DESCRIPTION

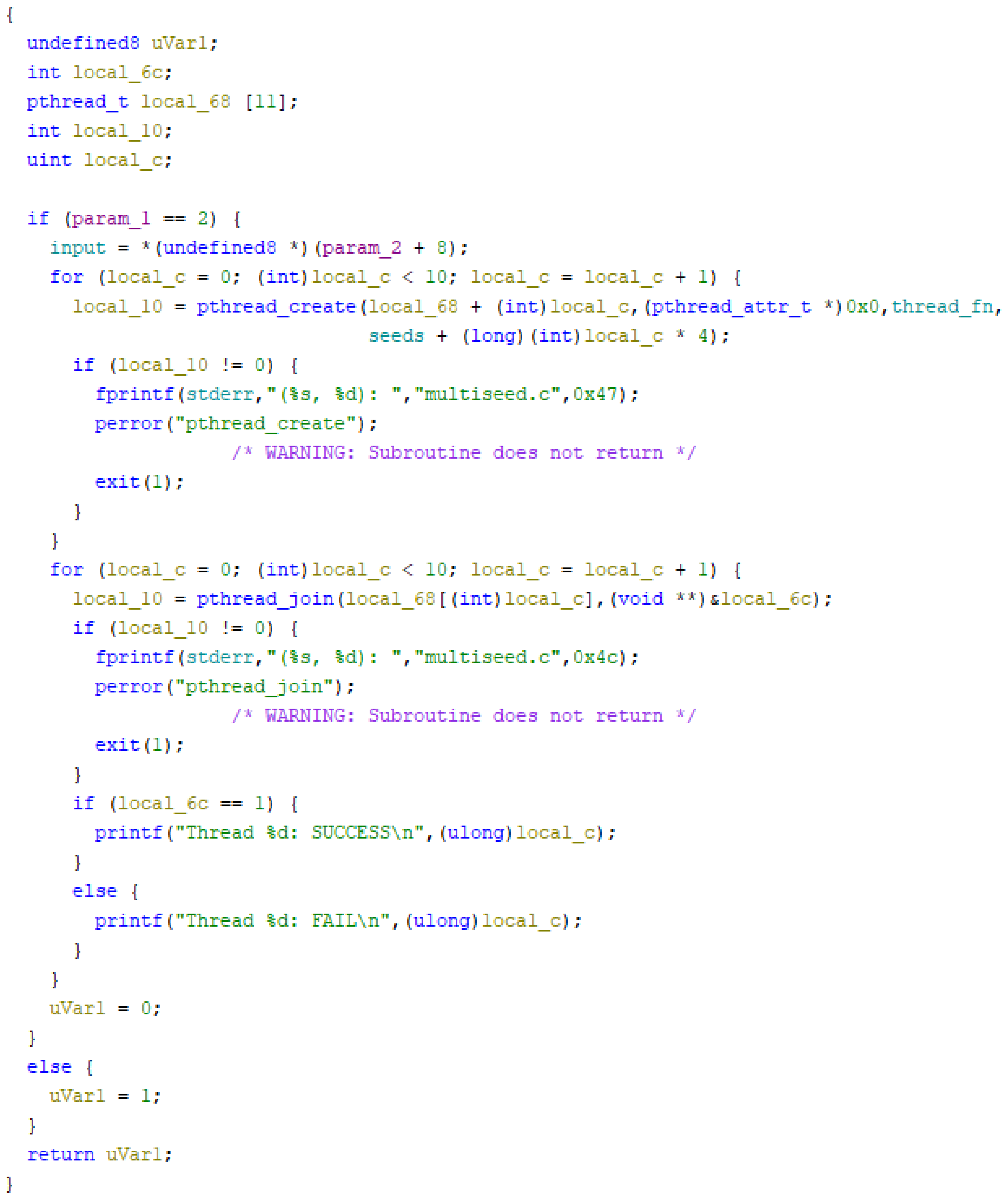
Some seed values are better than others.

RESOURCES

As part of the challenge I received an executable file called **multiseed** as an attachment for analysis.

