NeverLAN CTF

Reverse Engineering : Reverse Engineer

Value : 300 pts

Difficulty : Medium

Description : Your flag will be in the normal flag{flagGoesHere} syntax.

Attachment : This program seems to get stuck while running... Can you get it to continue past the broken function?

Revseng & revseng-osx

Solution :

A binary is given for this challenge. When we try to execute this program, there's a segmentation fault. Let's open it in Ghidra to see why.

```
1
2  /* WARNING: Removing unreachable block (ram,0x00101161) */
3
4  int main(int argc,char **argv)
5
6  {
7    int x;
8
9    foo();
10   return 0;
11 }
12
```

In the decompiled window, we can see that the function main only calls the function foo.

```c
void foo(void)

{
  char *y;
  int x;

  x = 0;
  while (x < 0x67) {
    x = x + 1;
    y[x] = 'c';
  }
  return;
}
```

Foo is just putting many ‘c’ chars in an array. The array has no size, that’s why the program crash.
But the content of this array is clearly not the flag. It seems to be hidden somewhere else.

If we look at the assembly, we can see a call at a function print in main. That call is not shown in the disassembled window.

```
                        main                                            XREF

00101145 55             PUSH    RBP
00101146 48 89 e5       MOV     RBP,RSP
00101149 48 83 ec 20    SUB     RSP,0x20
0010114d 89 7d ec       MOV     dword ptr [RBP + local_1c],argc
00101150 48 89 75 e0    MOV     qword ptr [RBP + local_28],argv
00101154 c7 45 fc       MOV     dword ptr [RBP + x],0x79
         79 00 00 00
0010115b 83 7d fc 00    CMP     dword ptr [RBP + x],0x0
0010115f 7f 0c          JG      LAB_0010116d
00101161 b8 00 00       MOV     EAX,0x0
         00 00
00101166 e8 3e 00       CALL    print
         00 00
0010116b eb 0a          JMP     LAB_00101177

                        LAB_0010116d                                    XREF
0010116d b8 00 00       MOV     EAX,0x0
         00 00
00101172 e8 07 00       CALL    foo
         00 00
```

The program moves the value 0x79 to x, then compares if it’s greater than 0. If it isn’t, the function print is called. 0x79 is always greater than 0, so the program always jump to the call of foo and never executes print.

Let’s take a look to that print function.

```
void print(void)

{
  undefined *puVar1;
  char *flg;
  char rb;
  char lb;
  char g;
  char a;
  char l;
  char f;

  puVar1 = (undefined *)malloc(0x15);
  *puVar1 = 0x77;
  puVar1[1] = 0x33;
  puVar1[2] = 99;
  puVar1[3] = 0x6f;
  puVar1[4] = 0x6e;
  puVar1[5] = 0x37;
  puVar1[6] = 0x72;
  puVar1[7] = 0x30;
  puVar1[8] = 0x6c;
  puVar1[9] = 0x74;
  puVar1[10] = 0x68;
  puVar1[0xb] = 0x33;
  puVar1[0xc] = 0x62;
  puVar1[0xd] = 0x31;
  puVar1[0xe] = 0x6e;
  puVar1[0xf] = 0x61;
  puVar1[0x10] = 0x72;
  puVar1[0x11] = 0x69;
  puVar1[0x12] = 0x33;
  puVar1[0x13] = 0x73;
  puVar1[0x14] = 0;
  printf("%c%c%c%c%c%s%c\n",0x66,0x6c,0x61,0x67,0x7b,puV
  return;
```

It seems to be the function that prints the flag. All of the chars are added one by one in the array.

We can translate these values with an ascii character chart.

These values gives us the flag:

flag{w3con7r0lth3b1nari3s}