시스템소프트웨어(CB2400108-059) - HW2:Bomblab

정보컴퓨터공학부 202255513 김대욱

폭탄이 터지는 것을 막기 위해 *gdb bomb* 명령어를 입력한 후, 아래와 같이 breakpoint를 걸어주어야 한다. 문제 해결을 위한 일련의 과정을 거쳐 정답을 알아낸 후, *continue* 명령어를 입력하여 breakpoint를 넘어가면 된다.

```
(gdb) b explode_bomb
Breakpoint 1 at 0x1b45
(gdb) b phase_1
Breakpoint 2 at 0x12b4
(gdb) b phase_2
Breakpoint 3 at 0x12d4
(gdb) b phase_3
Breakpoint 4 at 0x133d
(gdb) b phase_4
Breakpoint 5 at 0x14e1
(gdb) b phase_5
Breakpoint 6 at 0x1556
(gdb) b phase_6
Breakpoint 7 at 0x15e9
```

[phase_1]

phase_1 에 breakpoint가 걸렸을 때, disas 명령어를 사용하여, assembly 코드를 확인할 수 있다. # 0x555555556b70에 phase_1의 정답이 들어있다.

정답 비교 함수는 <strings_not_equal> 이다.

```
(gdb) disas
Dump of assembler code for function phase_1:
=> 0x00005555555552b4 <+0>:     sub    $0x8,%rsp
   0x00005555555552b8 <+4>:     lea    0x18b1(%rip),%rsi        # 0x555555556b70
   0x00005555555552bf <+11>:    callq  0x555555555841 <strings_not_equal>
   0x00005555555552c4 <+16>:    test   %eax,%eax
   0x00005555555552c6 <+18>:    jne    0x5555555552cd <phase_1+25>
   0x00005555555552c8 <+20>:    add    $0x8,%rsp
   0x00005555555552cc <+24>:    retq
   0x00005555555552cd <+25>:    callq  0x555555555b45 <explode_bomb>
   0x00005555555552d2 <+30>:    jmp    0x5555555552c8 <phase_1+20>
End of assembler dump.

(gdb) x/s 0x555555556b70
0x555555556b70: "You can Russia from land here in Alaska."
(gdb)

(gdb) run
Starting program: /home/sys059/202255513/HW2/bomb12/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
You can Russia from land here in Alaska.
Phase 1 defused. How about the next one?
```

*** phase_1 정답 : You can Russia from land here in Alaska.**

[phase_2]

phase_2의 어셈블리 코드를 확인해보자.

```
0x00005555555552ed <+25>:    callq  0x555555555b81 <read_six_numbers>
0x00005555555552f2 <+30>:    cmpl   $0x1,(%rsp)
0x00005555555552f6 <+34>:    jne    0x555555555301 <phase_2+45>
0x00005555555552f8 <+36>:    mov    %rsp,%rbx
0x00005555555552fb <+39>:    lea    0x14(%rbx),%rbp
0x00005555555552ff <+43>:    jmp    0x555555555311 <phase_2+61>
0x0000555555555301 <+45>:    callq  0x555555555b45 <explode_bomb>
```

위 코드에서 <read_six_numbers>를 통해 6개의 숫자를 입력해야 함을 알 수 있다. <+25>에서 6개의 숫자를 입력받아 stack에 저장하고, <+30>에서 첫 번째 입력 값이 1인지 확인한다. %rsp = $0x1 이 아니라면, <+34>에서 <+45>로 이동하여, <explode_bomb> 함수를 호출하게 되어 폭탄이 터진다.

```
0x00005555555552f8 <+36>:    mov    %rsp,%rbx
0x00005555555552fb <+39>:    lea    0x14(%rbx),%rbp
```

<+36>에서 %rbx에 %rsp값($0x1)을 넣어두고, <+39>에서 %rbx + 0x14 인 값을 %rbp에 넣는다. 이후, 코드에서 <+61>으로 이동하게 된다.

```
0x0000555555555308 <+52>:    add    $0x4,%rbx
0x000055555555530c <+56>:    cmp    %rbp,%rbx
0x000055555555530f <+59>:    je     0x555555555321 <phase_2+77>
0x0000555555555311 <+61>:    mov    (%rbx),%eax
0x0000555555555313 <+63>:    add    %eax,%eax
0x0000555555555315 <+65>:    cmp    %eax,0x4(%rbx)
0x0000555555555318 <+68>:    je     0x555555555308 <phase_2+52>
0x000055555555531a <+70>:    callq  0x555555555b45 <explode_bomb>
0x000055555555531f <+75>:    jmp    0x555555555308 <phase_2+52>
0x0000555555555321 <+77>:    mov    0x18(%rsp),%rax
0x0000555555555326 <+82>:    xor    %fs:0x28,%rax
0x000055555555532f <+91>:    jne    0x555555555338 <phase_2+100>
```

<+61>에서 (%rbx)의 값을 %eax에 넣은 후, <+63>의 코드를 보면, %eax = %eax + %eax이다. 해당 코드 부분에서 값을 2배하게 된다. int 자료형의 값이므로, %eax의 값과, (%rbx) + 0x4 위치의 값을 비교하여 값이 같다면, <+52>로 이동하게 되는 반복문의 형태를 띄게 된다.

