컴시개 Lab2 보고서

전공: 수학과 학년: 3학년 학번: 20191274 이름: 장유빈

1. problem1

```
sub     $0x28,%rsp
```
스택에 40바이트를 할당함.

```
mov     $0x402004,%edi
callq   0x401030 <puts@plt>
```
puts("Provide your input: "); 부분이다.

```
mov     %rsp,%rsi
mov     $0x402018,%edi
callq   0x401040 <__isoc99_scanf@plt>
```
scanf("%31s", buf); 부분이다. 이를 통해 %rsp 부분부터 buf 문자열을 의미하는 것을 알 수 있다. 이를 유의하자.

```
mov     $0x0,%eax
```
나는 'int temp1 = 0;'으로 이해했다.

```
mov     $0x0,%eax
```
마찬가지로, 나는 'int temp1 = 0;'으로 이해했다.

```
movslq %eax,%rdx
movzbl 0x404038(%rdx),%edx
```
이 어셈블리 코드를 진행하기 전 'x/1xb 0x404038'을 진행해보자.

```
) x/1xb 0x404038
4038 <msg>: 0x43
```
이 명령어를 통해 msg 문자열의 첫 문자에 대문자 'C'가 저장되어 있는 것을 확인할 수 있다. %rdx 레지스터를 temp2라는 임의의 변수에 대입해보면, 'temp2 = msg[temp1]'으로 이해할 수 있다.

```
test    %dl,%dl
je      0x401176 <main+64>
```
이는 msg[temp1]이 0임을 확인하는 구문으로, msg 문자열의 문자가 널 문자임을 체킹하고 있는 것이다. 현재는 널 문자가 아니지만 널 문자라고 가정해보고 <main+64>번지로 점프해보자.

```
<+64>:     mov     $0x1,%eax
```
temp1을 1로 설정하고 있다.

```
test    %eax,%eax
je      0x40119a <main+100>
```
temp1이 1이므로 <main+100>으로 점프하지 않을 것이다.

```
<+73>:    mov    $0x40201d,%edi
<+78>:    callq  0x401030 <puts@plt>
```
puts의 첫 번째 인자로 들어갈 0x40201d 부분을 확인해보자. 'x/s 0x40201d' 명령어를 써서 확인해보자.

```
"You passed the challenge!"
```
우리가 원하는 문자열이 인자로 들어감을 확인할 수 있다.

```
mov    $0x0,%eax
add    $0x28,%rsp
retq
```
'return 0;'을 하기 위한 준비를 하고 스택 영역을 반환하고 함수를 종료하는 구문이다.
자, 이제 다시 위로 올라가서, msg[temp1]이 널이 아닌 상황을 보자.

```
movslq %eax,%rcx
cmp    %dl,(%rsp,%rcx,1)
jne    0x401193 <main+93>
```
이 구문은 buf[temp1]과 msg[temp1]을 비교하는 구문이다.
같지 않다고 가정해보고 <main+93>으로 점프해보자.

```
<+93>:    mov    $0x0,%eax
<+98>:    jmp    0x40117b <main+69>
```
temp1을 0으로 설정하고 <main+69>로 점프해야한다.

```
test   %eax,%eax
je     0x40119a <main+100>
```
temp1이 0이므로 <main+100>으로 점프해야 한다.

```
<+100>:   mov    $0x402038,%edi
<+105>:   callq  0x401030 <puts@plt>
<+110>:   jmp    0x401189 <main+83>
```
'x/s 0x402038'을 명령어를 입력해보면,

```
) x/s 0x402038
2038:        "No, that is not the input I want!"
```
실패 명령어를 puts 함수의 인자로 전달하고 <main+83>으로 점프해서 함수를 종료하는 것을 확인할 수 있다. 자, 이제 위로 다시 올라가서 buf[temp1]과 msg[temp1]이 같다고 가정해보자.

```
add    $0x1,%eax
```
temp1에 1을 더해준다.

```
jmp    0x40116b <main+37>
```
temp1에 1을 더해주고 <main+37>로 점프한다.

```
<+37>:    movslq %eax,%rdx
<+40>:    movzbl 0x404038(%rdx),%edx
```
다시, 같은 상황이 반복이다.
따라서 msg[temp1]이 널 문자가 될 때까지 buf[temp1]이 msg[temp1]과 같아야 성공 문자열을 출력할 수 있다는 것을 알 수 있다. 이 말은 즉슨, 우리가 입력한 문자열이 msg에 저장되어있는 문자열과 같아야 함을 알 수 있다. 자, 이제 msg에 저장되어 있는 문자열을

