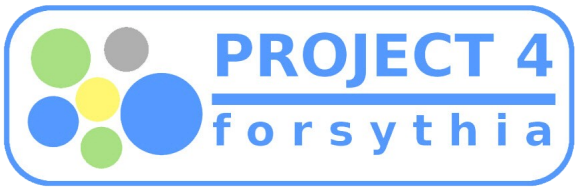

forsythiaLAB

HOW TO MAKE THE SHELL CODE

with reference [How to make the shell code] by mongii

on UBUNTU 10.04 Lucid Lynx

tyback

tyback@gmail.com

forsythiaLAB.co.cc

PROJECT4//forsythiaLAB.,2010.

* 이 문서는 해커스쿨 연구소인 Wiseguys 의 공개 강좌문서 중 mongii 님의 ‘쉘코드 만들기’ 를 참고하여 현재 우분투 리눅스의 최신버전인 루시드 링스(Ubuntu 10.04) 에서 실제 작성 및 테스트를 해보고 그 과정 및 내용을 정리한 문서입니다.

* 문서에 잘못된 내용이 있거나 추가할 사항이 있으면 언제든 알려주십시요.

안녕하세요, 개나리연구소 tyback 입니다.

패킷스톰이나 exploit-DB 같은 곳에 들어가보면 날이면 날마다 최신 보안취약점을 이용한 익스플로잇이 올라오는데요 , 버퍼 오버플로우 공격을 위한 코드 중 상당수가 겉 보기는 C 언어 같은데 알 수 없는 내용으로 도배를 한 코드가 올라와서 '도대체 이게 뭐하는 물건이지 -ㅅ-??' 고개를 갸웃거리는 경우...
이제 막 컴퓨터, 리눅스, 해킹에 입문하신 분들이라면 한 번 쯤은 있었을 겁니다.

.....

물론 저도 마찬가지에요;;;;

그러던 찰나 해커스쿨을 둘러보다가 연구소 Wiseguys 에 올라온 쉘코드 만들기 강좌를 보고는 " 아!! 이거구나!!! " 싶어 무작정 따라하기는 했는데 정리가 잘 안되서 말이죠....
그리고 실제로 해보면 강좌와는 약간 다른 결과가 나오기도 합니다.
음... 그래서 정리하는 김에 '쉘 코드 만들기 강좌의 강좌' 를 끄적이는거니까 , 쉘 코드가 뭔지 궁금하신 분들은 부담없이 보시고 실제로 리눅스 시스템에서 손가락 운동 겸 해보시길 바랍니다.
참고로 전 우분투 루시드 링스에 커널버전은 2.6.32-24-generic 되겠습니다.

우선 강좌에 나온데로 printf 가 아닌 write 로 문자열을 출력하는 프로그램을 만들어줍니다.

```
==========================================
int main()
{
       write(1, "Hello, Students!\n", 17);
}
==========================================
```

write() 함수는 대략 이렇게 써먹는다고 합니다.

```
============================================================
write(STDOUT_FILENO, buf, n);
write(출력대상,"문자열",출력될 문자열의 길이);

#define STDIN_FILENO    0       /* Standard input.  */
#define STDOUT_FILENO   1       /* Standard output.  */
#define STDERR_FILENO   2       /* Standard error output.  */
============================================================
```

소스코드 작성이 완료되었으면 소스를 컴파일하고 실행해봅시다.
이 때 중요한 것 한가지!!!
반드시 -static 옵션을 추가 해줘야 한다는 사실입니다.
만약 옵션을 추가하지 않았을 경우에 생기는 문제에 대해서는 조금 있다가 보여드리겠습니다.

프로그램이 실행되고 출력이 제대로 되는 것을 확인했으면 강좌데로 gdb를 실행시켜줍니다.

```
tyback@tyback-lucidLTS:~/projects/shell$ gdb write
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/tyback/projects/shell/write...(no debugging symbols found)...done.
(gdb)
```

뭐라고 말하는 건지 신경쓰면 지는거라는 사실을 명심하면서 복잡하게 생각하지 말고 하라는데로 입력해봅시다.

