CSLAB2 Report

전공: 컴퓨터공학과 학년: 2학년 학번: 20241589 이름: 김홍래

1.

Problem1의 어셈블리를 살펴보면 다음과 같이 출력된다. 이들을 차근차근 읽어보자.

Dump of assembler code for function main:

0x0000000000401136 <+0>: sub $0x28,%rsp

스택 포인터의 주소를 0x28, 즉 40만큼 감소 시켜서 자리를 확보한다.

0x000000000040113a <+4>: mov $0x402004,%edi

첫번째 인자인 %rdi에 0x402004를 저장한다. 0x402004는 "Provide your input:"이라는 문자열이 저장된 장소이다.

0x000000000040113f <+9>: callq 0x401030 <puts@plt>

0x401030을 호출하는데 이게 바로 Puts 함수이다. 이 함수를 호출해서 위의 문자열을 출력한다.

0x0000000000401144 <+14>: mov %rsp,%rsi

두번째 인자인 %rsi에 %rsp의 값을 넣는다. 이는 아까 <+0>에서 줄여준 부분에 나중에 scanf를 받기 위한 공간의 주소를 %rsi에 넣는다.

0x0000000000401147 <+17>: mov $0x402018,%edi

첫번째 인자인 %rdi에 scanf에 넘길 문자열이 담긴 주소를 옮겨준다. x/s를 해보면 "%31s"가 출력된다.

0x000000000040114c <+22>: mov $0x0,%eax

0x0000000000401151 <+27>: callq 0x401040 <__isoc99_scanf@plt>

0x0000000000401156 <+32>: mov $0x0,%eax

Scanf를 호출하고, eax는 횟수 세는용 인덱스다.

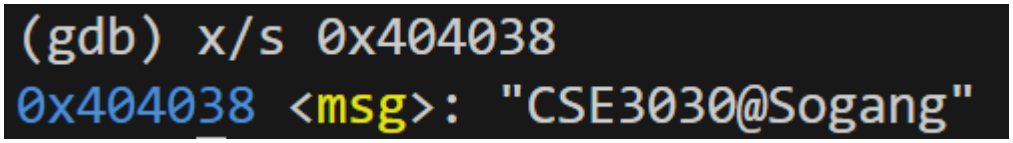
0x000000000040115b <+37>: movslq %eax,%rdx

레지스터 eax를 rdx로 확장시킨다. 확장은 sign-extension을 따른다.

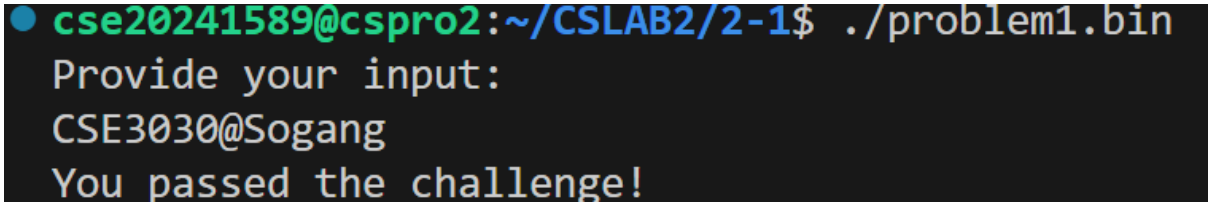
0x000000000040115e <+40>: movzbl 0x404038(%rdx),%edx

Edx에 0x404038에 rdx를 더한 주소를 edx에 대입하라는 라인이다. 밑에 test %dl,%dl 부분을

보면 rdx가 0인지 아닌지를 보고 있다. 근데 0x404038에 값을 집어넣은 적도 없지만, 여기에 +인덱스만큼 한 곳의 주소를 찾는 게 배열이 이곳에 있다는 걸 알았다. 그래서 0x404038에 어떤 값을 넣은 기록도 없는 걸 봤을 때, 찾아야 할 값이 들어있는 위치로 보였다. 이를 string으로 출력해보니



우리가 찾아야 할 문자열처럼 보이는 문자열이 나왔다. 이를 bin 파일을 실행해서 입력해보면



맞는 문자열을 찾을 수 있었다.

