Design and Implementation of Virtualized Code Protection(VCP) For Anti-Reverse Engineering

작성자: 이용일(foryou2008@nate.com)

2008-04-17

Abstract

Virtualized Code Protection(VCP)은 가상화 기술을 이용하여 외부로부터 프로그램의 코드 해석(Reverse Code Engineering)을 어렵게 하는 기술이다.
본 문서는 VCP 에 대한 설계 및 구현 내용을 포함하고 있다.

Table of Index

1. Introduction
    1.1 Virtualized Code Protection?
    1.2 no Full-Emulator. It's a Half-Emulator.

2. Design
    2.1 Virtualized Environment
    2.2 Virtualized Instruction Format
    2.3 Stub Code
    2.4 State Switching
    2.5 Data Structures of VM
    2.6 VM Dispatch Loop
    2.7 Program Counter Mapping Table(SPC-TPC Mapping Table)
    2.8 Base Relocation Problem
    2.9 External Call Problem
    2.10 Virtualized File Structure

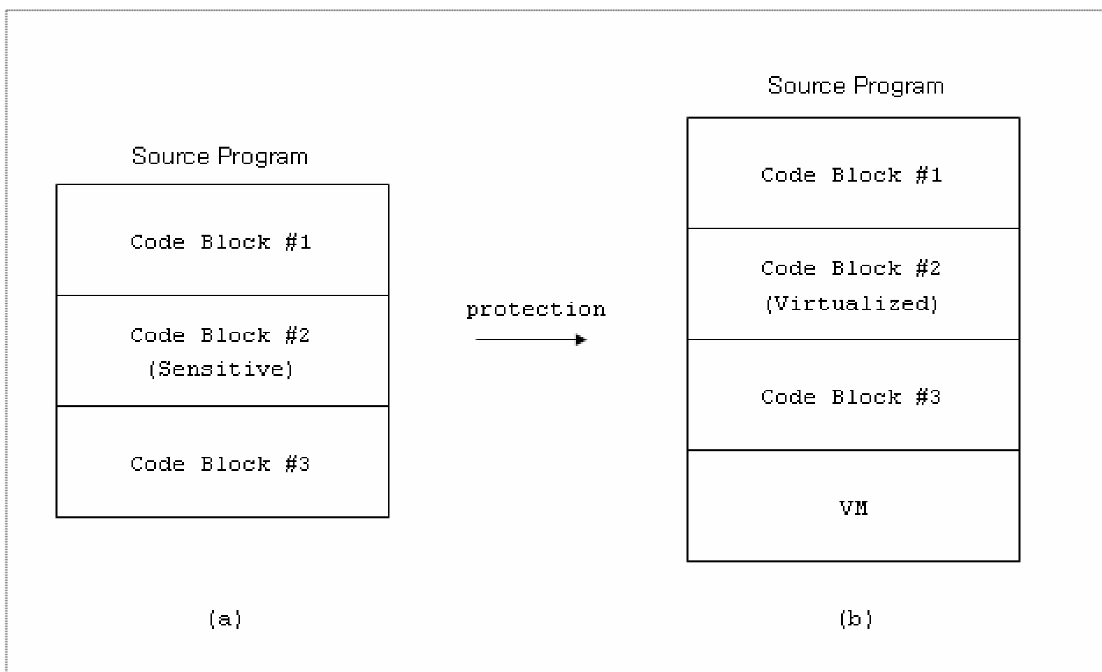
3. Implementation
    3.1 Source.exe
    3.2 Binary Translation
    3.3 The VM
    3.4 The Two Sections
    3.5 The Stub Code
    3.6 I love you

4. References

Chapter 1. Introduction

1.1 VCP(Virtualized Code Protection)?

VCP 는 프로그램의 주요 코드 영역(Sensitive Code Block)을 리버스 코드 엔지니어링으로부터 보호하기 위해, VM 만이 해석할 수 있는 가상화된 코드 영역(Virtualized Code Block)으로 변환(Binary Translation)한다. 변환은 런타임(Dynamic)에 이루어지는 것이 아니라 그 이전(Static)에 이루어진다. 인텔 코드 명령어로부터 변환된 코드는 일반적인 디스어셈블러로는 해석할 수 없다. 변환된 코드 영역의 해석(Decode) 및 실행(Execute)은 CPU 대신 VM 이 처리하게 된다.



[그림 1] VCP를 이용한 변환 과정

VCP 는 인텔 계열의 Windows PE 파일을 대상으로 한다. VCP 는 인텔 명령어 코드에서 변환된 코드와 변환된 코드를 실행(emulation)하기 위한 VM으로 구성된다.
이러한 변환된 코드와 VM은 Windows PE 파일의 일부분(section)으로 어태치(Attach) 된다.

* Virtualized Code 를 가상화된 코드로 표현하려 했으나, 의미 전달에 문제가 있을 것으로 생각되어 변환된 코드로 표현하겠다.

VCP 구현을 위해 기본적으로 고려해야 할 사항은 다음과 같다.

1) 원래의 코드(인텔 코드)를 어떻게 변환할 것인가? (Binary Translation)

2) 변환된 코드는 VM에서 어떻게 실행되는가? (Emulation)

3) 변환된 코드와 VM을 Windows PE 파일에 어떻게 포함시킬 것인가?

4) 어떻게 해야 해석하기 어려운 변환된 코드를 만들 수 있을까?

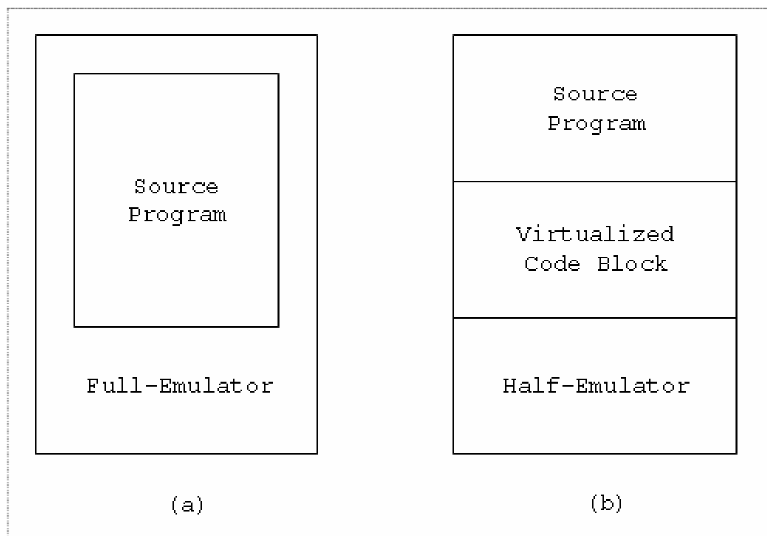
4)의 경우에 대해서는 자세히 다루지 않을 것이다. 코드를 변환하고 변환된 코드를 어떻게 실행할 것인지에 대해 중점적으로 얘기할 것이다. 이것이 기본이 되어야 하기 때문이다. 즉 1) 2) 3)에 대한 내용만 충분히 이해한다면 4)의 요구 사항에 대한 설계는 어렵지 않을 것이다. (변환된 코드의 Opcode만 Randomize하게 설정하도록 설계해도 외부에서 해석하기가 상당히 어려워진다.)

1.2 no Full-Emulator. It's a Half-Emulator.

[그림 2]의 (a)는 Full-Emulator의 모습이다. 일반적인 Emulator는 Full-Emulator와 같은 방식을 사용한다. Full-Emulator 는 소스 프로그램을 Emulation하기 위해 다음과 같은 절차를 따른다.

a. 소스 프로그램을 위한 메모리 할당
b. 할당된 메모리에 소스 프로그램 복사 (소스 프로그램 로딩)
c. 소스 프로그램의 Emulation 진행(Fetch-Decode-Execute)

Full-Emulator의 소스 프로그램은 Emulator의 제어 속에 살고 있다는 현실을 모르고, 착각(illusion) 속에 살게 된다. 이것(illusion)은 가상화의 중요한 특징 중의 하나이다.



[그림 2] Full-Emulator 와 Half-Emulator

반면 VCP는 일반적인 Emulator와는 달리, [그림 2]의 (b)와 같은 반쪽짜리 Emulator (Half-Emulator)의 모습이다. 이것은 소스 프로그램의 전체를 Emulation하는 것이 아니라, 소스 프로그램 중 일부(변환된 코드)만을 Emulation하기 때문에 붙여진 이름이다.

Half-Emulator는 소스 프로그램의 일부(변환된 코드)를 Emulator에서 Emulation 하고, 변환된 코드를 제외한 나머지 소스 프로그램은 CPU 가 직접 실행하는 방식이다.

Full-Emulator가 소스 프로그램의 Supervisor라면, Half-Emulator와 소스 프로그램은 서로 동등한 위치에 있다. 소스 프로그램 실행 중에 변환된 코드 영역을 만나면 Half-Emulator로 CPU 제어권을 넘기며, 변환된 영역의 Emulation이 끝나면 다시 소스 프로그램으로 CPU 제어권을 넘긴다.

Half-Emulator와 소스 프로그램은 서로 동등한 권한을 가지기 때문에, 메모리 주소 공간도 서로 공유한다. 이러한 이유로, 변환된 코드의 Emulation을 진행할 때, Half-Emulator의 메모리 영역에 접근을 시도하는 경우가 발생할 수 있다. 이에 대한 해결책은 2.5 절에서 다룬다.

	Half-Emulator	Full-Emulator
적용	VCP	일반적인 코드 Emulator
소스 프로그램과의 관계	Friend!	Supervisor
소스 프로그램의 로딩	운영체제의 로더가 처리	Emulator에서 수행
소스 프로그램의 실행	CPU에서 직접 실행	Emulation
Isolation	No	Yes

* 참고로 Full-Emulator 및 Half-Emulator 는 설명의 편의를 위해 저자가 지어낸 이름이다.

Chapter 2. Design

2.1 Virtualized Environment

VM은 변환된 코드의 Emulation을 위해 레지스터 블록(Register Block)을 가진다.

* 레지스터 블록은 실제 레지스터를 가리키는 것이 아니라 Emulation을 위해 VM 에서 할당한 메모리 영역이다. 변환된 코드가 레지스터에 접근하는 명령어일 경우, VM은 실제 레지스터에 접근하는 것이 아니라, 레지스터 블록을 참조하도록 Emulation한다. 예를 들어 eax에 0을 저장하는 변환된 명령어가 있을 때, VM은 실제 eax 레지스터에 0을 저장하는 것이 아니라, 레지스터 블록 메모리(레지스터 블록 기준 주소 + EAX 인덱스 * 4)에 0을 저장함으로써 Emulation을 수행한다.

