**#Phase\_1**

|  |
| --- |
| **Phase\_1 어셈블리어 표현** |
|  |
| **확인해보면 Phase\_1+15 문장에서 strings\_not\_equal 함수를 불러오고 eax레지스터 값을 test후 eax가 0이 아니면 explode\_bomb 함수를 불러 bomb이 폭발하게 된다.**  **해결을 위해 strings\_not\_equal 함수의 어셈블리어 표현을 확인해보자** |
| **Strings\_not\_equal 어셈블리어 표현** |
|  |
|  |
|  |
| **먼저 함수에서 rdi와 rsi를 입력받는것으로 보아 함수의 파라미터는 rdi와 rsi에 저장되어있다고 예상할수 있다.**  **더욱 자세하게 String\_not \_equal 함수를 분석하기 위해서는 string\_length함수부터 확인해 볼 필요가 있다.** |
| **String\_length 함수 어셈블리어 표현** |
|  |
| **먼저 rdi레지스터를 cmp하는것으로 보아 우리가 입력 값이 저장되어있는 주소 값이 rdi에 입력된다고 예상할 수 있다.**  **+4에서는 입력 값이 없으면 0을 리턴 한다고 나와있다.**  **그 후 +14~+21까지 반복하는 문장으로 생각할 수 있다. 그 내용은 입력 값 저장 주소에 1을 더하고 결과값을 하나 더하고 그 자리가 비어있는지 확인하며 비어있을 때 까지 반복한다.**  **즉 이름과 같이 입력 받은 문자열의 길이를 반환하는 함수이다.** |
| **Strings\_not\_equal 함수 분석** |
| **먼저 +14~+32까지의 과정은 rsi와 rdi에 있는 문자열의 길이를 확인하고 비교하는 과정이다 이때 비교 결과가 다르면 바로 함수는 끝나게 되고 rax값은 rsi에 저장되어 있는 문자열의 길이가 된다. 즉 두 문자열이 다르면 0이 아닌 다른 값을 리턴 한다는 것을 예상할 수 있다.** |
| **레지스터 값 확인 (rdi & rsi)** |
|  |
| **rdi레지스터에는 input\_strings 즉 입력한 hi가, rsi에는 I am just a renegade hockey mom이 저장되어있는 것을 확인 할 수 있다. 그러므로 우리는 input에 rsi에 저장되어있는 값을 입력하려 strings\_not\_equal 리턴값이 0이게 해야 함으로 첫 번째 입력 값은 I am just a renegade hockey mom. 이 된다.** |
| **실행 결과** |
|  |

**#Phase\_2**

|  |  |  |
| --- | --- | --- |
| **Phase\_2 어셈블리어 표현** | | |
|  | | |
| **+26까지 입력받을 변수의 저장공간을 확보하고 변수를 저장하는 내용이다.**  **Phase\_2를 해석하기 위해서는 +29의 read\_numbers 함수 먼저 해석해볼 필요가 있다.** | | |
| **Read\_numbers 함수 어셈블리어 표현** | | |
|  | | |
|  | | |
| **Read\_number+4 ~ + 34까지의 레지스터 데이터 저장 결과**   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **rdx** | **rcx** | **Stack1** | **Stack2** | **Stack3** | **R9** | **R8** | | **Rsi값** | **rsi주소+4의 값** | **rsi주소+18의 값** | **rsi주소+14의 값** | **rsi주소+10의 값** | **rsi주소+c의 값** | **rsi주소+8의 값** |   **즉 rsi값에 대해 여러 값을 레지스터에 저장하고 rip의 주소+0x14dd의 값을 rsi에 넣고 그 후 리턴값을 0으로 설정하였다.**  **그래서 read\_numbers+38까지 ni 명령어를 통해 진행 후 rip의 주소+0x14dd의 값을 확인해 보았다.** | | |
