윈도우 프로그래머를 위한 PE 포맷 가이드
실행 파일 보안의 원리

목차

소개

인기 있는 프로그램은 많은 사람들의 공격 대상이 된다. 호기심 때문에 프로그램을 분석하는 사람들도 있고, 더러는 상업적인 이익을 얻기 위해서 프로그램을 악의적으로 사용하기도 한다. 이러한 해커들의 공격으로부터 실행 파일을 보호할 수 있는 방법에 대해서 살펴본다.

연재 가이드

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

연재 순서

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

2007. 11 코드 패칭
2007. 12 바이러스
2008. 01 진화하는 코드
2008. 02 실행 압축의 원리
2008. 03 실행 파일 보안의 원리
2008. 04 실행 파일 프로텍터

필자소개

신영진 pop@jiniya.net, http://www.jiniya.net
웰비아닷컴에서 보안 프로그래머로 일하고 있다. 시스템 프로그래밍에 관심이 많으며 다수의 PC 보안 프로그램 개발에 참여했다. 현재 데브피아 Visual C++ 섹션 시삽과 Microsoft Visual C++ MVP로 활동하고 있다. C와 C++, Programming에 관한 이야기를 좋아한다.

필자 메모

H모 통신 모뎀이 고장 나서 A/S 기사를 불렀다. 모뎀이 뜨거워지면 끊기는 것 같아서 증상을 말하자 모뎀 교체는 해주질 않고 다짜고짜 악성코드 때문이라며 막 머라고 한다. 그러면서 한 때 필자가 개발에 참여하기도 했었던 악성코드 제거 프로그램을 설치한다.

WoW 길챗을 보고 있자면 다음과 같은 진풍경이 벌어지곤 한다.
A: 엄마 감시하에 와우 하다보니 알트탭 신공이 늘어서 이젠 0.54초만에 화면 전환을 한다는...
B: A야, 알트탭 하지마. 메모리에 무리 가서 컴퓨터에 좋지 않아. 걍 창모드로 해.
C: 난 10시간 넘게 알트탭 하면서 해도 별로 그런 거 못 느끼겠던데
B: 난 삼일 이상은 컴터 안꺼
A: 역시 B는 우리랑은 다른 차원의 사람이야

음악방송에서 다음과 같은 멘트가 나온다.
CJ: 정보검색사 자격증을 따려고 준비하려고요. 6개월 정도 걸린다는데 학원을 알아보고 해야겠네요. T님이 IT로 오시려고 하는거냐고 물어보시네요. 그런건 아니고. 아 웹프로그래머 시라고요. 웹프로그래머면 뭐 게임만들고 3D 디자인하고 그런 거 하는 거 아닌가요?

생각보다 많은 사람들은 IT와 무관하게 살아간다는 느낌이 드는 요즘이다. 저런 상황에 처할 때마다 어떻게 하는 것이 옳은 일인지 궁금해질 때가 많다. 잘못된 지식을 바로잡아 주는 것이 맞는 것인지 아니면 속 편하게 조용히 있는 게 좋은 것인지 말이다. 물론 대부분의 경우에 필자는 조용히 있는 편이다.

Introduction

얼마 전 스타크래프트 프로리그 결승전을 보러 갔었다. 평소 안티만 있는 줄 알았던 한 선수에게 사인을 받으려고 끝없이 줄을 서 있는 사람들을 보면서 안티가 많다는 것은 팬도 많다는 말을 실감할 수 있었다. 프로그램도 마찬가지여서 인기 있는 프로그램일수록 안티도 늘어나고 공격을 하는 사람도 늘어난다. 상용 프로그램의 경우에는 더 말할 필요도 없다. P2P 프로그램을 타고 돌아다니는 수많은 키 생성 프로그램과 시리얼 키를 보고 있자면 이 말을 어느 정도 실감할 수 있다.

그렇다면 이런 악성 사용자 내지는 해커로부터 프로그램을 보호할 순 없는 것일까? 결론부터 말하자면 S급 해커로부터 클라이언트 프로그램을 보호한다는 것은 사실상 불가능하다. 밀실에 물건을 숨기는 문제를 생각해보면 간단하게 그 이유를 짐작할 수 있다. 아주 복잡한 구조물로 가득 찬 밀실에 금고를 하나 숨긴다고 생각해 보자. 누군가 금고를 방안에 숨기고 10분 뒤에 다른 사람이 들어가서 그 금고를 찾는 것이다. 과연 그 금고를 완전하게 숨길 수 있는 방법이 있을까? 당연히 없다. 금고는 결국 방안에 존재하고 그것을 찾느냐 못 찾느냐는 보물찾기를 하는 사람의 능력에 달린 것이기 때문이다. 마찬가지로 클라이언트 프로그램의 코드 또한 CPU가 실행을 시킨다. 아무리 복잡한 테크닉들을 쓰더라도 어느 순간에 CPU는 결정적인 명령어 목록을 수행할 것이고 유능한 해커라면 그것들을 모두 볼 수 있을 것이기 때문이다. 이쯤이면 천문학적인 예산이 소요된 소프트웨어도 하루면 크랙이 나오는 이유를 조금은 이해할 수 있을 것도 같다.

