윈도우 프로그래머를 위한 PE 포맷 가이드
진화하는 코드

목차

저작권

소개

함수와 프로그램은 같은 곳에 있어야 하는 것일까? 함수가 실행 파일이나 DLL 밖으로 나온다면 어떨까? 필요할 때만 생성해서 사용하는 함수를 만들 수는 없을까? 카멜레온처럼 그 때 그 때 함수 코드가 바뀌도록 할 수는 없을까? 이번 시간에는 이러한 발칙한 상상들에 대한 해답을 찾아본다.

연재 가이드

운영체제: 윈도우 2000/XP
개발도구: Visual Studio 2005

기초지식: C/C++, Win32 API, Assembly
응용분야: 보안 프로그램

연재 순서

2007. 08. 실행파일 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

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

없음.

Introduction

도둑을 잡기 어려운 이유는 모두 다른 모습을 하고 있기 때문이다. 얼굴에다 도둑이라고 써놓고 다니는 도둑은 없다. 반면에 경찰을 알아보기 쉬운데 그 이유는 경찰이라고 써진 똑같은 제복을 입고 있기 때문이다.

컴파일러가 생성해낸 딱딱한 코드는 변하지 않는다. 매번 똑같은 과정을 되풀이하고 똑같은 결과물을 내놓는다. 보통은 큰 문제가 없어 보이는 이 사소한 사실에도 해커가 개입하면 문제가 달라진다. 그들은 똑같은 코드를 손쉽게 분석해서 금방 패치해 버리기 때문이다. 코드 패칭이 얼마나 쉬운 작업인지는 이미 지난 연재를 통해서 소개한 적이 있다. 그렇다면 이렇게 손쉽게 이루어지는 코드 패칭에 대응할 수 있는 방법은 없을까? 있다. 바로 도둑이 되는 것이다. 코드가 매번 바뀐다면 그들이 분석하기가 한결 어려워질 것이기 때문이다.

해커가 코드를 공격하는 기법은 크게 정적 분석과 동적 분석으로 나눌 수 있다. 정적 분석은 프로그램을 실행하지 않고 디스어셈블러가 분석해낸 결과물을 토대로 프로그램을 살펴보는 것이다. 반대로 동적 분석은 프로그램을 실행시킨 다음 디버거를 사용해서 실행되는 과정을 따라가면서 프로그램을 살펴보는 과정으로 이루어진다. 이번 시간에 소개하는 함수 추출 기법을 사용하면 정

적 분석 과정을 어렵게 만들 수 있고, 코드 변형 기법을 사용하면 동적 분석을 어렵게 할 수 있다.

함수의 시작과 끝

우리가 앞으로 진행할 논의의 모든 중심에는 함수가 있다. 복사할 코드, 추출할 코드, 변경할 코드 모두 기본은 함수 단위이다. 함수 단위가 기본이 되는 이유는 그것이 C에서 의미를 가지는 가장 기본적인 코드 집합이기 때문이다.

함수와 관련된 작업을 하기에 앞서서 가장 먼저 해야 할 일은 함수의 시작과 끝을 알아내는 것이다. 그래야 어떤 부분을 복사할지 변경할 지가 결정되기 때문이다. 함수의 시작은 모두 잘 알고 있는 것처럼 함수 이름 그 자체이다. C언어에서 함수 이름은 자신의 메모리 시작 위치를 가리키는 포인터이기 때문이다. 하지만 컴파일러에서 제공하는 정보가 없어서 함수의 끝을 알아내는 작업은 쉽지 않다. 결국은 컴파일러가 코드를 생성하는 특징을 사용해서 끝을 찾아내는 방법 밖에는 없다. 대표적인 세 가지 방법에 대해서 알아보도록 하자.

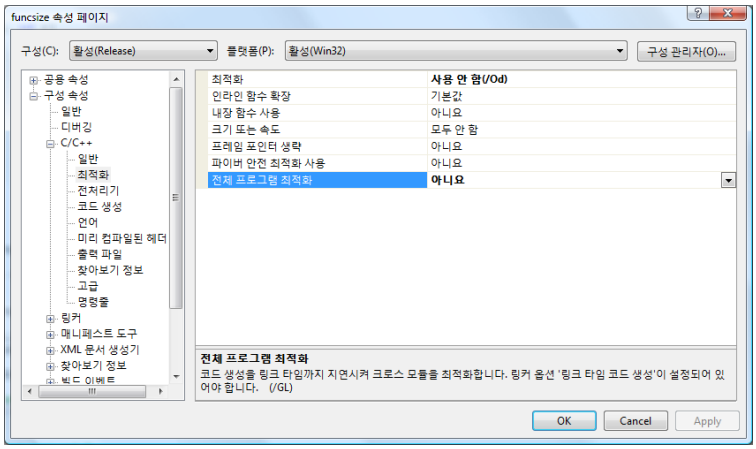
한 함수의 시작은 다른 함수의 끝을 의미한다라는 사실을 적극 활용하는 것이 첫 번째 방법이다. 컴파일러는 파일을 시작부터 끝까지 순서대로 내려 가면서 번역을 한다. 따라서 위쪽에 있는 내용이 메모리 상에 낮은 주소에 맵핑되고, 아래쪽으로 갈 수록 높은 주소가 된다. 결국 한 함수의 끝은 바로 다음에 시작하는 함수의 시작이 되는 셈이다. <리스트 1>에 이러한 특징을 사용한 GetFunctionSize0 함수의 코드가 나와있다. GetGrade 함수는 길이를 측정할 대상 함수이고, GetGradeEnd 함수는 길이를 측정하기 위해서 임의로 추가한 함수이다. GetGrade 함수의 끝은 GetGradeEnd의 시작이 되기 때문에 GetGrade 함수의 길이는 GetGradeEnd – GetGrade가 된다.

