시스템프로그래밍 레포트

GDB를 활용하여 쉘 실행하기

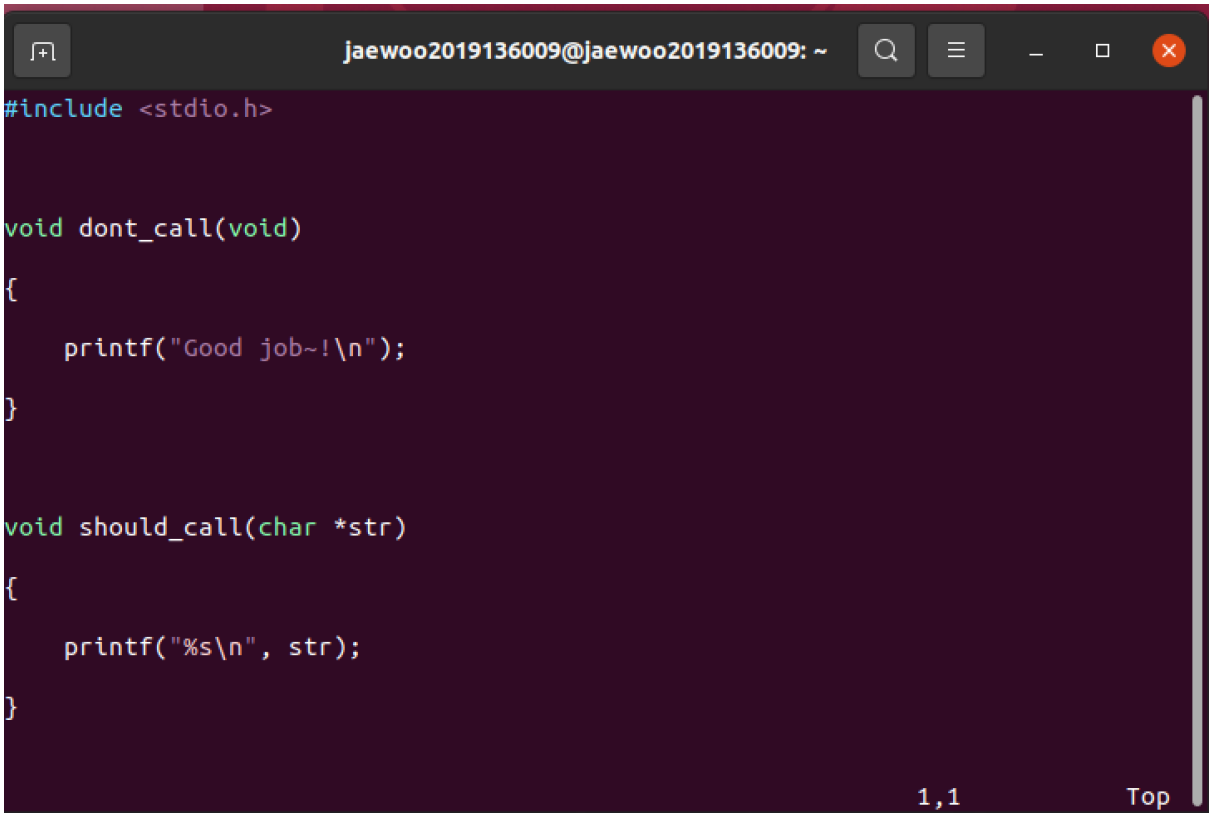
이름	곽재우
학번	2019136009
분반	01
작성일자	2023/09/13

1. GDB란 무엇일까

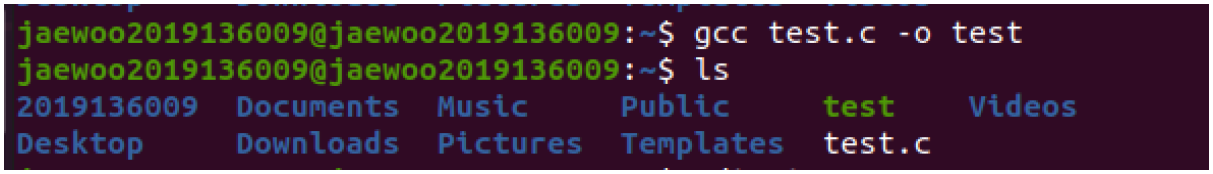
GDB는 GNU 소프트웨어 시스템을 위한 기본 **디버거**라고 한다. 디버거의 목적은 다른 프로그램 수행 중에 그 프로그램 내부에서 무슨 일이 일어나고 있는지 혹은 프로그램이 잘못 실행되었을 때 무슨 일이 일어나고 있는지 보여주는 것이다. GDB는 C, C++, 등으로 짠 프로그램을 디버그 할 수 있다.