이번 시간에 우리는 해커의 공격으로부터 실행 파일을 보호하는 다양한 방법에 대해서 살펴볼 것이다. 간단한 암호를 체크하는 프로그램을 통해서 단계별로 해커가 공격할 수 있는 경로를 살펴보고 그것에 대응할 수 있는 방법들을 찾아보도록 하자.

1레벨: 시작

<리스트 1>에 이번 시간에 사용할 예제 프로그램이 나와 있다. 간단하게 사용자에게 암호를 입력 받고 검사하는 프로그램이다. 암호가 맞으면 ok를 출력하고 틀리면 바로 종료되는 간단한 프로그램이다.

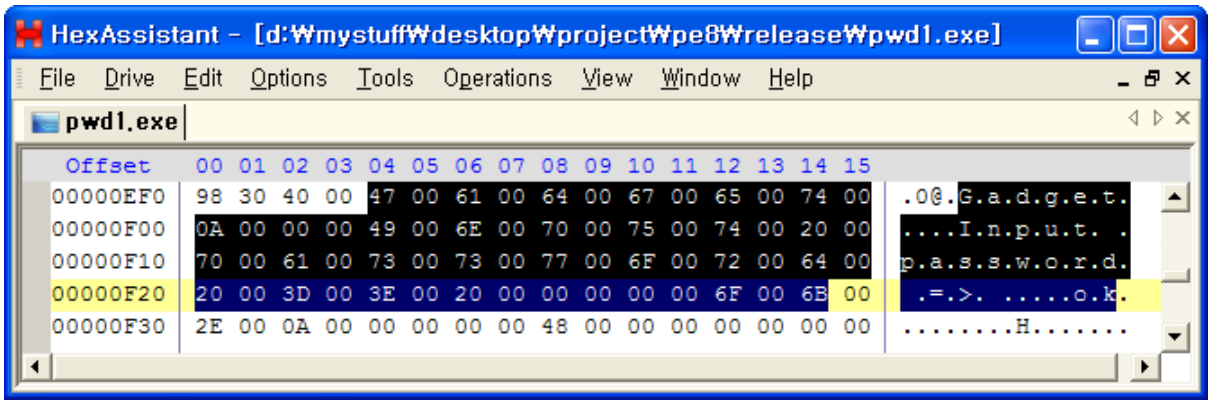
리스트 1 pwd1 프로그램

```
#include <stdio.h>
#include <string.h>
#include <tchar.h>
#include <windows.h>

LPCTSTR g_password = _T("Gadget₩n");

int main()
```

```
{
	const DWORD BUF_SIZE = 80;
	TCHAR buf[BUF_SIZE + 1];

	_tprintf(_T("Input password => "));
	_fgetts(buf, BUF_SIZE, stdin);
	if(_tcscmp(buf, g_password) == 0)
		_tprintf(_T("ok.₩n"));

	return 0;
}
```

2레벨: 데이터 보안

암호를 알아내기 위해서 해커가 가장 먼저 하는 일은 이미지 자체를 살펴보는 작업이다. 그리곤 화면에 출력되는 내용들을 검색해보거나, 문자열 추출 도구를 사용해서 어떤 문자열들이 포함되어 있는지를 살펴볼 것이다. <화면 1>에는 헥사 에디터를 통해서 password를 찾은 화면을 보여준다. 우리가 암호로 선정했던 Gadget이란 단어가 그대로 보인다. 아마도 해커는 이 암호를 대입해 보고는 ok 메시지를 보게 될 것이다.



