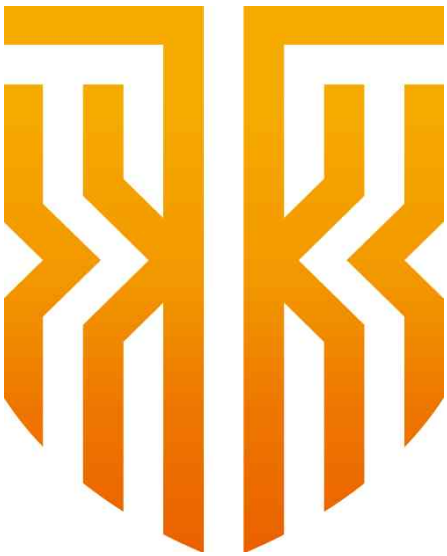
과제 2
GDB를 활용하여 쉘 실행하기



분 반	1 분 반
과 목 명	시스템프로그래밍
교 수	이 원
학번 / 이름	2018136121 / 조 원 석
제 출 일	2023.09.14.

목차

1. 개괄

가. GDB 정의[1)]

GNU 프로젝트의 디버거로, 다른 프로그램이 실행되는 동안 다른 프로그램 **'내부'**에서 무슨 일이 일어나고 있는지, 혹은 다른 프로그램이 **충돌**할 때 무엇을 하고 있었는지 보여주는 기능을 수행한다.

나. GDB 명령어 정리[2)3)]

- 기본 설정

layout asm : 어셈블리 정보창 출력
layout reg : 레지스터 정보창 출력
set disassembly-flavor intel : 인텔 아키텍쳐로 설정
info registers : 레지스터 정보 확인
ex) 레지스터 종류 설명
rax (eax) : **누산기(accumulator)** 레지스터. **산술연산(덧셈, 나눗셈, 곱셈)이나 논리연산을 수행한 반환값**이 저장
rbx (ebx) : **베이스** 레지스터
rcx (ecx) : **카운터** 레지스터. **반복** 명령어 사용 시 반복 카운터로 사용되는 값을 저장
rdx (edx) : **데이터** 레지스터. 산술연산과 I/O 명령에서 **rax(eax)와 함께** 사용
rsi (esi) : **source 인덱스** 레지스터
rdi (edi) : **destination 인덱스** 레지스터
rbp (ebp) : **베이스 포인터** 레지스터. **스택의 시작 지점 주소**를 저장
rsp (esp) : **스택 포인터** 레지스터. **스택의 가장 마지막 지점 주소**를 저장
rip : **명령 포인터** 레지스터이다. **현재 명령의 위치**를 가리킴

- 프로그램 실행

r <명령인자> : 실행(run)
c : 중단된 프로그램 실행 재개(continue)
kill : 프로그램 실행 종료
s <숫자> : 몇 단계 씩 수행(step
ni, si : 한 단계씩 진행
finish : 현재 함수의 리턴까지 실행
return <return 값> : 함수를 실행하지 않고 리턴
q : 디버깅 종료

1) https://www.sourceware.org/gdb/
2) https://movefast.tistory.com/102
3) https://chelseafandev.github.io/2023/02/10/tip-for-gdb-debugging/

- 중단점(breakpoint)

b <함수이름/라인넘버/*메모리주소> : breakpoint 설정

-> **b *main+43** : main 문자열(함수이름참조)이 참조하는 주소 +43

-> **break 8048a8**

i b : 설정된 breakpoint 정리

delete <breakpoint 번호> : 설정된 breakpoint 삭제

enable <breakpoint 번호> : breakpoint 활성

disable <breakpoint 번호> : breakpoint 비활성

clear <함수명> : 함수 내의 모든 breakpoint 삭제

- 메모리 확인

display[/표시형식] <변수명> : 변수의 값이 유효한 코드 부분에서 값을 계속해서 출력(표시형식 : d, x, t)

undisplay : display 모드 종료

print(혹은 **p**) **[/표시형식]** <변수명or*메모리주소or레지스터이름)

