

Code Virtualized

Anti Reversing Technique

sonicpj@nate.com

ezbeat.tistory.com

박지훈



Preface

이번 기술문서를 작성하게 된 계기는 코드게이트 해킹대회를 하면서 새로운 안티리버스 기법을 접해보았기 때문입니다. 바로 그 기법이 코드 가상화란 기법인데 이에 대한 자세한 설명은 문서를 써가면서 차차 설명하도록 하겠습니다. 이 코드 가상화 기법은 아주 다양하게 변형되고 만드는 사람 마음대로 바뀌서 사용할 수 있습니다. 이런 점에서 다른 안티리버스 기법보다 더 매력을 느끼는 점 인 것 같습니다. 반대로 그 만큼 개발자가 마음대로 주물럭거릴 수 있는 만큼 구현도 다른 방법보다 더 어렵다는 것은 피할 수 없는 사실인거 같구요.

실제로 이 문서를 쓰기 전에 코드 가상화를 사용한 안티리버스 기법에 대해서 잘 쓰여진 문서가 하나 있습니다. 작성자는 이용일님이시구 문서 제목은 “Design and Implementation of Virtualized Code Protection(VCP) For Anti-Reverse Engineering”입니다. 이 문서를 처음부터 끝까지 다 읽고 실습까지 해본결과 저도 제가 나름대로 이해한다고 문서를 제 구성을 해보고 싶었습니다. 복습도 되고 나중에 이 기법을 사용한 CrackMe 같은 해킹 문제 만들 기라던지 아니면 실제 상황에서 적용할만한 기회가 올지도 모르기 때문이죠. 이번 문서에서는 제가 읽은 문서에서 사용한 코드 가상화 기법을 토대로 필요한 부분은 조금 수정하는 방식으로 작성하도록 하겠습니다. 또한 이미 이 방법을 사용해 코드 가상화 발표를 전국 CERT 연합 회의 U3에서 발표를 한 적이 있습니다. 그 때 발표를 하면서 다시 제대로 문서를 만들어서 올린다고 했는데 이제야 만들게 되네요. 이점 죄송스럽게 생각합니다. 그리고 이 기법을 사용해 발표나 기술문서를 쓰는데 있어서 위 문서 작성자이신 이용일님에게 미리 다 양해를 구하고 허락을 받은 상태입니다. 흔쾌히 허락해 주셔서 감사합니다.

문서를 작성하면서 최대한 초보자도 쉽게 볼 수 있게끔 작성을 시작해 볼 것인데 어렵게 느껴지신다면 저에게 “왜 이렇게 문서를 만들었어요??” 라고 따끔하게 지적바랍니다. 까이는걸 은근히 즐기는 저...:: 주말에 서점을 가서 이것저것 책을 둘러보다가 호감대화법이라는 책이 있었습니다. 거기서 나온 말인데 “진정한 실력자는 상대에게 쉽게 설명한다는 것”입니다. 그런다고 제가 진정한 실력자란 말이 아니라 저도 하나하나 알아가는 입장이므로 어렵게 설명하는 것이 오히려 부담이고 일이므로 쉽게 설명하겠다는 것입니다. 그리고 대부분 사람들이 이해하기 쉽고 선호하는 Top-down approach 방법을 사용해 문서를 작성해 보도록 하겠습니다. 전체적인 그림을 먼저 그려본 후 하나씩 파헤쳐가는 방향을 말한 것입니다. 또한 본 문서에서의 모든 내용은 Intel CPU의 OS는 Windows 관점으로 작성되었습니다.

본 문서에서는 편의상 반말로 작성을 하도록 하겠습니다. 양해부탁 드립니다.

목차

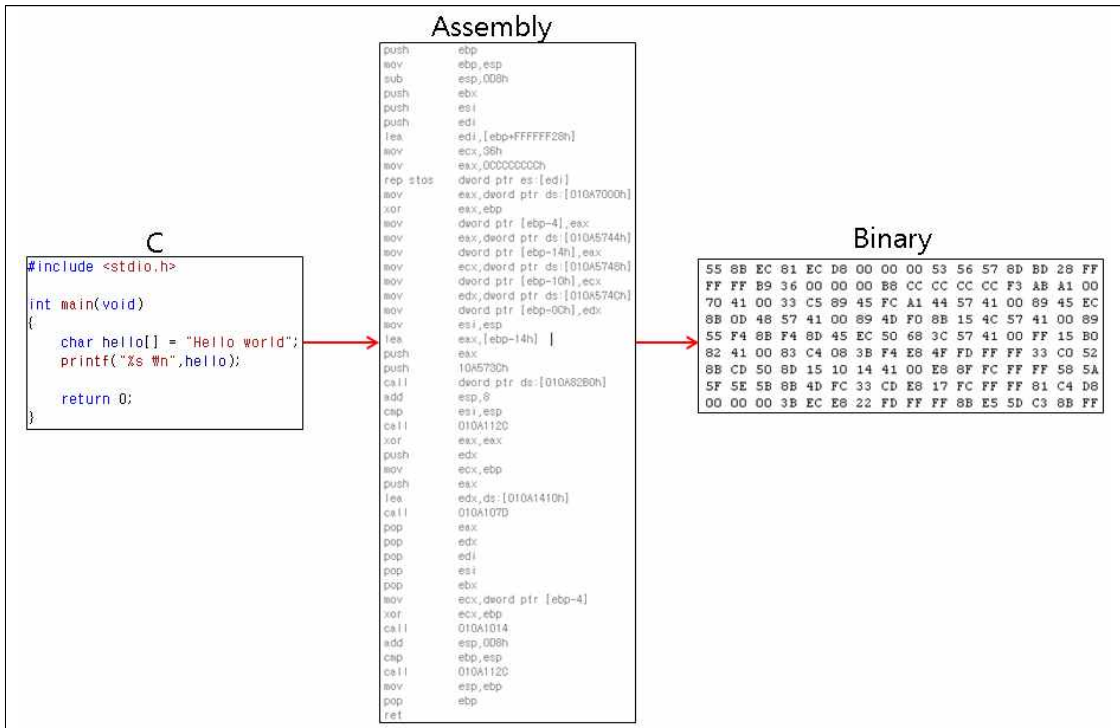
1. Introduction	3
1.1 Reversing	3
1.2 Anti Reversing	5
1.2.1 Function	5
1.2.2 Technique	7
1.2.3 Obfuscation	7
2. Code Virtualized	9
2.1 Intro Code Virtualized	9
2.2 Overall Structure	10
2.3 Change Instruction	14
2.4 Virtual CPU	19
2.4.1 Control Code	19
2.4.2 OPCODE Handler	22
2.5 Implementation	32
2.5.1 Create Section	33
2.5.2 Insert VCPU, VCODE	34
2.6 Execution	38
3. Conclusion	39
4. Reference	40

1. Introduction

이 문서를 작성하기 전 Reversing와 Anti Reversing에 대해서 간략히 알아볼 것이다. 이에 대한 내용은 어느 책이던 인터넷을 뒤져봐도 아주 방대한 양의 내용이 나올 것이므로 자세한 설명은 생략하겠다.

1.1 Reversing

Reversing 은 reverse와 engineering의 합친 말이다. 즉, 역으로 무언가를 분석하는 기술을 말하는 것이다. Reversing은 여러 분야에서 많이 사용되지만 우리가 중점적으로 관심을 두어야할 부분은 소프트웨어 reversing이다. 소프트웨어라 함은 exe가 될 수도 있고 dll, ocx, sys 등등 대부분 실행 파일이 될 수 있다. 그러면 reversing은 어떤 식으로 진행이 되는 것인가? 보통 소프트웨어를 개발할 때에는 프로그래밍 언어를 가지고 소스코드를 작성 한 후 컴파일러에서 해당 소스코드를 컴퓨터(정확히는 processor)가 이해할 수 있는 언어 (binary code)로 변경하는 작업을 거치게 된다. 이 때 중간에 어셈블리 언어로 바뀌고 다시 어셈블러가 바이너리 코드로 바꿔주는 작업도 있다. 이를 간략한 그림으로 나타내보겠다.



[그림 1] C코드 간략한 컴파일 과정

위 그림은 컴파일 과정을 아주 간략하게 나타낸 것이다. 위 그림처럼 C -> Assembly -> Binary 순으로 컴파일 되는 것이 정상적인 과정이다. 하지만 Reversing은 그 역과정이다. 즉, Binary 코드를 Disassembly 해주는 툴을 사용해 assembly 코드로 해당 바이너리 코드를 볼 수 있는 것이다. 위 바이너리 파일을 windows에서 가장 대표적으로 꼽히는 reversing 툴인 ollydbg로 열어서 봐보았다.

```

PUSH EBP
MOV EBP,ESP
SUB ESP,008
PUSH EBX
PUSH ESI
PUSH EDI
LEA EDI,DWORD PTR SS:[EBP-08]
MOV ECX,36
MOV EAX,CCCCCCC
REP STOS DWORD PTR ES:[EDI]
MOV EAX,DWORD PTR DS:[10A7000]
XOR EAX,EBP
MOV DWORD PTR SS:[EBP-4],EAX
MOV EAX,DWORD PTR DS:[10A5744]
MOV DWORD PTR SS:[EBP-14],EAX
MOV ECX,DWORD PTR DS:[10A5748]
MOV DWORD PTR SS:[EBP-10],ECX
MOV EDX,DWORD PTR DS:[10A574C]
MOV DWORD PTR SS:[EBP-C],EDX
MOV ESI,ESP
LEA EAX,DWORD PTR SS:[EBP-14]
PUSH EAX
PUSH Study_Co_010A573C
CALL DWORD PTR DS:[&MSVCR100D.printf]
ADD ESP,8
CMP ESI,ESP
CALL Study_Co_010A112C
XOR EAX,EAX
PUSH EDX
MOV ECX,EBP
PUSH EAX
LEA EDX,DWORD PTR DS:[10A1410]
CALL Study_Co_010A107D
POP EAX
POP EDX
POP EDI
POP ESI
POP EBX
MOV ECX,DWORD PTR SS:[EBP-4]
XOR ECX,EBP
CALL Study_Co_010A1014
ADD ESP,008
CMP EBP,ESP
CALL Study_Co_010A112C
MOV ESP,EBP
POP EBP

```

[그림 2] OllyDbg로 열어본 binary file

assembly code에 익숙한 사람이라면 위 코드만 보고도 어떠한 코드인지 쉽게 이해할 것 이다. 그렇다고 assembly를 꼭 잘해야만 reversing을 할 수 있는 것은 아니다. reversing을 할 때 사용되는 명령어는 intel cpu기준으로 수천 개 중 30~40개 정도 밖에 되지 않으며 그 중에서도 많이 쓰이는 명령어는 10~20개 내외이다. 그렇기 때문에 시간을 조금만 투자해 계 속적으로 분석을 해보면 충분히 위 코드를 읽는데는 별 지장이 없을 것이다. 하지만 assembly 코드를 직접 작성할 때는 어느 정도 assembly를 공부해야할 것이다. 왜냐하면 코 드를 작성하는데 있어서 규칙들이 있기 때문이다. 적절한 비유를 들라면 영어에서 독해와 작 문의 차이라고 해도 될 것 같다. 작문이 더 쉽게 느껴지시는 분도 계실려나..

그리고 위 assembly코드를 더 상위 언어인 C언어 같은 형태로도 바꿀 수 있는데 이때 2가지 방법이 존재한다. 첫 째는 hand-ray라는 방법인데 이 방법은 직접 어셈 코드를 한줄 한줄 따 라가면서 직접 사람이 바꾸는 방법이고, 두 번째는 IDA 툴에 존재하는 hex-ray라는 툴을 사

용하는 것이다. 이는 assembly -> C 로 바꿔주는 자동화된 툴인데 소프트웨어를 분석할 때 상당히 유용하게 사용되는 IDA의 플러그인이다. 그렇다고 hand-ray 능력이 필요 없는 것은 아니다. 진정한 리버서라면 hand-ray정도는 할 줄 알아야 된다고 생각한다.

지금까지 대충 소프트웨어 reversing이 어떠한 방법으로 시작이 되는지 알아보았다. 진정한 reversing을 하려면 많은 경험과 노하우 테크닉이 필요한데 이는 스스로 깨우치기를 바라는 마음에서 독자들의 숙제로 남겨두겠다. 필자가 생각하는 리버서는 breakpoint를 거는 능력이 뛰어날수록 뛰어난 리버서라고 생각된다. 단순히 ollydbg에서 F2를 빨리누르는 것을 말하는 것이 아니다...

이제 이러한 reversing이 어디에 사용되는지에 대해서 알아보자.

필자는 독자들이 쓰는 프로그램이 어떠한 프로그램들이 있는지 궁금하다. 가장 첫 번째로 사용하고 있는 소프트웨어는 아마 OS일 것이다. Windows던 Linux던.. Windows 같은 경우는 라이선스를 돈을 주고 산 후 사용해야 할 것인데 아마 독자들 중 인터넷에 돌아다니는 windows keygen을 사용해 인증을 해서 사용하시는 분들도 계실 것이다. 필자도 한 때 그렇게 이용해 본적이 있다. 그리고 OS 뿐만 아니라 다른 라이선스를 필요로 하는 프로그램들을 이용하는데 전부 돈을 주고 구매해서 사용하는가? 아마 이 모든 것을 돈을 주고 집에서 정품을 사용하는 사람은 거의 없을 거라고 생각된다. 그러면 이러한 keygen이나 크랙판 프로그램을 내놓는 사람들은 누구일까?? 그리고 어떠한 방법에 의하여 이런 것을 만들어 낼 수 있을까?? 바로 크래커가 리버싱이라는 기술을 사용해 크랙을 하는 것이다.