프로그램 상의 main 함수를 역어샘블링(disassembling) 하는 명령어입니다.
실제로 해보면 아시겠지만 강좌에서 나오는 내용과 미묘하게 다른 것을 알 수 있을 겁니다.
그리고... 위에서 얘기한 -static 옵션을 추가했을 때와 그렇지 않았을 때의 차이도 함께 보여드리겠습니다.

```
=======Ubuntu 10.04 에서 나온 결과==================
(gdb) disassem main
Dump of assembler code for function main:
   0x08048250 <+0>:  push   %ebp
   0x08048251 <+1>:  mov    %esp,%ebp
   0x08048253 <+3>:  and    $0xfffffff0,%esp
   0x08048256 <+6>:  sub    $0x10,%esp
   0x08048259 <+9>:  movl   $0x11,0x8(%esp)
   0x08048261 <+17>: movl   $0x80a75a8,0x4(%esp)
   0x08048269 <+25>: movl   $0x1,(%esp)
   0x08048270 <+32>: call   0x804f880 <write>
   0x08048275 <+37>: leave
   0x08048276 <+38>: ret
End of assembler dump.
(gdb)
```

=========**"-static" 옵션**을 추가하지 않았을 때==========

```
(gdb) disassem main
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  and    $0xfffffff0,%esp
   0x080483ea <+6>:  sub    $0x10,%esp
   0x080483ed <+9>:  movl   $0x11,0x8(%esp)
   0x080483f5 <+17>: movl   $0x80484d0,0x4(%esp)
   0x080483fd <+25>: movl   $0x1,(%esp)
   0x08048404 <+32>: call   0x804830c <write@plt>
   0x08048409 <+37>: leave
   0x0804840a <+38>: ret
End of assembler dump.
(gdb)
```

==========

==========해커스쿨 강좌에 나온 결과==========

```
(gdb) disassem main
Dump of assembler code for function main:
0x80481e0 <main>:       push   %ebp
0x80481e1 <main+1>:     mov    %esp,%ebp
0x80481e3 <main+3>:     sub    $0x8,%esp
0x80481e6 <main+6>:     sub    $0x4,%esp
0x80481e9 <main+9>:     push   $0x11
0x80481eb <main+11>:    push   $0x808ce68
0x80481f0 <main+16>:    push   $0x1
0x80481f2 <main+18>:    call   0x804ccf0 <write>
0x80481f7 <main+23>:    add    $0x10,%esp
0x80481fa <main+26>:    leave
0x80481fb <main+27>:    ret
End of assembler dump.
(gdb)
```

==========

결과가 다르게 나온다고 자기는 특이 케이스라고 생각하지는 마시길 -ㅅ-....

약간 다르게 나오기는 하지만 필요한 옵션을 제대로 넣어주면 적어도 삽질은 피할 수 있으니까 말이죠... -static 옵션을 넣었을 경우에는 <write> 라고 뜨는 반면 옵션을 빼먹었을 경우에는<write@plt> 라고 뜨는데요, 다음 과정에서 write 함수를 똑같이 역어샘블 했을 때 마찬가지로 결과가 판이하게 다르게 나오니까 꼼꼼하게 읽어보고 차근차근 따라해보시기 바랍니다.

음... 어셈블리어 자체를 배운적이 없어서(찾아본적은 더더욱 없어서;;;) 무슨 말인지 하나도 모르겠네요;;;;; 강좌를 봅시다.

다른건 몰라도 이건 꼭 알아두라면서 4줄을 짚어주네요. main+9 부터 main+18 까지 입니다.

```
==========================================
0x80481e9 <main+9>:     push   $0x11
0x80481eb <main+11>:    push   $0x808ce68
0x80481f0 <main+16>:    push   $0x1
0x80481f2 <main+18>:    call   0x804ccf0 <write>
==========================================
```

자... 보면 push(밀다, 밀어넣다.) $0x11 = 0x11 을 밀어넣어라. 어디에?? 컴퓨터니까... 메모리?? 스택???

계속해서... 0x808ce68 도 밀어넣고 0x1 도 똑같이 밀어넣고 마지막으로 0x804ccf0 을 콜!!!
옆에 친절하게 write 라고 써있네요 ㅇㅅㅇ...
그럼 도대체 저 값들은 뭘 의미하는 걸까요??

숫자를 0x** 라고 쓰면 컴퓨터에서는 보통 16진수를 의미하니까....

