20160074 화학공학과 고진민

[컴퓨터 SW시스템 개론](http://lms.postech.ac.kr/Main.do?cmd=viewCourseMain&mainDTO.courseId=20180920140072901661213&gubun=class_learner) HW2

1. (3.64)
2. **&A[i][j][k] = xA + L(i \* S \* T + j \* T + k)** 이다. 여기서 xA는 A[0][0][0]의 주소이고, L는 data size이다. 이 경우는 long data형이므로 8byte이다. S, T는 미리 정해진 상수이다.
3. Line 2에서 rax = 3\*rsi를 수행한다. Line 4에서 rax = rsi + 4\*rax를 수행하므로 이 때 rax에 저장된 값은 13\*j라고 할 수 있다. 이후 line 4~5에서는 rdi를 6bit left shift 한다. 이 상황에서 rsi에 저장된 값은 64\*i 라고 할 수 있다. Line 6에서 rdi = rdi+rsi를 수행하므로 rdi에는 최종적으로 65\*i가 저장됨을 알 수 있다. Line 7에서 rdi에 rax를 더하면 rdi에는 65\*i + 13\*j가 저장된다. 우리가 A에서 구한 식으로 미루어보아 T = 13, S = 5임을 알 수 있다. 다음으로 line 8에서 rdx = rdx + rdi 이므로 최종적으로 65\*i + 13\*j + k가 rdx에 저장된다. Line 11을 보면 3640이 A의 size로 반환된다. 이를 65로 나누면 56이 나오고, 우리는 long 형 변수가 4 byte임을 알고 있다. 하지만 저장이 8 byte로 되어 있으므로 이를 다시 8로 나눈 값인 7이 R이 됨을 알 수 있다. 따라서, **R = 7, T = 13, S = 5** 이다.
4. (3.67)

|  |  |
| --- | --- |
|  | <- rsp + 104 |
|  |  |
|  |  |
|  |  |
| z (8 byte) | <- rsp + 24 |
| &z (8 byte) | <- rsp + 16 |
| y (8 byte) | <- rsp + 8 |
| x (8 byte) | <- rsp |



여기까지가 함수 eval에서 함수 process를 호출하기 전에 저장되는 데이터이다.

1. Line 8에서 rdi로 넘겨주는 값은 rsp+64 이다. 즉, 주소값을 점겨주었다.
2. Rsp에 저장된 주소를 이용해 접근한다. Rsp dependent하게 접근을 수행하고 있다.
3. r에는 다음과 같이 데이터가 저장된다. r.u[0]에는 s.a[1]이, r.u[1]에는 s.a[0]이 저장된다. 여기에 r.q에 s.p의 주소를 역참조해 해당 값, eval에서는 z에 해당하는 값을 저장한다.

|  |  |
| --- | --- |
|  | <- rsp + 104 |
| … |  |
| z | <- rsp + 80 |
| x | <- rsp + 72 |
| y | <- rsp + 64 |
| … |  |
| z (8 byte) | <- rsp + 24 |
| &z (s.p, 8 byte) | <- rsp + 16 |
| y (s.a[1], 8 byte) | <- rsp + 8 |
| x (s.a[0], 8 byte) | <- rsp |

Process에서 어떤 과정이 일어나는지 살펴보자. 먼저 function call이 일어났으므로 return address 와 old rbp가 push 되어 rsp가 8byte 만큼 감소한 상태이다. 기존에는 rdi에 rsp+64에 해당하는 주소가 저장되어 있었다. 여기에 저장된 주소값은 감소하기 전의 rsp이다. Line 2에서 이 주소를 rax로 옮긴다. Line 3~4에서 rdx에 z를 옮긴다. Line 5에서 rcx에 y를 옮긴다. Line 6에서 rcx의 값을 rdi가 가리키고 있는 위치, 즉, rsp+64에 넣는다. Line 7에서 rcx에 x를 옮긴다. Line 8에서 rcx의 값을 rdi+8이 가리키고 있는 위치, 즉, rsp+72에 넣는다. Line 9에서 rdx의 값인 z를 rdi+16이 가리키고 있는 위치, 즉, rsp+80에 넣는다.