레지스터 블록은 IA-32 아키텍쳐가 가지고 있는 레지스터 셋(Register Set)과
대응(State Mapping)한다.

레지스터 블록

Index	Offset(Byte)	Mapping
0	0	eax
1	4	ebx
2	8	ecx
3	C	edx
4	10	esi
5	14	edi
6	18	ebp
7	1C	esp
8	20	eip
9	24	eflags
A	28	tmp

변환된 코드에서 지정한 메모리 주소는 Emulation 되지 않는다. 변환된 코드에서 가리키는 메모리 0x1000번지는 VM에서의 0x1000번지와 동일하다. 이유는 소스 프로그램 전체에 대한 Emulation을 수행하는 것이 아닌, 변환된 코드에 대해서만 부분적으로 Emulation을 수행하기 때문이다. 따라서 변환된 코드의 Emulation시에 VM 이 위치한 메모리 영역에 접근을 시도하는 문제가 발생할 수 있다.(2.5절에서 자세히 다룬다.)

2.2 Virtualized Instruction Format

기본적으로 RISC 타입을 따른다.
워드 인코딩 방식은 리틀 엔디언(Little Endian) 방식이다.

1 BYTE	1 BYTE	4 BYTE	4 BYTE
Opcode	Operand Info	Operand 1	Operand 2

오퍼랜드를 1 개 이하를 가질 경우, 오퍼랜드 필드는 생략할 수 있다.

1) Opcode
명령어의 ID이다. 1 바이트이므로 총 256개(0 ~ 255)의 Opcode가 존재한다.
리버스 엔지니어링을 좀 더 어렵게 하기 위해 Opcode의 난수화(Randomize)가 필요하다.

2) Operand Info
명령어의 오퍼랜드 주소 지정 방식을 나타낸다.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Ref_Size		Operand1 Info			Operand2 Info		

Ref_Size는 오퍼랜드가 참조하는 값의 크기를 결정한다.

Bit 7	Bit 6	설명
0	0	8 Bit
0	1	16 Bit
1	0	32 Bit
1	1	64 Bit

Operand1 Info와 Operand2 Info는 오퍼랜드 1과 오퍼랜드 2에 대한 주소 지정 방식을 결정한다.

Bit 2	Bit 1	Bit 0	설명
	0	0	Register
	0	1	[Register]

	1	0	[Immediate]
	1	1	Immediate
1			Relocation이 필요하다.

Operand1 Info, Operand2 Info의 최상위 비트는 Relocation 비트로써, 오퍼랜드의 주소가 Relocation 되어야 할 필요가 있을 때 설정된다. (Relocation에 대한 설명은 2.5절에서 다룬다.)

* Relative Address
인텔 계열 명령어에서는 다음 명령어 위치(EIP)로부터 상대적인 주소를 지정할 수 있다.
call rel# 이나 jmp rel# 과 같은 형식으로 사용되는 것을 말한다.
Virtualized Instruction 은 이러한 주소 지정 방식을 지원하지 않기 때문에 call rel# 이나 jmp rel# 과 같은 명령어의 오퍼랜드를 모두 absolute하게 변환해야 한다.

* SIB(Scale-Index-Base) 주소 지정 방식
인텔 계열 명령어에서는 [베이스 + 인덱스 * 스케일]와 같은 SIB 형식을 지원한다.
Virtualized Instruction 은 이러한 주소 지정 방식을 지원하지 않는다.
따라서 이러한 주소 지정 방식을 가진 인텔 명령어의 경우, 다음과 같은 규칙을 통해 변환을 수행한다.

인텔 명령어의 주소 지정 모드 변환 규칙

인텔 Instructions	Virtualized Instructions

명령어 레지스터1, [레지스터2 + disp#]	mov tmp, 레지스터2
	add tmp, disp#
	명령어 레지스터1, [tmp]

명령어 레지스터1,
[레지스터2 + 레지스터3 * 스케일]
	mov tmp, 레지스터3
	mul tmp, 스케일
	add tmp, 레지스터2
	명령어 레지스터1, [tmp]

인텔 명령어의 주소 지정 모드 변환 규칙 적용 예

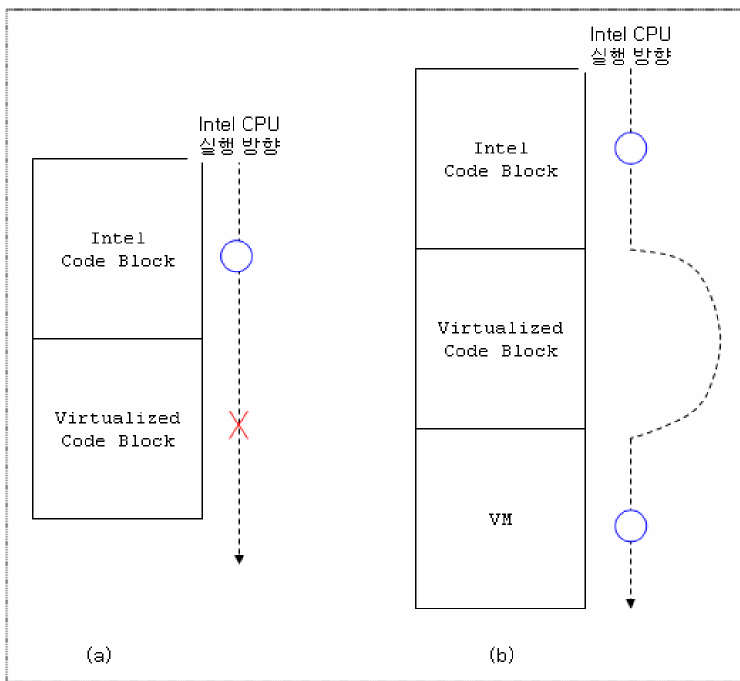
인텔 Instructions	Virtualized Instructions

mov eax, [edi + 4]	mov tmp, edi
	add tmp, 4
	mov eax, [tmp]

mov eax, [edi + esi * 4]	mov tmp, esi
	mul tmp, 4
	add tmp, edi
	mov eax, [tmp]

2.3 Stub Code

VCP 로 보호된 프로그램은 소스 프로그램(인텔 코드)과 변환된 코드가 공존(Hybrid)하는 형태를 가진다. 이러한 Hybrid한 프로그램의 실행을 위해서는 명령어의 해석(Decode) 및 실행(Execute) 주체가 2 가지로 구분되어야 한다. 즉 소스 프로그램(인텔 코드)의 해석 및 실행은 인텔 CPU가 직접 처리하고, 변환된 코드의 해석 및 실행은 VM에서 처리(Emulation)해야 한다. 이는 변환된 코드의 실행 시점에서는 VM 이 CPU 사용 권한을 획득해야 함을 의미한다.



[그림 3] Hybrid한 프로그램의 실행

[그림 3]의 (a)는 CPU가 순차적으로 명령어를 수행했을 때의 모습이다. CPU가 변환된 코드 영역을 CPU 명령어로 판단하고, 이를 잘못 해석하여 에러가 발생한다.

[그림 3]의 (b)는 변환된 코드 영역을 실행하기 전에, VM으로 CPU의 제어권을 옮기는 과정을 설명한 것이다. VM이 CPU의 제어권을 획득하면, VM이 변환된 코드 영역의 해석 및 실행(Fetch-Decode-Execute)을 진행하게 된다.

[그림 3]의 (b)와 같은 형태를 구현하기 위해 스텁 코드(Stub Code)를 변환된 코드 영

역의 시작 위치에 삽입하게 된다.

영역	내용	해석 및 실행
인텔 Code	...	인텔 CPU
Stub Code	push Addr of Virtualized Code jmp VM	인텔 CPU
Virtualized Code	...	VM
VM	pop eax ; virtualized code addr ...	인텔 CPU

2.4 State Switching

변환된 코드 영역에 대한 Emulation은 소스 프로그램의 연장(continuation)으로 봐야 한다. 변환된 코드 영역은 소스 프로그램의 일부분을 변환한 것이기 때문이다.

다시 말하면, 소스 프로그램으로부터 변환된 코드 영역으로의 switching시에 소스 프로그램에서 사용하던 레지스터 값을 그대로 변환된 코드 영역에서 사용할 수 있도록 해야 한다는 것이다.

이를 위해 State Switching이 필요하다. State Switching이란 소스 프로그램에서 변환된 코드 영역으로 switching할 때, 소스 프로그램이 사용하던 레지스터값을 변환된 코드에서 사용할 수 있도록 레지스터 블록에 복사하는 과정이다.

다음은 State Switching을 수행하는 VM 의 시작 코드 부분이다.

```
mov          [REG_BLOCK_EAX], eax
mov          [REG_BLOCK_EBX], ebx
mov          [REG_BLOCK_ECX], ecx
mov          [REG_BLOCK_EDX], edx
```

```
mov            [REG_BLOCK_ESI], esi
mov            [REG_BLOCK_EDI], edi
mov            eax, [ebp + RESERVED_STACK]
mov            [REG_BLOCK_EBP], eax
lea            eax, [ebp + RESERVED_STACK + 0Ch]
mov            [REG_BLOCK_ESP], eax
mov            eax, [ebp + RESERVED_STACK + 8]
mov            [REG_BLOCK_EIP], eax
```

이러한 State Switching이 먼저 이루어진 후에 변환된 코드 영역의 Emulation이 진행된다. Emulation에서는 실제 레지스터를 사용하는 것이 아니라, 레지스터 블록에 있는 멤버를 사용한다.

영역	명령어	설명
Source Program #1	mov eax, 10 mov ebx, 20	eax = 10 ebx = 20
Stub Code	push Virtualized Code Addr jmp VM	VM으로 jump
Virtualized Code	mov eax, 0 mov ebx, 0	변환된 코드
Source Program #2	mov ecx, eax mov edx, ebx	ecx = eax = 0 edx = ebx = 0
VM	…	Register Block 할당
	mov [REG_BLOCK + EAX], eax mov [REG_BLOCK + EBX], ebx …	State Switching
	pop [REG_BLOCK + EIP]	EIP = Virtualized Code Addr
	…	…

	mov [REG_BLOCK + EAX], 0 mov [REG_BLOCK + EBX], 0	Emulation
	…	…
	mov eax, [REG_BLOCK + EAX] mov ebx, [REG_BLOCK + EBX] …	State Switching Back
	jmp Source Program #2	Source Program #2로 분기

변환된 코드 영역의 Emulation이 끝나고, 소스 프로그램(인텔 명령어 코드)으로 다시 복귀할 때 변환된 코드 영역에서 사용하던 레지스터 블록의 값을 실제 레지스터로 복사하는 State Switching 과정(앞의 과정과 반대 과정)이 이루어진다.