리스트 1 종료를 알리는 함수를 사용해서 길이를 측정하는 코드

```
char GetGrade(int score)
{
   char grade = 'F';

   if(score >= 90) grade = 'A';
   else if(score >= 80) grade = 'B';
   else if(score >= 70) grade = 'C';
   else if(score >= 60) grade = 'D';

   return grade;
}

SIZE_T GetFunctionSize0(PVOID addr, PVOID end)
{
   return (SIZE_T) end - (SIZE_T) addr;
```

```
}
```

이 방법을 테스트할 때 주의해야 할 점은 반드시 Release 버전으로 해야 한다는 점이다. 이것은 다른 방법에도 공통된 내용이다. 이유는 VC++ 컴파일러의 경우 Debug 버전으로 컴파일할 경우에는 함수의 주소가 실제 함수 내용을 가리키고 있지 않기 때문이다. 그리고 <화면 1>에 나타난 것과 같이 프로젝트 속성을 조절해 주는 것이 좋다. 최적화 옵션을 켜 둘 경우 일부 함수가 인라인 형태로 변형되거나 링크에서 참조가 없어서 빠지는 경우가 생길 수 있다. 또한 전체 프로그램 최적화에서 링크 타임 코드 생성 기능을 사용하면 앞서 우리가 가정한 컴파일이 파일 처음부터 끝으로 가면서 진행된다는 사실이 무용지물이 된다. 컴파일된 코드의 배치가 링크 타임에 다시 뒤죽박죽이 되기 때문이다.



화면 1 전체 프로그램 최적화 옵션

이 방법의 가장 큰 장점은 앞선 제약 조건만 지킬 경우에는 어떤 함수에 대해서도 길이 측정을 할 수 있다는 점이다. 반면에 단점으로는 불필요한 길이 체크를 위한 함수를(CheckPasswordEnd) 선언해야 한다는 점과 컴파일 조건이 까다롭다는 점과 컴파일러가 함수 사이사이에 패딩을 위해 집어넣는 0xCC(INT 3)까지 길이로 측정된다는 점이다.

두 번째 방법은 함수의 끝을 나타내는 표시를 찾는 방법이다. 함수 끝을 나타내는 표시가 무엇일까? 바로 리턴(ret)이다. 여기에 또한 컴파일러가 생성하는 코드에 대한 가정이 들어간다. 두 가지 가정은 항상 컴파일러는 함수에 대해서 스택 프레임을 생성한다는 것과 단일 진입점, 단일 종료점(single entry, single exit) 원칙을 준수한다는 점이다. 이 두 가지 사실을 기준으로 함수의 끝을 찾아내는 코드가 <리스트 2>에 나와있다. GetFunctionSize1 함수는 두 가지 형태의 종료 코드를 체크한다. 각각 __cdecl과 __stdcall 함수에 대한 종료 코드이다. 최적화 옵션을 켤 경우에는 이보다 좀 더 다양한 종료 코드가 생성된다. 이 방법의 가장 큰 단점은 명령어를 정확하게 인지하지 못하기 때문에 중간에 주소로 나오는 데이터를 종료 표시로 오인할 수 있다는 점이다. 실제로 GetFunctionSize1 함수의 경우 길이를 측정할 함수 중간에 0x8be55dc2란 주소가 나타난다면 그곳을 함수의 끝으로 판단해버린다.

리스트 2 종료 코드 검사

```
SIZE_T GetFunctionSize1(PVOID addr)
{
  const UINT MAX_SEARCH_DEPTH = 0x1000;
  PBYTE mem = (PBYTE) addr;

  for(UINT i=0; i<MAX_SEARCH_DEPTH; ++i)
  {
    // mov esp, ebp; pop ebp; ret
    if(mem[0] == 0x8b && mem[1] == 0xe5
      && mem[2] == 0x5d && mem[3] == 0xc3)
      return i + 4;

    // mov esp, ebp; pop ebp; ret N
    if(mem[0] == 0x8b && mem[1] == 0xe5
      && mem[2] == 0x5d && mem[3] == 0xc2)
      return i + 6;

    ++mem;
  }

  return 0;
}
```

세 번째 방법은 두 번째 방법을 보다 정교하게 수정한 버전이다. 바이너리 데이터를 단순하게 비교하지 않고 디스어셈블리 엔진을 사용해서 코드 단위로 조각 낸 다음 비교하는 것이다. 따라서 두 번째 방법의 단점이었던 오진하는 문제가 없다는 점을 장점으로 꼽을 수 있다. <리스트 3>에는 이 방법을 사용한 GetFunctionSize2 함수가 나와있다. 0xC2와 0xC3는 모두 ret에 대한 명령어 코드이다. 코드에 사용된 디스어셈블리 엔진은 PVDasm으로 참고자료에 있는 URL에서 무료로 다운로드 받을 수 있다.