‘x/1xb’ 명령어를 반복하여 사용하여 하나하나씩 확인해보자.

```
) x/1xb 0x404039
4039 <msg+1>:        0x53
```

위의 과정을 반복하여 문자열을 확인하면,
msg에 저장되어 있는 문자열은 “CSE3030@Sogang”임을 확인할 수 있고,
따라서 우리는 “CSE3030@Sogang”을 입력해야 하는 것을 유추할 수 있다.
이상이다.

```
# TODO: Complete this function (do not touch anything else).
def solve():
    prog = Program("./problem1.bin")
    print(prog.read_line()) # Read the initial message of the program.
    prog.send_line("CSE3030@Sogang") # Send your input to the program.
    print(prog.read_line()) # Read the response from the program.


if __name__ == "__main__":
    solve()
```

2. problem2

```
  sub    $0x18,%rsp
```

함수의 시작에 앞서 스택 영역에 24바이트를 할당하고 있다.

```
mov    $0x402004,%edi
callq  0x401030 <puts@plt>
```

puts(“Provide your input: ”); 부분이다.

```
lea    0x4(%rsp),%rcx
lea    0x8(%rsp),%rdx
lea    0xc(%rsp),%rsi
mov    $0x402018,%edi
callq  0x401040 <__isoc99_scanf@plt>
```

scanf(“%d %d %d”, &x, &y, &z); 부분이다. 이를 통해 0xc(%rsp)에는 x가, 0x8(%rsp)에는 y가, 0x4(%rsp)에는 z가 저장되어 있음을 알 수 있고 저는 이를 유용하게 이용할 것입니다.

```
  mov    $0x0,%eax
```

저는 ‘int temp1 = 0;’으로 이해했습니다.

```
mov    0xc(%rsp),%edx
cmp    $0x40,%edx
jle    0x4011be <main+136>
```

이 어셈블리 코드는 x가 60보다 작거나 같은지 체킹하고 있습니다.
x가 60보다 크다고 가정하고 점프하지 말아봅시다.

```
mov    $0x1,%ecx
```

저는 이 부분은 ‘int temp2 = 1;’로 이해했습니다.

```
cmp    $0x60,%edx
jle    0x40117a <main+68>
```

x가 96보다 작거나 같은지 묻고 있습니다. x가 96보다 작거나 같다고 가정하고 <main+68>로

점프해봅시다.

```
mov     0x4(%rsp),%eax
cmp     $0x200,%eax
jg      0x40118a <main+84>
```

z가 512보다 큰지 묻고 있습니다. z가 512보다 크다고 가정하고 <main+84>로 점프해봅시다.

```
 cmp     $0x230,%eax
 jle     0x401196 <main+96>
```

z가 560보다 작거나 같은지 묻고 있습니다. 그렇다고 가정하고 <main+96>으로 점프해봅시다.

```
mov     0x8(%rsp),%esi
mov     %esi,%edi
sub     %edx,%edi
sub     %esi,%eax
add     %eax,%eax
cmp     %eax,%edi
jne     0x4011aa <main+116>
```

y-x와 2*(z-y)가 같지 않은지 묻고 있습니다. 같다고 가정해봅시다. 즉, 3*y == x + 2*z라고 가정해봅시다.

```
 test    %ecx,%ecx
 jne     0x4011c5 <main+143>
```

아까 temp2를 1로 설정했으므로 <main+143>으로 점프할 것입니다.

```
mov     $0x402021,%edi
callq   0x401030 <puts@plt>
jmp     0x4011b4 <main+126>
```