| **read\_numbers+38까지 ni 명령어를 통해 진행 후 rip의 주소+0x14dd의 값 확인** | | |
|  | | **마지막을 보면 정수형 7개가 저장되어 있는 것을 볼 수 있다. 즉 이번 phase\_2의 입력 값은 정수형 7개로 확인가능하다.** |
| **다시 실행 후 입력값에 1 2 3 4 5 6 7 입력 후 read\_numbers 까지 진행 후 레지스터값 확인** | | |
|  | **Phase\_2의 어셈블리어 표현을 다시 확인해보자 +29번 줄에서 read\_numbers 실행 후 rsp값을 비교를 하여서 rsp을 확인해보고 내가 입력한 1 2 3 4 5 6 7중 가장 앞에 숫자가 나오자 나머지 rsp+18까지 진행 해 보았다.**  **그랬더니 1 2 3 4 5 6 a가 저장이 되어있다.** | |
| **이후 과정 어셈블리어 확인** | | |
| **+34 에서 (%rsp)와 0x0을 비교해 (%rsp)가 작거나 같으면 폭탄이 터지게 된다. 즉 첫번째 숫자는 1이상이어야 한다.**  **그 후 +40~+48 & +69~79까지 각각의 레지스터에 저장 값은 이렇게 된다.**   |  |  |  |  | | --- | --- | --- | --- | | **Rbp** | **Ebx** | **Edx** | **Eax** | | **Rsp 주소의 값** | **1** | **1** | **0** |   **그리고 점프문에서 test ebx ebx 는 ebx&ebx 과 같은 방식으로 동작한다. 즉 ebx&ebx = ebx가 0이거나 음수일 경우 점프를 수행한다.**  **+83~+88까지의 레지스터 저장 값은**   |  |  |  |  | | --- | --- | --- | --- | | **Rbp** | **Ebx** | **Edx** | **Eax** | | **Rsp 주소의 값** | **1** | **1\*2** | **0+1** |   **그리고 점프 문에서 ebx와 eax가 같으면 점프수행을 안하니 +92로 넘어가게된다.**  **그후 +92를 수행하고 다음 레지스터 저장값은**   |  |  |  |  | | --- | --- | --- | --- | | **Rbp** | **Ebx** | **Edx** | **Eax** | | **Rsp 주소의 값** | **1** | **1\*2+rsp주소의 값** | **0+1** |   **이 된다 즉 edx에는 2+rsp값이 된다. 이때 +95에서는 2+rsp 가 rsp주소+4의 값과 다르면 폭탄이 터지게 된다. 즉 두번 째 숫자는 첫 번째 숫자 +2가 된다.**  **다시 +57로 점프하고 +64까지 진행하면 레지스터 저장 값은**   |  |  |  |  | | --- | --- | --- | --- | | **Rbp** | **Ebx** | **Edx** | **Eax** | | **Rsp+4 주소의 값** | **1+1** | **1\*2+rsp주소의 값** | **0+1** |   **그 후 ebx가 7이 되기 전까지는 +107로 점프를 안하고 계속 진행을 하게 된다.**  **+69~+79까지 수행후 레지스터 저장 값은**   |  |  |  |  | | --- | --- | --- | --- | | **Rbp** | **Ebx** | **Edx** | **Eax** | | **Rsp+4 주소의 값** | **1+1** | **1** | **0** |   **이고 ebx가 1이상이므로 +92로 점프 수행 안하고 계속 진행한다.**  **+83~+88까지의 레지스터 저장 값은**   |  |  |  |  | | --- | --- | --- | --- | | **Rbp** | **Ebx** | **Edx** | **Eax** | | **Rsp+4 주소의 값** | **1+1** | **1\*2** | **0+1** |   **이 되고 eax가 ebx와 같아질때까지 +83~+88을 반복하게 된다. 