리스트 3 디스어셈블러를 사용한 방법

```
SIZE_T GetFunctionSize2(PVOID addr)
{
  DISASSEMBLY disasm;
  PBYTE codes = (PBYTE) addr;

  disasm.Address = 0;
  FlushDecoded(&disasm);
```

```
  const DWORD MAX_SEARCH_DEPTH = 0x1000;
  for(DWORD i=0; len < MAX_SEARCH_DEPTH; ++i)
  {
    Decode(&disasm, (char *) codes, &i);
    if(codes[disasm.Address] == 0xC3
      || codes[disasm.Address] == 0xC2)
      return len;

    disasm.Address += disasm.OpcodeSize + disasm.PrefixSize;
    FlushDecoded(&disasm);
  }

  return 0;
}
```

함수 실행 시키기

함수가 복사된 장소에 따라서 실행 시키는 방법을 세 가지로 나눌 수 있다. 스택, 힙, 가상 메모리가 그것이다. 각각의 방법으로 앞서 소개한 GetGrade 함수를 실행시키는 코드가 <리스트 4>, <리스트 5>, <리스트 6>에 나와있다.

리스트 4 스택에서 함수 실행

```
GetGradeT func;
SIZE_T len = GetFunctionSize2(GetGrade);

BYTE stack[512];
memcpy(stack, GetGrade, len);
func = (GetGradeT)(PVOID) stack;
printf("%c₩n", func(50));
```

리스트 5 힙에서 함수 실행

```
PBYTE heap;
heap = new BYTE[len];
memcpy(heap, GetGrade, len);
func = (GetGradeT)(PVOID) heap;
printf("%c₩n", func(60));
```

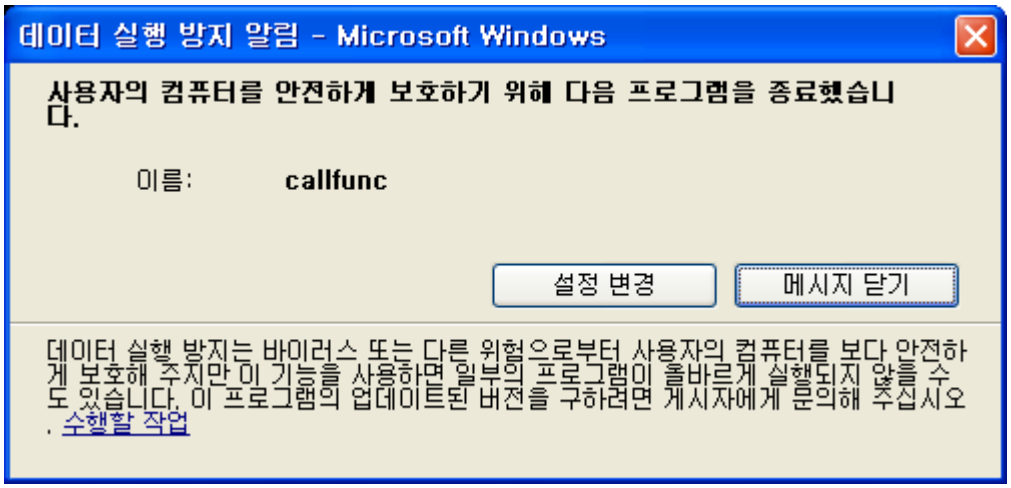
리스트 6 가상 메모리에서 함수 실행

```
PVOID vm;
vm = VirtualAlloc(NULL, len, MEM_COMMIT | MEM_RESERVE
```

```
                        , PAGE_EXECUTE_READWRITE);
memcpy(vm, GetGrade, len);
FlushInstructionCache(GetCurrentProcess(), vm, len);
func = (GetGradeT) vm;
printf("%c₩n", func(60));
```

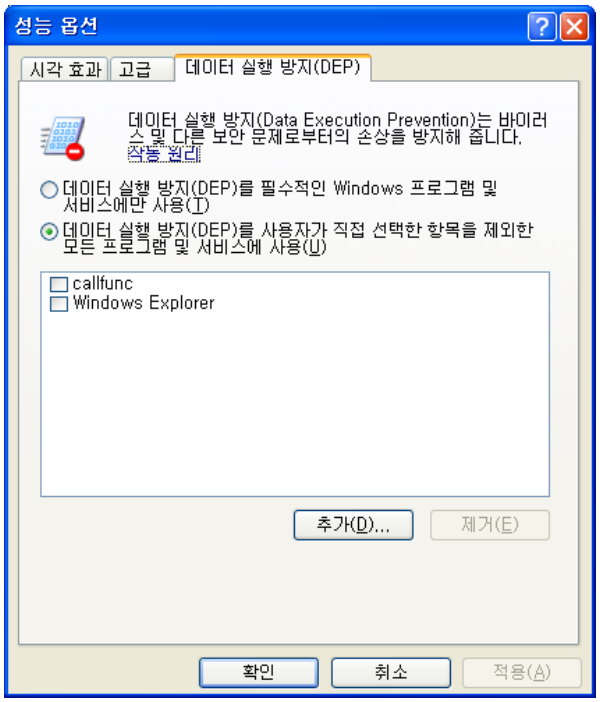
간단한 코드이기 때문에 별도의 설명은 필요 없을 것 같다. 대신 스택과 힙에서 코드를 실행시킬 경우에 발생할 수 있는 문제점에 대해서 살펴보도록 하자. 많은 수의 바이러스나 웜, 쉘코드는 힙이나 스택 오버플로를 사용해서 자신의 코드를 실행시킨다. 이런 문제를 해결하기 위해서 Windows XP SP2 이후부터는 DEP(Data Execution Prevention)이란 기술이 개발되었다. DEP은 실행 퍼미션이 없는 메모리(주로 스택과 힙)에서 코드가 실행되는 것을 방지하기 위한 기술이다.

