기 술 문 서

Anti Reversing Techniques

박지훈
sonicpj@nate.com

Preface

프로그램 개발 완료 후 다가올 리버싱의 위험

새로운 트랜드에 대한 열망은 가득 차 있으나 아이디어 고갈에 허덕이며 새로운 프로그램이 등장하지 않는 요즈음. 어느 날 개발자 A씨는 몇 년간의 고된 개발 기간을 거쳐 'COREA'라는 프로그램을 만들어 상용화 하는데 성공했다. 이 제품은 현재의 인터넷 사업 풍토에서 분위기를 완전히 뒤엎어준 획기적인 솔루션으로 이름이 알려지기 시작했다. 당연히 프로그램으로 인한 매출은 급등하였으며 유료 사용자들은 나날이 늘어갔다. 몇 년간의 힘든 고생이 빛을 발하는 순간이었으며 A씨의 순이익도 덩달아 늘어만 갔다. 그러던 어느 날 'COREA'라는 프로그램의 크랙 버전이라는 것이 유포되기 시작했다. 크랙 버전을 사용하면 유료 등록을 하지 않아도 'COREA'를 얼마든지 이용할 수 있다. 매출과 직결되는 위험한 이 크랙 버전은 인터넷 여기저기 퍼지기 시작하면서 A씨를 힘들게 하였다. 정보를 입수해 본 결과 'COREA'의 크랙 버전은 C라는 해커가 개발했으며 'COREA'를 리버스 엔지니어링 하여 유료 등록 루틴을 무력화 시켜 버렸다는 것을 알게 된다. 당연히 그 때문에 무료 이용자라도 누구든지 사용할 수 있는 상태가 되며 이 크랙 버전이 퍼지면서 매출은 급감하고 만다. 그리고 A씨와 회사의 목줄은 점점 조여만 갔다.

리버싱에 대항하는 A씨의 반격, 그 후...

A씨는 유료 등록 루틴을 좀 더 어렵게 변경해 다음 버전을 릴리즈했다. 힘들게 수정한 만큼 이제 기존 크랙 방식은 사용되지 않을 거라고 확신했다. 그러나 너무 안일한 생각이었다. 새 버전을 발표하자마자 하루 만에 또다시 크랙버전이 등장했다. A씨는 크랙 버전을 입수해 다시 유심히 살펴봤다. 그리고 자신이 코딩한 등록 루틴이 이번에도 무력화 된 것을 발견한다. 어떻게 이런 일이 가능한 것일까? 어떻게 해야 크랙 버전의 유포를 막을 수 있을까? A씨는 고민한 결과 자신과 친분이 있는 B씨가 몇몇 해커와도 교류가 있다는 것을 알고 그 루트를 통해 해커를 만나 컨설팅을 받는다. 해커가 해놓은 답은 하나였다. "당신의 프로그램은 너무 분석하기 쉽습니다. 각종 디버거 등에 완전히 무방비 상태여서 아무리 어렵게 등록 루틴을 개발해 봤자 리버스 엔지니어링을 통해 금방 유린이 됩니다."라는 소리를 들었다. 즉 내가 만든 코드를 보호하기 위해 디버깅 방지, 리버싱 방지를 위한 기능이 필수적이라는 사실을 깨닫게 된다. 이후 A씨는 '안티 디버깅(Anti-Debugging)' 이라는 기술에 대해 알게 된다. 그리고 관련 기법을 총동원하여 솔루션에 적용시키기에 이르렀다.

Micro Software 잡지 2008년 8월
실전강의실3 - 안티 리버싱과 리버스 엔지니어링

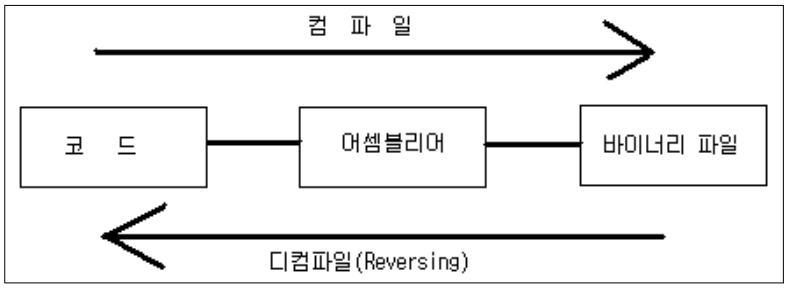
Content

1. Reversing

리버싱(reverse + engineering)이란 이미 완성된 소프트웨어를 역으로 분석하는 과정이다. 개발자가 소스코드를 작성하고 자신이 사용하는 컴파일러로 컴파일 시켜 바이너리 파일(소프트웨어)을 만든다. 이런 바이너리 파일은 사람이 읽기 힘들도록 되어 있지만 리버싱을 이용해 사람이 알아 볼수 있도록 기존의 소스를 복원하는 것이다.(디컴파일)

아무리 리버싱을 한다고 해도 기존의 소스를 전부 알아내기는 힘들지만 어셈블리 코드를 분석해 원래 소스의 구조와 원리 정도는 알아내게 되는 것이다.

예를 들어 해당 소프트웨어에서 종료를 누르면 어느 부분에서 어떤 코드가 어떻게 실행되는지는 알 수 있다. 위 설명을 간단히 그림으로 표현해 보겠다.



