Share-it : How bad is Picsou's Secret Sharing?

Augustin Bariant

May 8, 2022

1 My personal life and CTF experience

I am Augustin Bariant, 24 years old, and this is my first solo CTF. I once participated in a 10 hours CTF in groups of 3 (Aviation ISAC Collegiate CTF Competition), though back then I was a beginner in most CTF fields, excluding cryptography. We still managed to rank 3rd.

On the academic point of view, I was really interested in mathematics very early on, and I started to take part of the French Mathematical Olympiads in the beginning of high school. Following my interests, I went to the Louis Le Grand preparatory class in Mathematics and Physics and finally got accepted to Ecole Polytechnique. There, I heavily gained interest in computer science and decided to specialize in cryptography, a meeting point of my two loving fields. I performed a double degree with KTH Royal Institute of Technology in Stockholm during my last year of Ecole Polytechnique. I am currently a Ph.D student at INRIA de Paris in symmetric cryptography. The core subject of my Ph.D is the cryptanalysis of symmetric schemes.

For more information, check out my personal page here.

This challenge marks my first use of a reverse engineering tool. I chose Ghidra, as I heard that it was the most accessible one.

2 Share-it: The challenge

Picsou modified the library libecc to create its own Piscou's Secret Sharing (PSS) as a derivation of the original Shamir's Secret Sharing protocol, and used the new protocol to share the GPS coordinates of a treasure. The new protocol can very surely be broken, but Picsou did not explain his modifications.

For this challenge, we have:

- A shared object of the sss library.
- One share of the secret.

Objective Understand how PSS works. Find an attack to recover the secret key from the share of the secret.

3 Overview of Shamir's Secret Sharing (SSS) Protocol

Shamir's Secret Sharing Protocol is a multi-user protocol in which a secret data known by a user Picsou is split into n derived shares, such that the secret data can only be recoverd if $k \leq n$ shares are known. k is called the threshold. In our case, we know neither k nor n. For Picsou's case, it can be useful to generate n shares the secret data with k close to n, such that:

- If something happens to Picsou, or on Picsou's call, Picsou's friends can agree to combine their shares and to recover the location of the treasure.
- If he has at least $n-k+1$ trusted friends, it is impossible for the untrusted friends to unite more than $k-1$ shares, in which case they can not recover the treasure location.

Shamir's Secret Protocol relies on the Lagrange interpolation of polynoms. Picsou generates a polynom P of degree k: the secret is the constant coefficient a_0 and all the other coefficients are generated randomly. Each share is a pair $(i, P(i))$. If k shares are known, then the polynom can be interpolated from k different values, and the constant coefficient (the secret) recovered. SSS would face some non-uniformity properties if applied on integers, so instead SSS is performed on a large finite $\mathbb{Z}/p\mathbb{Z}$.

4 Reverse-Engineering Share-it

Since share-it is a library shared object of a modified version of sss.c implemented in the libecc, I first downloaded the libecc library and used the Makefile to compile the shared object from the sss folder. I opened both shared objects with Ghidra, used the decompiler from Ghidra, and started looking for functions which had the same signatures in both binaries. Note that the function names in the share-it binary were removed while they were not for the real sss binary. Eventually, I found the **sss_generate** function from the share-it binary. Then, I went through the **sss_generate** function the following way:

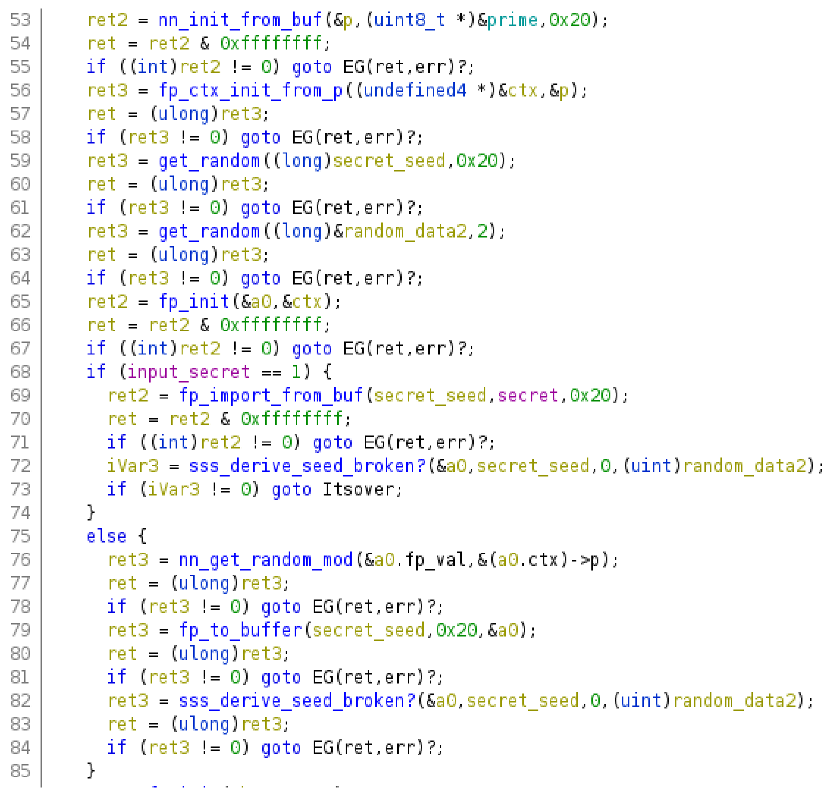
- I performed a step by step comparison between both decompiled codes (and also followed the source code in sss.c), and noted the changes.
- When encountering variables of which I knew the type and name in the original library, I renamed and retyped them (using types imported from the library file).
- When needed, I stepped into called functions to check for other changes.

- When encountering odd behaviour, I sometimes updated the structures (e.g. the fp struct) or the function signatures (e.g. **_sss_derive_seed**), though I am not sure if it was necessary or if it just came from a prior bad manipulation.

Eventually, I found out that **sss_generate** was not modified by itself, but **_sss_raw_generate** and **_sss_derive_seed** were both modified.

Illustrated on figure 1, **_sss_raw_generate** was modified as follows:

- It generates an additionnal 2-byte value, that I call random_data2 (line 62).
- It verifies if **input_secret=1** but in this case, it would still set a_0 to the return value of a call to **_sss_derive_seed**. It means that the value of input_secret does not matter (line 72).
- It calls a modified **_sss_derive_seed** with an extra input: random_data2 (line 72,82) and later on for the coefficient generation.



```
53    ret2 = nn_init_from_buf(&p,(uint8_t *)&prime,0x20);
54    ret = ret2 & 0xffffffff;
55    if ((int)ret2 != 0) goto EG(ret,err)?;
56    ret3 = fp_ctx_init_from_p((undefined4 *)&ctx,&p);
57    ret = (ulong)ret3;
58    if (ret3 != 0) goto EG(ret,err)?;
59    ret3 = get_random((long)secret_seed,0x20);
60    ret = (ulong)ret3;
61    if (ret3 != 0) goto EG(ret,err)?;
62    ret3 = get_random((long)&random_data2,2);
63    ret = (ulong)ret3;
64    if (ret3 != 0) goto EG(ret,err)?;
65    ret2 = fp_init(&a0,&ctx);
66    ret = ret2 & 0xffffffff;
67    if ((int)ret2 != 0) goto EG(ret,err)?;
68    if (input_secret == 1) {
69      ret2 = fp_import_from_buf(secret_seed,secret,0x20);
70      ret = ret2 & 0xffffffff;
71      if ((int)ret2 != 0) goto EG(ret,err)?;
72      iVar3 = sss_derive_seed_broken?(&a0,secret_seed,0,(uint)random_data2);
73      if (iVar3 != 0) goto Itsover;
74    }
75    else {
76      ret3 = nn_get_random_mod(&a0.fp_val,&(a0.ctx)->p);
77      ret = (ulong)ret3;
78      if (ret3 != 0) goto EG(ret,err)?;
79      ret3 = fp_to_buffer(secret_seed,0x20,&a0);
80      ret = (ulong)ret3;
81      if (ret3 != 0) goto EG(ret,err)?;
82      ret3 = sss_derive_seed_broken?(&a0,secret_seed,0,(uint)random_data2);
83      ret = (ulong)ret3;
84      if (ret3 != 0) goto EG(ret,err)?;
85    }
```

Figure 1: Overview of the decompiled code on **_sss_raw_generate** modified section.

