# 어셈블리 언어 실습

# 어셈블리 언어 실습

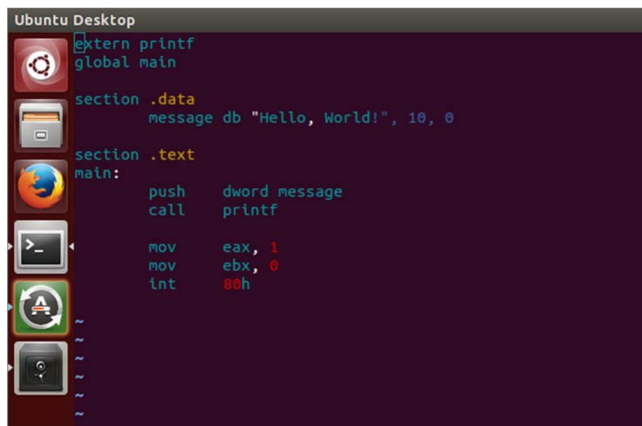
## 사용할 도구

- **qemu**
  - 가상화, ARM 어셈블리 프로그래밍
- **nasm**
  - 어셈블러(x86)
- **as (GAS)**
  - 어셈블러(ARM)
- **ld**
  - 리눅스용 링커
- **objdemp**
  - 오브젝트 덤프
- **gdb**
  - 디버거

```
stud@stud:~$ objdump -d a.out

a.out:     file format elf32-i386


Disassembly of section .plt:

08048170 <printf@plt-0x10>:
 8048170:       ff 35 4c 92 04 08       pushl  0x804924c
 8048176:       ff 25 50 92 04 08       jmp    *0x8049250
 804817c:       00 00                   add    %al,(%eax)
        ...

08048180 <printf@plt>:
 8048180:       ff 25 54 92 04 08       jmp    *0x8049254
 8048186:       68 00 00 00 00          push   $0x0
 804818b:       e9 e0 ff ff ff          jmp    8048170 <printf@plt-0x10>

Disassembly of section .text:

08048190 <main>:
 8048190:       68 58 92 04 08          push   $0x8049258
 8048195:       e8 e6 ff ff ff          call   8048180 <printf@plt>
 804819a:       b8 01 00 00 00          mov    $0x1,%eax
 804819f:       bb 00 00 00 00          mov    $0x0,%ebx
 80481a4:       cd 80                   int    $0x80
```

# 어셈블리 언어 실습

- **환경 셋팅**
  - 실습용 리눅스 이미지를 VMplayer에서 불러온 후 실행
  - ID: stud / PASS: 123123
  - apt-get install nasm
  - apt-get install vim
  - 터미널 실행 후 경로 생성 x86_exam/hello
  - 경로 이동 후 vi hello.asm 실행
  - 샘플 코드 작성
  - `nasm -felf32 hello.asm && ld -I/lib/ld-linux.so.2 -lc --entry main hello.o`




```
X86_exam/hello/hello.asm

extern printf
global main

section .data
        message db "Hello, World!", 10, 0

section .text
main:
        push        dword message
        call        printf

        mov         eax, 1
        mov         ebx, 0
        int         80h
```

# 어셈블리 언어 실습

## PUSH, POP

- **PUSH: 스택에 데이터를 삽입**
  - push word / dword / word
  - push 명령어는 자동으로 ESP 를 4 바이트 감소 시킴
- **POP: 스택에서 데이터를 꺼냄**
  - 스택에서 4 바이트를 꺼낸 후 지정한 레지스터에 삽입
  - pop 명령어는 자동으로 ESP 를 4 바이트 증가 시킴

**[Vim 환경 세팅]**

```
$ cd
 //home 경로로 이동

$ vi .vimrc
 //.vimrc 숨김 파일 오픈(vim의 설정 파일)
________________________________________
set number
:wq
________________________________________
```

# 어셈블리 언어 실습

## PUSH, POP

- **프로그램 실행해 보기**
  - 오브젝트 코드 생성
    - `nasm -felf32 exam1.asm`
  - 링킹
    - `ld --entry main exam1.o`
  - 실행
    - `./a.out`
  - Segmentation Fault!
  - x86 어셈블리 언어는 가변 길이 명령어

