DESCRIPTION

I found a strange program, that encrypts a sum of numbers you provide. It seems pretty useless. Can you take a look at it?

Connect to 141.85.224.106:31346 and get the flag.

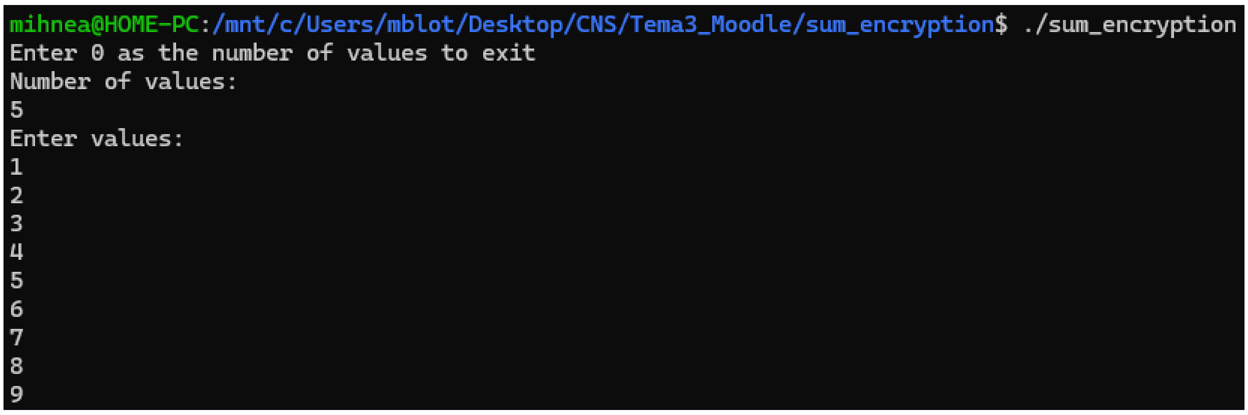
Your local time and the time on the remote machine might not be precisely the same. If your exploit doesn't work you may need a 1-5 second adjustment to your system time to match the remote time.

RESOURCES

As part of the challenge I received an executable file called **sum_encryption** as an attachment for analysis as well as a **libc** version used for compilation.

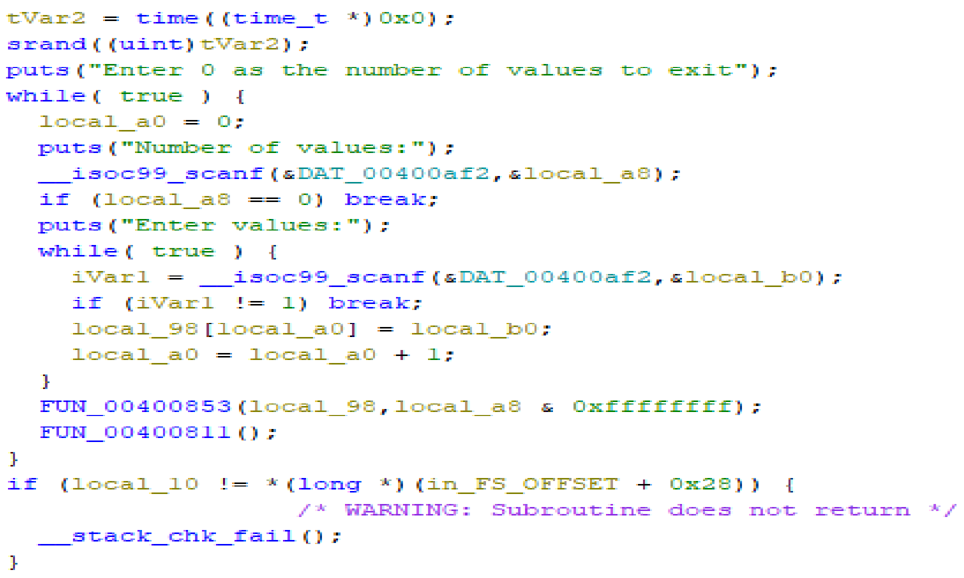
APPROACHES

1. The first approach here was to run the program and after running it I could see that it is requiring an input for the **number of values** and then a series of values. As an observation here is that we have entered a number **of values of 5** but we have entered more than 5 values and the reading from keyboard still did not stop.



```
mihnea@HOME-PC:/mnt/c/Users/mblot/Desktop/CNS/Tema3_Moodle/sum_encryption$ ./sum_encryption
Enter 0 as the number of values to exit
Number of values:
5
Enter values:
1
2
3
4
5
6
7
8
9
```