이러한 리버싱 기술이 악용하는 곳에만 쓰이는 것은 아니다. 현재 악성코드를 분석하는 리버서들 또한 리버싱이라는 기술을 유용하게 사용하고 있다. 이렇듯 리버싱이라는 기술을 사용하면 실행파일을 가지는 순간 해당 소프트웨어의 소스코드를 얻었다고 해도 과언이 아니다. 그렇게 되면 그 파일을 마음대로 수정해 버려 악용을 할 수도 있다는 것이다. 그러면 이러한 리버싱을 막는 방법을 없을까?? 그 다음 1.2절에서 이에 대한 방법에 대해 또 간략히 소개를 할 것이다. 이 문서의 본 목적은 코드 가상화이기 때문이다.

1.2 Anti Reversing

1.1절에서 Reversing이 무엇인지 알아보았다. 그리고 이번엔 이러한 리버싱을 못하게 막는 기술들에 대해서 알아 볼 것인데 이러한 기술을 통 틀어서 Anti Reversing 기술이라고 하는 것이다. 이러한 기술은 상용 소프트웨어 개발자에게도 유용하게 사용되고 또한 악성코드 제작자에게도 유용하게 사용된다. 보안업체에서 자기가 만든 악성코드를 분석하지 못하게끔 방해 를 하기 위해서이다. 이러한 기술을 어떻게 사용할지는 독자들이 알아서 생각하시길 바란다. 필자는 기술만 소개할 것이다.

1.2.1 Function

함수를 이용한 방법이다. 가장 Anti Reversing 함수 중 대표적인 API 함수를 꼽으라면 IsDebuggerPresent()함수가 있다. 하지만 가장 대표적인 함수이기 때문에 이 함수를 막는 방법은 널리 널리 알려져 있으며 분석 중에 자동으로 우회가 되도록 플러그인도 다 만들어져 있기 때문에 잘 사용되는 방법은 아니다. 그래도 대표적인 함수이므로 이 함수만 소개하고 넘어

가도록 하겠다. 해당 함수는 프로세스가 디버거에 붙어 있을 경우 1을 아닐 경우 0을 리턴하는 함수이다. 이를 적절히 이용해 리버싱을 방해할 수 있을 것이다.

함수 내부를 봐보겠다.

```
MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+30]
MOVZX EAX,BYTE PTR DS:[EAX+2]
```

[그림 3] IsDebuggerPresent()

```
ntdll!_TEB
+0x000 NtTib : _NT_TIB
+0x000 ExceptionList : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x000 Next : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler : Ptr32 _EXCEPTION_DISPOSITION
+0x004 StackBase : Ptr32 Void
+0x008 StackLimit : Ptr32 Void
+0x00c SubSystemTib : Ptr32 Void
+0x010 FiberData : Ptr32 Void
+0x010 Version : Uint4B
+0x014 ArbitraryUserPointer : Ptr32 Void
+0x018 Self : Ptr32 NT_TIB
+0x000 ExceptionList : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 StackBase : Ptr32 Void
+0x008 StackLimit : Ptr32 Void
+0x00c SubSystemTib : Ptr32 Void
+0x010 FiberData : Ptr32 Void
+0x010 Version : Uint4B
+0x014 ArbitraryUserPointer : Ptr32 Void
```

[그림 4] mov eax, FS:[0x18]

FS:[0x18]은 TEB 자기 자신을 가리키고 있는 포인터이다. C++에서 this를 생각하시면 될 것 같다.

```
ntdll!_TEB
+0x000 NtTib : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId : _CLIENT_ID
+0x028 ActiveRpcHandle : Ptr32 Void
+0x02c ThreadLocalStoragePointer : Ptr32 Void
+0x030 ProcessEnvironmentBlock : Ptr32 _PEB
+0x034 LastErrorValue : Uint4B
+0x038 CountOfOwnedCriticalSections : Uint4B
+0x03c CsrClientThread : Ptr32 Void
+0x040 Win32ThreadInfo : Ptr32 Void
```

[그림 5] mov eax, [eax+0x30]

그 자기 자신에서 0x30만큼 떨어진 위치의 값을 가져오고 있는데 해당 값은 PEB구조체 이다.

```
ntdll!_PEB
+0x000 InheritedAddressSpace : UChar
+0x001 ReadImageFileExecOptions : UChar
+0x002 BeingDebugged : UChar
+0x003 BitField : UChar
+0x003 ImageUsesLargePages : Pos 0, 1 Bit
+0x003 IsProtectedProcess : Pos 1, 1 Bit
+0x003 IsLegacyProcess : Pos 2, 1 Bit
+0x003 IsImageDynamicallyRelocated : Pos 3, 1 Bit
```

[그림 6] movzx eax, [eax+0x2]

PEB구조체에서 0x2만큼 떨어진 곳에는 BeingDebugged라는 구조체 멤버가 존재하며 해당 값이 디버깅 시에는 1 값으로 세팅이 되어 있다. 그 값은 eax에 넣으면서 리턴을 하는 것이다. 이를 막아주는 플러그인들을 보면 이 값을 0으로 바꿔버리는 플러그인도 있고 아니면 해당 함수를 인라인 후킹해서 그냥 점프시켜버리는 플러그인들도 있다.

대부분 함수들은 그 함수만 딱 딱 써서 리턴 값 혹은 인자 값들을 체크함으로써 안티 리버싱 기법을 구현하고 이 방법을 뚫는 방법 또한 리턴 값 조작 혹은 인자 값 조작만 하면 되므로 우회 방법 또한 간단하다.

이러한 함수들은 많이 있지만 여기서 다 설명하는 것은 본 문서의 주제에 어긋나므로 뒤 참고 문서에 문서 제목을 적어놓도록 하겠다. 해당 문서를 보면 관련 함수들과 예제 소스도 있으며 1.2.2 절에서 설명할 technique에 대한 내용과 예제 소스 또한 전부다 있으므로 꼭 읽어보면 도움이 될 것이다.

1.2.2 Technique

이 방법은 단순히 함수하나만 써서 안티 리버싱 기법을 구현하는 것이 아니다. 구현하는 함수 하나하나만 보면 안티 리버싱과 관련이 없는 함수들이지만 각 함수들을 사용해 안티 리버싱 기술을 구현하는 것이다. 간단한 예제를 봐보자.

보통 ollydbg를 키게 되면 프로세스 명이 OLLYDBG.EXE이다. 이렇게 현재 켜져 있는 프로세스 명을 검사해 OLLYDBG.EXE가 존재하면 죽여버리는 코드를 작성하면 된다.

해당 코드를 작성하는데 사용하는 함수는

CreateToolhelp32Snapshot()

Process32First()

Process32Next()

OpenProcess()

TerminateProcess()

이렇게 총 4개의 API함수만 사용하면 구현할 수 있다.

예제 코드는 뒤 참고문서를 보시면 될 것 같다.

이 방법 외에도 프로세스 attach를 막기 위한 DebugActiveProcess 함수 후킹, RDTSC 어셈블리 명령어를 사용해 프로세스 실행 시간 체크, SEH핸들러를 사용한 안티리버싱 기법 등등 다양한 방법이 존재하므로 독자들도 직접 자기만의 방법을 사용해 생각나는 대로 만들어보면 재미있을 것이다.

1.2.3 Obfuscation

마지막으로 소개할 방법은 난독화라는 방법인데 제목에서도 알 수 있듯이 assembly코드를 읽기 어렵게끔 코드를 난독화 시켜버리는 것이다. 보통 안티 리버싱 기법을 적용하면 100% 리버싱으로부터 안전하다는 생각을 가질 수 있다. 하지만 이는 철저히 잘못된 생각이며 안티 리버싱 기법이 있으면 그걸 우회하는 방법 또한 계속 적으로 나오기 마련이다. 그래서 리버싱

을 못하게 하는데 있어서 초점은 바로 “리버싱을 수행하는데 얼마나 많은 시간을 걸리게 할 것인가?” 가 바로 안티 리버싱의 목적이다. 리버서들도 결국 사람이며 인내심의 한계가 존재하며 체력의 한계도 존재한다. 그렇기 때문에 리버싱을 하는데 있어서 너무 많은 시간이 걸리게 되면 대부분 리버싱을 포기하게 되는 것이다. 이럴 때 가장 유용하게 사용되는 방법이 난독화다. 이 방법은 시간을 투자해서 읽어나가는 수밖에 없다. 딱히 우회 방법이 존재하지 않기 때문이다. 이러한 코드 난독화 방법에도 Stack Based, Arithmetic, Logical 등등 여러 방법이 존재하는데 이 방법에 대해서는 뒤 참고문서에 올려놓도록 하겠다.

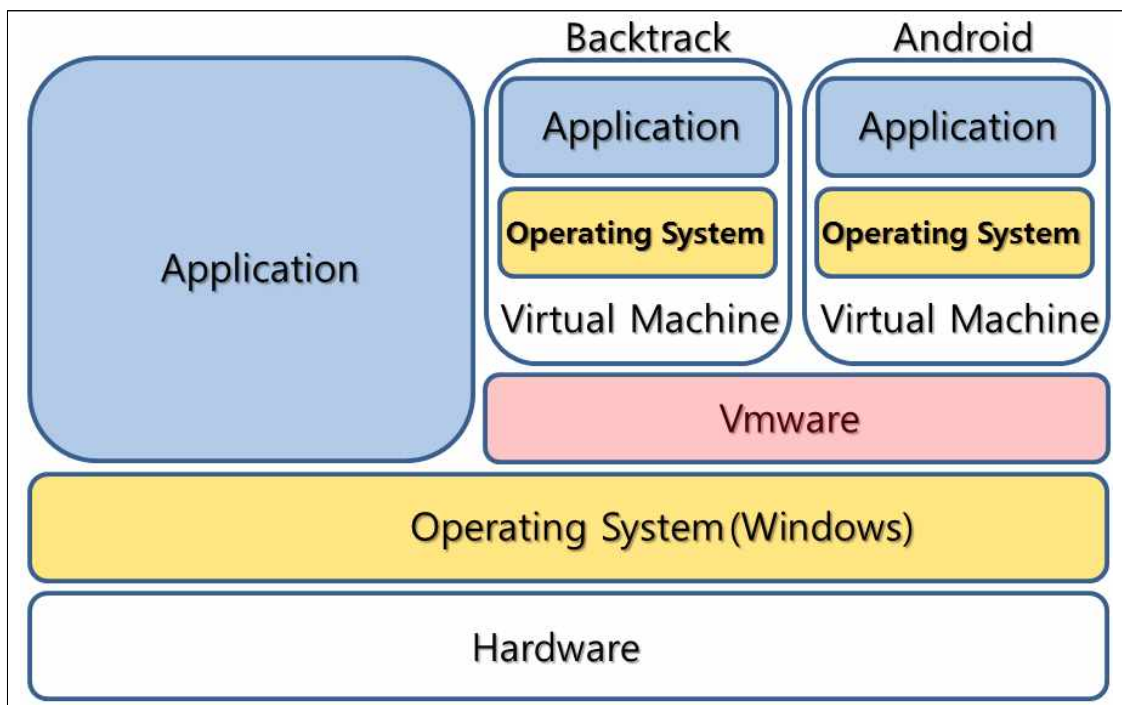
드디어 소개를 하게 된다. 이러한 난독화 방법 중 하나인 코드 가상화를 사용한 방법이 있는데 이 방법이 바로 본 문서의 주요 내용이다. 필자의 생각으로는 이 방법을 사용한 안티 리버싱 기술은 안티 리버싱 기술 중 상급에 속하는 기술이라고 말할 수 있다. 그만큼 다른 기법들보다 구현 방법이 어렵다는 것이 단점 아닌 단점이다. 코드 가상화 기술을 난독화의 하위 주제로 놔두려다가 이 자체가 본 문서의 주 주제이므로 큰 타이틀로 따로 빼서 설명을 하도록 하겠다.

2. Code Virtualized

코드 가상화는 안티 리버싱 기술 중 하나이다. 말 그대로 코드를 가상화 시켜서 리버싱을 하는데 있어서 방해하는 것이다. 이제 이 방법에 대해서 차례로 알아보기로 하자.

2.1 Intro Code Virtualized

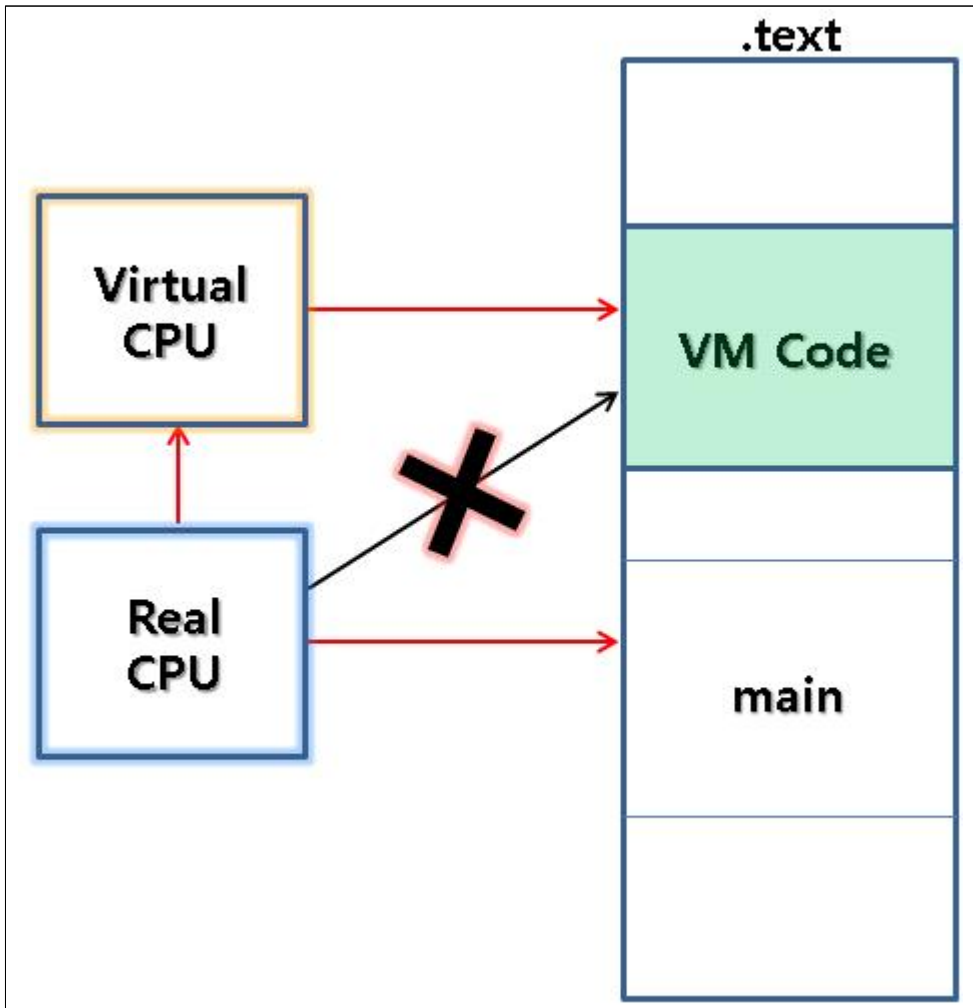
코드 가상화를 알아보기 전에 가상화가 무엇인지에 대해 잠시 알아보자. 가상화라는 말이 들어봤을 것이다. Vmware, VirtualBox 이런 것들도 다 가상화 머신들이고 프로그래밍에서도 java 같은 경우도 바로 processor가 명령어들을 처리하는 것이 아닌 따로 Java Virtual Machine(JVM)을 뒤서 해당 실행 엔진이 자바 바이트코드를 실행시키는 것이다. 그렇기 때문에 모든 자바 프로그램은 CPU나 OS의 종류와 무관하게 동일하게 동작한다는 것이 보장된다. 잠깐 그림을 봐보자.