**[exam1.asm]**

```
pwd: x86_exam/exam1/exam1.asm
$ vi exam1.asm
```

```
global main

section .text
main:
        push    10h
        push    0x12345678
        pop     eax
        pop     ebx
:wq
```

```
$ nasm -felf32 exam1.asm
 //asm 파일을 nasm으로 어셈블리

$ ld -entry main exam1.o
 //어셈블리로 생성된 오브젝트 파일을 링크

$ file a.out
 //실행 파일임을 확인

$ ./a.out
 //링크된 실행 파일을 실행
```

# 어셈블리 언어 실습

## Segmentation Fault in Linux

- **프로그램이 허용되지 않은 메모리 영역에 접근을 시도하거나, 잘못된 방법으로 메모리 영역에 접근을 시도할 경우 발생**
  - 잘못된 메모리 영역에 대한 접근 발생
    - 프로그래머가 생각하지 못한 사용자 입력 값이 있거나
    - 그냥 프로그램을 잘못 짬
    - 프로그램에 문제가 있다는 의미
      - ➔ 취약점? Exploitable?

- **끝맺음을 제대로 하지 않았기 때문에 에러가 발생**
  - GDB 에서 보면 다음 실행할 명령어 를 제대로 찾지 못하는 것을 확인 가능
  - 프로그램의 끝은 시스템 콜을 이용해야 함

# 어셈블리 언어 실습

## objdump

- **프로그램의 에러 확인하기**

- **objdump**
  - 오프젝트 파일을 덤프해보는 것 (-d 옵션 사용)
    - disassemble: 실행파일에서 어셈블리를 추출하는 것
    - `$ objdump -d a.out`

[exam1.asm]

```
global main

section .text
main:
            push        10h
            push        0x12345678
            pop         eax
            pop         ebx
:wq
```

```
stud@stud:~/x86_exam/exam1$ objdump --help
Usage: objdump <option(s)> <file(s)>
 Display information from object <file(s)>.
 At least one of the following switches must be given:
  -a, --archive-headers    Display archive header information
  -f, --file-headers       Display the contents of the overall file header
  -p, --private-headers    Display object format specific file header contents
  -P, --private=OPT,OPT... Display object format specific contents
  -h, --[section-]headers  Display the contents of the section headers
  -x, --all-headers        Display the contents of all headers
  -d, --disassemble        Display assembler contents of executable sections
  -D, --disassemble-all    Display assembler contents of all sections
  -S, --source             Intermix source code with disassembly
  -s, --full-contents      Display the full contents of all sections requested
  -g, --debugging          Display debug information in object file
  -e, --debugging-tags     Display debug information using ctags style
  -G, --stabs              Display (in raw form) any STABS info in the file
  -W[lLiaprmfFsoRt] or
  --dwarf[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,
          =frames-interp,=str,=loc,=Ranges,=pubtypes,
          =gdb_index,=trace_info,=trace_abbrev,=trace_aranges,
          =addr,=cu_index]
```

```
stud@stud:~/x86_exam/exam1$ objdump -d a.out

a.out:     file format elf32-i386


Disassembly of section .text:

08048100 <main>:
 8048100:       6a 10                   push   $0x10
 8048102:       68 78 56 34 12          push   $0x12345678
 8048107:       58                      pop    %eax
 8048108:       5b                      pop    %ebx
```

# 어셈블리 언어 실습

## GDB

- **프로그램이 실행되는 동안 그 프로그램 안에서 어떤 일이 일어나는지 확인하는 도구**
  - 리눅스 환경에서의 리버싱 필수 도구
  - $ man gdb

```
GDB(1)                                GNU Development Tools                                GDB(1)

NAME
       gdb - The GNU Debugger

SYNOPSIS
       gdb [-help] [-nh] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps]
           [-tty=dev] [-s symfile] [-e prog] [-se prog] [-c core] [-p procID]
           [-x cmds] [-d dir] [prog|prog procID|prog core]

DESCRIPTION
       The purpose of a debugger such as GDB is to allow you to see what is going on "inside" another program while it executes -- or what another
       program was doing at the moment it crashed.

       GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

       ·   Start your program, specifying anything that might affect its behavior.

       ·   Make your program stop on specified conditions.

       ·   Examine what has happened, when your program has stopped.

       ·   Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

       You can use GDB to debug programs written in C, C@t{++}, Fortran and Modula-2.

       GDB is invoked with the shell command "gdb".  Once started, it reads commands from the terminal until you tell it to exit with the GDB command
       "quit".  You can get online help from GDB itself by using the command "help".

       You can run "gdb" with no arguments or options; but the most usual way to start GDB is with one argument or two, specifying an executable
       program as the argument:
```

