리버싱 교육 2회차

201702075 조수환

ARGOS

INDEX

ARGOS

리버싱 교육 2회차

어셈블리어 유형

- 변수 할당
- 조건문
- 반복문

ARGOS

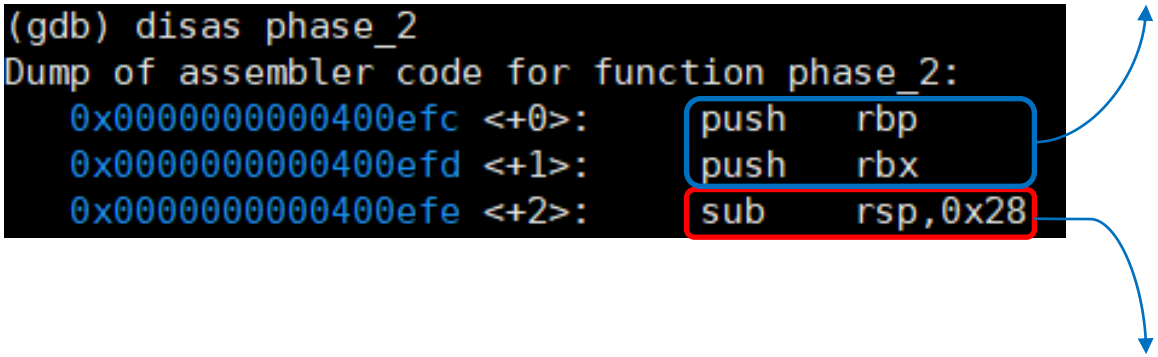
변수 할당

ARGOS

disas 명령어로 함수를 까보면 항상 보이는 구문이 있습니다.

이를 함수 프롤로그라 하는데 함수를 실행하기 위한

준비 과정이라고 생각하시면 됩니다. (Bomb Lab 해결에 영향 X)