0x11 = 17
0x808ce68 = 134694856 뭐냐 넌 -ㅅ-;;;
0x01 = 1

자~ 자~ 이제 불과 5분전의 기억을 더듬어볼 차례 입니다.
write 를 이용해 프로그래밍을 했을 때 안에 적어준 내용... 기억하시려나 모르겠네요.
전 이 컴퓨터 저 컴퓨터 다 해본다고 설쳐대는 바람에 꿈속에서도 나타납니다만...
인자로 1,"Hello,Students!\n",17 을 집어 넣어줬었죠??
그래!! 대충 맞는 것 같아!!! 하시는 분들... 원래 대충하면 안됩니다만 맞습니다. 그 내용입니다.

강좌에 보니까 x/s 0x808ce68 라고 입력해서 내용을 확인하는데 그럼 대체 x/s 는 뭐 하는 물건일까요??

```
================================================================================
(gdb) help x/s
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
  t(binary), f(float), a(address), i(instruction), c(char) and s(string).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
```

```
The specified number of objects of the specified size are printed
according to the format.

Defaults for format and size letters are those previously used.
Default count is 1.  Default address is following last thing printed
with this command or "print".
(gdb)
```

==========

gdb 안에서 help 뭐뭐뭐 입력하니까 해당 명령어에 대한 도움말만 뜨네요.
어디보자... 그러니까 메모리 조사 : x/FMT 주소 라는군요.
정리하자면 메모리 주소를 조사하는 명령어가 되겠습니다.

x/s 뒤에 입력하는 값은 메모리 주소나 인자 값이 되는 건가??

역시나 대~충 감으로 때려잡고 계속 하겠습니다.
x/s 가 뭔지도 알았고 그 뒤에 입력하는 값이 메모리 주소 내지는 인자값 이라는 것도 알았는데 mongii 님 강좌와는 달리 우분투에 나오는 내용은 조금 다르게 나온단 말이지요....

==========

```
0x08048259 <+9>:     movl   $0x11,0x8(%esp)
0x08048261 <+17>:    movl   $0x80a75a8,0x4(%esp)
0x08048269 <+25>:    movl   $0x1,(%esp)
0x08048270 <+32>:    call   0x804f880 <write>
```

==========

눈에 익은 숫자를 보고 대충 짚어본 코드입니다.
x/s 로 메모리 주소 0x80a75a8 를 조사해봅시다.

==========

```
(gdb) x/s 0x80a75a8
0x80a75a8:     "Hello,students!\n"
(gdb)
```

==========

이런 세상에!!! 0x80a75a8 호실에 헬로 스튜던트가 살고있었네요!!!
아예... 죄송합니다.... 끼익~!

17 을 밀어넣고(push) 뭐시기를 밀어넣고 1을 밀어넣고 마지막으로 write 를 콜!!
그리고 헬로 스튜던트를 짜잔~
여기서 주목할 점은 인자를 반대로 밀어넣는다는 사실입니다.
그러니까 소스코드에 적을 때는 123 이라고 적어도 실제 메모리에 올릴 때는 321 순으로 올라간다는 것입니다.

```
==========================================================================
(gdb) disassem write
Dump of assembler code for function write:
   0x0804f880 <+0>:  cmpl   $0x0,%gs:0xc
   0x0804f888 <+8>:  jne    0x804f8ab <write+43>
   0x0804f88a <+0>:  push   %ebx
   0x0804f88b <+1>:  mov    0x10(%esp),%edx
   0x0804f88f <+5>:  mov    0xc(%esp),%ecx
   0x0804f893 <+9>:  mov    0x8(%esp),%ebx
   0x0804f897 <+13>: mov    $0x4,%eax
   0x0804f89c <+18>: int    $0x80
   0x0804f89e <+20>: pop    %ebx
   0x0804f89f <+21>: cmp    $0xfffff001,%eax
   0x0804f8a4 <+26>: jae    0x80518e0 <__syscall_error>
   0x0804f8aa <+32>: ret
   0x0804f8ab <+43>: call   0x80506e0 <__libc_enable_asynccancel>
   0x0804f8b0 <+48>: push   %eax
   0x0804f8b1 <+49>: push   %ebx
   0x0804f8b2 <+50>: mov    0x14(%esp),%edx
   0x0804f8b6 <+54>: mov    0x10(%esp),%ecx
   0x0804f8ba <+58>: mov    0xc(%esp),%ebx
   0x0804f8be <+62>: mov    $0x4,%eax
   0x0804f8c3 <+67>: int    $0x80
   0x0804f8c5 <+69>: pop    %ebx
   0x0804f8c6 <+70>: xchg   %eax,(%esp)
   0x0804f8c9 <+73>: call   0x8050660 <__libc_disable_asynccancel>
   0x0804f8ce <+78>: pop    %eax
   0x0804f8cf <+79>: cmp    $0xfffff001,%eax
   0x0804f8d4 <+84>: jae    0x80518e0 <__syscall_error>
   0x0804f8da <+90>: ret
End of assembler dump.
(gdb)
==========================================================================

===이건 참고용..."-static" 옵션을 뺐을 때 결과물===
(gdb) disassem write
Dump of assembler code for function write@plt:
   0x0804830c <+0>:  jmp    *0x804a004
   0x08048312 <+6>:  push   $0x8
   0x08048317 <+11>: jmp    0x80482ec
End of assembler dump.
```