: 현재 또는 지정된 위치의 프로그램 **소스 코드** 출력

x/[개수][표시형식][단위크기] <주소/함수명> : 지정된 위치의 메모리 데이터를 지정된 형식으로 출력

-> 표시형식 : d, o, x, t, f, s, I 등

-> 단위 크기 : b, h, w, g

ex) x/x <주소> : 해당 주소의 내용물을 바이트 단위로 출력

-> x/10x : opcode 10개 출력

-> x/10i : 명령어 10개 출력

-> x/10s : 문자열 10개 출력

-> x/10wx : word단위로 10개 출력

-> x/s, x/i, ... etc

ex) mov $eax, [0xbfffefc4] : 대괄호는 포인터를 의미

-> 대괄호 없이 0xbfffefc4일 경우 그 주소의 내용

-> 포인터 : 그 주소에 저장된 **포인터주소가 참조하는 주소의 내용물을 저장**하는 것.

-> '*'기호 사용 : x/s *0xbfffefc4 처럼 한단계 더 보아야 함

ex) 레지스터도 포인터처럼 사용해서 내용물을 볼 수 있다.

-> x/s $eax : eax에 저장된 내용(주소일수도있음)

-> x/s ***$eax** : eax에 저장된 **포인터가 가리키는 내용**

- 프로그램 코드 확인

list <함수명/라인번호/변수명> : 현재 or 지정위치의 프로그램 소스코드 출력

disas <함수명/메모리주소> : 함수 디스어셈블리 내용 (기계어 코드)

- 변수값 변경 감지

watch <변수명> : 변수에 데이터 써질 때 실행 멈춤 (like breakpoint)
rwatch <변수명> : 변수 읽을 때 실행 멈춤
awatch <변수명> : 변수 관련 모든 경우에 실행 멈춤

- 기타 명령어

backtrace, where : 현재 실행 위치의 주소와 스택 상태 출력
i locals : 모든 지역 변수 목록과 현재 값 출력(= info locals)
i program : 프로그램 실행 정보 출력(= info program)
i reg : 현재 레지스터 정보 출력(= info reg)
set <수정할주소의 포인터> = <내용> : **메모리 데이터 변경**
ex) 잘못된 경우 : 레지스터 주소에 문자열 주소를 넣을 때
set $esp = 0x08048585 : **esp가 바뀜!!**
ex) 옳은 경우 : 레지스터가 참조하는 주소에 문자열 주소를 넣을 때
set *0xbfffefa0 = 0x08048585 레지스터는 포인터참조(*)를 사용할 수 없다!
즉, 레지스터의 주소를 적을 것!

다. system("/bin/sh")[4]

- system함수의 주소?

```
(gdb) b *main          // main 함수의 주소를 breakpoint
(gdb) r                // 파일 실행. 이 상태에서만 system 함수의 주소를 확인 가능
(gdb) p system         // system 함수의 주소 출력함.
```

- "/bin/sh"의 주소?

```
(gdb) find &system,+9999999999,"/bin/sh"      // find 함수를 이용해 찾을 수 있음
```

- 문제는 어떻게 풀었니?

① 퀴즈1을 활용하여 should_call 메서드를 system 메서드로 바꿔준다.
② 퀴즈2에서 요구하는 문자열 리터럴을 no way에서 /bin/sh로 바꿔준다.

```
// 퀴즈 1 활용. 메서드 복사
(gdb) set var func = system
// /binay가 출력됨.  4byte : no w <- /bin
(gdb) set *0x55555555600f =*0x7ffff7dd8698
// /bin/sh가 출력됨. 4byte : ay <- /sh
(gdb) set *(0x55555555600f+4) =*(0x7ffff7dd8698+4)
```

