Description

You must find the key to to decrypt the flag hidden in the binary.

Solution

This challenge is solvable either using dynamic analysis or static analysis. The static approach requires understanding of the function `srand()` and identifying the decrypted flag array. The dynamic approach forces the solver to bypass the anti-debugging check and identify where the plaintext flag is stored on the stack.

Static Analysis

I used Ghidra's decompiler and disassembler to provide an overview of the binary as well as imported functions. The provided sample still has the function names which makes finding the main loop easy.Immediately, `setup()` is called within `main()`. `Setup` proceeds to pass a constant value to `srand()`. This sets the pseduorandom function seed value which ensures a repeatable sequence of values. This sequence of values will generate the key used to encrypt the flag.

```
void setup(void)

{
  srand(0x7e41);
  puts("setup complete");
  return;
}
```

Once `setup()` returns, ghidra detected a list of integer values which are the encrypted flag. The anti-debugging trick may be ignored during static analysis as the code is not executed.

Decompile: main - (test)

```
  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  setup();
  local_98 = 0x55cf58a1;
  local_94 = 0xe5863;
  local_90 = 0x6e2b697c;
  local_8c = 0x58716238;
  local_88 = 0x1c681233;
  local_84 = 0x1f91d896;
  local_80 = 0x4384135f;
  local_7c = 0x7a48ebff;
  local_78 = 0x751ec628;
  local_74 = 0x5cfb0d00;
  local_70 = 0x440dc90f;
  local_6c = 0x8db4a1;
  local_68 = 0x768c5673;
  local_64 = 0x4b4b905c;
  local_60 = 0x5aab56e3;
  local_5c = 0x69f41056;
  local_58 = 0x6c38c33d;
  local_54 = 0x5643640c;
  local_50 = 0x477b61ac;
  local_4c = 0x519164f5;
  local_48 = 0x69007fd6;
  local_44 = 0x6025ed2;
  local_40 = 0x6284c735;
  local_3c = 0x5eb5b8e8;
  local_38 = 0x5680d866;
  local_34 = 0x5007fd80;
  local_30 = 0x653ff172;
  local_2c = 0x6d410d66;
  local_28 = 0x2ebb5c7b;
  local_24 = 0x15dad3ab;
  local_20 = 0x3e230119;
  local_1c = 0x48ab487;
  puts("Can you guess my secret key?");
  in = (uchar *)0x0;
  out = (uchar *)0x0;
  lVar1 = ptrace(PTRACE_TRACEME,0,1);
  if (lVar1 < 0) {
    printf("Slick move Rick, try again.");
```

The final function called is `decrypt()` which accepts the encrypted flag structure. The key is generated from calling `rand()` and xored with the current index of the encrypted flag to produce the plaintext flag. The result ofthe xor operations is stored in a separate list.

```
for (counter = 0; counter < 0x20; counter = counter + 1) {
  uVar1 = rand();
  auStack152[counter] = uVar1;
  auStack280[counter] = *(uint *)(ctx + (long)counter * 4) ^ auStack152[counter];
}
```

To retrieve the flag, write a simple C/C++ program to decrypt the flag using the identified constant seed.

Dynamic Analysis

The binary exits when attempting to run within a debugger such as GDB and within bash.Below is output from GDB and executing without a debugger.

```
GEF for linux ready, type `gef' to start, `gef config' to configure
90 commands loaded and 5 functions added for GDB 9.2 in 0.00ms using Python engine 3.8
Reading symbols from print-sol...
(No debugging symbols found in print-sol)
gef➢  r
[*] Failed to find objfile or not a valid file format: [Errno 2] No such file or directory:
'system-supplied DSO at 0x7ffff7ffe000'
setup complete
Can you guess my secret key?
Slick move Rick, try again.[Inferior 1 (process 23914) exited with code 071]


guest@doge:~/home/guest$ ./sample.sus
setup complete
Can you guess my secret key?
almost there
guest@doge:~/home/guest$
```

The binary is not stripped of symbols and user defined function names which is confirmed within gbd `info functions`. Some of the result are listed below with the respective memory address.

```
0x0000000000401c90  __do_global_dtors_aux
0x0000000000401cd0  frame_dummy
0x0000000000401d05  setup
0x0000000000401d26  decrypt
0x0000000000401ddc  main
0x0000000000401f80  get_common_indices.constprop
0x0000000000402300  __libc_start_main
```

Setting a breakpoint at `main()` will allow us to review the the assembly `disassemble main` which includes a call/reference to `ptrace()`.