(gdb)
//응??? 뭐라고 써있는거야??
==========

write 함수를 역어샘했습니다. 뭐라고 하는지 하나도 모르겠네요....
그냥 화면에 저거 하나만 딱 띄워놓고 있으면 마치 뭔가 하는 것 처럼 보여서 참 보기에는 좋은데 이걸 하나하나 뜯어볼 생각을 하니 진짜 막막하네요.....

여기서 눈여겨 보라는 부분은 바로 이 부분!!
==========

```
   0x0804f88b <+1>:  mov    0x10(%esp),%edx
   0x0804f88f <+5>:  mov    0xc(%esp),%ecx
   0x0804f893 <+9>:  mov    0x8(%esp),%ebx
   0x0804f897 <+13>: mov    $0x4,%eax
   0x0804f89c <+18>: int    $0x80
```

==========

'mov' 같은 어샘블리 명령어를 OPcode라고 하는데, 처음보는 옵코드가 있네요.
int 는 interrupt 의 약자인데 시스템에 특정 신호를 보내는 역할을 한다고 합니다.
0x80 인터럽트는 커널에서 사용자들에게 제공해주는 함수를 호출하라는 의미라는데 바로 한 줄위에서 어떤 함수인지 받아서 호출한다는군요.
(주석: 이 역시 실험을 조금 해보면 알겠지만 정확하게는 스택에서 가장 먼저 나오는 값을 받아서 호출합니다.)
0x4 니까... 4번째?? 4번?? 그러니까 시스템 호출 테이블(목록) 4번째 항목을 불러온다는 말이군요.

그럼 도대체 우분투는 그 시스템 호출 테이블이 어디있다는 거죠 -ㅅ-???
음... 그래도 리눅스니까... 내부는 비슷하니까 강좌대로 해볼까??
==========

```
tyback@tyback-lucidLTS:~$ cd /usr/include/asm
tyback@tyback-lucidLTS:/usr/include/asm$ ls -l u*
-rw-r--r-- 1 root root   339 2010-07-06 00:34 ucontext.h
-rw-r--r-- 1 root root    82 2010-07-06 00:34 unistd.h
-rw-r--r-- 1 root root  9846 2010-07-06 00:34 unistd_32.h
-rw-r--r-- 1 root root 22035 2010-07-06 00:34 unistd_64.h
tyback@tyback-lucidLTS:/usr/include/asm$ cat unistd.h | grep write
tyback@tyback-lucidLTS:/usr/include/asm$ -ㅅ- 응?? 없다고???
tyback@tyback-lucidLTS:/usr/include/asm$ cat unistd.h
# ifdef __i386__
#  include "unistd_32.h" -> 32비트니까 이거인듯....
# else
#  include "unistd_64.h"
```

```
# endif
tyback@tyback-lucidLTS:/usr/include/asm$ cat unistd_32.h | grep writ
#define __NR_write             4 -> 찾았다!!!
#define __NR_writev          146
#define __NR_pwrite64        181
#define __NR_pwritev         334
tyback@tyback-lucidLTS:/usr/include/asm$
```

==========

훗... 삽질 한 번 해보니 금방 나오네요. /usr/include/asm/unistd_32.h 에 있습니다.
grep 명령으로 write 가 있는 내용만 뽑아봤습니다.
보니까 정말로 write 라고 써있네요.
확인했으니까 이걸로 끝.

