있다. 이 버퍼를 이미 앞에서 설명했던 방법으로 출력한다. 인라인어셈블리 명령으로 __regs_text 버퍼주소를 ebx 에 로딩하고 게임 속에 존재하고 있는 채팅출력창 함수 0x00422F20(게임 바이너리가 업데이트 되어 신버전으로 주소바뀜) 를 직접 호출한다. 이때 스택정리 add esp, 0x08 은 필수이다.(이유를 모르면 호출규약을 공부하라) 또한, 버퍼의 주소를 mov 를 통해서 넘길 것인가 아니면 lea 명령을 통해서 넘길 것인가를 주의해야 한다. mov 명령을 통한 버퍼지정이 안될 수도 있다. 그럴경우 lea 명령으로 레지스터에 로딩시킨다.

<인라인어셈블리 명령만 설명>

```
__asm {
// EAX, ECX, EDX, EBX, ESI, EDI, EBP, ESP 의 4바이트 짜리 레지스터 총 8 개를 스택에 저장하여 32 바이트(16진수 0x20) 만큼 스택을 증가시키는 명령
      pushad

// [구 버전] 게임이 업데이트 되어 주소가 바뀌었음. 예전 채팅출력창 주소..
// mov    eax, 0x00422E00
// [신 버전] 현재 필자가 테스트 하는 시점의 최신 채팅출력창 주소
      mov    eax, 0x00422F20

// 버퍼의 주소를 ebx 에 로딩함. mov 명령으로 바꿀수도 있으나 오류가 계속 날 경우 lea 명령 사용
      lea    ebx, __regs_text

// 인수 전달을 위해 스택에 ebx(버퍼주소) 저장, 채팅출력창 함수 호출 시 두번째 인수임. (arg_8 = ebp+0x0C)
      push   ebx

// 필자가 출력창부분을 임의적으로 후킹하면서 뽑아낸 상수 값이다. 이 값이 게임 상단에 대문짝 만하게 찍히는 역할을 하는 것으로 추정됨.
// 채팅출력창 함수 호출 시 첫번째 인수임. (arg_4 = ebp+0x08)
      push   0x05

// [보충설명]
// CALL 명령은 두가지가 있다. 일반적인 함수호출(오프셋기법)과 직접호출(다이렉트 호출)이다. 혹시 D2HackIt 소스의 Intercept 함수를 분석해 봤는가?
// 쉘코드를 만들어본 경험이 있는가? 차이를 설명하자면 먼저, CALL 0xXXXXXXXX 형태의 일반적인 호출은 0xXXXXXXXX 주소가 실제함수의 주소가 아니다.
// 어셈블리를 많이 접하지 못해본 사람은 착각할 수 도 있을까봐 설명하는데 0xXXXXXXXX 주소는 현재 CALL 명령으로부터 실제함수가 위치한 절대주소까지
// 거리가 얼마나 얼마나 떨어져 있는가를 나타내는 오프셋 수치이다. 예를들어 CALL 명령이 위치된 주소가 만약 1000 번지일 경우 2000 번지인 실제함수
```

```
// 를 호출하려면 2000 - (1000 + 5) 의 계산으로 나온 CALL 995 가 되는 것이다. 5 는 CALL 명령이 총 5 바이트 길이를 차지하기 때문에 CALL 명령길이
// 만큼을 추가적으로 더 빼야된다. 이 같은 방식은 지역메모리 공간이 아니라 DLL 에 위치한 함수도 호출할 수 있으며 현재 D2HackIt 의 Intercept
// 함수가 이렇게 계산해서 가로채주는 역할을 한다. 우리의 DLL 이 삽입되서 인터셉트함수가 실행되면 타켓프로세스 공간에 CALL 명령이 삽입되는데
// 호출할 주소가 우리의 DLL 속에 있으므로 삽입된 CALL 위치부터 후킹함수까지의 거리를 계산해서 그 오프셋값이 춰해지는 것이다.
// DLL 은 EXE 보다 높은 메모리번지에 로딩되므로 DLL 안의 후킹함수 - (후킹하려고 찾은 루틴위치(타켓 EXE 또는 DLL) + 5) 의 계산공식이 사용된다.
// 이것이 우리가 보게되는 아주 일반적인 함수호출 모습이다.
// 그렇다면 두번째인 다이렉트 호출은 직접적으로 실제함수의 절대주소를 지정해서 호출 할 수 있겠구나라고 생각할 것이다. 예상이 맞았다. 그러나
// 항상 주소를 레지스터에 넣어서 호출해야만 한다. 쉘코드를 만들때도 이 방법을 아주 많이 사용하는데 그 이유는 오프셋을 계산할 필요가 없다는
// 점과 오프셋을 계산하려고 해도 원하는 함수가 어디에 짱박혀있는지 모르는 상태에서는 불가능하기 때문이다. 그래서 해커들은 미리 자신의 시스템
// 에서 리버싱을 수행한 후 프로그래밍할 때 eax 레지스터 같은 곳에 알아낸 함수의 주소를 하드코딩으로 박아서 쉘코드를 만든다. 다른 사람들의
// 시스템이 해커의 시스템과 같은 환경이라면 그 해커가 만든 쉘코드를 사용해도 정상적으로 작동한다. 대부분의 응용 호출기법이 이런식으로 이루어
// 지고 있다. 여기서도 마찬가지로 0x00422F20 이 채팅출력창 함수라는 것을 리버싱을 통해서 이미 알고있었기 때문에 직접적으로 절대주소를 지정해서
// 호출 하는 것이다. (물론, 호출방법이 이것만 있는 것은 아니다. push, pop 의 조합을 통한 ret(리턴) 호출도 있으나 이건 어디까지나 비정상적인
// 방법이므로 추후에 발표할 "Advanced Windows Shellcode in Unicode" 문서에서 다룰 예정이다.)

// 다음의 호출을 통해서 채팅출력창 함수가 강제 호출된다.
      call    eax

// 다음은 호출규약을 공부하면 알 수 있는 명령이다. Calling Convention 이라고 검색하면 많은 정보를 볼 수 있을 것이다. 호출규약 공부시는 C++
// 호출규약도 필히 익혀둔다. 오늘날의 프로그램들이 VC++ 로 개발된 것이 태반이고 기본적으로 C++ 계열로 개발되기 때문에 필수사항이라고 할 수
// 있다. 일단 간단한 설명을 하면 위에서 두개의 인수를 스택에 저장했으므로 함수호출시 함수 안에서 스택정리가 일어나지 않는다면 직접 제거해줘야
// 한다. 그렇지 않을경우 ESP(스택포인터) 가 꼬여버려서 GameChatHooker 함수가 끝나고 리턴될때 엉뚱한 주소로 점프하면서 개작살 나게된다.
// (최초 작업시 함수안에서 스택정리가 되는 줄 알고 이 한 줄 때문에 하루가 가까운 시간을 허비했었다. D2HackIt 에 참고가 없는 부분임. 또한,
//  D2HackIt 에는 필자가 구현하고 있는 스택복구도 기능도 없다. 그래서 D2HackIt 은 스택복구를 안하려고 mov 와 같은 스택과 관련없는 부분만
//  찾아서 후킹을 걸고 있다. )
      add     esp, 0x08

// 다음 명령은 위에서 스택에 저장했던 8 개의 레지스터들을 원래값으로 복구한다.
// 바로 위의 add    esp, 0x08 을 해주지 않으면 실제로 popad 에서 push/pop 명령짝이 안맞기 때문에 스택에서 값을 꺼내면 엉뚱한 값이 뽑아지므로
// 스택이 꼬여버린다. 그러므로 리턴할 때 작살나는 것이다.
      popad
```

[그림 72] 채팅입력창에 레지스터 출력 커맨드 입력



레지스터 상태 출력에 성공한 모습

캐릭터의 머리통에 있었던 채팅내용이 없어진 것이 보이는가?

서버로 레지스터 상태 출력이 전송되지 않을 것이다.

[그림 73] 레지스터 상태 출력화면 (팀명이 정해지지 않아서.. RootClub 이넹.. powerhacker.net 으로 찍었어야 되는디.. 지금쯤 패치됐을라나.. 쩝 ㅡ.ㅡ;)

자.. 위의 화면을 보라. "##debug" 를 입력하니까 레지스터 상태가 출력되는 것이 보이는가? ㅎㅎㅎ 아주 재밌다. 그런데 여기서 문제가 한가지 있다는 것을 발견했다. "#" 이 파티원과 대화를 할때 사용하는 커맨드 식별자라는 것을 나중에 알게 되었다. 그래서 "#" 을 2개 입력해야지 1개가 출력된다. 위에서 DebugView 에 나타난 문자열을 보면 "#debug" 만 보일 것이다. 그냥 "##debug 를 입력하면 뭐 어때.. 출력만 되면 장땡이지.." 라고 생각할 무렵 한가지 더 문제가 생겼는데 파티를 맺지 않으면 ##debug 라고 쳐도 "대화가 가능한 파티원이 없습니다" 라는 메시지가 나오면서 레지스터 출력이 안되는게 아닌가... -_-; 큭.. 덴장..
그래서 몇번의 테스트를 거친 후 "-" 문자가 원래 게임의 채팅커맨드에 포함되어 있지 않는 것을 확인하고 -debug 로 명령을 바꿨다. 그리고 마지막 한가지 문제를 해결해야 했는데 그 문제가 하이라이트라고 할 수 있다. 그것은 바로 서버에 내가 친 명령이 전송되지 않도록 하는 것이다. 다음이 수정된 소스이다.

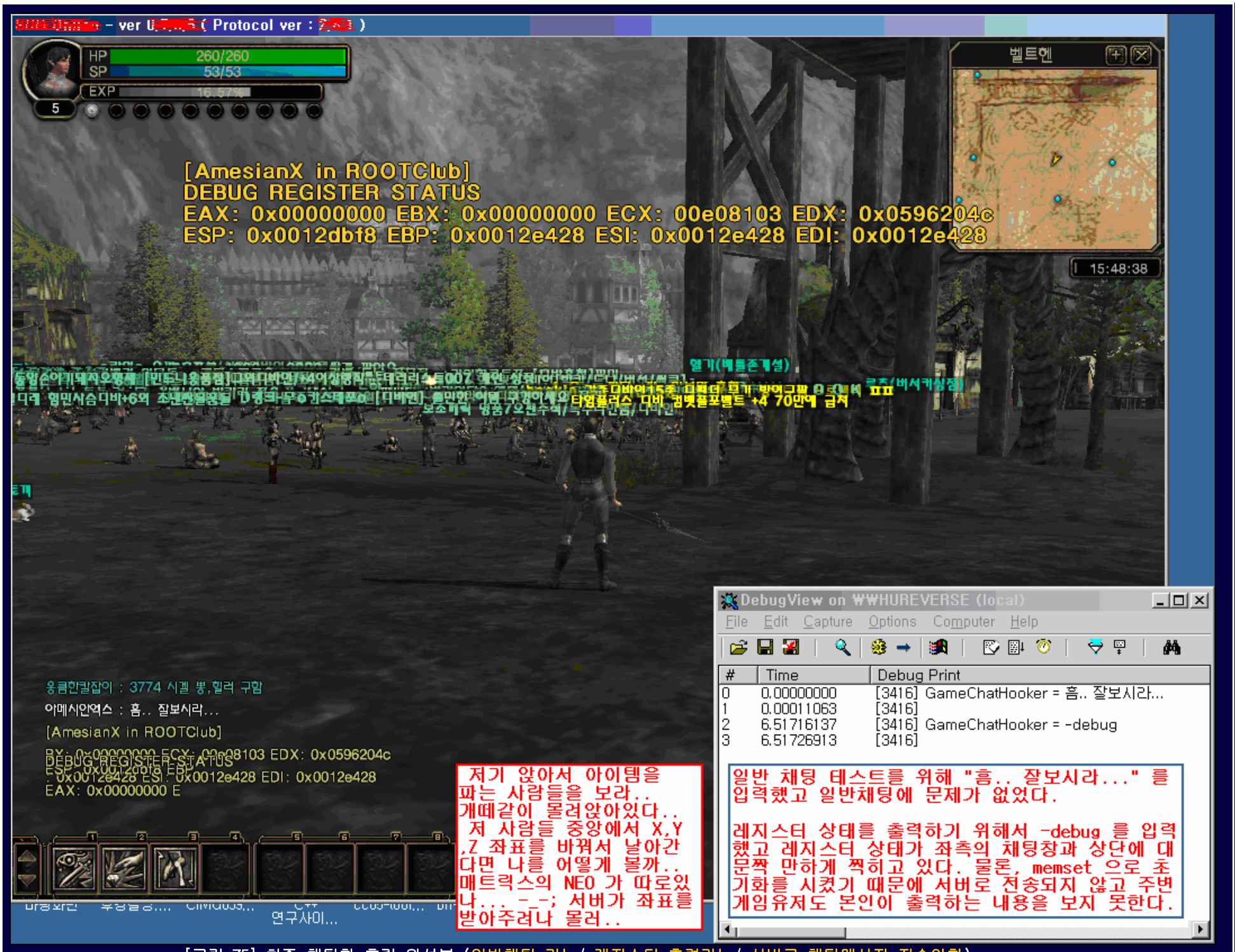
```
/*
    Programming : Reversing by AmesianX in powerhacker.net
*/

////////////////////////////////////////////////////////////////////////////
#define THIS_IS_SERVER
#include "..₩D2HackIt.h"
#include "DbgPrint.h"
////////////////////////////////////////////////////////////////////////////
// GameChatHooker()
////////////////////////////////////////////////////////////////////////////
void __fastcall GameChatHooker(char *Chat)
{
    #ifdef _DEBUG
    DbgPrintf("GameChatHooker = %s₩n", Chat);
    #endif

    char __regs_text[2048];

    if(strcmp(Chat,"-debug") == 0)
    {
        REGS __regs;

        __asm {mov __regs.eax, eax}
        __asm {mov __regs.ebx, ebx}
        __asm {mov __regs.ecx, ecx}
        __asm {mov __regs.edx, edx}
        __asm {mov __regs.esp, esp}
        __asm {mov __regs.ebp, ebp}
        __asm {mov __regs.esi, esi}
        __asm {mov __regs.edi, edi}

        sprintf(__regs_text, "[AmesianX in ROOTClub]₩n₩nDEBUG REGISTER STATUS₩n₩nEAX: 0x%.8x EBX: 0x%.8x ECX: %.8x EDX: 0x%.8x₩n"
                             "ESP: 0x%.8x EBP: 0x%.8x ESI: 0x%.8x EDI: 0x%.8x", __regs.eax, __regs.ebx, __regs.ecx, __regs.edx,
                             __regs.esp, __regs.ebp, __regs.edi, __regs.edi);

        //  char MsgBuff[2048] = "님! 죄송한데요.. 뭐 좀 하나만 물어봐도 될까요?";

        __asm {
            pushad
//          [구 버전]
//          mov     eax, 0x00422E00
//          [신 버전:0x00422F20]
            mov     eax, 0x00422F20
            lea     ebx, __regs_text
            push    ebx
            push    0x05
            call    eax
            add     esp, 0x08
            popad
        }

        Chat[0x00] = 0x00;
        memset(__regs_text, 0, sizeof(__regs_text));
    }
}
```

Chat[0x00] 에 0x00(문자열의 종료를 알리는 널문자) 값을 넣어줌으로써 채팅입력창 버퍼를 클리어시킴.
__regs_text 버퍼를 깨끗이 비워서 화면출력후 다른곳에 사용될지 모르므로 청소함.
이렇게 하면 서버로 아무런 전송도 가지 않는다.
그러므로 화면에 출력 메시지만 보게 될 것이다.

[그림 74] 완료된 레지스터 출력소스(함수의 구조는 여기만 바뀌었음)

이전과 비교해서 이 함수만 바뀌었을 뿐이고 나머지 함수는 이전과 같다. 다만 주소는 틀릴 것이다. 그것은 게임 바이너리가 업데이트 되기 때문이다. 그러나 찾으려고 한다면 못들 못할까..

위 그림74 가 그림71 과 달라진 부분을 비교해보면 #debug 가 -debug 로 바뀌었고 마지막에 Chat[0x00] = 0x00 이 추가된 점, 그리고 __regs_text 버퍼를 초기화 하는 것이다. 설명은 그림에 있다. 이렇게 하면 다음과 같이 레지스터 상태 출력만 나오고 서버로 전송되는 채팅메시지가 전혀 출력되지 않을 것이다. 후킹함수에서 버퍼를 출력하고 바로 지워버렸기 때문에 서버로 전송되지 않는 것이다. 원래는 채팅입력창에 입력한 레지스터 커맨드인 -debug 와 채팅출력창에 출력되는 레지스터 상태 문자열이 서버로 전송되야 정상인데 이를 버퍼청소로써 막아버린 것이다.



[그림 75] 최종 채팅창 후킹 완성본 (일반채팅 가능/ 레지스터 출력가능/ 서버로 채팅메시지 전송안함)

위의 화면을 보면 일반채팅이 정상적으로 잘 작동되는 것을 보이기 위해서 "흠.. 잘보시라..." 를 입력했다. 그리고 -debug 를 입력해서 레지스터 상태를 출력되도록 하였다. 일부러 문서를 작성하기 위해서 후킹을 했지만 참 멋지게 작동되는 것 같다. 필자는 X, Y, Z 좌표도 알고있다. 만약 저 게임속 개떼 같이 앉아 물건을 파는 유저들 사이에서 공중으로 솟구쳐오를 수 있는 커맨드를 만들어서 테스트하면 어떤 반응이 나올까... 상상만해도 아주 재밌는 반응들이 나올 것 같다. 그러나 그렇게 하지는 않을 것이다. 왜냐면 게임유저들이 재밌게 즐기는데 방해를 하고 싶지않다. 그리고 필자의 본래 의도가 범용후킹을 설명하기 위한 것이므로 이 정도에서 그만 둘 것이다. (날라다니는 걸 보여주면 험한 꼴 보게 될 것 같아서 두렵다.. -_-;)

[보너스] *포맷스트링 버그 존재 (어차피 이 게임은 유저용이므로 쉘의 의미는 없다. 그냥 버그라고 할 수 있음)*

[그림 76] 게임 클라이언트에 포맷스트링 버그가 발견됨

채팅창후킹을 하다가 우연히 발견했다. 위의 화면은 아무런 후킹도 걸지 않은 원래 게임자체 화면이다. 채팅창에 %x 를 입력하면 메모리 주소를 출력하는 전형적인 포맷스트링 취약점과 동일한 모습이다. 어차피 큰 의미는 없는데 유저용 게임이므로 쉘코드를 딸 필요가 없으므로 그냥 게임시 버그정도의 의미가 전부다. 그렇지만 이 버그를 상대방 채팅창으로 쏘아 보낼 수 있다면 꽤 흥미롭지 않은가? 원래 이 게임은 채팅창에 % 를 입력할 수 없다. % 는 길드를 맺은 유저와 대화를 할때 지정하는 식별자이기 때문에 채팅창에

입력해도 이 문자가 상대방한테 전달되지 않는다. 그러나 우리는 앞에서 설명한 후킹을 좀 더 연구하면 상대방 채팅창에 % 문자도 보낼 수 있다. 그렇다면 %x 문자를 받은 유저는 이 버그 때문에 최소한 클라이언트가 터져서 밖으로 튕겨나가게 될 것이고, 쉘코드를 주입해서 공격한다면 상대방의 시스템으로 침투도 할 수 있을 것이다.. ㅎㅎㅎ

실전 Lesson-2 : 인터넷 익스플로어 후킹으로 암호화 훔쳐보기

Lesson-2 에서는 우리가 인터넷 사이트를 들릴때마다 설치되는 암호화 모듈들의 통신내용을 후킹으로 가로채는 것을 보여준다. 필자는 D2HackIt 을 범용으로 사용하는 예를 Lesson-1 만으로는 보여주기에는 부족하다고 판단되어 인터넷에서 다운로드 받아서 작동하는 ActiveX 프로그램들을 대상으로 주제를 선정하였다. 또한 개인적으로 친한동생이 실제 모의해킹 업무에서 공격테스트를 진행한다고 하기에 도움이 되고자 이 챕터를 만들게 되었다. 그 친구가 디버거로 해당루틴을 잡는 방법을 질문했을 때 디버거로 작업하기엔 무리라고 생각되어 후킹을 사용하면 공격하기 쉬울거라고 말해줬는데 실제로 많은 분량을 말로 다 설명해 줄 수 없었다. 필자는 말만 앞서는 사람보다 한줄의 코드로 인정받기를 원한다.

모의해킹 히스토리...

필자가 현업에서 빡시게 일할 때의 경험상 보안업계의 주 고객은 돈을 돌리는(?) 사이트다. 이런 부류의 사이트는 정말 별 쓰잘데기 없는 것에도 상당히 민감하게 반응하기 때문에 x용권이란 단어도 필히 삼가하고 싶다. 이 돈돌리는 사이트는 상당히 빡시게 작업을 해도 취약점이 별반 나오는게 없을 때가 생기는데 그 이유가 이 암호화와 인증이라는 것 때문이다. 뭘 좀 해볼려고 하면 암호화 되있고 또 좀 해볼려고 하면 인증서를 넣으란다. 필자처럼 인터넷 뱅킹 사용법도 모르는 사람은 여간 짜증나는게 아니다. 간혹가다가 또 어떤 사이트는 뚫어보라고 약을 올리기까지 한다. 필자도 겪었지만 실제로 필자가 아는 친구가 더 심한 꼴을 겪었다. 의뢰한 곳에서 뚫으면 인센티브를 주겠지만 못 뚫으면 되려 깎는다고 했었다고 한다. 인센티브야 어차피 받아도 그 친구가 받는게 아니니까 상관없다지만 되려 깎인다면 그 친구의 입장이 모가될까.. 결국엔 뚫었지만 그 친구에게 직접주라고했던 인센티브는 고스란히 회사가 먹어치웠다고 하는 전설이.. 그 친구가 받은건 인센티브가 아니라 스트레스만 덤으로 받았다고 한다. 재주는 해커가 부리고 돈은 해커의 주인이 먹는다는 개념이 판을친다... 썩을..

필자는 돈돌리는 사이트나 나라사이트를 업무로 맡는걸 굉장히 싫어했었는데 정말 재미도 없고 건드릴 것도 찾기 힘든데다 심지어는 로그인할 인증서도 없었다. 최소한 인증서는 회사가 제공해야 하는게 예의 아닌가.. 본인 인증서를 갖고 공격을 하라니 말도안되는 경우를 봤나.. 뚫어야 한다는 중압감 때문에 스트레스로 줄담배를 필때도 많았다. 재미가 있어야 해킹을 할 것 아닌가.. 해킹을 공부한 이유가 재미있어서인데 이건 아주 졸려죽을 판국이었다.. 그래서 몸담고있는 회사에 해킹방법을 개발해야 한다고 주장하니까 프로젝트 투입할 인력도 모자른 판국에 그렇게 리스크(risk)가 큰 곳에 인력을 낭비하는 것은 높은 사람의 결정이 있어야 한다나 뭐라나.. 그래도 필자는 업무에 써먹을 괜찮은 웹스캐너라도 하나 회사에 만들어주고 나오고 싶었다. 왜냐면 회사가 모의해킹에 사용되면 좋을만한 스캐너를 구입하지 않았기 때문이다. 필자가 아는 분이 만든 성능좋고 국내실정에 잘맞는 웹스캐너가 있었는데 그런걸 구매하지 않는다니.. 그래서 업무하면서 필요한 부분을 보충해줄 스캐너를 직접 구상하고 있었는데 이건 뭐 코딩도 안들어갔는데 오늘은 이것 좀 보라고 끊고가고 저것좀 보라고 끊고.. 물론 매번은 아니었지만 의지가 있을때 한자리에 앉아서 계속 몰두할 수 있는 연속성이 보장되지 않는한 개발의욕의 상실로 이어졌다. 이제는 다시 재미삼아 할 수 있는 여건이 되었기 때문에 이전에 이미 만들었어야 했던 웹스캐너나 웹프락시 같은 업무에서 꼭 필요했었던 툴들을 하나씩 만들어서 여러사람이 이용할 수 있도록 해보려고 한다. 이 땅의 하드스터디 해커들이 업무에서 편하게 일할 수 있도록... 먹고살기 힘든 판국에 당면해 있어도 이 문서는 그런 필자의 의지와 노력이 반영되어 약 한달간의 테스트 기간을 거쳐서 만들어졌다. 이 문서가 필자의 의지를 나타내는 시초가 될 것이다...

무엇을 후킹할려고 한 것인가?