*** phase_2 정답 : 1 2 4 8 16 32**

[phase_3]

```
0x000055555555535b <+30>:    lea    0x14(%rsp),%r8
0x0000555555555360 <+35>:    lea    0x185f(%rip),%rsi        # 0x555555556bc6
0x0000555555555367 <+42>:    callq  0x555555554f90 <__isoc99_sscanf@plt>
```

<+42>에서 scanf 함수가 있는 것으로 보아, 입력값을 받아 처리하는 것을 알 수 있다. 입력 값의 형태를 알기 위해서는, %rsi로 전달되는 0x555555556bc6의 내용을 통해 예측할 수 있다. x/s 명령어를 통해 확인해보면 아래와 같다.

```
(gdb) x/s 0x555555556bc6
0x555555556bc6: "%d %c %d"
```

두 개의 숫자와 하나의 문자를 입력값으로 필요하다는 것을 알게 되었다.

```
0x0000555555555371 <+52>:    cmpl   $0x7,0x10(%rsp)
0x0000555555555376 <+57>:    ja     0x55555555547e <phase_3+321>
0x000055555555537c <+63>:    mov    0x10(%rsp),%eax
```

<+52>에서 0x7보다 값이 큰 값이 들어온다면, <+321> explode_bomb으로 이동하여 폭탄이 터지게 된다. 따라서 첫 번째 값은 0~7 사이의 값이 들어와야한다.

```
0x0000555555555397 <+90>:    mov    $0x63,%eax
0x000055555555539c <+95>:    cmpl   $0x297,0x14(%rsp)
0x00005555555553a4 <+103>:   je     0x555555555488 <phase_3+331>
0x00005555555553aa <+109>:   callq  0x555555555b45 <explode_bomb>
0x00005555555553af <+114>:   mov    $0x63,%eax
0x00005555555553b4 <+119>:   jmpq   0x555555555488 <phase_3+331>
0x00005555555553b9 <+124>:   mov    $0x73,%eax
0x00005555555553be <+129>:   cmpl   $0x12c,0x14(%rsp)
0x00005555555553c6 <+137>:   je     0x555555555488 <phase_3+331>
0x00005555555553cc <+143>:   callq  0x555555555b45 <explode_bomb>
```

두 번째 입력값은 %eax로 0x63 값이 저장된다. 아스키코드로, 소문자 'c' 이고, 세 번째 입력값은 0x297로 숫자 '663'이다.

*** phase_3 정답 : 0 c 663**

[phase_4]

```
0x00005555555554fa <+25>:    mov    %rsp,%rdx
0x00005555555554fd <+28>:    lea    0x1989(%rip),%rsi        # 0x555555556e8d
```

입력 값의 형태를 알기 위해서는, %rsi로 전달되는 0x555555556e8d의 내용을 통해 예측할 수 있다. x/s 명령어를 통해 확인해보면 아래와 같다.

```
(gdb) x/s 0x555555556e8d
0x555555556e8d: "%d %d"
```

첫 번째 입력값이 0xe보다 작거나 같을 경우 <+56>으로 이동하여 폭탄이 터지지 않고 이동하게 된다.

```
0x000055555555550e <+45>:    cmpl   $0xe,(%rsp)
0x0000555555555512 <+49>:    jbe    0x555555555519 <phase_4+56>
```

<+79>에서 두 번째 입력값이 0xa가 아닐 경우, 폭탄이 터지므로 두 번째 입력값은 0xa 즉 10이 된다.

```
0x0000555555555530 <+79>:    cmpl   $0xa,0x4(%rsp)
0x0000555555555535 <+84>:    je     0x55555555553c <phase_4+91>
0x0000555555555537 <+86>:    callq  0x555555555b45 <explode_bomb>
0x000055555555553c <+91>:    mov    0x8(%rsp),%rax
```

func4 에서 나온 결과값이 0xa와 같아야 하므로, %edx = 14, %esi = 0, %edi = 첫 번째 입력값 임을 이용하여, func4의 결과값이 10일때의 첫 번째 입력값을 구하면 된다.

```
0x0000555555555519 <+56>:    mov    $0xe,%edx
0x000055555555551e <+61>:    mov    $0x0,%esi
0x0000555555555523 <+66>:    mov    (%rsp),%edi
0x0000555555555526 <+69>:    callq  0x5555555554ad <func4>
0x000055555555552b <+74>:    cmp    $0xa,%eax
0x000055555555552e <+77>:    jne    0x555555555537 <phase_4+86>
```

func4 내부 함수에서 <+24>에서 %eax가 10이 될때, 해당 phase를 해결할 수 있으므로, 적절한 값을 찾으면 %edi는 3이 되는 것을 알 수 있다.

```
0x00005555555554bf <+18>:    jg     0x5555555554c9 <func4+28>
0x00005555555554c1 <+20>:    cmp    %edi,%ebx
0x00005555555554c3 <+22>:    jl     0x5555555554d5 <func4+40>
0x00005555555554c5 <+24>:    mov    %ebx,%eax
0x00005555555554c7 <+26>:    pop    %rbx
0x00005555555554c8 <+27>:    retq
0x00005555555554c9 <+28>:    lea    -0x1(%rbx),%edx
0x00005555555554cc <+31>:    callq  0x5555555554ad <func4>
0x00005555555554d1 <+36>:    add    %eax,%ebx
0x00005555555554d3 <+38>:    jmp    0x5555555554c5 <func4+24>
0x00005555555554d5 <+40>:    lea    0x1(%rbx),%esi
0x00005555555554d8 <+43>:    callq  0x5555555554ad <func4>
0x00005555555554dd <+48>:    add    %eax,%ebx
0x00005555555554df <+50>:    jmp    0x5555555554c5 <func4+24>
```