다시 디버거로 돌아와서

```
   0x0804f88b <+1>:  mov    0x10(%esp),%edx
   0x0804f88f <+5>:  mov    0xc(%esp),%ecx
   0x0804f893 <+9>:  mov    0x8(%esp),%ebx
   0x0804f897 <+13>: mov    $0x4,%eax
   0x0804f89c <+18>: int    $0x80
```

mov 는 a 를 b로 복사하는 것을 말하는데 쉼표로 구분짓습니다.
그러니까... +1 에서는 0x10 이라는 값을 edx(레지스터) 로 복사하는 것을 말하고 같은 원리로 0xc, 0x8, 0x4 도 차례대로 복사합니다.

여기서 또 한가지 봐둬야하는 것은 레지스터 이름입니다. edx, ecx, ebx, eax 의 역순으로 레지스터에 각 해당하는 값들이 저장되는 것을 확인할 수 있습니다. 그리고 이 값들이 write 함수의 인자로 작용합니다.

인자로 사용할 값들 옆에 써있는 %esp 는 Stack Pointer, %ebp 는 Base Pointer 라는데 거기에 대한 설명은... 언젠간 하겠죠;;;;;;;

휴 -ㅅ-... 일단 어떻게 읽어야 되는지는 감을 잡았네요.
지금까지 한 얘기를 정리해서 write 프로그램이 작동하는 과정(그러고 보니 'write()' 함수랑 이름을 똑같이 만들어서 헷갈리네요;;;) 즉, 헬로 스튜던트가 출력되는 과정을 적어보면 =>

(1) 스택에 17이 먼저 저장.
(2) 스택에 문자열 "Hello, Students!\n" 의 시작주소 저장.
(3) 스택에 1 저장.
(4) write 함수 호출.

(5) 17을 edx에 저장
(6) 문자열 시작주소를 ecx에 저장
(7) 1을 ebx에 저장
(8) 시스템 호출 번호 4번(write)을 eax에 저장
(9) 시스템 호출 테이블을 부르는 int 0x80 인터럽트 발생.
(10) 레지스터 값을 참고해 write() 실행.

이런식이 된답니다. 이 과정들을 이제 어샘블리어로 코딩하면 되는데
강좌를 참고해서 코딩을 해보겠습니다.
어샘블리 프로그램의 소스코드 확장자는 *.s 입니다.

```
==================================
.LC0:
        .string "Hello, Students!\n"
.globl main
main:
        movl $0x04, %eax
        movl $0x01, %ebx
        movl $.LC0, %ecx
        movl $0x11, %edx
        int $0x80
        ret
==================================
```

강좌에서는 레지스터 이름 순서대로 정리해서 코딩했는데, 코딩 순서에 상관없이
해당 레지스터에 인자나 주소값 들이 제대로 들어가기만 하면 됩니다.

시스템 호출 테이블을 부르는 인터럽트가 맨 마지막에 들어가야 하는건 당연한거죠?
특별한 경우가 아니라면 컴퓨터에게 있어서 외상이나 가불같은 건 없습니다 -ㅅ-...
뭘 주고 뭘 해야지 뭘 저질러 놓고 나중에 뭘 주는건 아니죠..

ret 은 뭐하는 물건인지 잘 모르겠습니다만... 귀찮아서 나중에 찾아보도록 하죠.
저게 없어도 컴파일 하고 프로그램을 실행시키면 잘만 돌아가니까요.

우분투에서 실행하본 결과 그 어떤 오류메세지 없이 정상적으로 실행됬습니다.
강좌에서는 한 줄 경고말씀 하시면서 실행되던데 그 부분을 고치기 위해 비슷한 과정으로 exit(0) 을 어샘블리어로 고쳐서 집어넣어줬습니다.
한 번 안해볼 수가 없죠 또...

```
=============================================================================
tyback@tyback-lucidLTS:~$ cat /usr/include/asm/unistd_32.h ¦ grep exit
#define __NR_exit               1 -> 이거니까...
```

```
#define __NR_exit_group            252
tyback@tyback-lucidLTS:~$vim exit

movl    $0x01, %eax
movl    $0x00, %ebx
int     $0x80
```

-> 위 내용을 ret을 지우고 맨 마지막에 추가해줍니다. 다음은 완성된 소스코드 입니다.

```
.LC0:
       .string "Hello, Students!\n"
.globl main
main:
       movl   $0x04, %eax
       movl   $0x01, %ebx
       movl   $.LC0, %ecx
       movl   $0x11, %edx
       int    $0x80
       movl    $0x01, %eax
       movl    $0x00, %ebx
       int     $0x80
```

==========

어샘블리어도 공부하고 참 좋은 것 같습니다.... 네...네...
컴파일 해서 실행해보시고 /-ㅅ-/.... 아, -static 옵션은 필요없습니다.