2. gcc 컴파일러로 컴파일 해 보기

GDB로 파일을 디버그 하려면 먼저 컴파일을 해야한다. 컴파일은 gcc 컴파일러로 수행할 수 있다.
먼저 vi 편집기로 test.c 파일을 작성하자.



그리고 gcc 컴파일러를 이용하여 컴파일한다. 이때 -o 옵션은 출력 파일의 이름을 제공한다는 의미로, 컴파일 된 프로그램의 이름을 지정하여 새로운 파일로 저장할 수 있다.
아래에서는 test.c 파일을 컴파일한 결과를 test 파일로 정의하는 것이다.



그러고 나서 아래와 같이 컴파일된 파일을 실행할 수 있다.

```
jaewoo2019136009@jaewoo2019136009:~$ ./test
no way
```

추가로 gcc 명령에는 -g 옵션도 있는데, GCC 컴파일러가 생성한 실행 파일에 디버깅 정보를 포함시키는 데 사용된다.

3. GDB를 활용하여 쉘 실행하기

이번 과제는 GDB를 활용하여 쉘을 실행하는 것이다. 기존 과제를 수행했던 환경은 UTM 가상머신에서 우분투를 사용하였으나, 강의에서 사용하는 교안 환경과 조금 차이가 있는 것을 발견했다. 따라서 이번 과제부터는 AWS에 우분투 서버를 올린 뒤, 로컬 환경에서 SSH 접속하여 수행할 것이다.

나의 최초 터미널 환경은 다음과 같다. 우분투 서버에 SSH 접속한 모습이다.

```
Last login: Wed Sep 13 07:36:09 2023 from 118.235.24.69
ubuntu@ip-[redacted]~$
```

cf) 터미널 환경에서 SSH 접속하는 명령은 아래와 같다.
- ssh -i./rcamp.pem ubuntu@[AWS 탄력적 IP 주소]

이제부터 위의 우분투 환경에서 과제를 수행할 것이다. 주어진 과제는 아래 코드를 컴파일한 실행 파일이 GDB 상에서 쉘을 실행하도록 실행 흐름을 바꿔보는 것이다. 코드는 아래와 같다.

```c
#include <stdio.h>

void dont_call(void)
{
    printf("Good job~!₩n");
}

void should_call(char *str)
{
    printf("%s₩n", str);
```

```
}

int main(int argc, char **argv)
{
    void (*func)(char *);

    func = should_call;
    func("no way₩n");

    return 0;
}
```

GDB에서 디버깅하기 위해서 아래와 같이 파일을 생성한다.

```
ubuntu@ip-172-31-43-168:~$ vi debug.c
```

debug.c 파일에는 위의 코드를 작성해주고 gcc 컴파일러로 컴파일한다.

```
ubuntu@ip-172-31-43-168:~$ gcc -g -o debug debug.c
ubuntu@ip-172-31-43-168:~$ ls
debug  debug.c
```

이제 GDB로 디버깅할 차례이다.

```
ubuntu@ip-172-31-43-168:~$ gdb ./debug
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./debug...
(gdb)
```

이제 main 함수에 브레이크를 걸고 실행해보자.

```
(gdb) b main
Breakpoint 1 at 0x1195: file debug.c, line 17.
(gdb) r
Starting program: /home/ubuntu/debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffe4c8) at debug.c:17
17              func=should_call;
```

현재 main 함수의 흐름을 보기 위해 아래와 같은 명령어를 사용한다.

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000555555555182 <+0>:     endbr64
   0x0000555555555186 <+4>:     push   %rbp
   0x0000555555555187 <+5>:     mov    %rsp,%rbp
   0x000055555555518a <+8>:     sub    $0x20,%rsp
   0x000055555555518e <+12>:    mov    %edi,-0x14(%rbp)
   0x0000555555555191 <+15>:    mov    %rsi,-0x20(%rbp)
   0x0000555555555195 <+19>:    lea    -0x39(%rip),%rax        # 0x555555555163
<should_call>
   0x000055555555519c <+26>:    mov    %rax,-0x8(%rbp)
=> 0x00005555555551a0 <+30>:    mov    -0x8(%rbp),%rax
   0x00005555555551a4 <+34>:    lea    0xe63(%rip),%rdx        # 0x55555555600e
   0x00005555555551ab <+41>:    mov    %rdx,%rdi
   0x00005555555551ae <+44>:    call   *%rax
   0x00005555555551b0 <+46>:    mov    $0x0,%eax
   0x00005555555551b5 <+51>:    leave
   0x00005555555551b6 <+52>:    ret
End of assembler dump.
```

기존 코드에서 func = should_call이 이미 실행된 상태이니, func에 system 함수를 다시 할당해주자.