무엇을 후킹할려고 하는가? 후킹해야 할 대상은 다음의 자바스크립트를 보고 정할 수 있다.

```
function EncForm(form)
{
	var INIdata = "";
	var eletemp = "";
	var filetemp = "";

	obj = ModuleInstallCheck();
	if (obj == null) {
		alert("암호화프레임(secureframe)을 찾을수 없습니다.");
		return false;
	}

	filetemp = GatherFileValue(form, 0, true);
	if (filetemp !=  "")
	{
		if ((form,filedata,value = obj,MakeFileData(0, cipher, filetemp)) == "") return false;
	}
	eletemp = GatherValue(form, 0, true);
	if ((INIdata = obj,MakeINIpluginData(0, cipher, eletemp, ""))=="") return false;


	if (typeof form,INIpluginData == "undefined")
	{
		if (		_plugin)
		{
			form,input,value = INIdata;
			form,input,name = "INIpluginData";
		} else {
			alert("INIpluginData(form,name)가 필요합니다.");
			return false;
		}
	} else {
		form,INIpluginData,value = INIdata;
	}
```

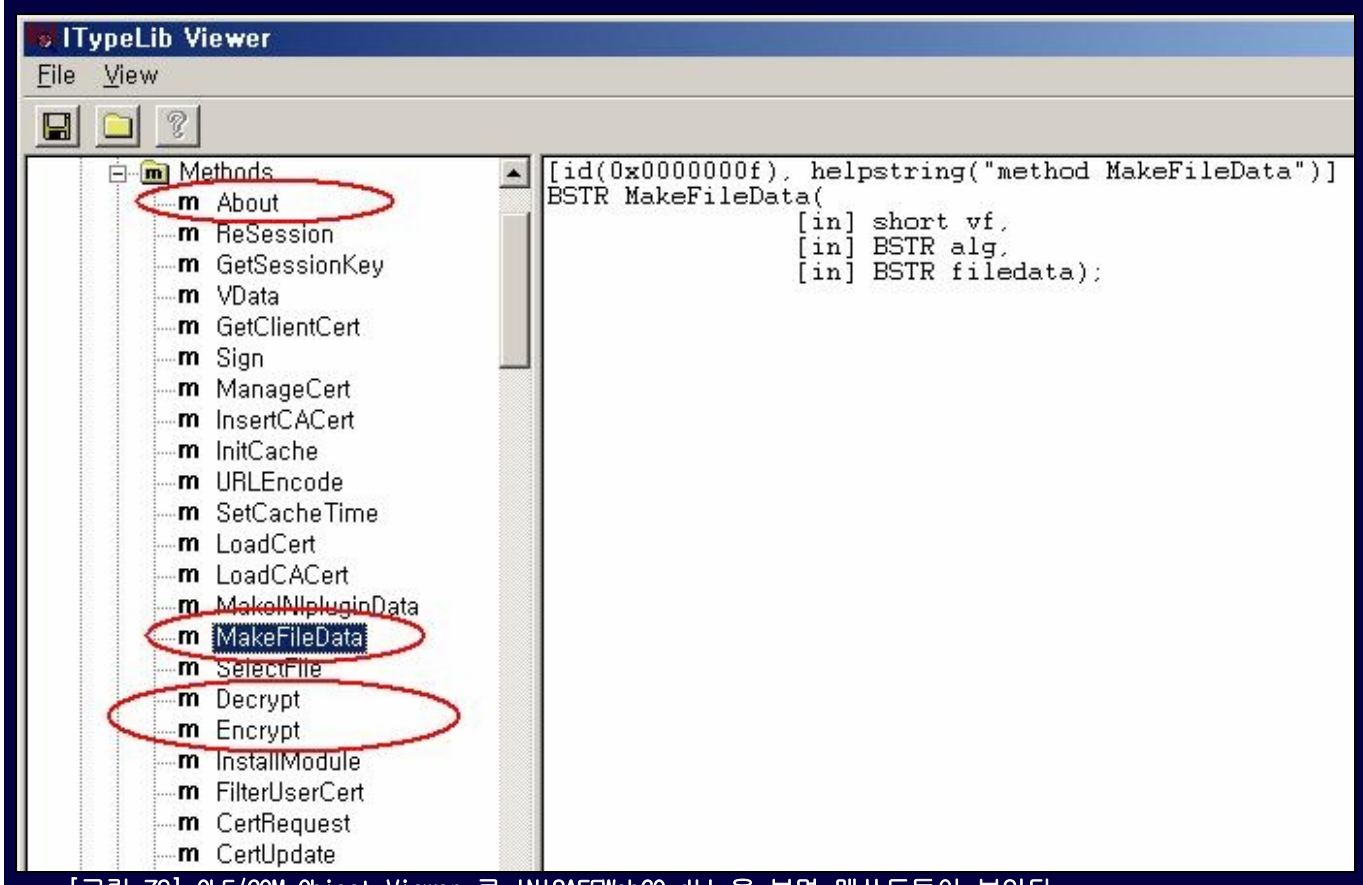
[그림 77] 폼 데이타 암호화(자바스크립트)

우리가 인터넷의 특정 사이트에서 돈에 관한 민감한 정보들을 처리할 때 INISAFE 라는 인증서 모듈이 실행되면서 웹에 관련된 정보들을 암호화 한다.

```
function Idecrypt(data)
{
	obj = ModuleInstallCheck();
	if (obj == null) return "";

	if (navigator,appName == 'Netscape')
		return unescape(obj,Decrypt(cipher, data));
	else
		return obj,Decrypt(cipher, data);
}
```

[그림 78] 복호화 루틴(자바스크립트)

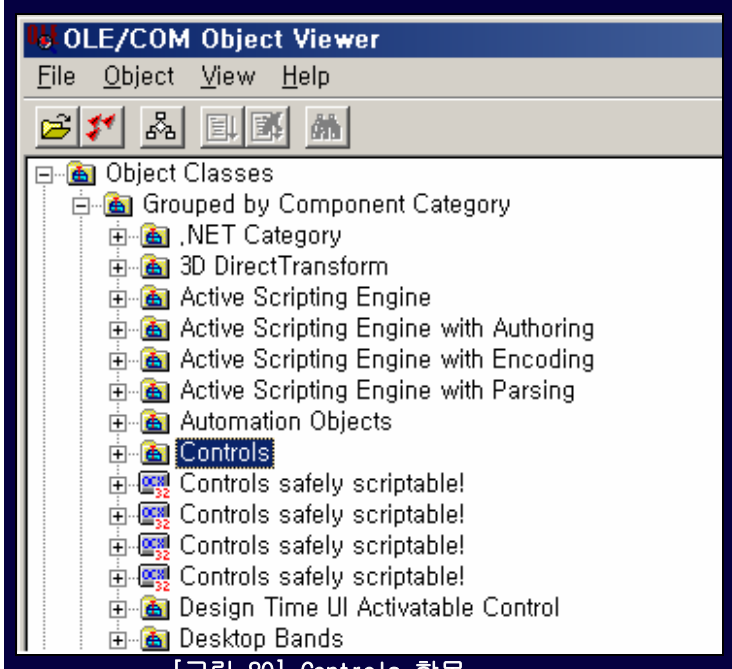
위의 자바스크립트가 실행되면서 서버로부터 받은 데이타들이 복호화 될 것이다. 위의 그림 77 과 78 은 각각 암/복호화가 자바스크립트로 구현된 것이며 이것은 인터넷에서 아무나 받을 수 있는 .js 의 자바스크립트 파일에 존재한다. 그림77 의 EncForm 자바스크립트 함수의 구현부분을 보면 obj.MakeFileData 와 같은 식으로 함수를 호출해서 데이타를 암호화 시킨다. 그림78 을 보게되면 반대로 obj.Decrypt 복호화 함수를 호출하고 있다. 위에서 MakeFileData 을 호출하는 것과 다를바가 없다. 그렇다면 이 함수들은 실제로 존재하는 것일까? 그렇다면 어디에 존재하는 것일까? 우리가 인터넷으로 사이트를 돌아다니다 보면 원하지 않아도 설치해야되는 인증서 모듈인 INISAFE 인증서의 COM 모듈(DLL 파일)에 있다. 이 함수들은 실제 COM 메쏘드로 존재하는 함수들이다.



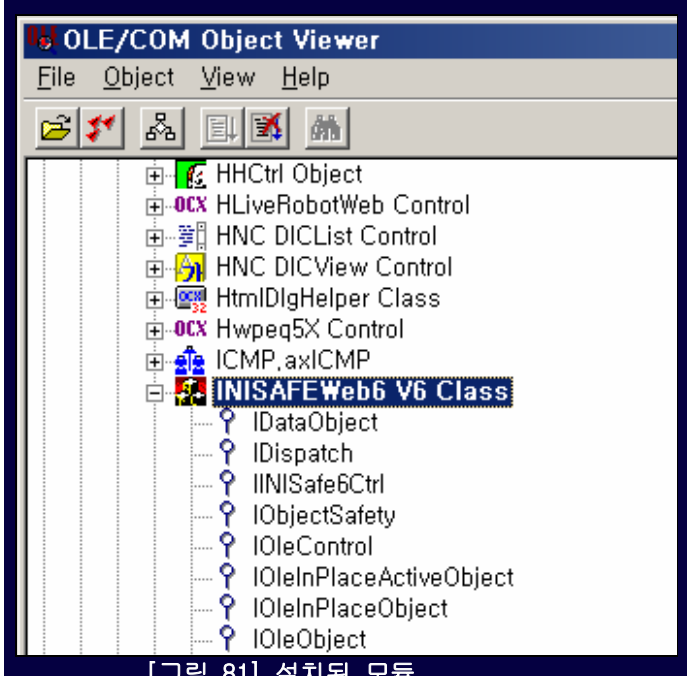
[그림 79] OLE/COM Object Viewer 로 INISAFEWeb60.dll 을 보면 메쏘드들이 보인다

이미 많은 사람들이 알고 있는 내용이지만 과정을 위해 재차 언급하겠다. OLE/COM 뷰어는 VC++ 을 설치하면 자동으로 설치되는 툴이며 자신의 PC 에 설치되어 있는 많은 종류의 COM 모듈(ActiveX 도 COM 임) 의 정보를 자세히 볼 수 있다. 위 그림79 에서는 INISAFEWeb 이라는 인증서 모듈이 필자의 PC 에 자동으로 COM 모듈을 등록해 버리기 때문에 나타나는 정보이다.(INISAFE 가 등록한 것이지 필자가 일부러 등록한 것이 아니다!!) 함수들의 목록이 나타나는데 필자는 여기서 보이는 함수들 중 About 이라는 함수를 일단 호출해봤다.

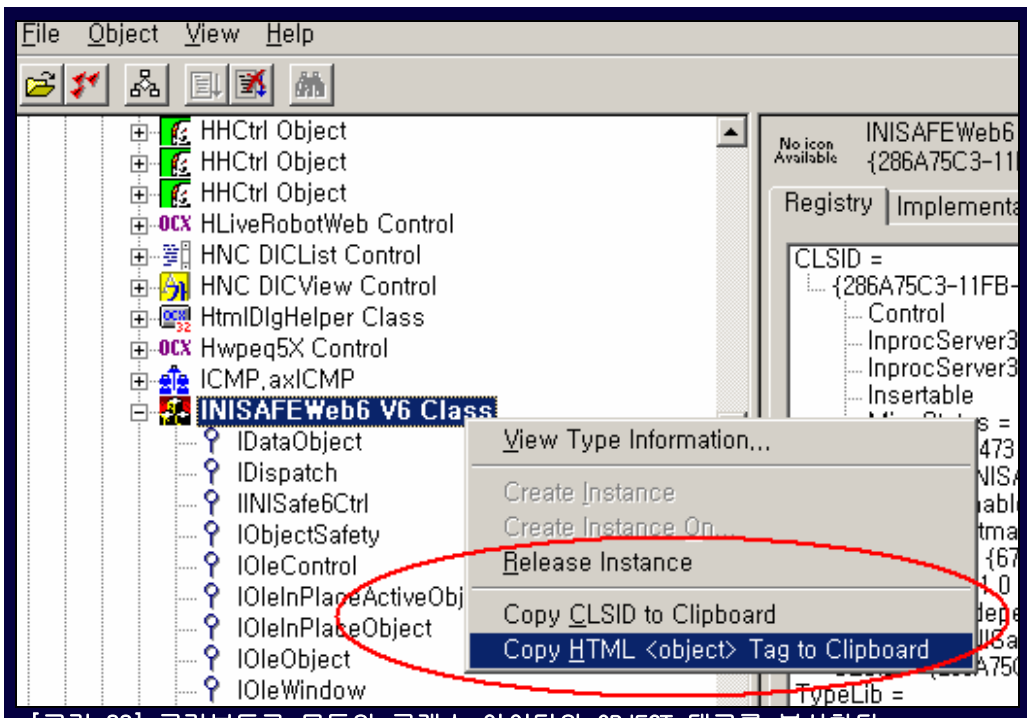
[COM 모듈의 함수 호출해 보기]



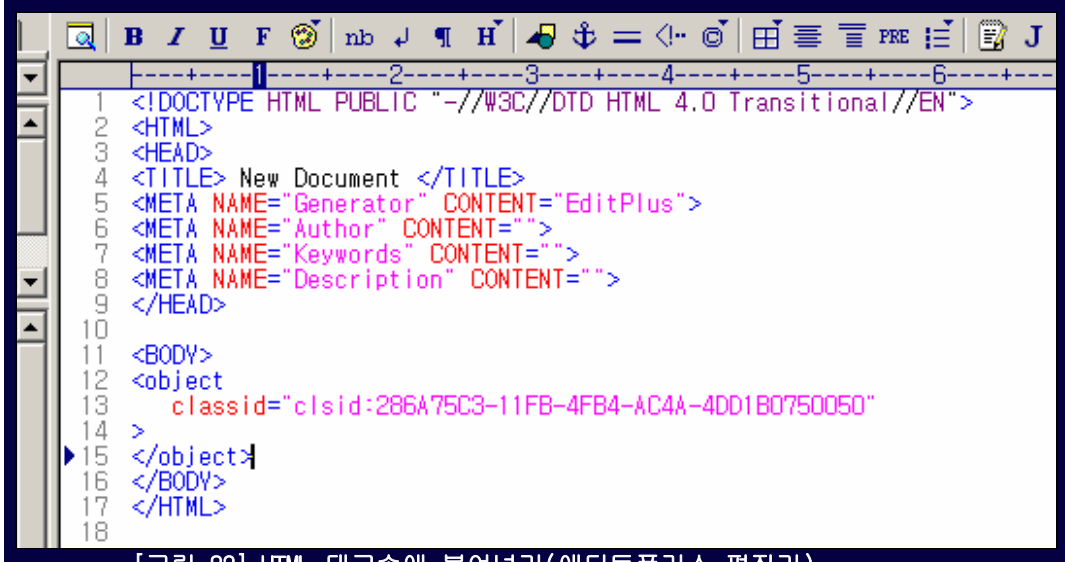
[그림 80] Controls 항목



[그림 81] 설치된 모듈

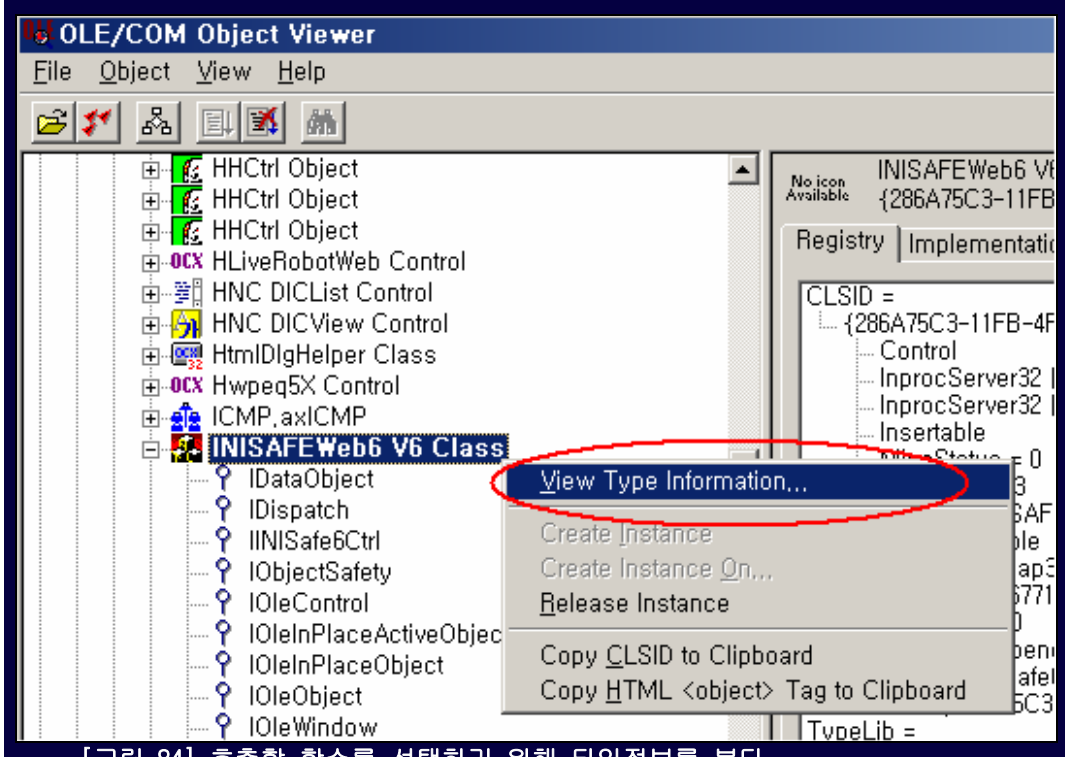


[그림 82] 클립보드로 모듈의 클래스 아이디와 OBJECT 태그를 복사한다

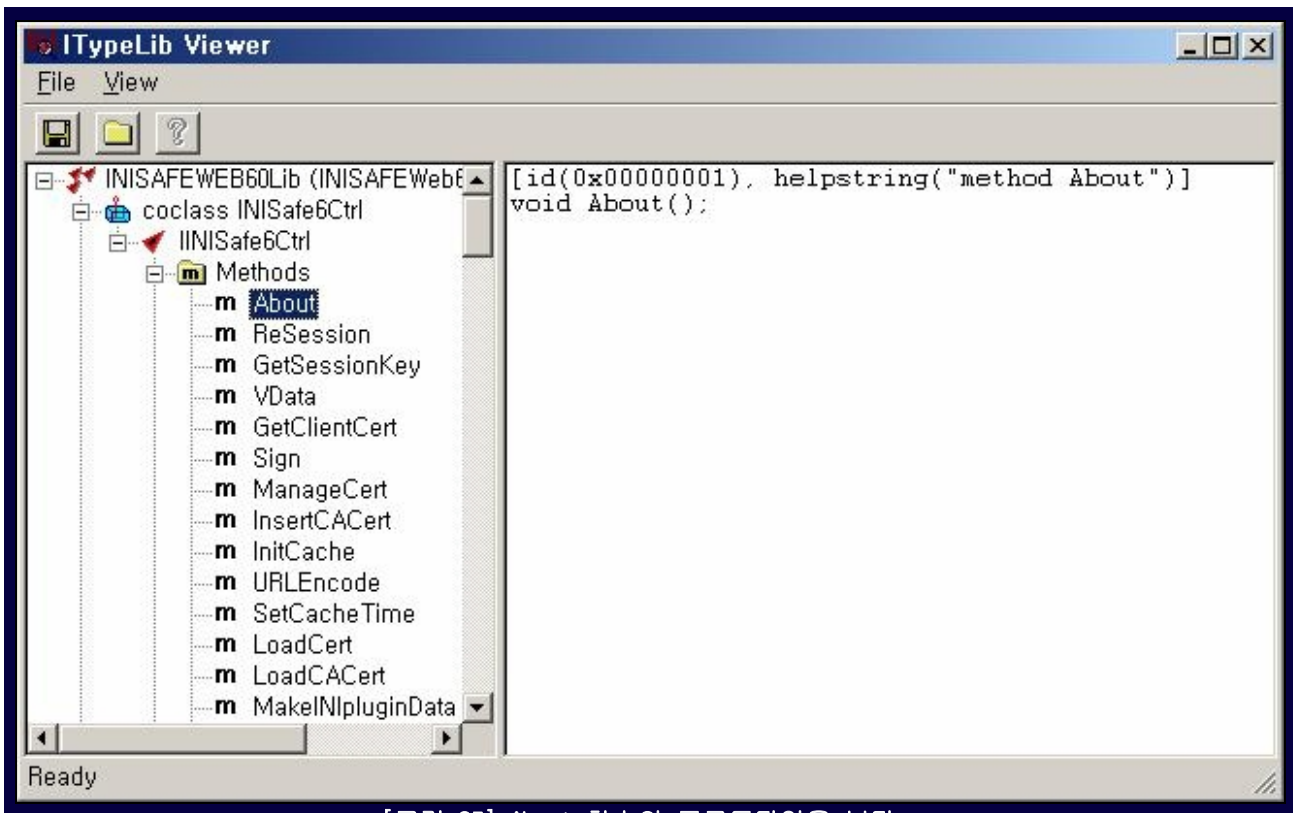


[그림 83] HTML 태그속에 붙여넣기(에디트플러스 편집기)

위와 같이 복사된 클래스 아이디와 OBJECT 태그를 에디트플러스 편집기로 기본 생성한 HTML 태그의 BODY 속에 붙여넣는다.



[그림 84] 호출할 함수를 선택하기 위해 타입정보를 본다

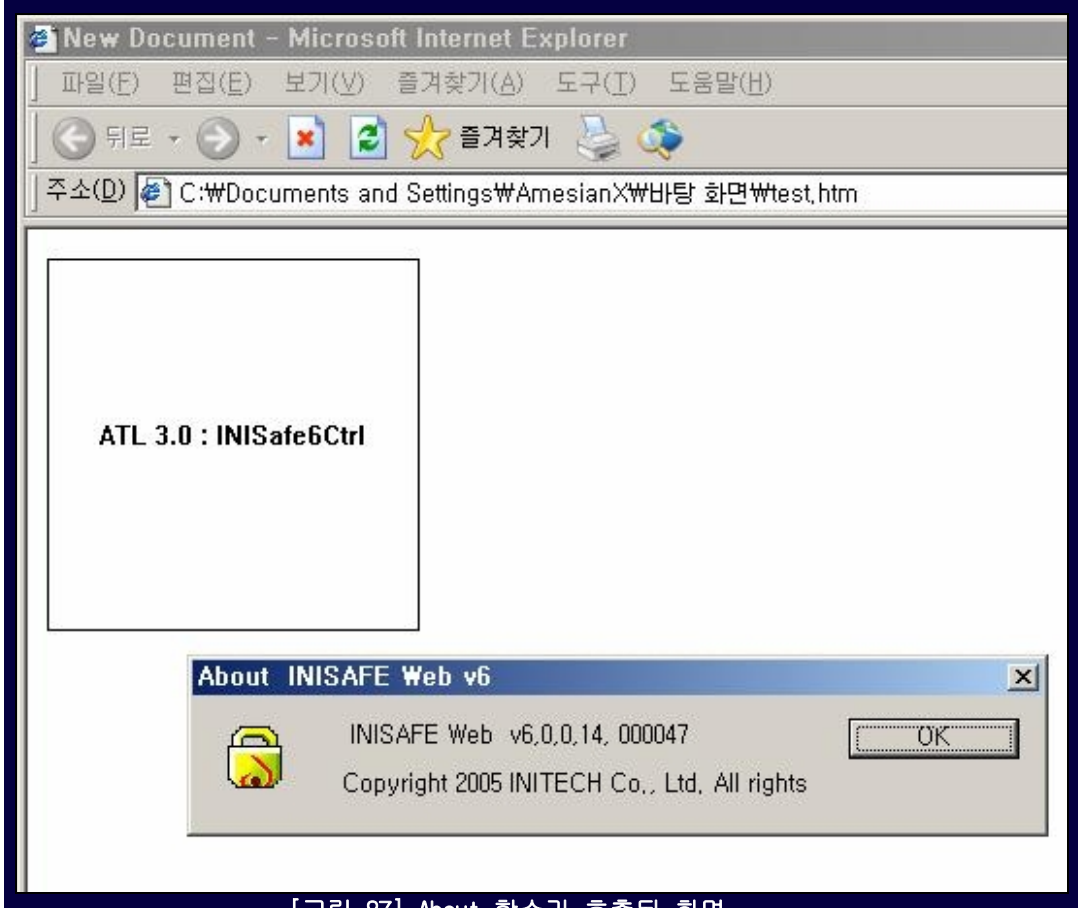


[그림 85] About 함수의 프로토타입을 본다.