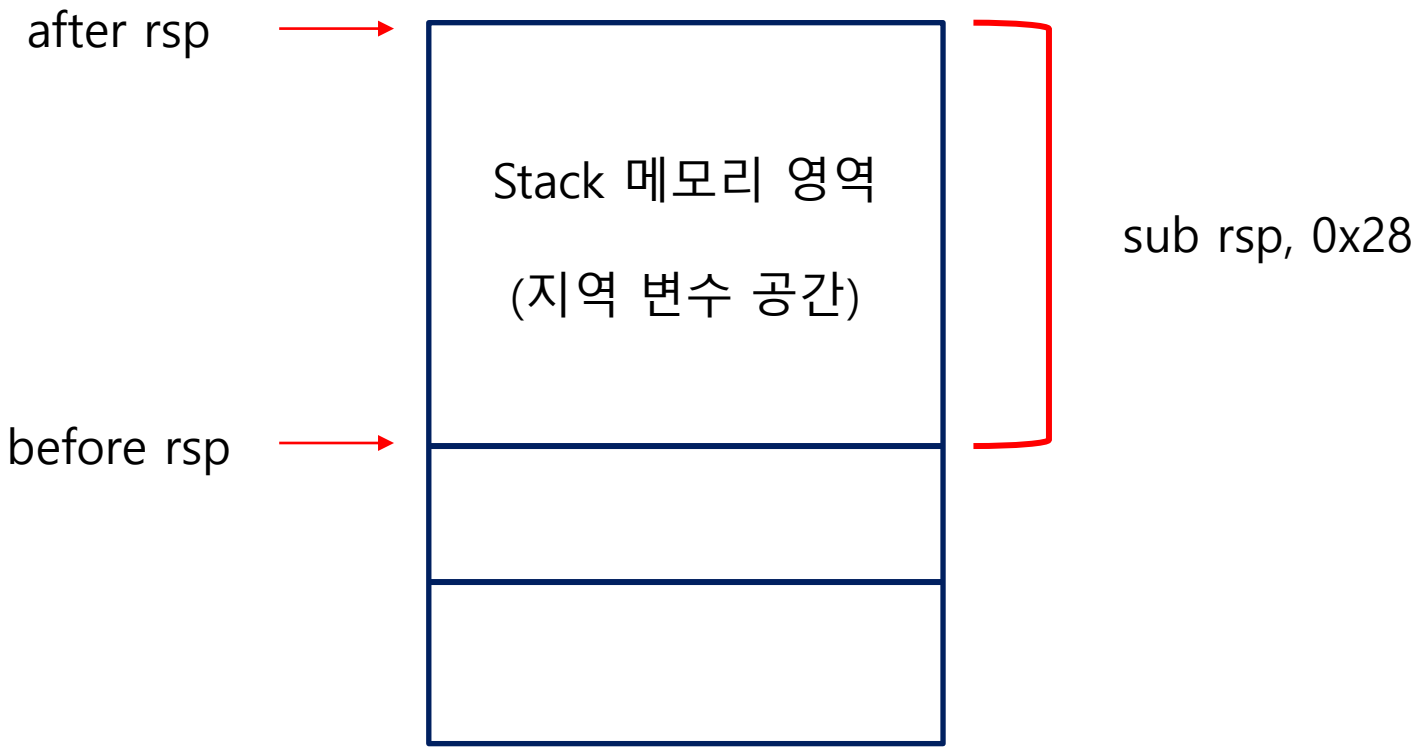


함수 프롤로그 이후 rsp(스택 포인터)에 대한 sub 명령의 의미

= 스택 포인터를 옮겨서 지역 변수가 들어갈 공간을 확보 하는 것

변수 할당

sub 명령어로 빼버린 만큼 지역 변수가 들어갈 공간이 생깁니다.



조건문

조건문은 보통 cmp와 jump의 조합으로 표현됩니다.

```c
int a;
scanf("%d", &a);

if(a == 5) {
    printf("five\n");
}
```

```
<+46>:    call   0x1090 <__isoc99_scanf@plt>
<+51>:    mov    eax,DWORD PTR [rbp-0xc]
<+54>:    cmp    eax,0x5
<+57>:    jne    0x11d0 <main+71>
<+59>:    lea    rdi,[rip+0xe3c]        # 0x2007
<+66>:    call   0x1070 <puts@plt>
<+71>:    mov    eax,0x0
```

위의 예시에서는 eax(== a)와 0x5(5)를 비교했을 때

같지 않으면 main+71로 jump(jne <main+71>)하여

printf의 실행 여부를 결정함

반복문

반복문은 조건 비교 -> 구문 실행의 구조를 가집니다.

이는 어셈블리어에서 cmp(비교)와 이전으로 jump(구문 실행)으로 표현됩니다.

```
for(int i=0 ; i<10 ; i++) {
    printf("test\n");
}
```

```
<+12>:    mov    DWORD PTR [rbp-0x4],0x0
<+19>:    jmp    0x116e <main+37>
<+21>:    lea    rdi,[rip+0xe9f]        # 0x2004
<+28>:    call   0x1050 <puts@plt>
<+33>:    add    DWORD PTR [rbp-0x4],0x1
<+37>:    cmp    DWORD PTR [rbp-0x4],0x9
<+41>:    jle    0x115e <main+21>
```

위의 예시처럼 변수를 비교해서 이전 구문으로 jump하는 구간이 있다면

반복문이라고 생각 하시면 됩니다.

리버싱 교육 2회차

과제 풀이(phase 2)

- phase 2

ARGOS

phase 2

```
0x0000000000400efd <+1>:     push   rbx
0x0000000000400efe <+2>:     sub    rsp,0x28
0x0000000000400f02 <+6>:     mov    rsi,rsp
0x0000000000400f05 <+9>:     call   0x40145c <read_six_numbers>
0x0000000000400f0a <+14>:    cmp    DWORD PTR [rsp],0x1
0x0000000000400f0e <+18>:    je     0x400f30 <phase_2+52>
0x0000000000400f10 <+20>:    call   0x40143a <explode_bomb>
0x0000000000400f15 <+25>:    jmp    0x400f30 <phase_2+52>
0x0000000000400f17 <+27>:    mov    eax,DWORD PTR [rbx-0x4]
0x0000000000400f1a <+30>:    add    eax,eax
0x0000000000400f1c <+32>:    cmp    DWORD PTR [rbx],eax
0x0000000000400f1e <+34>:    je     0x400f25 <phase_2+41>
0x0000000000400f20 <+36>:    call   0x40143a <explode_bomb>
0x0000000000400f25 <+41>:    add    rbx,0x4
0x0000000000400f29 <+45>:    cmp    rbx,rbp
0x0000000000400f2c <+48>:    jne    0x400f17 <phase_2+27>
0x0000000000400f2e <+50>:    jmp    0x400f3c <phase_2+64>
0x0000000000400f30 <+52>:    lea    rbx,[rsp+0x4]
0x0000000000400f35 <+57>:    lea    rbp,[rsp+0x18]
0x0000000000400f3a <+62>:    jmp    0x400f17 <phase_2+27>
0x0000000000400f3c <+64>:    add    rsp,0x28
0x0000000000400f40 <+68>:    pop    rbx
0x0000000000400f41 <+69>:    pop    rbp
0x0000000000400f42 <+70>:    ret
```

저것들만 실행하지 않으면 된다. 그 이전의 조건 구문을 보자

phase 2

먼저 read_six_numbers 라는 함수를 call 합니다.

