이민규

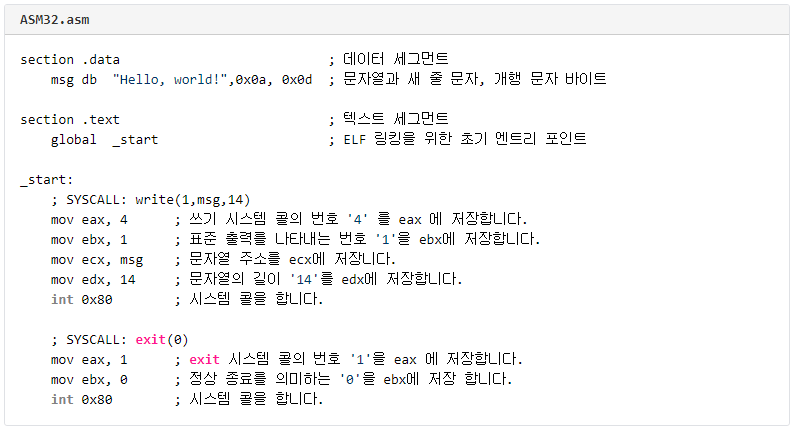
1. The basics technic of Shellcode
   1. Shellcode

Shellcode란, 명령 shell을 실행하여 공격자가 해당 시스템을 제어하기 때문이다. Machine code로 작성된 작은 크기의 프로그램이며 일반적으로 어셈블리어로 작성 후 기계어로 변환한다.

* 1. Assembly-Language

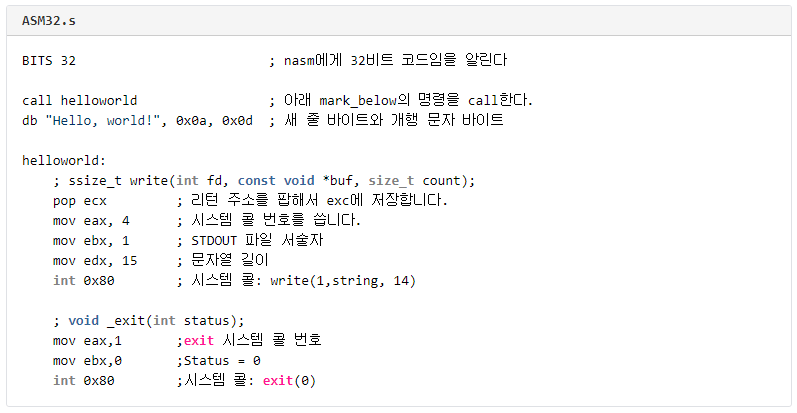
“int 0x80”, “syscall” : “int” 명령어의 피연산자 값으로 0x80을 전달하면 eax에 있는 시스템 함수를 호출, “syscall” 명령어를 호출하면 RAX에 있는 시스템 함수를 호출한다. 32bit, 64bit 마다 시스템 함수의 콜 번호가 다르다.

* 1. Assembly code example



* 1. Change to Shellcode

위의 예제는 혼자서 동작하지 않기 때문에 Shellcode가 아니다. 독립적으로 동작하기 위해서는 텍스트, 데이터 세그먼트를 사용하지 않아야 한다. 여기서 hello world 문자열을 전달하기 위해서는 데이터 세그먼트가 필요한데, 이는 call 명령어를 응용해 해결할 수 있다.



call helloworld를 수행하면 스택에 그 다음 명령어의 주소를 스택에 push 하므로 db “Hello world!”, 0x0A, 0x0D의 주소를 스택에 넣게 된다. helloworld 함수에서 pop ecx를 하게 되면 문자열의 주소가 ecx에 들어가게 되어 정상적으로 문자열을 전달 할 수 있게 된다.

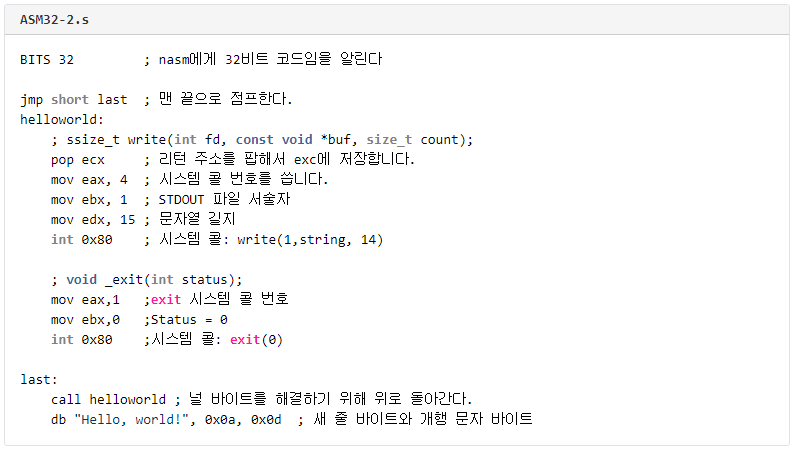


이를 파이썬으로 읽어 Machine code를 얻을 수 있다. 다시 이를 아래와 같은 코드를 통해 테스트할 수 있다.