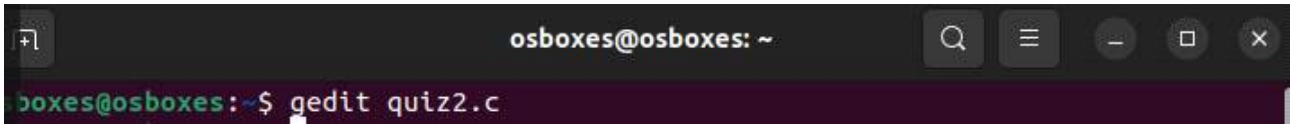
4) https://jade9reen.tistory.com/72

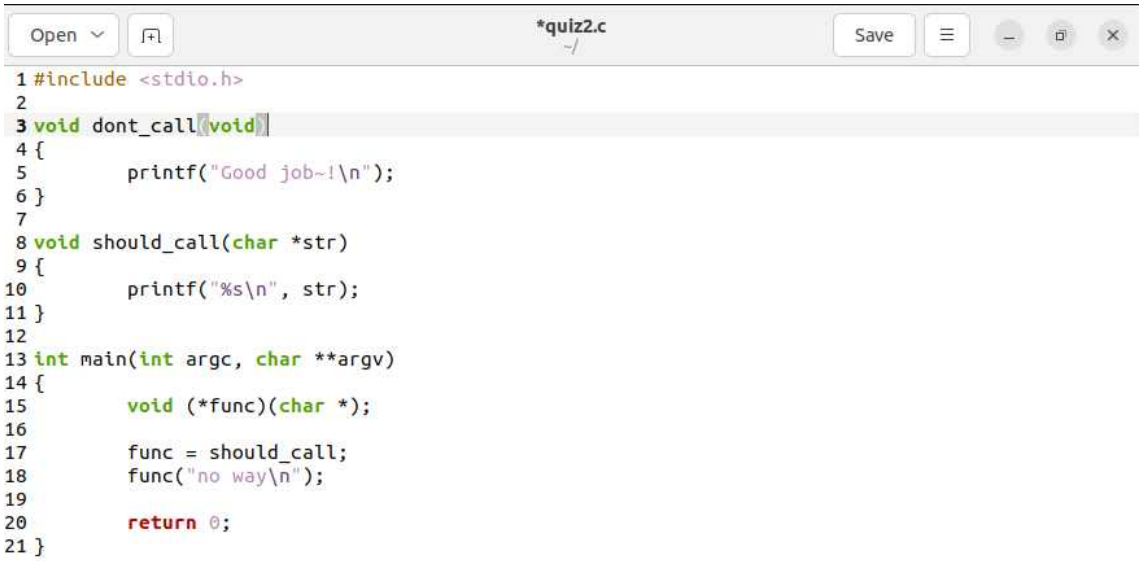
2. 리눅스에 GDB 설정하기

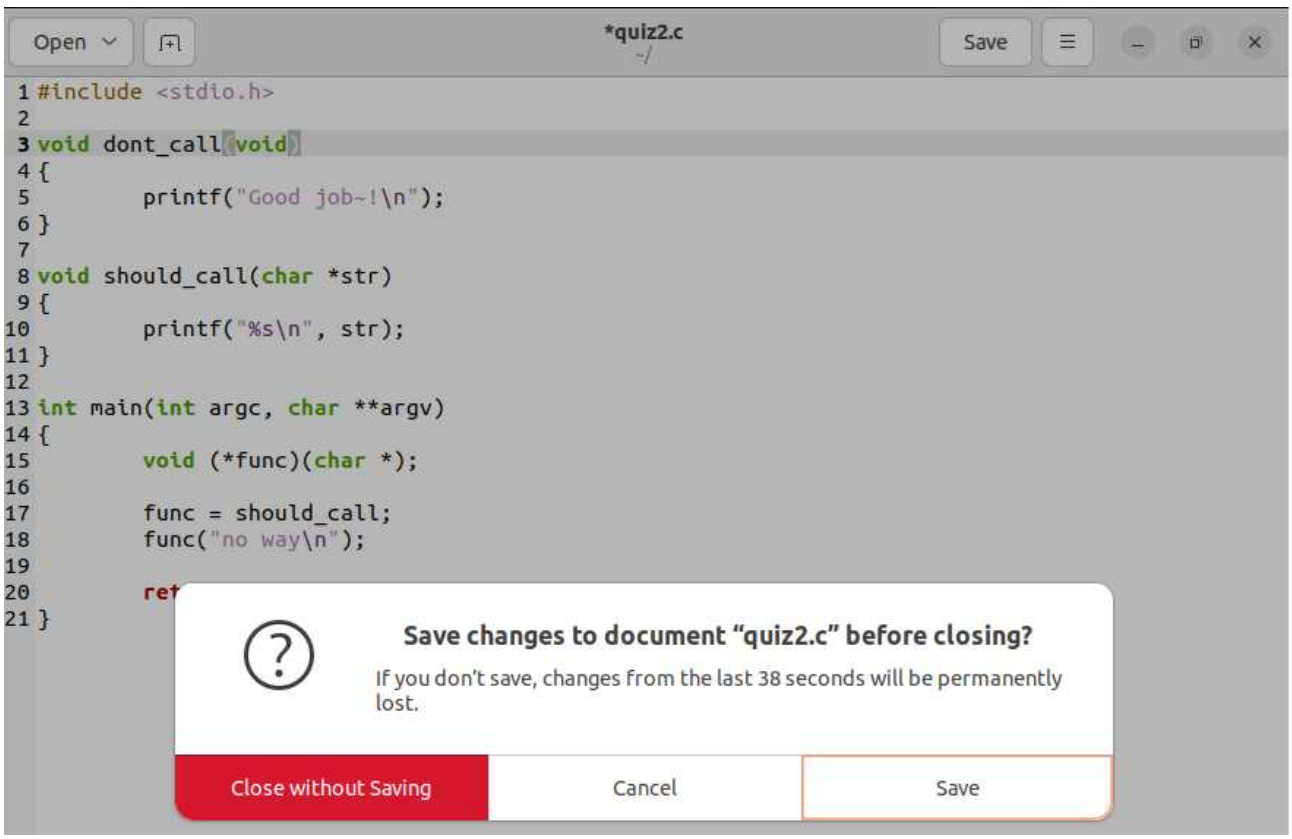
가. 개괄

- 소스 코드 입력 후 저장한다.

gedit [소스코드명]







- gcc 컴파일러를 이용해 실행 파일을 만든 후 gdb로 실행한다.

gcc [소스코드명] -o [실행파일명] -g // gcc로 실행파일 만들기

gdb ./[실행파일명] // gdb로 디버깅하기

```
osboxes@osboxes:~$ gcc quiz2.c -o quiz2 -g
osboxes@osboxes:~$ gdb ./guiz2
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"...
./guiz2: No such file or directory.
(gdb)
```

- disas를 통해 main 함수의 디스어셈블리 단위의 주소 알아내기

disas [함수명/메모리주소] // 함수 디스어셈블리 내용 (기계어 코드)

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000001182 <+0>:     endbr64
   0x0000000000001186 <+4>:     push   %rbp
   0x0000000000001187 <+5>:     mov    %rsp,%rbp
   0x000000000000118a <+8>:     sub    $0x20,%rsp
   0x000000000000118e <+12>:    mov    %edi,-0x14(%rbp)