이렇듯 크래커들은 리버싱으로 개발자가 만들어 놓은 소프트웨어를 역분석해 코드를 알아내버려서 인증하는 부분을 우회해 버린다거나 코드를 바꿔서 부당한 이익을 취한다거나 할 수 있다. 그렇기 때문에 개발자들은 크래커들이 자기 소프트웨어를 리버싱하지 못하도록 방어하는 기술이 안티(Anti) 리버싱 기술이다.

2. Anti Reversing

앞에서 설명했듯이 안티 리버싱이라는 것은 크래커들로부터 리버싱을 차단하려는 것이다. 하지만 완벽하게 리버싱을 차단하는 것은 불가능하다. 그렇기 때문에 안티 리버싱 기법을 많이 적용시켜 역분석을 하는 크래커들을 지치게 해서 포기하게 하는 것이다. 안티 리버싱 기술도 신뢰성 있고 안전함에 따라 많은 비용이 들어간다.

안티 리버싱 기술이 필요한 경우는 예를 들어 파일 복제 방지할 때나 저작권 관리, 사용자들이 돈을 지불하고 정품인증 하는 소프트웨어 등 이런 것들에서는 꼭 필요한 기술이다.

안티 리버싱에도 심볼 제거, 패킹, 코드 체크섬, 안티 디버깅 기술 등 여러 가지가 있는데 간단한 안티 리버싱 기술부터 알아볼 것이다.

3. Simple anti reversing techniques

3.1. Remove the symbol information

첫 번째로 소개할 것은 심볼 정보를 제거 하는 것이다. 리버싱을 해본 사람들은 알겠지만 문자열, dll이 익스포트하는 함수만으로도 해당 바이너리 파일에서 패스워드를 검증하는 부분이 어디인지 오류창을 띄워주는 부분이 어디인지를 쉽게 알 수 있다. 그렇기 때문에 바이너리 파일안의 문자열 정보를 제거하는 것이다. 혹은 문자열들을 의미 없는 문자열로 바꾸는 것이다. C기반 언어들은 일반적으로 바이너리 파일에 심볼 정보가 포함되지 않지만 바이트 코드 기반의 언어인 경우에는 내부 주소대신 이름을 참조하기 때문에 리버싱하기가 아주 쉬워진다. 그렇기 때문에 바이트 코드 기반인 경우는 꼭 심볼 정보는 제거, 변경 해줘야 할

것이다.

3.2. Packing

두 번째로 소개할 것은 패킹이다. 리버서들이 코드 분석을 방지하기 위해 코드를 패킹 하는 것이다. 이렇듯 바이너리 파일을 패킹해 놓으면 원래 소스를 볼 수 없기 때문에 정적 분석은 불가능 해진다. 실제 소스를 봐보자.

```
#include <stdio.h>

int main(void)
{
        int a = 10;
        int b = 20;
        int result;

        result = a + b;

        return 0;
}
```

```
00411360   55              PUSH EBP
00411361   8BEC            MOV EBP,ESP
00411363   81EC E4000000   SUB ESP,0E4
00411369   53              PUSH EBX
0041136A   56              PUSH ESI
0041136B   57              PUSH EDI
0041136C   8DBD 1CFFFFFF   LEA EDI,DWORD PTR SS:[EBP-E4]
00411372   B9 39000000     MOV ECX,39
00411377   B8 CCCCCCCC     MOV EAX,CCCCCCCC
0041137C   F3:AB           REP STOS DWORD PTR ES:[EDI]
0041137E   C745 F8 0A0000  MOV DWORD PTR SS:[EBP-8],0A
00411385   C745 EC 140000  MOV DWORD PTR SS:[EBP-14],14
0041138C   8B45 F8         MOV EAX,DWORD PTR SS:[EBP-8]
0041138F   0345 EC         ADD EAX,DWORD PTR SS:[EBP-14]
00411392   8945 E0         MOV DWORD PTR SS:[EBP-20],EAX
00411395   33C0            XOR EAX,EAX
00411397   5F              POP EDI
00411398   5E              POP ESI
00411399   5B              POP EBX
0041139A   8BE5            MOV ESP,EBP
0041139C   5D              POP EBP
0041139D   C3              RETN
```

<그림 1>

위 소스를 디버거로 열어보면 위와 같이 보이는데 패킹을 하지 않고 봤을 때의 코드이다. 다른 부분의 코드를 보더라도 암호화 되지 않고 전부 읽을 수 있는 코드이다.