1. Return은 rax에 그 value가 저장되고, 함수에 넘겨주는 값은 보통 rdi에 저장된다. 이외에도 주소를 이용하여 각종 value에 접근해 사용이 가능하다. Rdi로 주소를 전송하면 이 값을 이용해 argument들에 접근이 가능하다. Struct의 경우는 한 종류의 데이터만을 넘기는 것이 아니기 때문에 주소를 넘겨서 struct를 구성하는 다양한 값들에 접근이 가능하도록 해야 한다.
2. (3.68)

Struct의 경우 입력 순서대로 데이터가 저장된다. Long이 있으므로, str2를 구성하는 데이터 중 가장 큰 값은 8 byte이다. setVal에서 q->t를 rax에 옮길 때 8byte만큼 떨어진 위치에서 입력받았으므로 B의 값은 8 이하라고 할 수 있다. Int는 4byte, short는 2byte이다. 저장 순서상 q->t 다음에 short s[A]가 저장되고, q->u가 저장된다. q->u는 32byte 이후에 저장되므로, 이 사이의 24byte를 int t와 short s[A]가 차지하고 있다. 8byte 단위로 저장 공간이 끊기므로, 최소 17byte 이상 저장이 되어야 한다. 이는 A의 값이 7 이상이어야 함을 의미하며, 24byte를 모두 사용한 경우는 A가 10인 경우이므로 A는 7~10의 정수이다.

Line 4를 보면 str1에서 long y가 184byte 후에 저장이 되고 역시 최대 크기의 데이터는 8byte이다. 이는 A와 B와 int형의 크기인 4의 곱이 177 이상 184 이하여야 함을 의미한다. 이 값이 2D array x의 크기이다. 4\*A\*B의 값은 180, 184가 가능하며 이는 A\*B로 45, 46이 가능함을 의미한다. 45는 3\*3\*5, 46은 2\*23이다. A와 B 모두가 10 이하여야 하므로 46은 불가능, 45의 경우 A가 9, B가 5인 경우 가능하다. 따라서 답은 **A = 9, B = 5**이다.

1. (3.69)

먼저, a\_struct의 크기는 0x118 byte 이하 0x111 byte 이상이다. 이는 line 2와 3에서 이용하는 rsi의 주소의 차이가 0x120이고 b\_struct의 first 변수가 int이기 때문이다. Line 4에서 rax에 5\*rdi, 즉 5\*i를 옮기고 line 5에서 rax = rsi + 8\*rax를 한다. 이는 rax에 rsi + 40\*i를 옮김을 의미한다. 이 때 rax에 저장된 값은 주소값인데, rsi가 bp를 의미하기 때문이다. 여기서 우리는 a\_struct의 array 하나의 크기가 40byte임을 알 수 있다. Line 6에서 rdx에 rax+8이 가리키는 위치의 데이터를 옮긴다. 이후 line 8에서 rax + 8\*rdx + 0x10 가 가리키는 위치에 앞서 line 2와 3에서 계산한 n의 값을 옮긴다.

먼저 CNT의 값을 구하자. 0x118은 10진법으로 280, 0x111은 273이다. 40의 배수를 만족하는 값은 280이므로 CNT는 280을 40으로 나눈 7이다. 즉, a는 bp+8의 위치부터 저장이 진행된다. a\_struct의 구성 요소에는 x와 idx가 있다. 또한 line 6을 보면 rax에서 8byte 떨어진 부분의 값을 rdx에 옮긴다. 이후 line 8에서 rax+0x10 + 8\*rdx에 rcx에 옮긴다. 여기서 rdx에 저장되어 있던 값이 idx일 것이고, 이 크기는 8byte 이하이다. 남은 32byte는 8로 나누면 4이고, 이는 long x[4]로 생각할 수 있다. 정리하면, **CNT = 7**, a\_struct는 다음과 같다.