*** phase_4 정답 : 3 10**

[phase_5]

앞서 풀었던 문제들과 마찬가지로, 입력값의 형태를 파악하면 정수 숫자 2개를 받고 있음을 알 수 있다.

```
(gdb) x/s 0x555555556e8d
0x555555556e8d: "%d %d"
```

```
0x0000555555555583 <+45>:    mov    (%rsp),%eax
0x0000555555555586 <+48>:    and    $0xf,%eax
0x0000555555555589 <+51>:    mov    %eax,(%rsp)
0x000055555555558c <+54>:    cmp    $0xf,%eax
0x000055555555558f <+57>:    je     0x5555555555c3 <phase_5+109>
```

위 코드에서, 첫 번째 입력 값이 0xf인 경우, <+109>로 이동하여 폭탄이 터지게 되므로 첫 번째 입력값의 마지막 bit는 0xf가 아니다. 그 뒤, %ecx와 %edx를 0으로 초기화한 후, lea 명령어를 이용하여 계산한 주소를 %rsi , 즉 array의 시작 주소로 넘겼음을 알 수 있다.

```
(gdb) x/32d 0x555555556c00
0x555555556c00 <array.3418>:    10      0       0       0       2       0       0       0
0x555555556c08 <array.3418+8>:  14      0       0       0       7       0       0       0
0x555555556c10 <array.3418+16>: 8       0       0       0       12      0       0       0
0x555555556c18 <array.3418+24>: 15      0       0       0       11      0       0       0
```

아래 코드는 반복문을 실행하는 코드로, array[%rax] 값이 %eax에 저장되며, %eax의 값이 15가 되어야 함을 알 수 있다.

```
0x00005555555555a2 <+76>:    add    $0x1,%edx
0x00005555555555a5 <+79>:    cltq
0x00005555555555a7 <+81>:    mov    (%rsi,%rax,4),%eax
0x00005555555555aa <+84>:    add    %eax,%ecx
0x00005555555555ac <+86>:    cmp    $0xf,%eax
0x00005555555555af <+89>:    jne    0x5555555555a2 <phase_5+76>
0x00005555555555b1 <+91>:    movl   $0xf,(%rsp)
0x00005555555555b8 <+98>:    cmp    $0xf,%edx
0x00005555555555bb <+101>:   jne    0x5555555555c3 <phase_5+109>
```

Array[6]에 15값이 저장되어 있으므로, %rax는 6이라고 가정을 한 뒤, 첫 번째 값에 집어넣으면 다음 명령어에서 %edx의 값이 15가 맞는지 확인하는 과정을 거치게 된다. %edx의 값은 처음 0으로 초기화된 이후 <+76>으로 돌아가는 과정이 있으므로, %edx와 %rax 값 모두 15를 만족시키는 첫 번째 입력값은 1, 2, 3, 4 ... 로 바꾸어가며 실행하면, 입력값이 5임을 알 수 있다. %rbx와 %rcx를 통해 두 번째 숫자가 저장되어 있는 %rsp+4의 값이 115라는 것을 확인할 수 있으므로 정답은 아래와 같다.

*** phase_5 정답 : 5 115**

[phase_6]

<read_six_numbers>

```
(gdb) x/20x 0x555555758230
0x555555758230 <node1>: 0x00000242      0x00000001      0x55758240      0x00005555
0x555555758240 <node2>: 0x000002fe      0x00000002      0x55758250      0x00005555
0x555555758250 <node3>: 0x00000309      0x00000003      0x55758260      0x00005555
0x555555758260 <node4>: 0x0000035d      0x00000004      0x55758270      0x00005555
0x555555758270 <node5>: 0x000003c7      0x00000005      0x55758110      0x00005555
(gdb) x/4x 0x5575811000005555
0x5575811000005555:     Cannot access memory at address 0x5575811000005555
(gdb) x/4x 0x0000555555758110
0x555555758110 <node6>: 0x00000297      0x00000006      0x00000000      0x00000000
```

node5의 오른쪽 두개의 주소값을 조합하여 검색을 해보면, node6 주소값 또한 확인할 수 있다. 6개의 node 주소 순서를 확인하면, 0x00000242(1), 0x00000297(6), 0x000002fe(2), 0x00000309(3), 0x0000035d(4), 0x000003c7(5)가 답이 된다.

*** phase_6 정답 : 1 6 2 3 4 5**