자, 하지만 아직 우리가 원하는 쉘코드를 만들지는 않았으니까 Let's Keep it up!!

우리가 지금까지 강좌를 보면서 만든 것은 기계어 코드가 아니라
기계어를 보다 쉽게 사용하기 위해 만든 어샘블리어로 만든 어샘블리 코드입니다.
고지가 하나 밖에 안남았습니다 -ㅅ-!!!!
지금까지 만든 초 간단 어샘블리 프로그램을 기계어로 바꿔봅시다.

==========

```
tyback@tyback-lucidLTS:~/projects/shell$ objdump -d shell

shell:     file format elf32-i386


Disassembly of section .init:

08048298 <_init>:
```

```
 8048298:     55                   push   %ebp
 8048299:     89 e5                mov    %esp,%ebp
 804829b:     53                   push   %ebx
 804829c:     83 ec 04             sub    $0x4,%esp
 804829f:     e8 00 00 00 00       call   80482a4 <_init+0xc>
```
[스크롤의 압박이 예상되어 중간 과감하게 생략]
```
 804849f:     e8 8c fe ff ff       call   8048330 <__do_global_dtors_aux>
 80484a4:     59                   pop    %ecx
 80484a5:     5b                   pop    %ebx
 80484a6:     c9                   leave
 80484a7:     c3                   ret

tyback@tyback-lucidLTS:~/projects/shell$
```
==========

-ㅅ-;;;;;;;;;;;;;

뭐...뭐지.... 이건....

분명히 어샘블리어로 적은 코드는 기껏해야 12줄 밖에 안되는데 나오는 물건은 이렇게나;;;;

mongii 님의 강좌를 한 번 읽어보신 분은 다른건 눈에 들어오지 않고 이것만 보고 계실거라 생각합니다.

==========

```
080483c6 <main>:
 80483c6:     b8 04 00 00 00       mov    $0x4,%eax
 80483cb:     bb 01 00 00 00       mov    $0x1,%ebx
 80483d0:     b9 b4 83 04 08       mov    $0x80483b4,%ecx
 80483d5:     ba 11 00 00 00       mov    $0x11,%edx
 80483da:     cd 80                int    $0x80
 80483dc:     b8 01 00 00 00       mov    $0x1,%eax
 80483e1:     bb 00 00 00 00       mov    $0x0,%ebx
 80483e6:     cd 80                int    $0x80
 80483e8:     90                   nop
 80483e9:     90                   nop
 80483ea:     90                   nop
 80483eb:     90                   nop
 80483ec:     90                   nop
 80483ed:     90                   nop
 80483ee:     90                   nop
 80483ef:     90                   nop
```
==========

바로 main 이라 적혀있는 부분이죠... 자고로 삽질을 피하기 위해선 감이 좋아야 되요 감이...

우리가 어샘블리어로 작성한 내용이 표시되고 그 왼쪽에 기계어로 표시되어 있죠?
저것들을 이제 '주~욱 붙여넣기만 하면 진정한 기계어 코드가 완성이 된다!!' 가 완벽한 시나리오인데 자세히 보세요...

응 -ㅅ-??

헬로 스튜던트는 어디갔죠?? 프로그램 짜면서 문자열의 시작주소를 적어준 것만 있어서
이걸 그대로 썼다간 죽도밥도 안되는데 말이죠... 쓰읍....
다시 한 번 삽질의 기운이 느껴지는군요...

```
80483d0:      b9 b4 83 04 08       mov    $0x80483b4,%ecx
```

조금 더 자세하게 얘기하자면 b9 b4 83 04 08 를 이어붙여서 /xb9/xb4/x83/x04/x08 로 적어서 기계어 프로그램으로 만들어 실행하게 되면 그 프로그램이 실행할 당시에 0x80483b4 라는 주소지에 헬로 스튜던트가 아닌 다른 문자열님이나 엉뚱한 분이 살고 계실 가능성이 농후해서 프로그램을 100만번 실행해도 헬로 스튜던트 님을 볼 수 없다는 얘기입니다.

저건 절대주소지 상대주소가 아니니까요...
그러니까 프로그램을 실행했을 때 0x80483b4 라는 주소지에 헬로 스튜던트 님이 살고 계시게끔 기계어를 또 만들어줘야 한다.... 는 말씀.