2.

이 문제는 숫자 세개를 입력받아서 내부의 숫자와 같은지를 찾는 문제다.

0x0000000000401144 <+14>: lea 0x4(%rsp),%rcx

0x0000000000401149 <+19>: lea 0x8(%rsp),%rdx

0x000000000040114e <+24>: lea 0xc(%rsp),%rsi

이부분을 보면 int형 세개를 받기 위해 스택 포인터를 이동시키고 있다.

그 뒤에 따라오는 부분을 보자.

0x0000000000401162 <+44>: mov 0xc(%rsp),%edx

0x0000000000401166 <+48>: cmp $0x40,%edx

0x0000000000401169 <+51>: jle 0x4011be <main+136>

0x000000000040116b <+53>: mov $0x1,%ecx

0x0000000000401170 <+58>: cmp $0x60,%edx

0x0000000000401173 <+61>: jle 0x40117a <main+68>

0xc면 세번째 값에 해당한다. 즉, c를 0x40, 0x60와 비교해서 작거나 같으면 점프, 아니면 그대로 진행한다. 점프하면 136으로 가는데 여기서 ecx에 0을 넣고 다시 돌아온다. 하지만 점프가 없었다면 그 밑에 53에서 1을 넣는다. 이를 보면 ecx가 성공 실패 여부를 기록하는 부분으로 추정된다. 그 뒤에 다시 edx, 즉 c를 0x60과 비교해서 작으면 ecx를 수정하는 과정을 거치지 않지만,

크면 ecx를 0으로 수정한다. 이걸 보면 c가 0x40에서 0x60사이일 때 ecx는 1이 되고, 그 밖이면 0을 넣는다.

```
0x000000000040117a <+68>:    mov    0x4(%rsp),%eax
   0x000000000040117e <+72>:    cmp    $0x200,%eax
   0x0000000000401183 <+77>:    jg     0x40118a <main+84>
   0x0000000000401185 <+79>:    mov    $0x0,%ecx
   0x000000000040118a <+84>:    cmp    $0x230,%eax
   0x000000000040118f <+89>:    jle    0x401196 <main+96>
   0x0000000000401191 <+91>:    mov    $0x0,%ecx
   0x0000000000401196 <+96>:    mov    0x8(%rsp),%esi
```

이후 코드를 보면 다음과 같다. 0x4, 0x8, 0xc는 각각 a,b,c의 주소를 담고있고, 이번엔 eax에 a를 넣었다. 전체 코드를 살펴보면 0x200 < a <= 0x230일 때 ecx에 0을 넣는 과정을 건너뛰고 있다. A가 저 범위 안에 있을 때만 유효하다.

```
0x0000000000401196 <+96>:    mov    0x8(%rsp),%esi
   0x000000000040119a <+100>:   mov    %esi,%edi
   0x000000000040119c <+102>:   sub    %edx,%edi
   0x000000000040119e <+104>:   sub    %esi,%eax
   0x00000000004011a0 <+106>:   add    %eax,%eax
   0x00000000004011a2 <+108>:   cmp    %eax,%edi
```

아까 edx = c , eax =a 였고, 이번엔 esi = b이다. Cmp 전까지 계산 과정을 보면 edi = b-c가 되고, eax = a – b 가 되고, 여기서 eax를 두배하므로 eax = 2(a-b)이다. Edi가 eax와 같지 않으면 116으로 점프, 이외에는 그대로 진행한다. 수식을 계산하면 116에는 0x402040을 edi에 대입하는데 이 주소에 있는 문자열은 실패문자고 곧바로 이를 출력한다. 점프를 하지 않았을 때, 수식은 edi = eax, 즉 b-c =2(a-b) 일때 점프를 안한다. 그 밑에 test는 ecx끼리 비교하는데 이는 세번째 조건인 b-c = 2(a-b) 말고 다른 앞의 두개의 조건이 참인지를 판별해서 ecx 가 0이 아닐 때 점프를 한다. 143의 0x402021에는 정답문자가 숨겨져 있다. 따라서 지금까지 구한 세가지 조건을 10진수로 표현해보면

64 < c <= 96, 512 < a <= 560 , b = (2a+c)/3

일 때가 우리가 찾는 경우이다.