DEP 기술이 적용된 컴퓨터에서 <리스트 4>나 <리스트 5>의 코드를 실행 시키면 <화면 2>에 나타난 것과 같은 대화 창이 뜬다. 여기서 메시지 닫기를 선택하면 프로그램은 잘못된 연산 오류와 함께 실행이 종료된다. 이는 운영체제가 실행 속성이 없는 메모리에서 코드가 수행되는 것을 감지하고 예외를 발생시키기 때문이다.



화면 2 DEP 확인 창

DEP을 설정하는 방법은 제어판의 시스템에서 할 수 있다. 시스템을 선택한 다음 성능 옵션을 선택하면 <화면 3>과 같이 데이터 실행 방지(DEP)란 탭이 나타나는 것을 볼 수 있다. 라디오 버튼을 선택해서 적절한 DEP 옵션을 지정한 다음 재부팅을 하면 DEP 기능이 활성화된다. DEP에 대한 보다 자세한 설명은 참고자료를 살펴보도록 하자.



화면 3 DEP 설정 화면

<리스트 6>에는 FlushInstructionCache란 생소한 함수 하나가 보인다. FlushInstructionCache 함수는 CPU의 명령어 캐시를 비우도록 하는 명령이다. 근래의 CPU는 대부분 캐시 기술을 사용해서 CPU 성능을 극대화 시킨다. 보통의 경우에 특별한 처리를 하지 않아도 이 명령어 캐시는 문제없이 잘 동작한다. 하지만 메모리 상의 코드의 내용이 변경되는 경우에는 CPU의 명령어 캐시에 저장된 내용과 메모리에 저장된 내용 사이에 불일치 문제가 발생할 수 있다. 이 문제를 해결하는 간단한 방법은 더 이상 명령어 캐시를 참조하지 않도록 캐시를 비우는 것이다. <리스트 6>에서는 그러한 용도로 FlushInstructionCache 함수가 사용되었다.

추출된 함수 코드

지금까지 함수를 복사하기 위해서 길이를 측정하는 방법과 그것을 토대로 복사해서 실행하는 방법에 대해서 살펴보았다. 메모리 상의 내용을 복사해서 실행시키는 것이나 파일에서 읽어온 내용을 실행시키는 것이나 코드만 동일하다면 똑같은 결과를 나타낸다. 따라서 굳이 함수가 실행 파일 내부에 있을 필요는 없다.

단순히 함수를 추출해서 외부 파일에 두는 것을 넘어서 그것을 난독화시켜 두거나 아니면 다른 데이터와 섞어두면 정적 분석을 더욱 어렵게 만들 수 있다. 설령 해커가 외부 파일에 핵심 코드가 있다는 것을 알아낸다고 해도 단순 디스어셈블러로는 그 코드를 역으로 알아내기가 쉽지 않기 때문이다.

지금의 컴퓨터는 메모리 상에 실행되는 코드와 데이터가 모두 동시에 저장되어 있는 구조를 택하고 있다. 그래서 앞서 살펴보았던 것처럼 데이터 영역에서 코드가 실행되는 문제가 발생하곤 한다. 이러한 특징은 디스어셈블러에도 영향을 미친다. 실제로 코드를 실행해 보기 전까지는 어떤

것이 코드이고 어떤 것이 데이터인지 구분을 해낼 수 없기 때문이다. 여기서 디스어셈블러 제작의 가장 큰 문제인 코드와 데이터 분리가 나오는 것이다. 사실상 현재의 컴퓨터 구조상 실행을 하지 않고 코드와 데이터를 완벽하게 구분해 내기란 불가능에 가깝다.

이러한 디스어셈블러의 맹점을 활용하면 추출된 코드를 더욱 분석하기 어렵게 만들 수 있다. 코드 사이에 데이터를 삽입하는 방법이 대표적이다. <표 1>에는 원본 코드와 함께 0xe9라는 데이터를 중간에 삽입한 코드를 보여주고 있다. 데이터가 삽입된 코드를 디스어셈블해보면 <화면 4>와 같이 전혀 엉뚱한 코드가 나타난다. 0xe9때문에 명령어의 해석 순서가 바뀐 것이다.