[그림 7] Virtual Machine 간략한 그림

위 그림은 가상 머신의 간략한 그림을 나타내보았다. Vmware라는 툴을 사용해 Windows에서 여러 가지 OS를 동시에 사용하고 있다. 그리고 해당 OS들은 하드웨어와 통신을 하기 위해서 Windows를 거쳐서 작동을 하게 되는데 이 때 그 사이에 Vmware가 중간자 역할을 해주는 것이다. Windows에서는 바로 Backtrack을 실행시킬 수 없으며 Android 또한 바로 실행시킬 수 없기 때문이다.

이처럼 코드 가상화도 마찬가지이다. 가상화된 코드가 있으면 CPU가 바로 그 명령어를 실행하는 것이 아닌 중간에 Virtual CPU가 대신해서 해당 명령어를 실행해 주는 것이다. 당연히 Virtual CPU를 작동시키는 것은 실제 CPU가 된다. 간략한 그림으로 봐보자.



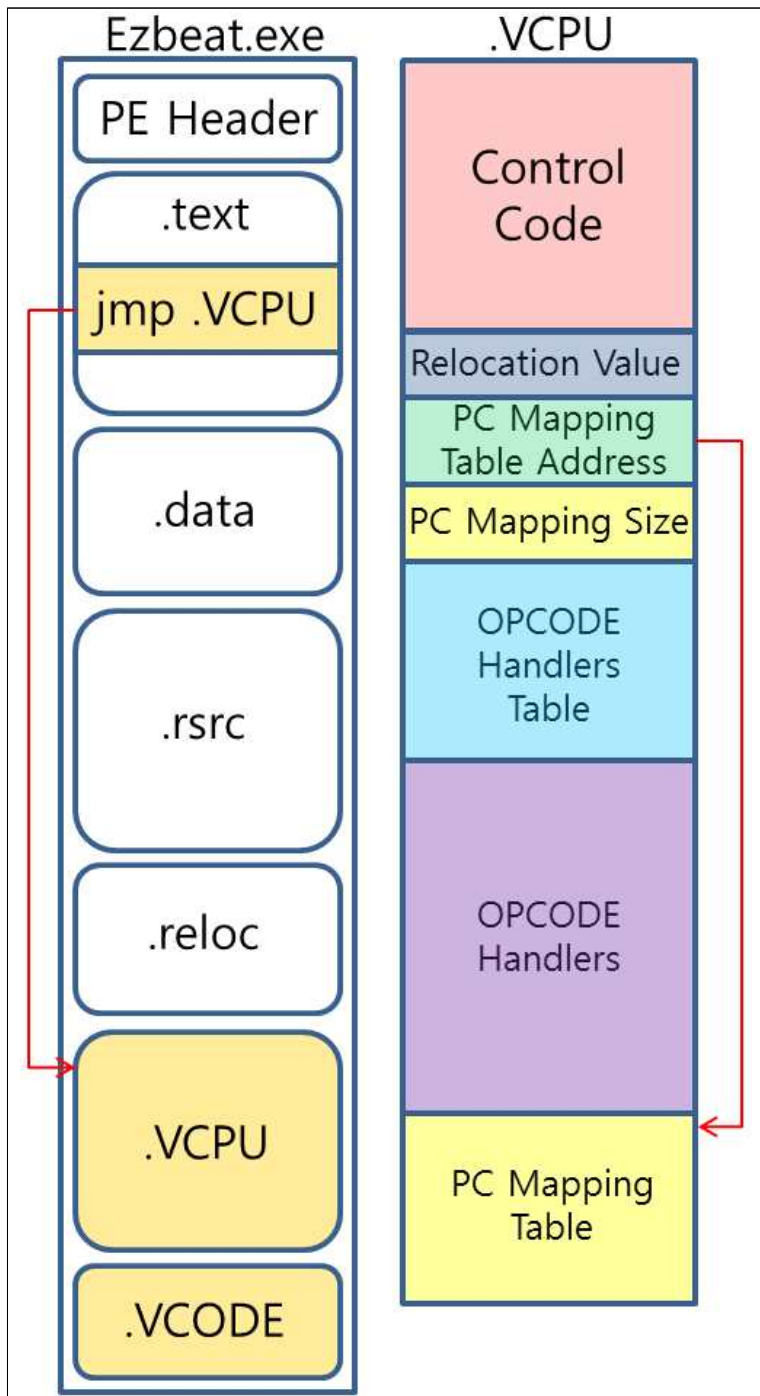
[그림 8] 가상화된 코드 실행 모식도

위 그림을 보면 가상화된 코드는 실제 CPU에서 실행시킬 수 없다. 왜냐하면 코드가 가상화가 되어서(VM Code) 실제 CPU에서는 해석할 수 없는 명령어 체계로 이루어져 있기 때문이다.

이제 이러한 기술을 구현하기 위해 하나씩 하나씩 만들어보겠다.

2.2 Overall Structure

실제 구현에 들어가기에 앞서 전체적인 그림을 한번 그려보고 가도록 하겠다. 항상 어떤 공부를 하다보면 하나하나 깊게 보고 있으면 전체적인 그림을 놓쳐서 제대로 이해가 안가는 경우가 허다하다. 필자의 머리가 좋지 않아 그런 것일 수도 있지만 이번에도 또한 문서를 작성하다가 빼먹거나 이해가 안갈 수도 있으므로 필자를 위해서 그림을 한번 그려보고 넘어가도록 하겠다. 이 그림이 코드 가상화를 하는데 있어서 기본 틀이 아님을 먼저 말해두고 싶다. 코드 가상화를 하는데 있어서 틀은 만드는 사람마다 다르게 정의할 수 있으며 여기서 보여주는 틀은 이 문서에서 설명할 코드 가상화 기법을 설명할 틀이라는 것을 알아두고 있으면 좋겠다.



[그림 9] 가상화 적용된 프로그램 & Virtual CPU 내부 구조

처음으로 봐볼 것은 가상화가 적용된 프로그램 내부 형태이다.

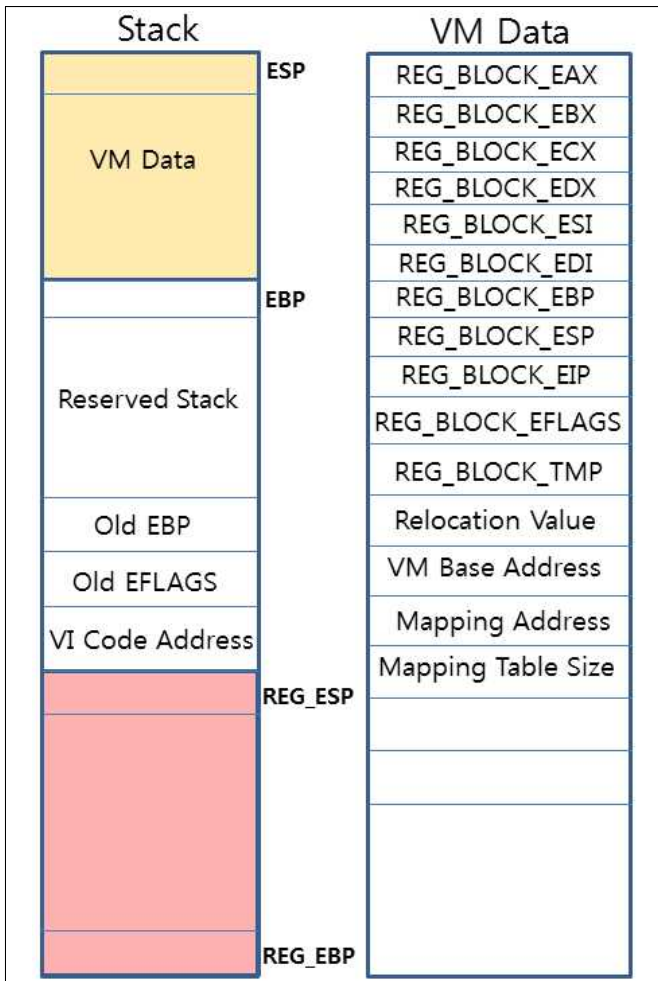
자세한 설명은 뒤쪽에서 하기로 하고 각각이 무엇을 나타내고 있는지만 적고 넘어가도록 하겠다.

.VCPU	가상화된 코드를 실행시키기 위한 Virtual CPU의 코드를 가지고 있는 섹션
.VCODE	가상화 시킨 코드를 넣어두는 섹션

.VCPU 내부

Control Code	Virtual CPU를 위한 스택 구성 & 데이터 구성 & 명령어 핸들러 호출
Relocation Value	Imagebase가 바뀌었을 경우 주소 값 조정을 위한 값
PC Mapping Table Address	PC란 Program Counter를 뜻하며 가상화 시킨 코드는 VCODE라는 섹션에 따로 저장을 하기 때문에 함수 호출 명령어나 분기문 주소들이 달라지게 된다. 그렇기 때문에 해당 분기문의 올바른 주소를 담아놓기 위한 테이블을 가리키는 주소
OPCODE Handlers Table	VCPU에서 처리 가능한 명령어들의 핸들러들이 있는 VA 값
OPCODE Handlers	실제 VCPU에서 처리 가능한 명령어 핸들러들
PC Mapping Size	Mapping Table 사이즈를 나타냄
PC Mapping Table	위에서 설명했듯이 정상적인 위치로 분기를 하기 위한 주소들이 있는 테이블. 각 테이블 엔트리는 오리지널 주소(4Byte) 바뀔 주소(4Byte)로 이루어져 있는 구조체로 정의했다.

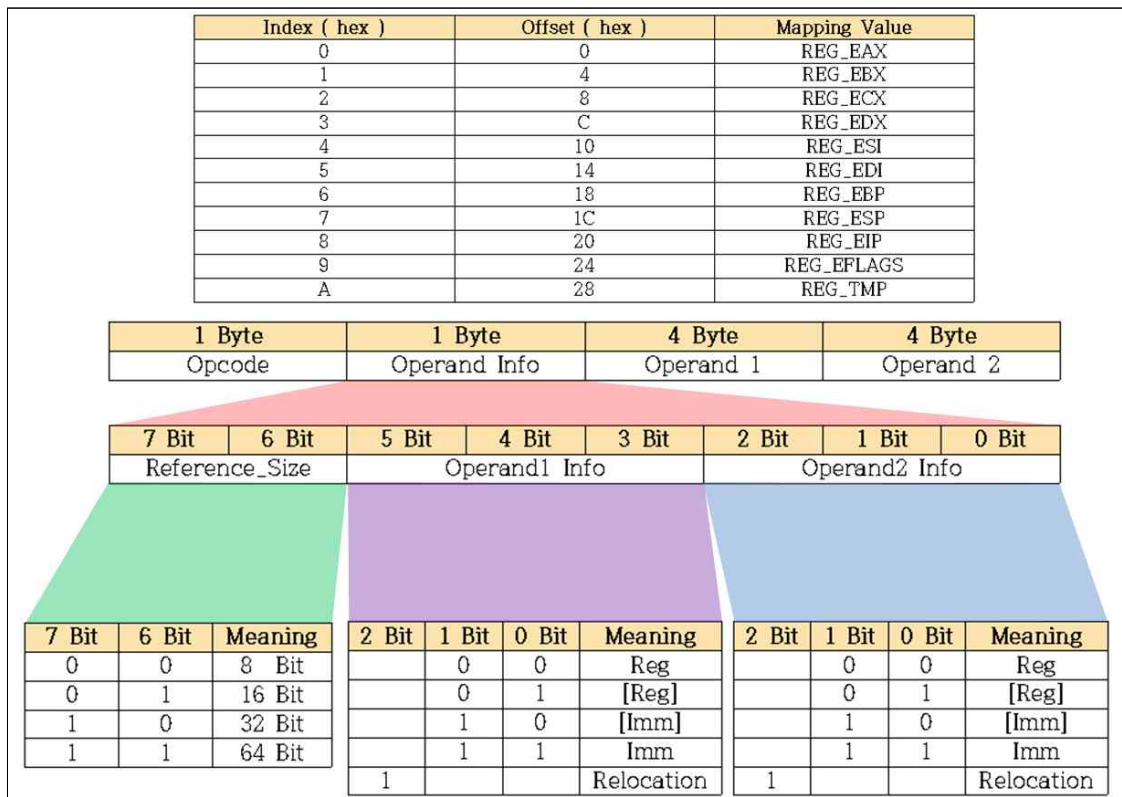
이제 Real CPU에서 Virtual CPU로 제어가 넘어왔을 때 재구성 되는 스택을 봐보겠다.
 스택 재구성 시키는 코드는 Control Code 처음 부분에 존재한다.