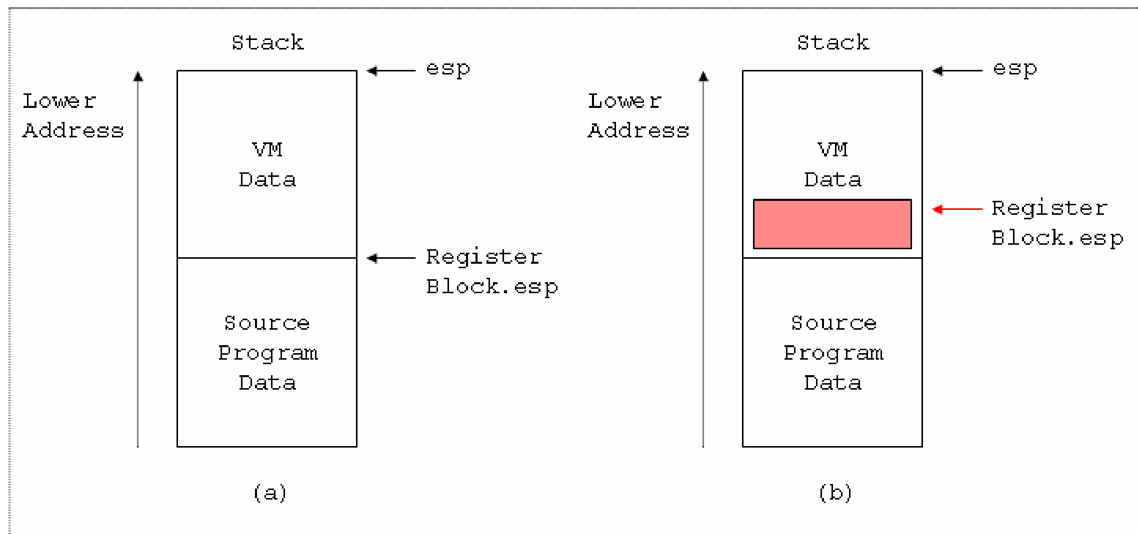
2.5 Data Structures of VM

VM은 변환된 코드의 Emulation을 수행하기 위해, 내부적으로 여러 데이터 구조를 포함하고 있다. 그렇다면 VM에 포함된 데이터 구조에는 어떤 것이 있을까? 가장 먼저 떠올려야 하는 것이 레지스터 블록이다. 그렇다면 레지스터 블록을 메모리 어딘가에 저장하고 있어야 할 것이다. 메모리 어디에 저장할 것인가?

바로 VM코드를 실행하는 스레드의 스택에 저장한다. 이렇게 스택에 저장하는 이유에 대해 생각해보자. 만약 멀티 스레드 환경에서 여러 스레드가 변환된 코드를 Emulation하는 상황일 때, 레지스터 블록을 스택에 저장하지 않는다고 가정해보자. 이 경우 VM은 각각의 스레드에 대한 레지스터 블록을 위해 힙을 할당할 것이고, 각 스레드 ID와 힙을 매핑하기 위한 자료 구조의 설계가 필요할 것이다. 또한 이러한 자료 구조는 여러 스레드가 공유하는 자원(shared resource)이기 때문에 동기화 프로세스에 대한 설계를 고민해봐야 한다. 이는 생각만큼 간단한 작업이 아니다. (참고로 3.2절에서 보게 될 VM코드는 어셈블리로 짜여져 있고, Win32 API를 사용하지 않는다. 만약 레지스터 블록을 스택에 저장하지 않는다면 레지스터 블록을 위한 힙 메모리 할당부터 동기화를 위한 코드까지 직접 짜거나, 아니면 힙을 할당하는 Win32 API와 스레드 동기화에 필요한 Win32 API의 위치를 구하여 이를 호출하는 코드를 작성해야 할 것이다.)

이러한 멀티 스레드 환경에서의 안정성과 설계의 단순화를 위해 스레드의 스택에 저장하는 것이다. 스레드의 스택은 스레드마다 고유하게 할당된다. 이러한 스레드의 스택 메모리 할당은 단지 esp 레지스터의 값을 할당을 원하는 크기만큼 감소시킴으로써 구현할 수 있다. 또한 스레드 스택에 위치한 레지스터 블록은 공유 자원이 아니기 때문에 동기화 프로세스를 고민할 필요가 없다.

하지만 스레드의 스택에 VM 자료 구조를 유지하는 것에도 문제가 있다. (하지만 위에서 살펴본 문제에 비하면 너무나 작은 문제다.) 다음 [그림 4]을 보자.



[그림 4] VM의 스택

[그림 4]의 (a)에서 VM이 다음과 같은 변환된 코드를 Emulation 한다고 생각해보자.

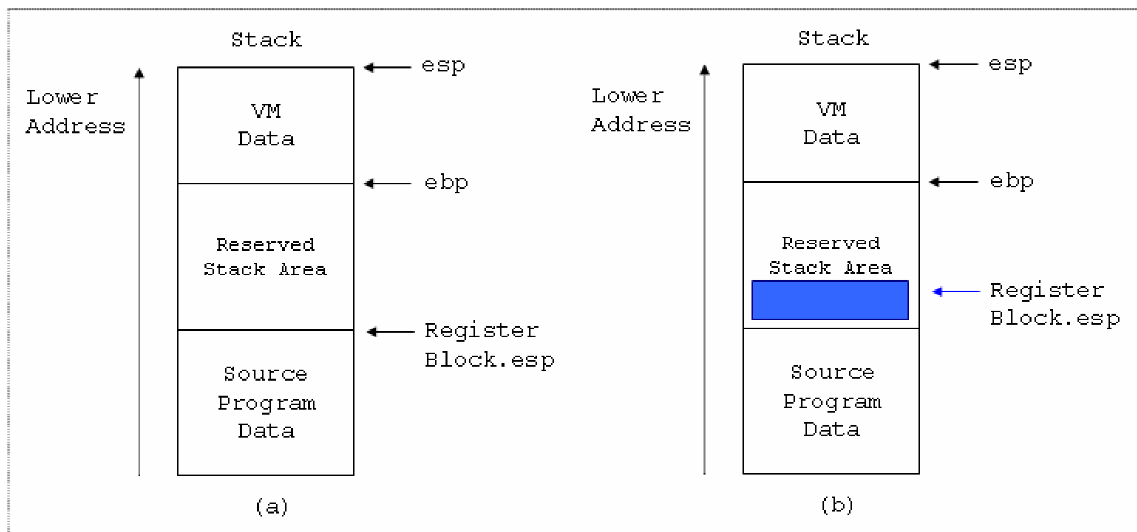
```
sub    esp, 4
mov    [esp], 10
```

위의 변환된 코드를 VM은 다음과 같이 Emulation 할 것이다.

```
RegisterBlock.esp -= 4;
*(int*)RegisterBlock.esp = 10
```

[그림 4]의 (b)는 변환된 코드를 Emulation 하는 상황을 나타낸 것인데, 여기서 문제는 변환된 코드의 Emulation시에 VM 이 관리하는 데이터 영역을 침범한다는 것이다. 이러한 일이 발생하는 이유는 소스 프로그램의 스레드 스택을 VM의 내부 자료를 저장하는 데 사용하기 때문이다.(물론 스레드 스택을 훼손시키지 않기 위해 VM의 내부 자료는 스레드 스택의 상위에 저장한다.)

이러한 문제를 해결하기 위해, 다음 [그림 5]과 같이 VM이 사용하는 스택을 더 상위로 위치시키고 변환된 코드의 Emulation 시에 필요한 스택 영역을 예약(Reserved)했다.



[그림 5] Emulation을 위한 Reserved Stack Area

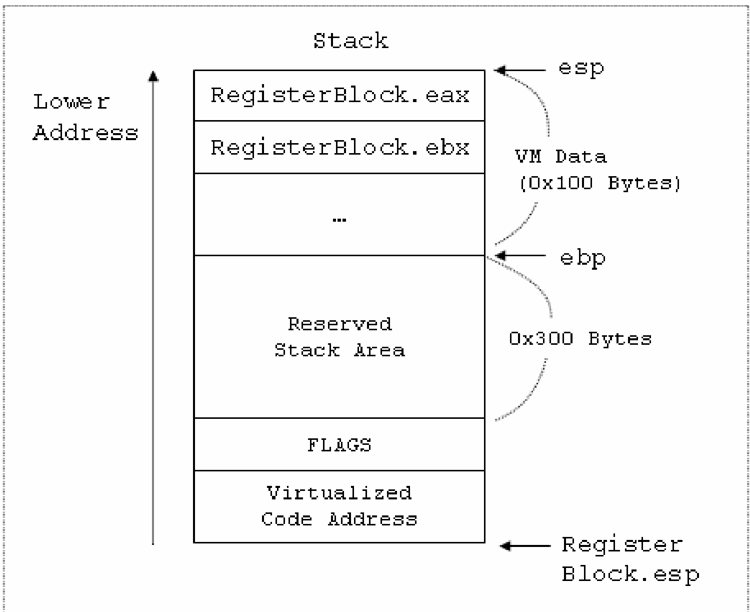
[그림 5]와 같이 스택을 구성하였을 경우, 만약 RegisterBlock.esp가 Reserved Stack을 다 사용하여 VM Data 영역으로 침범하는 일이 다시 발생할 수 있다. 이처럼 Regiseter.esp이 가리키는 위치가 ebp이 가리키는 위치보다 작을 경우, 즉 VM Data영역을 침범하는 시도가 발견될 경우, VM Data영역을 스택의 더 상위로 복사하고 Reserved Stack을 확장하여 문제를 해결할 수 있다.

이에 대한 Pseudo Code는 다음과 같다.

```
if (RegisterBlock.esp < ebp)
{
        VMDataSize = ebp - esp;
```

```
        NewVMData = RegisterBlock.esp - ReservedStackAreaSize ;
        for (int i = 0; i < VMDataSize; i++)
        {
                *(NewVMData - i) = *(ebp - i);
        }
        esp = NewVMData - VMDataSize;
        ebp = NewVMData;
}
```

[그림 6]은 VM이 실제 구성하고 있는 스택의 구조를 나타낸다.