이를 직접 까서 분석할수도 있겠으나,

직관적으로 함수명을 보고 추정해보셔도 됩니다. (6개의 숫자를 읽는 함수)

```
0x0000000000400efe <+2>:     sub    rsp,0x28
0x0000000000400f02 <+6>:     mov    rsi,rsp
0x0000000000400f05 <+9>:     call   0x40145c <read_six_numbers>
0x0000000000400f0a <+14>:    cmp    DWORD PTR [rsp],0x1
0x0000000000400f0e <+18>:    je     0x400f30 <phase_2+52>
0x0000000000400f10 <+20>:    call   0x40143a <explode_bomb>
```

DWORD PTR [rsp]와 1을 비교해서 그 값이 같다면

Jump를 통해서 폭탄을 피할 수 있습니다.

근데 DWORD PTR [rsp]가 뭔지 모르겠군요.

phase 2

DWORD PTR [피연산자]는 피연산자의 주소에 있는 값을 말합니다.

그러니까 DWORD PTR [rsp]는 rsp(스택 포인터)가 가리키는 곳의 값을 말하죠.

그렇다면 rsp는 현재 어디를 가리키고, 그 주소엔 어떤 값이 들어 있을까요?

```
Phase 1 defused. How about the next one?
1 2 3 4 5 6

Breakpoint 1, 0x0000000000400efc in phase_2 ()
```

```
   0x0000000000400f05 <+9>:     call   0x40145c <read_six_numbers>
=> 0x0000000000400f0a <+14>:    cmp    DWORD PTR [rsp],0x1
```

먼저 phase_2에 6개의 숫자를 입력하고

read_six_numbers 이후까지 프로그램을 진행합니다.

phase 2

x/d $rsp (rsp의 주소의 값을 정수(d)의 형태로 읽겠다(x).)

```
(gdb) x/d $rsp
0x7fffffffe370: 1
```

현재 rsp가 가리키는 주소의 값은 1입니다.

입력을 바꿔 해보시면 알겠지만, 이건 우리가 입력한 6개의 수 중 첫번째 입니다.

여기서 rsp 이후의 값을 보고싶다면 명령어에 범위를 추가하시면 됩니다.

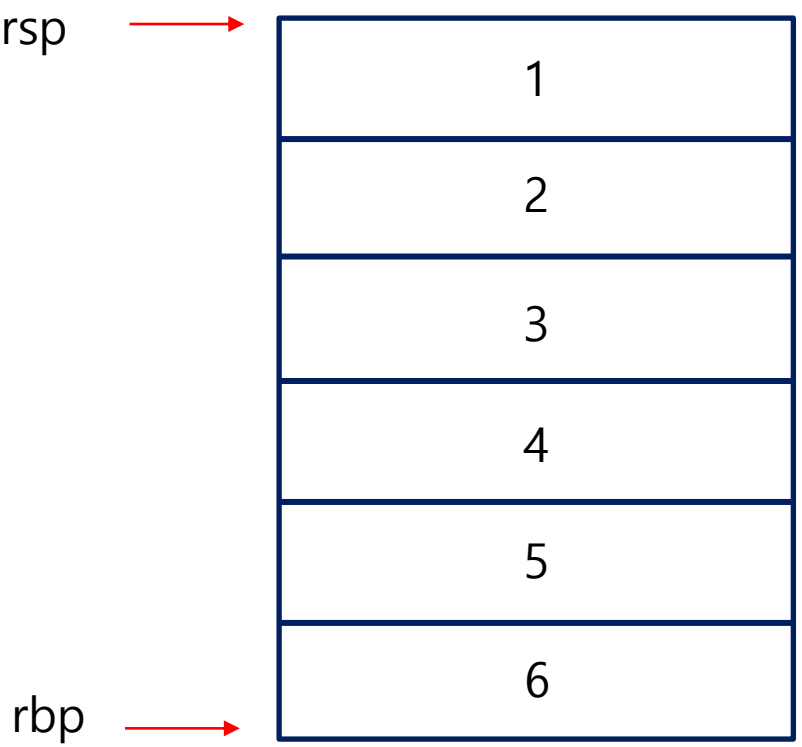
```
(gdb) x/10d $rsp
0x7fffffffe370: 1       2       3       4
0x7fffffffe380: 5       6       4199473 0
0x7fffffffe390: 4203024 0
```

phase 2

우리가 입력한 6개의 숫자가 차례로 stack 메모리 영역에 저장되어 있습니다.