'x/s 0x402021'로 puts의 인자로 들어갈 문자열을 체킹해봅시다.

```
 x/s 0x402021
2021:       "You passed the challenge!"
```

즉, 성공 문자열을 출력하는 케이스에 도착했다는 것을 알 수 있습니다.
이를 종합해보면 x는 64보다 크고 96보다 작거나 같아야 하며, z는 512보다 커야하고, 560보다 작거나 같아야합니다. 또, '3*y == x + 2*z'를 만족시키면서 x, y, z를 우리는 입력해야합니다. 따라서, 저는,

```
# TODO: Complete this function (do not touch anything else).
def solve():
    prog = Program("./problem2.bin")
    print(prog.read_line()) # Read the initial message of the program.
    prog.send_line("80 400 560")
    print(prog.read_line())

if __name__ == "__main__":
    solve()
```

80, 400, 560을 입력했습니다.

3. problem3

```
<+0>:     push   %rbp
<+1>:     push   %rbx
```

callee-saved register이므로 데이터를 백업시켜 놓고 있습니다.

```
 sub    $0x18,%rsp
```

스택(영역)에 24바이트 할당하고 있습니다.

```
 mov    $0xa,%ebp
```

저는 'int temp1 = 10;'으로 이해했습니다.

```
 mov    $0x0,%ebx
```

저는 'int temp2 = 0;'으로 이해했습니다.

```
 jmp    0x40114e <main+24>
```

<main+24>로 점프해봅시다.

```
<+24>:    cmp    $0x2,%ebx
<+27>:    jg     0x401198 <main+98>
```

temp2가 2보다 크면 <main+98>로 점프함을 알 수 있습니다.

2보다 크다고 가정하고 <main+98>로 점프해봅시다.

```
<+98>:    cmp    $0x191,%ebp
<+104>:   je     0x4011b6 <main+128>
```

temp1이 401과 같으면 <main+128>로 점프하라는 구문입니다.

같다고 가정하고 <main+128>로 점프해봅시다.

```
 <+128>:   mov    $0x40201b,%edi
 <+133>:   callq  0x401030 <puts@plt>
 <+138>:   jmp    0x4011aa <main+116>
) x/s 0x40201b
201b:     "You passed the challenge!"
```

이를 통해, temp1이 401이 되면 성공 문자열을 출력하게 됨을 알 수 있고, 우리는 3번의 기회에서 temp1을 10에서 401로 만들어야 함을 알 수 있습니다.

이제 다시 위로 올라가서 'temp2 = 0'인 상황부터 차근차근 살펴봅시다.

```
 mov    $0x402004,%edi
 callq  0x401030 <puts@plt>
```

puts("Provide your input: "); 부분입니다.

```
 lea    0xc(%rsp),%rsi
 mov    $0x402018,%edi
  callq  0x401040 <__isoc99_scanf@plt>
```

scanf("%d", &x); 부분입니다. 이를 통해서 0xc(%rsp)에 x의 값이 저장될 것임을 알 수 있습니다. 저는 이를 유용하게 이용할 것입니다.

```
 mov    $0x0,%eax
```