   0x0000000000001191 <+15>:    mov    %rsi,-0x20(%rbp)
   0x0000000000001195 <+19>:    lea    -0x39(%rip),%rax        # 0x1163 <should call>
   0x000000000000119c <+26>:    mov    %rax,-0x8(%rbp)
   0x00000000000011a0 <+30>:    mov    -0x8(%rbp),%rax
   0x00000000000011a4 <+34>:    lea    0xe64(%rip),%rdx        # 0x200f
   0x00000000000011ab <+41>:    mov    %rdx,%rdi
   0x00000000000011ae <+44>:    call   *%rax
   0x00000000000011b0 <+46>:    mov    $0x0,%eax
   0x00000000000011b5 <+51>:    leave
   0x00000000000011b6 <+52>:    ret
End of assembler dump.
(gdb) b *main
Breakpoint 1 at 0x1182: file quiz2.c, line 14.
```

- breakpoint를 이용해 main에 중단점을 설정한다.

b [함수이름/라인넘버/*메모리주소] // breakpoint 설정

ex) **b *main** : main 문자열(함수이름참조)이 참조하는 주소

```
(gdb) b *main
Breakpoint 1 at 0x1182: file quiz2.c, line 14.
```

- r로 breakpoint로 잡은 주소부터 실행하여 n으로 넘기며 should_call의 주소를 찾는다.