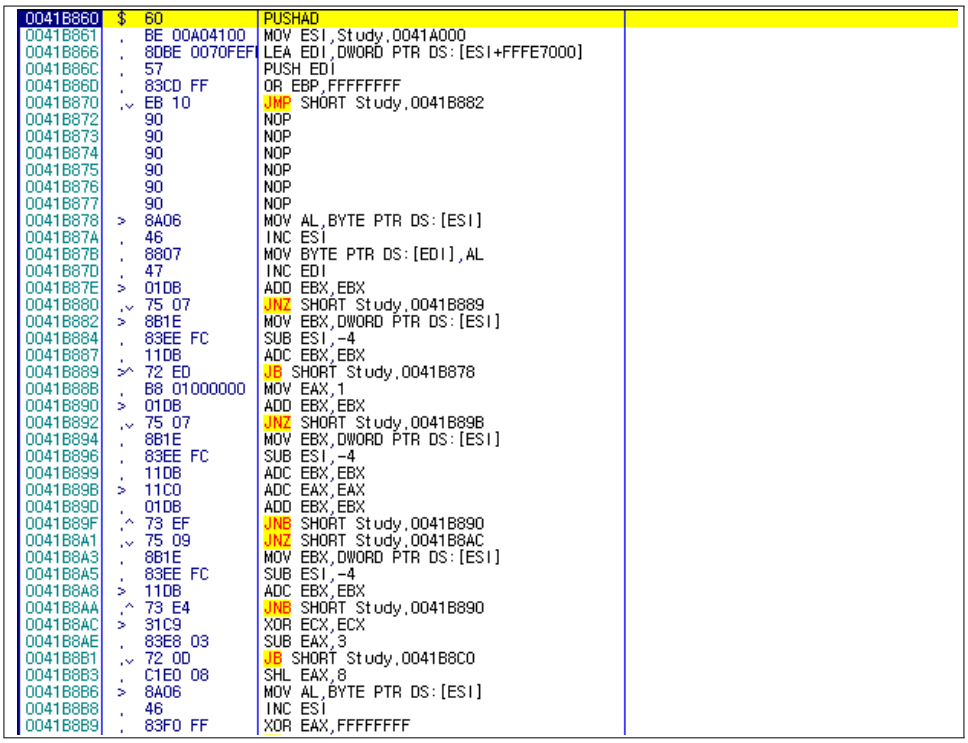
이제 패킹 되었을 때의 코드를 봐보자. 패커의 종류만해도 엄청나게 많고 사용자가 직접 패커를 만들 수도 있지만 여기선 간단한 패킹인 UPX를 사용하겠다.

```
        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     40960 ->      8704   21.25%   win32/pe      Study.exe

Packed 1 file.
```

<그림 2>

패킹이 끝났고 다시 디버거로 코드를 열어보겠다.

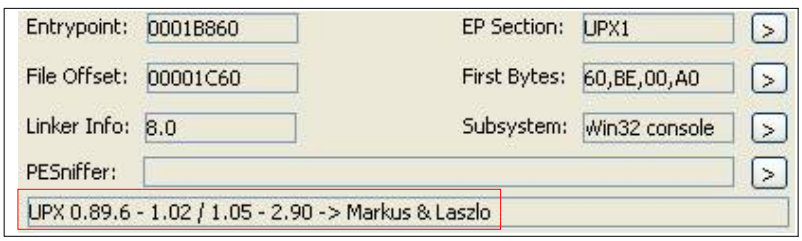


```
0041B860  $  60            PUSHAD
0041B861  .  BE 00A04100   MOV ESI,Study.0041A000
0041B866  .  8DBE 0070FEFF LEA EDI,DWORD PTR DS:[ESI+FFFE7000]
0041B86C  .  57            PUSH EDI
0041B86D  .  83CD FF       OR EBP,FFFFFFFF
0041B870  .v EB 10         JMP SHORT Study.0041B882
0041B872     90            NOP
0041B873     90            NOP
0041B874     90            NOP
0041B875     90            NOP
0041B876     90            NOP
0041B877     90            NOP
0041B878  >  8A06          MOV AL,BYTE PTR DS:[ESI]
0041B87A  .  46            INC ESI
0041B87B  .  8807          MOV BYTE PTR DS:[EDI],AL
0041B87D  .  47            INC EDI
0041B87E  >  01DB          ADD EBX,EBX
0041B880  .v 75 07         JNZ SHORT Study.0041B889
0041B882  >  8B1E          MOV EBX,DWORD PTR DS:[ESI]
0041B884  .  83EE FC       SUB ESI,-4
0041B887  .  11DB          ADC EBX,EBX
0041B889  >^ 72 ED         JB SHORT Study.0041B878
0041B88B  .  B8 01000000   MOV EAX,1
0041B890  >  01DB          ADD EBX,EBX
0041B892  .v 75 07         JNZ SHORT Study.0041B89B
0041B894  .  8B1E          MOV EBX,DWORD PTR DS:[ESI]
0041B896  .  83EE FC       SUB ESI,-4
0041B899  .  11DB          ADC EBX,EBX
0041B89B  >  11C0          ADC EAX,EAX
0041B89D  .  01DB          ADD EBX,EBX
0041B89F  .^ 73 EF         JNB SHORT Study.0041B890
0041B8A1  .v 75 09         JNZ SHORT Study.0041B8AC
0041B8A3  .  8B1E          MOV EBX,DWORD PTR DS:[ESI]
0041B8A5  .  83EE FC       SUB ESI,-4
0041B8A8  >  11DB          ADC EBX,EBX
0041B8AA  .^ 73 E4         JNB SHORT Study.0041B890
0041B8AC  >  31C9          XOR ECX,ECX
0041B8AE  .  83E8 03       SUB EAX,3
0041B8B1  .v 72 0D         JB SHORT Study.0041B8C0
0041B8B3  .  C1E0 08       SHL EAX,8
0041B8B6  >  8A06          MOV AL,BYTE PTR DS:[ESI]
0041B8B8  .  46            INC ESI
0041B8B9  .  83F0 FF       XOR EAX,FFFFFFFF
```

<그림 3>

아까와 코드가 완전히 다르다. Comment 부분에도 아무 것도 없을뿐더러 코드 자체도 우리가 알던 소스가 전혀 아니다. 이렇듯 패킹이 되어 버리면 정적 분석으로는 알 수가 없다. 그렇다고 해서 절대로 못 푸는 것은 아니다. 크래커들이 파일을 분석 할 시간을 늘릴 수 있다는 것이다. 뒤에서도 말 할 것이지만 안티 리버싱이란 기술이 아무리 뛰어나도 많은 시간과 인내가 있으면 뚫리는 것이다. 하지만 얼마나 많은 끈기를 요구하게 하느냐가 관건이다. 실력이 안되는 크래커들은 안티 리버싱 기술이 강력 할수록 늪에 빠지게 되고 결국은 리버싱을 포기하게 되는 것이다.