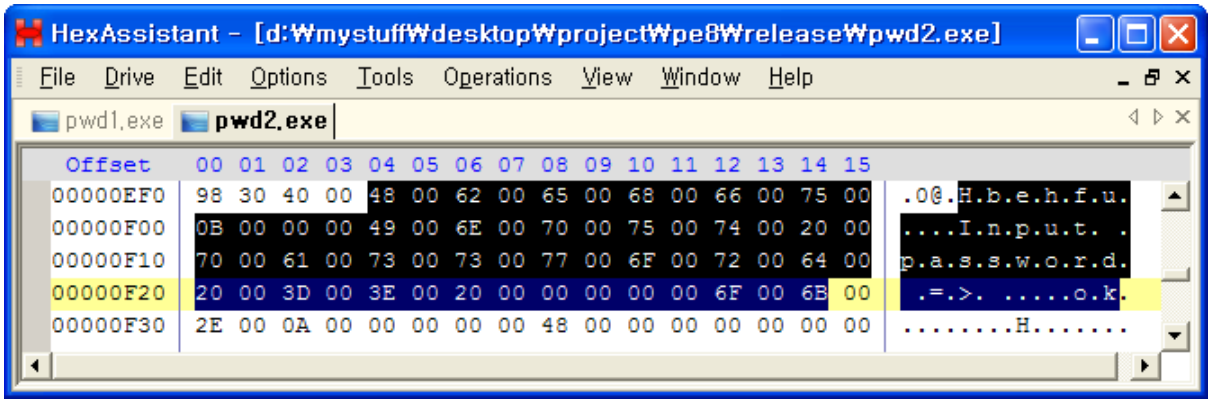
화면 1 pwd1의 헥사 덤프

일년 내내 전시하는 미술 작품과 일년에 딱 두 시간 공개되는 미술 작품 중 어떤 것이 도난 당할 확률이 높을까? 당연히 일년 내내 전시하는 미술 작품이다. 이는 프로그램에서도 마찬가지다. 중요한 데이터가 메모리에 존재하는 시간이 길 수록 노출될 확률은 높고 해커가 그것을 찾아낼 확률도 증가한다. 따라서 항상 중요한 데이터는 암호화를 기본으로 하고 필요한 경우에만 그것을 복원해서 사용하는 것이 좋다. 필요하지 않다면 복호화를 하지 않는 편이 더 좋다. pwd1 프로그램의 암호 같은 경우에도 복호화를 수행할 필요가 없다. 암호화된 데이터를 저장해두고, 사용자의 입력 값을 암호화해서 비교하면 된다. 이렇게 수정한 pwd2 프로그램이 <리스트 2>에 나와 있다.

리스트 2 pwd2 프로그램

```
#include <stdio.h>
#include <string.h>
#include <tchar.h>
#include <windows.h>

LPCTSTR g_password = _T("Hbehfu₩xb");

LPTSTR Encrypt(LPTSTR str)
{
	LPTSTR p = str;
	while(*p) { ++*p; ++p; }
	return str;
}

int main()
{
	const DWORD BUF_SIZE = 80;
	TCHAR buf[BUF_SIZE + 1];

	_tprintf(_T("Input password => "));
	_fgetts(buf, BUF_SIZE, stdin);
	if(_tcscmp(Encrypt(buf), g_password) == 0)
		_tprintf(_T("ok.₩n"));

	return 0;
}
```

<화면 2>는 pwd2 프로그램에서 password를 찾아보 화면이다. Hbefu라는 암호 문자열이 보이지만 해커가 그것을 입력하더라도 더 이상 ok 메시지를 볼 순 없다. 왜냐하면 암호화된 문자열이기 때문이다.



화면 2 pwd2의 헥사 덤프

3레벨: 체크섬

암호를 찾지 못한 해커의 다음 공격 대상은 코드다. 코드를 직접 패칭해서 암호가 틀리더라도 맞춘 것처럼 넘어가도록 프로그램을 조작하는 것이다. <화면 3>처럼 디스어셈블러로 코드를 분석한 다음 암호 비교 후 점프하는 jnz 명령어를 jz로 변경해 버리는 것이다. 실제로 헥사 에디터를 통해서 해당 부분의 코드인 0x75, 0x0e를 0x74, 0x0e로 변경해 버리면 암호가 틀린 경우에 ok 메시지가 출력되는 것을 확인할 수 있다.

```
.text:0040108B loc_40108B:                             ; CODE XREF: _main+84↑j
.text:0040108B                 test    eax, eax
.text:0040108D                 jnz     short loc_401099
.text:0040108F                 push    offset aOk_     ; "ok.₩n"
.text:00401094                 call    esi
.text:00401096                 add     esp, 4
.text:00401099
.text:00401099 loc_401099:                             ; CODE XREF: _main+8D↑j
.text:00401099                 mov     ecx, [esp+0ACh+var_4]
.text:004010A0                 pop     esi
.text:004010A1                 xor     ecx, esp
.text:004010A3                 xor     eax, eax
.text:004010A5                 call    __security_check_cookie
.text:004010AA                 add     esp, 0A8h
.text:004010B0                 retn
.text:004010B0 _main           endp
```

화면 3 암호 비교 부분의 디스어셈블 코드

패칭은 프로그램이 실행되기 전의 파일 단계에서 수행되기 때문에 사전에 막는 것은 사실상 불가능하다. 따라서 최선책은 사후에라도 패치된 사실을 알아내서 적절한 대응을 하는 것이다. 이렇게 정보의 위, 변조 여부를 체크하는 것을 무결성 검사라고 한다. 보통은 MD5 해시 등을 사용해서 이러한 무결성 검사를 수행한다. <리스트 3>에는 체크섬을 사용해서 중요한 코드 부분이 변조 되었는지를 체크하도록 수정한 프로그램의 소스 코드가 나와 있다. 코드가 변경되었다면 바로 프로그램이 종료된다. pwd3을 컴파일해서 실행 파일을 만든 다음 디스어셈블러로 점프 문을 패치할 경우에는 프로그램 실행이 아예 되지 않는 것을 알 수 있다.