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2  <HTML>
3  <HEAD>
4  <TITLE> New Document </TITLE>
5  <META NAME="Generator" CONTENT="EditPlus">
6  <META NAME="Author" CONTENT="">
7  <META NAME="Keywords" CONTENT="">
8  <META NAME="Description" CONTENT="">
9  </HEAD>
10
11 <BODY>
12 <object name="test" classid="clsid:286A75C3-11FB-4FB4-AC4A-4DD1B0750050"></object>
13
14 <script>
15 document.all.test.About();
16 </script>
17
18 </BODY>
19 </HTML>
20
```

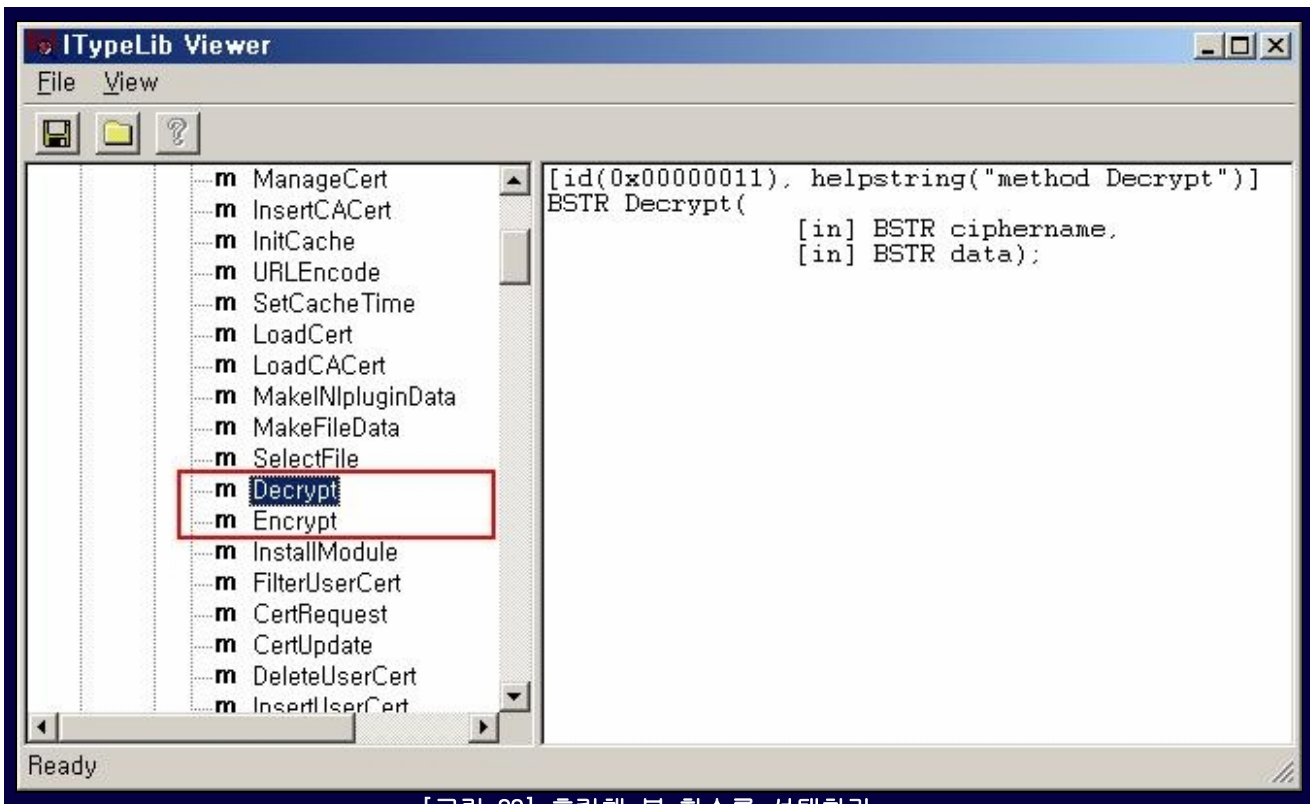
[그림 86] About 함수 호출코드를 자바스크립트로 작성

위와 같이 객체의 이름을 test 라고 주고 자바스크립트로 test 객체의 About 함수를 호출해본다.



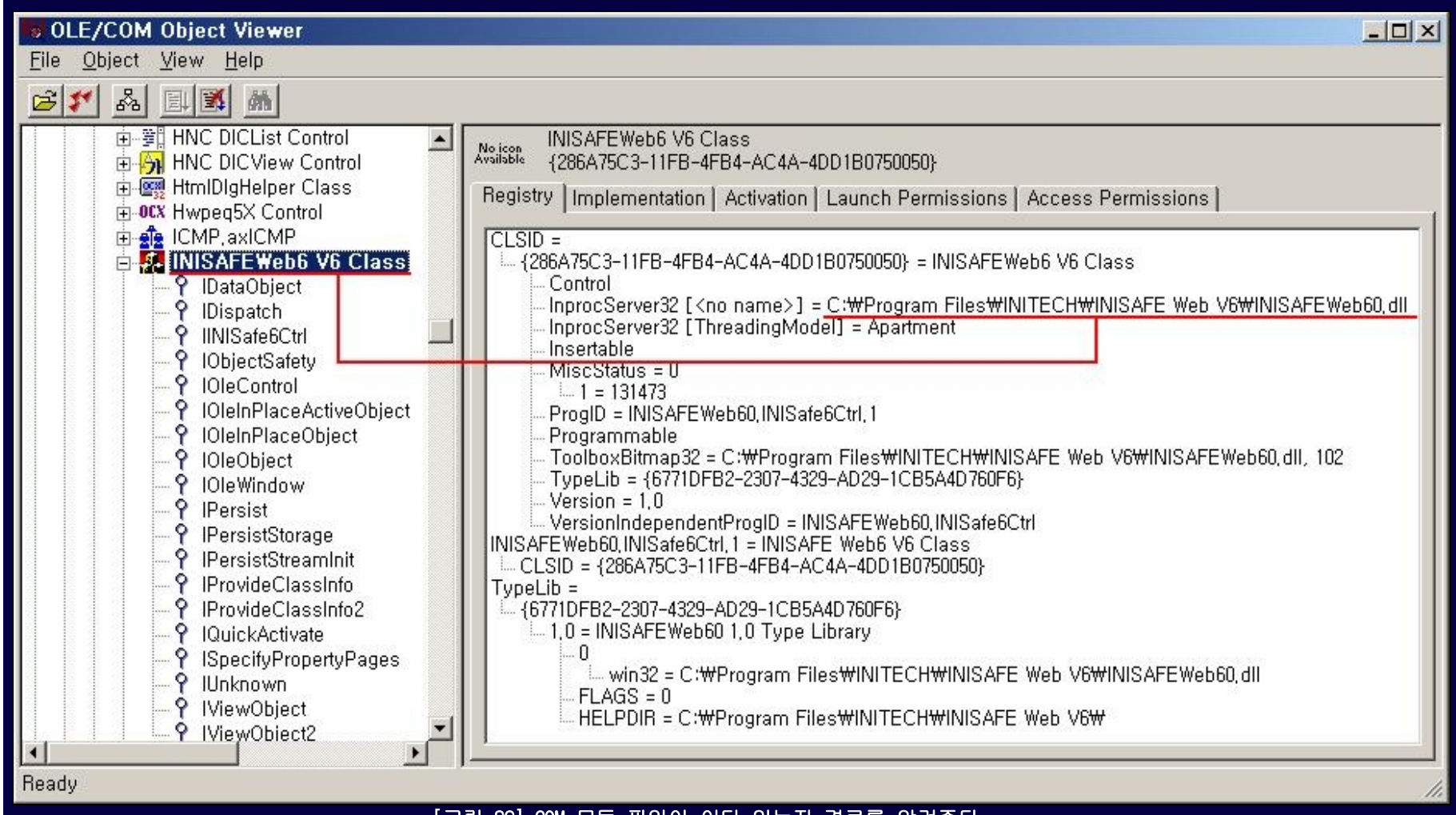
[그림 87] About 함수가 호출된 화면

우리의 PC 에 설치된 COM 모듈은 클래스 아이디(CLSID)만 알면 어떤 프로그래밍 언어를 사용하더라도 호출이 가능하다는 것은 프로그래밍을 해본 사람은 다 알 수 있는 내용일 것이다. 심지어는 PHP 로도 호출할 수 있다.



[그림 88] 후킹해 볼 함수를 선택하라

위의 두 함수는 필지가 후킹을 해볼려고 선택한 함수들이다. 그림78 에서 사용된 Decrypt 함수도 볼 수 있다. 오른쪽에 Decrypt 함수의 원형(프로토타입) 이 보인다. 이 함수는 두개의 인자로 ciphername 과 data 를 받아서 복호화 하는 루틴이라는 것을 알 수 있다. Encrypt 함수 역시 Decrypt 와 같은 프로토타입을 갖는다. 그럼 이 두 함수를 위에서 테스트했던 방식으로 자바스크립트 문법으로 호출할 수 있을 것이다. 하지만 문제가 한가지있다. 그것은 우리가 ciphername 과 data 로 넘어가는 인자 값이 어떤식으로 이루어져있는지 정보를 알지 못한다는 것이다.(혹자는 "나 그거 아는데.."라고 할지 모른다. 포인트를 제대로 잡고 읽어야 한다. 이 짓을 하는 목적이 후킹을 사용하는 범용적인 예에 불과하지 인증서 암복호화만을 위한 목적은 아니다.) 그렇기 때문에 후킹을 걸고 정상적인 통신과정에서 Decrypt 나 Encrypt 함수가 호출될 때 인자들을 가로채서 뭐가 넘어가고 있는지 눈으로 볼 필요가 있는 것이다. (사실상 후킹이 됐다면 굳이 자바스크립트를 사용할 필요가 없다. 후킹함수에 프로그래밍해서 넣으면 되기 때문이다.)



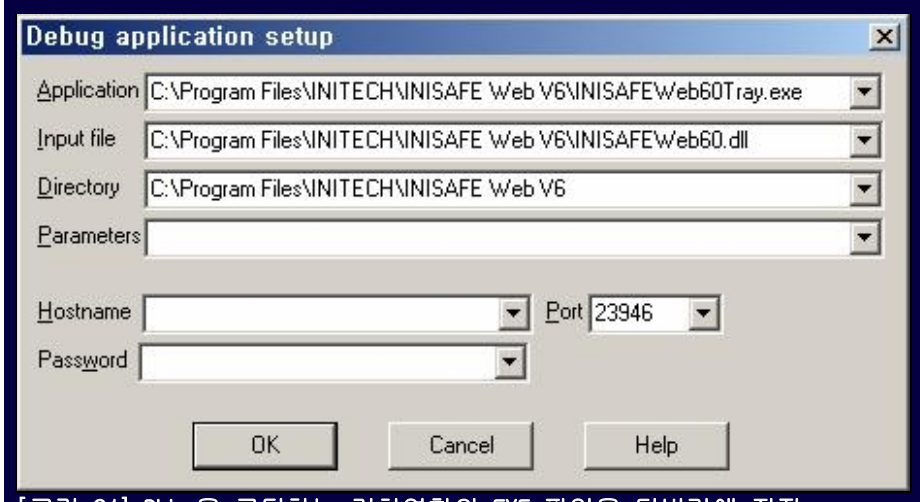
[그림 89] COM 모듈 파일이 어디 있는지 경로를 알려준다

OLE/COM Object Viewer 에서 해당 COM 모듈의 실제 파일이 어디있는지 경로가 나와있으므로 해당 파일을 IDA Pro 로 열어서 역어셈블을 한다.

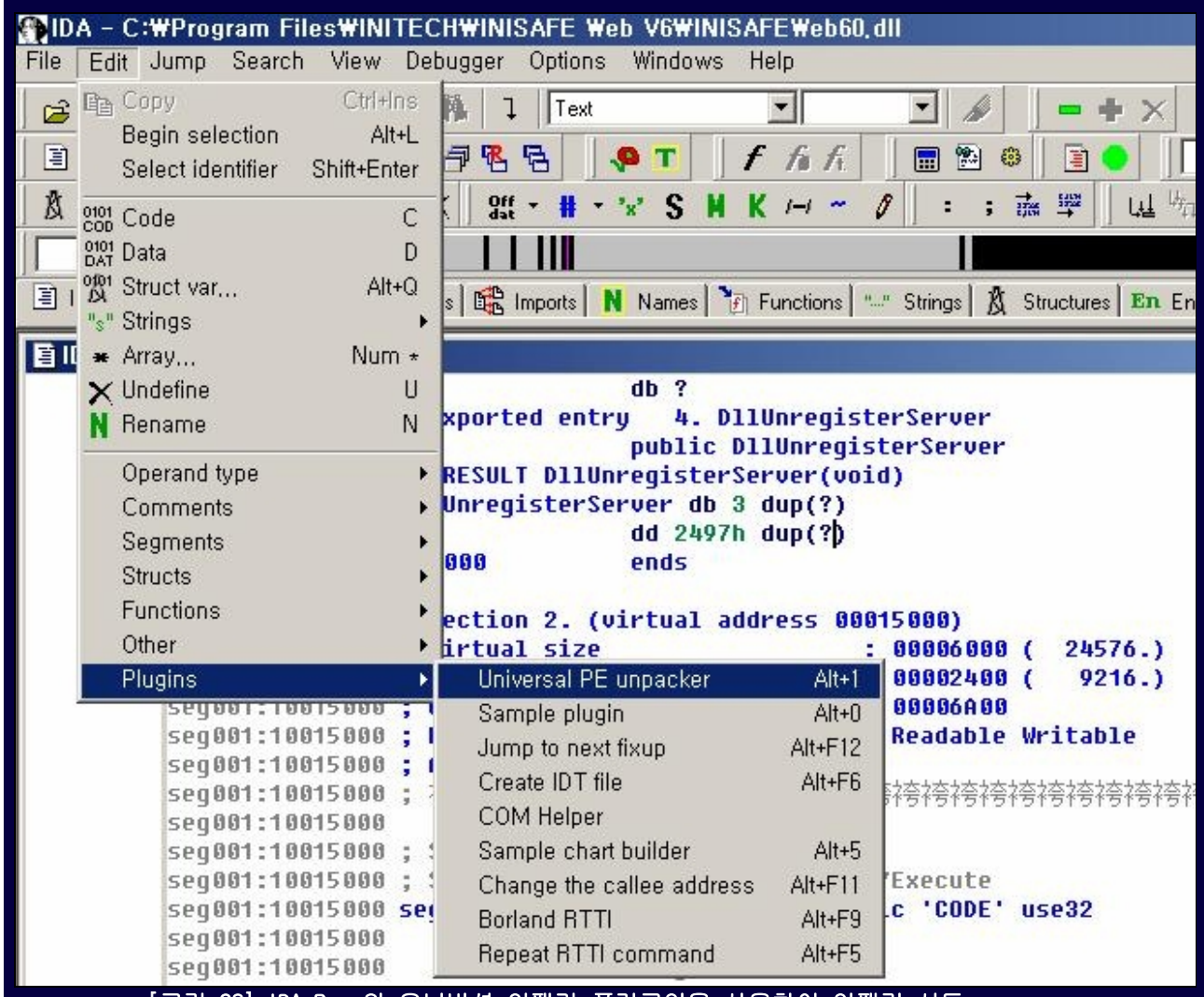
```
seg000:10002000                 dd 0BF5719F5h, 33AD3970h, 858E43CBh, 0D44BE1F5h, 0CBAB9AC3h
seg000:10002000                 dd 474B43DEh, 0EB4558EAh, 0F2544091h, 4C564105h, 0D966F149h
seg000:10002000                 dd 0FDA1704Bh, 0CFE2990Ch, 68C60C8Bh, 3A8EEF96h, 38E648C6h
seg000:10002000                 dd 9AC24941h, 93CBA119h, 0E2F7E2A7h, 9755D120h, 0CA74F32Fh
seg000:10002000                 dd 0BCE87DF5h, 2A30205Fh
seg000:10002800                 dd 68E1BDBDh, 1A236CB6h, 0F956458Ch, 32A21DE9h, 0CE7F727Ch
seg000:10002800                 dd 0B1E5EFF0h, 1F582FB6h, 7EFF967Ah, 194B5F09h, 769E9A59h
seg000:10002800                 dd 0DF7EDD51h, 9FC756D3h, 15027A5Ah, 57DA5DBAh, 6B335283h
seg000:10002800                 dd 91D99CEh, 0ABBEBE59h, 655986BBh, 0E770C058h, 3FD73C31h
seg000:10002800                 dd 94217B21h, 27B0EB73h, 71D492A9h, 0E96727B0h, 7B76BA12h
seg000:10002800                 dd 0EA6F0ACBh, 37F820D3h, 0F41512F6h, 0B6A7C1E3h, 822D89DFh
seg000:10002800                 dd 7D9144FDh, 286F42D0h, 0C473958Eh, 0FC94CBDAh, 6F9B6A26h
seg000:10002800                 dd 0ABC9FB0Dh, 2BC1C7B6h, 3020A614h, 0C5D12913h, 7F0EA9D5h
seg000:10002800                 dd 10B79EF9h, 7A7CE961h, 331C46D6h, 4B99EF4Dh, 94DBE674h
seg000:10002800                 dd 0C45BF6AAh, 0BB36BF73h, 0B5FE86D6h, 75DCF655h, 9F0D56B6h
seg000:10002800                 dd 0B69F46A0h, 35825A9Ch, 5D104C61h, 539166A8h, 7C768821h
seg000:10002800                 dd 64BFE26Ch, 9247A956h, 0ACD2EC8h, 0E0752B2Eh, 5477F309h
seg000:10002800                 dd 0E83F0A7Eh, 1A9E88A6h, 9AAFF0E5h, 0ABE82318h, 0DF2EB168h
seg000:10002800                 dd 1498FBA7h, 61AA4A96h, 20EF37BAh, 3AC51FFh, 0E3A5BE8h
seg000:10002800                 dd 52706ECCh, 0CCBFF4B9h, 2540C37Dh, 0C9498F3h, 0F8F142Fh
seg000:10002800                 dd 0F66586C2h, 7DF354B6h, 31C0B9E7h, 9712413Dh, 23D47E12h
seg000:10002800                 dd 0E775FD3h, 0AF8155C9h, 5D319E24h, 2B82413h, 788B76B0h
seg000:10002800                 dd 69768E1Eh, 265EF23Ch, 0E112861Fh, 23B548C1h, 0A0E9A738h
seg000:10002800                 dd 3ED1FC0Dh, 121E37F6h, 0DBF270BFh, 0AE9A780Ah, 0E84D4B6Ch
seg000:10002800                 dd 0E32FA2D9h, 0DE56ED2Bh, 6CA7CA4Bh, 79EA4063h, 0BE6B8E69h
seg000:10002800                 dd 0D957DA87h, 9F489D08h, 5B57DCECh, 6B723B26h, 3C7B4503h
seg000:10002800                 dd 0EF1BA76Eh, 46DFFBEh, 0A83D7D2Fh, 0C2AB0D72h, 5EB29925h
seg000:10002800                 dd 3E63579Dh, 6D7AE0F4h, 0D76D92BAh, 0E7C31DA6h, 134D79Eh
seg000:10002800                 dd 0CC88804Ch, 0DD6065EAh, 1660BB7Bh, 313B538Ch, 0B68381CEh
```

[그림 90] 팩킹되어 있다. (-_-;)

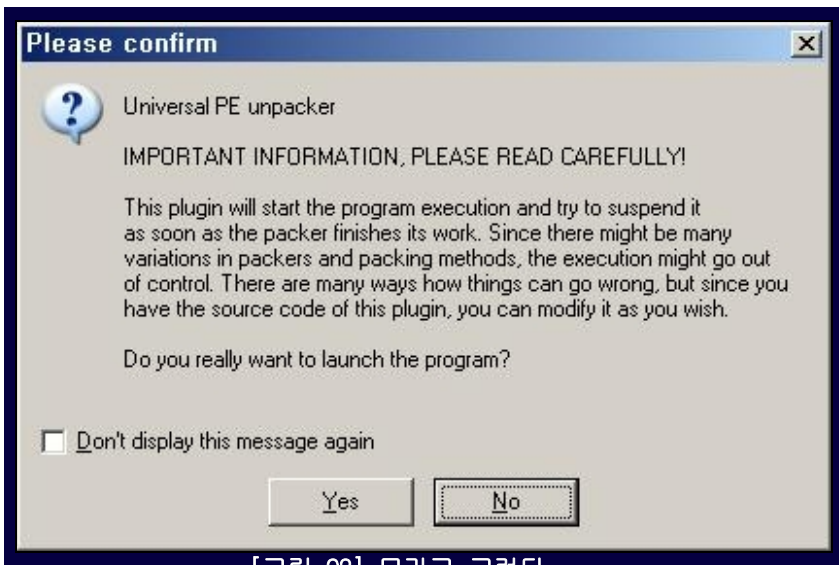
필자가 문서를 준비하고 있을 때는 분명히 팩킹이 안되어있었는데 얼마안된사이 팩킹이 되었는지 DLL 파일을 역어셈블 시키면 위와 같이 나온다.
일단 유니버셜 언팩커라고 하는 IDA Pro 의 만능 언팩킹 기능을 사용해서 간단히 해결해 보려고 했다.



[그림 91] DLL 을 로딩하는 런처역할의 EXE 파일을 디버거에 지정

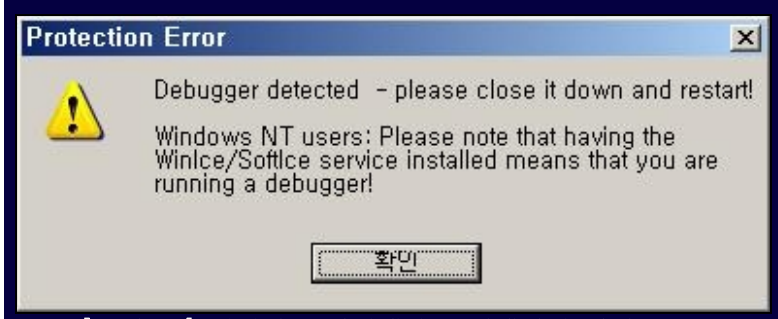


[그림 92] IDA Pro 의 유니버셜 언팩커 플러그인을 사용하여 언팩킹 시도



[그림 93] 모라고 그런다

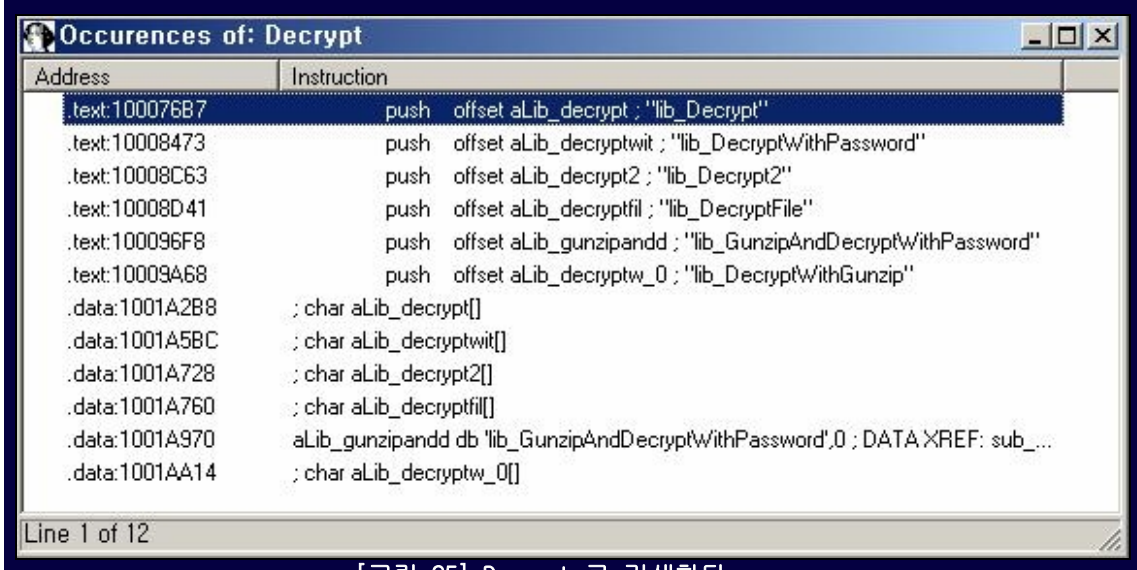
플러그인을 사용하면 잘 작동 안될 수 도 있으니까 구미에 맞게 소스를 수정해서 쓰라는 경고가 나온다.
YES 를 누르면 디버거 모드로 들어가면서 다음과 같은 메시지가 나오게 된다.



[그림 94] 디버거를 탐지했다고 언팩킹이 안됨

디버거를 탐지하였으니 종료한다고 나온다. Lesson-1 의 온라인 게임 후킹문서를 쓰던 며칠사이 안티디버깅 루틴과 팩킹이 INISAFE 에 추가되어 있었다. 왠일인가라는 생각이 들었다. 여태까지 돈과 관련된 민감한 사이트의 인증서 프로그램을 독식한 이 회사의 제품이 안티디버깅이나 팩킹이 안되어 있었던 것에 의아해하고 있었던 필자의 뒷통수를 때렸다. 드디어 ActiveX 의 DLL 도 팩킹이 되는 시점에 왔구나라는 생각이 들었다. 이런.. 제길... 필자가 문서를 작성하는데 한없이 시간을 투자할 여건이 안되기 때문에 언팩킹에 메달려있을 시간이 없었다. 어차피 언팩킹문서는 다음버전 문서에 포함해서 발표할 작정이었기 때문이다.
그래서 설치된 파일을 모두 다 지우고 다시 사이트에 접속해서 ActiveX 모듈을 다시 설치했다. 그랬더니 재미난 일이 생겼는데 팩킹이 되지 않은 버전이 설치되는 것이었다. 버전을 봤더니 v6,0,0,17 버전이었고 팩킹된 버전은 끝자리가 14 였다. 오히려 최신버전이 팩킹기능을 없애고 나온 것이다. 또, 웃긴건 팩킹된 버전을 쓰는 사이트와 그렇지 않은 사이트가 뒤섞여 있다는 것이다. 아마 필자생각에 모든 EXE 와 DLL 을 팩킹시켜서 배포하니까 느려서 다시 팩킹이 안된 상태로 재배포 한 것 같다는 생각이 든다. 우리나라 같이 속도에 민감한 나라는 웹에서 프로그램이 뜨는데 1초만 느려져도 "파파팍" 눈치채는 사람이 많아서 그런가.. ㅎㅎ 어쨌든 간단하게 테스트만 진행하고 넘어갈 것이므로 다음 설명으로 넘어가겠다.

다음은 INISAFEWeb60.dll 파일에서 Decrypt 로 검색한 것이다.



[그림 95] Decrypt 로 검색한다.

```
push    ecx
and     dword ptr [ebp-10h], 0
push    esi
mov     esi, ecx
call    sub_1000701D
test    eax, eax
jz      short loc_10007702
push    offset aLib_decrypt ; "lib_Decrypt"
push    dword ptr [esi+4] ; hModule
call    ds:GetProcAddress
test    eax, eax
jz      short loc_10007702
```

[그림 96] Decrypt 함수는 다른 DLL 에 있다.

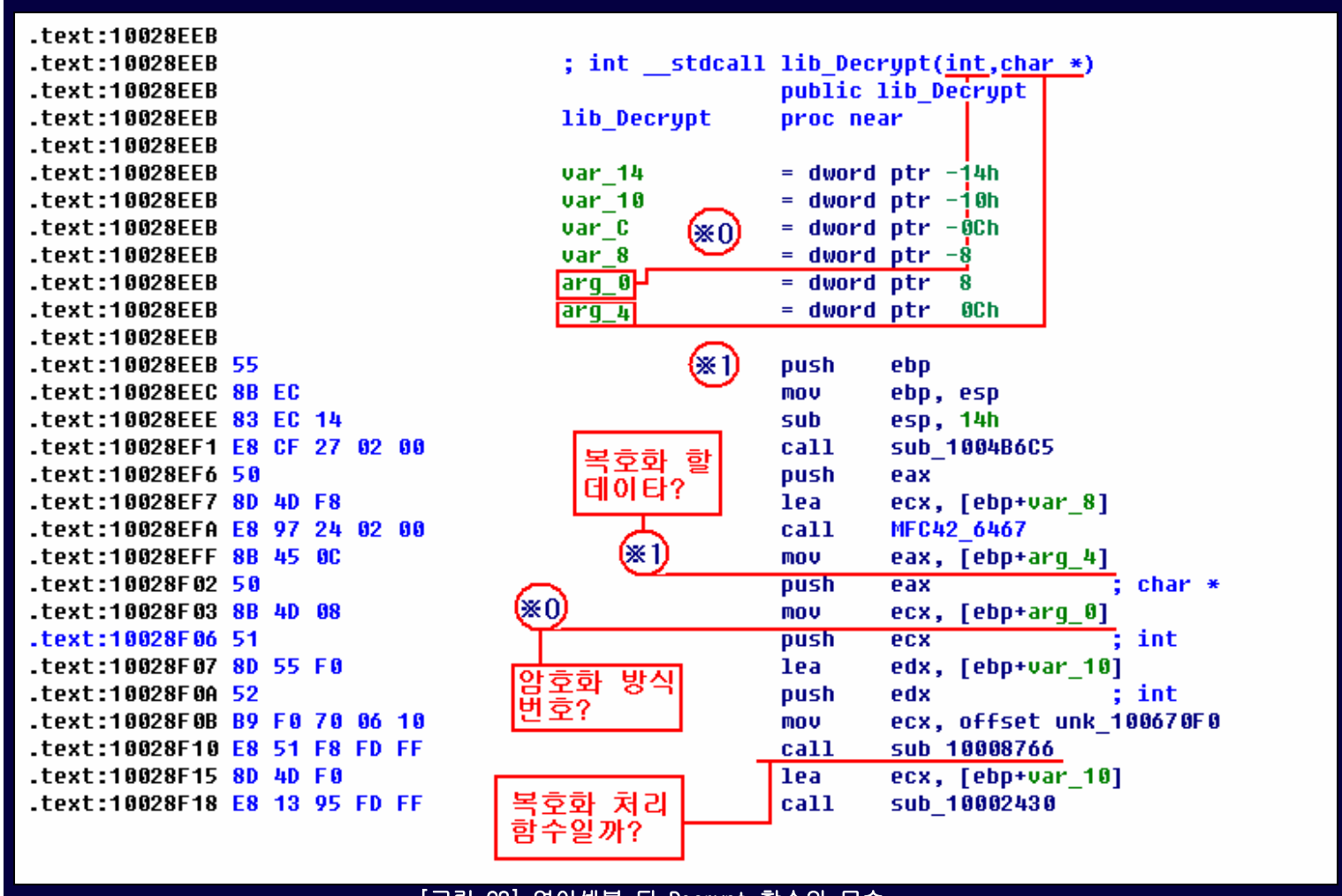
검색한 것을 따라가면 그림96 과 같이 GetProcAddress 가 나오는 것을 볼 수 있다. 이 경우 lib_Decrypt 문자열 주소가 저장되고 GetProcAddress 함수가 바로 뒤따라 나오는 걸로 봐서 다른 DLL 파일에 Decrypt 함수가 있다는 것을 바로 알 수 있다. GetProcAddress 함수가 주로 다른 DLL 에 있는 함수를 선택해서 사용할 때 쓰이는 함수이기 때문이다.

그럼 어떤 DLL 에 있을까? lib_Decrypt 문자열을 더블클릭하고 크로스레퍼런스를 따라가라. 그리고 그 위치에서 약간 윗쪽으로 올라가서 보면 다음과 같은 곳이 보인다.