표 1 데이터 삽입을 통한 디스어셈블러 교란

원본 코드	데이터 삽입 코드
mov eax, 0 jz $label1 $label1: push 0x12345678 pop eax xor eax, eax	mov eax, 0 jz $label1 _emit 0xe9 $label1: push 0x12345678 pop eax xor eax, eax

```
.text:004113BE B8 00 00 00 00                 mov     eax, 0
.text:004113C3 74 01                          jz      short near ptr loc_4113C5+1
.text:004113C5
.text:004113C5                loc_4113C5:                        ; CODE XREF: .text:004113C3↑j
.text:004113C5 E9 68 78 56 34                 jmp     near ptr 34978C32h
.text:004113CA                ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴
.text:004113CA 12 58 33                       adc     bl, [eax+33h]
.text:004113CD C0 33 C0                       sal     byte ptr [ebx], 0C0h
```

화면 4 데이터가 삽입된 코드를 디스어셈블한 화면

코드가 추출되어 밖으로 나오는 순간 코드는 말랑말랑한 데이터가 된다. 개발자의 상상력에 따라서 코드는 무궁무진한 방법으로 가공될 수 있다. 물론 클라이언트에서 실행되는 코드를 100% 보호할 순 엌지만 더 많이 가공할수록 공격자가 분석하는 과정은 더 힘들어진다는 점을 기억할 필요가 있다.

진화하는 코드

백신이 처음 개발된 직후에 바이러스 개발자들에게는 고민이 생겼다. 힘들게 만든 바이러스가 단순 시그니쳐 기반의 패턴 매칭에 너무도 무력하게 검출되었기 때문이었다. 그래서 바이러스 개발자들은 동일한 바이러스 코드를 다른 형태로 가공하는 방법을 연구하기 시작했고 마침내는 생물체처럼 그 때 그 때 변화하는 코드를 만들어내는 단계에 이르렀다. 코드가 진화하는 수준에 이른 것이다.

아이러니하게도 요즘은 이러한 방법이 보안 소프트웨어에서도 광범위하게 사용된다. 대표적인 경

우가 게임 프로그램이다. 게임 프로그램을 공격하는 해킹 툴의 경우 게임 업데이트에 대응하기 위해서 게임 내에 사용되는 함수 시그니쳐를 사용해서 해당 함수의 주소를 추적하는 기법을 사용한다. 대부분의 핵심 함수는 패치가 되더라도 그 내용이 변경되지 않기 때문이다. 이런 함수들에 코드 변형 기법을 적용하게되면 함수 시그니쳐가 그 때 그 때 바뀌기 때문에 이런 패턴 기반으로 함수를 실행 시점에 찾아내는 것을 불가능하게 만들 수 있다.

코드를 변형시키는 대표적인 세 가지 아이디어에 대해서 소개하고 그것을 실제로 구현하기 위해서 필요한 자료구조를 설계하는 방법에 대해서 살펴보도록 하자.

NOP 노이즈 첨가

NOP 노이즈 첨가 방식은 원본 코드 중간 사이사이 임의로 NOP를 추가적으로 집어넣는 방법이다. 코드 패칭 시간에 소개한 것과 같이 NOP 명령어가 아무 일도 하지 않는다는 특징을 이용한 것이다. <표 2>에 나타난 것과 같이 왼쪽에 있는 원본 코드와 무작위로 코드 사이에 NOP를 추가한 변형된 코드의 실행 결과는 동일하다.

표 2 NOP 첨가

원본 코드	변형된 코드

MOV EAX, 1
XOR EAX, EAX
SUB EAX, 2
	MOV EAX, 1
	NOP
	NOP
	XOR EAX, EAX
	SUB EAX, 2
	NOP

방식이 단순한 만큼 이러한 방법은 손쉽게 변형된 코드를 탐지해 낼 수 있다. 코드에서 NOP를 무시하고 나머지 결과만 취하면 되기 때문이다. 실제로 이런 형태의 바이러스를 탐지하는 백신의 대부분은 첨가된 NOP를 제거하고 나머지 부분만을 시그니쳐로 판독한다.

호출/점프 노이즈 첨가

'모로 가도 서울만 가면 된다'는 말을 실제 코드로 구현한 방법이라고 생각하면 된다. 원본 코드에 임의의 호출이나 점프 노이즈를 첨가하는 방식으로 코드를 변형하는 방법을 말한다. 앞선 NOP 노이즈의 경우 NOP이 제거된 원본 코드를 만들어 내기가 너무 쉽다는 단점을 해결한 버전이라고 생각하면 된다.

표 3 점프 노이즈 첨가 코드

원본 코드	변형된 코드1	변형된 코드2
MOV EAX, 1 XOR EAX, EAX	MOV EAX, 1 **JMP $label1**	MOV EAX, 1 **JMP $label1**

SUB EAX, 2	**$label1:** XOR EAX, EAX SUB EAX, 2	**XOR EAX, EAX** **TEST EAX, EAX** **MOV EAX, 3** **$label1:** XOR EAX, EAX SUB EAX, 2

<표 3>에는 점프 노이즈를 첨가해서 코드를 변형하는 예제가 나와 있다. 변형된 코드1은 단순히 점프 노이즈만을 첨가한 버전이다. 이 버전의 경우에는 NOP과 같이 단순히 JMP $label1 명령어만 제거하면 손쉽게 원본 코드를 얻을 수 있다. 하지만 변형된 코드2는 코드 흐름상 절대 호출이 되지 않는 JMP $label1과 $label1 사이에 쓸모 없는 명령어를 추가로 삽입하는 형태로 변형을 한 것이다. 저 부분에는 어떤 코드를 삽입하더라도 상관 없다. 왜냐하면 절대 실행되지 않을 코드 이기 때문이다. 이와 같이 더미 코드를 삽입해서 흐름을 변경하는 경우에는 코드를 직접 실행하며 추적하기 전까지는 원본 코드와 동일하다는 사실을 알아내기가 쉽지 않다.