리스트 3 pwd3 소스 코드

```
#define CODE_CHECKSUM(sum, s, e) ₩
	__asm { ₩
		__asm push e ₩
		__asm push s ₩
		__asm call Checksum ₩
		__asm mov sum, eax ₩
		__asm add esp, 8 ₩
	}

DWORD Checksum(PVOID s, PVOID e)
{
	PBYTE bs = (PBYTE) s;
	PBYTE be = (PBYTE) e;
	DWORD sum = 0;

	while(bs < be)
	{
		sum += *bs;
		++bs;
	}

	return sum;
}

int main()
{
	const DWORD BUF_SIZE = 80;
	TCHAR buf[BUF_SIZE + 1];
	DWORD sum;

	CODE_CHECKSUM(sum, ic_start, ic_end);
	if(sum != 0x74b3)
		return 0;

ic_start:
	_tprintf(_T("Input password => "));
```

```
	_fgetts(buf, BUF_SIZE, stdin);
	if(_tcscmp(Encrypt(buf), g_password) == 0)
		_tprintf(_T("ok.₩n"));
ic_end:

	return 0;
}
```

4레벨: 코드 보안

단순 패칭이 통하지 않는다면 해커는 코드를 좀 더 진지하게 읽을 것이다. 그의 다음 공격 대상은 encrypt 함수다. encrypt 함수는 암호를 생성하는 핵심 부분이기 때문에 이를 해커가 이해했다면 프로그램은 해커에게 넘어간 것이나 다름없다. 시중에 나와 있는 수많은 키 생성 프로그램들이 그러한 형태로 제작되는 것이다. 따라서 해커로부터 encrypt 함수를 숨겨야 한다. 하지만 여기엔 한가지 문제가 있다. 컴퓨터는 그 코드를 실행 시켜야 하고, 해커는 보지 말아야 한다는 딜레마가 그것이다. 어쩔 수 없이 그 사이에서 절충을 하는 수 밖에는 없다. 우리는 해커가 단순히 디스어셈블러를 통해서 손쉽게 순수 encrypt 함수 코드에 접근하는 것을 막는 것을 목표로 할 것이다.

그렇다면 디스어셈블러가 encrypt 코드를 해석하지 못하도록 만드는 방법엔 무엇이 있을까? 여기엔 크게 두 가지 방법이 있다. 하나는 1월호에 소개했던 것과 같이 함수 자체를 실행 파일로부터 분리를 해내는 것이고, 다른 하나는 앞서 소개한 데이터 보안과 같이 코드를 암호화 시켜놓고 실행될 때 복호화해서 코드를 실행하는 것이다. <리스트 4>에는 코드를 복호화해서 실행하는 두 번째 방법을 사용한 프로그램의 소스 코드가 나와 있다.

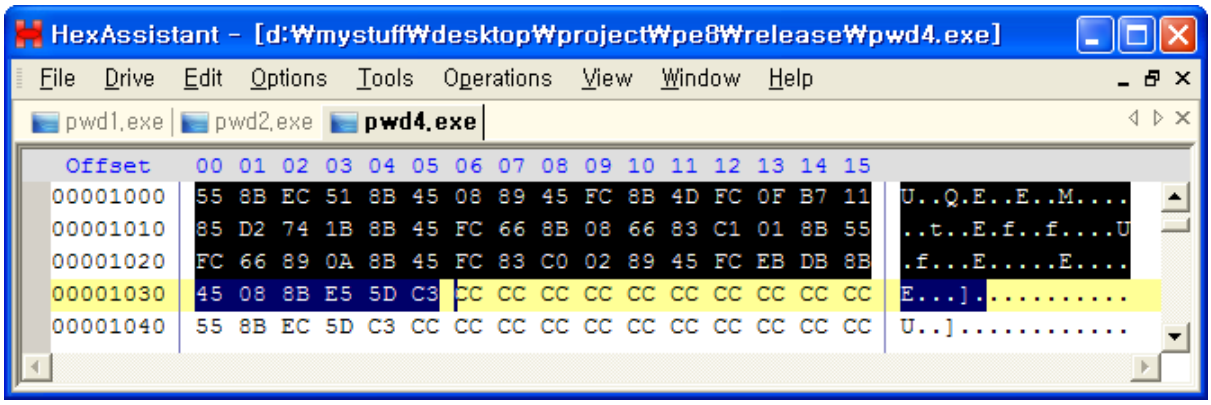
리스트 4 pwd4 소스 코드

```
LPTSTR Encrypt(LPTSTR str)
{
	LPTSTR p = str;
	while(*p) { ++*p; ++p; }
	return str;
}

void EncryptEnd() {}

void DecryptCode(PVOID s, PVOID e)
{
	PBYTE bs = (PBYTE) s;
```