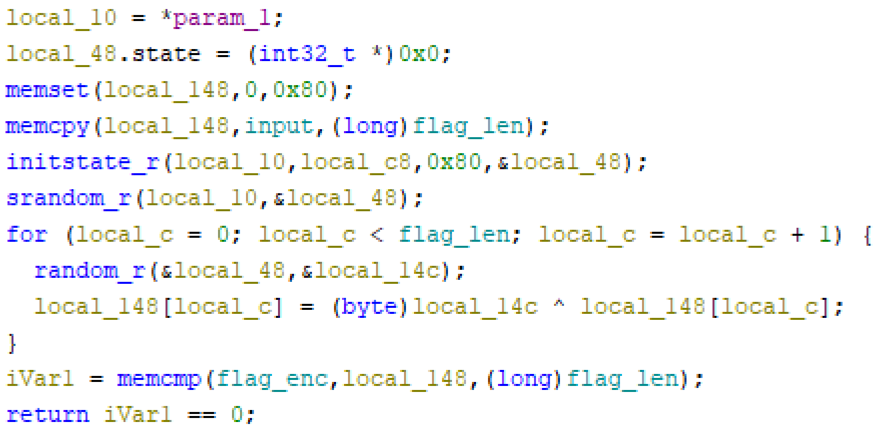
APPROACHES

1. The first try here was to run the program and give also an input and I have seen that all the threads return failed as a response;
2. Thus, I had no other idea than to decompile the executable. The **main** function decompiled can be seen below:



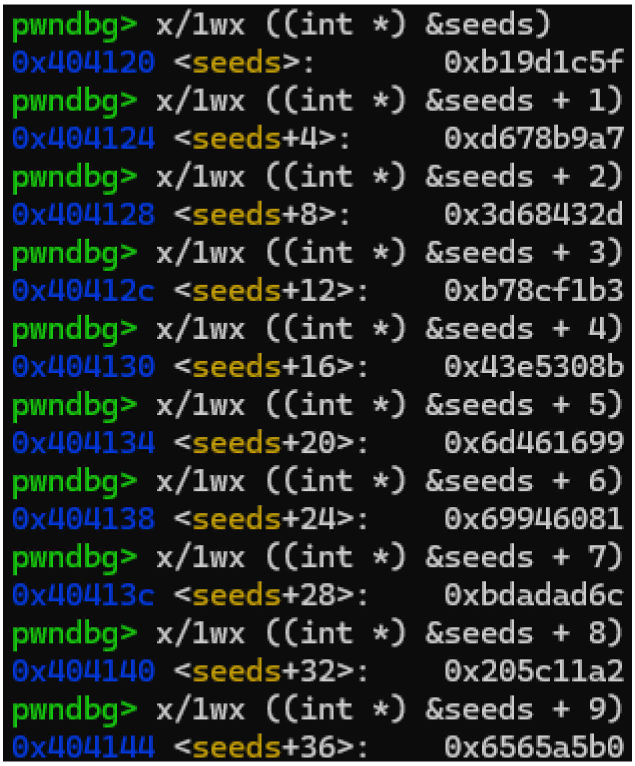
```
{
  undefined8 uVar1;
  int local_6c;
  pthread_t local_68 [11];
  int local_10;
  uint local_c;

  if (param_1 == 2) {
    input = *(undefined8 *)(param_2 + 8);
    for (local_c = 0; (int)local_c < 10; local_c = local_c + 1) {
      local_10 = pthread_create(local_68 + (int)local_c,(pthread_attr_t *)0x0,thread_fn,
                                seeds + (long)(int)local_c * 4);
      if (local_10 != 0) {
        fprintf(stderr,"(%s, %d): ","multiseed.c",0x47);
        perror("pthread_create");
                    /* WARNING: Subroutine does not return */
        exit(1);
      }
    }
    for (local_c = 0; (int)local_c < 10; local_c = local_c + 1) {
      local_10 = pthread_join(local_68[(int)local_c],(void **)&local_6c);
      if (local_10 != 0) {
        fprintf(stderr,"(%s, %d): ","multiseed.c",0x4c);
        perror("pthread_join");
                    /* WARNING: Subroutine does not return */
        exit(1);
      }
      if (local_6c == 1) {
        printf("Thread %d: SUCCESS\n",(ulong)local_c);
      }
      else {
        printf("Thread %d: FAIL\n",(ulong)local_c);
      }
    }
    uVar1 = 0;
  }
  else {
    uVar1 = 1;
  }
  return uVar1;
}
```