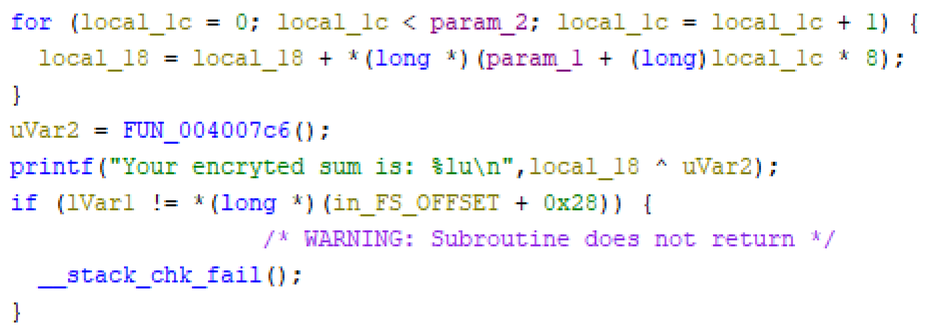
2. Thus, the decompilation of the **main function** looks like this:



```
tVar2 = time((time_t *)0x0);
srand((uint)tVar2);
puts("Enter 0 as the number of values to exit");
while( true ) {
  local_a0 = 0;
  puts("Number of values:");
  __isoc99_scanf(&DAT_00400af2,&local_a8);
  if (local_a8 == 0) break;
  puts("Enter values:");
  while( true ) {
    iVar1 = __isoc99_scanf(&DAT_00400af2,&local_b0);
    if (iVar1 != 1) break;
    local_98[local_a0] = local_b0;
    local_a0 = local_a0 + 1;
  }
  FUN_00400853(local_98,local_a8 & 0xffffffff);
  FUN_00400811();
}
if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                  /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

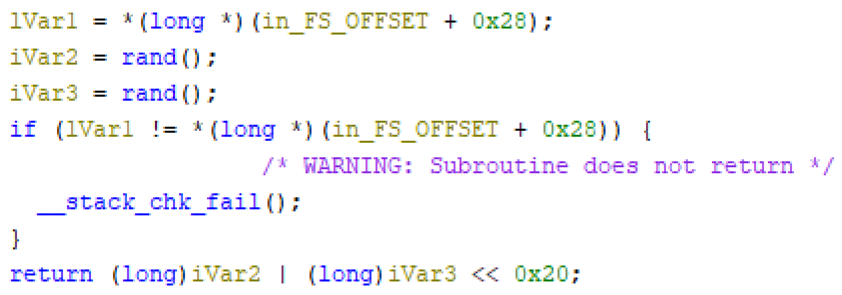
3. Here we can see that the program is waiting for an input from keyboard and checks if it is 0. If it is, the function ends. However, if the number is different than 0, it is ignored and we enter another **while true** loop and we read numbers until **EOF**.

4. When **EOF** is reached, another function is called that can be seen below:



```
for (local_1c = 0; local_1c < param_2; local_1c = local_1c + 1) {
  local_18 = local_18 + *(long *)(param_1 + (long)local_1c * 8);
}
uVar2 = FUN_004007c6();
printf("Your encryted sum is: %lu\n",local_18 ^ uVar2);
if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                  /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

5. This function, **sums all the numbers read in the array until the first number read** and then it prins an encrypted sum which is the previous mentioned sum **XORed** with a value return from another function that can be seen below:



```
lVar1 = *(long *)(in_FS_OFFSET + 0x28);
iVar2 = rand();
iVar3 = rand();
if (lVar1 != *(long *)(in_FS_OFFSET + 0x28)) {
                  /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
return (long)iVar2 | (long)iVar3 << 0x20;
```

6. So, the final **encrypted_sum** seen on the screen is actually a **XOR** between the sum of the first **num** numbers read where **num** is the first given number as input and a random generated number by formula **rand() | rand () << 0x20**;

7. So the idea of the exploit here is the same as in the previous exercises. We want to override the return addreess to a **pop_rdi_ret** gadget that will put the first parameter as **puts@got** and then will call **puts@plt** with it such that we leak the address of **puts** in our version of **libc** and then having the **offset** of **puts** we can determine the start of **libc** and any function inside it. We are interested of **system(„/bin/sh")**. We can do that as there is no limit in reading numbers from **stdin.**

8. The only problem here is that we also have a **canary value** on the stack which we have to see in order to use it at further overflows. Here we can use **encrypted sum** that gets printed on the screen as mentioned before. This encrypted **sum is (sum of first num numbers) ^ random_number** but we also control the first num numbers. **The size of the array is 17** so if we give **num = 18** and we control the first 17 numbers from our input, then the 18th number is going to be the **cannary value**. We can also generate the same **random_number** if we just create a program that returns **rand() | rand () << 0x20** if the random values have the same seed. This program is called **random_numbers_generator.c**

9. So having the random number and knowing the first 17 numbers of the array because we are the ones giving them, we can obtian the canary value as following:

Canary = (encrypted_sum ^ random_number) – sum of first 17 numbers

10. Now, having the **canary value** we firstly do a overflow to obtain the address of **puts** in this version of **libc** and then having the offset we obtain the address of **system** and then we do another buffer overflow where we call system(**„/bin/sh”**).

11. There is only one more issue here. There can be a situation where the time on the local host is different than the one on the server so that is why I run the **python script** in a loop that sometime It might happen to be the same time and use the seed that we want.

12. If you run the script (**python3 script.py**), and wait for some luck regarding the time, you will get a shell on the server and if we run **cat /home/ctf/flag** we will get the flag which is:

CNS_CTF{805528b3469d6ff4f9283ee466d3b87b}