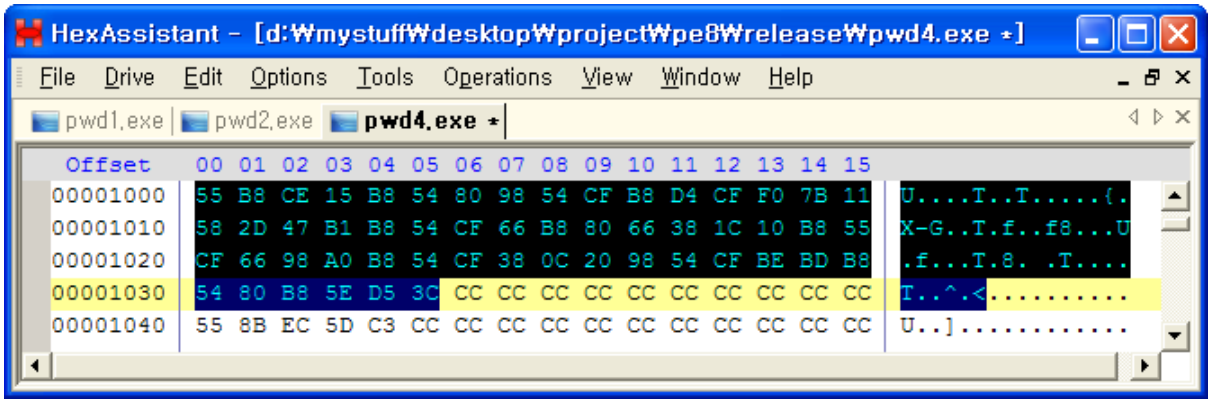
```
	PBYTE be = (PBYTE) e;

	DWORD old;
	DWORD sz = be - bs;
	VirtualProtect(s, sz, PAGE_EXECUTE_READWRITE, &old);

	while(bs < be)
	{
		*bs = (*bs << 4) | (*bs >> 4);
		++bs;
	}

	VirtualProtect(s, sz, old, &old);
}

int main()
{
	const DWORD BUF_SIZE = 80;
	TCHAR buf[BUF_SIZE + 1];
	DWORD sum;

	CODE_CHECKSUM(sum, ic_start, ic_end);
	if(sum != 0x2b75)
		return 0;

ic_start:
	DecryptCode(Encrypt, EncryptEnd);
	_tprintf(_T("Input password => "));
	_fgetts(buf, BUF_SIZE, stdin);
	if(_tcscmp(Encrypt(buf), g_password) == 0)
		_tprintf(_T("ok.₩n"));
ic_end:

	return 0;
}
```

pwd4를 컴파일해서 정상적으로 실행 시키기 위해서는 encrypt의 코드 부분을 직접 손으로 암호

화를 해 주어야 한다. 여기서 코드 암호화에 사용된 방법은 바이트의 상위 4비트와 하위 4비트를 교환해 주는 것이다. 이는 헥사 에디터를 사용하면 손쉽게 할 수 있다. 일단 IDA와 같은 디스어셈블러를 통해서 encrypt의 코드를 찾은 다음, 헥사 에디터를 사용해서 그 부분을 암호화하면 된다 (<화면 5>, <화면 6> 참고). 이렇게 암호화를 한 다음 IDA로 암호화된 파일을 열어보면 <화면 6>에 나타난 것과 같이 엉뚱하게 디스어셈블되서 나타나는 것을 볼 수 있다.



화면 4 pwd4의 원본 Encrypt 코드 부분



화면 5 pwd4의 Encrypt 코드 부분을 암호화한 부분

```
.text:00401000 ; void Encrypt
.text:00401000 Encrypt         proc near               ; CODE XREF: _main+86↓p
.text:00401000                                         ; DATA XREF: _main+47↓o
.text:00401000                 push    ebp
.text:00401001                 mov     eax, 54B815CEh
.text:00401006                 sbb     byte ptr [eax-2B4730ACh], 0CFh
.text:0040100D                 lock jnp short loc_401021
.text:00401010                 pop     eax
.text:00401011                 sub     eax, 54B8B147h
.text:00401016                 iret
.text:00401016 ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴
.text:00401017                 db 66h
.text:00401018                 dd 386680B8h, 55B8101Ch
.text:00401020                 db 0CFh
.text:00401021 ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴
.text:00401021
.text:00401021 loc_401021:                             ; CODE XREF: Encrypt+D↑j
.text:00401021                 cbw
.text:00401023                 mov     al, ds:38CF54B8h
.text:00401028                 or      al, 20h
.text:0040102A                 cwde
.text:0040102B                 push    esp
.text:0040102C                 iret
.text:0040102C Encrypt         endp ; sp = -8
.text:0040102C
.text:0040102C ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴
.text:0040102D                 db 0BEh, 0BDh, 0B8h
.text:00401030                 dd 5EB88054h, 0CCCC3CD5h, 2 dup(0CCCCCCCCh)
```

화면 6 pwd4의 암호화된 Encrypt 부분을 디스어셈블한 화면

5레벨: 안티 덤프

똑똑한 해커라면 <화면 6>과 같은 엉터리 코드를 보는 순간 이건 뭔가 잘못됐다는 것을 눈치챈다. 그런 영리한 해커들의 첫 번째 도우미는 메모리 덤프 유틸리티다. 보통 대부분의 프로그램이 pwd4와 같이 메모리 상에서는 압축이나 암호화가 해제된 상태로 있기 때문에 덤프를 사용하면 원본 코드를 그대로 볼 수 있는 것이다. pwd4 또한 덤프해서 IDA로 분석해보면 원본 코드를 그대로 볼 수 있다. 따라서 암호화를 했다면 당연히 덤프를 방지하는 것이 다음 수순이라 할 수 있겠다.

지피지기면 백전백승이란 말처럼 덤프를 막기 위해서는 우선 덤프에 대해서 정확하게 알 필요가 있다. 덤프란 메모리를 읽어서 그대로 저장하는 것을 말한다. 당연히 아무 메모리나 저장하는 것은 아니다. 덤프 대상이 되는 프로세스의 섹션 부분만 저장하는 것이다. 여기서 필요한 메모리 영역을 찾는 방법이 크게 두 가지가 있다. 하나는 PE 파일의 헤더 정보를 읽어서 덤프를 뜨는 것이고, 다른 하나는 VirtualQuery와 같은 API를 사용해서 연결된 섹션 정보를 수집해서 덤프를 뜨는 것이다.