[그림 6] VM의 초기 스택(State Switching 작업이 끝난 이후)

2.6 VM Dispatch Loop

VM 코드는 다음과 같은 Dispatch Loop 구조를 가진다.

```
BYTE opcode;
While(1)
{
        // Opcode를 가져온다.(Fetch)
        Opcode = GetByte(RegisterBlock.eip);
        // Increment EIP
        RegisterBlock.eip++;
        // Opcode를 처리하는 핸들러 호출
        opcodeTable[Opcode].handler();
}
```

opcodeTable은 Emulation 함수의 주소를 가진 배열이다.

opcodeTable Index는 opcode와 동일하다.

Index	Address
0	'mov' handler address
1	'xor' handler address
…	…
255	'end' handler address

예를 들어 mov에 대한 opcode가 0이라고 가정했을 때, opcodeTable[0].handler()를 호출하는 것이다.

오퍼랜드에 대한 패치(Fetch)는 각 명령어 핸들러에서 수행한다.(오퍼랜드에 대한 패치를 포함한 디코딩 과정을 Main Dispatch Loop에 포함하는 것이 더 나은 구조이다. 즉 Decode and Dispatch Loop 구조가 바람직하다.)

다음은 실제 VM 의 Main Dispatch Loop 부분이다.

```
decode:
```

```
mov     ebx, [REG_BLOCK_EMULATOR_BASE]
add     ebx, offset opcode_table          // opcode_table address
call    get_opcode
mov     ecx, dword ptr [ebx + eax * 4]    // opcode_table[opcode]
add     ecx, [REG_BLOCK_TARGET_BASE]
mov     eax, ebp
sub     eax, EMULATOR_STACK
push    eax                   // 파라미터는 RegisterBlock의 주소
call    ecx                   // 명령어핸들러(&RegisterBlock); 호출
add     esp, 4
jmp     decode
```

다음 코드는 mov 명령어 핸들러(vcp_mov)다.(명령어의 핸들러 타입은 거의 동일하다.) vcp_mov의 파라미터(VCPRegisterBlock *)는 Main Dispatch Loop에서 전달받는다. vcp_mov의 시그너쳐 중 일부인 HANDLER_TYPE은 naked 함수를 선언하도록 정의되어 있다. vcp_mov는 naked 함수이기 때문에 함수의 지역 변수에 대한 메모리 할당 및 해제를 컴파일러 대신 직접 처리해주어야 한다. vcp_mov 코드 중 sub esp, __LOCAL_SIZE를 통해 지역 변수에 대한 할당을 구현하고, add esp, __LOCAL_SIZE 를 통해 해제를 구현한다.

```
HANDLER_TYPE vcp_mov(VCPRegisterBlock *pBlock)
{
        HANDLER_START
        HANDLER_ID(0x00)

        int eip, op_data1, op_data2;
        OperandInfo operandInfo;

        __asm
        {
                push    ebp
                mov     ebp, esp
                sub     esp, __LOCAL_SIZE
```

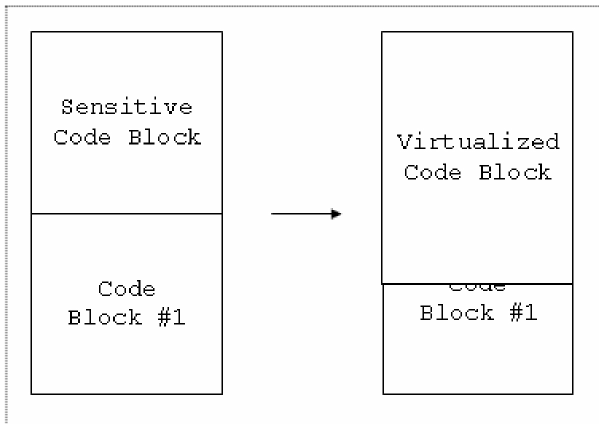
```
}

eip = pBlock->eip;
eip++; // skip opcode

// fetch operandInfo, operand1, operand2
operandInfo = *(OperandInfo *)eip;
eip += sizeof(operandInfo);
op_data1 = *(int*)eip;
eip += sizeof(op_data1);
op_data2 = *(int*)eip;
eip += sizeof(op_data2);

// increment eip
pBlock->eip = eip;

switch(operandInfo.o2_addrmode)
{
case OPR_REG:
	op_data2 = pBlock->registers[op_data2];
	break;

case OPR_INREG:
	op_data2 = pBlock->registers[op_data2];
	op_data2 = *(int*)op_data2;
	break;

case OPR_INIMM:
	if (operandInfo.o2_reloc)
		op_data2 += pBlock->targetBase;
	op_data2 = *(int*)op_data2;
	break;
case OPR_IMM:
```

```
		break;
	default:
		// ??
		break;
	}

	switch(operandInfo.o1_addrmode)
	{
	case OPR_REG:
		pBlock->registers[op_data1] = op_data2;
		break;

	case OPR_INREG:
		op_data1 = pBlock->registers[op_data1];
		*(int*)op_data1 = op_data2;
		break;

	case OPR_INIMM:
		if (operandInfo.o1_reloc)
			op_data1 += pBlock->targetBase;

		*(int*)op_data1 = op_data2;
		break;
	case default:
		// ??
		break;
	}

	__asm
	{
		add	esp, __LOCAL_SIZE
		pop	ebp
		ret
```

```
        }


        HANDLER_END
}
```

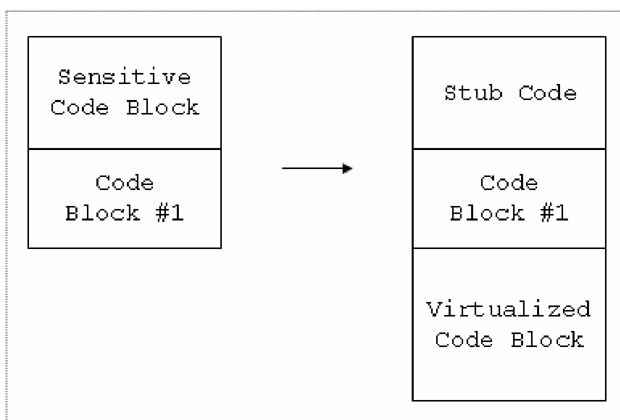
2.7 Program Counter Mapping Table(SPC-TPC Mapping Table)

인텔 명령어와 변환된 명령어의 포맷은 다르다. 그리고 명령어의 크기도 다르다.
일반적으로 변환된 명령어의 크기가 더 크다.



[그림 7] 원래의 코드와 변환된 코드의 크기 비교

[그림 7]에서 보듯이, 원래의 코드 영역보다 이를 변환한 코드 영역의 크기가 더 크기 때문에, 원래의 코드 영역의 위치에 변환된 코드를 저장하게 되면 원래의 코드 영역을 벗어난 영역(#1)까지 데이터를 덮어써 버리게 된다.



[그림 8] 새로운 영역에 변환된 코드를 복사

이러한 문제를 해결하기 위해, [그림 8]과 같이 변환된 코드 영역을 파일의 새로운 영역에 복사하게 된다. (원래의 코드 영역에는 Stub Code를 저장한다.)

여기서 문제 제기를 위해, 다음의 소스 프로그램을 예제로 들겠다.

Address	인텔 Instruction	설명
0x1000		VCP_START
0x1000	05 01 00 00 00	add eax, 1
0x1005	3d 0a 00 00 00	cmp eax, 0ah
0x100A	7c 0c	jl 0x1000
0x100A		VCP_END

소스 프로그램의 VCP_START 부터 VCP_END 영역까지 변환이 이루어진다.
변환이 이루어진 코드의 크기 문제로 새로운 영역(0x2000 번지)에 변환된 코드를 저장하기로 한다. (add의 opcode는 1, cmp는 2, jl는 3으로 가정한다.)

Address	Virtualized Instruction	설명
0x2000	01 83 00 00 00 00 01 00 00 00	add eax, 1
0x200A	02 83 00 00 00 00 0a 00 00 00	cmp eax, 0ah
0x2014	03 b8 00 10 00 00	jl 0x1000

위와 같이 변환하였을 경우, 문제는 바로 변환된 코드 중 jl 명령어의 오퍼랜드에 있다. 변환된 코드가 새로운 영역(0x2000)으로 복사되었기 때문에, jl 명령어의 오퍼랜드가 가리키는 0x1000 번지는 의미가 없는 것이다. 0x2000 번지로 변경하는 것이 올바르다.

위 문제를 통해 소스 프로그램에서 사용한 주소는 변환된 코드의 주소로 변경되어야 한다는 것을 알 수 있다.

Program Counter Mapping Table(SPC-TPC 매핑 테이블)은 소스 프로그램에서 사용한 주소와 변환된 코드의 주소에 대한 매핑 정보를 보관하고 있는 테이블이다.

위의 예에서는 다음과 같은 SPC-TPC 매핑 테이블이 존재해야 할 것이다.

SPC	TPC
0x1000	0x2000

VM 은 방금 살펴본 jl 0x1000 코드를 Emulation할 때, 먼저 SPC-TPC 매핑 테이블의

엔트리중 0x1000과 일치하는 SPC가 존재하는지 검색하고, 만약 존재한다면 이를 TPC로 변경하여 0x2000번지로 점프하게 된다. 만약 매핑 테이블에 SPC가 존재하지 않는다면, 변환된 코드에 인코딩된 주소(0x1000)로 점프하게 된다.

SPC-TPC 매핑 테이블을 생성하기 위해 다음과 같은 2단계의 변환 작업이 이루어지는데 다음 소스 프로그램의 예를 가지고 이에 대해 자세히 알아보자.

Address	인텔 Instruction	설명
0x1000		VCP_START
0x1000	05 01 00 00 00	add eax, 1
0x1005	3d 0a 00 00 00	cmp eax, 0ah
0x100A	7c 0c	jl 0x1000
0x100C	EB 00	jmp 0x100E
0x100E	6a 01	push 1
		VCP_END

(VCP_START는 변환 작업대상 코드 영역의 시작을 가리키고, VCP_END는 변환 작업 대상 코드 영역의 끝을 가리킨다.)

1) 첫번째 단계

SPC-TPC 매핑 테이블을 만들고 SPC를 설정하는 것이다.

VCP_START/VCP_END 영역을 스캐닝하여, Branch 명령어(jmp, call 등)를 찾는다.

발견된 Branch 명령어에서 사용하는 주소 오퍼랜드를 SPC-TPC 매핑 테이블의 SPC 항목에 추가한다. (이 때 주소 오퍼랜드가 Relative 주소라면 이를 absolute 하게 변환한 후 SPC 항목에 추가한다. 왜냐하면 변환된 명령어에는 Relative 주소 지정 방식이 존재하지 않기 때문이다.)

따라서 소스 프로그램의 0x100A 라인과 0x100C 라인에서 사용된 주소값 0x1000, 0x100E가 SPC 항목으로 추가될 것이다.