[그림 10] 스택 구성도와 VM Data에 들어 있는 내용

REG라고 앞에 붙은 것은 Virtual CPU에서 사용되는 레지스터나 플래그들이다. 빨간색으로 된 부분은 Real CPU에서 사용되던 스택 영역과 같은 부분이다. 그리고 주황색으로 된 부분엔 Virtual CPU에서 사용되는 레지스터와 상태 값들이다. 옆 그림을 보면 Relocation Value, VM Base Address, Mapping Table Address, Mapping Table Size 들의 값도 전부 들어있는 것을 확인 할 수 있다. 해당 값들은 OPCODE Handler에서도 사용되므로 VM Data 영역에 넣어 둔 것이다. 실제 구성되는 코드는 뒤쪽에서 보도록 하겠다.

마지막으로 Virtual CPU에서 처리할 수 있는 명령어(Instruction) format을 봐보자.



[그림 11] Virtual CPU에서 처리 가능한 Instruction Format

일단 형태는 위와 같이 생겼고 명령어 구조는 Intel이나 ARM, MIPS에서 처리하는 명령어 구조와는 차원이 다르게 간략하다. 복잡하게 만들수록 해당 명령어를 처리해야할 OPCODE Handler의 각각의 코드들 또한 엄청나게 길어지고 복잡하게 된다. 그러면 단점은 무엇일까?? 처리 가능한 명령어 형태가 간단하다는 것이다.

간단한 예를 들어보겠다. Intel 같은 경우는 index를 사용한 연산이 가능하다.

```
mov [ebp-0x14],ebx ; ebp-0x14 위치에 있는 주소가 가리키는 곳에 ebx 값 넣어라
```

하지만 우리가 만들 Virtual CPU에서는 [그림 11] 명령어 체계를 사용하기 때문에 위 명령어는 처리 할 수 없다. 이에 대해 처리하는 방법에 대해서는 뒤쪽에서 알아보도록 하겠다.

이제 구성해야할 전체적인 그림을 다 보았다. 종합해서 우리가 만들어야할 내용을 봐보자.

1. 오리지널 어셈 코드를 어떻게 변경할 것인가?
2. Virtual CPU는 어떻게 만들 것인가?
3. 만든 Virtual CPU를 어떻게 타겟 프로그램에 넣어서 실행을 시킬 것인가?

가장 내용이 많아질 부분은 2번 인 것 같다. 일단 차례대로 알아보도록 하겠다.

2.3 Change Instruction

가장 먼저 변경할 부분은 오리지널 어셈 코드를 Virtual CPU가 처리할 수 있는 명령어 구조로 바꾸는 작업을 먼저 해보겠다.

실제 코드를 바꾸기 전에 몇 가지 예제를 보도록 하자.

위에서 설명하지 않고 넘어갔던

```
mov [ebp-0x14],ebx
```

이 코드를 바꿔보도록 하겠다.

Virtual CPU에서 알아먹을 수 있는 명령어 셋으로 바꾸면 다음과 같이 된다.

Original Code	VM Code
mov [ebp-0x14],ebx	mov tmp,ebp sub tmp,0x14 mov [tmp],ebx

offset 연산을 지원하지 않기 때문에 tmp(REG_TMP)라는 임시 레지스터를 하나 더 만들어 두고 해당 레지스터를 활용하는 것이다.

이제 실제로 변경할 코드를 봐보자.

먼저 프로그램을 하나 만들어보았다.

```
#include <stdio.h>

int main(void)
{
    int value = 0;
    printf("Code Virtualized Test!! \n");
    printf("http://ezbeat.tistory.com \n");

    scanf("%d",&value);

    value ^= 0xffffffff;
    printf("value : %d \n",value+1);

    return 0;
}
```

[그림 12] 타겟 프로그램 소스 코드

위 프로그램은 단순히 입력 받은 값을 2의 보수 취한 값을 출력하는 프로그램이다. 이제 ollydbg로 저 main함수 부분 코드를 봐보도록 하겠다.

Disassembly	Comment
PUSH EBP	
MOV EBP, ESP	
PUSH ECX	
PUSH 0040B9A0	ASCII "Code Virtualized Test!! "
MOV DWORD PTR SS:[EBP-4], 0	
CALL 004010E4	
PUSH 0040B9BC	ASCII "http://ezbeat.tistory.com "
CALL 004010E4	
LEA EAX, DWORD PTR SS:[EBP-4]	
PUSH EAX	
PUSH 0040B9D8	ASCII "%d"
CALL 004010C7	
MOV EAX, DWORD PTR SS:[EBP-4]	
NOT EAX	
MOV DWORD PTR SS:[EBP-4], EAX	
INC EAX	
PUSH EAX	
PUSH 0040B9DC	ASCII "value : %d "
CALL 004010E4	
ADD ESP, 18	
XOR EAX, EAX	
MOV ESP, EBP	
POP EBP	
RETN	

[그림 13] main함수 어셈 코드

회색으로 된 부분을 제외한 나머지 부분은 함수 프로로그와 에필로그이므로 제외하고 그 가운데 부분을 가상화 시킬 것이다. 이 때 [그림 11]을 참조하여 만들 것이다.

아래는 변경된 결과이다. (100% 완성 된 건 아님)

PUSH ECX	
<u>push ecx</u>	<u>0a 80 02000000</u>
PUSH 0040B9A0	
<u>push 0x0040b9a0</u>	<u>0a 98 a0b94000</u>
MOV DWORD PTR SS:[EBP-4], 0	
<u>mov tmp,ebp</u>	<u>01 80 0a000000 06000000</u>
<u>sub tmp,4</u>	<u>06 83 0a000000 04000000</u>
<u>mov [tmp],0</u>	<u>01 8b 0a000000 00000000</u>
CALL 004010E4	
<u>call 0x004010e4</u>	<u>0e 98 e4104000</u>
PUSH 0040B9BC	
<u>push 0x0040b9bc</u>	<u>0a 98 bcb94000</u>
CALL 004010E4	
<u>call 0x004010e4</u>	<u>0e 98 e4104000</u>

```

LEA EAX,DWORD PTR SS:[EBP-4]
mov tmp,ebp          01 80 0a000000 06000000
sub tmp,4           06 83 0a000000 04000000
lea eax,[tmp]       02 81 00000000 0a000000

PUSH EAX
push eax            0a 80 00000000

PUSH 0040B9D8
push 0x0040b9d8    0a 98 d8b94000

CALL 004010C7
call 0x004010c7    0e 98 c7104000

MOV EAX,DWORD PTR SS:[EBP-4]
mov tmp,ebp          01 80 0a000000 06000000
sub tmp,4           06 83 0a000000 04000000
mov eax,[tmp]       01 81 00000000 0a000000

XOR EAX,FFFFFFFF
xor eax,0xffffffff 09 83 00000000 ffffffff

MOV DWORD PTR SS:[EBP-4],EAX
mov tmp,ebp          01 80 0a000000 06000000
sub tmp,4           06 83 0a000000 04000000
mov [tmp],eax       01 88 0a000000 00000000

INC EAX
inc eax             03 80 00000000

PUSH EAX
push eax            0a 80 00000000

PUSH 0040B9DC
push 0x0040b9dc    0a 98 dcb94000

CALL 004010E4
call 0x004010e4    0e 98 e4104000

ADD ESP,18
add esp,0x18       05 83 07000000 18000000

XOR EAX,EAX
xor eax,eax        09 80 00000000 00000000

end                 1c 00 cccccccc

```

빨간색(밑줄)으로 된 부분이 Intel Instruction에서 Virtual CPU에서 처리 가능한 Instruction으로 바꾼 결과이다.

Opcode 같은 경우는 각자 만들기 나름이지만 필자는 아래와 같이 OPCODE Handler를 만들어서 사용하고 있다. 아직은 28개의 명령어 밖에 만들지 못했지만 아래의 명령어 정도만 있어도 대부분 원하는 코드는 가상화 시킬 수 있다.

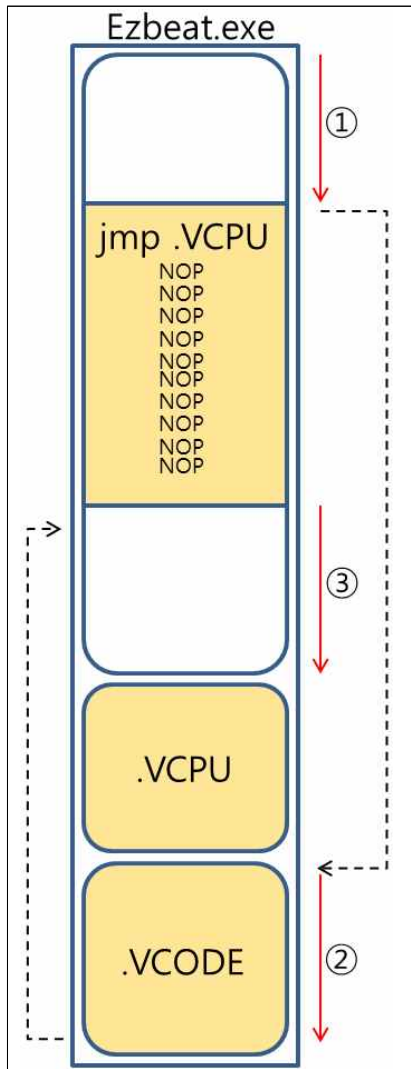
00.	Reserved		
01.	mov		
02.	lea		
03.	inc	0되면 ZF = 1	
04.	dec	0되면 ZF = 1	
05.	add		
06.	sub		
07.	and		
08.	or		
09.	xor		
0a.	push		
0b.	pop		
0c.	cmp		
0d.	test		
0e.	call		
0f.	jmp		
10.	JE	Left = Right	ZF = 1
11.	JNE	Left ≠ Right	ZF = 0
12.	JA(JNBE)	Left > Right	CF = 0 and ZF = 0
13.	JAE(JNB)	Left ≥ Right	CF = 0
14.	JB(JNAE)	Left < Right	CF = 1
15.	JBE(JNA)	Left ≤ Right	CF = 1 or ZF = 1
16.	JG(JNLE)	Left > Right	ZF = 0 and SF = 0
17.	JGE(JNL)	Left ≥ Right	SF = 0
18.	JL(JNGE)	Left < Right	SF = 1
19.	JLE(JNG)	Left ≤ Right	ZF = 1 or SF = 1
1a.	retn		
1b.	Reserved		
1c.	end		

바꾸어야 하는 명령어들은 Intel명령어를 대상으로 바꾸는 것이기 때문에 Virtual CPU에서 처리 가능한 어셈 코드 또한 Intel 명령어와 똑같이 작동하도록 만들어 냈다.

쪽 보면 1c번째에 end라는 명령어가 있는데 이 명령어는 Virtual CPU에서만 작동하는 명령어이다. 해당 명령어는 가상화된 코드 부분을 다 실행시켰을 때 다시 원래 코드로 돌아올 때 사용하는 명령어이다. 프로그램 전체를 가상화 시킬 경우 필요 없는 명령어이지만 우리의 목

적은 특정 위치만을 가상화 시키는 것이기 때문에 Intel CPU와 Virtual CPU는 서로 공존해야한다. 즉, 가상화 명령어를 만나면 Virtual CPU가 돌려주어야하고 다시 Intel 명령어를 만났을 경우에는 Intel CPU가 돌리게끔 해주어야한다.

간략한 그림으로 전체적인 흐름을 살펴보자.



[그림 14] 실행 흐름

오리지널 코드 부분은 위쪽에 .VCPU로 점프하는 구문을 빼고는 전부 NOP처리 하고 가상화된 코드 부분은 따로 섹션을 만들어서 빼놓았다. 왜 그런지 이유는 뒤 쪽에서 알아보도록 하겠다. 일단 [그림 14]를 보면 ①,③번 은 Intel CPU가 처리하는 부분이고 ②번은 Virtual CPU가 처리하는 부분이다. 이와 같이 ②번이 끝나면 다시 ③번 위치로 와줘야 되는데 이 때 사용되는 것이 end 명령어이다.

이제 명령어는 다 바꾸었으므로 코드 가상화의 핵심인 Virtual CPU를 만들어보기로 하자.

2.4 Virtual CPU

위쪽에서 [그림 9]를 보면 Virtual CPU가 구성되어 있는 섹션은 .VCPU섹션이다. 내부를 보면 Control Code, Relocation Value, PC Mapping Table Address, PC Mapping Size, OPCODE Handlers Table, OPCODE Handlers, PC Mapping Table 구성으로 이루어져 있다. 하지만 실질적으로 코딩을 해야 할 부분은 단 두 부분이다.

바로 Virtual CPU에서 코드를 처리하는데 있어서 필요한 스택 구성, Virtual CPU에서 사용할 데이터들 구성, .VCODE섹션에서 Opcode 하나씩 꺼내서 해당 OPCODE Handler 호출을 담당하는 Control Code와 해당 명령어를 처리하는 루틴인 OPCODE Handler들 만 코딩을 해주면 된다. 나머지 부분은 그냥 값 코드 영역이 아닌 단순 데이터 영역이므로 값만 설정해 놓고 가져오기만 하면 되는 부분이기 때문이다.

일단 여기서는 구성만 하고 실질적으로 프로그램에 넣어서 적용시키는 방법은 뒤쪽에서 보도록 하겠다.