이 결과는**   |  |  |  |  | | --- | --- | --- | --- | | **Rbp** | **Ebx** | **Edx** | **Eax** | | **Rsp+4 주소의 값** | **1+1** | **1\*2\*2** | **0+1+1** |   **가 된다.**  **그후 +92~+95까지 진행 후 레지스터 저장 값은**   |  |  |  |  | | --- | --- | --- | --- | | **Rbp** | **Ebx** | **Edx** | **Eax** | | **Rsp+4 주소의 값** | **1+1** | **1\*2\*2+rsp+4 주소의 값** | **0+1+1** |   **이 된다.**  **결론은 이 과정을 총 6번 반복하여 첫번째 숫자부터 일곱번째 숫자까지 모두 비교를 하게 된다. 이때 n번쨰의 숫자와 n+1번째의 숫자와 차이는 2^n승만큼의 차이이다.**  **즉 n번쨰의 숫자는 (2^n)-1이 된다.**  **결국 입력 해야 하는 값은1 3 7 15 31 63 127이 된다.** | | |
| **실행 결과** | | |
|  | | |

**#Phase\_3**

|  |  |
| --- | --- |
| **Phase\_3 어셈블리어 표현** | |
|  | |
|  | |
|  | |
|  | |
| **우선 46번째 끼지는 phase\_2처럼 입력받은 값을 저장하기 위한 과정이기 때문에 그때까지 진행하고 rsi의 레지스터에 있는 값을 확인해보자.** | |
| **입력 해야 하는 값 확인** | |
|  | |
| **진행 후 우리가 입력해야 하는 값은 “정수 문자 정수” 꼴임을 확인할 수 있다.** | |
| **+46 문장에서 sscanf 함수 호출후 결과인 eax는 우리가 입력한 자료의 개수다. 즉 2개 이하를 입력하면 +51에서 cmp후 +54에서 점프를 하게 되어 폭탄이 터지게 된다.**  **우리가 입력한 값들이 어디에 저장되어있는지 확인 하기 위해 “정수 문자 정수” 꼴을 입력하고 다시 실행해보자.** | |
| **“1 a 2”꼴 입력 후 입력 값 저장되어있는 레지스터 찾기** | |
| **먼저 아래를 확인하면 비교 인자가 0x14(%rsp)와 0x10(%rsp)이므로 두 개의 레지스터에 들어있는 값을 확인해 보았다.** | |
|  | **예상대로 0x10(%rsp) 에는 첫 번째 입력 값의 주소**  **0x14(%rsp) 에는 세 번째 입력 값의 주소가 들어있었다. 아직 중간에 있는 문자가 저장된 레지스터는 찾지 못하였다.** |
| **이후 과정 어셈블리어 확인** | |
| **+56~+63 까지는 첫 번째 입력 값에 대한 cmp 과정이다.**  **+60까지 진행하게 되면 eax레지스터의 값에는 첫 번째 입력 값 - 0x1b가 저장되어있다. 또한 점프 문에서 점프를 하지 않기 위해서는 eax의 저장 값이 7과 같아야 한다. 즉 첫 번째 입력값은 34이다.**  **이후 +72~+88 까지의 과정은 단순히 +312로 점프하기 위한 과정이다.**  **그리고 +312~+317과정까지 진행하면 세 번째 입력 값 즉 두 번째 정수 값은 0x3a5와 같지 않으면 폭탄이 터지니 세번째 입력값은 0x3a5를 10진수로 변환한 933이다.**  **그 후 점프 문을 통해 +349를 가서 %al 과 rsp+0xf를 비교하는 연산이다. 여기서 rsp+0xf가 두 번째 입력 값일 것 같아 확인 해본 결과**  **우리가 입력한 a가 나왔다.**  **그리고 +312~+317동안 rax에는 0x69가 복사가 되었고 즉 al에는 0x69의 값이 들어있다. 이것을 아스키 코드로 변환하면 ‘i’가 된다. 즉 두 번째 입력 값은 i이다.** | |
| **실행 결과** | |
|  | |

**#Phase\_4**

|  |  |
| --- | --- |