```
.data:1001A190                  db    0
.data:1001A191                  db    0
.data:1001A192                  db    0
.data:1001A193                  db    0
.data:1001A194 aIniwebcrypto_d  db 'WINIWebCrypto.dll',0 ; DATA XREF: sub_1000701D+36↑o
.data:1001A1A6                  align 4
.data:1001A1A8 ; char aLib_about[]
.data:1001A1A8 aLib_about       db 'lib_About',0        ; DATA XREF: sub_1000713B+C↑o
.data:1001A1B2                  align 4
.data:1001A1B4 ; char aLib_resession[]
.data:1001A1B4 aLib_resession   db 'lib_ReSession',0    ; DATA XREF: sub_1000715D+C↑o
.data:1001A1C2                  align 4
.data:1001A1C4 ; char aLib_getsession[]
```

[그림 97] INIWebCrypto.dll 에 실제 함수가 있음

그렇다.. 그림97 에서 처럼 실제 함수는 INIWebCrypto.dll 파일에 있다는 것이다. 그렇다면 실제 함수가 있는 곳으로 가보자. 다시 IDA Pro 로 역어셈블을 시도한다.

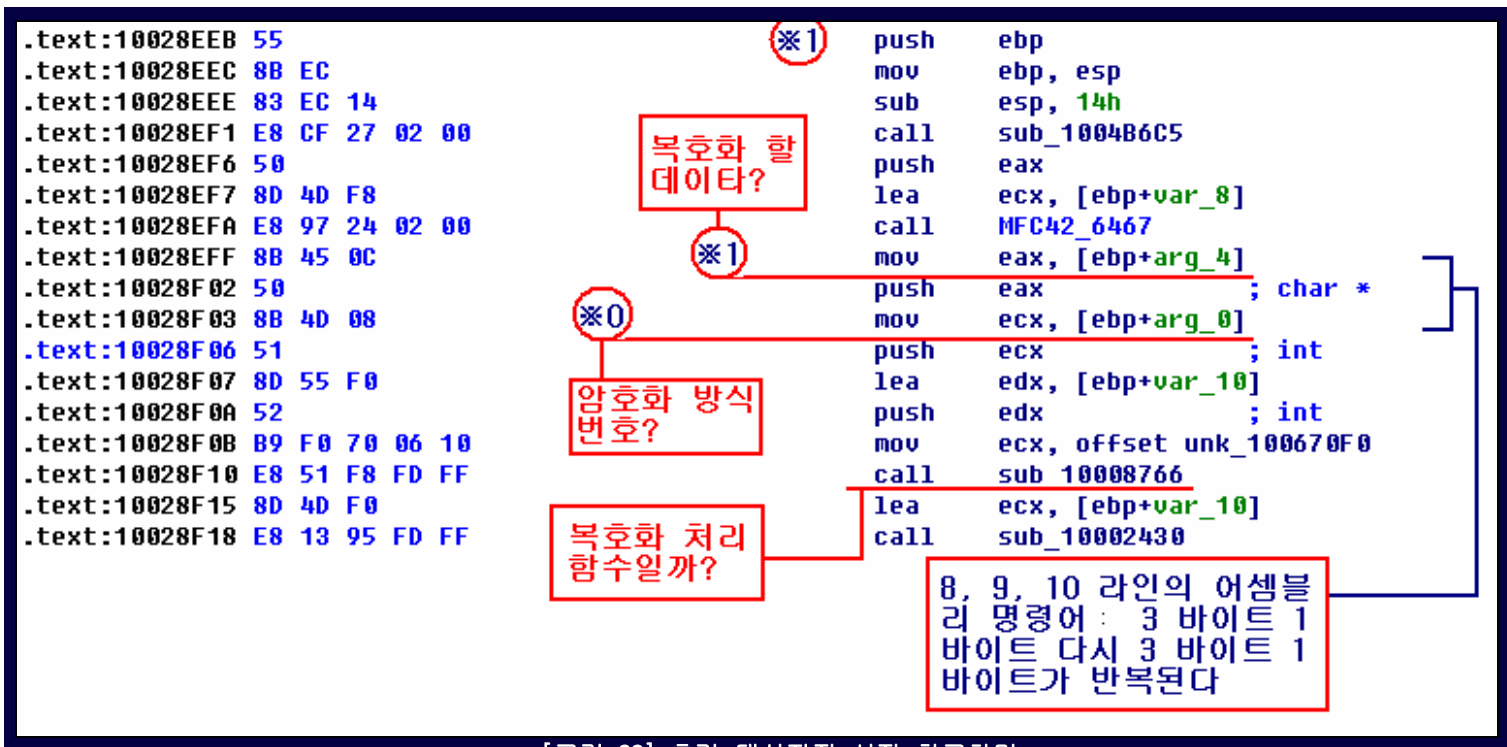


[그림 98] 역어셈블 된 Decrypt 함수의 모습

위 그림98 은 INIWebCrypto.dll 파일의 Decrypt 함수가 역어셈블된 부분인데 여기서 ※0 항목이 이 Decrypt 함수의 첫번째 인수이고 ※1 항목이 두번째 인수이다. 그림78 의 자바스크립트로 호출하는 Decrypt 함수를 보라. obj.Decrypt(ciphername, data) 라고 호출할 것이다. 여기서 ciphername 은 arg_0([ebp+0x08]) 이고 첫번째 인수이며 그림98 에서 ※0 항목이다. 마찬가지로 data 는 arg_4([ebp+0x0C]) 이고 두번째 인수이며 그림98 에서 ※1 항목이다. 즉, arg_0 인 [ebp+0x08] 값과 arg_4 인 [ebp+0x0C] 값을 후킹으로 가로채서 외장 로거(External Logger)인 DebugView 에 출력하도록 할 수 있다.

자.. 여기서 질문을 한가지 해보자..
코드인터셉트 후킹은 지점을 알아야 가로챌 수 있다고 했다. 가로채려는 지점을 어디로 선정하면 가장 최적의 경우가 될 것인가?
그 지점은 ebp 를 베이스로 지정할 수 있는 시점부터라면 어디든 상관이 없다. 다시말해 mov ebp, esp 라는 코드가 그림98의 두번째 어셈블리 라인에서 보일텐데 이 ebp 와 esp 를 일치시키는 시점이 바로 Decrypt 함수가 스택을 안전하게 사용하기 위해 스택기준점(베이스 프레임포인터) 을 잡는 시점이다. 그렇기 때문에 최소한 2번째 라인보다는 뒤에서 가로채기를 해야 한다는 것이다. 생각을 해보라.. 예전에 BOF 를 죽도록 공부하던 시절을 기억하는가.. 리턴어드레스라고 하면 자다가도 벌떡일어나서 ebp+0x04 라고 저절로 튀어나오던 때가 있었다.. ebp+0x04 가 리턴어드레스라고 무턱대고 외칠때 ebp 는 이미 함수속으로 진입한 시점의 값이고 스택한테 뒷일을 부탁받은 상태이다. 즉, 스택(주의! esp 라고 안부르고 스택이라고 말하고 있다)과 ebp 는 하나가 된 상태이다.(mov ebp, esp) 스택은 ebp 에게 뒷일을 부탁한다고 혼자서 달리기 시작한다. ebp 는 함수에 진입해서 스택한테 뒷일을 부탁받으면 그 자리에서 계속 기다리고 있지만 스택은 나중에 돌아오겠다는 약속만 한채 혼자서 몸을 던진다. 그래서 우리는 ebp 가 스택의 고정값이라는 점을 이용해 리턴어드레스를 쉽게 지정할 수 있게 되는 것이다. 또한 ebp 를 이용하여 ebp+0x08 은 함수호출 시 넘어온 첫번째 인수위치를 뜻하고 ebp+0x0C 는 두번째 인수가 된다는 공식이 되는 것이다.(EBP 와 ESP 의 은밀한 관계를 알고 싶으면 뒤에서 덧붙일 그림을 참고하라!) 이것이 베이스프레임 포인터(ebp)의 역할이다. 그러므로 이 코드가 후킹되면 곤란하다.

후킹위치 선정의 설명이 길어졌는데 어쨌든 세번째 어셈블리 라인부터 후킹지점이 될 수 있는지 따져보자. 그림98 의 sub esp, 14h 는 스택포인터(esp)에서 0x14 를 빼서 직접적인 스택포인터의 변경을 가한다. (스택포인터는 EIP 처럼 CPU 가 사용하는 스택의 현재위치를 말하므로 스택이라고 말해도 개념상 동일하다) 원래의 위치에서는 이 명령이 지역변수 공간할당으로 작용하지만 우리의 후킹함수 속으로 복사되서 작동할 땐 0x14 만큼 스택포인터에서 빼버리기 때문에 우리의 후킹함수가 혼란스럽게 된다.(우리의 후킹함수라고 스택을 사용 안하는건 아니니까) 결국 변경된 스택포인터를 기준으로 리턴주소를 참조하게 되므로 나중에 프로그램이 뻑나게 된다. 또한, 우리는 Lesson-1 에서 후킹시 CALL 을 후킹하면 우리의 후킹함수 속에서 호출되므로 뻑난다고 하였다. 그러므로 CALL 도 후킹지점이 될 수 없다. 그렇다면 이제 5 번째 6번째를 보라. push eax / lea ecx, [ebp+var_8] 두 명령의 OPCODE 길이를 합친다고해도 총 5 바이트가 되지 못한다. 다시 다음 그림99 에서 8, 9, 10 어셈블리 라인을 보라.



[그림 99] 후킹 대상지점 선정 참고화면

위에서 보면 8, 9, 10 이라고 설명을 붙여 놓았는데 무슨 소리냐면 8 라인의 어셈명령과 9 라인의 어셈명령이 OPCODE 를 합쳐보면 총 4 바이트이다. 마찬가지로 9, 10 라인의 OPCODE 를 합쳐봐도 명령의 길이가 총 4 바이트이다. 즉, 5 바이트 이상이 되야 후킹함수를 삽입할 수 있기 때문에 이를위해 8, 9, 10 과 같이 세개의 어셈블리 명령을 가로채야 한다는 결론이 나올 수 있다.(8, 9 라인의 4 바이트와 10 라인의 1 바이트를 가로채어 5 바이트를 후킹하는 건 할 수 없다. 각 OPCODE 는 고정된 명령크기를 갖고있기 때문이다.) 얼마나 비효율적인가? 후킹함수에 3 개의 어셈블리 명령이 복사되므로 3 바이트, 1 바이트, 3 바이트의 총 7 바이트가 복사되며 후킹함수가 작동할 때마다 실행될 것이다. 더 비효율적인 것은 3 개의 어셈블리 명령이 그 속에서 아무런 탈을 일으키지 않도록 보장하는 코드를 작성해야 한다는 점이다. 지금같은 8, 9, 10 어셈명령을 후킹할때 발생하는 문제점은 push eax 때문에 후킹함수 속에서 4 만큼 스택을 증가시키므로 복구작업을 하지않으면 리턴 시 엉뚱한 주소로 점프해서 프로그램이 박살날 것이다. 혹시라도 8, 9, 10 어셈명령들 중에서 변수를 참조하는 놈이 있었다고 가정했을때 ebp 기준기법이 아닌 esp 기준기법이 사용됐었다면 더 골치아픈 변수 복구작업도 해줘야 한다. 후킹시 얼마나 쉽게 후킹을 하느냐 아니냐는 후킹지점을 얼마나 잘 선정하느냐가 될 수 있는 것이다. (디아블로2 후킹 지점이 스택을 다루는 부분을 피해간 이유일 것이다.)

다행히도 필자는 깔끔하고 다루기 쉬운 1 개의 어셈명령으로 된 후킹대상 지점을 바로 14 번째라인에서 찾았다. 그림99 에서 보는 것처럼 mov ecx, offset unk_100670F0 명령이며 코드 길이가 5 바이트이다. 또한 ecx 레지스터값 밖에 바꾸지 않아서 스택복구같은 처리가 따로 필요없다. 게다가 이 지점에서 [ebp+0x08] 처럼 인수를 뽑는 것도 아주 쉽다.(호출규약을 보면 알 수 있음) 이제 이 부분을 후킹하는 코드를 만들었다.

```
/*
    Programming : Reversing by AmesianX in powerhacker.net
*/

////////////////////////////////////////////////////////////////////////////
#define THIS_IS_SERVER
#include "..₩D2HackIt.h"
#include "DbgPrint.h"
////////////////////////////////////////////////////////////////////////////
// Decrypt()
// ciphername = 암호화 방식
// data = 디코딩 데이타
////////////////////////////////////////////////////////////////////////////
void __fastcall Decrypt(int ciphername, char *data)
{
    #ifdef _DEBUG
    DbgPrintf("Decrypt = %d, %s₩n", ciphername, data);
    #endif
}

////////////////////////////////////////////////////////////////////////////
// Decrypt_STUB()
// ------------------------------------------------------------------------
// [ebp+0x08] = ciphername
// [ebp+0x0C] = data
////////////////////////////////////////////////////////////////////////////
void __declspec(naked) Decrypt_STUB()
{
    __asm {
        nop                                 // Make room for original code
        nop
        nop
        nop
        nop
        nop
        nop

//      .text:10028F0B B9 F0 70 06 10       mov     ecx, offset unk_100670F0

        pushad
        mov     edx, DWORD PTR [ebp+0x0C]
        mov     ecx, DWORD PTR [ebp+0x08]
        CALL    Decrypt
        popad

        ret
    }
}
```

[그림 100] 후킹코드 작성

위와 같이 후킹코드를 작성하며 Lesson-1 에서 GameChatHooker.cpp 와 같이 후킹함수만 넣어둔 파일을 작성해서 VC++ 의 프로젝트에 추가해주면 된다.

그리고 ServerStartStop.cpp 에 패치루틴을 수정한다.

```
    ////////////////////////////////////////////////////////////////////
    // Check if D2HackIi.ini exisits
    ////////////////////////////////////////////////////////////////////
    if (_access(psi->IniFile, 0))
    {
        LPSTR t=new char[strlen(psi->IniFile)+50];
        sprintf(t, "Unable to open ini-file:Wn%s", psi->IniFile);
        #ifdef _DEBUG
        DbgPrintf(t);
        #endif

        delete t;
        return FALSE;
    }

    DbgPrintf("ProcessID = %dWn", psi->pid);

    // 메모리의 바이너리 이미지(DLL)에서 Decrypt 함수지점 검색
    if (!GetFingerprint("D2HackIt", "Decrypt", psi->fps.Decrypt))
    {
        #ifdef _DEBUG
        DbgPrintf("Decrypt Not Found!");
        #endif

        return FALSE;
    }

    // Decrypt 함수부분 인터셉트
    Intercept(INST_CALL, psi->fps.Decrypt.AddressFound, (DWORD)&Decrypt_STUB, psi->fps.Decrypt.PatchSize);
    return TRUE;
}
```

[그림 101] ServerStart 함수부분 코드

Lesson-1 에서처럼 GetFingerprint 함수부분과 Intercept 함수부분 말고는 특이한게 없을 것이다.
나머지 프로토타입 선언과 구조체 추가부부은 Lesson-1 에 이미 다 설명된 부분이므로 추가 설명하지 않는다.

```
1  [FingerprintData]
2  ; Decrypt
3  Decrypt=INIWebCrypto.dll,5,12,8B450C508B4D08518D55F052B9F0700610E851F8FDFF8D4DF0
```

후킹할 함수인 Decrypt 가 DLL 에 있으니까.. DLL 을 지정한다. 이것이 바로 D2HackIt 의 강점 중에 하나다.

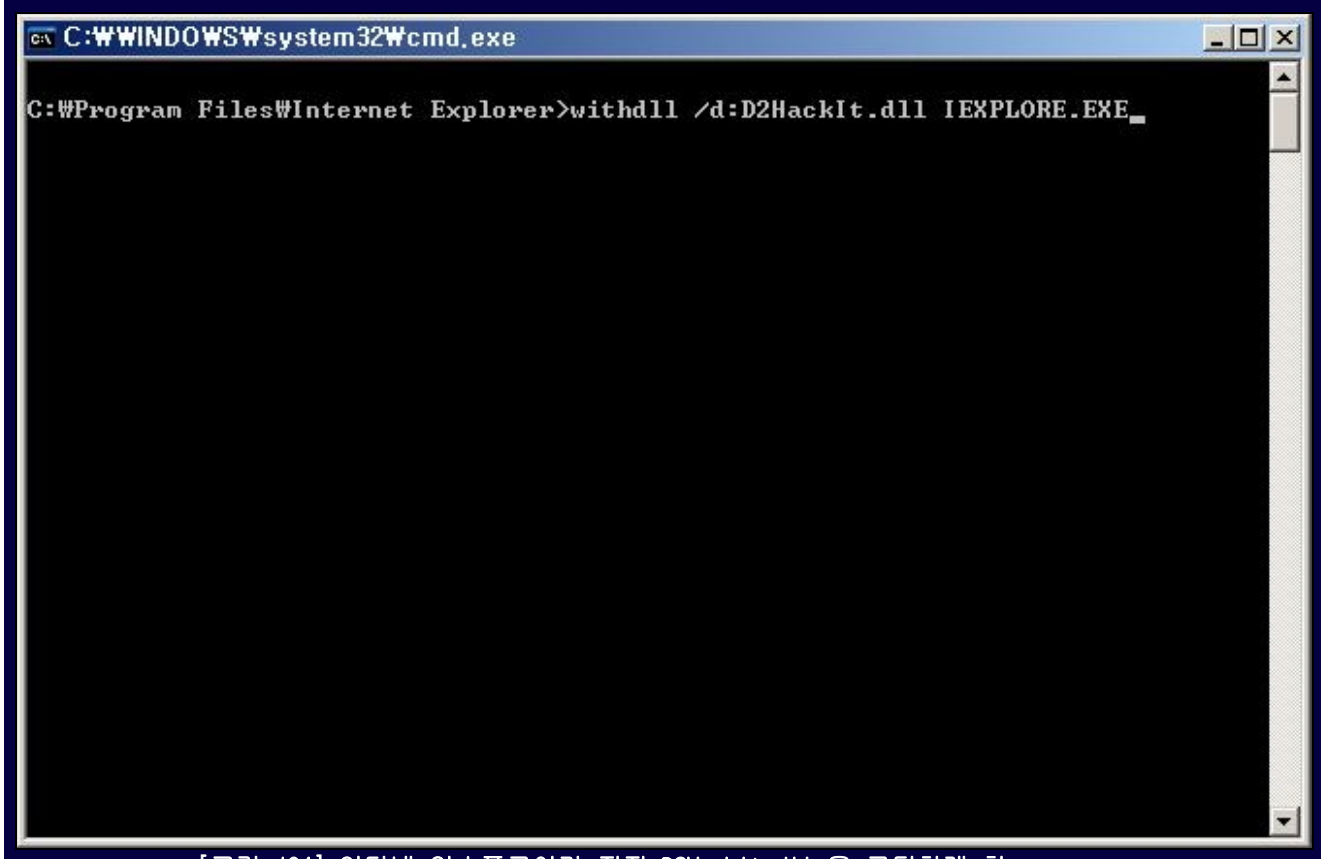
[그림 102] 후킹할 실행파일 이미지(DLL)와 검색할 바이너리스트링 지정

DLL 도 PE 구조 이기 때문에 DLL 에 DLL 을 끼워넣을 수 있다. 또한 이런 방식은 이미 필자가 다른 프로그램들을 후킹해보면서 많이 써왔던 방법이다.

```
C:WWINDOWSWsystem32WCMD.exe

C:WProgram FilesWINITECHWINISAFE Web V6>setdll /d:D2HackIt.dll INIWebCrypto.dll
Adding D2HackIt.dll to binary files.
  INIWebCrypto.dll:
    D2HackIt.dll
    inicrypto30.dll -> inicrypto30.dll
    NSLDAP32V11.dll -> NSLDAP32V11.dll
    VERSION.dll -> VERSION.dll
    WININET.dll -> WININET.dll
    MFC42.DLL -> MFC42.DLL
    MSVCRT.dll -> MSVCRT.dll
    KERNEL32.dll -> KERNEL32.dll
    USER32.dll -> USER32.dll
    GDI32.dll -> GDI32.dll
    ADVAPI32.dll -> ADVAPI32.dll
    SHELL32.dll -> SHELL32.dll
    ole32.dll -> ole32.dll
    OLEAUT32.dll -> OLEAUT32.dll
    urlmon.dll -> urlmon.dll

C:WProgram FilesWINITECHWINISAFE Web V6>
```

[그림 103] DLL 이라서 아예 DLL 바이너리 속에 정보를 박아버린다

앞서 Lesson-1 에서는 withdll 을 사용하였는데 지금은 상황이 좀 다르다. 왜냐면 DLL 은 직접 CreateProcess 로 생성해서 실행할 수 없기 때문이다.(rundll 이 있다고 테클걸려나.. ㅎㅎ) DLL 을 리버싱해서 함수를 후킹하면 이놈을 끌고 들어가는 실행파일(EXE)이 있어야 된다. 그럼 실행파일을 withdll 로 실행시키면 되지 않겠냐고 하겠지만, 그것도 맞는 말인데 생각을 한번 잘 해보자.. 이 실행파일이란 놈은 내가 실행시켜서 작동안하는 경우도 있다. 즉, ActiveX 계열 프로그램은 인터넷의 특정 사이트에 접속했을때 서버에서 전송받는 ActiveX 코드로 작동되는 것이 대부분이다. 아마 직접 실행시키면 실행되는 경우보다 안되는 경우가 더 많을 것이다. 그렇다면 실행파일을 뜯어서 분석할 것인가? 언제 실행파일까지 분석하고 앉아있나.. 요즘같이 먹고살기 힘든 세상에 시간이 그렇게 많이 남아도는 사람이 본인만 있지는 않을 것이다.(ㅋㅋㅋ 사실 본인도 바빠서 실행파일 분석까지는 할 수 없고 하고 싶지도 않다.. 재미도 없을 뿐더러 아주 지겨운 작업이다..)
그렇기 때문에 우리는 INIWebCrypto.dll 이 알아서 D2HackIt.dll 을 끌고 들어가라(로딩하라)고 요구해야 한다. 이 작업을 하기 위해서 Detours 후킹 라이브러리에서 제공하는 setdll 이라는 유틸리티를 사용했다.. (휴~ 엄청 많이 썼다.. 힘들어 죽갔군..)
그림103 에서 빨간 줄을 쳐 놓은 것이 보이는가..?
D2HackIt.dll 이 INIWebCrypto.dll 파일의 임포트테이블에 처박히는 모습을 찍은 것이다. 이제부터 INIWebCrypto.dll 을 끌고 들어가는 놈은 D2HackIt.dll 도 같이 끌고들어가서 INIWebCrypto.dll 을 후킹하게 될 것이다. 우리는 인터넷을 사용하는 도중 어떤 HTML 코드에 의해서 ActiveX 가 실행되던지 상관할 필요가 없다. 단지 우리는 DebugView 만 켜놓고 로깅을 보면된다.
그러나 세상만사 쉬운게 하나도 없다고 About 함수를 호출하는 자바스크립트 명령을 Decrypt 함수로 바꾼뒤 html 파일로 저장해서 열어봤더니 INIWebCrypto.dll 을 로딩할 수 없다는 오류가 발생하였다. 예상에 없었던 상황이 발생하니 어디에서 오류가 나는지 확인을 해야했다. 그러나 DebugView 에 아무런 로깅도 남지 않는 것이었다. 문제점을 찾던중 INIWebCrypto.dll 대신 INISAFEWeb60.dll (실제 COM모듈)에 D2HackIt.dll 을 심어보기도 했지만 프로세스 익스플로어로 봤을때 D2HackIt.dll 이 로딩조차 되지 않고 있다는 것을 알았다. 그래서 필자는 방법을 바꿔가며 여러가지 테스트 했고 결국 프로세스 공간에서 이미지를 갖고오지 못하고 있다는 것을 알게 되었다. 전혀 예상치 못한 일이 생긴 것이다. 그래서 방법을 바꿔 DLL 에 D2HackIt.dll 을 삽입하지 않고 인터넷 익스플로어를 실행시키면서 주입하도록 변경하였다.



[그림 104] 인터넷 익스플로어가 직접 D2HackIt.dll 을 로딩하게 함

그래서 필자는 위와 같이 withdll 을 이용하여 D2HackIt.dll 을 인터넷 익스플로어에 주입시켰다.
이때 ServerStartStop.cpp 파일을 수정하였고 첫부분의 DLL 강제로딩 목록에 INISAFE 모듈들을 나열하였다.