```
(gdb) set func=system
```

이제 func는 system 함수를 가리키고 있으므로 func의 호출은 곧 system 함수의 호출이다. 이제 우리가 해야할 것은, 실제 코드의 func("no way₩n")의 인자인 "no way₩n"를 "/bin/sh"로 바꾸는 것이다. 앞서 말했듯이 func를 호출하면 system이 호출되어 system("no way₩n")인 상태이기 때문이다. 그렇다면 인자로 전달된 문자열은 어떻게 바꿀까?

인자로 전달된 문자열을 바꾸려면 해당 문자열이 저장된 레지스터의 값을 바꾸면 된다. 그렇게 하기 위해 함수가 호출되는 부분 전으로 가서 함수의 인자로 들어가는 레지스터 값을 "/bin/sh"로 바꿔치기 해주면 된다. 먼저 함수가 호출되는 부분이 어디인지 알아야

한다. 다시 disas main 명령어로 main 함수의 흐름을 잘 보자.

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000555555555182 <+0>:     endbr64
   0x0000555555555186 <+4>:     push   %rbp
   0x0000555555555187 <+5>:     mov    %rsp,%rbp
   0x000055555555518a <+8>:     sub    $0x20,%rsp
   0x000055555555518e <+12>:    mov    %edi,-0x14(%rbp)
   0x0000555555555191 <+15>:    mov    %rsi,-0x20(%rbp)
   0x0000555555555195 <+19>:    lea    -0x39(%rip),%rax        # 0x555555555163
<should_call>
   0x000055555555519c <+26>:    mov    %rax,-0x8(%rbp)
=> 0x00005555555551a0 <+30>:    mov    -0x8(%rbp),%rax
   0x00005555555551a4 <+34>:    lea    0xe63(%rip),%rdx        # 0x55555555600e
   0x00005555555551ab <+41>:    mov    %rdx,%rdi
   0x00005555555551ae <+44>:    call   *%rax
   0x00005555555551b0 <+46>:    mov    $0x0,%eax
   0x00005555555551b5 <+51>:    leave
   0x00005555555551b6 <+52>:    ret
End of assembler dump.
```

현재 system 함수를 호출하기 전 상태이다. 어셈블리어를 조사해본 결과 call이 함수를 호출하는 어셈블리어인 것을 확인할 수 있었다. 따라서 call 부분이 system 함수를 호출하는 부분이니 해당 라인<+44>에 브레이크를 걸고 레지스터 값을 조작해보자. 아래와 같이 브레이크를 걸어준다.

```
(gdb) b *main+44
Breakpoint 2 at 0x5555555551ae: file debug.c, line 18.
```

이제 func("no way₩n") 부분에 브레이크 포인트가 생겼다. 그리고 실행해보면 해당 포인트 전까지 실행될 것이다. 실제로 n 명령어로 실행하고 main 함수의 흐름을 보면 다음과 같다.

```
(gdb) n

Breakpoint 2, 0x00005555555551ae in main (argc=1, argv=0x7fffffffe4c8) at debug.
c:18
18              func("no way\n");
```

< main+44 라인에 브레이크 포인트를 걸고 실행한 모습>

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000555555555182 <+0>:     endbr64
   0x0000555555555186 <+4>:     push   %rbp
   0x0000555555555187 <+5>:     mov    %rsp,%rbp
   0x000055555555518a <+8>:     sub    $0x20,%rsp
   0x000055555555518e <+12>:    mov    %edi,-0x14(%rbp)
   0x0000555555555191 <+15>:    mov    %rsi,-0x20(%rbp)
   0x0000555555555195 <+19>:    lea    -0x39(%rip),%rax        # 0x555555555163
<should_call>
   0x000055555555519c <+26>:    mov    %rax,-0x8(%rbp)
   0x00005555555551a0 <+30>:    mov    -0x8(%rbp),%rax
   0x00005555555551a4 <+34>:    lea    0xe63(%rip),%rdx        # 0x55555555600e
   0x00005555555551ab <+41>:    mov    %rdx,%rdi
=> 0x00005555555551ae <+44>:    call   *%rax
   0x00005555555551b0 <+46>:    mov    $0x0,%eax
   0x00005555555551b5 <+51>:    leave
   0x00005555555551b6 <+52>:    ret
End of assembler dump.
```

이제 system 함수를 호출하기 직전에 이르렀다. 해당 시점에서 레지스터의 값을 조사할 필요가 있다. 먼저 $rax는 함수를 호출하는 부분이니 당연히 func에 해당하는 함수가 들어가 있을 것이다.