| **Phase\_4 어셈블리어 표현** | |
|  | |
| **Phase\_4도 Phase\_1~3까지와 비슷하게 +0~+32까지 입력 값을 저장 할 공간을 확보 하고 +39에서 입력 값을 불러온다. 우선 +39까지 진행 후 rsi 레지스터에 있는 값을 확인해보면 우리가 입력 해야 할 값들의 형태를 알 수 있을 것이다.** | |
| **입력해야 하는 값 확인** | |
|  | |
| **확인 결과 정수 2개를 입력해야 한다.** | |
| **정수 2개(1 2) 입력 후 입력 값 저장 위치 찾기`** | |
| **우선 +39에서 sscanf 함수 호출 후 eax값에는 입력 값의 개수인 2가 저장되어 +47 점프 문을 통해 +55로 점프 수행은 안 하게 된다.**  **이후 아래쪽 어셈블리어를 확인해보면 0x4(%rsp) 그리고 (%rsp)에 대한 비교 문장이 있는 것을 보아 우리가 입력한 값은 이 두 곳에 저장되어있음을 예측할 수 있다.** | |
|  | **예상대로 %rsp에 첫 번째 입력 값 1이**  **0x4(%rsp)에 두 번째 입력 값 2가 저장되어있다.** |
| **이후 과정 어셈블리어 확인** | |
| **우선 +49~+55를 확인해보면 첫 번째 입력 값은 16진수0xe와 비교를 하였을 때 같거나 작아야 한다. 즉 첫 번째 입력 값은 14이하이어야 한다.**  **그 후 +60~ +70까지의 레지스터에 저장되는 값은 아래와 같다.**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **(%rsp)** | **0x4(%rsp)** | **%edx** | **%esi** | **%edi** | | **첫 번쨰 입력 값** | **두 번쨰 입력 값** | **0xe** | **Ox0** | **첫 번째 입력 값** |   **이후 +73에서 func4를 호출하는데 이 함수가 어떤 기능을 하는지 확인 할 필요성이 있다.** | |
| **Func4 어셈블리어 표현** | |
|  | |
| **+0~+20까지 진행 후 ebx값과 edi값에 따라 다른 점프 문으로 이동해 각각 다른 레지스터 값을 변경 시킨 후 ebx와 edi값이 같아질 때 까지 반복한다.**  **이때 eb x > edi일 경우에는 edx에rbx-1이 저장되고**  **반대로 ebx < edi일 경우에는 esi에rbx+1이 저장된다**  **그 후 call을 통해 반복할 때 마다 나왔던 rbx값을 모두 더 한 값을 리턴 값으로 가지게 된다.**  **처음 func4 호출 후 +0~+22까지 진행 후 레지스터 저장 값은**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **eax** | **ebx** | **edx** | **esi** | **Edi** | | **14** | **7** | **14** | **0** | **첫 번째 입력 값** |   **Ebx>edi일경우 새로운 func4호출 후 +0~+22 까지 진행 후 레지스터의 저장값은**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **eax** | **ebx** | **edx** | **esi** | **Edi** | | **14-(이전Rbx-1)** | **(14+이전Rbx-1)/2** | **14** | **이전Rbx-1** | **첫 번째 입력 값** |   **Ebx<edi일경우 새로운 func4호출 후 +0~+22 까지 진행 후 레지스터의 저장값은**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **eax** | **ebx** | **edx** | **Esi** | **Edi** | | **14-(이전Rbx-1)** | **(14+이전Rbx+1)/2** | **14** | **이전Rbx+1** | **첫 번째 입력 값** |   **위에 경우 모두 ebx 계산결과 소수부분은 버린다.**  **이제 가능한 ebx값을 구하면 ( 편의상 세 번째 까지만 구하겠다. )**   |  |  |  | | --- | --- | --- | | **첫 번째** | **두 번째** | **세 번째** | | **7** | **10, 11** | **11(두번째가 10), 12(두번째가 10or11)**  **13(두번째가 11)** | | |