2.4.1 Control Code

```
PUSHFD;
PUSH EBP;
SUB ESP,0x300; //Reserved Stack
MOV EBP,ESP;
SUB ESP,0x100; //VM Data
MOV DWORD PTR [EBP-0x100],EAX; //REG_EAX
MOV DWORD PTR [EBP-0xFC],EBX; //REG_EBX
MOV DWORD PTR [EBP-0xF8],ECX; //REG_ECX
MOV DWORD PTR [EBP-0xF4],EDX; //REG_EDX
MOV DWORD PTR [EBP-0xF0],ESI; //REG_ESI
MOV DWORD PTR [EBP-0xEC],EDI; //REG_EDI
MOV EAX,DWORD PTR [EBP+0x300];
MOV DWORD PTR [EBP-0xE8],EAX; //REG_EBP
LEA EAX,DWORD PTR [EBP+0x30C];
MOV DWORD PTR [EBP-0xE4],EAX; //REG_ESP
MOV EAX,DWORD PTR [EBP+0x308];
MOV DWORD PTR [EBP-0xE0],EAX; //REG_EIP
MOV EAX,DWORD PTR [EBP+0x304];
MOV DWORD PTR [EBP-0xDC],EAX; //REG_EFLAGS
CALL L1:
L1:
POP EAX;
SUB EAX,0x69; //VCPU Section Base VA 값 조정
NOP;
NOP;
//VCPU Section Base VA 값 VM Data에 저장
MOV DWORD PTR [EBP-0xD0],EAX;
```

```

ADD EAX,0x0F0;           //Relocation Value 있는 주소 위치 설정
MOV EAX,DWORD PTR [EAX]; //Relocation Value 값 ( 재배치 값 )
MOV DWORD PTR [EBP-0xD4],EAX;
MOV EBX,DWORD PTR [EBP-0xE0];
ADD EBX,EAX;           //push 한 VI 코드 영역 VA 값에 재배치 값 더해줘서 조정
MOV DWORD PTR [EBP-0xE0],EBX;
MOV EAX,DWORD PTR [EBP-0xD0];
ADD EAX,0x0F4;         //PC Mapping Table Base VA 있는 주소 위치 설정
MOV EAX,DWORD PTR [EAX]; //PC Mapping Table Base VA 값
//PC Mapping Table Base VA 값에 재배치 값 더해줘서 조정
ADD EAX,DWORD PTR [EBP-0xD4];
MOV DWORD PTR [EBP-0xCC],EAX;
MOV EAX,DWORD PTR [EBP-0xD0];
ADD EAX,0x0F8;         //Mapping Table Size 위치
MOV EAX,DWORD PTR [EAX]; //Mapping Table Size 값
MOV DWORD PTR [EBP-0xC8],EAX;

L3:
MOV EBX,DWORD PTR [EBP-0xD0];
ADD EBX,0x0FC;         //OPCODE Table VA 값
CALL L2; //VCODE Section으로부터 REG_EIP 읽어서 OPCODE 1Byte 가져옴
//가져온 OPCODE에 맞는 Handler VA 값 가져옴
MOV ECX,DWORD PTR [EBX+EAX*0x4];
//Handler VA 값에 재배치 값 더해줘서 조정
ADD ECX,DWORD PTR [EBP-0xD4];
MOV EAX,EBP;
SUB EAX,0x100;
PUSH EAX;              //VM Data 주소 push
CALL ECX;              //해당 OPCODE Handler 호출
ADD ESP,0x4;
JMP L3;

L2:
XOR EAX,EAX;
MOV ECX,DWORD PTR [EBP-0xE0];
MOV AL,BYTE PTR [ECX];
RETN;

```

[그림 15] .VCPU 섹션에서 Control Code 부분

전체 코드는 위와 같이 된다. 주석을 자세하게 달아놨기 때문에 어셈 코드 한줄 한줄 읽으면
서 스택 구성이 어떻게 되는지 직접 그려보면서 살펴보면 충분히 이해할 수 있을 것이다.

여기서 잠깐 실제 코드를 돌리다가 Virtual CPU로 넘어오는 과정만 살펴보자. 이 과정을 모
르면 위 코드를 분석하는 도중 막히는 부분이 있을 것이다.

```

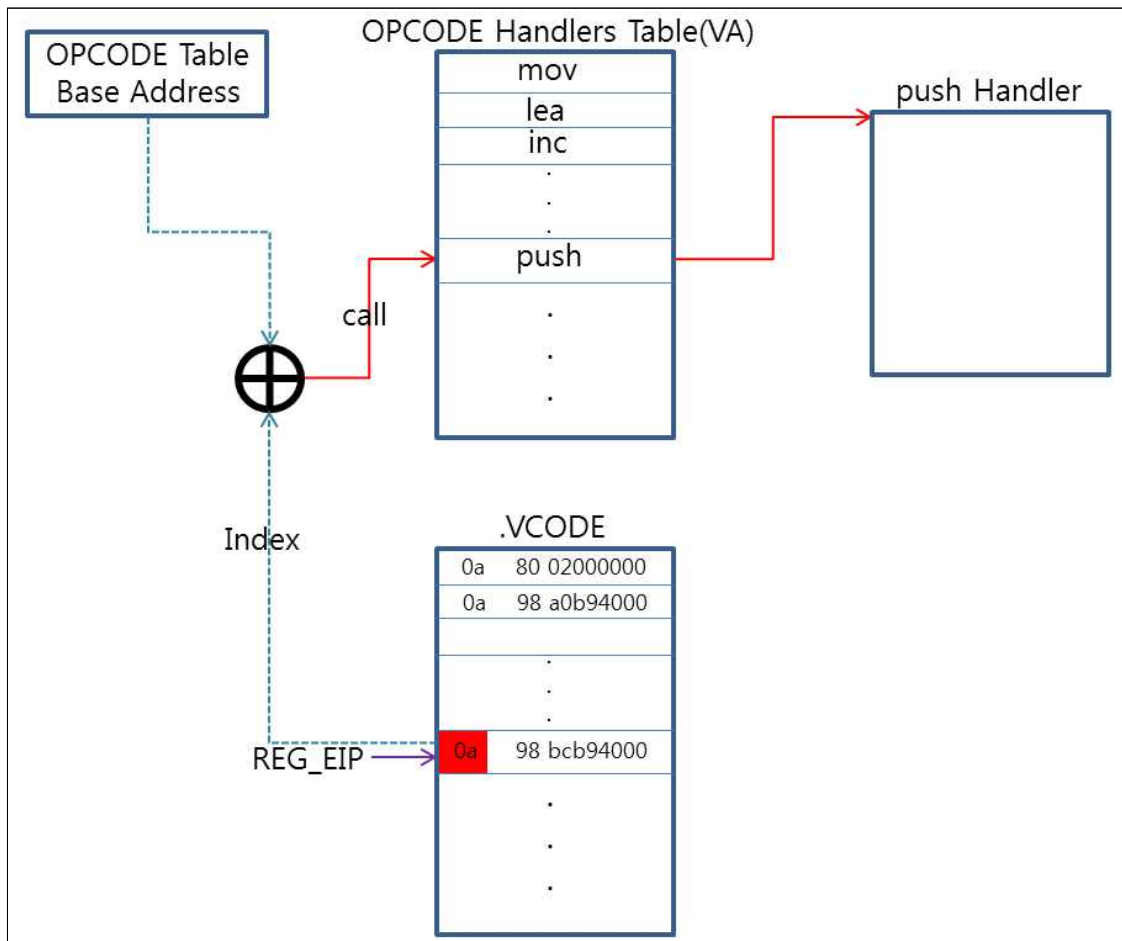
push REG_EIP      ; REG_EIP에 설정해야 할 값 ( 처음엔 .VCODE Base Address )
jmp .VCPU         ; VCPU로 점프

```

그리고 코드 중간 중간 보면 0x69, 0xf0, 0xf4, 0xf8, 0xfc 이런 값들이 있는데 이 값은

.VCPU섹션 Base Address에서 해당 Relocation Value, PC Mapping Table Address 등등 이러한 데이터 값들까지 떨어진 offset 값이다. .VCPU 같은 경우 위와 같이 코드를 짜놓고 프로그램에 아예 박아버리기 때문에 offset이 달라질 일은 없다. 그렇기 때문에 그냥 하드코딩을 해놓은 것이다. 만약 Virtual CPU를 수정해야할 일이 생기면 offset 값도 달라지기 때문에 그게 맞게 다시 해당 값들을 설정해 주어야 한다.

그리고 L3 아래로 있는 코드들이 무한루프를 돌면서 end명령을 만날 때 까지 VCODE섹션에서 OPCODE 부분을 읽어와 해당 명령어 핸들러를 호출해주는 부분이다. C코드도 아닌 어셈블리로 이루어져 있기 때문에 직관적으로 잘 이해가 안갈 수도 있으니 그림으로 나타내보겠다.



[그림 16] OPCODE읽고 Handler호출하는 과정

파워포인트로 열심히 그려보았지만 잘 이해가 가는 그림인지는 모르겠다. 간략히 위 그림과 코드를 조합해 가면서 설명을 해보도록 하겠다.

```
MOV EBX,DWORD PTR [EBP-0xD0];
ADD EBX,0x0FC;
```

이 부분이 OPCODE Table Base Address를 가져오는 부분이다.

EBX 레지스터에 해당 값이 저장되어 있다.

CALL L2;

내부를 보면 [EBP-0xE0] 가 있는데 이건 REG_EIP를 나타내고 있다.

다음에 OPCODE Handler내부 코드를 봐볼 것인데 코드를 보면 읽어들인 값 크기에 따라 REG_EIP 값을 증가시켜주는 부분을 확인할 수 있을 것이다.

어쨌든 CALL L2; 는 현재 REG_EIP가 가리키고 있는 명령어의 OPCODE를 읽어오는 함수이다. 그렇게 읽어들인 값을 Index 값으로 사용해 OPCODE Table Base Address 값이 저장되어 있는 EBX 값에 더해져서 OPCODE Handlers Table에 있는 주소를 호출하는 것이다.

2.4.2 OPCODE Handler

OPCODE Handler는 각 명령어를 처리하는 모듈들이다. 현재 이 문서에서 만들어진 Virtual CPU 같은 경우 28개의 명령어를 처리할 수 있다. 이에 대한 명령어나 Index값은 18페이지에 나와 있다. 이제 각 핸들러 코드를 봐볼 건데 다 볼 필요 없이 중요한 핸들러 몇 개만 보면 나머지는 조금씩 응용만 하면 바로바로 만들 수 있다.

실제로 핸들러를 보기 전에 핸들러들이 사용하는 구조체들을 먼저 봐보자.

```
enum {REG,REL_REG,REL_IMM,IMM};

/*
index  Mapping
0      eax
1      ebx
2      ecx
3      edx
4      esi
5      edi
6      ebp
7      esp
8      eip
9      eflags
A      tmp
*/

typedef struct MappingTable{
    int oriAddress;
    int chgAddress;
}MappingTable;

typedef struct VCPRegisterBlock {
    int registers[11];
    int RelocationValue;
```

```

    int VMBaseAddress;
    MappingTable* PCMappingAddress;
    int MappingTableSize;
}VCPRegisterBlock;

typedef struct OperandInfo{
    unsigned char Operand2Info:2;
    unsigned char Operand2InfoRelocation:1;
    unsigned char Operand1Info:2;
    unsigned char Operand1InfoRelocation:1;
    unsigned char Ref_Size:2;
}OperandInfo;

typedef struct BranchInfo{
    int CF:1;
    int Reserved1:1;
    int PF:1;
    int Reserved2:1;
    int AF:1;
    int Reserved3:1;
    int ZF:1;
    int SF:1;
    int Reserved4:24;
}BranchInfo;

```

[그림 17] Handler에서 사용하는 구조체들

가장 위에 있는 구조체는 MappingTable 구조체이다. int형 변수 2개로 이루어져 있으며 해당 구조체는 call이나 jmp문 같은 branch명령어에서 사용하는 구조체이다.

VCPRegisterBlock은 VM data 위에서부터 차례로 적어놓은 구조체이다.

해당 구조체는 모든 핸들러에서 사용하므로 핸들러 호출 시 매개변수 값으로 넘겨주어야한다. Virtual CPU의 Control Code 내부 코드 중 일부이다.

```

MOV EAX,EBP;
SUB EAX,0x100;
PUSH EAX;      //VM Data 주소 push
CALL ECX;     //Call Handler

```

EBP에서 0x100만큼 떨어진 위치는 VM Data가 존재하는 위치이며 해당 값을 Call전에 push 하고 있는 것을 확인 할 수 있다.

그 다음 구조체인 OperandInfo 구조체 또한 모든 핸들러에서 사용하는 구조체이다.

해당 구조체는 [그림 11]에서 확인 할 수 있듯이 참조 사이즈, Operand 정보들을 읽어와 그 게 맞는 루틴으로 점프시켜주는 역할을 한다.

마지막 구조체인 BranchInfo는 조건 점프문 핸들러에서 필요한 구조체이다.

inc, dec, cmp, test 같은 명령어에서 계산 결과에 따른 플래그 값 설정을 REG_EFLAGE에 미리 셋팅을 해주어야한다.