```
(gdb) r
Starting program: /home/osboxes/quiz2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=0, argv=0x0) at quiz2.c:14
14      {
(gdb) n
17              func = should_call;
(gdb) n
18              func("no way\n");
```

나. 퀴즈 1. system 함수 설정하기

- disas를 통해 should call의 주소를 다음과 같이 **0x555555555163**이라는 것을 알아낼 수 있었다. func에서 다음이 가리키는 주소를 이제 system 함수의 주소를 가리키게 한다.

```
(gdb) disas *main+34
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    0xe64(%rip),%rdx        # 0x55555555600f
   0x00005555555551ab <+41>:    mov    %rdx,%rdi
   0x00005555555551ae <+44>:    call   *%rax
   0x00005555555551b0 <+46>:    mov    $0x0,%eax
   0x00005555555551b5 <+51>:    leave
   0x00005555555551b6 <+52>:    ret
End of assembler dump.
```

- p system을 통해 system 함수의 주소를 알아내고 퀴즈1에서 set을 이용해 should_call을 dont_call로 바꾸었던 과정처럼 system함수를 대입해준다.

print(혹은 p) [/표시형식] <변수명or*메모리주소or레지스터이름)

: 현재 또는 지정된 위치의 프로그램의 **소스 코드** 출력

set <수정할주소의 포인터> = <내용> : **메모리 데이터 변경**

```
(gdb) p system
$1 = {int (const char *)} 0x7ffff7c50d60 <__libc_system>
(gdb) find &system,+9999999999,"/bin/sh"
0x7ffff7dd8698
warning: Unable to access 16000 bytes of target memory at 0x7ffff7e268a0, halting search.
1 pattern found.
(gdb) set var func = system
```

- x/i func으로 확인해보면 func의 메모리 데이터가 should_func이 아닌 system으로 갱신되었음을 확인할 수 있다.

x/[개수][표시형식][단위크기] [주소/함수명]

: 지정된 위치의 메모리 데이터를 지정된 형식으로 출력

```
(gdb) x/i func
   0x7ffff7c50d60 <__libc_system>:      endbr64
```

다. 퀴즈 2. 문자열 리터럴 처리

- disas를 통해 0x200f였던 문자열 리터럴의 주소가 0x55555555600f로 나타난 것을 확인할 수 있다. 이제 이 주소에 "/bin/sh"를 할당하려고 한다.

```
(gdb) disas *main+34
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    0xe64(%rip),%rdx        # 0x55555555600f
   0x00005555555551ab <+41>:    mov    %rdx,%rdi
   0x00005555555551ae <+44>:    call   *%rax
   0x00005555555551b0 <+46>:    mov    $0x0,%eax
   0x00005555555551b5 <+51>:    leave
   0x00005555555551b6 <+52>:    ret
End of assembler dump.
```

- find &system,+9999999999,"/bin/sh"를 이용해 system이 참조하고 있는 부분중에 "/bin/sh"를 갖고 있는 곳의 주소를 알아낸다.
0x7ffff7dd8698임을 알 수 있다.

```
(gdb) find &system,+9999999999,"/bin/sh"
0x7ffff7dd8698
warning: Unable to access 16000 bytes of target memory at 0x7ffff7e268a0, halting search.
1 pattern found.
```

- 문자열 리터럴 "no way"를 가리키고 있는 *0x55555555600f에 *0x7ffff7dd8698을 대입한다. 다만, **다음과 같이 할당할 경우 4byte씩 할당을 진행하므로, "no way"와 "/bin/sh"의 크기에 미치지 못하기에 포인터의 주소를 4byte 옮겨서 같은 과정을 한 번 더 진행**해준다.

```
(gdb) set *0x55555555600f = *0x7ffff7dd8698
(gdb) set *(0x55555555600f+4) = *(0x7ffff7dd8698+4)
```

- 최종적으로 c를 통해 기존 코드를 이어서 진행해주면 /bin/sh이 실행됨을 확인할 수 있다. ps를 통해 bash쉘이 실행됨을 확인할 수 있다. 여기서 exit를 두 번 사용해서 탈출하는데 한 번은 /bin/sh를 위해 exit, 두 번째 exit는 gdb를 나가기 위해 사용함을 알 수 있다.