| **Phase\_4 이후 과정 어셈블리어 확인** | |
| **Func4가 리턴 한 후 +78에서 리턴 값 과 0x1f를 비교하고 이때 같지 않으면 폭탄이 터지게 된다 . 여기서 rax값이 0x1f가 되게 하려면 func4 과정 반복 중 나와야 하는 rbx값은 7,11,13 이다. 즉 rbx 가 13일 때 반복을 멈춰야 하며 즉 첫 번째 입력 값은 13이어야 한다는 의미이다.**  **이후에 또 0x4(%rsp)와 0x1f를 비교하고 같지 않으면 폭탄이 터지고 같으면 정상적으로 phase\_4가 끝나게 된다. 즉 두 번째 입력은 31이어야한다.** | |
| **실행 결과** | |
|  | |

**#Phase\_5**

|  |
| --- |
| **Phase\_5 어셈블리어 표현** |
|  |
| **먼저 string\_length와 strings\_not\_equal함수들은 Phase\_1에서 확인한 것과 같이 각각 문자열의 길이를 리턴 하고 문자열이 같을 경우에만 0을 리턴 하는 함수이다.**  **+38에서 eax와 6이 같아야 123으로 점프를 수행 안 해 폭탄이 안 터진다. 즉 우리가 입력 해야 할 문자열의 길이는 6글자이다.**  **이후 +38~+43은 +50~+73까지의 반복 문 수행 전 변수 설정 하는 과정이다. +65와 +69를 확인해 보면 rax레지스터 값은 반복 문 내에서 +65에서만 바뀐다. 즉 반복 문은 총 6번 반복을 한다.**  **+50을 확인하면 (rbx,rax,1)은 rbx문자열에서 rax번쨰의 문자를 나타낸다.**  **그리고 +54는 rbx문자열에서 rax번째 문자와 0xf와 & 연산한 결과이다. 즉 문자의 아스키 코드와 0xf와 &연산을 하였다는 의미로 0~0xf까지의 값을 가질 수 있다.**  **그 후 (%rcx,%rdx,1)을 edx에 입력하고 dl을 각각 rsp+rax+1에 저장하는 과정이다.**  **마지막으로 반복이 끝나면 +75에서 rsp+7 즉 마지막에 null값을 만들어 준다.**  **그러고 난 후 반복 문을 통해 만든 rsp와 0x1886(%rip)문자열을 비교하여 같으면 폭탄이 안 터지는 Phase이다.**  **이를 해결하기 위해서는 rsp에 저장되는 문자열을 확인 해야 하고 그러기 위해서는 rcx가 무엇인지 확인을 해야 한다.**  **또한 0x1886(%rip)의 문자열도 확인해보자** |
| **0x1886(%rip) 문자열 확인 ( 확인을 위해 strings\_not\_equal 까지 함수 진행 )** |
|  |
| **Rcx 확인후 rsp문자열 만들기** |
|  |
| **위에서 설명하였듯이 rsp+rax+1 즉 rsp+1~+6까지에는 (%rcx,rdx,1)에서 가장 작은 1바이트를 넣는다. 즉 rcx문자열 앞에서 rdx+1번째 문자를 rsp에 넣는다고 할 수 있다.**  **Rsp+1부터 +6까지 flames를 차례대로 넣기 위해서 필요한 rdx값은 각각 아래와 같다.**  **rdx값은 또한 우리가 입력한 문자열의 문자의 아스키 코드와 0xf와의 &연산 결과이다.**   |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **0x1886(%rip)** | **f** | **l** | **a** | **m** | **e** | **S** |  |  |  |  |  |  | | **Rdx값** | **0x9** | **0xf** | **0x1** | **0x0** | **0x5** | **0x7** |  |  |  |  |  |  |   **즉 우리가 입력해야 할 값은 아스키 코드를 16진수로 나타내었을 때 일의 자리가 rdx의 값과 같은 문자이다. 