첫 번째로 볼 핸들러는 mov 핸들러이다.

```
//인자 2개
void vcp_mov(VCPRegisterBlock* pBlock) //1
{
    int eip,op_data1,op_data2;
    OperandInfo operandInfo;

    //registers[8] = REG_EIP
    eip = pBlock->registers[8];
    eip++;

    //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->registers[8] = eip;

    //두 번째 오퍼랜드의 정보를 오퍼랜드 정보에 맞게 가져옴
    switch(operandInfo.Operand2Info)
    {
        //두번째 인자가 REG
    case REG:
        op_data2 = pBlock->registers[op_data2];
        break;
        //두번째 인자가 [REG]
    case REL_REG:
        op_data2 = pBlock->registers[op_data2];
        op_data2 = *(int*)op_data2;
        break;
        //두번째 인자가 [IMM]
    case REL_IMM:
        if(operandInfo.Operand2InfoRelocation)
            op_data2 += pBlock->RelocationValue;
        op_data2 += *(int*)op_data2;
        break;
    }
```

```

        //두번째 인자가 IMM
    case IMM:
        break;

    default:
        break;
}

//첫 번째 오퍼랜드의 정보를 오퍼랜드 정보에 맞게 가져옴
switch(operandInfo.Operand1Info)
{
    //첫번째 인자가 REG
    case REG:
        pBlock->registers[op_data1] = op_data2;
        break;
    //첫번째 인자가 [REG]
    case REL_REG:
        op_data1 = pBlock->registers[op_data1];
        *(int*)op_data1 = op_data2;
        break;
    //첫번째 인자가 [IMM]
    case REL_IMM:
        if(operandInfo.Operand1InfoRelocation)
            op_data1 += pBlock->RelocationValue;
        *(int*)op_data1 = op_data2;
        break;
    //첫번째 인자가 IMM
    case IMM:
        break;

    default:
        break;
}
}

```

[그림 18] Virtual CPU에서 처리 가능한 mov Instruction Handler

주석을 달아놓긴 했지만 짧게 설명을 해보면

먼저 오퍼랜드 정보와 오퍼랜드들 값을 지역변수로 다 가져온다.

그리고 가져온 크기만큼 다음 명령어 위치를 가리키고 있을 수 있도록 REG_EIP 값을 증가시켜준다. 이제 가져온 오퍼랜드와 오퍼랜드 정보를 이용해 해당 명령어에 맞는 행동을 수행한다. 처음엔 두 번째 오퍼랜드가 REG, [REG], [IMM], IMM 인지를 판단 후 제대로 된 값을 가져오고 그 다음엔 첫 번째 오퍼랜드가 REG, [REG], [IMM], IMM 인지를 판단 후 두 번째 오퍼랜드에서 가져온 값을 적용 시키는 것이다.

mov 명령어는 두 번째 오퍼랜드 값을 첫 번째 오퍼랜드로 복사를 시키는 것이기 때문에 극

맞게 코딩을 해주면 된다. 이게 만들어진 코드의 전부이다.

그렇기 때문에 lea같은 명령어를 만들더라도 위 코드에서 첫 번째 오퍼랜드 가져오는 부분만 수정해 주면 된다. 나머지 코드는 다 똑같다. 이러한 방식으로 add, sub, inc ,dec 등등 다 쉽게 만들 수 있다.

이제 mov 명령어보다 좀 까다로운 call 명령어를 구현해보도록 하겠다.

아래는 call 명령어 핸들러이다.

```
//인자 1개
void vcp_call(VCPRegisterBlock* pBlock) //e
{
    int eip,op_data1;
    OperandInfo operandInfo;
    bool find = false;

    //registers[8] = REG_EIP
    eip = pBlock->registers[8];
    eip++;

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

    //push retn address
    pBlock->registers[7] -= 4;
    *(int*)(pBlock->registers[7]) = eip;

    //첫 번째 오퍼랜드의 정보를 오퍼랜드 정보에 맞게 가져옴
    switch(operandInfo.Operand1Info)
    {
        //첫번째 인자가 REG
    case REG:
        op_data1 = pBlock->registers[op_data1];
        break;
        //첫번째 인자가 [REG]
    case REL_REG:
        op_data1 = pBlock->registers[op_data1];
        op_data1 = *(int*)op_data1;
        break;
        //첫번째 인자가 [IMM]
    case REL_IMM:
        if(operandInfo.Operand1InfoRelocation)
            op_data1 += pBlock->RelocationValue;
        op_data1 = *(int*)op_data1;
        break;
        //첫번째 인자가 IMM
    case IMM:
        break;
    }
```

```

default:
    break;
}

//call할 주소가 mapping table에 있는지 확인
for(int i = 0; i < pBlock->MappingTableSize; i++)
{
    if(pBlock->PCMappingAddress[i].oriAddress == op_data1)
    {
        find = true;
        //REG_EIP에 있는 값 바꿈
        pBlock->registers[8] =
            pBlock->PCMappingAddress[i].chgAddress
            + pBlock->RelocationValue;
        break;
    }
}

//call할 주소가 외부에 있는 경우
if(find == false)
{
    if(operandInfo.Operand1InfoRelocation)
        op_data1 += pBlock->RelocationValue;

    //호출 주소 임시 저장
    pBlock->registers[0xa] = op_data1;

    //naked함수가 아니기 때문에 처음에 함수 프로로그로 인한 ebp 값 변경이 생겼음
    __asm{
        MOV EAX,DWORD PTR [EBP+0xC];
        MOV EBX,DWORD PTR [EBP+0x10];
        MOV ECX,DWORD PTR [EBP+0x14];
        MOV EDX,DWORD PTR [EBP+0x18];
        MOV ESI,DWORD PTR [EBP+0x1C];
        MOV EDI,DWORD PTR [EBP+0x20];
        MOV ESP,DWORD PTR [EBP+0x28];

        //호출될 함수 주소 push후 ebp값 변경
        //먼저 ebp변경하면 스택 베이스가 달라지므로 offset값 접근 불가
        PUSH DWORD PTR [EBP+0x34];
        MOV EBP,DWORD PTR [EBP+0x24];

        RETN;          //스택에 push된 함수 주소로 점프
    }
}
}

```

[그림 19] Virtual CPU에서 처리 가능한 call Instruction Handler

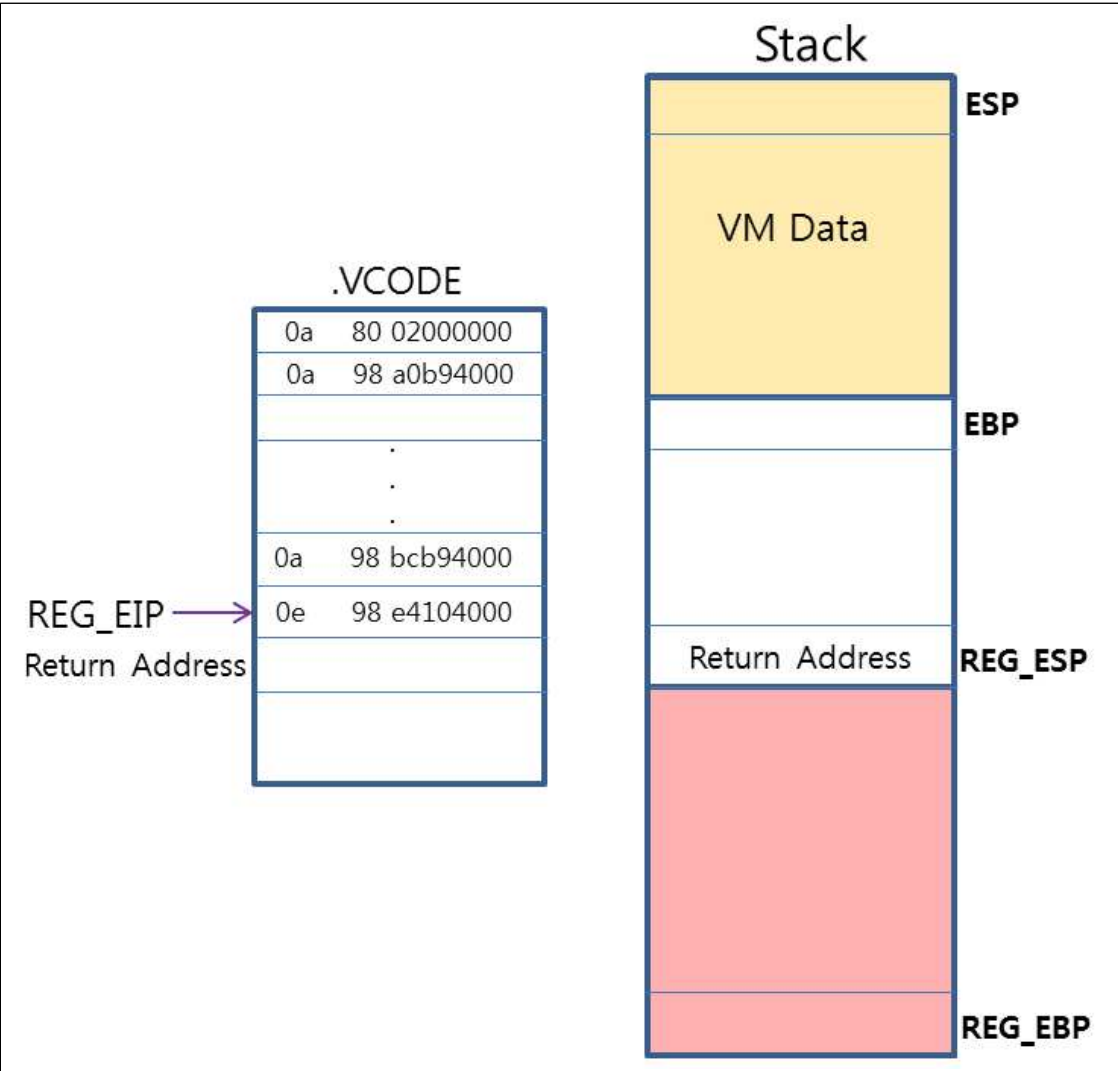
call 명령어 핸들러에서 대해서는 설명할 것이 좀 많다.
코드 부분 부분 자세히 살펴보자.

```

//push retn address
pBlock->registers[7] -= 4;
*(int*)(pBlock->registers[7]) = eip;

```

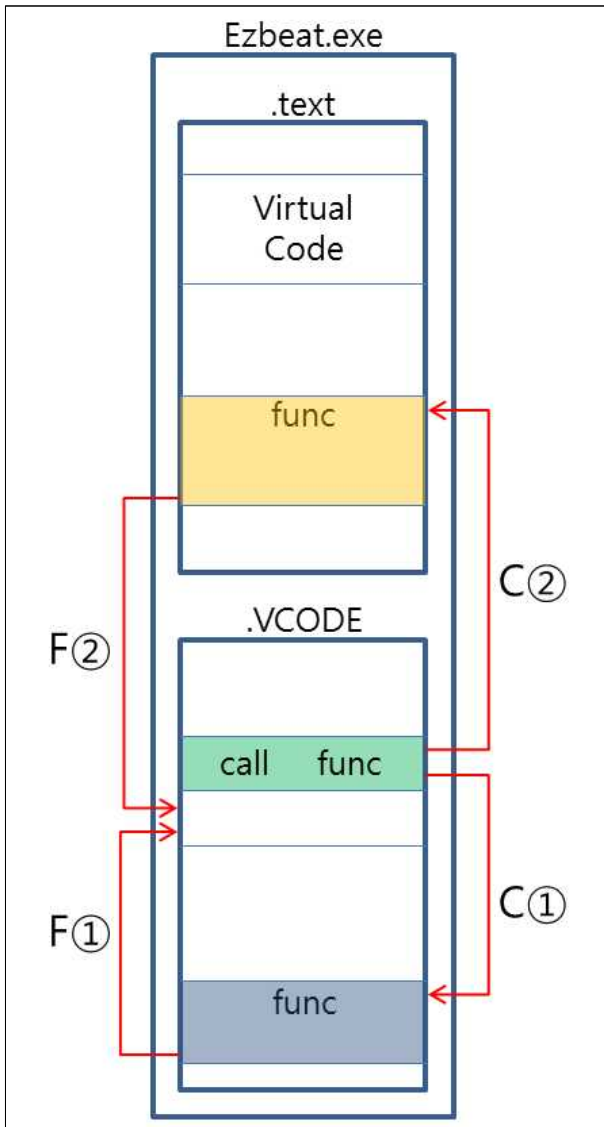
먼저 이 부분이다. registers[7]은 REG_ESP를 나타내고 있다. 그리고 REG_ESP레지스터가 가리키는 곳에 REG_EIP를 넣는다. 여기 부분은 보통 call명령어를 수행하게 되면 리턴 될 주소(call 명령어 다음 명령어)가 스택에 자동으로 push되고 함수 프로로그가 시작되는데 이 때 리턴 될 주소를 넣어주는 부분이다. 그림으로 보자.



[그림 20] call 명령어의 retn 주소 넣는 과정

OPCODE - 0e 는 call을 나타낸다. 그렇기 때문에 그 다음 명령어가 있는 주소가 리턴 주소가 되며 해당 값을 오른쪽 그림처럼 스택에 넣어준다.

그 다음에 오퍼랜드 값에서 호출해야할 함수 주소를 얻어온다. 이제 여기서 부터가 또 중요한데 해당 함수 호출 주소가 .VCODE 섹션에 있는 주소인지 아니면 외부에 있는 주소인지에 따라 루틴이 달라지게 된다. 이것도 그림으로 봐보자.



[그림 21] VCODE External call & Internal call

먼저 C①은 VCODE내부에서 VCODE 내부의 함수를 호출한 경우이다. 이 경우 코드를 처리하는 CPU는 계속해서 Virtual CPU이므로 함수가 끝나고 리턴주소로 다시 돌아와도(F①) 따로 무언가 처리를 하지 않아도 된다. 이번엔 C②이다. 이 경우는 VCODE 외부의 함수를 호출하는 경우인데 이 경우 외부 함수 호출을 하게 되면 코드를 실행하는 주체가 Intel CPU가 되어 있다. 이러한 상황에서 함수가 끝나고 다시 스택에 존재하는 리턴 주소대로 VCODE쪽으로 돌아오게 되면(F②) 어떻게 될까?? VCODE에 있는 명령어들은 Virtual CPU만 처리할 수 있다. 하지만 현재 상황에선 Intel CPU가 처리를 하고 있는 상황이므로 프로그램이 정상적으로 명령어를 처리하지 못하고 뺏나게 될 것이다. 이 경우 OPCODE - 0e 인 call을 했을 때 해당 주소가 외부에 있을 경우 함수 호출이 끝나고 다시 돌아왔을 때 명령어를 정상적으로 처리하기 위해 제어를 다시 Virtual CPU쪽으로 바꿔주어야 할 것이다. 다시 바꿔주는 명령어는 [그림 15] 아래쪽에 나와있다. 이 내용을 토대로 완성하지 못한 VCODE 부분을 다시 완성시켜 보도록 하겠다.