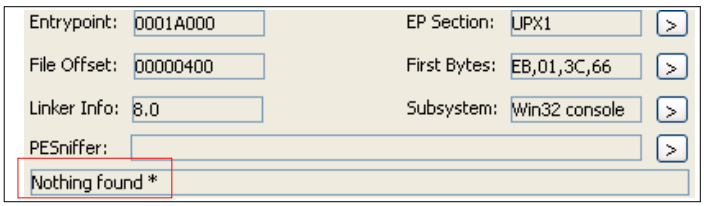
위에서 보여준 패킹은 언패킹 하는 방법이 널리 퍼져 있고 툴도 만들어져 있다. 위 패킹된 바이너리 파일의 정보를 봐보자.



<그림 4>

간단하게 어떤 방식으로 패킹이 되어 있는지를 확인 할 수 있다.

하지만 PE확인 툴을 우회하는 기능을 사용한다거나 사람들이 알지 못하는 패커을 사용한다거나 자기가 직접 패커를 만들어서 사용하게 된다면 리버서들은 어떤 패킹방법으로 패킹이 되어있는 지의 정보를 얻지 못하게 될 것이고 코드를 분석하는데 많은 시간을 투자하게 될 것이다.



<그림 5>

위 그림은 잘 알려지지 않은 패커로 패킹했을 때의 결과이다. 어떠한 정보도 얻지 못하고 라이브 디버깅을 해봐야한다. 그리고 잘 알려져 있으면서도 언패킹하기 어려운 패커로는 Themida, Winlicense, VMProtect 같은 패커들이 있다. 이 패커들을 가지고 패킹을 할 경우 UserMode, KernalMode 디버거로 분석을 하려고 Attach하면 안티 디버거 기술 때문에 디버거가 꺼지는 현상, PE확인 툴로도 해당 패커를 확인 할 수 없는 것 등을 볼 수 있을 것 이다. 또 한 가지 장점으로는 패킹을 잘 이용하면 바이너리 파일 용량도 줄일 수 있다. 위에서 설명한 UPX패킹 전 후의 용량을 비교해 보겠다.

파일은 Starcraft.exe 파일로 해보겠다.

<그림 6>

패킹 전	Microsoft Visual C++ 7.0 [Debug]	1.16MB (1,220,608 바이트)
패킹 후	UPX 0.89.6 - 1.02 / 1.05 - 2.90 -> Markus & Laszlo	557KB (570,880 바이트)

<그림 7>

패킹 전 후의 바이너리 파일 정보와 용량을 나타낸 것이다.

용량을 보면 알겠지만 절반 이상의 용량이 줄어든 것을 볼 수 있다. 어떤 바이너리 파일들은 4배까지도 줄어든다. 이렇듯 패킹이라는 것은 잘 사용하면 크래커들이 리버싱을 못하도록 방해도 할 수 있을뿐더러 방대한 양의 바이너리 파일을 용량도 줄일 수 있다는 것이다.

4. Anti debugger techniques

4.1. Anti debugger

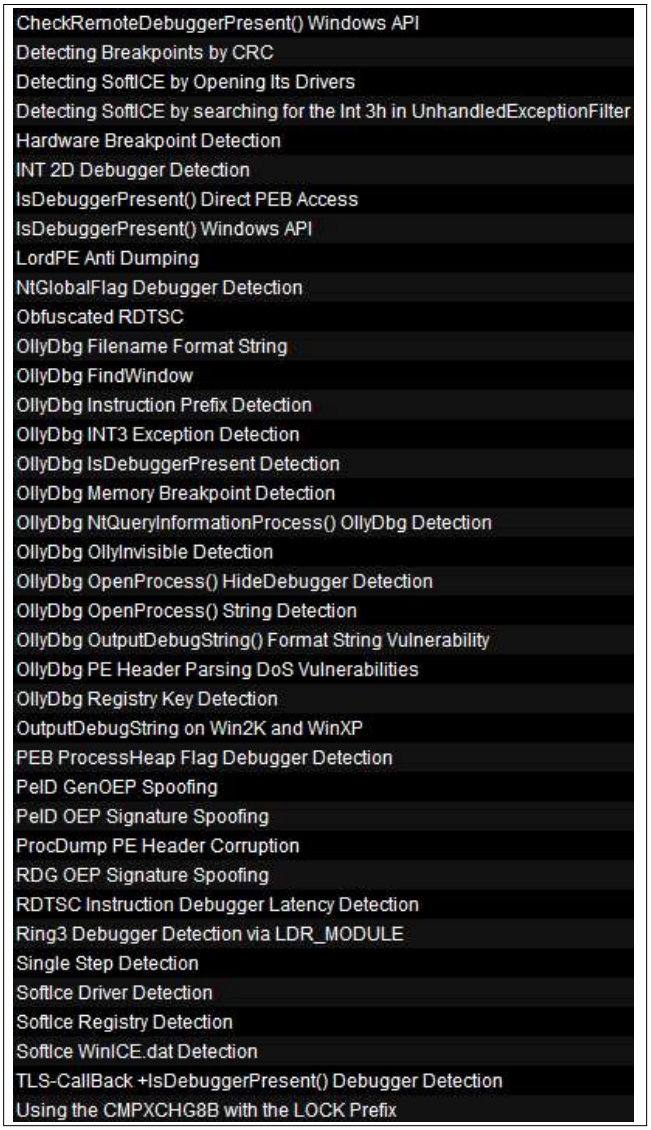
안티 디버거라는 것은 안티 리버싱 기술에 속한다. 대부분 리버싱을 하기 위해서는 디버거를 사용하고 디버거 안에서 브레이크 포인트를 설정하고 트레이스 해가면서 분석을 한다.