편의상 아스키 코드 범위가 0x61~0x7a인 영어 소문자에서 고르면**   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **Rdx값** | **0x9** | **0xf** | **0x1** | **0x0** | **0x5** | **0x7** | | **문자의 아스키코드** | **0x69** | **0x6f** | **0x61** | **0x70** | **0x65** | **0x67** | | **문자** | **i** | **o** | **a** | **p** | **e** | **G** |   **이 된다. 즉 우리가 입력 해야 할 문자는 ioapeg이다.** |
| **실행 결과** |
|  |

**#Phase\_6**

|  |  |
| --- | --- |
| **Phase\_6 어셈블리어 표현** | |
|  | |
|  | |
|  | |
| **Phase\_2와 같은 read\_numbers 함수를 호출해 입력한 값을 받아 확인을 한다. 즉 7개의 정수 입력이 필요한 것을 알 수 있고 각각의 숫자들의 주소는 아래와 같이 저장된다.**   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **%rsp**  **=%r13** | **%rsp+0x4**  **=%r13+0x4** | **%rsp+0x8**  **=%r13+0x8** | **%rsp+0xc**  **=%r13+0xc** | **%rsp+0x10**  **=%r13+0x10** | **%rsp+0x14**  **=%r13+0x14** | **%rsp+0x18**  **=%r13+0x18** | | **첫 번째** | **두 번째** | **세 번째** | **네 번째** | **다섯 번째** | **여섯 번째** | **일곱 번째** |   **여기서 %rsp에 저장되어 있는 주소값이 read\_numbers 이전에 %r13에 mov 인스트럭션을 실행해서 rsp=r13이다.**  **이후 과정에서 +94로 점프 수행하고 +97에서 첫 번째 숫자가 eax에 넣어지고 eax-1의 값이 6을 초과하면 폭탄이 터지게 된다. 즉 첫 번째 숫자는 7이하이다.**  **그 후 과정을 진행하다 보면 +66 부터 +84 까지 6번을 반복하게 된다. 이 과정은 현재 rbp에 들어있는 값에 대해 입력 값과 중복이 있으면 폭탄이 터진다. 첫 번째 시도 때 rbp에는 첫번 쨰 입력 값이 들어있다. 이후 중복이 없으면 +86부터 +118까지 진행하여 rbp에는 그 다음 입력 값이 들어 오게 된다. 즉 +66부터 +118까지는 입력 값에 대해 각각 중복이 있는지 확인하는 작업이다. 이때 중복이 있으면 폭탄이 터지므로 입력 값은 중복이 없어야 한다. 모두 중복 확인을 한 후 없게 되면 +113 점프문을 통해 +120으로 진행하게 된다.** | |
| **+120 이후 어셈블리어 확인** | |
| **+120 이후 과정에 ecs에는 rsp+rsi\*4의 값을 가지는데 이것의 형태를 보아서 각각의 입력 값에 대한 반복 문임을 예측 할 수 있다. 그 후 rdx에는 node1의 주소를 넣어주고 ecx와 1을 비교해 두가지의 점프 문으로 나눠진다.**  **우선 ecx가( = n번째 숫자의 크기 ) 1보다 클 경우 각 문장에 따라 rdx와 각 레지스터의 값은 아래와 같다.**   |  |  |  | | --- | --- | --- | | **+120~+140** | **+145~+154** | **+156~+165** | | **Rdx=node의 주소**  **Eax=n( = 입력 값의 순번 )** | **Rdx = node의 주소 +8\*(n번째 숫자의 크기)** | **Rsp+20+8\*(n-1) = rdx**  **여기서 n은 입력 값의 순번** |   **그리고 ecx가 1 보다 작거나 같을 경우 각 문장에 따라 rdx의 값은 아래와 같다.