그 해답을 강좌를 적어주신 mongii 님께서 알려주시길...
문자열 시작 주소를 스택에 저장해뒀다가 그 다음 스택에서 %ecx 레지스터에 저장 "하면 된다!!!" 고 말이죠. 하면 됩니다... 합시다.. 여러분... 삽질...

```
==================================
.globl main
main:
        call func
        .string "Hello, Students!\n"
func:
        movl $0x04, %eax
        movl $0x01, %ebx
        popl %ecx
        movl $0x11, %edx
        int $0x80
        movl $0x01, %eax
        movl $0x00, %ebx
        int $0x80
==================================
```

생각하지 못한거라 참 어렵게 보이네요... 봅시다 봅시다.

call 명령으로 func 를 불러옵니다.
그리고 call 명령 다음에 들어가는 내용의 주소가 스택에 저장됩니다.
(call 명령에 의해 어떤 함수가 호출되면 함수 종료후 실행되는 다음 내용의 주소가 스택에 저장된답니다.)

func 를 보면
	eax 에 0x04 = write
	ebx 에 0x01 = 1
	ecx 에는 스택의 가장 맨 꼭대기에 있는 값을(popl)
	edx 에는 0x11 = 17
을 저장하게 되어있고 0x80 인터럽트.
이건 write(1,스택 맨 꼭대기 값,17) 라고 나오겠네요.
그리고 다시
	eax 에 0x01 = 1
	ebx 에 0x00 = 0
을 놓고 0x80 인터럽트... 이건 exit(0) 였죠??

써놓고 보니 이해가 되네요... 휴;;;
그럼 이것도 기계어로 바꿔주기 위해서 objdump 를 해보도록 하죠.
그전에 컴파일은 필수 입니다!!!
==========

```
080483b4 <main>:
 80483b4:	e8 12 00 00 00		call   80483cb <func>
 80483b9:	48			dec    %eax
 80483ba:	65			gs
 80483bb:	6c			insb   (%dx),%es:(%edi)
 80483bc:	6c			insb   (%dx),%es:(%edi)
 80483bd:	6f			outsl  %ds:(%esi),(%dx)
 80483be:	2c 20			sub    $0x20,%al
 80483c0:	53			push   %ebx
 80483c1:	74 75			je     8048438 <__libc_csu_init+0x38>
 80483c3:	64 65 6e		outsb  %fs:%gs:(%esi),(%dx)
 80483c6:	74 73			je     804843b <__libc_csu_init+0x3b>
 80483c8:	21 0a				and    %ecx,(%edx)
	...

080483cb <func>:
 80483cb:	b8 04 00 00 00		mov    $0x4,%eax
 80483d0:	bb 01 00 00 00		mov    $0x1,%ebx
```

```
80483d5:     59                  pop    %ecx
80483d6:     ba 11 00 00 00      mov    $0x11,%edx
80483db:     cd 80               int    $0x80
80483dd:     b8 01 00 00 00      mov    $0x1,%eax
80483e2:     bb 00 00 00 00      mov    $0x0,%ebx
80483e7:     cd 80               int    $0x80
80483e9:     90                  nop
80483ea:     90                  nop
80483eb:     90                  nop
80483ec:     90                  nop
80483ed:     90                  nop
80483ee:     90                  nop
80483ef:     90                  nop
```

==========

자, 다 됐습니다.
그런데 이번에도 역시나 '문자열이 하나도 보이지 않는다 ;ㅂ;!!!'
머엉 -ㅂ-....
이거슨... 대체...

그러나 우리의 바이블인 그 분의 강좌를 보면 말이죠....
화면에 출력된 내용은 프로그램 내용을 억지로 16진수에서 어샘블리어로 바꾸면서 생기는 문제로 저 안에 헬로 스튜던트가 아스키코드로 저장되어있으니 잘 찾아보시라... 고 말이죠;;;
그리고 실제로 노가다를 해보면 답이 나옵니다.

==========

```
48       H
65       e
6c       l
6c       l
6f       o
2c 20    , [ ]
53       S
74 75    t u
64 65 6e  d e n
74 73    t s
21 0a    !
```

==========

훗 -ㅅ-... 찾았다....
이제 이것들을 줄줄이 이어붙여주면 기계어코드가 됩니다.
다같이 모종삽을 들고 삽질을 해봅시다.