이번엔 변경 전 Intel 명령어는 빼고 적겠다.

1000	push ecx	0a 80 02000000
1006	push 0x0040b9a0	0a 98 a0b94000
1012	mov tmp,ebp	01 80 0a000000 06000000
1022	sub tmp,4	06 83 0a000000 04000000
1032	mov [tmp],0	01 8b 0a000000 00000000
1042	call 0x004010e4	0e 98 e4104000
1048	push 0x1058	68 xxxxxxxx
1053	jmp .VCPU	e9 xxxxxxxx
1058	push 0x0040b9bc	0a 98 bcb94000
1064	call 0x004010e4	0e 98 e4104000
1070	push 0x1080	68 xxxxxxxx
1075	jmp .VCPU	e9 xxxxxxxx
1080	mov tmp,ebp	01 80 0a000000 06000000
1090	sub tmp,4	06 83 0a000000 04000000
1100	lea eax,[tmp]	02 81 00000000 0a000000
1110	push eax	0a 80 00000000
1116	push 0x0040b9d8	0a 98 d8b94000
1122	call 0x004010c7	0e 98 c7104000
1128	push 0x1138	68 xxxxxxxx
1133	jmp .VCPU	e9 xxxxxxxx
1138	mov tmp,ebp	01 80 0a000000 06000000
1148	sub tmp,4	06 83 0a000000 04000000
1158	mov eax,[tmp]	01 81 00000000 0a000000
1168	xor eax,0xffffffff	09 83 00000000 ffffffff
1178	mov tmp,ebp	01 80 0a000000 06000000
1188	sub tmp,4	06 83 0a000000 04000000
1198	mov [tmp],eax	01 88 0a000000 00000000
1208	inc eax	03 80 00000000
1214	push eax	0a 80 00000000
1220	push 0x0040b9dc	0a 98 dcb94000
1226	call 0x004010e4	0e 98 e4104000
1232	push 0x1242	68 xxxxxxxx
1237	jmp .VCPU	e9 xxxxxxxx
1242	add esp,0x18	05 83 07000000 18000000
1252	xor eax,eax	09 80 00000000 00000000
1262	end [복귀 주소]	1c 00 xxxxxxxx

[그림 22] 완성된 가상화된 명령어 (VA 값만 채워 넣으면 됨)

현재 xxxxxxxx 로 된 부분은 실제 메모리에서 VA 값을 보고 적어야하기 때문에 보류해둔 것이다. 옆 1000부터 쓴 값은 임의의 가상 주소이다. 그리고 주소에 표시된 부분 혹은 파란색으로 쓰여진 부분은 Intel CPU에서 처리되는 명령어 부분이고 나머지 부분은 Virtual CPU에서 처리되는 부분이다. 현재 필자가 쓴 코드에서 call되는 함수들은 외부에 있는 printf와 scanf 뿐이기 때문에 전부 call 명령어 이후에 위와 같은 처리를 해준 것이다. 다시 말하는 거지만 만약 VCODE 섹션에 있는 함수를 호출할 경우에는 call 명령어 이후 아무런 처리를 해주지 않아도 된다.

이제 이러한 정보를 알았으니 그 아래쪽 call 명령어 핸들러를 살펴보자.

```

//call할 주소가 mapping table에 있는지 확인
for(int i = 0; i < pBlock->MappingTableSize; i++)
{
    if(pBlock->PCMappingAddress[i].oriAddress == op_data1)
    {
        find = true;
        //REG_EIP에 있는 값 바꿈
        pBlock->registers[8] =
            pBlock->PCMappingAddress[i].chgAddress
            + pBlock->RelocationValue;
        break;
    }
}

```

매핑 테이블 크기만큼 루프문을 돌면서 호출하려는 주소가 VCODE 내부 함수인지를 체크하고 있다. 만약에 있다면 바뀐 주소를 REG_EIP에 넣고 call 명령어 핸들러는 끝나게 된다. 그러면 다시 Virtual CPU는 해당 REG_EIP에서 OPCODE를 꺼내 계속 실행할 것이다.

이제 그 아래 코드는 호출할 주소가 외부에 있을 경우이다.

```

//call할 주소가 외부에 있는 경우
if(find == false)
{
    if(operandInfo.Operand1InfoRelocation)
        op_data1 += pBlock->RelocationValue;

    //호출 주소 임시 저장
    pBlock->registers[0xa] = op_data1;

    //naked함수가 아니기 때문에 처음에 함수 프로로그로 인한 ebp 값 변경이 생겼음
    __asm{
        MOV EAX,DWORD PTR [EBP+0xC];
        MOV EBX,DWORD PTR [EBP+0x10];
        MOV ECX,DWORD PTR [EBP+0x14];
        MOV EDX,DWORD PTR [EBP+0x18];
        MOV ESI,DWORD PTR [EBP+0x1C];
        MOV EDI,DWORD PTR [EBP+0x20];
        MOV ESP,DWORD PTR [EBP+0x28];

        //호출될 함수 주소 push후 ebp값 변경
        //먼저 ebp변경하면 스택 베이스가 달라지므로 offset값 접근 불가
        PUSH DWORD PTR [EBP+0x34];
        MOV EBP,DWORD PTR [EBP+0x24];

        RETN;          //스택에 push된 함수 주소로 점프
    }
}

```

먼저 전체적인 설명을 해보면 VCODE 외부에 있는 함수가 호출되기 때문에 명령어를 실행하는 주체가 다시 Intel CPU로 바뀌게 된다. 이 때 VCODE에서 사용하고 있던 레지스터들을 복구해주지 않으면 정상적으로 Intel CPU가 실행할 준비를 하지 못하고 코드를 실행하면서 뺏이 나게 될 것이다.

호출할 함수 주소를 임시 레지스터인 REG_TMP에 넣어두고 나머지 레지스터들을 다 복구

시킨다. 이 때 ebp는 가장 나중에 복구 시킨다. 왜냐하면 다른 VM Data 영역에 접근 시 ebp를 기준으로 접근을 하게 되는데 ebp가 바뀌어 버리면 offset 값이 바뀌게 되면서 정상 작동을 하지 않기 때문이다. 함수 호출하는 방법은 스택에 주소를 push해 두고 retn을 사용 함으로서 해당 주소로 가게 되는 방법을 사용했다.

이제 외부에서 호출된 함수가 끝나고 다시 VCODE 영역으로 왔을 땐 바로 push와 jmp명령어를 만나게 되면서 제어를 다시 Virtual CPU로 바꿔줄 것이다.

마지막으로 end 명령어를 알아보려 했지만 end 명령어는 call명령어의 마지막 부분만 따서 잘 만들면 되므로 생략하도록 하겠다. 그리고 다른 명령어들은 위 두 명령어 코드를 잘 활용 하면 금방 금방 만들 수 있을 것이다.

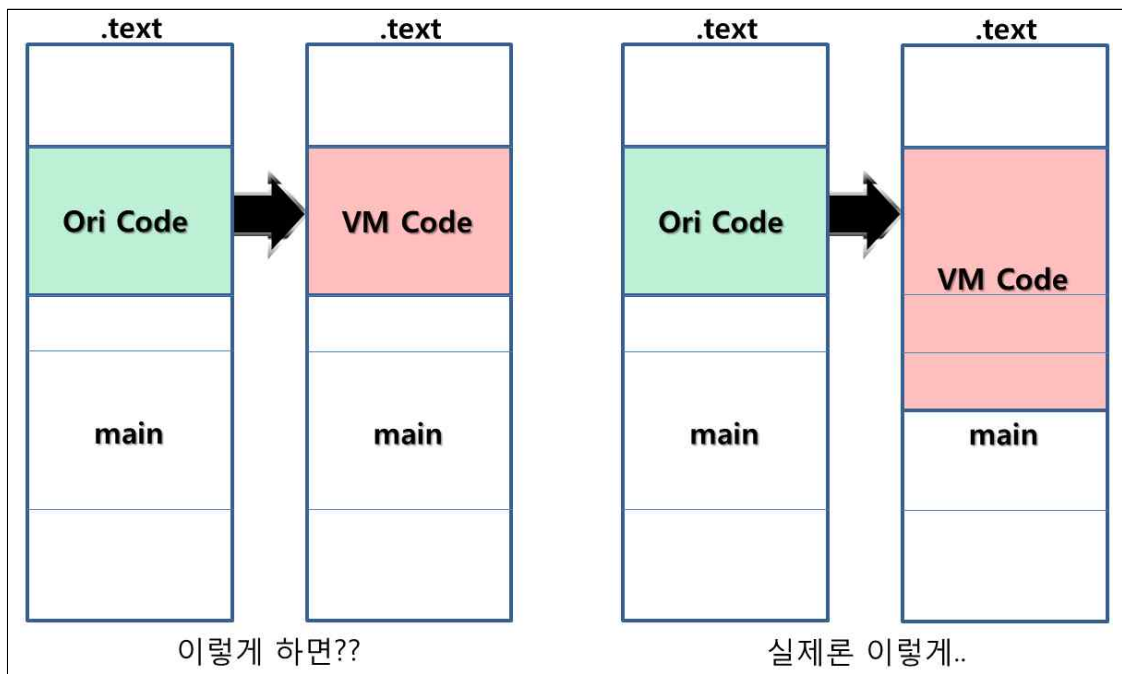
이제 가상화 시킬 타겟 코드 변경도 끝났고 Virtual CPU도 만들었다. 마지막 과제인 실제 프로그램에 적용하는 방법을 알아보자.

2.5 Implementation

이제 마지막 단계인 구현 단계까지 왔다. 마무리를 잘하여 성공적인 결과를 볼 수 있도록 기대감을 안고 계속 달려보자.

구현 단계는 크게 복잡한 것이 없다. 위에서 만든 .VCPU와 .VCODE를 넣을 섹션을 만들고 각 섹션들에 적용만 시켜주면 된다.

여기서 잠깐 각 섹션을 만드는 이유에 대해서 살펴보고 가자. .VCPU 섹션 같은 경우 새롭게 코드를 만들어서 프로그램에 넣어야 되니까 만든다고 쳐도 .VCODE 같은 경우는 오리지널 코드를 가상화 시켜서 바꾸는 것 뿐 이니까 그냥 원래 있던 코드 부분에 넣어도 되지 않을까? 라는 생각이 들 수도 있다. 아래 그림을 봐보자.



[그림 23] 코드 변경 시 코드 크기 증가

왼쪽 그림처럼만 구현이 되면 좋겠지만 실질적으로 코드를 변경시켜보면 오른쪽과 같은 결과

를 볼 수 있다. 왜 그럴까?

Intel CPU 명령어 같은 경우 아무리 길어야 7~8Byte 수준이고 평균적으로 3~5Byte의 크기를 가지고 있다. 하지만 우리가 변경할 명령어는 오퍼랜드가 하나면 6Byte의 크기를 지니고 오퍼랜드가 2개면 10Byte의 크기를 지닌다. 또한 Intel CPU가 처리하는 명령어 구조를 우리는 지니고 있지 않다.

Original Code	VM Code
mov [ebp-0x14],ebx	mov tmp,ebp sub tmp,0x14 mov [tmp],ebx

위 코드를 보더라도 오리지널 코드 한 줄이 변환 될 때 총 세 줄의 코드가 나오는 것을 확인할 수 있다. 실제 코드를 보면

mov [ebp-0x14],ebx 는 “895D EC” 로 3Byte만에 표현이 되지만

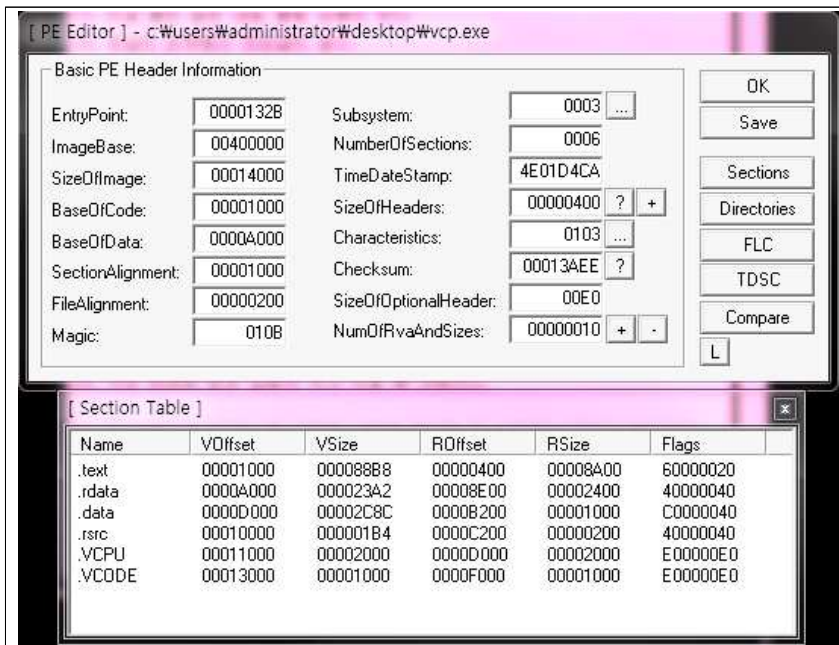
변환된 명령어는 각각 10Byte로 총 30Byte로 표현이 된다.
무려 10배나 크기가 증가하게 되는 것이다.

현재 이 문서에서 변환되기 전 오리지널 코드의 크기는 67Byte이고 [그림 22]에 변환된 코드를 보면 변환된 코드의 크기는 268Byte이다.

