**Defcon2017 - crackme**

\_\_int64 \_\_fastcall main(\_\_int64 a1, char \*\*a2, char \*\*a3)

{

char \*input; // rbx@1

signed \_\_int64 v4; // rax@1

input = (char \*)calloc(1uLL, 81uLL);

puts("enter code:");

fflush(stdout);

fgets(input, 80, stdin);

v4 = input80\_processing(input);

printf("sum is %ld\n", v4);

return 0LL;

}

Calloc 은 동적할당해서 0으로 초기화

81개의 메모리 할당후 인풋으로 80개 받고 마지막 널로 채운다.

LODWORD는 다음을 참고한다.

<https://reverseengineering.stackexchange.com/questions/8296/whats-the-function-of-lodword-and-hidword>

input은 캐릭터형 포인터 80글자와 1 개행문자를 가지고 있음.

\_\_int64는 8바이트 int형 자료형

QWORD는 8바이트 자료형

Movsx에 대한 설명

<http://reverseengine.tistory.com/entry/%EC%A0%9C-13%ED%8E%B8-Movzx-Movsx>

어쨋든

다음과 같이 디컴파일 된다.

v1 = sub\_556496BBC93B(\*(\_BYTE \*)input) >> 3;

v2 = (sub\_556496BBC955(\*(\_BYTE \*)(input + 1)) + v1) >> 3;

v3 = (sub\_556496BBC971(\*(\_BYTE \*)(input + 2)) + v2) >> 3;

v4 = (sub\_556496BBC98B(\*(\_BYTE \*)(input + 3)) + v3) >> 3;

v5 = (sub\_556496BBC9A5(\*(\_BYTE \*)(input + 4)) + v4) >> 3;

v6 = (sub\_556496BBC9C1(\*(\_BYTE \*)(input + 5)) + v5) >> 3;

v7 = (sub\_556496BBC9DB(\*(\_BYTE \*)(input + 6)) + v6) >> 3;

v8 = (sub\_556496BBC9F5(\*(\_BYTE \*)(input + 7)) + v7) >> 3;

v9 = (sub\_556496BBCA0F(\*(\_BYTE \*)(input + 8)) + v8) >> 3;

v10 = (sub\_556496BBCA29(\*(\_BYTE \*)(input + 9)) + v9) >> 3;

v11 = (sub\_556496BBCA43(\*(\_BYTE \*)(input + 10)) + v10) >> 3;

v12 = (sub\_556496BBCA5D(\*(\_BYTE \*)(input + 11)) + v11) >> 3;

v13 = (sub\_556496BBCA79(\*(\_BYTE \*)(input + 12)) + v12) >> 3;

v14 = (sub\_556496BBCA93(\*(\_BYTE \*)(input + 13)) + v13) >> 3;

v15 = (sub\_556496BBCAAD(\*(\_BYTE \*)(input + 14)) + v14) >> 3;

v16 = (sub\_556496BBCAC7(\*(\_BYTE \*)(input + 15)) + v15) >> 3;

v17 = (sub\_556496BBCAE1(\*(\_BYTE \*)(input + 16)) + v16) >> 3;

v18 = (sub\_556496BBCAFB(\*(\_BYTE \*)(input + 17)) + v17) >> 3;

v19 = (sub\_556496BBCB17(\*(\_BYTE \*)(input + 18)) + v18) >> 3;

v20 = (sub\_556496BBCB31(\*(\_BYTE \*)(input + 19)) + v19) >> 3;

v21 = (sub\_556496BBCB4B(\*(\_BYTE \*)(input + 20)) + v20) >> 3;

v22 = (sub\_556496BBCB65(\*(\_BYTE \*)(input + 21)) + v21) >> 3;

v23 = (sub\_556496BBCB7F(\*(\_BYTE \*)(input + 22)) + v22) >> 3;

v24 = (sub\_556496BBCB99(\*(\_BYTE \*)(input + 23)) + v23) >> 3;

v25 = (sub\_556496BBCBB5(\*(\_BYTE \*)(input + 24)) + v24) >> 3;

v26 = (sub\_556496BBCBCF(\*(\_BYTE \*)(input + 25)) + v25) >> 3;

v27 = (sub\_556496BBCBE9(\*(\_BYTE \*)(input + 26)) + v26) >> 3;

v28 = (sub\_556496BBCC05(\*(\_BYTE \*)(input + 27)) + v27) >> 3;

v29 = (sub\_556496BBCC1F(\*(\_BYTE \*)(input + 28)) + v28) >> 3;

v30 = (sub\_556496BBCC3B(\*(\_BYTE \*)(input + 29)) + v29) >> 3;

if ( (sub\_556496BBCC52(\*(\_BYTE \*)(input + 30)) + v30) >> 3 != 19 )

remote 디버깅해보면 알겠지만 인풋 31개에 대해서 각각 함수를 호출하여 알맞은 값인지 확인한다. 그리고 한단계 한단계 넘어가는데 sar 3 이있는데 이건 별로 중요한 루틴이 아닌 것 같다.

그리고 함수의 return 값은 eax에 저장된다. 어셈에 적혀있다.

인자 값이 넘어올 때, RAX, RBX, RDI가 이용되었다.

아이다상에서 mov는 오른쪽에서 왼쪽으로 대입이다.

byte ptr [rdi] : 해당 주소에 있는 값을 불러온다.