```
#define THIS_IS_SERVER
#include "..₩D2HackIt.h"

// These are the dll's we want to force-load to get them in memory.
//char* NeededDlls[] = { "D2Common.dll", "D2Game.dll", "D2Multi.dll", "D2Client.dll", NULL };
char* NeededDlls[] = { "C:₩₩Program Files₩₩INITECH₩₩INISAFE Web V6₩₩INISAFEWeb60.dll",
                       "C:₩₩Program Files₩₩INITECH₩₩INISAFE Web V6₩₩INIWebCrypto.dll",
                       "INIcrypto30.dll", NULL };
```

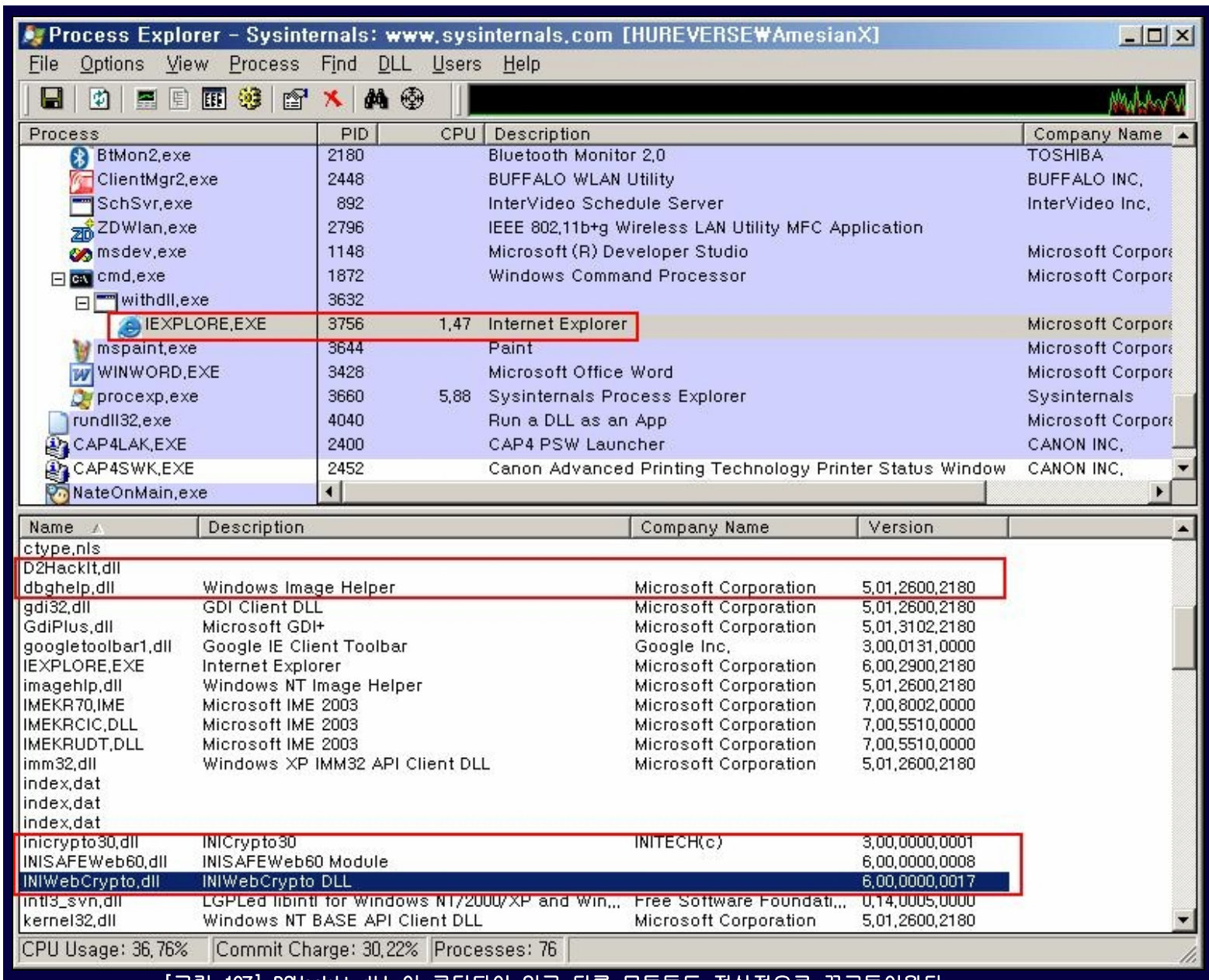
[그림 105] DLL 을 강제로 로딩함(디아블로2 후킹에서 사용하는 방법)

여기서 주의할 것이 하나 있는데 각각의 파일들이 어떤 디렉토리에 위치해 있는가이다. D2HackIt.dll 과 D2HackIt.ini 는 C:₩WINDOWS₩SYSTEM32 디렉토리에 있다. 그리고 INIWebCrypto.dll 과 INISAFEWeb60.dll 은 C:₩Program Files₩INITECH₩INISAFE Web V6 디렉토리에 있으며 INIcrypto30.dll 은 D2HackIt 과 마찬가지로 SYSTEM32 디렉토리 밑에 있다. 필자가 withdll 을 이용하여 인터넷 익스플로어에 D2HackIt.dll 을 끼워넣는 위치는 C:₩Program Files 밑에있는 Internet Explorer 디렉토리이다. 그렇기 때문에 강제로딩 항목에 전체경로를 적어준 것이다. 이렇게 하지 않으면 프로세스 익스플로어로 봤으때 DLL 들이 강제로딩 되지 않는다.(환경변수의 PATH 에 적어주지 않았으므로.. 이때까지 이 부분이 문제가 되리라고 미처 생각을 하지 못했음. 뒤에서 이유가 나옴..)
환경설정 파일인 D2HackIt.ini 도 다음 처럼 바꿔야 한다.(D2HackIt 은 자신이 존재한 디렉토리를 베이스로 하기 때문에..)

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----0----+----1----+----2-
[FingerprintData]
; Decrypt
Decrypt=C:₩Program Files₩INITECH₩INISAFE Web V6₩INIWebCrypto.dll,5,12,8B450C508B4D08518D55F052B9F070061 0E851F8FDFF8D4DF0
```

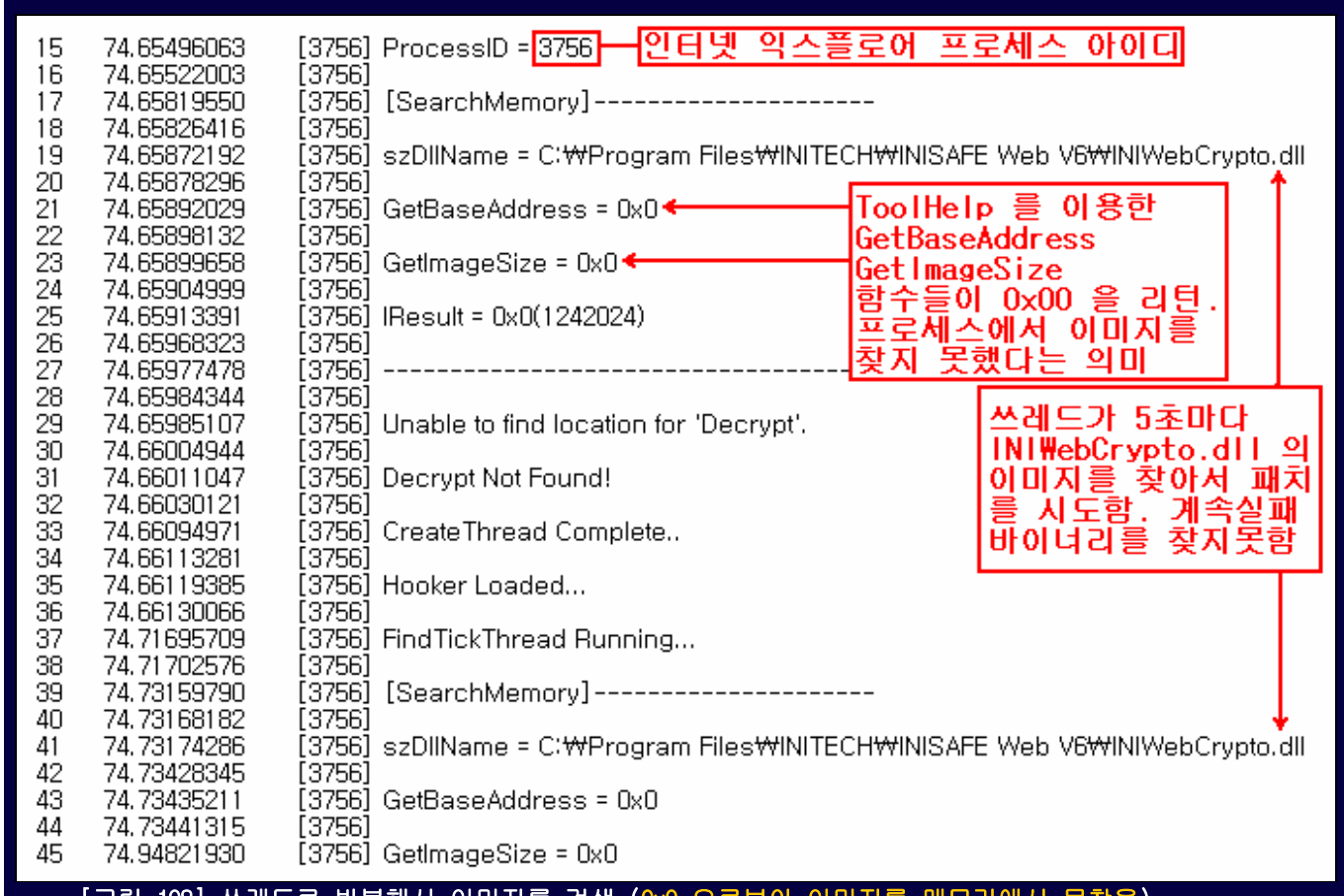
[그림 106] 환경설정파일 D2HackIt.ini 파일에 전체경로 지정

그 결과 그림107 처럼 프로세스 정보를 보면 정상적으로 인터넷 익스플로어의 프로세스 공간에 DLL 들이 강제로딩된 것을 확인할 수 있다.
(필자의 인터넷 익스플로어는 실행시키면 그냥 빈페이지임.. 그렇기 때문에 강제로딩 되었음을 알 수 있음)



[그림 107] D2HackIt.dll 이 로딩되어 있고 다른 모듈들도 정상적으로 끌고들어왔다.

위 그림107 은 인터넷 익스플로어 실행시 D2HackIt.dll 을 주입시킴으로써 D2HackIt.dll 이 정상적으로 다른 DLL 들을 끌고 들어왔음을 보여준다. 인터넷 익스플로어의 프로세스 공간에 로딩되어 있기 때문에 당연히 INIWebCrypto.dll 이 수정될 것으로 예상했으나 프로세스 익스플로어에 뻔히 보이는 로딩된 이미지를 찾지 못하는 것이었다. 심지어 DLL 이 로딩되는 시간차가 발생되는 것이 아닌가 생각되어(INIWebCrypto.dll 이 로딩되는 것보다 D2HackIt 이 먼저 로딩되는 상황이 생긴 것으로 착각..) INIWebCrypto.dll 이미지를 메모리에서 찾지 못할 경우 5 초 간격으로 반복해서 바이너리를 스캔하는 함수를 쓰레드로 작성하여 돌렸다. 그 시도가 다음의 화면모습이다.



[그림 108] 쓰레드로 반복해서 이미지를 검색 (0x0 으로보아 이미지를 메모리에서 못찾음)

그림108 과 같이 GetBaseAddress 와 GetImageSize 함수가 계속 0x00 을 반환하면서 실패를 반복하고 있다. 이 두개의 함수는 DbgHelp.dll 의 toolhelp 관련 함수로 프로세스를 뒤져서 로딩된 메모리 위치와 그 이미지의 사이즈를 리턴하게 된다. 결국 이 부분을 해결하기 위해서 이삼일을 공중으로 날려버렸고 문제점을 알아내게 되었다. 이 문제점은 뒤에서 보겠지만 시작에 불과하다..

DebugView on ₩₩HUREVERSE (local)

File Edit Capture Options Computer Help

#	Time	Debug Print
1925	26.76077080	[428]
1926	26.76082611	[428] lpme->szModule = MSVCRTD.dll, ModuleName = C:₩Program Files₩INITECH₩INISAFE Web V6₩INIWebCrypto.dll
1927	26.76091003	[428]
1928	26.76096535	[428] lpme->szModule = dbghelp.dll, ModuleName = C:₩Program Files₩INITECH₩INISAFE Web V6₩INIWebCrypto.dll
1929	26.76104736	[428]
1930	26.76110458	[428] lpme->szModule = INISAFEWeb60.dll, ModuleName = C:₩Program Files₩INITECH₩INISAFE Web V6₩INIWebCrypto.dll
1931	26.76119041	[428]
1932	26.76124954	[428] lpme->szModule = MFC42.DLL, ModuleName = C:₩Program Files₩INITECH₩INISAFE Web V6₩INIWebCrypto.dll
1933	26.76133156	[428]
1934	26.76138878	[428] lpme->szModule = OLEPRO32.DLL, ModuleName = C:₩Program Files₩INITECH₩INISAFE Web V6₩INIWebCrypto.dll
1935	26.76241302	[428]
1936	26.76247978	[428] lpme->szModule = MFC42LOC.DLL, ModuleName = C:₩Program Files₩INITECH₩INISAFE Web V6₩INIWebCrypto.dll
1937	26.76256943	[428]
1938	26.76262283	[428] lpme->szModule = INIWebCrypto.dll, ModuleName = C:₩Program Files₩INITECH₩INISAFE Web V6₩INIWebCrypto.dll
1939	26.76271248	[428]

toolhelp.cpp

```
// Loop through all other modules
while (TRUE)
{
    #ifdef _DEBUG
    DbgPrintf("lpme->szModule = %s, ModuleName = %s₩n", lpme->szModule, ModuleName);
    #endif
    if (!strcmpi(lpme->szModule, ModuleName)) { CloseHandle(hSnapshot); return TRUE; }
    if (!pfep->toolhelp.Module32Next(hSnapshot, lpme)) { CloseHandle(hSnapshot); return FALSE; };
}
```

[그림 109] toolhelp.cpp 디버깅으로 불일치 스트링 찾음

그림109 와 같이 toolhelp.cpp 를 디버깅해보니 FindImage_toolhelp 함수에서 lpme->szModule 과 ModuleName 이 서로 불일치하고 있었다. 즉, lpme->szModule 은 메모리에 로딩된 모듈의 이름이고 ModuleName 은 우리가 환경설정 파일에 입력한 디렉토리 경로까지 모두 포함된 모듈네임(우리가 후킹하려고 하는 DLL 파일이름과 경로)이기 때문에 발생되는 문제였다. 이를 strcmpi 함수로 비교하니 당연히 이미지를 못찾은 것으로 나올 수 밖에 없는 것이다. -_-; 다음과 같이 수정해서 이 문제를 해결하였다.

```
// ------------------------------------------------------------------
// Loop through loaded images to get the MODULEINFO32 you need.
//////////////////////////////////////////////////////////////////////
BOOL PRIVATE FindImage_toolhelp(LPSTR ModuleName, MODULEENTRY32* lpme)
{
    // Get a snapshot
    char *Module;

    HANDLE hSnapshot = pfep->toolhelp.CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, psi->pid);
    if ((int)hSnapshot == -1) return FALSE;

    lpme->dwSize=sizeof(MODULEENTRY32);

    // Get first module, this is needed for win9x/ME
    if (!pfep->toolhelp.Module32First(hSnapshot, lpme)) { CloseHandle(hSnapshot); return FALSE; };

    // Loop through all other modules
    while (TRUE)
    {
        int p=strlen(ModuleName);
        while (p)
        {
            if (ModuleName[p] == '₩₩')
                { Module = (ModuleName+p+1); p=0;}
            else
                p--;
        }

        #ifdef _DEBUG
        DbgPrintf("lpme->szModule = %s, Module = %s, ModuleName = %s₩n", lpme->szModule, Module, ModuleName);
        #endif
        if (!strcmpi(lpme->szModule, Module)) { CloseHandle(hSnapshot); return TRUE; }
        if (!pfep->toolhelp.Module32Next(hSnapshot, lpme)) { CloseHandle(hSnapshot); return FALSE; };
    }
}
```

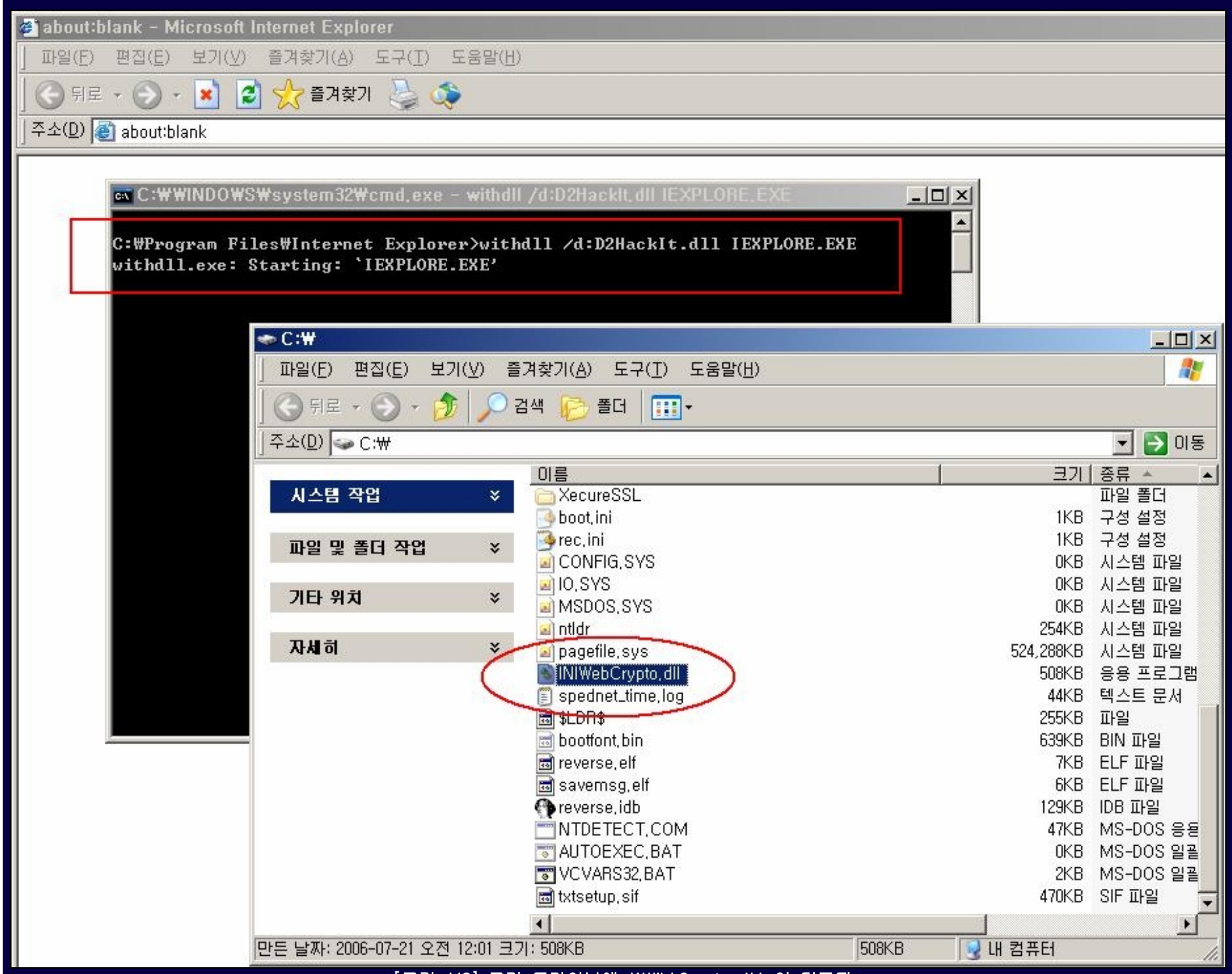
[그림 110] FindImage_toolhelp 함수를 이름만 비교하도록 수정함

위의 그림110 와 같이 FindImage_toolhelp 함수를 수정하여 GetBaseAddress 함수와 GetImageSize 함수가 정상적으로 작동하였다. 인터넷 익스플로어의 프로세스공간에 로딩된 INIWebCrypto.dll 이미지 주소를 정상적으로 뽑아내고 있었다. 그러나 쳡쳡산중이라고 문제가 모두 해결되진 않았다.
프로세스 익스플로어로 찍어봤을때 분명히 인터넷 익스플로어의 프로세스 공간에 INIWebCrypto.dll 이 로딩된 것을 확인했는데 자꾸 바이너리스트링을 못찾았다고 나오는 것이었다. 정말 못찾은 건지 아니면 GetFingerprint 함수가 필자에게 공갈을 치는건지 확인할 필요가 있었다. 그래서 언팩킹에 사용하려고 작성한 덤프루틴을 사용하여 메모리에 로딩되어있는 INIWebCrypto.dll 의 머리채를 잡고 밖으로 끄집어내기로 하였다. 즉, 메모리 이미지를 로컬드라이브로 덤프시키기로 결심했다. 다음의 화면은 필자가 로컬드라이브로 덤프하는 것을 보여준다.

```
    DbgPrintf("ProcessID = %dWn", psi->pid);

    // 메모리에 로딩된 INIWebCrypto.dll 모듈을 덤프하는 루틴
    HANDLE hFile;
    DWORD dwWritten;

    DWORD BaseAddress;
    DWORD ImageSize;

    // GetBaseAddress 와 GetImageSize 함수는 DbgHelp.dll 의 toolhelp 관련 함수로써 메모리의 프로세스 이미지를
    // 탐색하여 일치하는 모듈의 베이스어드레스와 이미지 사이즈를 리턴한다.
    BaseAddress = GetBaseAddress("C:WWProgram FilesWWINITECHWWINISAFE Web V6WWINIWebCrypto.dll");
    ImageSize   = GetImageSize("C:WWProgram FilesWWINITECHWWINISAFE Web V6WWINIWebCrypto.dll");

    // INIWebCrypto.dll 파일을 로컬 드라이브 C 에 생성하고 메모리에 로딩된 이미지 내용을 라이팅한다.
    hFile=CreateFile("C:WWINIWebCrypto.dll", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    WriteFile(hFile, (unsigned char *)BaseAddress, ImageSize, &dwWritten, NULL);

/*
    // 메모리의 바이너리 이미지(DLL)에서 Decrypt 함수지점 검색
    if (!GetFingerprint("D2HackIt", "Decrypt", psi->fps.Decrypt))
    {
        #ifdef _DEBUG
        DbgPrintf("Decrypt Not Found!");
        #endif

//      5 초 간격 핑거프린팅 쓰레드 생성
//      DWORD dummy = 0;
//      psi->TickShutDown = 0;
//      psi->TickThreadHandle = CreateThread(NULL, 0, FindTickThread, (void*)0, 0, &dummy);
//      #ifdef _DEBUG
//      DbgPrintf("CreateThread Complete..Wn");
//      #endif
//      psi->TickThreadActive = psi->TickThreadHandle!=NULL;

        return TRUE;
    }
```

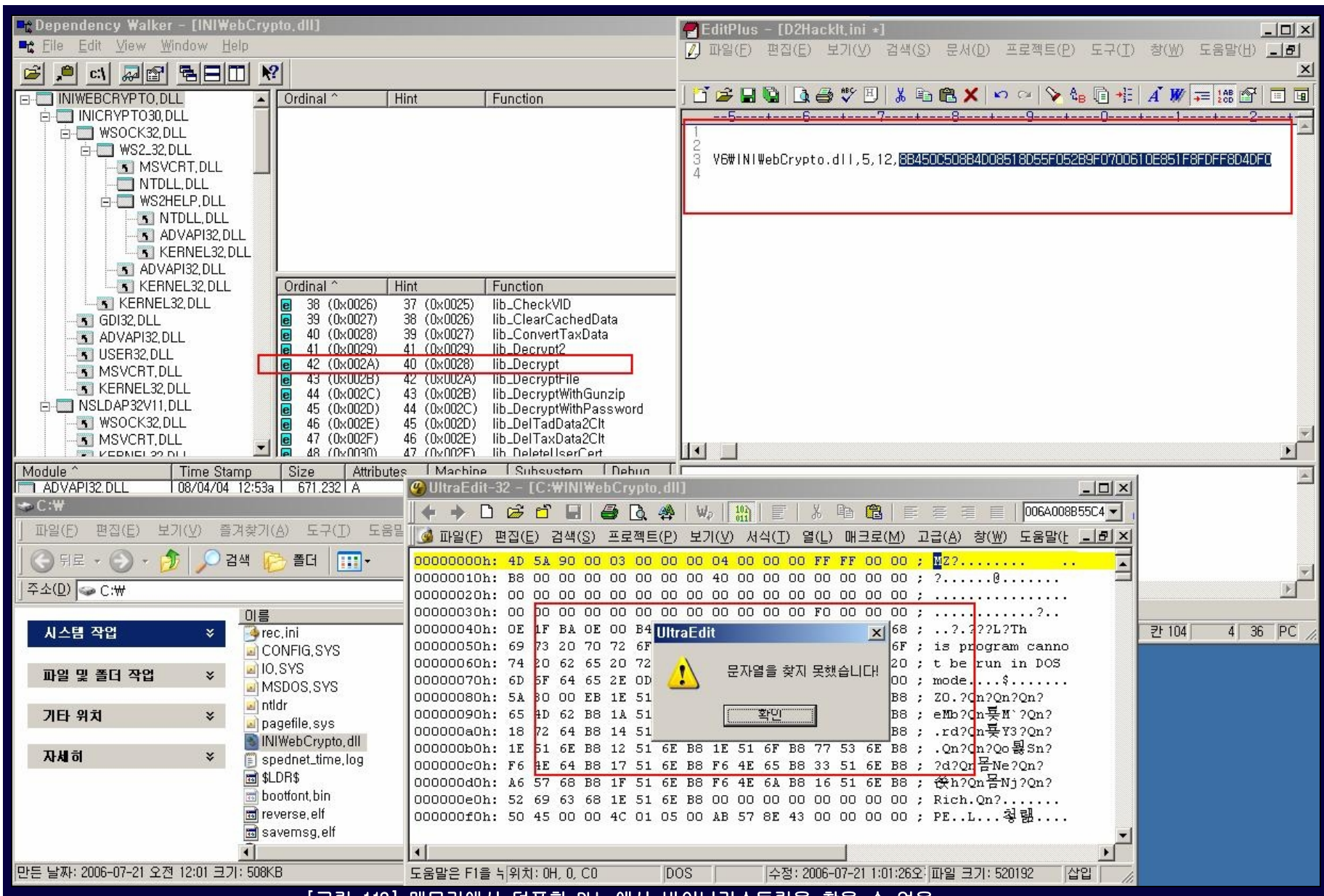
[그림 111] 언팩킹 용도로 쓰던 덤프루틴을 이용하여 메모리의 모듈을 하드로 끄집어내림

위 그림111 의 빨간박스안에 있는 코드를 ServerStartStop.cpp 파일의 ServerStart 루틴 마지막 부분에 집어넣고 GetFingerprint 함수와 Intercept 함수부분은 잠시 주석처리를 한다.(덤프만 하려함) 그리고 다음과 같이 withdll 로 D2HackIt.dll 을 넣고 인터넷 익스플로어를 실행하면 인터넷 익스플로어의 프로세스 공간에 로딩된 INIWebCrypto.dll 의 이미지를 로컬 드라이브로 덤프뜰 수 있다.



[그림 112] 로컬 드라이브에 INIWebCrypto.dll 이 덤프됨

위에서 보는 것과 같이 메모리의 INIWebCrypto.dll 모듈이 하드에 정상적으로(?) 덤프되었다. 정말 정상적인지 아닌지는 다음의 화면에서 Dependency Walker 라는 VC++ 의 유틸리티로 확인하여 정상인 것을 확인하였다. 자.. 여기서 메모리를 덤프한 이유는 오류가 발생한다고 한 지점이 타겟모듈에서 바이너리스트링을 못찾았기 때문이므로 직접 GetFingerprint 함수가 하는 역할을 수동으로 확인함으로써 문제점을 확인하려는 것이다. 그렇기 때문에 덤프한 INIWebCrypto.dll 파일을 울트라에디트 프로그램으로 직접 열어서 바이너리스트링이 찾아지는지 확인하였다.