저는 'int temp3 = 0;'으로 이해했습니다.

```
mov    0xc(%rsp),%eax
cmp    $0x7,%eax
ja     0x40114b <main+21>
```
x의 값이 7보다 크다고 가정하고 <main+21>로 점프해봅시다.

```
<+21>:    add    $0x1,%ebx
```
아무런 행동도 없이 temp2의 값이 1 증가함을 알 수 있습니다. 따라서, 우리는 x의 값을 입력할 때 0에서 7까지의 정수를 입력해야 함을 알 수 있습니다.
0에서 7까지 입력했다고 가정합시다.

```
mov    %eax,%eax
jmpq   *0x402060(,%rax,8)
```
입력한 x의 값에 따라 점프할 주소가 다르다는 것을 알 수 있고 2번째 문장을 통해 switch문임을 확인할 수 있습니다. 0에서 7까지 8개의 값이므로 'x/8xg 0x402060' 명령어를 입력하여 알맞은 점프테이블을 확인합시다.

```
(gdb) x/8xg 0x402060
0x402060:       0x0000000000401148      0x000000000040114b
0x402070:       0x0000000000401183      0x0000000000401188
0x402080:       0x000000000040114b      0x0000000000401191
0x402090:       0x000000000040114b      0x000000000040118c
```
0에서 7까지 입력했을 때 어디로 점프하는지 나와있습니다.
'Case 3:', 'Case 7:', 'Case 0:' 상황을 봅시다.

```
401138 <+82>:    add    %ebp,%ebp
40113a <+84>:    jmp    0x40114b <main+21>
```
위의 상황은 x에 3을 입력했을 때입니다. 위의 코드를 통해 처음 3을 입력하면 10에서 20으로 temp1의 값이 바뀜을 알 수 있습니다.
다음으로 x에 7을 입력해봅시다.

```
<+86>:    imul   %ebp,%ebp
<+89>:    jmp    0x40114b <main+21>
```
x에 7을 입력하면, 20에서 400으로 temp1의 값이 바뀔 것입니다.
마지막으로 3번째 입력 때 x에 0을 입력해봅시다.

```
1148 <+18>:    add    $0x1,%ebp
```
temp1의 값이 400에서 401로 값이 바뀔 것입니다. 그리고 3번의 기회가 지나갔으므로 temp2의 값은 2보다 커지게 되어 아까 이야기 했던 <main+98>로 점프하게 될 것이고, temp1의 값이 401인지 체킹할 것입니다. 401을 올바르게 만들었으니, 성공 문자열을 출력하게 될 것입니다. 따라서, 저는 차례대로 3, 7, 0을 입력했습니다.

```python
# TODO: Complete this function (do not touch anything else).
def solve():
    prog = Program("./problem3.bin")
    print(prog.read_line()) # Read the initial message of the program.
    prog.send_line("3")
    print(prog.read_line())
    prog.send_line("7")
    print(prog.read_line())
    prog.send_line("0")
    print(prog.read_line())


if __name__ == "__main__":
    solve()
```

4. problem4

```
<+0>:     push   %rbp
<+1>:     push   %rbx
```

Callee-Saved Register라서 데이터를 백업해놓고 있다.

```
sub    $0x48,%rsp
```

스택에 72바이트 할당하고 있다.

```
mov    $0x402004,%edi
callq  0x401030 <puts@plt>
```

puts("Provide your input: "); 부분이다.

```
lea    0x20(%rsp),%rsi
mov    $0x402018,%edi
callq  0x401060 <__isoc99_scanf@plt>
```

scanf("%31s", buf); 부분이다. 이를 통해서 0x20(%rsp)부터 buf 문자열이 저장되기 시작될 것임을 알 수 있다.

```
mov    $0x0,%eax
```

나는 'int temp1 = 0;'으로 이해했다.

```
lea    0x20(%rsp),%rdi
callq  0x401040 <strlen@plt>
```

buf를 인자로 하는 strlen함수 호출 부분이다.

```
mov    %eax,%ebx
```

n = strlen(buf); 부분이다. 이를 통해서 %ebx 레지스터가 n을 의미함을 알 수 있다.

```
cmp    $0xb,%eax
je     0x4011ae <main+88>
```

buf 문자열의 길이가 11인지 체킹하고 있다. buf 문자열의 길이가 11이라고 가정하고 <main+88>로 점프해보자.

```
<+88>:    mov    $0x1,%ebp
```

나는 'int temp2 = 1;'로 이해했다. 이 temp2의 값이 무엇이냐에 따라서 나중에 성공 문자열을 출력할지 말지 결정될 것이다. 나중에 더 이야기하겠다.

```
jmp    0x401190 <main+58>
```

일단 <main+58>로 점프해보자.

```
<+58>:    mov    $0x1a,%edx
<+63>:    mov    $0x0,%esi
<+68>:    mov    %rsp,%rdi
<+71>:    callq  0x401050 <memset@plt>
```