이렇듯 안티 디버거라는 것은 이러한 디버거를 탐지하는 기술이다. 이런 기술들을 사용해 디버거가 탐지될 경우 해당 디버거를 꺼버릴 수도 있고 바이너리 파일을 꺼버릴 수도 있다.

또한 파일을 패킹 시켜놓을 경우 크래커들은 라이브 디버깅을 해가며 언패킹을 해야 하므로 디버거를 사용해야 한다. 그렇기 때문에 패킹과 함께 안티 디버깅 기술을 사용하면 리버싱을 방어하는데 효과를 더욱 볼 수 있다.

4.2. IsDebuggerPresent API

안티 디버거를 기술을 사용하기 전에 이런 기술들이 들어있는 여러 API함수들이 있다. 얼마나 많은 안티 리버싱 기술과 안티 디버깅 API함수가 있는지 봐보자.



<그림 8>

보시는 바와 같이 많은 안티 리버싱 기술과 안티 디버깅 기술이 있지만 위의 기술이 전부는 아니다.

이제 우리가 이번에 알아볼 안티 디버깅 API함수로는 IsDebuggerPresent()함수이다.

위에서 보여준 패킹과 마찬가지로 위 함수도 디버거를 탐지하는 함수들 중에 가장 기본적인 함수이다. 2가지를 보여줄 것인데 함수를 사용한 코드와 코드 안에 인라인 어셈을 한 코드를 가지고 설명을 할 것이다.

먼저 위 함수를 사용한 간단한 코드를 짜보자.

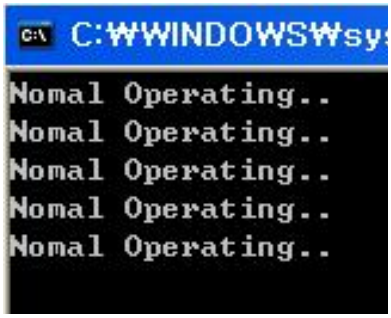
(참고, IsDebuggerPresent() 함수는 디버깅 시 리턴 값이 1이다.)

정상 실행

```
#define _WIN32_WINNT 0x501
#include <stdio.h>
#include <Windows.h>

int main(void)
{
    while(1)
    {
        Sleep(1000);

        if(IsDebuggerPresent())
            printf("Debuggering...₩n");
        else
            printf("Nomal Operating.. ₩n");
    }
}
```



<그림 9>

위 코드를 정상실행 시 옆 그림과 같이 출력이 된다.

정상 실행을 하였으므로 IsDebuggerPresent()의 리턴 값은 0이 돼서 Nomal Operating을 출력하게 된다.

(위 소스에서 #define _WIN32_WINNT 0x501이라고 한 부분이 있는데 어떤 함수를 사용할 때 해당 프로그램이 실행될 OS를 지정해 주는 것이다.

위 소스에서는 IsDebuggerPresent()라는 함수를 사용하기 위해 define시켜준 것이다. #include<Windows.h>헤더 위쪽에 선언을 해 주어야 한다.)

_WIN32_WINNT 표

_WIN32_WINNT	WINVER
Windows Server 2008	0x0600
Windows Vista	0x0600
Windows Server 2003 SP1, Windows XP SP2	0x0502
Windows Server 2003, Windows XP	0x0501
Windows 2000	0x0500

<그림 10>

)

위 코드를 디버깅 상태에서 실행을 시켜보겠다. 그리고 그때 IsDebuggerPresent()에서 리턴 되는 값을 봐보겠다.

디버깅 중

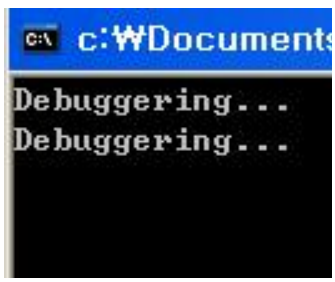
```
#define _WIN32_WINNT 0x501
#include <stdio.h>
#include <Windows.h>

int main(void)
{
    int a;
    while(1)
    {
        a = IsDebuggerPresent();
        Sleep(1000);

        if(a)
            printf("Debuggering...₩n");
        else
            printf("Nomal Operating.. ₩n");
    }
}
```

조사식 1

이름	값	형식
a	1	int



<그림 11>

디버깅 상태로 넘어오니 IsDebuggerPresent()리턴 값이 1이 되면서 Debuggering이라는 문자열이 출력 되는 것을 볼 수 있다.

이제 IsDebuggerPresent()함수가 어떤 일을 하는지는 알게 되었고 Assembly코드로 어떻게 수행을 하고 있는지 봐보겠다.

```
mov eax,fs:[0x30]
lea edx,[eax];
xor eax,eax
mov al,byte ptr [edx+0x2]
RETN
```

<그림 12>

위 그림이 IsDebuggerPresent()함수 내부를 봐본 것이다.
설명을 하기전에 기본적으로 TEB구조체와 PEB구조체를 알아야한다.