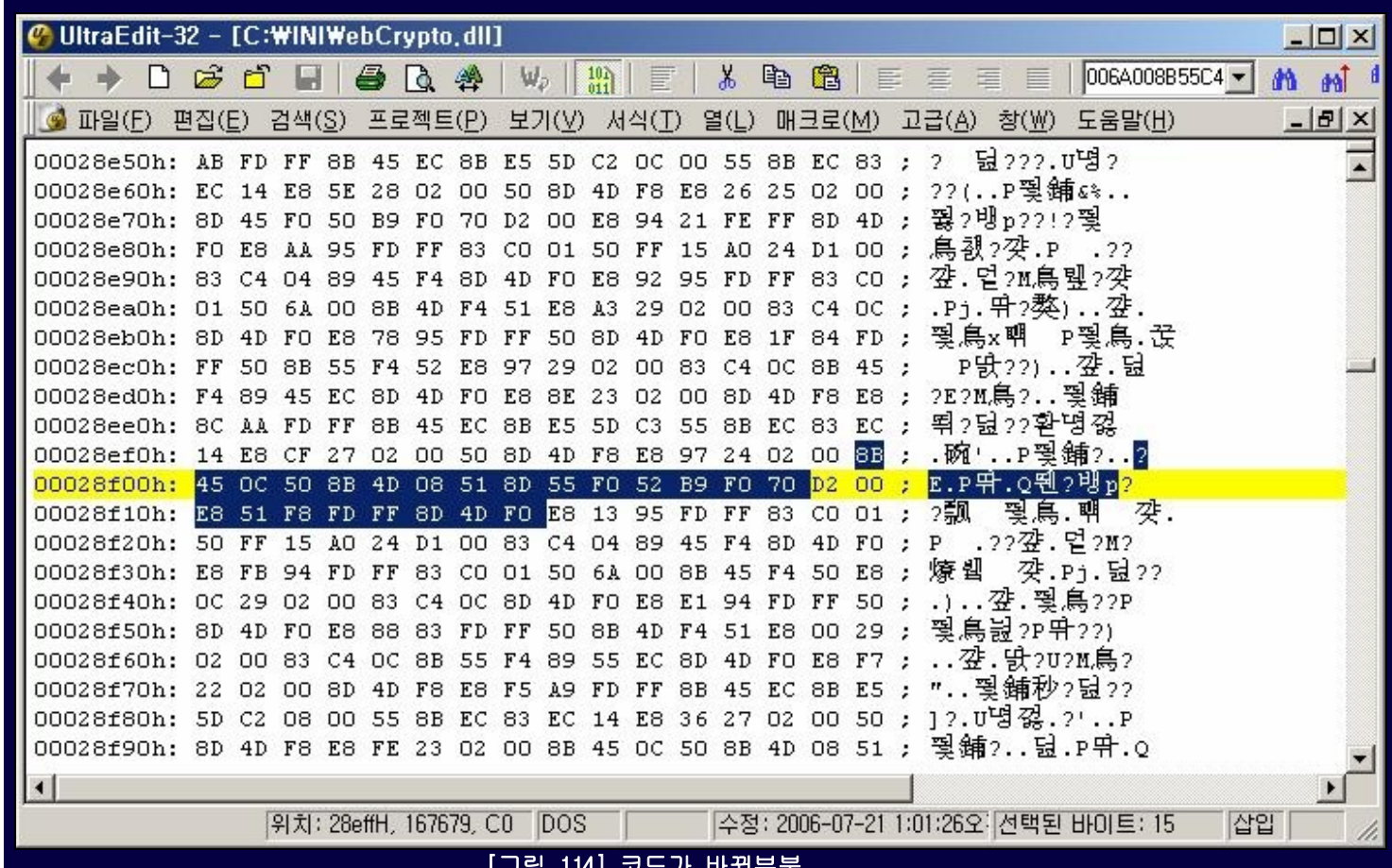
[그림 113] 메모리에서 덤프한 DLL 에서 바이너리스트링을 찾을 수 없음

헉.. 바이너리스트링을 찾지 못하는 것이었다. 이런경우에 상당히 착각하기 쉬운데 우리가 흔히 EXE 파일의 PE 포맷에 있는 베이스어드레스가 메모리 상의 베이스어드레스와 일치한다고 DLL 도 메모리에 그 주소 고대로 올라갈 것이라고 착각하는 것과 같은 이치다.

메모리에 로딩된 바이너리의 TEXT 영역 코드는 변화한다

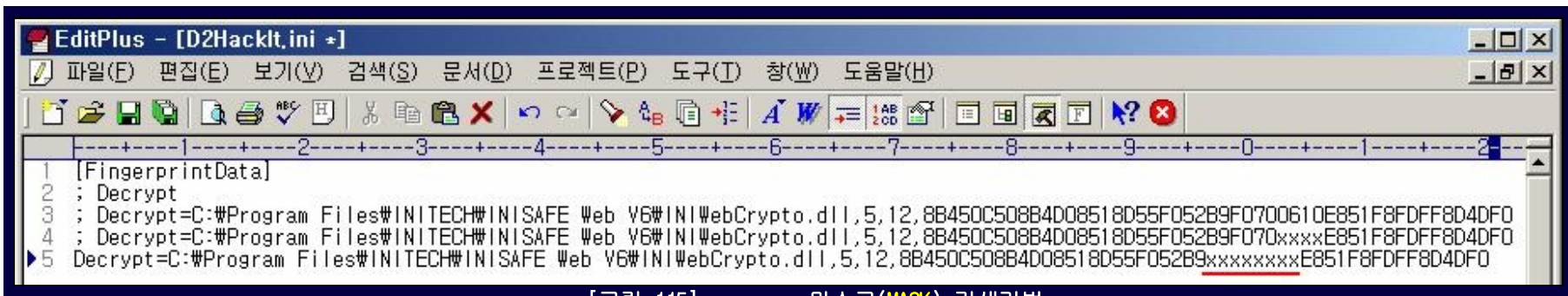
실행파일의 실행코드가 변경된다는 것은 아마 아는 사람은 잘 알고 있었을 테지만 알지 못하는 사람도 많을 것이다. 그 이유는 윈도우즈의 실행파일 로더가 메모리를 재배치하기 때문에 발생하게된다. 대부분인지 무조건인지는 정확히 알 수 없으나 DLL 의 경우 이런일이 종종 생기게된다. 즉, 위에서 바이너리스트링을 못찾은 이유는 코드영역의 프로그램 코드가 메모리 재배치의 영향을 받아 변경되었기 때문이다. 확인을 해볼까?

먼저 찾을 바이너리스트링의 일부분을 쪼개서 울트라에디트에서 찾기를 한다. 이때 1회 이상 일치되면 안된다.
유일한 지점이 발견될 때까지 바이너리스트링을 쪼개서 찾아야 한다.



[그림 114] 코드가 바뀐부분

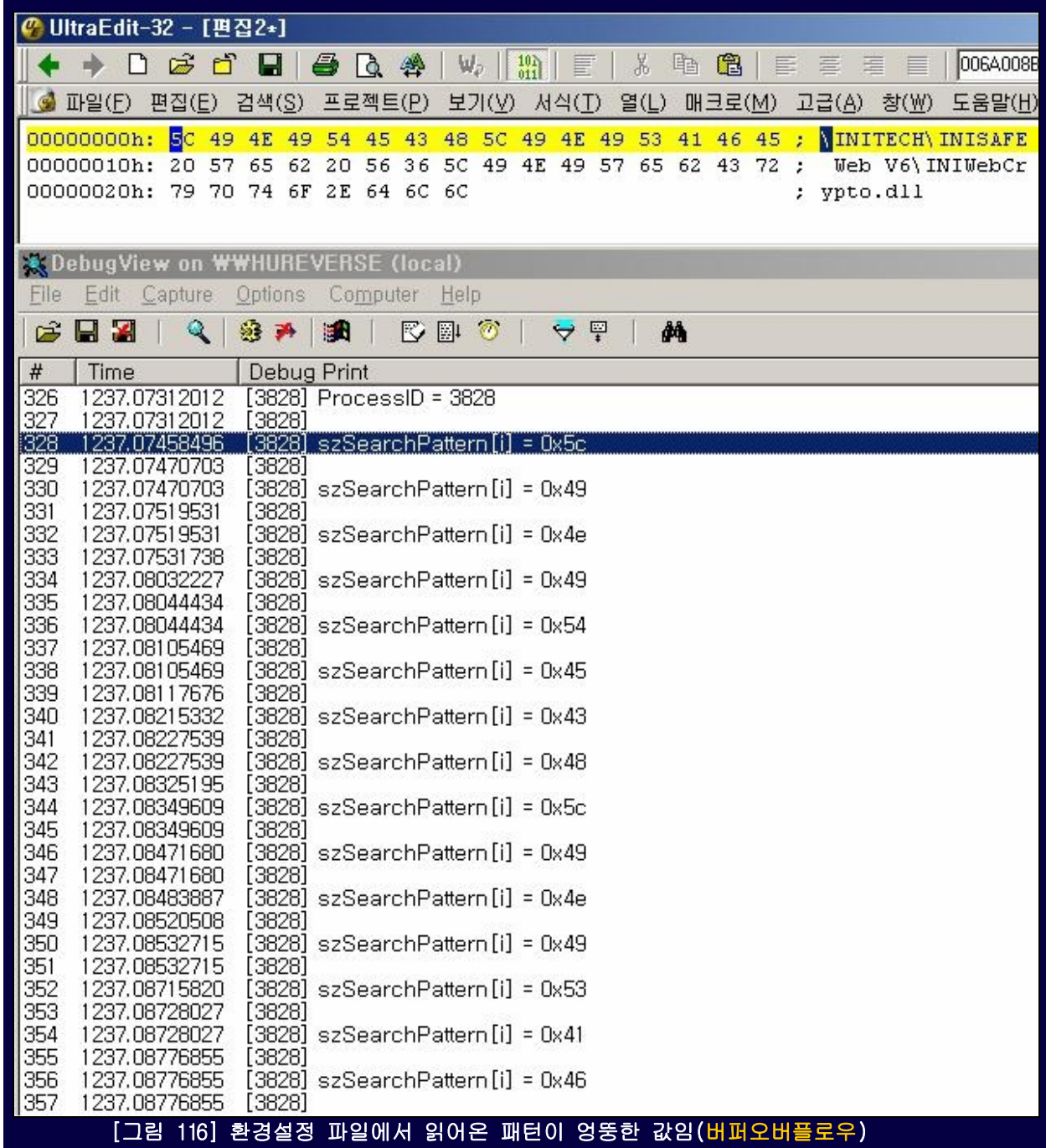
그림114 은 필자가 보기좋게 편집한건데 찾은 부분이 두개로 쪼개져 있는 것을 볼 수 있다. 이런 부분은 덤프한 파일에서 정확히 한 군데만 존재하였다. 그렇기 때문에 실행코드가 바뀐 것을 알 수 있다. 그림99 를 보라 이 변경된 코드는 mov ecx, offset unk_100670F0 명령부분에서 주소에 해당하며 OPCODE 로는 B9 F0 70 06 10 (주소는 거꾸로:LittleEndian)이다. 이 mov 명령에서 데이타 위치를 참조하는 주소의 일부분이 06 10 에서 D2 00 으로 바뀐 것이다. 그렇다면 DLL 이 로딩되는 주소가 동적으로 바뀌게 되면 이 주소도 동적으로 바뀌는가? 그렇다... 쩝.. 언제 어떻게 이 수치가 바뀔지 모르며 그 위치의 주소도 정확히 알 수 없다. 그렇다면 매번 덤프를 떠서 찾아야 하는가? 아니면 추가적으로 코딩을 해야 하는가?
Lesson-1 을 정독하였다면 GetFingerprint 함수가 울트라에디트보다 더 강력한 기능이 하나 있다고 언급한 필자의 말을 기억하고 있을 것이다...
그렇다.. 그 막강한 기능은 바로 이런데 써먹으라고 있는 기능이다.



[그림 115] xxxxxxxx 마스크(MASK) 검색기법

위와 같이 바이너리스트링을 지정할 때 xxxxxxxx 부분은 주소가 변경되는 부분이고 나머지는 고정된 부분의 스트링이다. 이것이 마스크(MASK) 검색기법이고 실제로 스타크래프트와 디아블로 시리즈로 유명한 블리자드사에서 패치를 내놓는 방식이 이 기법을 사용한 것이라고 D2HackIt 제작자들이 말하고 있다.
이제 메모리에 로딩된 INIWebCrypto.dll 이미지에서 제대로 바이너리 스트링을 찾겠지? ㅎㅎㅎ 산넘어 산이고 세상만사 쉬운일이 하나도 없다.
여전히 GetFingerprint 함수가 바이너리 스트링을 못찾았다고 DebugView 에 출력을 하고 있었다.

이제 버그는 다 찾은 것 같은데 마지막으로 테스트해도 여전히 바이너리스트링을 못찾고 Decrypt Not Found 만 DebugView 에 찍히는데 이유를 알 수가 없었다. 이런 경우가 있나... 덤프까지 떴는데 특정부분을 찾지 못하다니 도대체 인터넷 익스플로어의 프로세스 공간은 철통방어가 되어있는 것인가... 별의별 방법을 다 써서 프로세스 공간을 휘집어 댔고 덤프까지 떠서 일일이 확인까지 했는데 못찾는다니 말이 안되는 상황이었다. 심지어 DEP(데이타실행방지) 인지.. 원지 CPU 의 방어장치 때문에 그런건지 아니면 Administrator 가 아니라서 그런건지 할 수 있는 건 다 테스트해봐야 했다. 필자가 사용하는 윈도우가 XP HOME 에디션인데 홈에디션은 XP Pro 하고 달리 Administrator 로 로그인 할 수 없다. 안전모드가 아니면 아예 로그인이 안되도록 MS 에서 막아놨기 때문에 XP Pro 로 업그레이드 시키고 Administrator 계정으로 로그인을 하고 실행시켜보았다. 말그대로 개뻘짓이라는 짓은 다 해보았다.
결국 필자는 마지막이라는 생각으로 그림115 의 환경설정파일이 버퍼에 제대로 들어가는지 확인하기로 하였다. 바이너리스트링이나 한번 찍어서 DebugView 에 출력해서 보기로 한 것이다. 그러나 정말 어이없는 버그가 존재했다.



[그림 116] 환경설정 파일에서 읽어온 패턴이 엉뚱한 값임(버퍼오버플로우)

그림116 과 같이 어이없게도 엉뚱한 바이너리스트링이 로딩되고 있었다. 필자가 확인을 위해 울트라에디트로 각각의 값을 HEX 로 찍어보자 오른쪽에 디렉토리 경로와 INIWebCrypto.dll 이라는 것이 보였다. 이런.. 찾을 바이너리스트링에 엉뚱한 값이 들어가고 있었던 것이다. 처음에 필자는 D2HackIt 의 파싱오류로 인해서 생기는 문제인지 알고 해당되는 루틴을 모두 까뒤집어 봤지만 파싱의 오류는 아니었다. 그런데 패턴을 찍는 부분에서 자꾸 엉뚱한 값이 들어가는게 아닌가.. 그래서 DebugView 로 출력시키는 디버깅 구문의 바로 앞단에 주목했다. 그랬더니 버퍼오버플로우가 발생하는 것 같다는 느낌이 들었다. 왜냐면 DebugView 로 바이너리스트링을 찍어보는 구문 앞부분에 strcpy 로 모듈 이름을 복사하는 부분이 있었기 때문이다. 직접만든 소스를 사용하지 않고 D2HackIt 을 개조하는 경우이기 때문에 모든 코드를 신뢰했던 필자의 생각에 오류가 있었던 것이다. Structs.h 파일을 열어서 복사되는 곳의 버퍼가 얼마나 잡혀있는지 보고서야 버퍼오버플로우가 맞구나라는 확신을 갖을 수 있었다. 즉, D2HackIt 개발자들은 환경설정 파일의 모듈 이름부분에 디렉토리 경로가 들어가지 않는 다고 지네들만의 묵시적인 동의를 했었던 것이다. 결국, 필자가 환경설정파일에 디렉토리 경로까지 집어넣어서 버퍼오버플로우가 발생한 것이었다.

IniFilesHandlers.cpp 일부분

```
            case 0:
                fps.PatchSize=atoi(&szReturnString[i+1]);
                #ifdef _DEBUG
                DbgPrintf("fps.PatchSize = 0x%x₩n", fps.PatchSize);
                #endif
                break;
            }
        }
    }
    strcpy(fps.ModuleName, szReturnString);
    #ifdef _DEBUG
    DbgPrintf("fps.ModuleName = %s₩n", fps.ModuleName);
    #endif
    delete szReturnString;
```

IniFileHandlers.cpp 파일의 GetFingerprint 함수의 strcpy 에서 버퍼오버플로우 발생

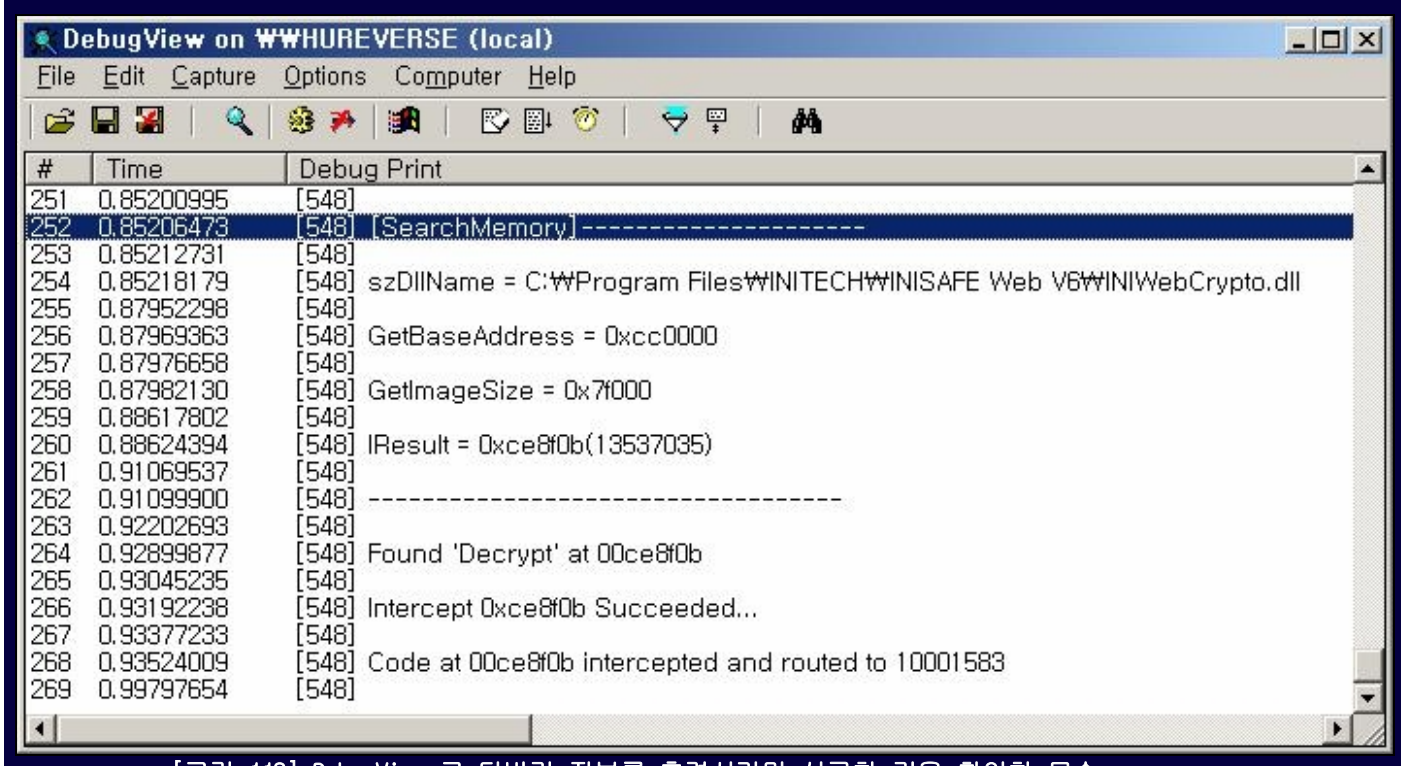
Structs.h 헤더파일의 일부분

```
#define MAX_FPS_NAME_LEN        0x40
#define MAX_FPS_MODULENAME_LEN  0x10
#define MAX_FPS_FINGERPRINT_LEN 0x80
typedef struct fingerprintstruct_t
{
    char        Name[MAX_FPS_NAME_LEN];
    char        ModuleName[MAX_FPS_MODULENAME_LEN];
    char        FingerPrint[MAX_FPS_FINGERPRINT_LEN];
    DWORD       Offset;
    DWORD       PatchSize;
    DWORD       AddressFound;
} FINGERPRINTSTRUCT;
```

디렉토리 경로 최대값인 0xFF 로 바꿔줌

[그림 117] D2HackIt 버퍼오버플로우 발생지점 BugFix 하기

이제 버그는 다 잡았다. 다시 withdll 로 인터넷익스플로어를 실행시키면 INIWebCrypto.dll 이 후킹되어 우리의 함수로 라우팅되는 것을 볼 수 있다.



[그림 118] DebugView 로 디버깅 정보를 출력시키며 성공한 것을 확인한 모습

위와 같이 0x00CE8F0B 의 Decrypt 루틴이 우리가 만든 후킹루틴인 0x10001583 로 라우팅되었다. 참고로 후킹소스에서 INIWebCrypto.dll 등등의 DLL 들을 강제로 로딩하면 그 프로세스 공간(인터넷 익스플로어의 프로세스 공간)에서는 절대로 다시 로딩되지 않는다. 그래서 D2HackIt 이 인터넷 익스플로어로 들어갈 시에 강제로 미리 로딩시키는 것이다. 이렇게 강제로딩으로 패치를 하게되면 인터넷 익스플로어에서 INIWebCrypto.dll 을 사용하려고 하는 놈들은 모두 D2HackIt 이 걸어놓은 후킹루틴으로 빠져들게 된다. DLL 의 의미를 되새겨 보라.. 공유 라이브러리 아닌가.. 만약 이런 방식의 후킹을 방어하려고 한다면 어떻게 될까? 해보지는 않았지만 대충 생각해보면 DLL 은 말 그대로 공유해서 쓰겠다는 의미이기 때문에 D2HackIt 이 DLL 을 잡고 있다면 이 공유법칙을 깨고 FreeLibrary 로 DLL 을 해제하려는 트릭을 쓰려는 후킹방어자들의 COM 모듈들이 모두 경고를 하나씩 먹게 될 것이다..(그럴까.. 잘 모르겠으나 참조카운트가 없어야 해제가 되기 때문에 이런 판단을 해본다..)

다음은 테스트를 해보는 것이다. 정말로 Decrypt 루틴을 사용할때 후킹이 되는지 그게 중요하니까..

```
 2  <HTML>
 3  <HEAD>
 4  <TITLE> New Document </TITLE>
 5  <META NAME="Generator" CONTENT="EditPlus">
 6  <META NAME="Author" CONTENT="">
 7  <META NAME="Keywords" CONTENT="">
 8  <META NAME="Description" CONTENT="">
 9  </HEAD>
10
11  <BODY>
12  <object name="test" classid="clsid:286A75C3-11FB-4FB4-AC4A-4DD1B0750050"></object>
13
14  <script>
15  document.all.test.Decrypt("1","01020303");
16  document.all.test.About();
17  </script>
18
19  </BODY>
20  </HTML>
```

아무거나 넣음 우리는 여기에 모가 넘어가는지 몰라서 후킹을 한 것이니까..(OLE/COM 뷰어로 프로토타입을 참고해서 호출인수를 준다)

[그림 119] Decrypt 함수의 후킹을 테스트할 자바스크립트 작성

위와 같이 앞서 About 함수를 호출했던 소스에서 Decrypt 를 호출하는 코드를 한줄 추가해준다. Decrypt 함수에 넘어가는 인수는 그림88 을 보면 프로토타입이 나와있다. BSTR 은 ATL 프로그래밍을 해본 사람은 알겠지만 문자열 타입이다. 즉, 인수로써 두개의 문자열을 필요로 한다. 더이상은 알 필요가 없다. 왜냐면 이 인수로 넘어가는 놈을 실시간으로 보고 제어하고자 후킹을 한 것인데 이 인수가 실제로 어떤 값을 가져야 하는지 미리 알아야할 필요가 없으니까 말이다..
withdll 로 D2HackIt 을 주입한 인터넷 익스플로어를 열어서 test.html 파일을 열어보면 DebugView 에 다음과 같이 후킹한 내용이 찍히게 된다.

DebugView on WWHUREVERSE (local)

File Edit Capture Options Computer Help

#	Time	Debug Print
529	1688.46667480	[3172]
530	1688.46667480	[3172] szDllName = C:₩Program Files₩INITECH₩INISAFE Web V6₩INIWebCrypto.dll
531	1688.46667480	[3172]
532	1688.46679688	[3172] GetBaseAddress = 0xcc0000
533	1688.46679688	[3172]
534	1688.46691895	[3172] GetImageSize = 0x7f000
535	1688.46691895	[3172]
536	1688.47241211	[3172] lResult = 0xce8f0b(13537035)
537	1688.47241211	[3172]
538	1688.47253418	[3172] ----------
539	1688.47253418	[3172]
540	1688.47265625	[3172] Found 'Decrypt' at 00ce8f0b
541	1688.47265625	[3172]
542	1688.47265625	[3172] Intercept 0xce8f0b Succeeded...
543	1688.47277832	[3172]
544	1688.47277832	[3172] Code at 00ce8f0b intercepted and routed to 10001583
545	1688.47290039	[3172]
546	1703.95349121	[3172] Decrypt = 3715176, 01020303
547	1703.95349121	[3172]

[그림 120] Decrypt 함수를 후킹한 내용이 출력됨

오케이!
후킹이 성공한 것이다. 아예 내친김에 이 상태로 사이트 몇군데 더 돌아다녀보면 다음과 같이 재밌는 데이타가 출력된다.