```
(gdb) c
Continuing.
[Detaching after vfork from child process 199596]
$ ps
    PID TTY          TIME CMD
   9553 pts/0    00:00:00 bash
 198217 pts/0    00:00:00 gdb
 198420 pts/0    00:00:00 quiz2
 199596 pts/0    00:00:00 sh
 199597 pts/0    00:00:00 sh
 199640 pts/0    00:00:00 ps
$ exit
[Inferior 1 (process 198420) exited normally]
(gdb) exit
osboxes@osboxes:~$ cd /bin/sh
```

- 여기서 /bin/sh을 통해 한 번 더 확인해보아도 알 수 있다.

```
osboxes@osboxes:~$ /bin/sh
$ ps
    PID TTY          TIME CMD
   9553 pts/0    00:00:00 bash
 199936 pts/0    00:00:00 sh
 199982 pts/0    00:00:00 ps
$ exit
```

- gdb에서 /bin/sh를 실행했을 때는 gdb에서 guiz2를 통해 /bin/sh를 실행하였기에 다음과 같이 ps의 PID와 CMD가 다음과 같이 나타나게 됨을 알 수 있다.

```
$ ps
    PID TTY          TIME CMD
   9553 pts/0    00:00:00 bash
 198217 pts/0    00:00:00 gdb
 198420 pts/0    00:00:00 quiz2
 199596 pts/0    00:00:00 sh
 199597 pts/0    00:00:00 sh
 199640 pts/0    00:00:00 ps
```

3. 발생한 이슈 정리

가. p system 이슈

```
(gdb) disas *main
Dump of assembler code for function main:
   0x0000000000001182 <+0>:     endbr64
   0x0000000000001186 <+4>:     push   %rbp
   0x0000000000001187 <+5>:     mov    %rsp,%rbp
   0x000000000000118a <+8>:     sub    $0x20,%rsp
   0x000000000000118e <+12>:    mov    %edi,-0x14(%rbp)
   0x0000000000001191 <+15>:    mov    %rsi,-0x20(%rbp)
   0x0000000000001195 <+19>:    lea    -0x39(%rip),%rax        # 0x1163 <should_call>
   0x000000000000119c <+26>:    mov    %rax,-0x8(%rbp)
   0x00000000000011a0 <+30>:    mov    -0x8(%rbp),%rax
   0x00000000000011a4 <+34>:    lea    0xe64(%rip),%rdx        # 0x200f
   0x00000000000011ab <+41>:    mov    %rdx,%rdi
   0x00000000000011ae <+44>:    call   *%rax
   0x00000000000011b0 <+46>:    mov    $0x0,%eax
   0x00000000000011b5 <+51>:    leave
   0x00000000000011b6 <+52>:    ret
End of assembler dump.
(gdb) p system
No symbol "system" in current context.
(gdb) p should_call
$1 = {void (char *)} 0x1163 <should_call>
(gdb) p dont_call
$2 = {void (void)} 0x1149 <dont_call>
```

- 위 사진 첫 부분에서 r로 실행하기 전에 p system을 해주었을 때, 현재 context에서 system 함수가 존재하지 않아 찾을 수 없다는 오류를 직면하게 되었다.

```
(gdb) r
Starting program: /home/osboxes/quiz2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
no way

[Inferior 1 (process 42608) exited normally]
```

- 실행하고 난 뒤 disas를 해주었을 때 아래와 같이 함수들에서 기존에 보이지 않던 주소 정보를 확인할 수 있었다.

```
(gdb) disas *main+34
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    0xe64(%rip),%rdx        # 0x55555555600f
   0x00005555555551ab <+41>:    mov    %rdx,%rdi
   0x00005555555551ae <+44>:    call   *%rax
   0x00005555555551b0 <+46>:    mov    $0x0,%eax
   0x00005555555551b5 <+51>:    leave
   0x00005555555551b6 <+52>:    ret
End of assembler dump.
```

- 최종적으로는 p system을 통해 system 함수의 주소를 구할 수 있었다.