이 경우로 c = 70, a = 520, b = 1110/3 = 370 이 답이다. 하지만 이렇게 찾아도 안되서 다시 생각해보니 스택 포인터는 높은 주소부터 역순으로 주소를 할당하는 점을 잊고 있었다. 따라서 사실은 어셈블리에서 처음에 주어진 조건은 c가 아니라 a에 대한 조건이었고 두 번째 조건은 a가 아니라 c에 대한 조건이었다. 따라서 정확한 조건은

64 < a <= 96, 512 < c <= 560 , b = (2c+a)/3

이므로 정확한 답은 a = 70, b = 370, c = 520이다.

```
cse20241589@cspro2:~/CSLAB2/2-2$ ./problem2.bin
Provide your input:
520 370 70
No, that is not the input I want!
cse20241589@cspro2:~/CSLAB2/2-2$ ./problem2.bin
Provide your input:
70 370 520
You passed the challenge!
```

3.

일단 첨부터 스택 포인터를 조정하고, ebp = a, ebx = 0을 대입한다.

0x0000000000401138 <+2>: sub $0x18,%rsp 스택포인터 내려서 공간 확보

0x0000000000401141 <+11>: mov $0x0,%ebx

0x0000000000401146 <+16>: jmp 0x40114e <main+24>

0x000000000040114e <+24>: cmp $0x2,%ebx

0x0000000000401151 <+27>: jg 0x401198 <main+98>

Ebx가 2보다 클 때 점프를 하라는 문장이다. Ebx는 0이었고 2보다 클 때 점프를 하라는 의미이므로 Ebx는 i이다. 그 밑에 mov와 callq를 보면 Provide your input 과 scanf가 있는걸 봐서 for문 내부가 이어진다. Eax에 scanf한걸 받고 그 밑에서 0x7과 비교를 하고 ja이므로 eax가 7보다 클 때 점프를 하라는 의미이다.

여기서 점프를 했을 때를 생각해보면 eax는 7보다 큰거고 도착지점에서 i인 ebx를 1만큼 늘려준다. 그러고 나서 다시 내려가는데 아직 2보단 작아서 아까 했던대로 input을 다시 받고 eax가 0x7보다 클 때 다시 점프하고 아니면 그대로 진행한다. 그러면 그대로 진행했을 때, 즉 eax가 0x7보다 작을 때를 생각해보자.

0x000000000040117c <+70>: jmpq *0x402060(,%rax,8)

이 문장을 만나서 0x402060 + rax*8에 들어있는 주소로 이동하라는 의미이다. 이는 switch문 역할

을 하는 것으로 rax로 가능한 값은 0부터 7이고 그보다 크면 default인 점프를 하게 된다. 아까 처음에 7보다 컸을 때가 결국 default 구문이었고, 이때는 별다른 작업 없이 반복횟수 1회 했다는 걸 기록해주고 그대로 다음 변수를 받는다. 그러면 다른 경우의 수를 살펴보기 위해 rax가 0부터 7까지일 때 어디로 점프하는지를 보기 위해 0x402060부터 8*8개의 메모리를 보자.

```
(gdb) x/64xb 0x402060
0x402060:       0x48    0x11    0x40    0x00    0x00    0x00    0x00    0x00
0x402068:       0x4b    0x11    0x40    0x00    0x00    0x00    0x00    0x00
0x402070:       0x83    0x11    0x40    0x00    0x00    0x00    0x00    0x00
0x402078:       0x88    0x11    0x40    0x00    0x00    0x00    0x00    0x00
0x402080:       0x4b    0x11    0x40    0x00    0x00    0x00    0x00    0x00
0x402088:       0x91    0x11    0x40    0x00    0x00    0x00    0x00    0x00
0x402090:       0x4b    0x11    0x40    0x00    0x00    0x00    0x00    0x00
0x402098:       0x8c    0x11    0x40    0x00    0x00    0x00    0x00    0x00
```

위에서부터 순서대로 rax = 0 부터 7인 경우이다. 각각의 메모리에 저장된 값들은 점프문의 목적지이다.