**   |  |  | | --- | --- | | **+120~+140** | **+156~+165** | | **Rdx=node의 주소**  **Eax=n( = 입력 값의 순번 )** | **Rsp+20+8\*(n-1) = rdx**  **여기서 n은 입력 값의 순번** |   **그러므로 node의 값이 무엇인지 그리고 node+8n의 값은 무엇인지 확인할 필요가 있다.**  **( 실행 결과 +8이 아니라 +0x10이 되었습니다.. 왜 그러는지는 잘 모르겠습니다…. )** | |
| **+165 이후 어셈블리어 확인** | |
| **+171~+207과정을 보면 +189~+207 까지 반복 문이 진행되고 있음을 확인할 수 있다. 그 과정을 자세히 보면 순서대로 rcx+8\*(n-1) 레지스터에 Rsp+20+8\*(n-1)의 값을 입력하는 과정이다.**  **그리고 나서+209~+241또한 반복 문이 진행됨을 확인할 수 있다. +239에서 비교하는 레지스터를 확인해 보면 (rbx) – (rbx+8) 을 진행해서 결과가 0이하이면 폭탄이 터지게 된다. 그리고 rbx는+171번 문장에서 초기에 rsp+20으로 설정이 되어있었고 반복문을 진행할수록 +8이 된다. 즉 결론은**  **Rsp+0x20 > Rsp+0x28 > Rsp+0x30 > Rsp+0x38 > Rsp+0x40 > Rsp+0x48 > rsp+0x50 이다.**  **그리고 이 식은 우리가 입력 입력한 숫자들을 node로 변환하고 그 변환 값들이 순번에 따라 작아져야 한다. 편의상 1 2 3 4 5 6 7을 입력하고 각각의 변환 값을 확인해 보자.**  **( 실행 결과 변환 값은 입력 숫자가 n일 경우 node n 으로 변환 되었습니다. 각 node값은 +120~ +165 반복 문 진행하면서 각각 저장 값들을 확인했습니다.)** | |
| **1 2 3 4 5 6 7 입력 후 저장 값 확인** | |
|  | **위에서부터 차례대로 1~7을 변환한 값들이다. 입력 순번 차례대로 큰 숫자부터 작은 숫자로 이어져야 하므로 우리가 입력해야 하는 값은**  **6734512 이다.** |
|  |
| **실행결과** | |
|  | |

**#Phase\_secret**

|  |  |  |
| --- | --- | --- |
| **main함수 어셈블리어 표현** | | |
|  | | |
| **각 phase가 끝날때마다 phase\_defused 함수가 호출이 되는 것을 확인할 수 있다. 그래서 phase\_secret이 defused에 있다고 예상을 하였다.** | | |
| **Phase\_defused 함수 어셈블리어 표현** | | |
|  | | |
| **Secret\_phase를 호출하기 위해서는 +41문장에서 점프를 수행 후 또한 +110에서도 점프를 수행해야 한다. 그리고 +157에서는 점프를 수행하면 안된다. 먼저 +41점프 조건을 확인하기 위해 phase\_defused에 b 옵션을 주어서 +41에서 x/d $rip+0x3622 확인을 해보자** | | |
| **x/d $rip+0x3622 값 확인** | | |
| **각 페이스마다 phase\_defused에 b 옵션을 주어서 +41에서 x/d $rip+0x3622 확인결과 페이스 진행마다 반환 값이 1이 증가했다. 즉 페이스 6까지 해결을 하면 자동으로 +41문장에서 점프를 수행하게 된다.** | | |
| **이후 과정 어셈블리어 확인** | | |
| **+102 에서 sscanf함수를 호출하고 그 반환값이 3이어야 secret\_phase를 호출 할 수 있다. 우선 진행후 scanf의 파라미터인 rdi와 rsi를 확인해보자** | | |