유사 코드 변형

어셈블리 명령으로 EAX에 5를 집어넣는 방법을 생각해보자. 과연 몇 가지 방법이 있을까? 셀 수 없이 많은 방법이 있다. 실제 사람들이 쓰는 언어가 가진 것과 같은 특징을 프로그래밍 언어도 가지고 있는 것이다. <표 4>에는 이러한 방법 네 가지가 나와 있다.

표 4 EAX에 5를 집어넣는 네 가지 방법

방법1	방법2	방법3	방법4
MOV EAX, 5	PUSH 5 POP EAX	XOR EAX, EAX OR EAX, 5	XOR EAX, EAX ADD EAX, 4 INC EAX

결국 동일한 일을 하기 때문에 원본 코드에 나타난 이러한 패턴은 서로 교환 될 수 있다. 원본 코드에서 1번 방법에 해당하는 바이트 코드를 발견했다면 그것을 2, 3, 4번 중에 하나로 변형하더라도 동일한 결과를 얻을 수 있는 것이다. 유사 코드 변형이란 이러한 형태로 코드를 변형시켜 나가는 방법을 말한다.

이 방법은 사실상 변형된 코드를 원본 코드로 회귀 시키는 것이 불가능하다. 그래서 요즘 백신의 경우에는 이런 방법에 대응하기 위해서 어셈블리 명령어 코드를 자체 가상 머신 속에서 실행한 결과를 확인하는 형태를 많이 사용한다. 결국 실행 결과는 동일하기 때문이다.

명령어 리스트 생성

앞서 소개한 설명만 보면 코드 변형이 무척 손쉬운 것처럼 보인다. 하지만 사실 실제 구현은 앞서 설명한 것만큼 간단하지 않다. 이유는 코드가 예시된 것처럼 간단한 경우는 거의 없기 때문이

다. 간단하게 흐름을 변형하는 점프 계열의 명령어가 원본 코드에 있는 경우에는 변형하는 과정에서 흐름 자체가 바뀌기 때문이다.

<표 5>에는 앞서 설명한 방법 중 가장 간단한 NOP 추가 방식으로 코드를 변형하는 것을 보여주고 있다. 원본 코드는 EAX가 3과 같으면 ECX에는 0이 들어가고(XOR ECX, ECX), 3이 아니면 ECX에 1이 저장되는 코드다. 이를 NOP을 추가해서 변형한 오른쪽 코드를 보면 NOP이 추가되면서 코드 흐름이 변경되어서 EAX값에 상관 없이 ECX는 0이 되는 코드가 됐다. JNZ $+10 명령어를 JNZ $+12로 변형해 주어야 원본과 동일한 역할을 하는 코드가 된다.

표 5 NOP 추가로 코드 흐름이 변경되는 경우

원본 코드	변형된 코드
CMP EAX, 3	CMP EAX, 3
JNZ $+10	JNZ $+10
XOR ECX, ECX	**NOP**
JMP $end	**NOP**
MOV ECX, 1	XOR ECX, ECX
$end:	JMP $end
	MOV ECX, 1
	$end:

이와 같이 실제 코드는 흐름 제어를 위한 명령어들이 다수 포함되어 있기 때문에 명령어가 삽입되거나 변형되면 그것에 영향을 받는 점프 명령어들도 함께 수정해 주어야 한다. 이러한 것을 추적하기 위해서는 적합한 자료 구조가 필요하다. <그림 1>에는 그런 기능을 할 수 있는 자료구조의 형태가 나와 있다. 각 명령어를 리스트로 연결하고 점프하는 부분들을 별도의 링크를 사용해서 연결한 것이다. 이 구조에 NOP을 추가하면 <그림 2>와 같은 구조가 된다. NOP가 추가되더라도 jmp 링크로 연결된 포인터는 그대로 유지되기 때문에 주소를 새롭게 계산해서 업데이트 할 수 있다.