# 어셈블리 언어 실습

- **GDB 실행해보기**

```
stud@stud:~/x86_exam/exam1$ gdb a.out
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/l
This is free software: you are free to change and redistribu
There is NO WARRANTY, to the extent permitted by law.  Type
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"
Reading symbols from a.out...(no debugging symbols found)...
(gdb) q
stud@stud:~/x86_exam/exam1$ gdb a.out
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/l
This is free software: you are free to change and redistribu
There is NO WARRANTY, to the extent permitted by law.  Type
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"
Reading symbols from a.out...(no debugging symbols found)...
(gdb)
```

**[a.out]**

```
pwd: x86_exam/exam1/
$ gdb a.out
 //a.out에 대해 gdb 실행

(gdb) disassemble main
 //a.out의 main함수를 disassemble
```

```
Dump of assembler code for function main:
   0x08048060 <+0>:   push   $0x10
   0x08048062 <+2>:   push   $0x12345678
   0x08048067 <+7>:   pop    %eax
   0x08048068 <+8>:   pop    %ebx
End of assembler dump.
 //어셈블리 코드 확인 가능 disas 약자로 실행 가능
```

```
(gdb) set disassembly-flavor intel
 //AT&T 방식을 Intel방식으로 출력
(gdb) disas main
```

```
Dump of assembler code for function main:
   0x08048060 <+0>:   push   0x10
   0x08048062 <+2>:   push   0x12345678
   0x08048067 <+7>:   pop    eax
   0x08048068 <+8>:   pop    ebx
End of assembler dump.
```

# 어셈블리 언어 실습

## 레지스터 정보 확인

- **레지스터 정보 확인**

```
(gdb) info register
The program has no registers now.
```

  - 프로그램이 실행되지 않았기 때문

- **프로그램 실행**
  - (gdb)run
    - Segmentation fault 발생

- **Breakpoint 설정**
  - (gdb) break main

- **main함수를 disassemble**
  - (gdb) disas main

**[a.out]**

```
pwd: x86_exam/exam1/a.out
(gdb) info register
 //프로그램이 실행되지 않았기 때문에 레지스터 없음

(gdb) run
```

```
Starting program:
/home/stud/x86_exam/exam1/a.out

Program received signal SIGSEGV, Segmentation
fault.
0x08048069 in ?? ()
```

```
(gdb) b main
 //main 함수에 breakpoint를 설정
(gdb) disas main
```

```
(gdb) b main
Breakpoint 1 at 0x8048060
(gdb) disas main
Dump of assembler code for function main:
   0x08048060 <+0>:   push   0x10
   0x08048062 <+2>:   push   0x12345678
   0x08048067 <+7>:   pop    eax
   0x08048068 <+8>:   pop    ebx
End of assembler dump.
```

# 어셈블리 언어 실습

## 레지스터 정보 확인#2

- **다시 프로그램 실행**

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/stud/x86_exam/exam1/a.out

Breakpoint 1, 0x08048060 in main ()
```

  - Breakpoint가 설정됨을 알 수 있음

- **다시 main함수를 disassemble**

- **다시 레지스터 정보 확인**
  - (gdb) info reg

**[a.out]**

```
pwd: x86_exam/exam1/a.out
(gdb) r
 //brekapoint를 main함수에 설정한 후 프로그램 실행

(gdb) disas main
```

```
Dump of assembler code for function main:
=> 0x08048060 <+0>:   push   0x10
   0x08048062 <+2>:   push   0x12345678
   0x08048067 <+7>:   pop    eax
   0x08048068 <+8>:   pop    ebx
End of assembler dump.
```

```
(gdb) info reg
```

```
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbffff0a0        0xbffff0a0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048060         0x8048060
```

# 어셈블리 언어 실습

## 특정 레지스터 값 확인

- **특정 레지스터 값만 확인 가능**
  - (gdb) info reg $eax
    - eax레지스터 값만 확인
  - (gdb) info reg $esp
    - esp레지스터 값만 확인
  - (gdb) info reg $eip
    - eip레지스터 값만 확인

**[a.out]**

```
pwd: x86_exam/exam1/a.out
(gdb) info reg $eax
```

```
(gdb) info reg $eax
eax            0x0    0
(gdb) info reg $esp
esp            0xbffff0a0        0xbffff0a0
(gdb) info reg $eip
eip            0x8048060         0x8048060
```