| **Rdi와 rsi 파라미터 확인 결과** | | |
|  | | **rdi에는 phase\_4의 입력인 13 31이 있었다. 즉 우리는 phase\_4의 입력에 뒤에 하나의 입력을 추가해야한다.** |
| **이후 +138 문장으로 넘어가면 phase\_4입력값을 rdi에 저장하고 rsi에는 어떤 값을 저장하고 비교를 수행해서 같을 경우에 secret\_phase를 호출할수있으므로 rsi에 저장하는 값을 확인해보자** | | |
| **0x55555555745b 값 확인** | | |
|  | **즉 phase\_4의 답에 EvilProfBomb을 입력후 진행을 하면 secret phase 진입 가능하다.** | |
| **Secret\_phase 어셈블리어 표현** | | |
|  | | |
| **먼저 사용 되는 함수는 read\_line 함수와 strtol 그리고 fun7이다. 그리고 폭탄이 안 터지게 하기 위해서는 +39와 +58에서 점프를 수행하면 안된다. 먼저 각 함수들의 반환 값에 대한 정보를 확인하기 위해 secret\_phase에 대한 입력 값 123을 넣어주고 각 함수들의 리턴 값을 확인해보자.** | | |
| **Read\_line 함수 호출후 반환값 확인** | | |
|  | | |
| **확인 결과 read\_line함수는 secret\_phase에 대한 입력 값을 반환한다.** | | |
| **Strtol 함수 호출 후 반환 값 확인** | | |
|  | | |
| **확인 결과 strtol 함수는 우리가 입력한 값을 정수 형으로 변환시켜준다.**  **그 후 +31에서 그 값에 -1 한 후 (입력값 -1 )-0x3e8의 결과가 0을 초과하면 폭탄이 터지게 된다. 즉 (입력값-1)-1000 =<0 이어야 하므로 입력 값은 1000이하이어야한다. 그리고 나서 +28과 +41을 통해 입력 값은 rsi로 전달이 되고 fun7의 파라미터가 된다.** | | |
| **Fun7 어셈블리어 표현 확인** | | |
|  | | |
| **확인을 해보면 rdi값과 입력값의 크기 비교에 따라 +17과 +24 둘중 하나의 점프문을 수행하게 된다.**  **우선 +17 점프 문을 수행하면 즉 rdi > rsi일경우 rdi = 0x8(rdi) 명령을 실행후 다시 fun7을 호출한다. 여기서 호출한 함수가 반환 값을 가지게 되면 최종 반환 값은 직전에 호출한 함수의 반한 값의 2배이다.**  **그리고 +24 점프 문을 수행하면 즉 rdi-rsi=/0 ( rdi < rsi 와 같다. ) 일경우 rdi = 0x10(rdi) 명령을 실행 후 다시 fun7 을 호출한다. 여기서 호출한 함수가 반환 값을 가지게 되면 최종 반환 값은 직전에 호출한 함수의 반환 값\*2에 1을 더한 값이다.**  **마지막으로 rdi = rsi 일 경우 반환 값은 0을 가지고 리턴을 수행한다.**  **우리가 원하는 반환 값은 2 이므로 처음 fun7함수 호출 시 +17로 점프를 수행해서 다시 fun7함수를 호출하고 그 함수 안에서 +24점프를 수행해서 다시 fun7을 호출해서 이번 함수 안에서는 반환값을 0으로 가지게 되면 최종 반환값은 2가 된다.**  **이제 각각 수행마다 rdi의 값을 확인하고 비교하면서 답을 넣어주면 끝이다.**  **아래는 각 함수 호출마다 확인되는 rdi 값이다.**   |  |  | | --- | --- | |  | **처음 fun7 함수 호출시 36이므로 우리가 입력해야 할 값은 36 미만이다.** | |  | **두 번째 fun7 함수 호출시 rdi값이 8이므로 우리가 입력해야 할 값은 8 초과이다.** | |  | **마지막 fun7 함수 호출시 rdi 값이 26 이므로 우리가 입력해야 할 값은 22이다.** | | **결국 마지막 secret\_phase의 답은 22가 된다.** | | | **수행결과** | | |  | | | | |