TEB구조체 : 운영체제가 동작하기 위해서 필요한 스레드 정보를 포함하는 구조체
PEB구조체 : 유저레벨에서 프로세스에 대한 정보를 저장하고 있는 구조체

먼저 TEB구조체를 Offset 0x30까지 나열해보자.

```
0x000   NtTib                           : _NT_TIB
0x01c   EnvironmentPointer              : Ptr32 Void
0x020   ClientId                        : _CLIENT_ID
0x028   ActiveRpcHandle                 : Ptr32 Void
0x02c   ThreadLocalStoragePointer       : Ptr32 Void
0x030   ProcessEnvironmentBlock         : Ptr32_PEB
```

<그림 13>

Offset 0x30에 PEB구조체를 가리키고 있는 것을 볼 수 있다.

PEB구조체도 Offset 0xC까지 봐보자.

```
0x000 InheritedAddressSpace     : UChar
0x001 ReadImageFileExecOptions  : UChar
0x002 BeingDebugged             : UChar
0x003 SpareBool                 : UChar
0x004 Mutant                    : Ptr32 Void
0x008 ImageBaseAddress          : Ptr32 Void
0x00c Ldr                       : Ptr32_PEB_LDR_DATA
```

<그림 14>

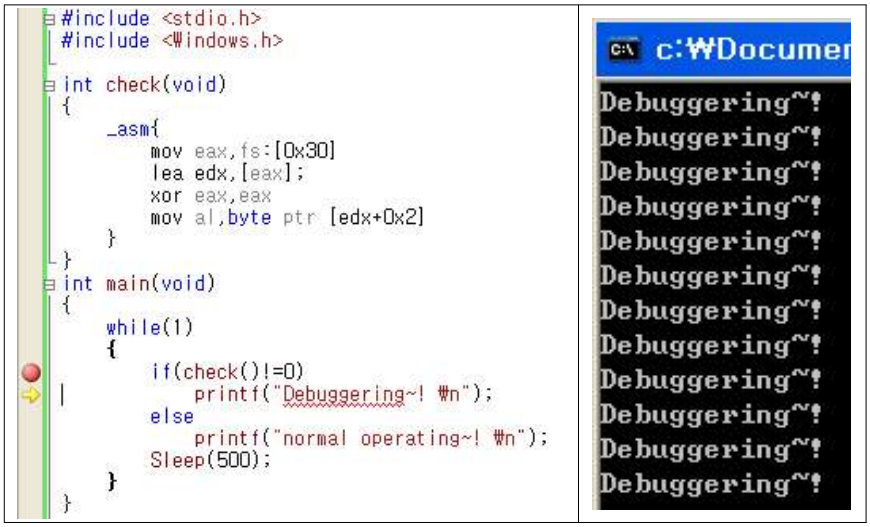
여기 까지 정보를 알고 있으면 <그림 12>에서의 코드를 이해 할 수 있다.

mov eax,fs:[0x30]
참고로 FS:[0]은 TEB구조체 처음을 가리키고 있다. 그러므로 FS:[0x30]은 PEB구조체를 가리키게 된다.
lea edx,[eax];
xor eax,eax
위 두 줄은 크게 볼 건 없다.(그 주소를 레지스트리 edx에 넣는 과정)

mov al,byte ptr [edx+0x2]
RETN
PEB를 가리키는 주소에서 0x2만큼 이동하면 BeingDebugged변수를 가리키게 되고 그 안에 들어있는 값을 리턴해 주는 것이다.

결론을 내려보면 IsDebuggerPresent()함수는 PEB구조체 안의 BeingDebugged변수 안의 값을 가지고 현재 파일이 디버깅 중인지 아닌지를 판단하는 것이다. 디버깅 중이면 BeingDebugged변수에 1값이 들어갈 것이고 아니라면 0값이 들어가게 되는 것이다.

위 정보를 토대로 코드를 인라인 어셈을 사용하여 코드를 만들어 실험해 보겠다.



<그림 15>
아까와 같이 디버거가 탐지되고 있는 것을 알 수 있다.

4.3. NtGlobalFlag, Heap.HeapFlags, Heap.ForceFlags

이번에는 NtGlobalFlag, Heap.HeapFlags, Heap.ForceFlags 값을 이용한 안티 디버깅 기술이다. 위에서 소개한 IsDebuggerresent()와 큰 차이는 없다.
NtGlobalFlag값이 정상 일땐 0으로 되어있고 WinDBG, OllyDebugger, ImmunityDebugger, IDA, Visual studio 등에서 프로세스를 디버깅하게 되면 NtGlobalFlag 값이 0x70으로 값이 설정된다. 코드를 짜서 봐보자. 리턴 값은 Ollydebugger로 봐보겠다.
(NtGlobalFlag는 PEB구조체에서 0x68만큼 떨어진 곳에 존재한다.)

```
#include <stdio.h>
#include <Windows.h>

int check(void)
{
    _asm{
        mov eax,fs:[0x30]
        lea edx,[eax]
        xor eax,eax
        mov al, byte ptr [edx+0x68]
    }
}
int main(void)
{
    while(1)
    {
        if(check()==0x70)
            printf("Debuggering~! \n");
        else
            printf("Normal operating~! \n");
        Sleep(500);
    }
}
```

```
004113B6  33C0        XOR EAX,EAX
004113B8  8A42 68     MOV AL,BYTE PTR DS:[EDX+68]
004113BB  5F          POP EDI
```

EAX 00000070
ECX 00000000
EDX 7FFD6000

<그림 16>

0x004113B8코드를 실행 후 EAX에 0x70이 들어간 것을 볼 수 있다. 올리디버거로 디버거 중이므로 당연히 0x70값이 들어간 것이다.