SPC	TPC
0x1000	
0x100E	

2) 두번째 단계

두번째 단계는 코드를 변환하는 동시에 TPC를 설정한다.

우선 VCP_START/VCP_END 영역의 변환 작업을 진행한다. 변환 작업을 진행하는 동안 SPC-TPC 매핑 테이블에 있는 SPC 라인의 변환 작업이 시작되면, SPC-TPC 매핑 테이블의 SPC 항목과 대응하는 TPC 항목에 변환된 코드의 위치를 저장한다.

먼저 VCP_START/VCP_END 영역의 0x1000 번지부터 변환 작업을 시작할 것이다.
(변환된 코드는 0x2000 부터 저장된다고 가정한다.)

0x1000 번지에서 변환된 코드는 0x2000번지에 저장이 된다.

Address	Virtualized Instruction	설명
0x2000	01 83 00 00 00 00 01 00 00 00	add eax, 1

이 때, 0x1000번지는 SPC-TPC 매핑 테이블의 SPC 항목에 존재하므로, SPC 항목과 대응하는 TPC 항목에 변환된 코드의 저장된 위치(0x2000)을 저장한다.

SPC	TPC
0x1000	0x2000
0x100E	

이러한 변환 작업이 진행되고, 0x100E번지에 있는 명령어(push 1)도 SPC-TPC 매핑 테이블의 SPC 항목에 존재하므로, 0x100E번지 명령어(push 1)는 변환이 진행되는 동시에 변환된 명령어가 저장된 위치(0x202c)도 TPC 항목에 저장된다.

Address	Virtualized Instruction	설명
0x2000	01 83 00 00 00 00 01 00 00 00	add eax, 1
…	…	…
0x202C	04 98 01 00 00 00	push 1

SPC	TPC
0x1000	0x2000
0x100E	0x202c

이러한 과정을 통해 SPC-TPC 매핑 테이블이 구성되면, VM이 참조할 수 있는 특정한 위치에 저장되며, Branch 명령어를 Emulation할 때마다 VM은 SPC-TPC 매핑 테이블을 참조하여 TPC 항목이 존재하면 주소 변환을 수행하게 된다.

점프하려는 주소가 SPC-TPC 매핑 테이블에 존재한다는 것은, 변환된 코드에서 또 다른 위치에 있는 변환된 코드로 점프한다는 것을 의미한다. 만약 SPC-TPC 매핑 테이블에 존재하지 않는다면, 점프하려는 주소는 변환된 코드 영역이 아닌 소스 프로그램의 위치를 가리키는 것이다. 따라서 이러한 경우에는 State Switching(레지스터블럭->실제레지스터) 후에 소스 프로그램으로 점프하면 된다.

2.8 Base Relocation Problem

Windows PE 파일은 실제 로드 주소와 ImageBase가 다를 경우, Base Relocation이 발생한다. 이러한 Relocation은 명령어 코드의 오퍼랜드(주소)를 수정한다. 따라서 VCP에서도 이를 고려한 설계가 필요하다.

다음과 같은 명령어 코드를 포함한 파일이 있다고 가정하자.
(이 코드는 ImageBase가 0x10000000 이라고 가정한다.)

① mov eax, [0x1000000A]

만약 이 파일의 실제 로드 주소가 0x20000000 이라면, Relocation이 발생한다. Relocation 후 코드 ①는 다음과 같이 변경된다.(로더가 변경함)

② mov eax, [0x2000000A]

VCP 에서 코드 ①을 아래와 같이 변환하였다고 가정하자.

③ mov eax, [0x1000000A]

여기서 문제가 발생한다. ③에 대한 Relocation이 필요한 경우가 발생했을 때, 원래는 로더가 Relocation을 담당했지만 변환된 코드에 대한 부분까지 로더가 관여하지 않는다. 왜냐하면 변환된 코드의 위치가 새로운 영역으로 변경 되었기 때문이다.

변환된 코드에 대한 Relocation을 로더에서 수행하지 못하므로, 이를 VM 에서 처리해야 한다. 2.2절에서 살펴 보았듯이 이러한 Relocation을 처리하기 위해 만든 플래그가 Operand Info의 Relocation 비트이다. 변환된 코드의 Operand Info에 Relocation 비트가 설정되어 있다면, VM은 Emulation시에 해당 오퍼랜드의 Relocation을 수행하게 된다.

코드에 Relocation 플래그를 설정하는 것은 변환 작업(Binary Translation) 시에 처리해야 할 일이고, 실제 Relocation을 수행하는 것은 VM 에서 처리해야 한다.

작업은 크게 2 단계로 나뉜다.

1) Translation 단계
변환할 소스 명령어의 Relocation 여부를 판단한다.(오퍼랜드가 relocatable한지에 대해 파악하는 가장 정확한 방법은 .reloc 섹션의 정보를 참조하는 것이다.)
만약 Relocation 이 발생할 수 있는 오퍼랜드일 경우에, 변환된 명령어의 오퍼랜드 정보에 Relocation 플래그를 설정한다.

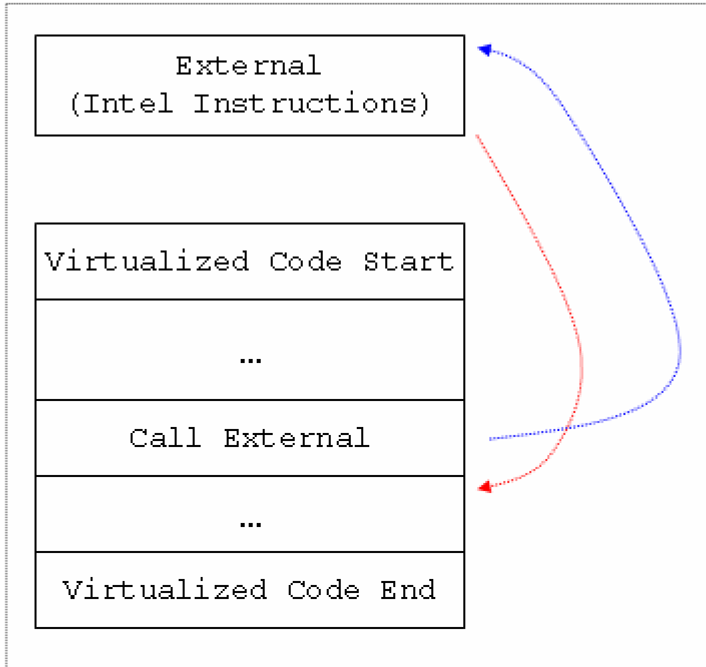
2) Emulation 단계
변환된 명령어의 오퍼랜드 정보에 Relocation 플래그가 설정되어 있을 경우, 해당 오퍼랜드의 Relocation 과정이 필요하다. 이를 위해선 다음과 같은 정보를 VM 에서 가지고 있어야 한다.

① 초기 ImageBase (Translation 시에 파악할 수 있는 정보이다.)
② 모듈의 로드 주소 (Emulation 시에 알 수 있는 정보이다.)

Relocation 수행 시 적용할 공식이다.

실제 주소 = (초기 ImageBase – 모듈의 로드 주소) + 오퍼랜드값

2.9 External Call Problem



[그림 9] External Call Problem

[그림 9]와 같이 변환된 코드 영역에서 call external와 같은 외부 소스 프로그램의 함수(external)를 호출하는 변환된 코드가 있다고 가정하자. State Switching을 통해서 외부 함수(external)가 실행되는 시점에서는 CPU에서 직접 해석 및 실행을 하게 될 것이다. 문제는 외부 함수(external)가 리턴할 때에 있다. 리턴하는 곳은 그림에서 보듯이 변환된 코드 영역에 있는 호출(call external) 명령어 다음 위치이다. 리턴하는 위치의 명령어가 변환된 명령어이기 때문에, CPU는 이것을 잘못 해석하는 문제가 발생한다.

따라서 이러한 문제는 호출(call external) 명령어의 다음 위치에 VM으로 전환하는 stub code(인텔 명령어)을 삽입함으로써 해결할 수 있다.

주소	명령어	명령어 분류	해석 및 실행	순서
0x1000	external()	IA-32 Instruction	CPU	4
…	…	IA-32 Instruction	CPU	5
0x1500	ret	IA-32 Instruction	CPU	6
0x2000	VCP_START	Virtualized Instruction	VM	1
…	…	Virtualized Instruction	VM	2

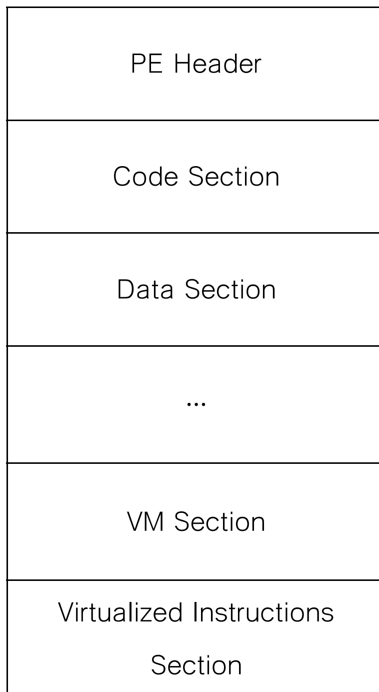
0x2020	call external	Virtualized Instruction	VM	3
0x202A	push 0x2034	Stub Code(IA-32)	CPU	7
0x202F	jmp vm	Stub Code(IA-32)	CPU	8
0x2034	…	Virtualized Instruction	VM	9
0x2100	VCP_END	Virtualized Instruction	VM	10

0x202A는 VM이 해석할 변환된 코드의 위치(0x2034)를 전달하기 위한 코드이다.

0x202F는 VM으로 전환하는 코드이다. 0x202F 명령의 실행 이후에 0x2034에 있는 변환된 명령어의 Emulation이 VM 에 의해 진행된다.