```
(gdb) p system
$4 = {int (const char *)} 0x7ffff7c50d60 <__libc_system>
(gdb)
```

나. /binay 출력 및 함수 할당 이슈

앞서 '1. 다'에서 아래와 같은 문장으로 위 이슈에 대해 단순하게 다루어 주었다.

① 퀴즈1을 활용하여 should_call 메서드를 system 메서드로 바꿔준다.

② 퀴즈2에서 요구하는 문자열 리터럴을 no way에서 /bin/sh로 바꿔준다.

// 퀴즈 1 활용. 메서드 복사

(gdb) set var func = system

// /binay가 출력됨. 4byte : no w <- /bin

(gdb) set *0x55555555600f =*0x7ffff7dd8698

// /bin/sh가 출력됨. 4byte : ay <- /sh

(gdb) set *(0x55555555600f+4) =*(0x7ffff7dd8698+4)

```
Breakpoint 1, 0x000055555555551a4 in main (argc=1, argv=0x7fffffffe188) at quiz2.c:18
18              func("no way\n");
(gdb) set func = system
(gdb) set *0x55555555600f = *0x7ffff7dd8698
(gdb) c
Continuing.
/binay
```

처음에 system을 할당할 때는 var을 쓰지 않은채로 **set func = system**을 해주었고,
문자열 리터럴에 주소를 할당할 때도 4byte씩만 할당해주는 것을 알지 못하여 계속해서 /binay만을 출력시키는 이슈에 직면하였다.

두 문제는 각각
set func = system을 set **var** func = system으로 바꿔주고
set *(0x55555555600f+4) =*(0x7ffff7dd8698+4)을 추가해줌으로써 해결할 수 있었다

다. find "/bin/sh" 이슈

- system에서 "/bin/sh"가 저장된 곳의 주소를 알고 싶었지만 프로세스의 디버그 없이는 find를 실행시킬 수 없다는 오류 문구를 접했다.

```
(gdb) find "/bin/sh"
You can't do that without a process to debug.
(gdb)
```

- 그러다가 한 블로그를 통해 내가 한 방식이 틀렸음을 알게 되었다.
그냥 find "/bin/sh"가 아닌 find &system,+9999999999,"/bin/sh"과 같이 system 함수의 주소를 참조하고 +999999999와 같은 정규 표현식을 이용해 "/bin/sh"을 검색해야 주소를 찾을 수 있는 것이었다.

- 결국 아래 사진과 같은 방식을 사용하여 system에서 "/bin/sh"가 저장된 주소를 알 수 있었다.

```
(gdb) find &system,+9999999999,"/bin/sh"
0x7ffff7dd8698
warning: Unable to access 16000 bytes of target memory at 0x7ffff7e268a0, halting search.
1 pattern found.
```

4. 느낀점

OpenSSH같은 경우는 sftp 개념을 접한 상태였었기에 이해하기 쉬웠지만, gdb는 이름조차 생소했었기에 과제를 시작함에 처음에는 많은 부담이 있었습니다. 하지만 혼자 gdb에 대한 기초 개념을 찾아보고, 수업시간에 찍어두었던 퀴즈1의 과정들, 그리고 과제를 진행하면서 필요했던 자료들을 모아 합쳐서 하나하나 수행했던 것이 짧은 시간동안 gdb 자체를 실제로 사용하는데 도움을 받은 것 같아 좋았습니다. 1주차와 2주차 과제를 진행하면서 수행자의 역량에 따라 같은 결과를 나타냄에도 다른 과정을 보여줌을 확인할 수 있었는데 단순히 "무엇을" 구현했는지, 즉 과제의 결과에만 집중하는 것이 아닌 "어떻게" 해결했는지에 초점을 맞출 수 있었다고 생각하여, 비교적 과정이 어느 정도 정해져 있던 다른 수업들의 과제들보다 더 재밌었습니다. 앞으로도 과정을 통해 정진해갈 수 있도록 과제를 통해 최대한 열심히 배워가고 싶습니다. 감사합니다.