```
(gdb) x/s $rax
0x555555555163 <should_call>:   "\363\017\036\372UH\211\345H\203\354\020H\211}\3
70H\213E\370H\211\307\350\321\376\377\377\220\311\303\363\017\036\372UH\211\345H
\203\354 \211}\354H\211u\340H\215\005\307\377\377\377H\211E\370H\213E\370H\215\0
25c\016"
```

system 함수가 출력될 것이라는 예상과는 다르게 should_call이 출력된다. 그러나 $rax 레지스터가 함수 주소값을 담고 있다는 것은 확실하다. 그러니 한칸 위의 mov 어셈블리어가 인자로 받는 두 레지스터 $rdx, $rdi를 조사해보자.

```
(gdb) x/s $rdx
0x55555555600e: "no way\n"
(gdb) x/s $rdi
0x55555555600e: "no way\n"
```

놀랍게도 두 레지스터에 모두 "no way₩n"가 들어가 있다. 내 생각으로는 해당 라인은 이미 실행되었기 때문에 mov 어셈블리 명령의 특성 상 2번째 값을 1번째 값으로 옮기기에 $rdi에 있던 "no way₩n"가 $rdx에도 있지 않나 싶다. 이 부분은 이해가 잘 되지 않아 두 개의 방법을 모두 시도해보았다. 첫 번째는 $rdx에 "/bin/sh"를 넣는 것이고, 두 번째는 $rdi에 "/bin/sh"를 넣는 것이다. 결과적으로 두 번째 시도인 $rdi에 "/bin/sh"를 넣었을 때만 쉘을 실행할 수 있었다. 만약 $rdx에 "/bin/sh"를 넣은 경우, 아래와 같이 쉘이

제대로 실행되지 않았다.

<$rdx 레지스터 값을 바꾼 모습>

```
(gdb) set $rdx="/bin/sh"
(gdb) n
[Detaching after vfork from child process 32933]
sh: 1: no: not found
20              return 0;
(gdb) Quit
(gdb) q
A debugging session is active.

        Inferior 1 [process 32929] will be killed.

Quit anyway? (y or n) y
```

<$rdi 레지스터 값을 바꾼 모습>

```
(gdb) x/s $rdi
0x55555555600e: "no way\n"
(gdb) set $rdi="/bin/sh"
(gdb) n
[Detaching after vfork from child process 32974]
$
$
$
$ ls
debug  debug.c
$
```

$rdi 레지스터에 저장된 문자열을 "/bin/sh"로 바꾸니 결과적으로 system("/bin/sh")가 호출되어 GDB상에서 쉘이 실행되는 모습이다. 쉘에서 ls 명령어로 현재 파일들을 찍어보면 잘 출력되는 것을 확인할 수 있다.

4. 고찰

이번 과제는 정말 어려웠다. GDB라는 디버거가 익숙치 않을 뿐더러 레지스터 값을 조작하는 것도 쉽지 않았다. 아쉬운 것은 b *main+44 부분에 브레이크를 걸고나서 $rdx, $rdi 두 레지스터에 모두 문자열이 들어있는데 $rdi의 값을 바꾸었을 때만 쉘이 정상적으로 실행되는 것이 잘 이해가 되지 않는 것이다. 완벽하게 이해하는 것은 다른 배경지식도 있어야하므로 역시나 쉽지 않은 일이다. 그러나 전반적으로 디버거로 값을 어떻게 조작하는지 감을 잡았고 GDB 상에서 어떻게 쉘이 실행되는지 알 수 있었다. 또한 이렇게 시스템 레벨에서 레지스터 값을 조작해보고 디버거를 딥하게 사용해볼 기회는 흔치 않다. 이렇게 고통을 겪으면서 또 한 층 성장한 기분이 든다.

Reference

https://kookiencream.tistory.com/96
https://cnu-cse-pgs.tistory.com/8
https://terrorjang.tistory.com/entry/gdb-%EA%B8%B0%EC%B4%88-%EC%82%AC%EC%9A%A9%EB%B2%952stepi-nexti-%EC%B0%A8%EC%9D%B4%EC%A0%90
https://coding-wonderland.tistory.com/19
https://woo-dev.tistory.com/44
https://sosal.kr/128