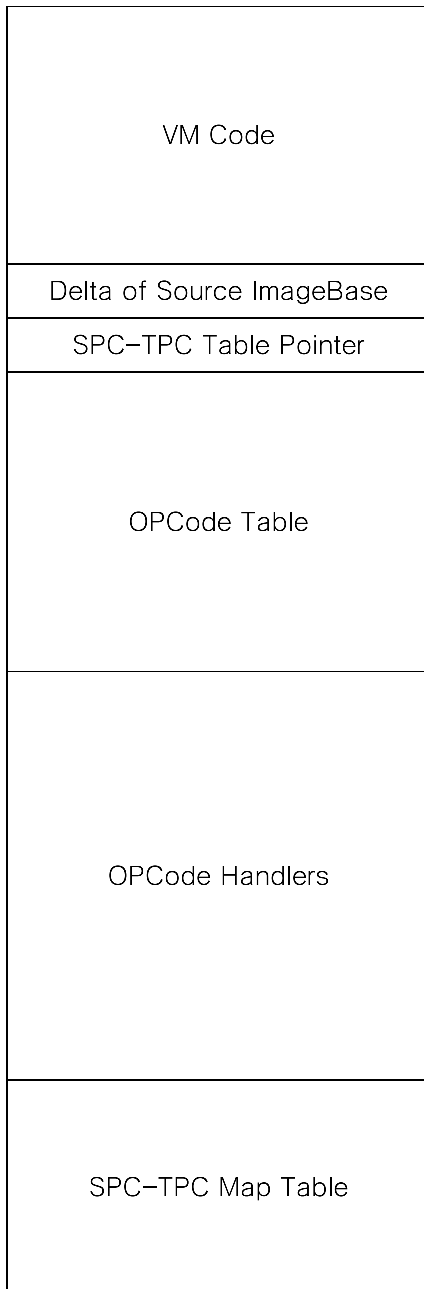
2.10 Virtualized File Structure

VCP 에 의해 보호된 최종 Windows PE 파일의 전체적인 구조이다.
변환된 명령어와 VM 은 각각 Windows PE 파일의 새로 만들어진 섹션에 저장된다.



섹션을 추가하기 위해서는 섹션 헤더를 추가해야 하고, PE Header의 여러 필드에 대한 조정이 필요하다.

VM Section은 다음과 같이 구성된다.



Delta of Source ImageBase 는 소스 프로그램의 ImageBase와 실제 소스 프로그램이 로드된 기준 주소와의 차이값을 가지고 있다. 이것은 Relocation을 위한 값으로, Relocation 비트가 설정된 변환된 명령어의 오퍼랜드에 더해진다. 이것은 소스 프로그램이 실행될 때 로더에 의해서 설정되며, 이는 .reloc 섹션의 설정으로 가능하다.

3. Implementation

3.1 source.exe

source.exe 파일은 VCP 적용을 위해 만든 테스트 프로그램이다.

source.exe 파일은 아래와 같이 test() 함수를 가지고 있는데, test() 함수의 색칠한 부분을 Virtualized Instruction으로 변환하고 VM 으로 Emulation하는 과정을 설명할 것이다.

```
void test()
{
	int i = 0;

	for (i = 0; i < 10; i++)
	{
		printf("I love you\n");
	}
}

int main(int argc, char* argv[])
{
	test();
	return 0;
}
```

색칠한 부분의 어셈블리 코드는 다음과 같다.

```
12:         int i = 0;
0040104A C7 45 FC 00 00 00 00 mov          dword ptr [ebp-4],0
13:
14:         for (i = 0; i < 10; i++)
00401051 C7 45 FC 00 00 00 00 mov          dword ptr [ebp-4],0
00401058 EB 09                jmp          source +43h (00401063)
0040105A 8B 45 FC             mov          eax,dword ptr [ebp-4]
0040105D 83 C0 01             add          eax,1
```

```
00401060 89 45 FC              mov        dword ptr [ebp-4],eax
00401063 83 7D FC 0A           cmp        dword ptr [ebp-4],0Ah
00401067 7D 0F                 jge        source +58h (00401078)
15:         {
16:             printf("I love you\n");
00401069 68 1C 20 42 00        push       offset string "I love you\n" (0042201c)
0040106E E8 8D 00 00 00        call       printf (00401100)
00401073 83 C4 04              add        esp,4
17:         }
00401076 EB E2                 jmp        source+3Ah (0040105a)
```

3.2 Binary Translation

test() 함수를 변환된 명령어로 변환하기 위해서는 Virtualized Instruction Set이 필요하다. 다음의 Virtualized Instruction Set과 Opcode이다.

OPCode	Virtualized Instruction
0x00	
0x01	mov
0x02	sub
0x03	jmp
0x04	add
0x05	cmp
0x06	jge
0x07	call
0x08	push
0x09	pop
0x0a	end

Virtualized Instruction Set을 가지고 test()의 라인을 변환해보자.

우선 12 라인을 보자.

```
12:          int i = 0;
0040104A C7 45 FC 00 00 00 00 mov             dword ptr [ebp-4],0
```

변환된 명령어의 주소 지정 모드에는 [reg + disp#]와 같은 방식을 지원하지 않기 때문에 다음과 같이 변환해야 한다.

```
mov      tmp, ebp
sub      tmp, 4
mov      dword ptr [tmp], 0
```

mov tmp, ebp를 Virtualized Instruction format에 맞게 인코딩하면 다음과 같다.

OPCODE(1BYTE)	OPERAND_INFO(1BYTE)	OPERAND1(4BYTES)	OPERAND2(4BYTES)
01	80	0a 00 00 00	06 00 00 00

OPERAND_INFO 가 0x80 인 이유는 다음과 같다.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	0	0	0	0	0

Bit 7, 6은 Ref_Size를 나타내는데 32비트를 표현한 것이며, Bit 5, 4, 3은 오퍼랜드1의 주소 지정 방식을 의미한 것인데 0 0 0 이므로 레지스터를 의미한다. Bit 2, 1, 0 은 오퍼랜드2의 주소 지정 방식을 의미하고 0 0 0 이므로 레지스터를 의미한다.

오퍼랜드1이 0a 00 00 00 인 이유는 tmp 레지스터의 인덱스가 10(0xa)이고 Little Endian으로 워드가 인코딩되기 때문이다.
오퍼랜드2는 06 00 00 00 으로 ebp 레지스터(인덱스 6)를 가리킨다.

이런 식으로 변환을 진행하면 다음과 같은 변환된 명령어를 얻을 수 있다.

```
00401051   mov   dword ptr [ebp-4],0
01 80 0a 00 00 00 06 00 00 00            mov      tmp, ebp
02 83 0a 00 00 00 04 00 00 00            sub      tmp, 4
```

```
01 8b 0a 00 00 00 00 00 00 00           mov     dword ptr [tmp], 0

00401058    jmp    source +43h (00401063)
03 b8 63 10 40 00                       jmp     00401063 (internal, spc_tpc_mapping)

0040105A    mov    eax,dword ptr [ebp-4]
01 80 0a 00 00 00 06 00 00 00           mov     tmp, ebp
02 83 0a 00 00 00 04 00 00 00           sub     tmp, 4
01 81 00 00 00 00 0a 00 00 00           mov     eax, dword ptr [tmp]

0040105D    add    eax,1
04 83 00 00 00 00 01 00 00 00           add     eax, 1

00401060    mov    dword ptr [ebp - 4], eax
01 80 0a 00 00 00 06 00 00 00           mov     tmp, ebp
02 83 0a 00 00 00 04 00 00 00           sub     tmp, 4
01 88 0a 00 00 00 00 00 00 00           mov     dword ptr [tmp], eax

00401063    cmp    dword ptr [ebp -4], 0ah
01 80 0a 00 00 00 06 00 00 00           mov     tmp, ebp
02 83 0a 00 00 00 04 00 00 00           sub     tmp, 4
05 8b 0a 00 00 00 0a 00 00 00           cmp     dword ptr [tmp], 0ah

00401067    jge    source +58h (00401078)
06 b8 78 10 40 00                       jge     00401078 (relocatable, external)

00401069    push   offset string "I love you\n" (0042201c)
08 b8 1c 20 42 00                       push    0042201c (relocatable)

0040106E    call   printf (00401100)
07 b8 00 11 40 00                       call    00401100 (relocatable, external)
==========================================================================
Stub Code(IA-32 Code)
```

```
68 ?? ?? ?? ??                          push    next vcode address
E9 ?? ?? ?? ??                          jmp     emulator_address (rel32)
==============================================================================


00401073    add    esp, 4
04 83 07 00 00 00 04 00 00 00           add     esp, 4


00401076    jmp    source+3Ah (0040105a)
03 b8 5a 10 40 00                       jmp     0040105a (internal, spc_tpc_mapping)
```

위의 결과에서 관심을 가져야 할 부분은 jmp와 call 명령어에 대한 변환 부분이다.

소스 프로그램에서 jmp rel#와 같이 상대적인 주소를 지정하는 명령어의 경우이다. 변환된 명령어에는 이러한 상대적인(relative) 주소 지정 모드가 존재하지 않기 때문에, 이를 absolute한 주소로 수정하여 인코딩하게 된다.

```
00401058    jmp    source +43h (00401063)
03 b8 63 10 40 00                       jmp     00401063 (internal, spc_tpc_mapping)
```

여기서 internal 태그는 jmp의 목적지가 변환된 코드 영역 내에 있다는 것을 의미하며, 이는 SPC-TPC 매핑 테이블을 통한 주소 변환이 필요하다는 것을 의미한다.

다음은 call 명령어에 대한 변환 결과이다.

```
0040106E    call   printf (00401100)
07 b8 00 11 40 00                       call    00401100 (relocatable, external)
==============================================================================
Stub Code(IA-32 Code)
68 ?? ?? ?? ??                          push    next vcode address
E9 ?? ?? ?? ??                          jmp     emulator_address (rel32)
==============================================================================
```

2.9절에서 살펴 보았듯이, printf() 함수는 변환된 코드 영역에 존재하지 않는 외부 함수(external)이다. 따라서 printf() 함수의 호출 명령 다음에 오는 명령어는 인텔 명령어(IA-32)로 구성된 Stub Code가 되어야 한다.