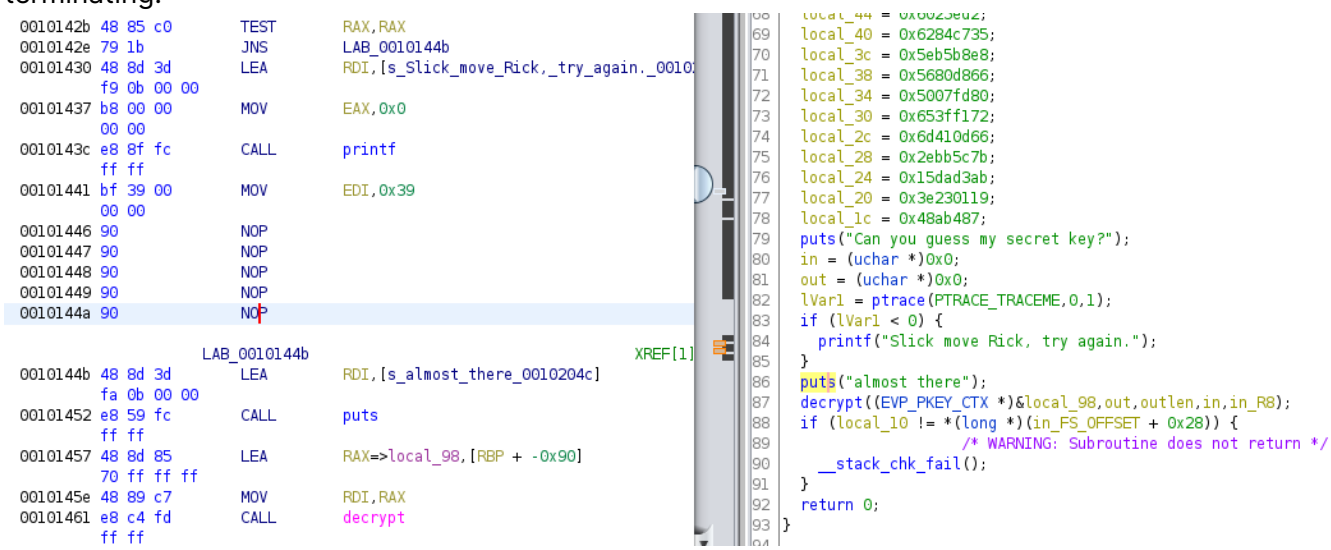
```
0x0000000000401f09 <+301>:      mov    ecx,0x0
   0x0000000000401f0e <+306>:   mov    edx,0x1
   0x0000000000401f13 <+311>:   mov    esi,0x0
   0x0000000000401f18 <+316>:   mov    edi,0x0
   0x0000000000401f1d <+321>:   mov    eax,0x0
   0x0000000000401f22 <+326>:   call   0x452000 <ptrace>
```

The `setup()` function passes an integer value to `srandom()` which seeds a pseduorandom sequence of values.

```
gef➤  disassemble setup
Dump of assembler code for function setup:
   0x0000000000401d05 <+0>:     endbr64
   0x0000000000401d09 <+4>:     push   rbp
   0x0000000000401d0a <+5>:     mov    rbp,rsp
   0x0000000000401d0d <+8>:     mov    edi,0x7e41
   0x0000000000401d12 <+13>:    call   0x4104e0 <srandom>
```

Googling about ptrace and debugging lead to several articles talking about anti-debugging techniques
[Ptrace Anti-debugging]
[Bypass the Ptrace call]

Utilizing the most recent version of Ghidra due to 9.x.x having issues with producing valid binaries. Patching the `exit()` call to NOPs forces the program to continue to execute code instead of terminating.



With the `exit()` call negated, the decrypt function will provide us with the plaintext flag somewhere in memory. Recalling the `setup()` which passed a constant value to `srandom()` caused the `random()` within a loop to stand out.