즉 현재 stack 메모리의 추정 상태는 다음과 같습니다.

```
(gdb) x/10d $rsp
0x7fffffffe370: 1       2       3       4
0x7fffffffe380: 5       6       4199473 0
0x7fffffffe390: 4203024 0
```



phase 2

한마디로 입력받은 첫번째 숫자는 반드시 1이어야 한다는 겁니다.

```
0x0000000000400efe <+2>:     sub    rsp,0x28
0x0000000000400f02 <+6>:     mov    rsi,rsp
0x0000000000400f05 <+9>:     call   0x40145c <read_six_numbers>
0x0000000000400f0a <+14>:    cmp    DWORD PTR [rsp],0x1
0x0000000000400f0e <+18>:    je     0x400f30 <phase_2+52>
0x0000000000400f10 <+20>:    call   0x40143a <explode_bomb>
```

이제 jump 구문에 따라 phase_2+52로 이동해봅시다.

```
<+52>:    lea    rbx,[rsp+0x4]
<+57>:    lea    rbp,[rsp+0x18]
<+62>:    jmp    0x400f17 <phase_2+27>
```

rbx 레지스터에 rsp+0x4 주소의 값(입력받은 두번째 숫자)을 넣습니다.

정수형 변수의 크기는 4byte 이므로 0x4만큼 더한다는건 다음 숫자를 의미합니다.

phase 2

이제 phase_2+27로 이동합니다.

```
<+27>:    mov    eax,DWORD PTR [rbx-0x4]
<+30>:    add    eax,eax
<+32>:    cmp    DWORD PTR [rbx],eax
<+34>:    je     0x400f25 <phase_2+41>
<+36>:    call   0x40143a <explode_bomb>
<+41>:    add    rbx,0x4
<+45>:    cmp    rbx,rbp
<+48>:    jne    0x400f17 <phase_2+27>
<+50>:    jmp    0x400f3c <phase_2+64>
```

eax에 DWORD PTR [rbx-0x4]의 값을 넣는다.

즉 rbx에서 4만큼 뺀 주소의 값을 넣는다는 겁니다.

rbx는 현재 rsp+0x4의 값을 갖고 있습니다.

즉, rbx-0x4는 곧 rbx가 원래 가리키던 값(현재는 rsp)를 말합니다.

phase 2

현재 eax에는 이전에 rbx가 가리키던 주소의 값이 들어있습니다.

```
<+27>:    mov    eax,DWORD PTR [rbx-0x4]
<+30>:    add    eax,eax
<+32>:    cmp    DWORD PTR [rbx],eax
<+34>:    je     0x400f25 <phase_2+41>
<+36>:    call   0x40143a <explode_bomb>
<+41>:    add    rbx,0x4
<+45>:    cmp    rbx,rbp
<+48>:    jne    0x400f17 <phase_2+27>
<+50>:    jmp    0x400f3c <phase_2+64>
```

다음 줄을 보면 add eax,eax 구문이 있습니다.

같은것끼리 더했으니 그냥 eax의 값이 두배가 된거죠.

이제 eax와 현재 rbx가 가리키는 값과 비교를 해서

그 값이 같다면 jump를 통해 폭탄을 피할수 있습니다.

phase 2

한마디로 입력받은 6개의 숫자는 항상 이전 숫자보다 두배 커야 합니다.

시작은 반드시 1이어야 하므로 정답은 1 2 4 8 16 32 가 되겠네요.

```
<+27>:    mov    eax,DWORD PTR [rbx-0x4]
<+30>:    add    eax,eax
<+32>:    cmp    DWORD PTR [rbx],eax
<+34>:    je     0x400f25 <phase_2+41>
<+36>:    call   0x40143a <explode_bomb>
<+41>:    add    rbx,0x4
<+45>:    cmp    rbx,rbp
<+48>:    jne    0x400f17 <phase_2+27>
<+50>:    jmp    0x400f3c <phase_2+64>
```

참고로 위의 구문은 rbx의 값을 4씩 증가시키면서

rbx와 rbp가 같아질 때 까지(6개의 수를 모두 비교할 때 까지)

반복을 하는 구문이라고 보시면 됩니다.

phase 2

Phase 2 끗!

```
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
```

이제 phase 3를 진행해야 하는데..

무리해서 다음 phase로 넘어가기 보다는

지금까지의 실습 중에서 애매하거나 모르는 내용을 충분히 알고

어셈블리어 구문에 대한 궁금중을 풀고 가는게 좋을것 같아요.

Q & A

Thank You for Listening

Created By ccoma