3.3 The VM

변환된 명령어는 VM에서 Emulation한다.
VM은 다음과 같이 어셈블리 코드로 작성되어 있다.

```
HANDLER_TYPE emulator()
{
        EMULATOR_START
_asm
{
// 1) Stack 초기화
//
//
//              |                       |   <- esp, Top(Lower Address)
//              |       VM Stack         |
//              |      (0x100bytes)      |
//              +===================+   <- ebp
//              |                       |
//              |    Reserved Stack      |
//              |      (0x300bytes)      |
//              |                       |
//              +===================+
//              |       old EBP          |
//              +===================+
//              |       old Eflags       |
//              +===================+
//              |     VCode Address      |
//              +===================+   <- RegisterBlock.esp
//


//start:
pushfd                          ; original eflags
push    ebp
```

```
sub             esp, RESERVED_STACK                 ; 0x300
mov             ebp, esp
sub             esp, EMULATOR_STACK                 ; 0x100

// 2) Register Block 초기화
//
mov             [REG_BLOCK_EAX], eax
mov             [REG_BLOCK_EBX], ebx
mov             [REG_BLOCK_ECX], ecx
mov             [REG_BLOCK_EDX], edx
mov             [REG_BLOCK_ESI], esi
mov             [REG_BLOCK_EDI], edi
mov             eax, [ebp + RESERVED_STACK]
mov             [REG_BLOCK_EBP], eax
lea             eax, [ebp + RESERVED_STACK + 0Ch]
mov             [REG_BLOCK_ESP], eax        ; ESP
mov             eax, [ebp + RESERVED_STACK + 8]
mov             [REG_BLOCK_EIP], eax        ; EIP

// 3) emulator_base를 구한다.
// VM 코드에 인코딩된 주소는 소스 프로그램과 호환되지 않는다.
// 다시 말하면, 그 주소는 VM이 소스 프로그램에 어태치된 상황을 고려하지 않은 것이다.
call    get_emulator_base
get_emulator_base:
pop             eax
sub             eax, offset get_emulator_base
mov             [REG_BLOCK_EMULATOR_BASE], eax

// 4) source_base
// Virtualized Instruction에 인코딩된 주소 오퍼랜드를 비롯하여,
// OPCode Table에 인코딩된 주소는 소스 프로그램의 ImageBase를 기준으로 설정되었다.
// Relocation 이 필요한 상황을 대비하여, source_base를 구한다.
// source_base는 소스 프로그램의 ImageBase에 대한 델타값으로 초기화된다.
```

```
// source_base = 소스 프로그램의 ImageBase + 소스 프로그램의 실제 로드 주소
// source_base에 대한 설정은 Windows Loader에서 한다.
// 이를 가능케 하는 것은 Relocation Table이다.
add         eax, offset source_base
mov         eax, [eax]
mov         [REG_BLOCK_SOURCE_BASE], eax

// RegisterBlock.eip의 Relocation
mov         ebx, [REG_BLOCK_EIP]
add         ebx, eax
mov         [REG_BLOCK_EIP], ebx

// 5) SPC-TPC Mapping Table의 주소
mov         eax, [REG_BLOCK_EMULATOR_BASE]
add         eax, offset spc_tpc_table_pointer
mov         eax, [eax]
add         eax, [REG_BLOCK_TARGET_BASE]
mov         [REG_BLOCK_SPC_TPC_TABLE], eax

// 6) Main Dispatch Loop
decode:
mov         ebx, [REG_BLOCK_EMULATOR_BASE]
add         ebx, offset opcode_table          // opcode_table address
call        get_opcode
mov         ecx, dword ptr [ebx + eax * 4]    // opcode_table[opcode]
add         ecx, [REG_BLOCK_TARGET_BASE]
mov         eax, ebp
sub         eax, EMULATOR_STACK
push        eax
call        ecx            // opcode_table[opcode](struct RegisterBlock *);
add         esp, 4
jmp         decode
```

```
get_opcode:
xor             eax, eax
mov             ecx, dword ptr [REG_BLOCK_EIP]
mov             al, byte ptr [ecx]
ret

source_base:
DWORD_TYPE(0x00000000)

spc_tpc_table_pointer:
DWORD_TYPE(0x77777777)

opcode_table:
...
```

다음은 call 명령어에 대한 핸들러(vcp_call)이다.

```
HANDLER_TYPE vcp_call(VCPRegisterBlock *pBlock)
{
        HANDLER_START
        HANDLER_ID(0x07)

        int eip, op_data1, i, find;
        OperandInfo operandInfo;
        SPC_TPC_TABLE *pTable;

        __asm
        {
                push    ebp
                mov     ebp, esp
                sub     esp, __LOCAL_SIZE
        }

        eip = pBlock->eip;
```

```
        eip++;    // skip opcode

        // fetch operandInfo, operand1, operand2
        operandInfo = *(OperandInfo *)eip;
        eip += sizeof(operandInfo);
        op_data1 = *(int*)eip;
        eip += sizeof(op_data1);

        // push return address
        pBlock->esp -= 4;
        *(int*)pBlock->esp = eip;

        // call 에 사용되는 주소가 SPC-TPC 매핑 테이블에 존재하는지 확인한다.
        // 만약 존재한다면 변환된 코드 영역의 위치로 변경되어야 한다.
        pTable = (SPC_TPC_TABLE *)pBlock->spc_tpc_table;
        for (i = 0; i < (pTable->size - sizeof(pTable->size)) / sizeof(SPC_TPC); i++)
        {
                find = 0;
                // 점프할 위치가 매핑 테이블에 존재한다.
                if (pTable->entry[i].spc == op_data1)
                {
                        // 점프할 위치가 변환된 코드 영역 내에 존재한다는 것을
                        // 의미한다. 따라서 RegisterBlock.eip 만 변경해준다.
                        find = 1;
                        pBlock->eip = pTable->entry[i].tpc + pBlock->targetBase;
                        break;
                }
        }

////////////////////////////////////////////////////////////////////////////////
// 점프할 위치가 변환된 코드 영역 외부(external)에 존재하는 경우   //
////////////////////////////////////////////////////////////////////////////////
        if (find == 0)
```

```
{
	// relocation bit가 설정되어 있다면, relocation 수행
	if (operandInfo.o1_reloc)
		op_data1 += pBlock->targetBase;

	pBlock->tmp = op_data1;

	__asm
	{
		// State Switching(RegisterBlock -> 인텔 Register)
		add		esp, __LOCAL_SIZE
		pop		ebp

		mov		eax, [REG_BLOCK_EAX]
		mov		ebx, [REG_BLOCK_EBX]
		mov		ecx, [REG_BLOCK_ECX]
		mov		edx, [REG_BLOCK_EDX]
		mov		esi, [REG_BLOCK_ESI]
		mov		edi, [REG_BLOCK_EDI]

		mov		esp, [REG_BLOCK_ESP]

		// 함수의 주소를 push하고
		push		[REG_BLOCK_TMP]

		mov		ebp, [REG_BLOCK_EBP]

		// 함수의 주소로 jmp
		ret
	}
}

__asm
```

```
        {
                add             esp, __LOCAL_SIZE
                pop             ebp
                ret
        }

        HANDLER_END
}
```

3.4 The Two Sections

다음은 변환될 소스 프로그램의 파일(PE) 구조이다.

PE Header
Code Section
Data Section
…
.vcp
.vcode

.vcp 섹션과 .vcode 섹션을 새로 추가한다.

.vcp 섹션에는 VM 코드와 명령어 핸들러 함수 코드, 그리고 opcode table 등이 포함된다.

.vcode 섹션에는 변환된 코드들이 위치한다.

[그림 10]은 .vcp 섹션 헤더이다.

pFile	Data	Description	Value
000002A0	2E 76 63 70	Name	.vcp
000002A4	00 00 00 00		
000002A8	00001000	Virtual Size	
000002AC	0002C000	RVA	
000002B0	00000F24	Size of Raw Data	
000002B4	0002A000	Pointer to Raw Data	
000002B8	00000000	Pointer to Relocations	
000002BC	00000000	Pointer to Line Numbers	
000002C0	0000	Number of Relocations	
000002C2	0000	Number of Line Numbers	
000002C4	E00000E0	Characteristics	
		00000020	IMAGE_SCN_CNT_CODE
		00000040	IMAGE_SCN_CNT_INITIALIZED_DATA
		00000080	IMAGE_SCN_CNT_UNINITIALIZED_DATA
		20000000	IMAGE_SCN_MEM_EXECUTE
		40000000	IMAGE_SCN_MEM_READ
		80000000	IMAGE_SCN_MEM_WRITE

[그림 10].vcp 섹션 헤더 분석(PEView)

[그림 11]은 .vcp의 raw data이다.

- source.exe
  - IMAGE_DOS_HEADER
  - MS-DOS Stub Program
  - IMAGE_NT_HEADERS
  - IMAGE_SECTION_HEADER .t
  - IMAGE_SECTION_HEADER .r
  - IMAGE_SECTION_HEADER .c
  - IMAGE_SECTION_HEADER .i
  - IMAGE_SECTION_HEADER .r
  - IMAGE_SECTION_HEADER .v
  - IMAGE_SECTION_HEADER .v
  - SECTION .text
  - SECTION .rdata
  - SECTION .data
  - SECTION .idata
  - SECTION .reloc
  - SECTION .vcp
  - SECTION .vcode

pFile	Raw Data	
0002A000	9C 55 81 EC 00 03 00 00	8B EC 81 EC 00 01 00 00
0002A010	89 85 00 FF FF FF 89 9D	04 FF FF FF 89 8D 08 FF
0002A020	FF FF 89 95 0C FF FF FF	89 B5 10 FF FF FF 89 BD
0002A030	14 FF FF FF 8B 85 00 03	00 00 89 85 18 FF FF FF
0002A040	8D 85 0C 03 00 00 89 85	1C FF FF FF 8B 85 08 03
0002A050	00 00 89 85 20 FF FF FF	E8 00 00 00 00 58 2D B5
0002A060	1B 40 00 89 85 30 FF FF	FF 05 29 1C 40 00 8B 00
0002A070	89 85 2C FF FF FF 8B 9D	20 FF FF FF 03 D8 89 9D
0002A080	20 FF FF FF 8B 85 30 FF	FF FF 05 2D 1C 40 00 8B
0002A090	00 03 85 2C FF FF FF 89	85 34 FF FF FF 8B 9D 30
0002A0A0	FF FF FF 81 C3 31 1C 40	00 E8 18 00 00 00 8B 0C
0002A0B0	83 03 8D 2C FF FF FF 8B	C5 2D 00 01 00 00 50 FF
0002A0C0	D1 83 C4 04 EB D7 33 C0	8B 8D 20 FF FF FF 8A 01
0002A0D0	C3 00 00 00 00 10 CF 42	00 00 C0 42 00 D9 C4 42
0002A0E0	00 0B C6 42 00 50 C7 42	00 00 C8 42 00 45 C9 42
0002A0F0	00 87 CA 42 00 9D CB 42	00 C7 CC 42 00 F8 CD 42
0002A100	00 CD CE 42 00 CC CC CC	CC CC CC CC CC CC CC CC

[그림 11].vcp 섹션의 Raw Data(PEView)

[그림 11]에서 빨강색으로 박스처리된 부분을 차례차례 설명하면,

첫번째 박스는 source_base 부분이고, 두번째 박스는 SPC-TPC 매핑 테이블의 주소를 나타낸다. 세번째 박스는 OPCode Table로써 핸들러의 주소를 담고 있다.