Illustrated on figure 2, **_sss_derive_seed** was very heavily modified:

- out is set to the secret_seed (line 21).
- If $j = 0$, we dont go through the if statement (line 21), and out is returned with the secret_seed value.
- The statement of line 23 sets (&new_seed) to {random_val2,j} (because we are in little endian).
- hmac_val = hmac(&new_seed,4,SHA_256,&new_seed + 2, 2) (line 25) DOES NOT take the secret seed in input, but only the index j and the random_data2.
- hmac_val is reducted modulo p (line 35).
- Finally, out is multiplied with the hmac_val (line 38).

Conclusion of the reverse: the coefficients are not randomly generated as in Shamir's Secret Protocol. Let us exploit that property to mount an attack.

5 Attack on PSS

The coefficient j of Picsou's polynom are generated using **_sss_raw_generate(out,secret_seed, j, random_data2)**, with the same 2-byte random_data2 and secret_seed for all coefficients. Let us denote s the secret_seed and r the random_data2. The coefficient a_j can be decomposed as:

$$a_j = f(j, r) \times s$$

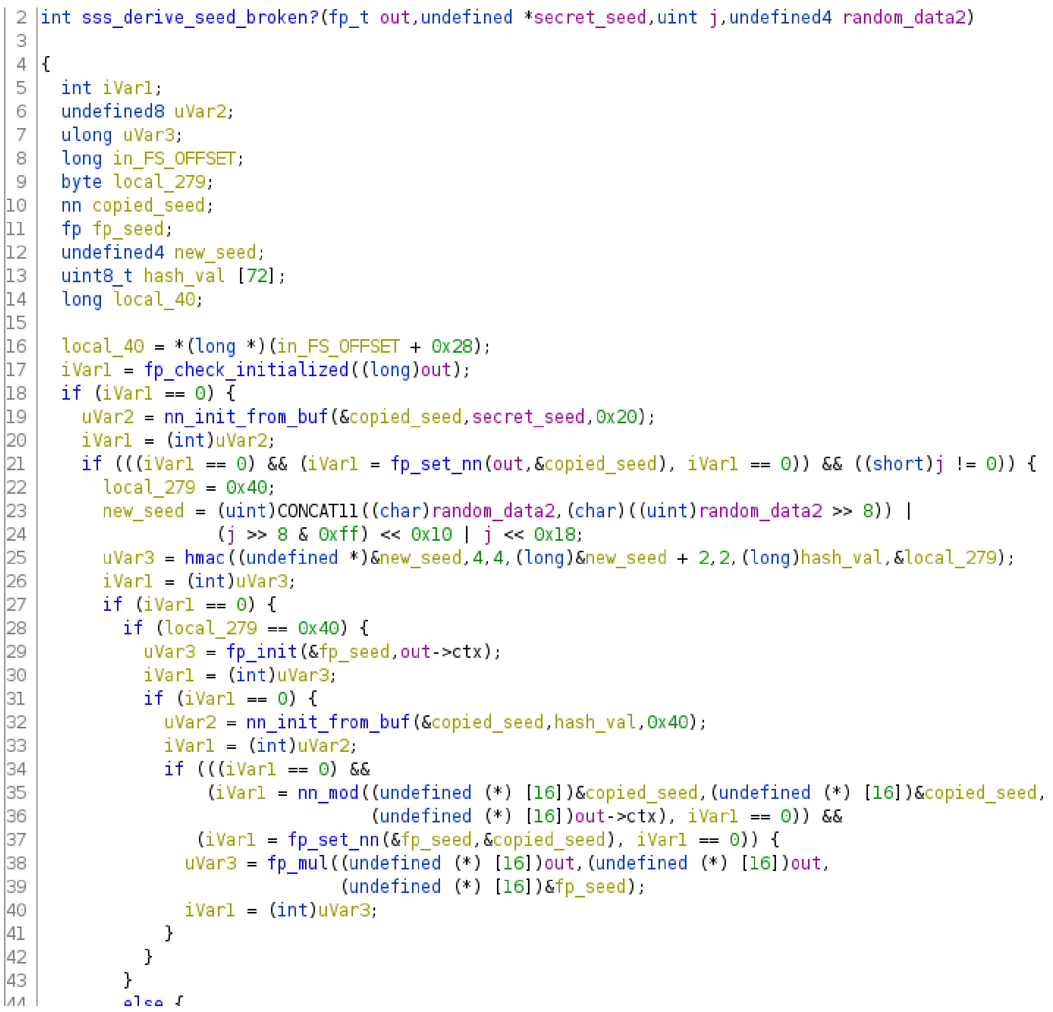
Where the function f is known (and encapsulates a hmac). Therefore:

$$P(\alpha) = s \times \sum_{i=0}^{k} f(j, r) \times \alpha^i$$

$$\iff \qquad s = P(\alpha) \times (\sum_{i=0}^{k} f(j, r) \times \alpha^i)^{-1}$$

α can be set to the index of the share and $P(\alpha)$ to the value of the share. Therefore the knowledge of r and of one share is enough to recover the secret_seed. In practice, we have to guess the value of r by looping through the 2^{16} possible values. Eventually, a hmac of the raw share (and the session index) is given with the raw share, so we can check the validity of the key by computing the hmac under this key and checking for hmac equality.

In order to minimize the possibilities of implementation errors during my attack, I placed my code into the sss.c source file of the library, and called the Makefile within the library to compile the sss. The code is given in the Share-it.c file. It should be included into the sss.c file and the function search should be called within the main function.



```
int sss_derive_seed_broken?(fp_t out,undefined *secret_seed,uint j,undefined4 random_data2)

{
  int iVar1;
  undefined8 uVar2;
  ulong uVar3;
  long in_FS_OFFSET;
  byte local_279;
  nn copied_seed;
  fp fp_seed;
  undefined4 new_seed;
  uint8_t hash_val [72];
  long local_40;

  local_40 = *(long *)(in_FS_OFFSET + 0x28);
  iVar1 = fp_check_initialized((long)out);
  if (iVar1 == 0) {
    uVar2 = nn_init_from_buf(&copied_seed,secret_seed,0x20);
    iVar1 = (int)uVar2;
    if (((iVar1 == 0) && (iVar1 = fp_set_nn(out,&copied_seed), iVar1 == 0)) && ((short)j != 0)) {
      local_279 = 0x40;
      new_seed = (uint)CONCAT11((char)random_data2,(char)((uint)random_data2 >> 8)) |
                 (j >> 8 & 0xff) << 0x10 | j << 0x18;
      uVar3 = hmac((undefined *)&new_seed,4,4,(long)&new_seed + 2,2,(long)hash_val,&local_279);
      iVar1 = (int)uVar3;
      if (iVar1 == 0) {
        if (local_279 == 0x40) {
          uVar3 = fp_init(&fp_seed,out->ctx);
          iVar1 = (int)uVar3;
          if (iVar1 == 0) {
            uVar2 = nn_init_from_buf(&copied_seed,hash_val,0x40);
            iVar1 = (int)uVar2;
            if (((iVar1 == 0) &&
                (iVar1 = nn_mod((undefined (*) [16])&copied_seed,(undefined (*) [16])&copied_seed,
                                (undefined (*) [16])out->ctx), iVar1 == 0)) &&
               (iVar1 = fp_set_nn(&fp_seed,&copied_seed), iVar1 == 0)) {
              uVar3 = fp_mul((undefined (*) [16])out,(undefined (*) [16])out,
                             (undefined (*) [16])&fp_seed);
              iVar1 = (int)uVar3;
            }
          }
        }
        else {
```

Figure 2: Interesting part of the decompiled **_sss_raw_generate** function.

Note for practical attack For optimization reasons, the multiplication function **fp_mul_monty** also multiplies with a constant factor (the Montgomery constant), depending on p and on the word size. I made sure to reuse the same number of **fp_mul_monty** as in the share-it binary.