Heap.HeapFlags, Heap.ForceFlags도 마찬가지로 디버깅 중이면 0x50000062, 0x40000060 이라는 값을 갖게 될 것이다.

Heap.HeapFlags 위치	Heap.ForceFlags 위치

mov eax,fs:[0x30]
mov eax,[eax+0x18]
mov eax,[eax+0xc]
	mov eax,fs:[0x30]
	mov eax,[eax+0x18]
	mov eax,[eax+0x10]

<그림 17>

위 두 개도 간단한 코드로 실험을 해 보겠다. 이번에는 올리디버거에서의 값만 보여주겠다.

Heap.HeapFlags	004113AE 64:A1 30000000 MOV EAX,DWORD PTR FS:[30] 004113B4 8B40 18 MOV EAX,DWORD PTR DS:[EAX+18] 004113B7 8B40 0C MOV EAX,DWORD PTR DS:[EAX+C] 004113BA 5F POP EDI EAX 50000062 ECX 00000000 EDX 10312D30 EBX 7FFDC000
Heap.ForceFlags	004113AE 64:A1 30000000 MOV EAX,DWORD PTR FS:[30] 004113B4 8B40 18 MOV EAX,DWORD PTR DS:[EAX+18] 004113B7 8B40 10 MOV EAX,DWORD PTR DS:[EAX+10] 004113BA 5F POP EDI EAX 40000060 ECX 00000000 EDX 10312D30 EBX 7FFDC000

<그림 18>

디버깅 중이므로 0x50000062, 0x40000060값을 지니게 된 것을 볼 수 있다.
이런 안티 디버깅 기술이 엄청나게 많이 있지만 여기서는 간단하게 어떤 식으로 안티 디버깅 기술이 적용되는지를 보여주기 위해서 몇몇 예제를 보여주는 것이다. 개발자들은 이제 탐지를 하기만 하면 안 되고 탐지가 된 후 어떤 대처를 할 것인가에 대한 코드도 만들어야 할 것이다. 이런 안티 디버깅 기술에 관련해서는 마지막으로 하나만 더 보여 줄 것이다.

4.4. Trap Flag

마지막으로 소개할 안티 디버깅 기술은 트랩 플래그(TF) 값을 설정해 예외 발생 여부로 디버거 탐지를 하는 것이다. 이 안티 디버깅 장점은 유저모드 디버거와 커널모드 디버거 모두 탐지를 할 수 있기 때문이다. 이번에도 인라인어셈블리로 코드를 짜볼 것 이다. 그 전에 몇 가지 알아두면 이해하기 편하다.

```
__try - __except : SEH(예외)를 처리하기 위한 키워드

__try
{
    // Guarded code
}
__except (EXCEPTION_EXECUTE_HANDLER) // Exception filter
{
    // Exception handling code
}

__try 부분에 예외를 검사할 코드를 넣습니다. 예외가 발생하면 __except 부분에서 지정한 코드가 실행되게 됩니다.

EXCEPTION_CONTINUE_EXECUTION (-1)
예외를 무시하고, 예외가 발생한 지점에서부터 프로그램을 계속 실행한다. 예를 들어 10 / i 에서 i가 0이라서 예외가 발생한 경우, 예외 처리 필터가 이 값이라면, 다시 10 / i부터 실행한다는 말이다.

EXCEPTION_CONTINUE_SEARCH (0)
except 블록 안의 코드를 실행하지 않고, 상위에 있는 예외 처리 핸들러에게 예외 처리를 넘긴다.

EXCEPTION_EXECUTE_HANDLER (1)
except 블록 안의 코드를 실행한 후, except 블록 아래의 코드를 실행한다.
```

<그림 19>

<그림 19>와 같은 예외처리 루틴을 사용할 것이다.

```
#include <stdio.h>
#include <Windows.h>

BOOL check(void)
{
    BOOL Exception = FALSE;
    __try{
        _asm{
            pushfd
            or dword ptr [esp],0x100
            popfd
            nop
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER){
        Exception = TRUE;
        return Exception;
    }
    return Exception;
}
int main(void)
{
    printf("Trap Flag \n");
    while(1)
    {
        if(check()==FALSE)
            printf("Debuggering~! \n");
        else
            printf("Normal operating~! \n");
        Sleep(500);
    }
}
```

<그림 20>

pushfd명령어는 모든 플래그 값들을 스택에 넣으라는 명령어이고,
popfd명령어는 스택에 있는 값들을 플래그에 넣으라는 명령어이다.

빨간색 네모 친 곳을 설명 하겠다.
처음에 check()함수에 들어오고 **pushfd**명령어를 수행 후의 플래그와 스택을 봐보겠다.