PE 포맷의 헤더 정보를 사용해서 덤프를 뜨는 경우에는 <리스트 5>에 나와 있는 것과 같이 PE 헤더 정보를 삭제함으로써 덤프를 방지할 수 있다. 유사한 방법을 사용하는 PE Tools와 같은 유틸리티는 anti_dump 프로그램의 덤프를 생성하지 못한다.

리스트 5 anti_dump 소스

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

PVOID GetPtr(LPCVOID addr, SIZE_T offset)
{
	return (PVOID)((DWORD_PTR) addr + offset);
}

int main()
{
	HANDLE module = GetModuleHandle(NULL);

	MEMORY_BASIC_INFORMATION mbi;
	VirtualQuery(module, &mbi, sizeof(mbi));
	DWORD old;
	BOOL b = VirtualProtect(module
				, mbi.RegionSize
				, PAGE_READWRITE
				, &old);

	PIMAGE_DOS_HEADER dos;
	PIMAGE_NT_HEADERS nt;
	PIMAGE_SECTION_HEADER sec;

	dos = (PIMAGE_DOS_HEADER) module;
	nt = (PIMAGE_NT_HEADERS) GetPtr(module, dos->e_lfanew);
	sec = (PIMAGE_SECTION_HEADER) GetPtr(nt, sizeof(*nt));

	for(int i=0; i<nt->FileHeader.NumberOfSections; ++i)
		sec[i].Misc.VirtualSize = 0;
	nt->FileHeader.NumberOfSections = 0;

	for(;;)
	{
```

```
            printf("hello₩n");
            Sleep(1000);
        }

        return 0;
}
```

<리스트 5>와 같은 PE 헤더 정보를 삭제하는 것은 만들기는 간단하지만 보다 정교한 덤프 유틸리티에게는 무용지물이다. 좀 더 귀찮더라도 근본적으로 덤프에 대응할 수 있는 방법이 필요하다. 덤프는 실행 파일과 관련된 내용만 저장하는 것이라고 설명을 했었다. 따라서 코드가 해당 섹션에만 존재하지 않는다면 덤프가 되지 않는다. 1월호에 소개한 것처럼 코드를 힙, 스택, 내지는 실행 시점에 할당된 가상 메모리에 두면 되는 것이다. <리스트 6>에 있는 pwd5는 가상 메모리를 사용해서 덤프에 encrypt가 노출되지 않도록 만든 것이다. 헥사 에디터를 사용해서 파일을 직접 수정하는 것이 귀찮기 때문에 앞서 작업했던 Encrypt 함수의 암호화된 바이트 코드를 복사해서 사용했다.

리스트 6 pwd5 소스 코드

```
typedef LPTSTR (*EncryptT)(LPTSTR);

BYTE g_encryptCode[54] =
{
    0x55, 0xB8, 0xCE, 0x15, 0xB8, 0x54, 0x80, 0x98
    , 0x54, 0xCF, 0xB8, 0xD4, 0xCF, 0xF0, 0x7B, 0x11
    , 0x58, 0x2D, 0x47, 0xB1, 0xB8, 0x54, 0xCF, 0x66
    , 0xB8, 0x80, 0x66, 0x38, 0x1C, 0x10, 0xB8, 0x55
    , 0xCF, 0x66, 0x98, 0xA0, 0xB8, 0x54, 0xCF, 0x38
    , 0x0C, 0x20, 0x98, 0x54, 0xCF, 0xBE, 0xBD, 0xB8
    , 0x54, 0x80, 0xB8, 0x5E, 0xD5,0x3C
};

class CRuntimeCode
{
private:
    PVOID m_ptr;

public:
    CRuntimeCode(PVOID buf, SIZE_T size)
```

```
    {
        m_ptr = VirtualAlloc(NULL
                             , size
                             , MEM_RESERVE | MEM_COMMIT
                             , PAGE_EXECUTE_READWRITE);
        if(!m_ptr)
            throw GetLastError();

        CopyMemory(m_ptr, buf, size);
        DecryptCode(m_ptr, (PBYTE) m_ptr + size);
    }

    ~CRuntimeCode()
    {
        VirtualFree(m_ptr, 0, MEM_RELEASE | MEM_DECOMMIT);
    }

    PVOID Code()
    {
        return m_ptr;
    }
};

int main()
{
    const int BUF_SIZE = 80;
    TCHAR buf[BUF_SIZE + 1];
    DWORD sum;
    EncryptT encrypt;

    CODE_CHECKSUM(sum, ic_start, ic_end);

    if(sum != 0x84f3)
        return 0;

ic_start:
    CRuntimeCode rc(g_encryptCode, sizeof(g_encryptCode));
```

```
	encrypt = (EncryptT) rc.Code();