3. We can see here that there are 10 threads that get spawned, each of them running the **thread_fn** function with a specific argument from the **seeds** vector. Seeds seem to be a vector of integers because each thread gets a value incremented by 4 compared to the previous thread.
4. Then, we can look to see what each thread does in the **thread_fn**



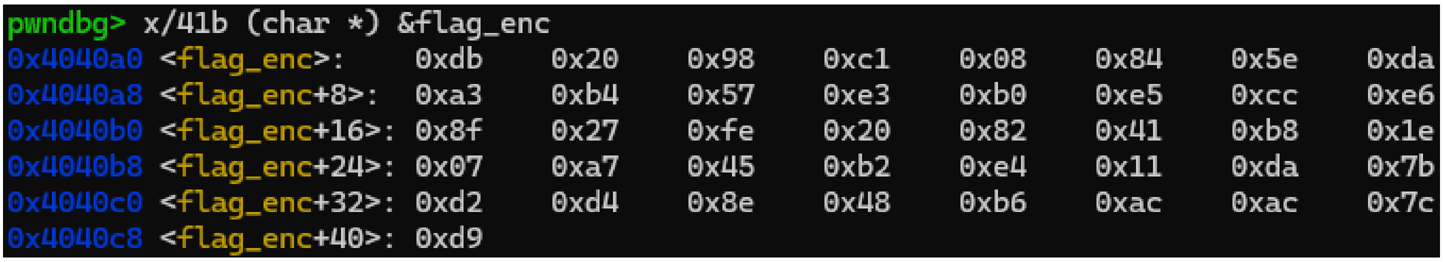
```
local_10 = *param_1;
local_48.state = (int32_t *)0x0;
memset(local_148,0,0x80);
memcpy(local_148,input,(long)flag_len);
initstate_r(local_10,local_c8,0x80,&local_48);
srandom_r(local_10,&local_48);
for (local_c = 0; local_c < flag_len; local_c = local_c + 1) {
  random_r(&local_48,&local_14c);
  local_148[local_c] = (byte)local_14c ^ local_148[local_c];
}
iVar1 = memcmp(flag_enc,local_148,(long)flag_len);
return iVar1 == 0;
```

5. Here we can see that each thread copies the user input value in a local variable, generates **flag_len** numbers according to the given seed and the XORs each of the bytes from the input string with the generated number and stores the result.
6. Then, at the end, the final result is compared with the **flag_enc** and if they match it means that the thread succeeds;
7. That being said, we have in the binary the **flag_enc** and the **seeds** vectors but we don't have the input. Since we know that **flag_enc[i] = random_generated_number(seed[i])[i] ^ input[i]** we can do a reverse engineering by extracting the **flag_enc** and the **seeds** from the executable, redo the generation of the numbers and do a XOR between the **flag_enc[i]** and the generated number. This will give the correct input for one of the threads.
8. The **seeds** vector can be seen in the following picture:



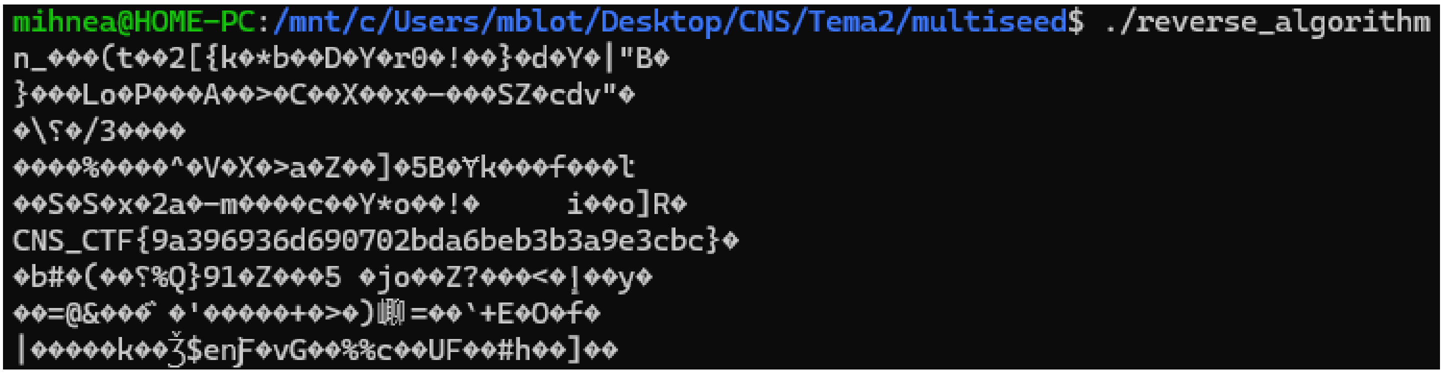
```
pwndbg> x/1wx ((int *) &seeds)
0x404120 <seeds>:       0xb19d1c5f
pwndbg> x/1wx ((int *) &seeds + 1)
0x404124 <seeds+4>:     0xd678b9a7
pwndbg> x/1wx ((int *) &seeds + 2)
0x404128 <seeds+8>:     0x3d68432d
pwndbg> x/1wx ((int *) &seeds + 3)
0x40412c <seeds+12>:    0xb78cf1b3
pwndbg> x/1wx ((int *) &seeds + 4)
0x404130 <seeds+16>:    0x43e5308b
pwndbg> x/1wx ((int *) &seeds + 5)
0x404134 <seeds+20>:    0x6d461699
pwndbg> x/1wx ((int *) &seeds + 6)
0x404138 <seeds+24>:    0x69946081
pwndbg> x/1wx ((int *) &seeds + 7)
0x40413c <seeds+28>:    0xbdadad6c
pwndbg> x/1wx ((int *) &seeds + 8)
0x404140 <seeds+32>:    0x205c11a2
pwndbg> x/1wx ((int *) &seeds + 9)
0x404144 <seeds+36>:    0x6565a5b0
```

9. The **flag_enc** buffer of length **flag_len = 41** can be seen in the following picture:



```
pwndbg> x/41b (char *) &flag_enc
0x4040a0 <flag_enc>:    0xdb    0x20    0x98    0xc1    0x08    0x84    0x5e    0xda
0x4040a8 <flag_enc+8>:  0xa3    0xb4    0x57    0xe3    0xb0    0xe5    0xcc    0xe6
0x4040b0 <flag_enc+16>: 0x8f    0x27    0xfe    0x20    0x82    0x41    0xb8    0x1e
0x4040b8 <flag_enc+24>: 0x07    0xa7    0x45    0xb2    0xe4    0x11    0xda    0x7b
0x4040c0 <flag_enc+32>: 0xd2    0xd4    0x8e    0x48    0xb6    0xac    0xac    0x7c
0x4040c8 <flag_enc+40>: 0xd9
```

10. Having this input information, and the decompilation from Ghidra I created the reverse algorithm as described in point 7. You can compile the code in file **reverse_algorithm.c** using **make** and then if you run the executable **./reverse_algorithm,** you can see that one of the **seeds** gives back the correct **flag:**



```
mihnea@HOME-PC:/mnt/c/Users/mblot/Desktop/CNS/Tema2/multiseed$ ./reverse_algorithm
n_���(t��2[{k�*b��D�Y�r0�!��}�d�Y�|"B�
}���Lo�P���A��>�C��X��x�–���SZ�cdv"�
�\ʕ�/3����
����%�����^�V�X�>a�Z��]�5B�Ɏk���f���ƖĿ
��S�S�x�2a�–m�����c��Y*o��!�      i��o]R�
CNS_CTF{9a396936d690702bda6beb3b3a9e3cbc}�
�b#�(��ʕ%Q}91�Z���5 �jo��Z?���<�|��y�
��=@&���̃ �'�����+�>�)唰=��`+E�O�f�
|�����k��ǯ$enƑ�vG��%%c��UF��#h��]��
```

11. Having this said, the correct flag for this challenge is:

CNS_CTF{9a396936d690702bda6beb3b3a9e3cbc}