DebugView on WWHUREVERSE (local)

File Edit Capture Options Computer Help

#	Time	Debug Print
2	6.75928354	[2036] Decrypt = 3715976, 11Pt5rTZy4HounW5WfkBdVaqgnYfvA6CUyBh9AjI+nwe+wcvDXS1qe6C/1s8sFsV
3	6.75928354	IVWIGAZpDsdUnMc9YgVuAg==
4	6.75928354	[2036]
5	6.75932837	[2036]
6	6.91520643	[2036] Decrypt = 3716136, Qo5r95jQJrLQokRNYO+Vog==
7	6.91520643	[2036]
8	6.91529322	[2036]
9	6.91774940	[2036] Decrypt = 3716136, uqtRyWIGPuFvStgAJIsaODTvUCo30Pcem802M/BgJEz1JRMS9BHBorRRHoqSNhCa
10	6.91774940	[2036] 5Y0s2vZsvlbfcgW/t2+NFG3UK5BAvhPi4zDZcK8ybYKI2a5eB4EFaAHQDDHApPx8
11	6.91774940	[2036] nrD4UiwWxpJzdSUF7cip2NLJzyDZ8/Ihmk13CicLK9RU5bC4iYO9OjNjdb+FdJjL
12	6.91774940	[2036] XQwJjgUozvFr54GKcts1CzsJPs4hGWzzrWyH0VD/kMfkEpdswYrNjK7ew5w6yR0d
13	6.91774940	[2036] Byctak4vsWdL88UcfWeSUkAb+AX5OmDUHVDHkREEfR7uZmg+R7APxU5CgKU+I93a
14	6.91774940	[2036] 7tBMk4Q2l4oz6q6AktP2uU2RITkXg+7+Fot6aj/D+R+PvlqI7vy4O/Epd2hwL8Mm
15	6.91774940	[2036] BbVLUzyFwbexQjIlYRfVlp+SCG08WzJMPVdrEwykmtHUSoFeTOBMXtfnwJAzoQUM
16	6.91774940	[2036] sh9RpfBYrxACSn7LavStVqoUMuwtAS1LfdZXWYAkv8a9eC34RLKr7v0HvAtUTOp2
17	6.91774940	[2036] CB+jbgzt+sFr1wF5mRKL0XEYcKPhh83AXB7kr9ScYy5FWY1ycN4N7blZy2RTOyUW
18	6.91774940	[2036] YEv+9BCFP+IYZ2Q4Bbp1L0/WY9rLTsWBNgTKE8zRjAi6h63ew+jdp+B9fqrSv2il
19	6.91774940	[2036] bwwArjloD1xfEOA/i80DkUHFJhkhlzw27np7r8xElhFKoxF5DSb0O2UyWO6YFgUD
20	6.91774940	[2036] eKd2J6YtcQYM26wDNFDtDfo7HEtVBQatgr2QQYL96oM4VzzxeuA9/JnZFEXK/rff
21	6.91774940	[2036] LQeggUp+Zqzw+A2cdEBQgTXct5gH+Y6tVmZb6W8Lah3zSfD88HSXmTjaZ5q0Al5U
22	6.91774940	[2036] axzKOlmaouD3M2ZG4lg410weOfkP48/beOCz9/pfMP2gJ+HNrfkoMre+2MearV1r
23	6.91774940	[2036] usAMD84pOlVWqQJMVCs58+sj7CBzn0veXV8l/yRgD2kYO1p1LyvGqDRYNK2LQLTL
24	6.91774940	[2036] fKN5m8aFSC5hQQMizFDr8FdqAxDaqr
25	6.91790628	[2036]
26	6.93023062	[2036] Decrypt = 3716136, 1JAXXw3FsvXyQNLIDtsITrgo57zrsUi/5y4dDZ3h4Tg=
27	6.93023062	[2036]
28	6.93031549	[2036]
29	6.93319607	[2036] Decrypt = 3715576, RZwWX0GNoFXs/Q1W4qIXW4fwQwsxuijpUVNYCKQkvY8NIOV63Ht5fUL3A2Vmx4M0
30	6.93319607	[2036] Pf7aCGqibmQuH3WsKF7wdyUOw7Ur/LazayZJ32igAOzdcKqcEsW84wcJZQfInyYr
31	6.93319607	[2036] VeoRvnea1mFOPfyfU3loSLVvzbyiRlvZcEiR/6nqVcXR+Y5xippk0nQ2u1EY6tMk
32	6.93319607	[2036] /5T/XpWX+ff9lbnm77Y+1qlmlHa6yYUexz+miDAqOkADxy7d5PK0+5mcBdboTDMb
33	6.93319607	[2036] 4Oav/7RMAfqfl5Ys+545xTXmrLJLoV2sC8XozJqD71N+sHRoOnt5jrd+vPhhtrZT
34	6.93319607	[2036] aJ7ynQBkbRsxt65Pl4mRxGBOyivN1Mxi5XvWz/Dp4KwEFBQp8wGtzpy70giSBArE
35	6.93319607	[2036] UsmTtP2+zw9REPV07Zswv3O3weMVbEmVH2NyDKpgdlvDWq5YALfpTgELI0SnTMMl
36	6.93319607	[2036] N5MbsMcH1a0v2rAX5s0dGcrHzt67DPMs5iXZQf3T+MuLSwI2uoHA3aYN2TDbjlmB
37	6.93319607	[2036] SJmehQ1gWTF5I5hA0tobyLI148c7gpno5egxE5G2pgLBHgCVsxCSun605tW7k9A1
38	6.93319607	[2036] J8D7syRT4puZvoi0gr7GF4HPD0YCDa5cmyxu1OmgPKAhSoj26YIHJlNCVYUAz/xs
39	6.93319607	[2036] jA5dfHFgxOnIPf2Gu7Vp/ljLxfyIwlWhFAwlysBSMXwsCirnqTj02wT+oDtfH6uG
40	6.93319607	[2036] SpKaFNO17+E4ygC0qOEmMumkrRtpUHBwGcgmZGdf/7oecZC4r/7w/csWlPu0A8zh
41	6.93319607	[2036]

[그림 121] Decrypt 함수를 사용하는 놈들의 암호화 통신들

그런데 그림121 에서 뭔가 이상한게 있다. 독자들이 눈치를 챘을지 모르겠다. 수치!!
항상 역공학 전에 정공학이 있어야 한다. 필자가 OpenSSL 을 분석해본 경험에 의하면 암호화 방식을 넘길때는 그냥 텍스트로 넘겼었다.
저런 큰 수치는 본적이 없었던 걸로 기억한다. 그렇다면 저 수치는 분명히 주소일 것이다. Lesson-1 에서 필자가 설명하던 내용을 기억하는가?
함수를 후킹할때 넘어가는 인자가 그냥 일반 수치값이라면 이런 경우는 극히 적으니까 그냥 포인터로 생각하고 다루라고 했던 점 말이다.
(IDA Pro 라는 역어셈블러를 너무 믿으면 이런 속임수에 빠져든다. IDA Pro 는 어셈블러로써 뛰어나지만 보완할 점이 많은 툴이다.)
앞에서 그림98 의 arg_0 가 int 값이 아니라 포인터라는 것을 이미 리버싱할 때 눈치챘어야 한다. "이거 포인터 같은데.." 라는 느낌이 들었다면 다행이다..
여기서 자세하게 오류를 겪는 것까지 다 설명하는 것은 필자의 감으로만 설명하면 독자들은 아무것도 배울 수 없다. 누구나 다 마찬가지 겠지만 오류의 과정이 없이 결과가 만들어지지는 않기 때문이다. 이 문서를 읽고 독자가 다른 프로그램을 대상으로 스스로 테스트를 진행해도 아마 오류를 피해갈 수 없을 것이다.

수치를 포인터로 다루도록 수정함

다음과 같이 소스를 수정해서 DebugView 에 출력하는 부분의 첫번째 인자가 문자열이 되도록 한다.

```
////////////////////////////////////////////////////////////
// Decrypt()
// ciphername = 암호화 방식
// data = 디코딩 데이타
////////////////////////////////////////////////////////////
void DbgPrintf(LPTSTR fmt, ...);

void __fastcall Decrypt(char *ciphername, char *data)
{
    DbgPrintf("Decrypt = %s, %sWn", ciphername, data);
}
```

[그림 122] 출력루틴 변경

빨간 밑줄이 그어진 부분이 수정된 곳이다. int 를 char * 출력으로 바꾼 것이다. 이미 앞에서 다 설명했지만 노파심에서 얘기하면 여기서 수정하면 프로토타입을 D2HackIt.h 파일에서 수정해야 한다. 자 이제 그럼 제대로 출력이 될 것인가?

그림121 의 데이타는 잘못된 암호화 방식이 출력되었기 때문에 다시한번 사이트를 돌아다니며 모아보자...

DebugView on WWHUREVERSE (local)

File Edit Capture Options Computer Help

#	Time	Debug Print
9	0.00250451	[3196] fGy8awX1OUUNFPoBZ0kDZIsGubMOVwzGXjb4DWXmtmRKwYmmUijHSHQto0vOKXgo
10	0.00250451	[3196] wc2nMvrZXOTZ5MwArqFsU87hlGlzB3LT7D3lh17CDfbk8e/Y7ltue1tP0z/6d4QR
11	0.00250451	[3196] 2kqyAkjCCsdYlDfBqgDmv2laiKpP9ClGp1d3e/jW0aa62EQllhzZRzuKOky7hXKr
12	0.00250451	[3196] pX+rw0vmW7/B/Q6dMjrodxnFfgCKlHS3o8l10H7a9rPVmh5MtMXSACjBqKiJU14d
13	0.00250451	[3196] dbWg/5vesG97ZuS48FXgldnaZFsyUULu6eiS496lSTFHblOZ1MyqcY245s9cupc+
14	0.00250451	[3196] cqnysNjzk5gAxd0blxuDEoBx6G/oGnkDQK0MVOhCPMQonGnWt80TDgPtAG6YG3Nl
15	0.00250451	[3196] eAJEqvJFyD1AVKTrLT30bDwm/Nqplh7hZSPM10J47OPF3aEi1QvacSDkDhEYEmXo
16	0.00250451	[3196] 0WPdiNfyCEpBzMjoUcf3XRMveVnhMnrUxsvtQrlz/g9AGw7fr5aw8eXPccyfCkL9
17	0.00250451	[3196] jEDOmEhuJ49FnDa/zr2z1+1gKutw9hYjVBxV+JUTBxBtb8Op3dmTg/ED6WA3s3NY
18	0.00250451	[3196] sQDmd/gRL3Yr13BRnycrmJvX8MeOY
19	0.00266123	[3196]
20	0.02077108	[3196] Decrypt = SEED-CBC, gwT2pDe9mW+kf2/ZqiCp77xatmWsSnvDXUnArHQlmtw=
21	0.02077108	[3196]
22	0.02085516	[3196]
23	0.02434248	[3196] Decrypt = SEED-CBC, Z2H4m3VBWDM+E54l6/ZyhSH9No9qSfhzRKR/AJmbgvv+zd74OLfkalwSfc/kyX6+
24	0.02434248	[3196] QW8z0Byu6oPLdXpjcTuV9mt/GsvCsCLw+T3HvaLqZt4NPMy5t2YvhhGYkTEr+3qD
25	0.02434248	[3196] akN8g/alGoE3zJGQRi1PoUy8RQn+uwqnYKGXONVuvcmmxfqtW0pOF4yG9oQkSjw1
26	0.02434248	[3196] pgxGyWTlctt5t9hoQBGpD5yXHOJooVJzKMDZsDPt20PjuunOd03vLb72h8ScPTk+
27	0.02434248	[3196] nQiKA9yrln+EO8584EKxk/VZwqnu43N4S3ecUw3PyKplKZwuALQsMQ30RZ0nE1LX
28	0.02434248	[3196] hiRBF628zWWj8no4KwLboO3Ta3ml/dOsjSCNjZxZ5oGNeBR+2i7v4cPhb3rhal6x
29	0.02434248	[3196] ScPH9O55CkwHHb7yQ/dFyeserGgjW6JEvnK9rP62dv7c2VdYnrpX209EWv4Cyqrt
30	0.02434248	[3196] 9Qmk2jJHWeRCuQ4nJ+PFc3Bb9REBKLnRTc7BkTQx7xpdm1itLxEuvqAZbHZLS/lw
31	0.02434248	[3196] zu1d9FJ16udyAjx/c1iq5Wzhf+0Su3pwQ3aTUhyocCHJBf2KOMikBRFrxFm+230A
32	0.02434248	[3196] fLFvpUKEMhtCoMY4S9fZm4yZbTrS23Zo6FvzA1txA/q6DQxhAAOEpec//SlmLLOP
33	0.02434248	[3196] ZhYOoalU+2Q1ZVhyTpnoCVcG/MMtU7KfOs65hHGQpn9878k8BypgUseoYg5E/OwJ
34	0.02434248	[3196] ez6tlq/l4UH3B6k29q6SyYVuNuSBJSMGLEmxq7ltNV+SVOVJb6SfMgSuLwwwlcSv
35	0.02434248	[3196]
36	0.02448160	[3196]
37	0.03057512	[3196] Decrypt = SEED-CBC, PsVzHlKxHwbnLaCqKeQ12A==
38	0.03057512	[3196]
39	0.03112658	[3196]
40	0.04248920	[3196] Decrypt = SEED-CBC, gwT2pDe9mW+kf2/ZqiCp77xatmWsSnvDXUnArHQlmtw=
41	0.04248920	[3196]
42	0.04257469	[3196]
43	0.06483506	[3196] Decrypt = SEED-CBC, RzyU3EUSX9V8J50arw6UVQ==
44	0.06483506	[3196]
45	0.06505883	[3196]
46	0.06712837	[3196] Decrypt = SEED-CBC, 2cXcyrwDZJxezL8qJqQlRA==
47	0.06712837	[3196]
48	0.06719737	[3196]
49	0.17671241	[3196] Decrypt = SEED-CBC, OuZ/hnj8pv38ougfcy8xTCdhRtt24ZAlsDqCVVo2lB0MXRxcTPWcSWvG6ot5DwSZ
50	0.17671241	[3196] onK8MYURKPwBNyKlrZ+4RQ==
51	0.17671241	[3196]
52	0.17680068	[3196]

[그림 123] 키사가 만든 SEED-CBC 방식을 사용

보는 것처럼 필자가 예상했던 대로 SEED-CBC 방식으로 암호화 된 내용들이었다. 첫번째 인자는 SEED-CBC 라는 문자열이 담긴 주소였고 수치가 아니었다. 이렇게 해서 Decrypt 함수를 후킹하는 것을 예로 보였다. 이제 Encrypt 함수도 Decrypt 함수와 구조하나 안틀리고 똑같기 때문에 후킹루틴을 만들 수 있을 것이다. 여기서 Encrypt 함수는 다루지 않는다. 그 이유는 민감한 정보가 노출되는 것을 원하지 않는 사람들도 있을 테니까...
그것은 독자의 몫이다.

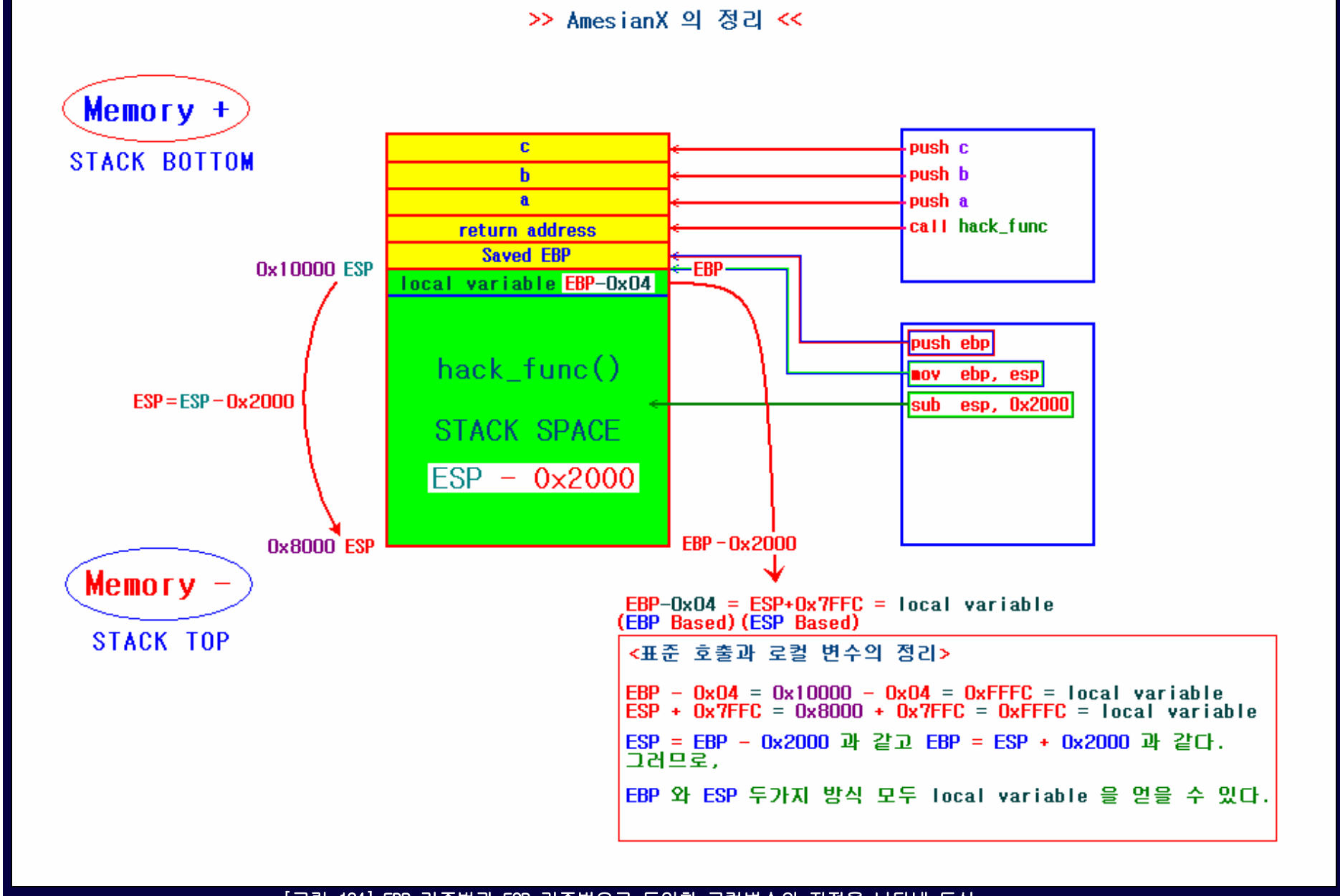
P.S: 앞에서 DLL 에 주입하다가 실패한 부분을 설명해 놓았었는데 문제점을 모두 해결한 시점에서는 DLL 에도 주입할 수 있다. 즉, 인터넷 익스플로어를 withdll 로 작동시킬 필요 없이 setdll 을 이용하여 DLL 에 D2HackIt.dll 을 박아버리면 다음부터는 인터넷 익스플로어만 작동시켜도 알아서 후킹 내용이 DebugView 로 출력될 것이다. 물론, 환경설정파일의 경로부분을 조금 수정해야 할 것이다..

이제 여러분은 대부분의 윈도우 프로그램을 소스코드 없이 제어할 수 있는 가장 기본적이고 정석적인 방법을 살펴보았다.
이 방법을 통하여 VC++ 로 제작된 프로그램 뿐만아니라 Borland C++ Builder, Delphi, VB 등등 기타 언어로 제작된 PE 구조의 실행가능한 포맷은 모두 후킹할 수 있다.
반복 연습으로 능숙한 후킹이 가능해지면 플러그인 제작에 꼭 도전해 보기 바란다. 윈도우 후킹에 자신감이 생기면 소스를 응용하여 타 플랫폼에 적용할 수 있는 연구를 해보는 것도 실력향상에 도움이 될 것이다. 이 후킹 방식은 플랫폼에 국한되지 않는 방법이기 때문이다. 연구를 거듭 할수록 무한한 가능성이 열릴 것이다.
이 문서가 여러분의 리버스 엔지니어링 프로젝트에서 많은 도움이 되기를 바란다.

감사합니다

- Art of Hooking by AmesianX in PowerHacker.net -

별책부록. 0x01 - EBP 기준법 / ESP 기준법



[그림 124] EBP 기준법과 ESP 기준법으로 동일한 로컬변수의 지정을 나타낸 도식

사설..

[필자의 COM 모듈에 대한 사견이므로 딴지걸지 않을 분만읽어주시길.. - 일반 DLL 과 COM DLL 의 개념] ——

자 혹시 프로그래밍을 잘 모르는 사람이라던지 개념을 약간 비뚤게 잡고있는 사람이 있을까봐 부가설명을 좀 하자면 지금까지 한 작업과 앞으로 하게 될 작업은 절대 불법이 아니라는 점이다. 그 이유를 설명하면 COM 모듈(컴포넌트:델파이의 컴포넌트가 COM 이다)로 프로그램을 만드는 목적이나 이유 자체는 아무나 가져다 쓰는 것을 전제로 한다. 즉, INISAFEWeb60.dll 은 아무나 호출해서 쓰도록 만든 파일이라는 소리다. 그렇기 때문에 필자의 PC 에 지가 알아서 등록을 해버리는 것이다. 무슨 소린지 이해가 안가는가? 그렇다면 당신은 일반적인 DLL 과 COM 의 DLL 개념을 알지 못하는 것이다.

우리가 프로그래밍 할 때 일반적인 DLL 의 함수를 사용하려면 DLL 안에 있는 함수들의 프로토타입이라는 것을 알아야만 가능하다. 이 프로토타입이란 우리가 아주 일반적으로 접하는 example.h 와 같은 헤더파일에 정의되는 프로그래밍 선언을 뜻한다. 주로 프로그래밍 소스에 아주아주 기본적으로 들어있다는 것을 이미 알 것이다. 그 헤더파일이란 것이 없다면 DLL 에 들어있는 함수들을 호출하는 것 자체가 이론적으로 불가능하다. 생각을 해보라 여러분이 프로그램을 개발했고 그 프로그램을 판매한다고 치자. 그렇다면 여러분은 제품에 프로그램의 개발소스를 넣겠는가? 안 넣을 것이다. 그 이유가 무엇인가? 소스를 넣게되면 제품을 산 사람이 그 소스를 변경하고 재컴파일해서 제품을 다시 만들 수 있다는 소리가 된다. 그래서 개발사는 제품의 소스가 없는 일명 실행파일(EXE 및 DLL 들)들과 리소스라고 불리는 데이타파일로 이루어진 패키지를 판매하게 된다는 것이다. 여기서 중요한 점이 거의 모든 개발사는 핵심 그래픽엔진, 암호화엔진, 네트워크엔진, 메신저엔진 등등과 같이 자신들의 핵심루틴들을 DLL(동적라이브러리 - 메모리에 붙여서 함수를 호출할 수 있는 개념)에 담는다는 것이다. 사용자는 이 회사 제품의 DLL 파일만 있으면 이러한 핵심기능들(그래픽, 암호화, 네트워크, 메신저같은 함수들)을 호출해서 자신이 개발하고 있는 프로그램에 사용할 수 있을까? 필자는 이러한 연구를 두번 진행했었다. 한번은 2001년경 개인적 연구를 했었고 다른 한번은 그로부터 몇년이 지나 2006년경 재직당시 리버싱 프로젝트를 진행할때였다. 이미 2001년에 실패한 경험이 있었기 때문에 불가능하다는 개인적인 결론을 알고 있었지만 혹시나 하는 마음에 2006년 연구당시 아주 심도있게 접근했지만 결국 해결하지 못하고 영원한 연구과제로 남겨뒀다.(이 방법이 나온다면 아마 세상이 발칵 뒤집힐 만한 파급효과를 가져올 것이다. 여태까지 나온 모든 윈도우즈 프로그램의 DLL 을 마음대로 이용할 수 있기 때문이다. 그러나 불가능한 건지 해외의 해커들도 이 분야의 연구결과를 시원히 제공하지 못하고 있다.) 이쯤 설명하면 왜 그걸 호출 못하냐고 반박하는 사람들이 생길것 같아서 추가적으로 설명을 하면 일반 "C" 익스포트(EXPORT) 방식의 함수들은 아무리 상용제품의 DLL 에 있다고 해도 호출할 수 있다. 왜냐면 이건 표준이라서 리버싱으로 헤더를 만들 수 있기 때문이다. 그러나 MS 윈도우즈 기반으로 나오는 상용제품들이 함수만 익스포트 하겠는가? 대부분의 핵심 DLL 들을 열어보면 함수만 익스포트 하는 것이 아니라 아예 클래스(CLASS)를 통째로 익스포트 한다. MS 는 비표준(?)으로 되어있는 확장 DLL 이라는 개념을 지원해서 클래스 익스포트를 지원하기 때문에 사실상 헤더가 없이는 DLL 에 있는 함수들을 호출할 수 없다는 것이다. 만약 이런 DLL 안에 있는 함수들(정확히는 클래스의 멤버함수들)을 호출하려면 클래스의 인스턴스가 필요하다. 집 클래스가 정의되어있는 헤더가 설계도 역할을 한다면 이 설계도를 갖고 집을 지으면 그 지은 집이 인스턴스가 된다. 즉, 메모리에 할당된 인스턴스란 놈이 없으면 멤버함수 호출시 메모리 접근오류를 범하게 된다. 이 경우만 있는 것이 아니다. 클래스의 멤버함수들은 다른 클래스의 인스턴스를 참조하는 경우가 굉장히 많다. 예를들어 집을 짓는데 집안에는 화장실이 있다고치자. 먼저 집 클래스로 집만 짓고 화장실 클래스로 화장실은 만들지 않았다. 그리고 집의 멤버함수중 하나가 사람이 화장실에서 볼일을 보게한다고 치자. 이때 집 클래스만 리버싱으로 헤더를 구현했다고 치더라도 화장실 클래스까지 구현하지 못하면 화장실은 만들 수 없으므로 강제로 집의 멤버함수를 호출할때 사람이 마룻바닥에 똥을 싸게 될 것이다. 결국 집의 영역을 더럽게 만들고 엑세스 바이올레이션이 발생하게 될 것이다. 즉, 헤더가 없이는 제품 개발사의 DLL(써드파티 DLL)에 들어있는 클래스나 함수들을 눈으로 뻔히 보면서도 호출할 수 없다는 것이다. 설계도가 없으면 아무것도 만들 수 없으니까.. (여기서 설명은 오로지 타겟 프로그램의 DLL 파일만 있다고 가정하며 독립적 실행이 가능한 일부 멤버함수는 매우 제한적으로 존재하므로 언급하지 않는다..)

이렇게 길게 설명한 것이 일반적인 DLL 을 설명하기 위해서 설명한 것이다. 즉, 일반적인 DLL 은 헤더파일 없이 DLL 안의 함수를 호출할 수 없으므로 개발자가 타 개발사의 DLL(일명 써드파티 DLL:thirdparty dll) 파일만으로 자신의 프로그램에 덧붙여 개발하는 것은 거의 불가능이라고 할 수 있다.(원론적인 내용이며, 클래스가 아닌 "C" 익스포트 계열은 호출이 가능하다. 그러나 아시다시피 관심있는 프로그램들은 거의 클래스를 EXPORT 하는 것이 추세임)

그렇다면 COM 은 어떤가? COM 이 개발되면서 고려된 점 중에 하나가 바로 위에서 설명한 내용이다. 즉, COM 모듈이라는 분산환경 인터페이스는 헤더파일이 없어도 개발자가 얼마든지 호출할 수 있는 API 를 제공한다. DLL 파일만으로, 그 안의 모든 함수들이 호출가능하도록 설계되었다는 것은 무슨 뜻이겠는가.. 아무나 갖다써라의 개념이다. 실제로 VC++ 에서 COM 모듈을 로딩시키면 자동으로 클래스 헤더가 생성되어 포함된다. 이런 COM DLL 들은 완전 오픈되어 있어서 웹 서비스의 ActiveX 를 자바스크립트로 생성하거나 ASP 로 서버 컴포넌트의 기능을 만들때, 비주얼베이직 스크립트로 OneWay 쉘코드를 짤때(쉘 객체생성부분), 델파이로 컴포넌트를 불러쓸때, VC++ 의 ATL/MFC 등등으로 프로그래밍 할 때 등등 어떤 언어를 막론하고 함수호출이 자유롭다. 완전히 오픈된 DLL 모델인 것이다. 이것이 COM 의 막강한 유연성으로 많은 개발자들 사이에서 각광을 받는 점이다.