Rax = 0일때부터 살펴보자. 점프 목적지는 0x401148이고 어셈블리문에서 +18에 해당하는 위치이다. 여기서 아까 맨 처음에 ebp는 0xa였고 +18을 지나면서 ebp는 1만큼 증가시리카는 문장을 지나면 0xb 가 된다. 그후 다시 +24로 가서 for문 확인을 하고 변수를 받는다.

Rax= 1이면 목적지는 0x40114b이고 이는 +21, default처럼 변수를 바꾸는 거 없이 i만 1만큼 증가시키고 그대로 input을 받는다.

Rax = 2이면 목적지는 0x401183, 즉 +77이다. 여기선 ebp를 1만큼 감소시키고 다시 21로 가서 i를 1만큼 증가시켜주고 다시 scanf를 한다.

Rax = 3이면 목적지는 0x401188, 즉 +82이다. 여기선 ebp + ebp = ebp 이므로 ebp를 두배시켜주고 다시 for문 처음으로 돌아간다.

Rax = 4면 0x40114b, Rax = 1일때랑 똑같다.

Rax = 5면 0x401191, ebp에 0xa를 대입해주고 돌아간다.

Rax = 6이면 0x40114b, default랑 똑같다.

Rax = 7이면, 0x40118c, ebp를 제곱하고 돌아간다.

이를 정리해보면

Case: 0 ebp++;

Case: 2 ebp--;

Case: 3 ebp*=2;

Case: 5 ebp = 10;

Case 7: ebp *= ebp;

Default:

이다. 이렇게 연산이 끝났으면 +27에서 +98로 점프를 해서 ebp가 0ㅌ191과 비교를 한다. 점프를 안했을 때 0x402038을 출력하는데 이는 오답결과를 내고 종료한다. 하지만 0x191과 같아서 점프를 했으면 +128로 가서 0x40201b를 출력하고 종료한다.

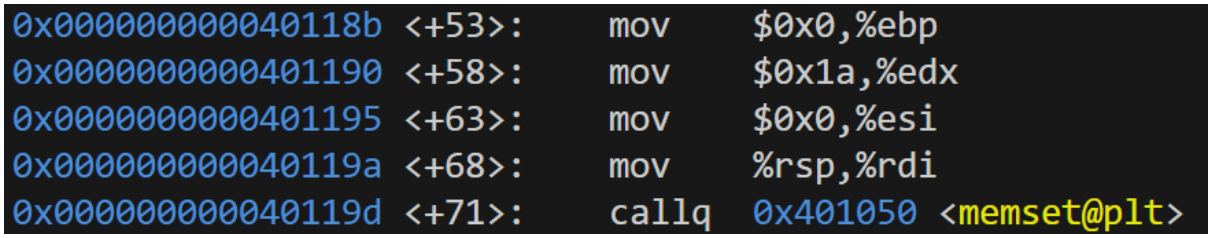
따라서 우리가 찾아야하는 정답은 저 위의 case문 계산을 세 번을 해서 0x191, 즉 256 + 144 + 1 = 401을 만드는 것이다. 이는 간단하다. (10*2)^2 + 1이 401이므로 두배, 제곱, +1을 순서대로 해주면 된다. 따라서 답은 3,7,0이다.

```
cse20241589@cspro2:~/CSLAB2/2-3$ ./problem3.bin
Provide your input:
3 7 0
Provide your input:
Provide your input:
You passed the challenge!
```

4.