Flag	Stack
C 0 ES 0023 P 1 CS 001B A 0 SS 0023 Z 0 DS 0023 S 0 FS 003B T 0 GS 0000 D 0 O 0 LastErr EFL 00000206	0012FD90 00000206 0012FD94 58DD1136 0012FD98 0012FF68 0012FD9C 0012FE9C 0012FDA0 7FFDE000 0012FDA4 CCCCCCCC

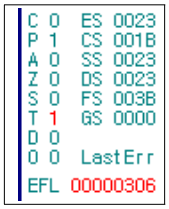
<그림 21>

플래그 값들이 전부 스택에 들어간다고 했는데 스택을 봐보니 0x206이라는 숫자만 들어갔다. 그 이유는 C, P, A, Z, S, T, D, O라고 된 부분이 플래그인데 빨간 네모 친 부분 EFL 부분이 위 플래그들을 전부 관리하는 값이라고 할 수 있다. 즉, 플래그들 중 값 하나만 바뀌어도 EFL도 다른 값으로 변경이 된다는 것이다. 그리고 위 플래그를 보면 T라고 된 부분이 TF이고 현재 0으로 되어있는 것을 볼 수 있을 것이다.
다음 줄을 봐보자. esp에 들어있는 값과 0x100을 OR연산을 하고 있다. 연산을 해보자.

0x206	1000000110_b
0x100	100000000_b
0x306	1100000110_b

<그림 22>

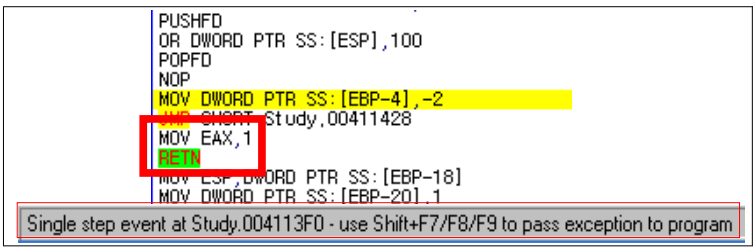
2진수 9번 째 자리를 1로 셋팅해 주고 있다. 이제 0x306이라는 값을 EFL에 들어가게 된다면 TF값은 1로 셋팅이 될 것이다. 마지막 줄인 **popfd**라는 명령어 수행 후를 봐보자.



<그림 23>

EFL값은 0x306으로 되어있고 TF는 1로 셋팅이 되어 있는 것을 볼 수 있다. 즉, TF가 1로 셋팅이 되었다는 것은 예외가 발생했다는 것이다. 이제 바로 예외처리 루틴으로 넘어가는가 싶더니 아래 nop이라는게 나왔다. nop은 크게 신경 쓸건 없지만 실험해본 결과 **TF가 1로 셋팅 되고 난 후 코드 한 줄을 더 실행 후 예외처리 루틴으로 가게 된다.** 그렇기 때문에 아무 의미 없는 nop을 한줄 집어넣은 것이다. 만약 그 한 줄에 프로세스를 종료 시키는 코드를 넣었다면 예외처리 루틴을 가지도 못하고 프로세스가 종료 될 것이다.

nop까지 실행하고 난 후의 코드를 보자.



<그림 24>

그림의 아래를 보면 예외가 발생했고 예외처리 루틴으로 가려면 Shift+F7/F8/F9를 사용하라고 하였다. 아마 예외처리 루틴으로 가서 트레이스 해보면 빨간색 네모 친 부분으로 가게 될 것이고 결국 check()는 리턴 값 true로 Nomal Operating이라는 문자열을 출력하게 될 것이다. 하지만 디버거에서 이런 트레이스 과정을 거치지 않고 바로 실행시켰다고 해보자.

그러면 디버거는 예외처리 루틴을 먹어버리고 아래 JMP문을 수행해서 예외처리 루틴을 건너 뛰어버리게 된다. 직접 봐보자. 현재 예외처리 루틴으로 가려면 Shift+F7/F8/F9를 사용하라고 했는데 그냥 F8(코드 한줄 실행, Visual Studio에서 F10과 동일)을 눌러보자.

F8한번 눌렀을 때 예외처리 먹어버림	**계속 트레이스 후** **예외처리 루틴 건너 뛰어버림**
PUSHFD OR DWORD PTR SS:[ESP],100 POPFD NOP MOV DWORD PTR SS:[EBP-4],-2 JMP SHORT Study.00411428 MOV EAX,1 RETN MOV ESP,DWORD PTR SS:[EBP-18] MOV DWORD PTR SS:[EBP-20],1	MOV DWORD PTR SS:[EBP-4],-2 JMP SHORT Study.00411428 MOV EAX,1 RETN MOV ESP,DWORD PTR SS:[EBP-18] MOV DWORD PTR SS:[EBP-20],1 MOV EAX,DWORD PTR SS:[EBP-20] MOV DWORD PTR SS:[EBP-EC],EAX MOV DWORD PTR SS:[EBP-4],-2 MOV EAX,DWORD PTR SS:[EBP-EC] JMP SHORT Study.00411428 MOV DWORD PTR SS:[EBP-4],-2 MOV EAX,DWORD PTR SS:[EBP-20] MOV ECX,DWORD PTR SS:[EBP-10] MOV DWORD PTR FS:[0],ECX

<그림 25>

설명한 대로 이다. 여기서 설명하고 싶은 것은 결국 이런 종류의 안티 리버싱 기술을 찾아내기 위해서는 차근차근 트레이스를 해 보는 수 밖에 없다. 이 부분을 찾아내질 못하고 그냥 지나치게 된다면 디버거탐지가 될 것이고 결국 역 분석에 실패를 하게 될 것이다.

5. References

[1] http://www.openrce.org/reference_library/anti_reversing

[2] 리버스엔지니어링 역분석 구조와 원리 - 박병익/이강석 공저

[3] 리버스 엔지니어링 비밀을 파헤치다 - 엘다드 에일람