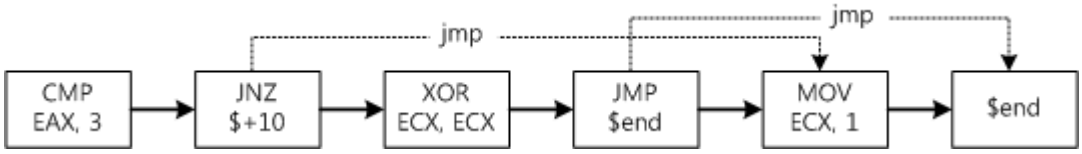


그림 1 원본 코드의 명령어 리스트 구조

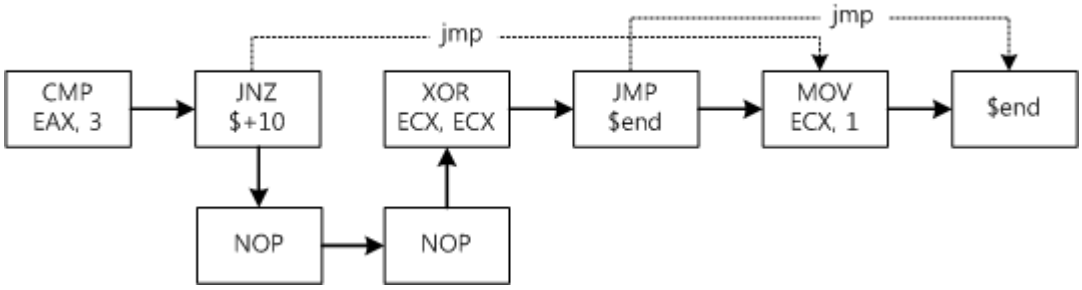


그림 2 변형된 코드의 명령어 리스트 구조

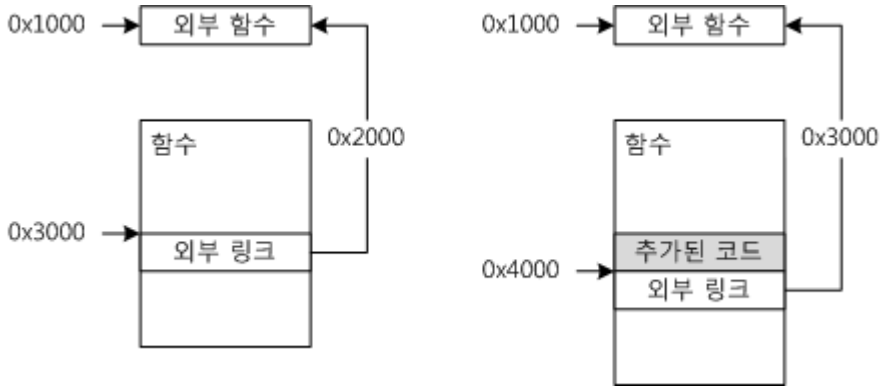


그림 3 외부 링크에 대한 처리

<그림 1>과 <그림 2>에 나와있는 함수 내부적인 링크 외에도 외부 링크에 대한 처리도 생각해야 한다. <그림 3>에는 외부 링크를 처리하는 방법이 나와 있다. 외부 링크란 변경할 함수 외부에 대한 참조가 발생하는 경우를 말한다. 대표적으로 상대 주소를 통해서 다른 함수를 호출하는 경우를 들 수 있다. 외부 대상의 주소는 고정적이기 때문에 그림에 나타난 것처럼 추가된 코드로 인해서 변경된 오프셋만큼 링크 주소를 수정해 주면 된다. 물론 완전 함수의 경우에는 이러한 외부 의존 요소가 전혀 없기 때문에 신경 쓸 필요가 없다.

<그림 1>과 <그림 2>의 각 노드를 나타내는 CInstruction32 클래스가 <리스트 7>에 나와 있다. codeSize, prefixSize, code는 디스어셈블된 실제 정보를 저장하는 역할을 한다. address 필드는 각 노드의 절대 주소를 저장하는 필드다. 추후에 리스트가 변경됐을 때 상대주소를 손쉽게 계산하기 위한 용도로 사용된다. jmp는 점선으로 표시된 점프 노드를 가리키기 위해서 사용된다. STL의 list 구조체의 넣어서 사용할 것이기 때문에 리스트를 위한 next, prev와 같은 링크 노드는 저장할 필요가 없다. 내부 링크 노드(LINK_INTERNAL)의 경우에는 SetLinkAddress를 호출해서 직접 주소를 업데이트 해주어야 하고, 외부 링크 노드(LINK_EXTERNAL)의 경우에는 Address를 수정할 때에 자동적으로 링크 주소가 계산된다.

리스트 7 CInstruction32 클래스

```
class CInstruction32
{
private:
   BYTE m_codeSize; // 코드 사이즈
```

```
  BYTE m_prefixSize; // 프리픽스 사이즈
  BYTE m_mask; // 노드 타입
  BYTE m_code[17]; // 코드 데이터
  class CInstruction32 *m_jmp; // 점프 노드 링크
  DWORD m_address; // 노드 주소
  enum { LINK_EXTERNAL = 1, LINK_INTERNAL = 2 };

  void SetMask(DWORD base, DWORD size)
  {
    m_mask = 0;
    int linkAddress;
    if(!GetLinkAddress(linkAddress))
      return;

    DWORD destAddress = m_address + linkAddress;
    if(destAddress < base || destAddress > base + size)
      m_mask |= LINK_EXTERNAL;
    else
      m_mask |= LINK_INTERNAL;
  }

public:
  CInstruction32(DWORD base
                 , DWORD size
                 , DISASSEMBLY &disasm
                 , PBYTE code)
  {
    m_codeSize = disasm.OpcodeSize;
    m_prefixSize = disasm.PrefixSize;
    memcpy(m_code, code, m_codeSize + m_prefixSize);
    m_address = disasm.Address;
    m_jmp = NULL;
    SetMask(base, size);
  }

  void Address(DWORD address)
  {
    int diff = address - m_address;
    if(diff && IsExternalLinkNode())
```

```
    {
      int linkAddress;
      if(GetLinkAddress(linkAddress))
        SetLinkAddress(linkAddress + diff);
    }

    m_address = address;
  }

  // 점프 명령어의 대상 주소를 구하는 함수
  bool GetLinkAddress(int &address) const
  {
    if((m_code[0] >= 0x70 && m_code[0] <= 0x7F)
        || m_code[0] == 0xeb)
    {
      address = (char) m_code[1];
      return true;
    }
    else if(m_code[0] == 0x0F && m_code[1] >= 0x80
          && m_code[1] <= 0x8F)
    {
      address = *(int *)(m_code + 2);
      return true;
    }
    else if(m_code[0] == 0xe8 || m_code[0] == 0xe9)
    {
      address = *(int *)(m_code + 1);
      return true;
    }

    return false;
  }
};
```