`0x0000000000401d35 <+15>: mov QWORD PTR [rbp-0x128],rdi` is the for loop counter/index.
`0x0000000000401dbc <+150>: cmp DWORD PTR [rbp-0x118],0x1f` checks if counter<32

Generate the key used to encrypt/decrypt the flag. The result of `rand()` is stored at address `rbp-0x114` which is a temporary variable and then stored in a list.

```
   0x0000000000401d57 <+49>:    call   0x410c30 <rand>
   0x0000000000401d5c <+54>:    mov    DWORD PTR [rbp-0x114],eax
   0x0000000000401d62 <+60>:    mov    eax,DWORD PTR [rbp-0x118]
   0x0000000000401d68 <+66>:    cdqe
   0x0000000000401d6a <+68>:    mov    edx,DWORD PTR [rbp-0x114]
   0x0000000000401d70 <+74>:    mov    DWORD PTR [rbp+rax*4-0x90],edx
```

We retrieve the value from the encrypted list of values and the value generate from `rand()` which are then xorred together. The result is stored in a final list.

```
0x55555555529a <decrypt+112>    mov    rax, QWORD PTR [rbp-0x128]
●→ 0x5555555552a1 <decrypt+119>    add    rax, rcx
   0x5555555552a4 <decrypt+122>    mov    eax, DWORD PTR [rax]
   0x5555555552a6 <decrypt+124>    xor    eax, edx
   0x5555555552a8 <decrypt+126>    mov    edx, eax
   0x5555555552aa <decrypt+128>    mov    eax, DWORD PTR [rbp-0x118]
   0x5555555552b0 <decrypt+134>    cdqe
 → 0x5555555552b2 <decrypt+136>    mov    DWORD PTR [rbp+rax*4-0x110], edx


x/50bs 0x7fffffffd8f0

0x7fffffffd8f0: 0x0000005400000043      0x0000007b00000046
0x7fffffffd900: 0x0000006f00000063      0x000000730000006e
0x7fffffffd910: 0x0000006100000074      0x000000740000006e
0x7fffffffd920: 0x000000730000005f      0x0000006500000065
0x7fffffffd930: 0x0000007300000064      0x000000610000005f
0x7fffffffd940: 0x0000006500000072      0x0000006e0000005f
0x7fffffffd950: 0x0000005f00000074      0x0000006500000073
0x7fffffffd960: 0x0000007500000063      0x0000007d00000072
```

The xor of the flag_encrypted and key produces the plaintext flag stored at 0×7fffffffd8f0.

Source Code

Build Command

`gcc secret.c -o test`

`C,T,F,{,c,o,n,s,t,a,n,t,_,s,e,e,d,s,_,a,r,e,_,n,t,_,s,e,c,u,r,}`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/ptrace.h>

#define flag_len 64

//char *flag= []

void setup(){
    srand(32321);
    printf("setup complete\n");
```

```
}

void decrypt(uint32_t *flag){
   //
    int xor_flag[32];
    uint32_t key[32];
    for(int i=0;i<32;i++){
      uint32_t val = rand();
      key[i] = val;
      xor_flag[i]= key[i]^(*(flag+i));
      //printf("%c,",xor_flag[i]);
     }
}


int main(int argc, char *argv[]){
    setup();
   uint32_t flag[32] =
{1439652001,940131,1848338812,1483825720,476582451,529651862,1132729183,2051599359,196495108
0,1559956736,1141754127,9286817,1988908659,1263243356,1521178339,1777602646,1815659325,14472
57100,1199268268,1368483061,1761640406,100818642,1652868917,1588967656,1451284582,1342700928
,1698689394,1832979814,784030843,366662571,1042481433,76199047};

//    int flag[32] =
{67,84,70,123,99,111,110,115,116,97,110,116,95,115,101,101,100,115,95,97,114,101,95,110,116,
95,115,101,99,117,114,125}; //cleartext
printf("Can you guess my secret key?\n");
if (ptrace(PTRACE_TRACEME,0,1,0)<0) {
    printf("Slick move Rick, try again.");
    exit(57);
}

printf("almost there\n");
decrypt(flag);
return 0;}
```