	_tprintf(_T("Input password => "));
	_fgetts(buf, BUF_SIZE, stdin);
	if(_tcscmp(encrypt(buf), g_password) == 0)
		_tprintf(_T("ok.₩n"));
ic_end:

	return 0;
}
```

6레벨: 안티 디버깅

궁극의 도구가 등장할 때가 되었다. 해커의 가장 강력한 무기인 디버거가 그것이다. 디버거는 실행 시점에 프로그램과 관련된 모든 사항을 모니터링 할 수 있도록 만들어졌다. 버그를 잡기 위한 용도로 개발된 것이지만 바이너리 코드를 분석하는데도 더할 나위 없이 좋은 툴이다. pwd5도 이러한 디버깅 앞에서는 무력화되고 만다. 코드를 차근차근 따라가면 결국은 Encrypt 함수를 만나게 되기 때문이다.

디버거에 대응하는 방법은 연구가 아주 많이 이루어진 분야 중에 하나다. 구글에서 anti debugging과 같은 검색어로 검색해보면 일일이 방문하기도 힘들만큼 많은 정보를 얻을 수 있다. 그 중에 대표적인 세 가지 방법에 대해서 알아보도록 하자.

첫 번째 방법은 윈도우 API를 사용하는 방법이다. IsDebuggerPresent라는 함수로 현재 프로세스가 디버깅 중이면 TRUE를 그렇지 않다면 FALSE를 반환한다. 따라서 디버거를 검출하고 싶은 시점에 IsDebuggerPresent 함수를 호출해서 디버거가 동작 중인지를 체크하면 된다. 워낙 잘 알려진 방법이라 디버거를 사용하는 해커들이 대부분 영순위로 무력화하는 함수 중의 하나이기도 하다. 참고자료에 이와 유사한 다른 여러 가지 방법에 대한 소개가 있다. 궁금한 독자들은 해당 내용을 읽어보도록 하자.

리스트 7 IsDebuggerPresent를 사용한 디버거 탐지 방법

```
if(IsDebuggerPresent())
	ExitProcess(0);

// 중요 코드
```

두 번째 방법은 디버거의 주된 기능인 트레이스를 탐지하는 방법이다. 디버거의 싱글 트레이스 기능은 소프트웨어 브레이크 포인터인 0xCC(int 3)을 통해서 구현된다. 따라서 메모리의 중요한

코드 영역을 스캔해서 0xCC가 있는지를 체크해보면 지금 브레이크 포인터가 걸려있는지 알 수 있다. 앞서 만들었던 CODE_CHECKSUM을 주기적으로 실행해주는 것도 한 방법이 될 수 있다. 어차피 디버거가 0xCC로 메모리를 덮어쓰면 체크섬 값이 바뀌기 때문이다.

리스트 8 소프트웨어 브레이크 포인터(0xCC) 체크를 통한 디버거 탐지 방법

```
PYBTE s, e;

__ams mov eax, ic_start
__asm mov s, eax
__asm mov eax, ic_end
__asm mov e, eax

while(s < e)
{
	if(*s == 0xCC)
		ExitProcess(0);
	++s;
}

ic_start:
// 중요 코드
ic_end:
```

세 번째 방법은 디버거로 트레이스할 경우에 발생하는 속도 저하를 탐지하는 방법이다. 사람이 직접 키를 눌러서 트레이스를 하기 때문에 CPU에서 그냥 수행하는 것과는 속도면에서 엄청난 차이가 발생한다. 중요한 연산 코드의 수행 속도를 측정해서 일정 속도 이상으로 느리다면 현재 디버깅 중이라고 판단하는 것이다.

리스트 9 시간 측정을 통한 디버거 탐지 방법

```
DWORD s = GetTickCount();
// 중요 코드
DWORD e = GetTickCOunt();
if(e - s > CODE_TIME_LIMIT)
	ExitProcess(0);
```

이러한 방법 외에도 디버거에 대항하는 효과적인 방법의 하나로 멀티스레드를 사용한 방법이 있다. 아직까지 멀티 스레드의 실행 흐름을 완벽하게 재현할 수 있는 디버거는 존재하지 않는다. 모

든 디버거의 싱글 스텝은 기본적으로 하나의 스레드를 대상으로 이루어진다. 따라서 의도적으로 실행 흐름을 분리된 스레드로 나눈 다음 병렬적으로 처리하면 트레이스에 효과적으로 대응할 수 있다. <리스트 10>에는 이러한 안티 디버깅이 적용된 pwd6의 소스 코드가 나와 있다. <리스트 6>과 같이 SetEvent를 통해서 흐름만 분절시킨 경우도 단순한 트레이스로는 연결 과정을 찾기가 쉽지 않다.

리스트 10 pwd6 소스 코드

```
typedef enum _WORKERCOMMAND
{
	CheckDebugger
	, CheckPassword
	, Exit
} WORKERCOMMAND;

typedef struct _WORKERPARAM
{
	WORKERCOMMAND commandType;
	LPTSTR password;
	HANDLE commandEvent;
	HANDLE completeEvent;
	DWORD rval;
} WORKERPARAM, *PWORKERPARAM;