그런데 INISAFEWeb60.dll 은 COM 모듈이라는 것이다. 즉, 누구에게나 오픈된 프로그래밍 모델인 COM 을 사용했으며 웹에 방문하는 사람들을 위해서 ActiveX(COM 임) 를 사용한다. "DCOM 제대로 배우기" 라는 책을 읽어보라. 필자가 알고있기로는 VC++ 의 ATL 프로그래밍으로 COM 모듈을 통합해서 자신의 프로그램에 합칠 수 있는 것으로 알고있다. 시도해보지 않아서 확답은 줄 수 없으나 필자의 생각은 인증서 프로그램 쯤이야 저 모듈을 가져다가 필요한 부분만 불러서 쓸 수 있다고 생각한다.

이것이 일반 DLL 과 COM 모듈 DLL 의 차이이다. 이미 배포한 개발자들이 모르고 개발했을리가 없으므로 오픈모델이라는 것에 암묵적인 동의를 한 것이나 마찬가지다. 재밌지 않

은가? COM 모듈(DLL) 을 자신의 개발소스에 포함시켜 완전 통합하는 것이 MS 의 정석적인 COM 개발 방법론에 있다.(심오한 시도는 추후에 문서로 작성해 보일 것이다...) 이미 개발자가 참견할 수 있는 단계는 배포와 동시에 개인의 영역으로 넘어갔으며 이를 이용해서 개발하는 사람도 역시 합법적인 프로그래밍을 하는 것이다. 윈도우즈의 API 를 사용하듯 말이다. 적어도 COM 의 세계에서는 이것이 통용된다고 볼 수 있다. 만약 배포하는 개발자들이 이런현상을 기대하지 않았다면 그냥 일반적인 서버/클라이언트 모델의 프로그램을 만들었어야한다. 설마 꾸미기 쉽고 유행따라 웹으로 가야겠다는 생각에 ActiveX 기반의 프로그램들을 줄기차게 만들어 낸다면 개발모델에 철저히 충실하라고 말하고 싶다.

필자의 화

(잡설란이므로 시간 널널하지 않으면 읽지마셈... 문서를 쓰고나면 몰려오는 새벽녘의 공허함과 수많은 생각들을 항상 문서의 마지막에 끄적이는게 필자의 습관(?)이다. 그래서 사람들이 블로그를 하는걸까.. 필자는 블로그의 가로길이가 너무 좁고 산만한 시스템 때문에 잘 안쓴다. 그래서 블로그 대신 문서의 마지막 란에 잡다구리한 넋두리나 개인적인 생각을 적곤한다... 영양가 없는 내용이 태반이니 심심하면 읽어보시길..)

후킹의 시작...

필자가 생각할 때 한글로 작성된 기술문서를 인터넷으로 구하는 것은 요즘도 상당히 어렵다고 생각한다. 특히 후킹기술에 관련된 문서는 근래에나 인터넷에 올라온 것이지 필자가 공부할때는 문서는 물론이고 한글로 된 책도 없었다. 처음 후킹(사실상 도스의 인터럽트 벡터 공부지만.. 리버스 엔지니어링이란 말도 몰랐음)이란 것에 관심을 가지게 된 것은 중학생시절 부터였는데 그 시절은 PC 통신이 인터넷을 대신하던 때라 모든 정보가 PC 통신으로 통했다. 간간히 인터넷을 연결해서 쓰긴 했지만 잘려버린 모뎀선을 엄마몰래 잇느라 새벽마다 미션임파서블을 방불케하는 안방 공략작전을 펼쳐야만 했다.

그 시절은...

그 당시 학생들에게 가장 인기가 있었던 것 중에 하나가 다들 알겠지만 "야겜" 이란 일본산 성인용(당시는 18금)게임이었다. PC 통신보다도 3.5인치 디스켓에의해 급속도로 퍼진 야겜은 주위에서 안해본 친구가 없을 정도로 파장을 몰고왔는데 상당한 충격이었다. 그러나 필자는 "야겜"에 나오는 미소녀의 현란한 H씬에 충격을 받은 것이 아니라 바로 그 야겜을 한글화 시킨 프로그래머의 실력에 충격을 받았다. 꼴에 크랙한답시고 도스용 소프트아이스(SoftICE)로 JMP 를 NOP 으로 바꾸며 마냥 좋아했던 필자는 제작자들의 프롤로그가 페이드인/아웃 되면서 스크롤 되는 게임화면에 큰 허탈감을 감추지 못했다. PC 통신으로 일본판과 한글판을 모두 받아다가 파일을 비교하며 분석해봤지만 한글화 시킨 파일역시 EXE 파일이었고 데이타 파일이 별도로 존재했다. 실행파일을 C 언어로 바꾸면 가능하지 않을까란 생각에 어떻게든 관련 툴을 찾아대기 시작했고 도스용 디컴파일러부터 유명한 디스어셈블러인 소서(Sourcer), 심지어는 램상주 바이러스소스까지 닥치는대로 모으고 공부했지만 그로인해 몇 년의 세월이 흘러버렸다. 그러나 C 언어의 포인터만 익히는데 무려 4~5년이 걸린 필자의 부족한 실력으로 어셈블리 단으로 이어지는 프로그램 개조란 참담한 실패의 반복만을 가져왔다. 그 후 대세를 거스를 기회조차 주지않고 도스는 빠르게 사양길에 접어들었고 윈도우즈 95로 이동해갔다.

기술이란 이동하는 것...

기술이란 한 시대를 대표하지만 결국 영원하지 않은 것이 현실이다. 8Bit 비디오 게임기였던 패밀리의 "드래곤볼" 이란 게임을 보고 게임을 만들기로 결심했었다. 386 PC 에 드래곤볼이 돌아가는 모습을 보고 싶었다. 그래서 C 언어로 게임 프로그래밍을 공부하기 시작했고 도스의 세그먼트와 싸워 320x200x256 해상도 출력에 성공했다. 그로부터 얼마나 지난 것일까 다시 640x480x256 이 화두가 되어가고 있는 것이었다. 640x480x256 이미지를 한번 출력해보려고 세그먼트 한계를 벗어나려 발악한지도 얼마 안된것 같은데 선형어드레스란 32Bit 보호모드의 개념이 튀어나와 열라 고생하며 익힌 세그먼트 개념에 혼란을 가중시켰다. TurboC 가 사장되며 WATCOM-C 라는 놈이 32bit 를 등에 엎고 대세를 바꿔버린 것이었다. 필자 같이 가난한 사람이 WATCOM-C 를 구하는 것은 정말 힘든 일이었다. 주위에서 프로그래밍을 하는 애들이라고는 컥베이직을 하는 애들이 전부였고 당시 금은방을 하는 부잣집 친구가 56k 전용선을 썼는데 여간 까탈스러운게 아니라서 구해달라고 부탁하기도 힘들었다. 중1 때 였던가.. 프로그래머인 친구형한테 당시 100만원이 넘는 고가였던 볼랜드 TurboC 패키지를 빌리기까지의 눈물겹던 고생을 또다시 시작할 생각을 하니 참으로 한숨만 나왔다. 결국 이리저리 알아보다가 도스용 djgcc 란 32bit GNU 컴파일러가 있다는 것을 알게되었고 선형어드레스를 공부해나갔다. 그러나 하늘도 무심하지.. 겨우 이제 뭔가 하나쯤 만들어 볼 수 있겠다 싶을 무렵, 당시 오락실 최고의 상한가를 치던 "버추어파이터"의 PC 버전이 불과 디스켓 두장짜리에 담겨 급속도로 퍼지면서 이미 윈도우즈 95의 시대가 되었다는 것을 증명이라도 하듯 필자 손으로 하드를 밀게 만들었다. 당시까지도 도스를 고집하며 윈도우즈95의 GUI 가 느리다고 욕하던 자신이 윈도우즈95를 설치하고 있었다.(우리집에서는 386이라 느렸지만 486 이상에서는 겁나게 빨랐다..) 기술이란 정말 빠르게 이동한다. 이동과 동시에 "새로운 공부의 시작"을 요구한다. 아니.. 지금은 이동하기도 전에 공부를 해야만한다. 마치 중학생이 고등학교 교과과정을 미리 공부하는 것처럼.. 이 글을 쓰는 시점도 "윈도우즈 비스타" 라는 놈이 필자와 대적하기위해 눈을 부릅뜨고 있는 것 같다. 시간이 지나면 이 비스타 놈도 굴복을 시켜야 필자도 살 수 있다. 누가 죽나 해보자.. 왜케 기술은 변화가 빠른걸까.. T.T

"어떻게 했길래 원래의 게임을 개조할 수 있었을까?" 바로 이 궁금증이 결국 필자를 여기까지 끌고와 버렸다.

게임을 만들어 보겠다고 프로그래밍을 시작한 것이 "어떻게 했길래 원래의 게임을 개조할 수 있었을까?" 라는 궁금증 때문에 해킹을 공부하게 만들었고 언더해커 그룹에까지 몸을 담그게 만드는 불쌍사를 가져왔다. 몸을 담근것 뿐만 아니라 이 때문에 사람들을 만나게 되었고 터전인 첫직장이 결정되었고 아예 인생 자체의 진로가 바뀌어버렸다. 처음 컴퓨터를 만졌을 때부터 게임을 만드는 프로그래머가 되는 것이 꿈이었을 뿐 해커나 해킹따위는 관심도 없었고 그런게 뭔지도 몰랐다. 그런데 한가지 궁금증이 이토록 인생을 바꿀만큼 큰 작용을 한 건 왜일까.. 가만히 생각해보면 질문에 대한 답을 찾지 못하고 방황한 것이 아니었나 생각된다. 해답을 찾지 못한 채.. 머리속에서 지우지 못한 채 여기까지 온 것 같다. 하지만.. 이제 어느정도 답을 찾은 것 같다. 아직도 완벽한 답을 모두 알아내지 못하고 진실은 저넘어에 있지만 (X 파일? -_-;) 진실을 탐구할 능력과 안목은 얻었다고 생각한다. 남들은 얼마 안걸리는 C 언어 포인터도 4~5 년이나 헤메고 후킹도 결국 엄청난 허송세월을 보내고서야 겨우 이해했지만 이것은 시작에 불과하다. 필자는 이제 방향을 틀어서 처음으로 돌아가야 한 다는 것을 잘 알고있다. 이미 앞서있는 외국의 많은 해커들처럼 무언가 만드는 것에 치중해야 한다. 더이상 소모적인 해킹공부는 하기 싫다. 남는 것도 전혀없다. 루트를 따는 것과 BOF 를 발생시키는 것과 웸해킹을 하는 것이 어떤 의미가 있을까에 대해서 진지하게 생각해 봐야할 때는 이미 오래전에 지났다. 언제였던가... 비록 나이는 아래지만 실력은 결코 그렇지 않았던 한 친구의 말이 생각난다. 음악가는 음악을 만들어서 사람들을 기쁘게하고 화가는 그림을 그려서 사람들을 기쁘게하는데 우리는 같은 예술을 하는데 도대체 무엇을 해서 사람들을 기쁘게 해줄 수 있을까? 라고.. (대부분 동감할 것이다.. 해킹을 공부하다보면 예술을 알게된다..) 뭔가 하기만 하면 불법이라는 낙인이 찍히는 것에 두려움을 갖고 아무런 활동도 보여줄 수 없다는 것을 서로 잘 알고있었기에... 그러나 사실 답은 이미 오래전부터 있었다. 아주 가까운 곳에 말이다.

지난 수 개월동안 필자는 해커들의 이상적인 활동을 생생하게 목격했다. 그들은 그들의 방향을 그냥 있는 그대로 즐기고 있을 뿐...

뛰어난 외국 해커들은 프로그램을 만들고 작동시키는 것이 행복 그 자체이며 많은 사람들과 같이 즐긴다. 필자가 그것을 가장 극명하게 확인할 수 있었던 분야가 임베디드이다. 임베디드 분야는 대부분 미개척지대이기 때문이다. 한동안 필자는 PSP 라는 작은 게임기에 빠져서 해커들이 노는 장면을 거의 스토커처럼 매일매일 감시했다. 그들은 PSP 라는 게임기에서 자신들이 만든 프로그램을 작동시키기 위해서 버퍼오버플로우라는 기술(MIPS CPU 를 대상으로 한 버퍼오버플로우였다..)을 사용하며 자신들이 만든 프로그램이 한계를 극복하게 하려고 커널을 후킹한다. 그리고 그들은 마지막으로 남들이 창조적인 활동(개발)을 하도록 독려하기 위해서 API 를 만들어 낸다. 그들의 모습은 버퍼오버플로우를 숭배하지 않으며 후킹을 기술로 취급하지 않는 것처럼 보였고 단지 프로그램을 만들기위해 API 제작에 필요한 모든 정보를 찾고 공유하는 것이 목적이자 바로 과정이었다. 그리고 더욱 놀라운 것은 그들은 Exploit 을 공유하기보다도 Exploit 을 할 수 있는 아이디어를 공유하는 독특한 모습을 보여줬다. 놀랄일이지만 터무니없어 보이는 Exploit 아이디어가 현재 PSP 에서 개인 프로그램을 돌릴 수 있는 실제 Exploit 이 되어 현재까지도 사용되고 있다. 누군가 PSP 1.0 버전이 메모리스틱을 빼도 프로그램이 계속 작동되는 것을 보고 메모리스틱 두개를 번갈아 바꿔끼면서 해보는건 어떻겠냐고 제의한 후 얼마 뒤에 스페인 해커로 보이는 이들이 Exploit 에 성공했다. 이 때문에 소니(sony) 사는 펌웨어 업데이트를 실시해 메모리 스틱을 빼면 프로그램이 종료되도록 변경했다. Exploit 을 배포하는 해커들은 모든 사람들에게 Exploit 이 성공하여 개인 프로그램을 돌리는데 성공하였음을 알렸으며 언제 어떤 사이트를 통해서 프로그램을 배포할 것이라는 선전포고(?)를 하였다. 그들이 선택한 방식인 Exploit 선전포고는 그야말로 활동하고 있음을 홍보하는 최상의 방법이었고 모든 사람들이 열광하는 듯 보였다. 국내의 해커들이 본받을 만한 노하우인것 같다.(그들 역시 언더이지만 홍보를 한다.. 그리고 기부금을 받기를 원한다) 그리고 이 정보를 거의 실시간으로 전달하는 여러 커뮤니티가 있었다. 그 후 실제 배포가 이루어졌고 그 모습은 더더욱 인상깊었는데 그들의 그룹은(팀이라고 하는게 더 어울릴 것이다) 로고 디자인에 누구, 개발에 누구, 문서작성에 누구 등등 모든 분야를 세분화해서 임무를 나누고 있었다. 우리나라처럼 해커그룹이 20 명이라면 20 명 모두 해킹을하지 않으면 하수로 취급받는 구조와는 완전 차별화를 이루고 있었다. 이런 모습은 외국의 경우에 자주 목격할 수 있는데 필자가 본 것이 틀리지 않았다면 국내에서도 이런 모습을 볼 수 있다. 필자만 그렇게 봤는지 모르지만 유일하게 프로그램이나 게임을 한글화하는 팀에서만 이런경우를 볼 수 있었다. (개발이라는 요소가 존재하는 팀) 필자가 생각하는 해커들의 활동은 이런모습이고(리소스 편집기만 사용하는 것이 아닌) 개발이라는 요소가 가미된 한글화를 진행하는 사람들이 진정 해커들이라고 말하고 싶다. 국내의 한글화 팀들은 개발이라는 요소를 갖추고 있으며 배포라는 것을 통해서 사람들에게 도움을 주고 번역과 개발, 디자인 같은 각각의 일을 분담하고 있다. 심지어는 그들의 기술과 정보를 공유한다.

이런 외국해커들의 활동에서 필자는 배워야 할 점을 보게됐는데 그건 바로 서로간의 공유와 연합이다. PSP 해킹을 성공한 팀이나 시도하는 팀들이 PS2DEV 라는 사이트에서 활동하는 사람들한테 고마움을 전하는데서 찾아볼 수 있었던 점이다. 해커팀들은 PS2DEV 라는 플레이스테이션과 PSP 공개해킹 포럼(실질적으로는 그룹이나 다름없지 않나..) 에서 정보를 얻고있었고 그 포럼에서 활동하는 사람과 일종의 조인트 해킹을 진행했던 것으로 보인다. 해킹이 성공한 다음에는 배포문서에 PS2DEV 에 감사한다라는 말이 쓰여져있었고 PS2DEV 에서 활동하는 사람의 닉네임이 적혀있기도 했다. 우리나라에서는 쉽지않은 일을 그들은 자연스럽게 한다. (해킹 성과도 잘내는 것 같다..) 여기에 한발 더 앞서서 PSP 뉴스만을 전문적으로 포스팅하는 커뮤니티들이 한단계 진보하면서 서로서로 앞다투어 자체 개발팀을 창단하는 것을 목격했는데 필자가 지원을 하자 다른 커뮤니티에서 자기 개발팀으로 와달라고 꼬시는 메일이 오기도 했다.

필자는 이런 외국해커들의(한국인도 있겠지요.. 다만 언어권으로 볼때..) 활동모습이 매우 자연스럽게 이루어지는 것이 해킹의 발전상이라고 생각한다. 해킹을 위한 한가지 주제를 놓고 아주 자연스러운 커뮤니티가 형성되며 그 속에서 또 자연스러운 연합(팀)이 탄생하고 개발정보와 기술이 오고가며 토론되고 뉴스거리가 탄생할 수 있는 환경이 해킹을 진보하도록 하는게 아닐까... 오래전 해커스랩이 국내에서 이런 역할을 아주 충실히 함으로써 해커그룹들이 탄생하였고 발전하였지만 지금은 서로간의 고립을 자초하는게 아닌가 생각한다. 공유는 이익집단 이외에는 금지되어야 하고 Exploit 이나 기술은 마치 무협지에나 나오는 내공처럼 생각하며 해킹기술을 설명한 문서는 동방불패의 "규화보전" 인양 생각하는 매우 폐쇄적이고 이기적이 되었다. 그러나 외국의 해커들이 보면 웃을지 모를 일이다.. 아마 많은 사람들이 동감할 테지만 외국애들이 해킹하는 것을 보면 자신이 우물안 개구리 밖에 안된다는 것을 알게된다. 필자는 임베디드 시스템을 해킹하는 해커들을 보고 정말 괴물과도 같은 해킹실력의 소유자라는 생각을 했다. 불가능해 보이는 암호화 시스템을 풀어내질않나(복호화 시킨 개발자는 소니의 클리에처럼 로봇을 갖고 노는 사람.. 여기서 로봇은 일본제...), 마음대로 커널을 후킹해서 펌웨어를 가상으로 돌리질 않나 시시각각 여러 게임기의 에뮬레이터들이 개발되질 않나.. 그런 사람들은 아마 PC 에서 해킹하는건 재미가 없어서 임베디드를 하는 것으로 보일 지경이다. (핵심적인 것을 만드는 놈들은 꼭 일본놈들이다. 필자의 생각에 임베디드에서 두각을 나타내는 나라가 독일과 프랑스 같은 유럽일 것으로 판단되지만 활동 무대가 미국일 수 있으니 콘솔을 해킹하는 것은 미국이 강세이고 콘솔기기를 만드는 것과 같은 고급기술을 구사하는 것은 일본 놈들인 것으로 보인다. PSP 에서 펌웨어를 에뮬레이팅시키는 DEVHOOK 도 일본인이 제작한 것으로 판단되고 여러 게임기 에뮬레이터를 만든 제작자를 따라가보면 일본인이 자주 나온다. 필자는 아무리 중국이 발전해도 일본에 숨어있는 실력자들은 넘어설

수 없다고 생각한다. 다른 분야에서는 더더욱 극명하게 나타난다. 예를들면 요즘 국내에 커뮤니티가 생기기 시작한 포저라고 하는 3D 인체 그래픽 프로그램에 관련된 프로그래밍은 이미 일본에서 오래전부터 있었는데 국내에서는 이제서야 커뮤니티가 활성화 되었고 활성화 된 시점에서 아직도 프로그래밍을 다루는 곳이 존재하지 않는다. (중국도 이보다는 낫겠다..) 이처럼 최소한 일본은 있는데 우리나라는 없는 것이 한두개가 아니것은 찾아보면 잘 알것이다. 인터넷을 돌아다녀봐도 일본꺼는 지원하는데 한국꺼나 중국꺼는 지원안되는 것이 무지 많이 있다. 이것이 일본의 로비(?) 때문이라고 한다면 그건 시샘이다. 시샘은 도움이 되지 않는다. 냉철하게 판단하면 그들의 실력을 인정하는 것이고 드러나지 않은 실력의 표출이고 숨어있는 놈들이 일본에 많다는 것을 암시하는 것이라고 생각한다. 모르긴 몰라도 최소한 세계는 아니더라도 동양에서 가장 뛰어난 해커는 일본에 살고있을 것이다. 다만 잠자코 있을뿐이라고 생각한다. 왜일까.. 월드컵에서 브라질 축구선수가 일본소속으로 나오는 것을 보면 그들이 데리고 있지 못할 이유도 전혀 없지 않은가... 국익에 도움이 된다면.. 어느 나라에 있던 뛰어난 해커를 돈으로 유혹해서 일본국적으로 만드는 것은 충분히 예측할 수 있다. 누군가 말하길 세계에서 일인자로 평가받는 사람들중 100 명 이상이 전부 일본인이라고... 과연 음모론일까.. 필자는 일본 사이트에서 그들의 기술수준을 볼때마다 놀라움을 감출 수 없다. 매우 상세하게 가르치고 공부하는 것이 그들의 스타일이기 때문이다. 대충이란 없어 보인다.. 독한놈들..)

잡설에 잡설을 거듭했는데 어쨌든 해야할 일은 극명히 정해져 있었다. 바로 해커로.. 개발자로.. 돌아가는 것이라고 생각한다. 왜 그토록 오픈소스를 외치고 해커들과 오픈소스를 연결시켜왔는지 그 정신을 이제는 이해할 수 있을 것 같다. 해커는 바이너리만 달랑 존재하는 프로그램에 새 생명을 불어넣는 작업을 후킹을 통해서 할 수 있고 이를 다시 API 화 시킬 수 있으며 이 API 는 죽은 프로그램과도 같은 바이너리에 새 생명을 API 로써 불어넣을 수 있다. 또한 해커들은 리버스 엔지니어링을 통해서 소스를 복원하는 것도 연구하며 현실로 옮겨왔다. 그리고 항상 자신들의 소스코드와 신기술을 공개하며 선두에서 달리는 것이 해커들이라고 생각한다. 곰곰히 한번 생각해보라. 아마 자신이 작성한 프로그램이나 문서가 있고 그것을 외부에 공개한 적이 한번도 없이 하드에 짱박혀 있다면 다시한번 열어보길 바란다. 지금에 와서는 한낱 쓰레기가 되어있을 것이다. 여러 사람들에게 도움이 되지 않는 것은 본인한테도 도움이 되지 않는다. 결국 쓰레기가 될 뿐이다. 그 이유는 기술은 변화하고 변화한 기술속에서 예전의 시점은 변화를 같이하지 않았기 때문에 의미가 퇴색되기 때문이다.

필자는 psp2dev.org 라는 PSP 해킹 사이트를 운영하면서 도스용 야겜을 윈도우에서 돌아가게 해주는 anise 라는 에뮬레이터를 PSP 로 포팅해서 배포한 적이있었다. 어느날 인터넷을 서핑하던 중 국내 제작자가 도스용 야겜을 윈도우에서 돌릴 수 있도록 에뮬레이터를 개발해서 공개했는데 공백기에 접어들어서 업데이트가 안되고 있었다. 그 후 또 한참 지나서 누군가 자우루스라는 PDA 용으로 anise 를 포팅해서 올려놓는 것이 아닌가.. 그 때가 바로 필자가 PSP 를 사고 해킹에 한참 열을 올리던 때였는데 PSP 에서 도스용 야겜을 돌릴 수 있다면 정말 많은 사람들이 재밌어 하겠구나라는 생각이 번뜩 들었다. 예전의 중고등학교시절 추억에 빠져들지 않을까하고 말이다. 일주일동안 밤낮을 안가리고 포팅에 매진해서 결국 배포를 하게 되었다. 그런데 몇달 뒤에 서버에 문제가 생겨서 자료가 날라가는 바람에 귀찮아서 더이상 운영을 하지 않았다. 그 뒤 몇달이 지나 인터넷에서 재미난 것을 보게되었다. 누군가 필자가 배포한 프로그램을 게임 커뮤니티에 공유했는데 댓글이 수십개 달려서 어떤 게임이 돌아가고 어떤 게임은 안돌아간다는 식으로 나름대로 돌릴 수 있는 게임목록을 주고받고 하는게 아닌가.. 순간 필자는 이것이 바로 뿌듯함이구나라는 것을 느꼈다. 배포당시에는 아무도 안하는줄 알고 업데이트도 안하고 나중에는 아예 배포도 중단했는데 프로그램을 원하는 사람들이 있었다는 것을 나중에서야 알게된 것이다. 그래서 필자는 다시 묵은 소스를 꺼내어 업데이트를 하고있다. 좀더 빠르고 좀더 편하게 사람들이 사용하라고 말이다. anise 를 처음 개발하고 인터넷에 공개한 사람은 anise 가 PSP 로 포팅된 것을 보면 또 다른 기쁨을 맛보지 않을까...

해커들은 예술가들처럼 그림이나 음악으로 사람들을 즐겁게 해줄 수 없지만 열심히 공부해서 얻은 분석능력과 개발능력으로 프로그램을 개발하여 사람들을 돕고, 즐겁게 할 수 있다. 심지어 불가능해 보일지라도 해커들은 개발할 수 있는 분석력과 끈기가 있다... 이미 오래전부터 그렇게 해왔고 지금도 그렇게 하고있는 해커들이 많이있다. 다만 해커라고 자부하는 사람들이 그렇게 하고 이끌어 나가기를 바랄뿐이다. 게임을 즐기는 것을 즐기기 보다 자신이 만든 게임을 즐겨주는 것을 더 즐기는 순리를 따라가야 할 것이다. 그것이 프로그램이 되었든 문서가 되었든 다같이 즐길 수 있는 해커문화를 꽃피울 수 있기를 희망한다. 힘차게 도약하며 피어나는 새 시대의 어린해커들이 거짓된 망상속에서 허우적되지 않게 하기위해서 이 글을 써본다... (무엇이 거짓된 망상인지는 깨달음을 얻고 스스로 판별해야 할 것이다... 그렇다고 산으로 도닦으러 가면 곤란함..) 벌써 아침이 밝았네.. -_-;

더이상 소모적인 도전에 자신들의 뛰어난 능력을 매진하지 말라...
한줄의 코드가 백줄이 되고 백줄의 코드가 백만줄이 되기전에 과감히 한줄의 코드를 공유하는 것이 중요하다...
이땅의 해커로 살려는 자들에게 외쳐보고 싶다... Return to Programmer...
CPU 와 대화하라.. 더 늦기 전에...