**typedef struct{**

**int idx;**

**long x[4];**

**} a\_struct;**

1. (3.70)
2. Union의 경우 한 번에 하나의 field밖에 사용하지 못하므로 e1과 e2가 같은 저장 장소를 사용한다. 따라서 e1와 e2의 시작 주소는 같다. e1.p의 경우는 offset이 0, e1.y의 경우는 8byte 뒤인 +8이다. 마찬가지로 e2.x는 offset이 0, e2.next는 long이 8byte 이므로 +8의 offset을 가진다.
3. 총 16 byte의 크기를 요구하게 된다. e1과 e2 둘 모두 각 structure가 가지는 두 요소에 대해 8byte가 필요하기 때문이다.
4. Line 2에서 rdi+8이 가리키는 위치의 값을 rax에 옮겼다. 이는 e1.y일수도, e2.next일수도 있다. Line 3에서 이 rax가 가지는 주소에 있는 값을 rdx에 옮긴다. 즉, line 2에서는 e2.next의 값을 옮긴다고 생각할 수 있다. Line 4에서는 다시 rdx가 저장하는 주소값이 가리키는 부분의 값을 rdx에 옮긴다. Line 5에서는 rax+8이 가리키는 값을 rdx에서 뺀다. Line 6에서는 이 rdx 값을 rdi가 가리키는 부분에 대입하고 끝난다. 이를 정리하면, 다음과 같은 결과를 얻을 수 있다.

**up->e2.x = \*(e2.next) – e1.y;**

데이터형을 모두 long 형으로 맞추기 위해서는 이 방법이 유일하다. Line 2~4 때문에 역참조가 일어날 수 있는 유일한 방법인 e2.next가 중앙에 들어가야 하며 같은 이유로 long 형을 맞춰 주려면 처음에는 e2.x, 마지막에는 e1.y가 들어가야 한다.

1. (3.72)
2. Line 4에서 16 byte만큼의 값을 빼 8 byte는 n에, 나머지 8byte는 빈 공간으로 남겨 두었다. Line 5에서 rax에 30 + rdi\*8의 값을 대입하는데, 이후 -16과 and 연산을 가하면 처음 4bit는 사라지고 16의 배수로 내림하게 된다. 이 때, 우리는 rdi가 n임을 알고 있다. 이는 n이 짝수인 경우는 8\*n+16, 홀수인 경우는 8\*n+24를 표현하게 된다.
3. Line 8에서 rsp에 15를 더한 값을 r8에 옮긴 뒤, 다시 -16과 andq 연산을 진행한다. 이것이 의미하는 것은 16의 배수가 되도록 올림하는 것이다. 만일 rsp가 이미 16의 배수였다면 15를 더해도 16k+15 꼴이었을 것이고, -16과 and 연산을 하면 16의 배수로 내림되므로 k는 같다. 하지만 16의 배수가 아닌 경우는 k가 1 증가한 상태에서 16의 배수로 내림되기 때문에, 이 과정은 16의 배수로 올림하는 과정이라고 생각할 수 있다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **n** | **s1** | **s2** | **p** | **e1** | **e2** |
| 5 | 2065 | 2001 | 2016 | 9 | 15 |
| 6 | 2064 | 2000 | 2000 | 16 | 0 |
| 5 | 2064 | 2000 | 2000 | 24 | 0 |
| 6 | 2065 | 2001 | 2016 | 1 | 15 |

위의 표에서 확인할 수 있듯이, 가장 최대 e1과 최소 e2를 가지는 경우는 n=5(홀수), s1=2064(16의 배수)인 경우이며 최소 e1과 최대 e2를 가지는 경우는 n=6(짝수), s1=2064(16의 배수+1)이다.

1. 위의 표에서도 확인 가능하나, array의 start address를 항상 16의 배수가 되도록 맞출 수 있다는 장점이 있다. 이를 이용하면 16byte를 단위로 사용하는 환경에서 유용하게 사용할 수 있다.

소스코드 참조 요망.

함께 첨부된 소스코드를 보면, main에서 함수 vulnerable\_function을 호출하는 것을 알 수 있다. 이 vulnerable\_function에서는 100byte짜리 buffer에 string을 입력받아 복사한다. Stack을 생각해보면, 100byte에 해당하는 만큼의 buffer 다음에는 main 함수의 old ebp, return address가 순서대로 저장되어 있다. 이는 우리가 100byte를 넘겨 return address에 해당하는 부분까지 입력을 하면 여기에 입력된 값으로 함수가 return됨을 의미한다. 가령 우리가 몰래 실행하고자 하는 not\_called의 함수 주소가 0x8048444라면, 이 값을 return address에 위치하도록 buffer를 입력하면 이 함수가 호출된다.