DWORD CALLBACK WorkerThread(PVOID param)
{
	PWORKERPARAM p = (PWORKERPARAM) param;
	DWORD obj;

	for(;;)
	{
		obj = WaitForSingleObject(p->commandEvent, INFINITE);
		if(obj != WAIT_OBJECT_0)
			break;

		p->rval = 0;
		switch(p->commandType)
		{
```

```
        case CheckDebugger:
            {
                DWORD sum;
                CODE_CHECKSUM(sum, ic_start, ic_end);

                if(IsDebuggerPresent() || sum != 0x52d1)
                {
                    p->rval = 1;
                    SetEvent(p->completeEvent);
                    return 0;
                }
            }
            break;

        case CheckPassword:
ic_start:
            {
                CRuntimeCode rc(g_encryptCode
                                , sizeof(g_encryptCode));
                EncryptT encrypt = (EncryptT) rc.Code();
                encrypt(p->password)

                if(_tcscmp(p->password, g_password) == 0)
                    printf("ok.₩n");
            }
ic_end:
            break;

        case Exit:
            SetEvent(p->completeEvent);
            return 0;

        }

        SetEvent(p->completeEvent);
    }
```

```
	return 0;

}

int main()
{
	const int BUF_SIZE = 80;
	TCHAR buf[BUF_SIZE + 1];
	DWORD sum;
	DWORD tid;
	HANDLE thread;



	CODE_CHECKSUM(sum, ic_start, ic_end);

	if(sum != 0x4799)
		return 0;

ic_start:
	WORKERPARAM wp;

	wp.commandEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
	wp.completeEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
	thread = CreateThread(NULL, 0, WorkerThread, &wp, 0, &tid);
	CloseHandle(thread);

	_tprintf(_T("Input password => "));
	_fgetts(buf, BUF_SIZE, stdin);
	wp.password = buf;

	wp.commandType = CheckDebugger;
	SetEvent(wp.commandEvent);
	WaitForSingleObject(wp.completeEvent, INFINITE);
	if(wp.rval != 0)
		return 0;

	wp.commandType = CheckPassword;
```

```
	SetEvent(wp.commandEvent);
	WaitForSingleObject(wp.completeEvent, INFINITE);
ic_end:

	return 0;
}
```

7레벨: 더미 코드, 랜덤 워크, 가상화

6가지를 모두 충실히 읽은 독자 분들이라면 아마도 이곳, 이곳, 이곳만 패치하면 여전히 무용지물이고 디버거로 트레이스가 가능하다는 생각을 하고 있을 것 같다. 안타까운 현실은 그것이 정답이다. 앞서 서두에 말했듯이 해커로부터 코드를 완벽하게 보호하는 것은 불가능하기 때문이다. 이런 마지막 단계에 봉착한 사람들이 생각해낸 방법은 해커가 사람이라는 특징을 이용한 기법들이다.

가장 먼저 소개할 것은 고전적인 방법으로 쓸 때 없는 코드를 프로그램에 추가하는 방법이다. 이런 코드를 잔뜩 추가함으로써 해커의 시선을 분산시키는 방법이다. 10평짜리 집에 물건을 숨기는 것과 100평짜리 집에 물건을 숨겼을 때 후자가 찾기가 어려운 것은 당연한 사실이다. 또한 이러한 더미 코드가 실제 핵심 코드와 유사한 형태를 보인다면 해커의 시선을 훨씬 더 효율적으로 분산 시킬 수 있다.

두 번째 방법은 랜덤을 사용한 것이다. 머리가 좋은 사람들은 한번 간 길은 금방 기억을 한다. 이러한 똑똑한 사람들에게 혼란을 주기 위해서 랜덤을 사용해서 그 때 그 때 가는 길을 다르게 만드는 것이다. 매번 트레이스 할 때 마다 코드가 달라진다면 해커도 당황할 수 밖에 없을 것이다. 이러한 방법은 주로 코드 제너레이터를 탑재하고 실행 시점에 랜덤한 코드를 마구 생성하는 기법을 사용한다.

끝으로 이러한 기술의 종착역은 가상화로 귀착된다. x86 어셈블리가 아닌 독자적인 포맷의 명령어를 사용한다면 해커가 그것을 해석하기가 무척 까다롭고 트레이스 하기도 쉽지 않기 때문이다. 어셈블리 명령어를 거의 실시간으로 고급 언어의 코드로 재현하는 해커라 하더라도 가상화된 코드를 금방 분석하기는 힘들다. 또한 최신의 가상 엔진들은 자체 변형(metamorphic) 기능을 탑재하고 있기 때문에 매번 코드 세트가 변형되기도 한다.

도전 과제

다음 시간에는 이번 시간에 배운 지식들을 토대로 간단한 실행 파일 프로텍터를 제작해 볼 것이다. 어떤 부분을 어떻게 자동화 할 수 있을지 한번 고민해 보도록 하자. 또한 자신만의 새로운 실행 파일 보호 기법에 대해서도 한번씩 생각해보는 시간을 가져보도록 하자.

참고자료

Windows Anti-Debug Reference
http://www.securityfocus.com/infocus/1893

"Assembly Language For Intel-Based Computers 4/e"
KIP R. IRVINE저, Prentice Hall

A Crash Course on the Depths of Win32™ Structured Exception Handling
http://www.microsoft.com/msj/0197/exception/exception.aspx