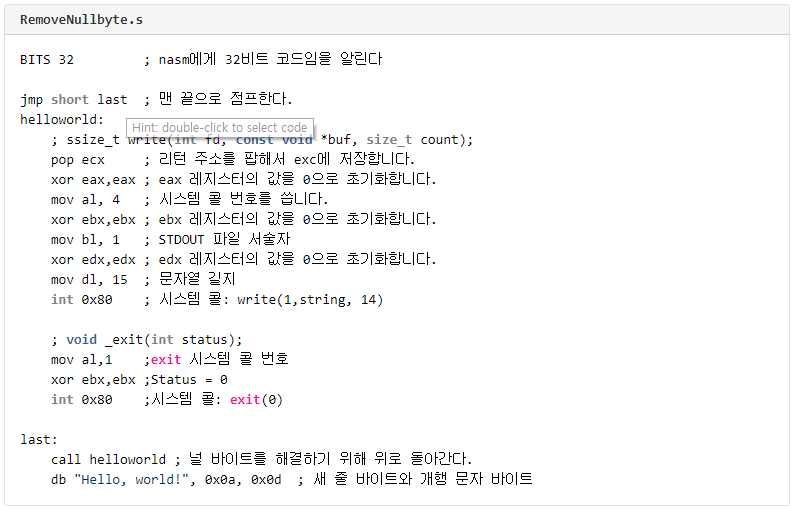


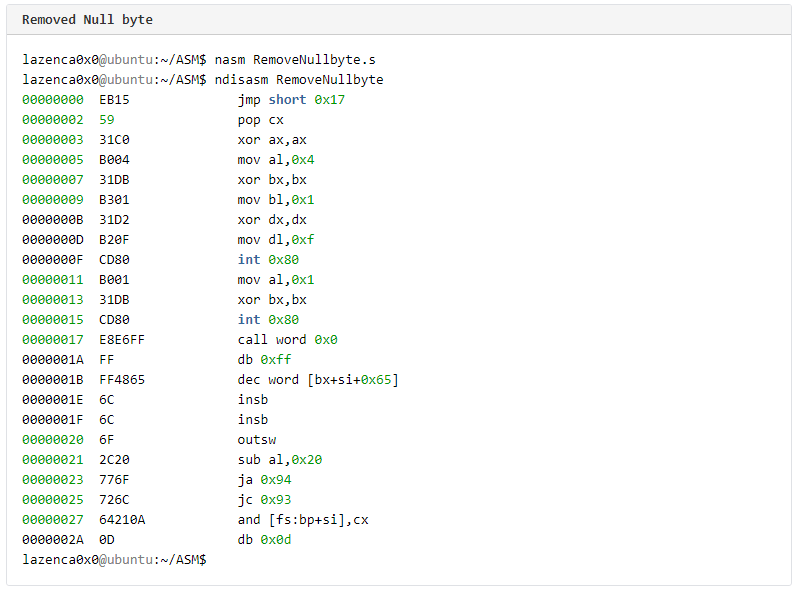
이 프로그램을 실행해 보면 오류가 나는데, 그 원인은 Shellcode 안에 들어있는 0x00 때문이다.

가장 먼저 call 명령어에 널 바이트가 존재하므로 helloworld보다 아래에 있는 함수에 먼저 점프 한 후, helloworld를 호출하게 되면 상대적 위치가 음수가 되므로 2의 보수 표현에 의해 0이 없어진다.



이 이후에 없애야 할 널 바이트는 mov명령어에 있는 0x00이다. 이는 넣는 값보다 레지스터의 크기가 크기 때문에 발생하는데, 무조건 al에 숫자를 넣게 되면 그전에 값이 남아있게 되므로 sub이나 xor을 이용해 레지스터를 0으로 초기화 해 줘야 한다. 이 때 sub 명령어보다 xor명령어가 플래그를 덜 수정하기 때문에 xor을 사용하는 것이 안전하다. 이렇게 다시 코드를 작성하면 다음과 같다.



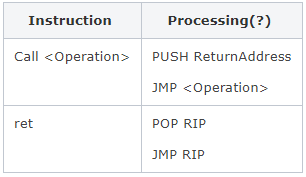




위 코드를 실행하게 되면 제대로 Hello world!가 출력된다.

1. Return to Shellcode
   1. call & ret

먼저 call과 ret 명령어의 이해가 필요하다.

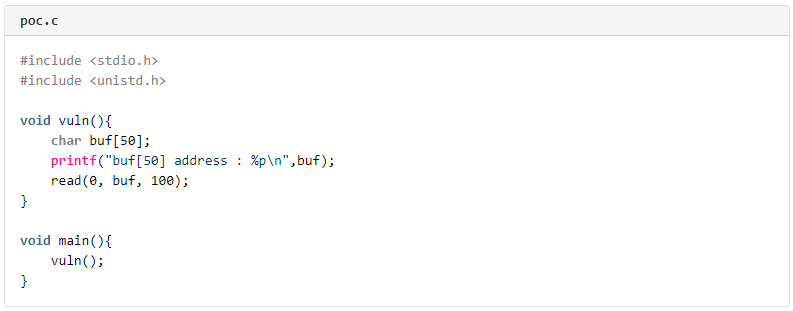


즉, ret이 실행되기 전에 스택의 ret 주소를 변경하면 프로그램의 흐름을 변경할 수 있다는 것이다

* 1. Permissions in memory

메모리에는 권한이 설정되어 있는데, 일반적으로 gcc에서는 DEP가 설정되어 있기 때문에 데이터가 저장되는 영역에는 실행권한이 없다.

* 1. Proof of concept



위의 코드는 buf[50]에 100개 문자열까지 입력을 받으므로 Stack Overflow가 일어날 수 있다. 이 프로그램을 디버깅 하면서 ret을 실행할 순서에 rsp가 가리키고 있는 스택의 최상위 메모리는 해당 함수가 종료 후 돌아갈 주소의 값을 담고 있다. main의 ret주소와 vul의 buf 주소의 차이는 72바이트이므로 문자를 72개 이상 입력하면 main의 ret주소를 덮어 쓸 수 있다.