어셈블리 코드를 CInstruction32 노드 리스트로 변환하고 주소 계산을 해주는 CCodeFactory 클래스가 <리스트 8>에 나와 있다. 앞서 소개한 PVDasm 엔진을 사용해서 제작되었다. m_instructions는 CInstruction32 노드를 리스트로 연결해둔 것이다. Attach 멤버 함수는서 어셈블리 코드로부터 m_instructions를 생성해 내는 기능을 한다. BuildLink는 점프 명령어로 이루어진 노드의 jmp 포인터가 적절한 링크를 가리키도록 만드는 일을 하고, FixLinkAddress는 점프 명령어의 상대 주소를

업데이트 하는 기능을 한다. 끝으로 UpdateAddress는 각 노드의 주소 필드를 적절한 값으로 재계산하는 역할을 한다.

리스트 8 CCodeFactory 클래스

```
typedef boost::shared_ptr<CInstruction32> Instruction32Ptr;
typedef std::list<Instruction32Ptr> Instruction32List;
typedef std::list<Instruction32Ptr>::iterator Instruction32Lit;

class CCodeFactory
{
private:
   Instruction32List m_instructions;

public:
   typedef Instruction32Lit iterator;

   void Attach(PBYTE buf, SIZE_T size)
   {
      DISASSEMBLY disasm;
      Instruction32Ptr inst;

      disasm.Address = 0;
      FlushDecoded(&disasm);

      for(DWORD i=0; disasm.Address < size; ++i)
      {
         Decode(&disasm, (char *) buf, &i);

         inst.reset(new CInstruction32(0
                                       , (DWORD) size
                                       , disasm
                                       , buf+disasm.Address));
         m_instructions.push_back(inst);

         disasm.Address += inst->Size();
         FlushDecoded(&disasm);
      }

      BuildLink();
```

```
}

void BuildLink()
{
   iterator it = begin(), e = end(), tmp;

   int jmp = 0;
   for(; it != e; ++it)
   {
      if(!(*it)->IsInternalLinkNode())
         continue;

      if(!(*it)->GetLinkAddress(jmp))
         continue;

      if(jmp > 0)
      {
         tmp = it; ++tmp;
         while(jmp) { jmp -= (*tmp)->Size(); ++tmp; }
      }
      else
      {
         tmp = it; --tmp;
         while(jmp) { jmp += (*tmp)->Size(); --tmp; }
      }

      (*it)->Jmp((*tmp).get());
   }
}

void UpdateAddress()
{
   iterator it = begin(), e = end();

   DWORD address = 0;
   for(; it != e; ++it)
   {
      (*it)->Address(address);
      address += (*it)->Size();
```

```
    }
  }

  void FixLinkAddress()
  {
    iterator it = begin(), e = end();

    DWORD rel;
    for(; it != e; ++it)
    {
      if(!(*it)->IsInternalLinkNode())
        continue;

      rel = (*it)->Jmp()->Address()
            - (*it)->Address() - (*it)->Size();
      (*it)->SetLinkAddress(rel);
    }
  }
};
```

CCodeFactory를 사용해서 코드를 변형하는 과정은 단순하다. Attach 함수로 원본 코드에 대한 명령어 리스트를 만든다. CCodeFactory에 있는 리스트 관련 함수인 begin, end, insert, erase 등을 사용해서 리스트에 코드를 삽입하거나 변형시킨다. 이 작업이 완료되고 나면 UpdateAddress와 FixLinkAddress를 호출해서 점프 명령어에 대한 경로를 변형시킨다.

도전 과제

지면 관계상 실제로 코드를 변형하는 복잡한 예제를 보여주진 못했다. 이달의 디스켓에 NOP을 통한 변형을 시키는 샘플 코드가 들어있기 때문에 참고하도록 하자. 실제로 여기 소개된 방법들을 사용해서 코드를 광범위하게 변형시킬 수 있는 코드 변형 엔진을 만들어 보도록 하자. 유사 코드 변형의 경우에는 유사 코드로 간주할 수 있는 테이블을 만드는 것이 가장 중요한 작업이다.

참고자료

"The Art of Computer Virus Research and Defense"
Peter Szor저, Addison-Wesley Professional

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

PVDasm
http://pvdasm.reverse-engineering.net/index.php?Section=3

데이터 실행 방지 기능에 대한 상세한 설명
http://support.microsoft.com/kb/875352

Data Execution Prevention
http://en.wikipedia.org/wiki/Data_Execution_Prevention