이 코드를 설명하겠다. 일단, %rsp부터 26바이트까지를 visit이라는 이름의 길이가 26인 char형 배열로 이해해보자. 그럼 이 코드는 memset(visit, 0, 26);이 된다. 즉, 길이가 26인 char형 배열의 값을 0으로 초기화하겠다는 의미의 함수호출이다. 나중에 이 visit 배열이 아주 중요한 역할을 한다.

```
mov    $0x0,%esi
mov    $0x0,%ecx
```
각각, 'int temp3 = 0;', 'int temp4 = 0;'으로 이해했다. 즉, %esi는 temp3, %ecx는 temp4를 의미한다.

```
jmp    0x4011b8 <main+98>
```
다시, <main+98>로 점프하자.

```
cmp    %ebx,%ecx
jge    0x4011fe <main+168>
```
temp4의 값이 n보다 크거나 같은지 묻고 있다. 일단 현재 temp4의 현재 값이 0이지만 n보다 크거나 같은 상황이 되었다고 가정하고, <main+168>로 점프해보자.

```
<+168>:   cmp    $0x5,%esi
<+171>:   jne    0x401207 <main+177>
```
temp3의 값이 5인지 아닌지 체킹하고 있다. 5라고 가정해보고 점프하지 말아보자.

```
test   %ebp,%ebp
jne    0x40121d <main+199>
```
temp2의 값이 0인지 아닌지 체킹하고 있다. 0이 아니라고 가정하고, <main+199>로 점프해보자. 즉, 1이라고 가정하자.

```
<+199>:   mov    $0x40201d,%edi
<+204>:   callq  0x401030 <puts@plt>
<+209>:   jmp    0x401211 <main+187>
```
'x/s 0x40201d'를 해보자. 성공 문자열을 출력함을 알 수 있다.

```
 x/s 0x40201d
01d:    "You passed the challenge!"
```
즉, temp4가 n이 되기 전에, 우리는 temp3을 5로 만들어야 되고, 아까 설정한 temp2 = 1을 유지해야 한다..!!
다시 돌아가서 temp4가 0인 상황에서 다시 시작해보자.

```
 movslq %ecx,%rax
 movzbl 0x20(%rsp,%rax,1),%eax
```
buf[temp4]의 값을 temp1에 저장하고 있다. 즉, 'temp1 = buf[temp4];'로 이해하면 된다.

```
lea    -0x61(%rax),%edx
cmp    $0x19,%dl
jbe    0x4011d1 <main+123>
```
97 <= buf[temp4] <= 122인지 체킹하고 있다. 즉, buf[temp4]가 소문자 알파벳인지 체킹하고 있다. 소문자 알파벳이라고 가정해보자..!! 그리고, <main+123>으로 점프해보자.

```
<+123>:   mov    %ebx,%edx
<+125>:   sub    %ecx,%edx
<+127>:   sub    $0x1,%edx
```
%edx 레지스터에 'n-1-temp4'의 값이 저장될 것이다.

```
movslq %edx,%rdx
cmp    %al,0x20(%rsp,%rdx,1)
je     0x4011e6 <main+144>
```
buf[temp4]와 buf[n-1-temp4]의 값을 비교하고 있다. 즉, 우리가 입력한 문자열 buf에서 temp4번째 문자와 n-1-temp4번째 문자를 비교하는 것이다. 즉, 펠린드롬(?)처럼 앞 문자와 뒤 문자가 일치한지 비교하고 있다. 같다고 가정하고 <main+144>로 점프해보자.

```
movsbl %al,%eax
sub    $0x61,%eax
```
buf[temp4]에서 97을 빼고 temp1에 다시 저장하고 있다.

```
movslq %eax,%rdx
cmpb   $0x0,(%rsp,%rdx,1)
jne    0x4011b5 <main+95>
```
visit[buf[temp4]-97]의 값이 0인지 아닌지 체킹하고 있다. 0이 아니라고 가정하고 <main+95>로 점프해보자.

```
+95>:   add    $0x1,%ecx
+98>:   cmp    %ebx,%ecx
+100>:  jge    0x4011fe <main+168>
```
아무런 행동도 하지않고 바로, temp4에 1을 더하고 있고 다시 상황이 반복되는 것을 알 수 있다.