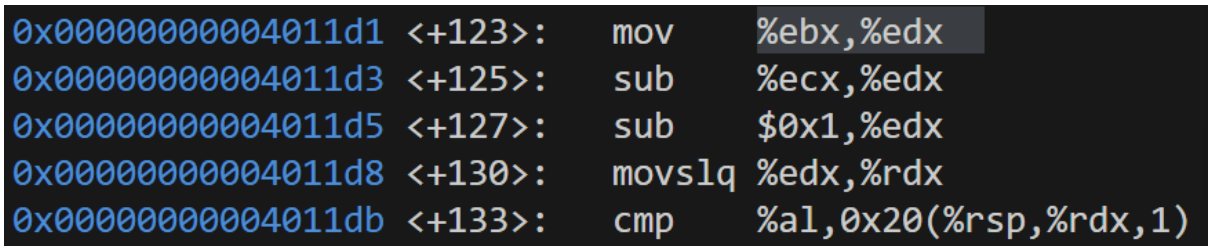
```
   0x0000000000401156 <+0>:     push   %rbp
   0x0000000000401157 <+1>:     push   %rbx
   0x0000000000401158 <+2>:     sub    $0x48,%rsp
   0x000000000040115c <+6>:     mov    $0x402004,%edi
   0x0000000000401161 <+11>:    callq  0x401030 <puts@plt>
   0x0000000000401166 <+16>:    lea    0x20(%rsp),%rsi
   0x000000000040116b <+21>:    mov    $0x402018,%edi
   0x0000000000401170 <+26>:    mov    $0x0,%eax
   0x0000000000401175 <+31>:    callq  0x401060 <__isoc99_scanf@plt>
   0x000000000040117a <+36>:    lea    0x20(%rsp),%rdi
   0x000000000040117f <+41>:    callq  0x401040 <strlen@plt>
   0x0000000000401184 <+46>:    mov    %eax,%ebx
   0x0000000000401186 <+48>:    cmp    $0xb,%eax
--Type <RET> for more, q to quit, c to continue without paging--c
   0x0000000000401189 <+51>:    je     0x4011ae <main+88>
```

처음부터 je까지는 통상적으로 "Provide your input:" 을 출력하고 scanf하는 과정이다. 여기서 0x48만큼의 스택 포인터를 내려서 자리를 확보한다. Eax를 0으로 초기화해서 scanf로 rsi는 문자열의 주소를 저장하고 있고, 0x20(%rsp)에 문자열이 저장된다. 그 뒤에 strlen을 호출해서 eax에 문자열의 길이를 계산해서 ebx에 넣는다. 이 문자열의 길이가 11과 같으면 +88로 점프를, 아니면 그대로 진행한다.

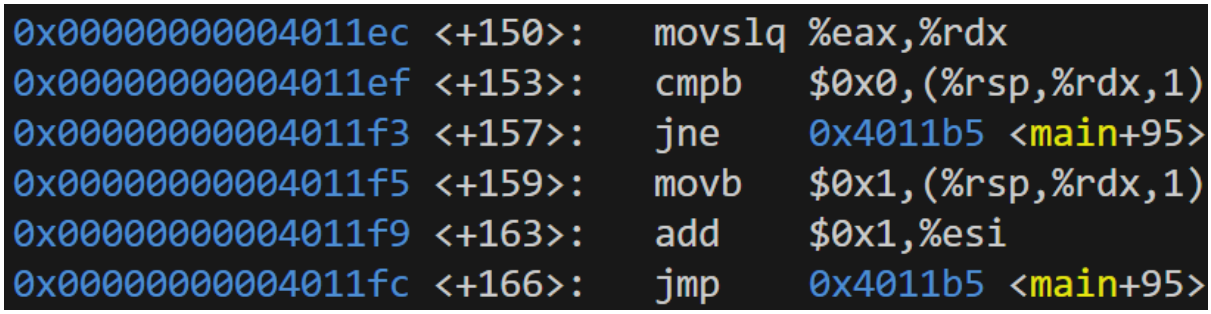


점프를 안했을 때, 즉 문자열 길이가 11이 아닐 때를 생각해보자. Ebp에 0을 저장한다. 점프를 했으면 +88로 가서 Ebp에 1이 저장되고 바로 다시 +58로 돌아온다. Ebp는 문자열 길이가 11인지 아닌지를 알려주는 레지스터이다. 그 뒤에 edx를 0x1a로, esi를 0x0으로, rdi를 rsp로 초기화한다. 그리고 memset 함수를 호출한다. 여기서 rdi, rsi, rdx는 순서대로 인자들이고, 이를 c에 대입해보면 Memset(rsp,0,26)이 된다. Memset은 C에서 첫번째 인자 자리부터 26개의 자리에 0으로 초기화하는 함수이다. 즉, rsp부터 그 위로 26개의 자리에 0으로 일단 초기화를 했다. 아직 이게 어디에 쓰이는지는 모르겠지만, 밑에 코드를 보면 아마 알게 될 것이다.