[그림 12]는 .vcode 섹션 헤더이다.

pFile	Data	Description	Value
000002C8	2E 76 63 6F	Name	.vcode
000002CC	64 65 00 00		
000002D0	00001000	Virtual Size	
000002D4	0002D000	RVA	
000002D8	000000B4	Size of Raw Data	
000002DC	0002B000	Pointer to Raw Data	
000002E0	00000000	Pointer to Relocations	
000002E4	00000000	Pointer to Line Numbers	
000002E8	0000	Number of Relocations	
de 0002EA	0000	Number of Line Numbers	
000002EC	E00000E0	Characteristics	
		00000020	IMAGE_SCN_CNT_CODE
		00000040	IMAGE_SCN_CNT_INITIALIZED_DATA
		00000080	IMAGE_SCN_CNT_UNINITIALIZED_DATA
		20000000	IMAGE_SCN_MEM_EXECUTE
		40000000	IMAGE_SCN_MEM_READ
		80000000	IMAGE_SCN_MEM_WRITE

[그림 12] .vcode 섹션 헤더 분석(PEView)

[그림 13]은 .vcode 섹션의 raw data이다.

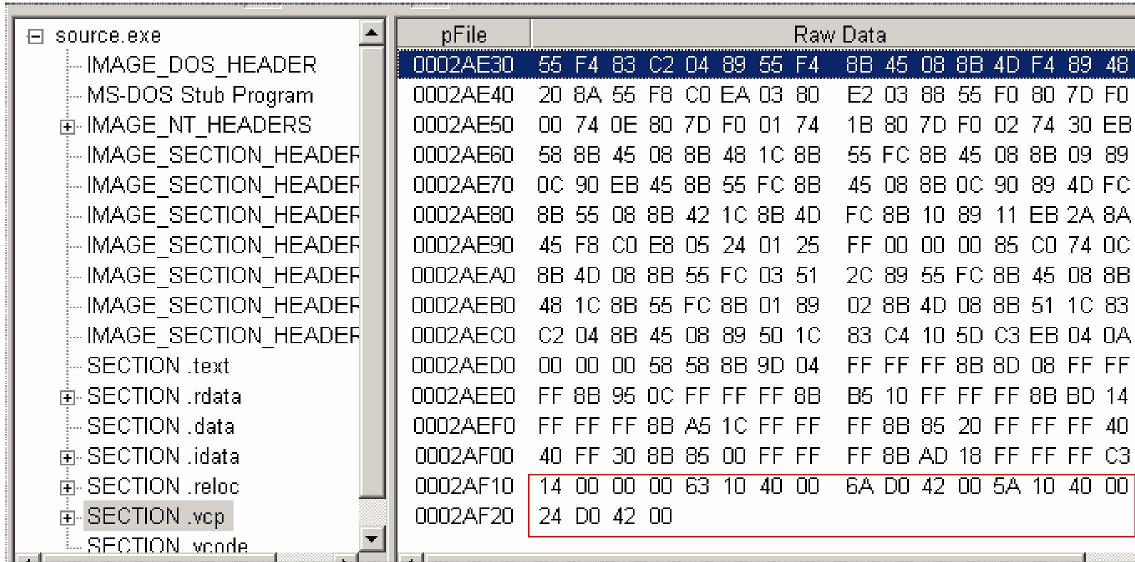
- source.exe
  - IMAGE_DOS_HEADER
  - MS-DOS Stub Program
  - IMAGE_NT_HEADERS
  - IMAGE_SECTION_HEADER .te
  - IMAGE_SECTION_HEADER .rc
  - IMAGE_SECTION_HEADER .d
  - IMAGE_SECTION_HEADER .id
  - IMAGE_SECTION_HEADER .re
  - IMAGE_SECTION_HEADER .vc
  - IMAGE_SECTION_HEADER .vc
  - SECTION .text
  - SECTION .rdata
  - SECTION .data
  - SECTION .idata
  - SECTION .reloc
  - SECTION .vcp
  - SECTION .vcode

pFile	Raw Data
0002B000	01 80 0A 00 00 00 06 00 00 00 02 83 0A 00 00 00
0002B010	04 00 00 00 01 8B 0A 00 00 00 00 00 00 00 03 B8
0002B020	63 10 40 00 01 80 0A 00 00 00 06 00 00 00 02 83
0002B030	0A 00 00 00 04 00 00 00 01 81 00 00 00 00 0A 00
0002B040	00 00 04 83 00 00 00 00 01 00 00 00 01 80 0A 00
0002B050	00 00 06 00 00 00 02 83 0A 00 00 00 04 00 00 00
0002B060	01 88 0A 00 00 00 00 00 00 00 01 80 0A 00 00 00
0002B070	06 00 00 00 02 83 0A 00 00 00 04 00 00 00 05 8B
0002B080	0A 00 00 00 0A 00 00 00 06 B8 78 10 40 00 08 B8
0002B090	1C 20 42 00 07 B8 00 11 40 00 68 A4 D0 42 00 E9
0002B0A0	5C EF FF FF 04 83 07 00 00 00 04 00 00 00 03 B8
0002B0B0	5A 10 40 00

[그림 13] .vcode 섹션의 Raw Data(PEView)

[그림 13]에서 빨강색으로 박스 처리된 부분은 Call External에 대한 Stub Code(인텔)이다.

[그림 14]는 SPC-TPC 매핑 테이블의 내용이다. (빨강색으로 박스처리된 부분)



[그림 14] SPC-TPC 매핑 테이블(PEView)

[그림 14]에 나타난 SPC-TPC 매핑 테이블을 분석하면 다음과 같다.

SPC-TPC 매핑 테이블

Size(4 Byte)	
SPC #1	TPC #1
...	...
SPC #n	TPC #n

[그림 14]에서 박스 처리된 바이너리를 분석하면, 20바이트(0x14)인 SPC-TPC 매핑 테이블인 것을 알 수 있다.

SPC	TPC
00401063	0042D06A
0040105A	0042D024

3.5 The Stub Code

test() 함수가 호출되면, 변환된 코드 영역을 Emulation하기 위해 VM으로 switching이 일어나야 하는데, 이러한 역할을 하는 Stub Code를 test() 함수 내에 추가한다.

00401038	. 68 00D04200	PUSH source.0042D000
0040103D	.-E9 BEAF0200	JMP source.0042C000
00401042	00	DB 00
00401043	00	DB 00
00401044	00	DB 00
00401045	00	DB 00
00401046	00	DB 00
00401047	00	DB 00
00401048	00	DB 00
00401049	00	DB 00
0040104A	00	DB 00
0040104B	00	DB 00
0040104C	00	DB 00
0040104D	00	DB 00
0040104E	00	DB 00
0040104F	00	DB 00
00401050	00	DB 00
00401051	00	DB 00
00401052	00	DB 00

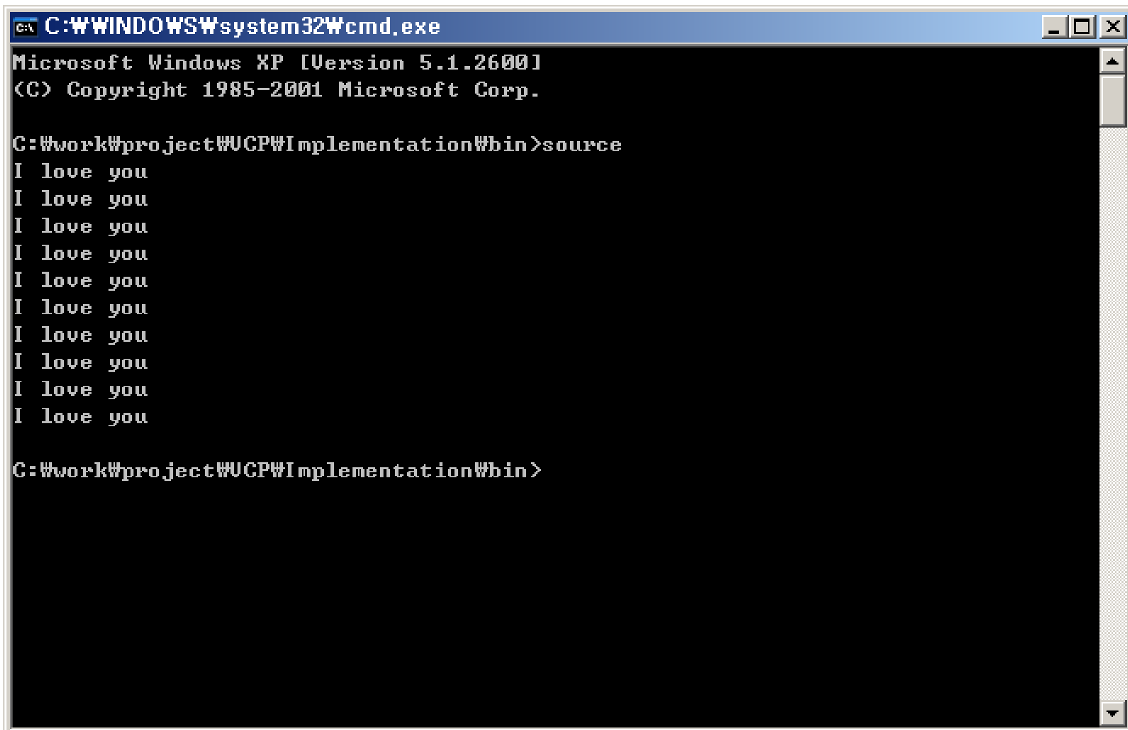
[그림 15] test 함수 내부의 Stub Code

[그림 15]의 코드 중, PUSH 0042d000의 0042d000은 .vcode 섹션의 첫 주소(변환된 코드 영역)를 의미하고, JMP 0042c000의 0042c000은 .vcp 섹션의 첫 주소(VM)를 의미한다.

따라서 42d000에 있는 변환된 코드를 VM에게 Emulation하라는 의미의 stub code 인 것이다.

3.6 I love you

[그림 16]은 source.exe의 test() 함수 코드를 Virtualized Code로 변환하여 이를 Emulation 한 결과이다.



[그림 16] VCP에 의해 변환된 source.exe의 출력 결과

지금까지 살펴본 테스트 바이너리(source.exe)는 BeistLab 홈페이지의 아래 링크를 통해 다운로드 할 수 있다.

http://beist.org/research/public/yong/vcp_source.exe.zip

4. References

1) Virtual Machines(JAMES E. SMITH, RAVI NAIR)

2) IA-32 Instruction Set Reference

3) Virtual Machine Rebuilding(Maximus)

3) An In-Depth Look into the Windows PE File Format(Matt Pietrek)

4) Inject your code to a Portable Executable file(Ashkbiz Danehkar)

Thanks to Beist For Helping and publishing my document.

I'm BeistLab friend!.