다시 돌아가서, visit[buf[temp4]-97]이 0이라고 가정해보자.

```
+159>:  movb   $0x1,(%rsp,%rdx,1)
+163>:  add    $0x1,%esi
+166>:  jmp    0x4011b5 <main+95>
 <+95>:   add    $0x1,%ecx
 <+98>:   cmp    %ebx,%ecx
 <+100>:  jge    0x4011fe <main+168>
```
visit[buf[temp4]-97]을 1로 설정하고 temp3에 1을 더하고 있다.

그리고 <main+95>로 점프해서 다시 아까처럼, 상황이 반복될 것이다.

이 점을 통해 visit배열은 우리가 입력한 buf 문자열에서 각 알파벳 등장 여부를 체킹하는 배열임을 알 수 있다. 알파벳의 개수가 26개이므로 딱 맞아 떨어진다. 즉, 등장하지 않은 알파벳이라면 1로 값을 설정하여 방문 표시를 해주고, 이미 등장한 알파벳이라면 이미 값이 1인 상태가 되어 아까처럼, <main+95>로 점프를 바로하여 temp3의 값이 변경되지 않은 채 다음 문자 비교를 위해 temp4에 값에 1을 더할 것이다. 그리고, 여태까지 본 어셈블리 코드 중에서 temp2의 값을 변경하는 코드는 없다. 즉 초기에 설정한 1에서 값을 건드리는 코드는 없다.

즉, 정리하자면 우리가 입력해야할 buf 문자열은 길이가 11이어야 하며, 펠린드롬처럼 앞과 뒤의 문자가 같아야 하며, temp3을 5로 만들어주어야 한다. 여기서 알아야 할 점은 11은 5 + 1 + 5로 나타낼 수 있다. 물론, 많은 경우로 성공 문자열을 출력할 수 있다.

하지만 나는 이런 경우를 떠올렸다.

앞의 5문자 중 앞의 4문자는 등장하지 않은 알파벳으로 구성하고, 나머지 1개의 문자는 앞의

4개의 문자 중 등장했던 알파벳으로 구성해서 temp3의 값이 4까지 만들고(물론, 뒤의 5문자는 앞의 5문자와 대응되게 해서 temp3의 값을 4로 만들어줘야 하는 것은 당연하다..!!), 나머지 부분인 5 + 1 + 5 부분에서 1에 해당하는 부분은 temp4와 n-1-temp4의 값이 같으므로 당연히 buf[temp4]와 buf[n-1-temp4]가 같은 상황이므로, 이때 앞의 문자들과 다른 소문자 알파벳을 하나 골라서 문자열을 구성해주면, temp3은 5가 될 것이다. 방금 설명했던 경우를 만족하는 길이가 11인 문자열은 많겠지만, 나는 "jangayagnaj"를 입력했다.

```
7
8 # TODO: Complete this function (do not touch anything else).
9 def solve():
0     prog = Program("./problem4.bin")
1     print(prog.read_line()) # Read the initial message of the program.
2     prog.send_line("jangayagnaj");
3     print(prog.read_line())
4
5 if __name__ == "__main__":
6     solve()
```

이 문제가 다른 문제에 비해 복잡한 어셈블리 코드로 이루어져 있어서 다시 요약해서 정리하자면, 우리가 입력해야 하는 문자열의 길이는 11이어야 하며, 펠린드롬처럼 앞 문자와 대응되는 뒤의 문자는 같아야 하며, 다시 등장한 알파벳이라면 카운팅이 무시되므로 즉, temp3의 값이 1 증가하지 않으므로, 새로운 알파벳 5개와 5개 중 하나를 골라서, 총 2 * 5 + 1 즉, 길이가 11인 문자열을 입력해야 합니다.

문제 2-1부터 2-4까지 check.py를 돌린 결과 올바르게 O표시가 모두 나타난 것을 확인할 수 있습니다.

```
cse20191274@cspro2:~/computersystem/Lab2$ ./check.py
[*] 2-1: O
[*] 2-2: O
[*] 2-3: O
[*] 2-4: O
```

항상 좋은 문제 감사드립니다..!!

이상입니다. 감사합니다.

-20191274 장유빈-