여기서 중요한거라면 컴퓨터에 이 숫자들이 16진수임을 표시해주어야 한다는건데 이는 \x 를 붙여줌으로써 표시하면 됩니다.

그리고 문자열!!

objdump 를 해보고 확인한 것만으로도 알 수 있는 내용으로 약간 걱정되는 부분이 있습니다.
objdump 를 했을 때 우리가 작성한 func 부분은 제대로 출력이 됐는데, 문자열을 포함하고 있었던 메인 옵코드들은 이상하게 출력된것을 확인할 수 있습니다. 이는 프로그램을 뜯어서 16진수로 나열하고 그 내용을 억지로 다시 어샘블리어로 변환하는 과정에서 생긴 문제인데, C 컴파일러도 이와 같은 경우가 발생하지 않겠느냐 하는 점 입니다.

다행이도 C 컴파일러는 \x 를 붙이지 않는 문자에 대해서는 아스키 코드 문자로 인식하고 컴파일을 진행한다고 합니다. 그래서 우리가 얻어낸 16진수 값 중에 문자열에 해당하는 값만 문자로 바꿔서 적어줘도 정상적으로 컴파일/실행이 된다고 합니다.

실제로 한 번 해보도록 하죠.

```
int main()
{
	char *code =
			"\xe8\x12\x00\x00\x00Hello, Students!\n\x00"

			//.globl main
			//main:
			//call func
			//.string "Hello, Students!\n"

			"\xb8\x04\x00\x00\x00\xbb\x01\x00\x00\x00\x59\xba\x11\x00\x00\x00"

			//func:
			//movl $0x04, %eax
			//movl $0x01, %ebx
			//popl %ecx
			//movl $0x11, %edx
			//int $0x80

			"\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80";

			//movl $0x01, %eax
			//movl $0x00, %ebx
			//int $0x80
```

```
                        //이 줄이 exit(0)를 만들어줬던 그거...

        void (*pointer)(void);

        pointer = (void *)code;

        pointer();
}
```

==========

 위 처럼 해당하는 기능에 맞춰서 딱딱 끊어서 작성해주는 것이 보기에도 좋고 에러도 발생하지 않습니다. 기껏 삽질해서 만들었는데 에러 뜨면 기분 완전 다운되잖아요 -ㅅ-....
 떨리는 마음으로 정성스럽게 독수라 타법으로 한 글자 한 글자 입력해서 컴파일 명령을 실행해주면...

==========

```
ctyback@tyback-lucidLTS:~/projects/shell$ cat preshell2.c
int main()
{
        char *code =
                        "\xe8\x12\x00\x00\x00\x48\x65\x6c\x6c\x6f\x2c\x20\x53\x74\x75\x64\x65\x6e\x
74\x73\x21\x0a\x00"
                        "\xb8\x04\x00\x00\x00\xbb\x01\x00\x00\x00\x59\xba\x11\x00\x00\x00"
                        "\xcd\x80\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80";

        void (*pointer)(void);

        pointer = (void *)code;

        pointer();
}
//메롱 ㅇㅠㅇ~ 전 16진수 코드 그대로 썼습니다.
tyback@tyback-lucidLTS:~/projects/shell$ gcc -o finalshell2 preshell2.c
tyback@tyback-lucidLTS:~/projects/shell$ ./finalshell2
Hello, Students!
tyback@tyback-lucidLTS:~/projects/shell$
```

==========

 감격 그 자체 ;ㅂ;!!!
 그동안 고생한 노력이 빛을 발하면서 에러 끝털없이 정상적으로 실행되는 아름다운 모습!!
 개나리는 특별하니까(???) 문자까지 전부 16진수 코드로 만들어서 이게 뭐하는 물건인지 알아볼 수 없게 만든 완전무결(??) 한 기계어 코드가 완성됐습니다!!!

크흙... 성공했어...

지금처럼 간단하게 문자열을 출력하는 코드만 만드는데 이런 삽질을 하는데 스톰에 올라오는 각종 쉘 코드들은 정말... 덜덜덜;;;;

아무튼 이 문서를 통해서 컴퓨터 메모리나 어샘블리어에 조금 더 관심을 갖고 재미있게 공부할 수 있었으면 합니다.

다음에는 추가적인 내용들을 한 번 해보도록 하겠습니다.
강좌는 이게 끝이 아니에요 -ㅅ-...
삽질은 계속 되야 하니까.....