그후 esi, ecx에 0으로 초기화하고 +98로 점프한다. 아까 문자열 11일 때를 담당하는 라인들을 건너뛴다. 여기서 ebx와 ecx를 비교하는데 ebx는 아까 문자열의 길이를 저장한 곳이고, ecx는 아까 0으로 초기화 했다. 점프는 ecx가 ebx보다 크거나 같을 때 점프하므로 ecx는 반복 횟수 i를 기록하는 공간으로 예상된다. 그러면 계속 진행해보면 +102에서 rax에 i를 저장하고 이를 eax에 또 움직이는데 movzbl 0x20(%rsp,%rax,1),%eax 이 있는걸 보면 eax에 rsp + 20 + rax*1 을 넣는다. 아까 rsp + 20에 문자열을 넣었으니 결국 반복 할때마다 eax는 처음 받은 문자열의 i번째 문자를 가리키고 있다는 의미이다. 즉, for문을 문자열 사이즈만큼 반복하는 중이다. 그후 edx에 rax-0x61의 값을 넣는데, 지금 우리가 하고 있는게 문자열 값을 받아와서 0x61만큼 감소시키는 중이다. 컴퓨터는 문자를 ascii 코드 값으로 인식하는데, 0x61이면 97, 즉 소문자 a의 ascii 값이다. 이는 뒤에 0x19, 즉 25보다 작거나 같을 때 +123으로 점프하고 아니면 ebp에 0을 넣고 동작한다. 즉, Ebp가 1일려면 문자열이 11이면서 동시에 문자가 알파벳이어야한다.



그 뒤에 과정이 좀 복잡한데 ebx는 아까 문자열 길이를 저장해둔 레지스터고, 이를 edx에 넣고, ecx 즉 i를 edx에서 빼고, 여기서 1을 더 뺀다. 그러면 rdx는 10-i가 된다. +133은 문자열 시작에서 10-i만큼 떨어진 곳을 보고 있는데, al은 eax로 아까 원래 i번째 문자, 뒤에 문자는 10-i번째 문자를 보고 있다. 즉, 앞의 문자와 뒤에 문자를 비교하고 있다. 예전에 백준에서 풀었던 회문이 생각나는 부분이었다. 그래서 이 둘이 같으면 +144로 점프, 아니면 ebp에 0을 넣는다. 따라서 만약 문자열이 회문이면 ebp가 1로 유지되고 아니면 0이 된다.



이어진 부분을 보면 보고 있던 i번째 문자에 0x61을 빼서 알파벳의 순서를 받고 있다. 그리고 이를 rsp + rdx*1, 즉 아까 맨 처음에 26개의 칸을 0으로 메모리 셋한 부분에 접근해서 이들이 0이 아니면 +95로 돌아가고, 0이면 1을 더해주고 돌아간다. 26개 칸을 a부터 z라고 생각하고, 이중 어느 알파벳이 있었는지 기록하는 것이다. 아까 esi를 0으로 초기화 했었는데, esi도 1씩 같이 올려준다. 이는 esi를 알파벳 개수 측정 용도로 보인다. 이 과정이 끝난 뒤, 다시 +95, for문으로 돌아간다. 렇게 해서 아까 문자열 개수만큼 반복을 한 뒤에 +100에서 +168로 점프를 하면 esi를 5와 비교해서 만약 esi가 5가 아니면 +177로 가서 0x402038을 출력하고 프로그램을 종료하는데, 0x402038은 실패문자가 담겨있다. 하지만 esi == 5를 만족하면 마지막으로 ebp가 1이면 +199로 이동하는데 이곳은 성공문자를 출력하는 곳이다. 따라서 이번 문제에서 출력 해야 되는 문자는

1. 문자열 길이가 11이어야 한다.
2. 알파벳으로만 구성되어야 한다.
3. 회문(i번째 문자와 size-i-1번째 문자가 동일)이어야 한다.
4. 알파벳을 정확히 5개만 써야된다.

따라서 우리가 찾는 문자열의 예시로 “qwerttrewq” 정도가 적당하다.