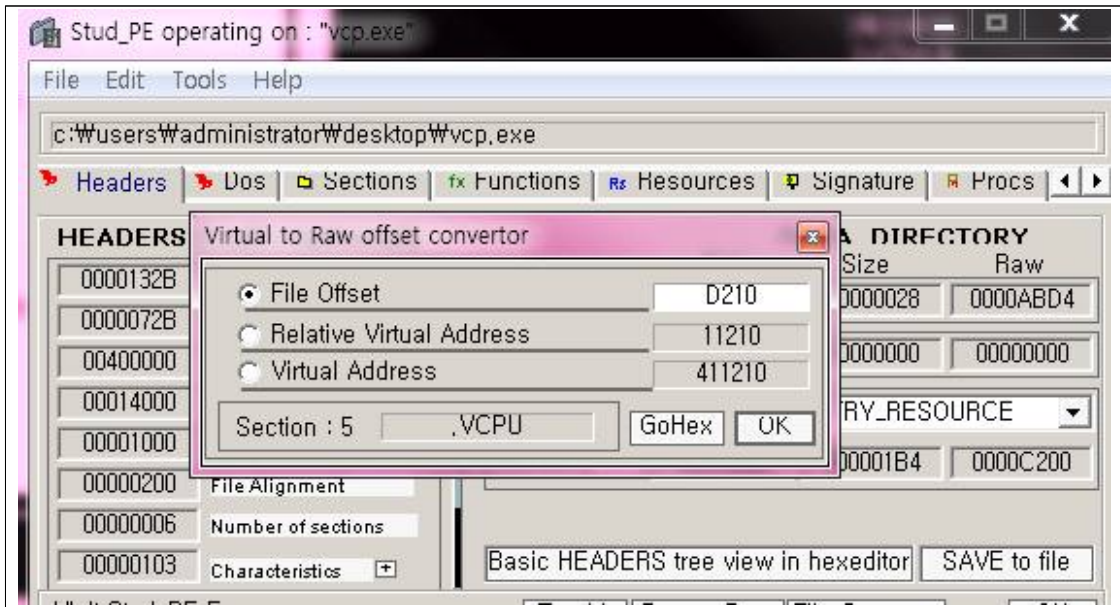
대략 4배가 증가한 것을 확인 할 수 있다. 이렇게 때문에 오리지널 코드가 있는 부분은 Virtual CPU로 제어를 넘겨주는 부분만 넣어주고 나머지는 전부 nop처리를 하던 뭘 하던 쓰레기 값을 넣어주고 실제 변환된 코드는 새로운 섹션을 만들어서 거기에 넣어준 것이다.

2.5.1 Create Section

수동적으로 Hex Editor를 사용해 일일이 값을 넣을 수도 있지만 유용한 툴이 있으므로 툴을 사용해서 섹션을 생성하도록 하겠다.



[그림 24] PE Editor를 통한 섹션 생성



[그림 28] Stud_PE 툴로 본 VA <=> RAW

File Offset 값이 VA 값으로 몇 인지 구할 때 필자는 위 툴을 사용해 구했다.

나머지 명령어들도 위와 같은 방식으로 그 아래 쪽 쪽 넣어 주면 된다.

이제 Virtual CPU에서 남은 부분은 PC Mapping Table인데 해당 값 또한 .VCPU내 빈 곳을 찾아서 오리지널 주소(4Byte), 바뀔 주소(4Byte) 순으로 PC Mapping Table Size 개수만큼 넣어주면 된다. 또한 PC Mapping Table 값을 넣은 File Offset 또한 VA 값을 구해 PC Mapping Table Address 위치에 적어주면 된다.

Virtual CPU 부분은 끝이 났다. 이번엔 .VCODE 부분을 채워볼텐데 이 부분은 [그림 22]에 있는 Hex 값만 뽑아내서 붙혀넣기 하면 된다. 여기서 중요할 점은 아직 채워넣지 않은 부분은 실제로 올디에서 값을 확인 후 VA 값을 채워 넣으면 된다. .VCODE 섹션의 시작 File Offset 값은 0xF000이고 해당 VA 값은 0x00413000이다. 이제 0x00413000을 시작 주소로 [그림 22]를 완성시켜 보았다.

0x00413000	push ecx	0a 80 02000000
0x00413006	push 0x0040b9a0	0a 98 a0b94000
0x0041300C	mov tmp,ebp	01 80 0a000000 06000000
0x00413016	sub tmp,4	06 83 0a000000 04000000
0x00413020	mov [tmp],0	01 8b 0a000000 00000000
0x0041302A	call 0x004010e4	0e 98 e4104000
0x00413030	push 0x0041303a	68 3a304100
0x00413035	jmp 0x00411000	e9 c6dfffff
0x0041303A	push 0x0040b9bc	0a 98 bcb94000
0x00413040	call 0x004010e4	0e 98 e4104000
0x00413046	push 0x00413050	68 50304100
0x0041304B	jmp 0x00411000	e9 b0dfffff
0x00413050	mov tmp,ebp	01 80 0a000000 06000000

0x0041305A	sub tmp,4	06 83 0a000000 04000000
0x00413064	lea eax,[tmp]	02 81 00000000 0a000000
0x0041306E	push eax	0a 80 00000000
0x00413074	push 0x0040b9d8	0a 98 d8b94000
0x0041307A	call 0x004010c7	0e 98 c7104000
0x00413080	push 0x0041308a	68 8a304100
0x00413085	jmp 0x00411000	e9 76dfffff
0x0041308A	mov tmp,ebp	01 80 0a000000 06000000
0x00413094	sub tmp,4	06 83 0a000000 04000000
0x0041309E	mov eax,[tmp]	01 81 00000000 0a000000
0x004130A8	xor eax,0xffffffff	09 83 00000000 ffffffff
0x004130B2	mov tmp,ebp	01 80 0a000000 06000000
0x004130BC	sub tmp,4	06 83 0a000000 04000000
0x004130C6	mov [tmp],eax	01 88 0a000000 00000000
0x004130D0	inc eax	03 80 00000000
0x004130D6	push eax	0a 80 00000000
0x004130DC	push 0x0040b9dc	0a 98 dcb94000
0x004130E2	call 0x004010e4	0e 98 e4104000
0x004130E8	push 0x004130f2	68 f2304100
0x004130ED	jmp 0x00411000	e9 0edfffff
0x004130F2	add esp,0x18	05 83 07000000 18000000
0x004130FC	xor eax,eax	09 80 00000000 00000000
0x00413106	end 0x00401046	1c 00 46104000

[그림 29] 완성된 가상화된 명령어

Intel CPU가 처리하는 jmp 명령어 뒤에 오는 주소는 0x00411000(.VCPU) 값으로 계속 같이만 오퍼랜드 값은 계속 변하게 되는데 jmp 명령어 같은 경우 오퍼랜드 값이 현재 주소로부터 점프할 주소까지의 offset 값이 들어가게 된다. 그렇기 때문에 계속 변한 것이다. (정확히는 jmp 명령어가 있는 줄의 코드 Byte수는 뺀다.)

이제 완성은 되었고 옆 Hex 값만 .VCODE 섹션에 복사 붙여넣기를 하면 된다.

모든 준비 과정은 끝났고 마지막으로 가상화를 당한 코드 시작 부분만 봐보겠다.

Address	Hex dump	Disassembly
00401000	\$ 55	PUSH EBP
00401001	. 8BEC	MOV EBP,ESP
00401003	. 68 00304100	PUSH 00413000
00401008	.- E9 F3FF0000	JMP 00411000
0040100D	90	NOP
0040100E	90	NOP
0040100F	90	NOP
00401010	90	NOP
00401011	90	NOP
00401012	90	NOP
00401013	90	NOP
00401014	90	NOP
00401015	90	NOP

[그림 30] 가상화 당한 코드 시작 부분

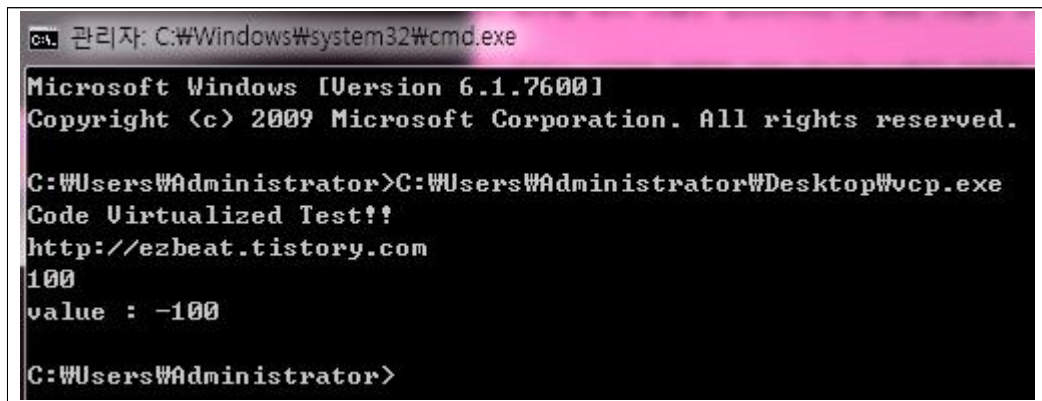
0x00401003 주소부터 가상화를 시켰고 처음 부분에는 Virtual CPU로 제어를 넘겨주어야 되

므로 .VCODE 섹션 주소를 push 하고 .VCPUS섹션으로 jmp를 한 것이다.
이제 정상적으로 실행되기를 기도하는 일만 남았다.

2.6 Execution

힘들게 구현해온 만큼 정상적으로 실행되기를 바라는 마음이지만 필자는 문서를 작성하기 전에 미리 정상 작동되는 것 까지 확인 후 문서 작성을 시작하였기 때문에 크게 긴장이 되진 않는다.

실행결과이다.



```
관리자: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>C:\Users\Administrator\Desktop\wcp.exe
Code Virtualized Test!!
http://ezbeat.tistory.com
100
value : -100

C:\Users\Administrator>
```

[그림 31] 코드 가상화 시킨 프로그램 실행 화면

정상적으로 잘 작동하는 것을 확인하였다.
허무하면서도 뿌듯하다.

처음부터 구현해온 방법을 차근차근 다시 정리해보자.

1. 명령어 포맷 지정
2. Virtual CPU - Control Code 만들기
3. Virtual CPU - OPCODE Handler 만들기
4. 코드 가상화 시키기
5. 섹션 생성
6. Virtual CPU 넣기
7. 가상화 된 코드 넣기
8. 가상화 되어진 부분 코드 변경
9. 프로그램 실행

대략 9가지만 적어봤는데 세부적으로는 더욱 많이 있었으며 각 순서는 편한 대로 하면 된다.

3. Conclusion

지금껏 위에서 구현해온 코드 가상화는 안티 리버싱 기법으로 보안 프로그래머에게 유용한 기술이 될 수 있으며 또한 악성코드 제작자에게 또한 유용한 기술이 될 수 있으므로 누구를 위한 기술이라고는 말 할 수 없고 리버서 방해하기 위한 기술이라고 생각하면 된다. 위 코드 가상화를 구현해오면서 초점은 “코드 가상화를 어떻게 할 것인가?”에 중점을 두고 작성되었다. 그렇기 때문에 위와 같이 코드 가상화를 그냥 적용시키면 효과를 100% 발휘 할 수 없다. 이 코드 가상화 기법을 사용해 진짜 안티 리버싱 기법을 적용하려면 좀 더 많은 생각을 해봐야한다. 필자가 이러한 방법에 대해서 몇 가지 생각해 보았다.

첫 번째는 Virtual CPU의 Control Code 부분을 난독화 시키는 것이다. 그리고 거기에 기술적이건 함수적이건 안티 리버싱 기술을 더 사용하는 것이다. 보통 좀 강력한 패커들을 보더라도 난독화에 각종 안티리버싱 기법, 함수들이 마구잡이로 사용되어 있는 것을 볼 수 있다.

두 번째는 첫 번째 방법을 적용 후 섹션들을 없애고 Virtual CPU나 가상화된 코드는 data 영역에 인코딩해서 넣어둔다. 그 후 프로그램 내부에서 VirtualAlloc 함수를 사용해 특정 메모리 부분을 할당하고 그 메모리 영역에 data 영역에 있는 Virtual CPU와 가상화된 코드를 디코딩 시킨 후 넣는 것이다. 그리고 그 루틴은 TLS Callback 루틴에 넣어두면 더욱 효과적이다.

세 번째는 명령어 포맷을 복잡하게 바꾸는 것이다. 다른 코드 가상화 기법을 보면 명령어 OPCODE 값을 구할 때 첫 번째 바이트와 두 번째 바이트를 XOR 한 값으로 사용하기도 하였다. 이처럼 명령어 체계를 복잡하게 구성해놔도 리버서 입장에서는 상당히 애를 먹을 것이다.

대략 세 가지 정도의 방법만 적어봤는데 생각을 조금만 더 해보면 재미있게 꼬아 놓을 수 있을 것이다.

이러한 코드 가상화 기법은 안티 리버싱을 위한 기술이지 프로그램을 빠르게 하기 위한 기술이 아님을 알아뒀으면 한다. CPU를 하나 더 만든다고 해서 “이거 빨라지는거 아니야?” 라고 생각하는 사람이 없길 바란다.

지금까지 긴 문서 읽어주신 분들께 너무 감사드리고 혹시 잘못된 부분이나 궁금하신 사항이 있으면 아래로 문의를 주시면 됩니다.

네이트 : sonicpj@nate.com

블로그 : <http://ezbeat.tistory.com>

4. Reference

- [1] Design and Implementation of VCP For Anti-Reverse Engineering - [이용일]
- [2] Anti-Debugging - A Developers View - [Tyler Shields]
- [3] Applied Binary Code Obfuscation - [Nicolaou George, Glafkos Charalambous]
- [4] <http://reversecode.com> - PE(Portable Executable) File Format