S3 - Simple Secure System

Challenge

During an investigation we noticed that one of the employees used to this tool to encrypt some sensitive information. However, we were not able to recover the original information to see what has been leaked. Can you develop a decryptor for this?

Flag format: CTF{sha256}

My solution

Let's open `chall` in Ghidra. Here's the main function :

```
undefined8 FUN_00100aba(int param_1,long param_2)

{
  int iVar1;
  size_t sVar2;
  undefined8 uVar3;
  long lVar4;
  undefined8 *puVar5;
  undefined8 *puVar6;
  long in_FS_OFFSET;
  byte bVar7;
  EVP_PKEY *local_24f8;
  FILE *local_24f0;
  BIO *local_24e8;
  rsa_st *local_24e0;
  undefined8 local_24d8 [150];
  undefined8 local_2028 [514];
  undefined8 local_1018 [513];
  long local_10;

  bVar7 = 0;
  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  lVar4 = 0x95;
  puVar5 = &DAT_00100e20;
  puVar6 = local_24d8;
  while (lVar4 != 0) {
    lVar4 = lVar4 + -1;
    *puVar6 = *puVar5;
    puVar5 = puVar5 + 1;
    puVar6 = puVar6 + 1;
  }
  *(undefined *)puVar6 = *(undefined *)puVar5;
  lVar4 = 0x200;
  puVar5 = local_2028;
  while (lVar4 != 0) {
    lVar4 = lVar4 + -1;
```

```
      *puVar5 = 0;
      puVar5 = puVar5 + 1;
    }
    *(undefined2 *)puVar5 = 0;
    strlen((char *)local_2028);
    lVar4 = 0x200;
    puVar5 = local_1018;
    while (lVar4 != 0) {
      lVar4 = lVar4 + -1;
      *puVar5 = 0;
      puVar5 = puVar5 + (ulong)bVar7 * 0x1ffffffffffffffe + 1;
    }
    *(undefined2 *)puVar5 = 0;
    sVar2 = strlen((char *)local_1018);
    if (param_1 == 1) {
      uVar3 = 0xffffffff;
    }
    else {
      local_24f0 = fopen(*(char **)(param_2 + 8),"r");
      __isoc99_fscanf(local_24f0,&DAT_00100e02,local_1018);
      fclose(local_24f0);
      local_24e8 = BIO_new_mem_buf(local_24d8,0x4a9);
      if (local_24e8 == (BIO *)0x0) {
        uVar3 = 0xfffffffc;
      }
      else {
        local_24f8 = d2i_PrivateKey_bio(local_24e8,&local_24f8);
        if (local_24f8 == (EVP_PKEY *)0x0) {
          uVar3 = 0xfffffffd;
        }
        else {
          local_24e0 = EVP_PKEY_get1_RSA(local_24f8);
          if (local_24e0 == (rsa_st *)0x0) {
            uVar3 = 0xfffffffe;
          }
          else {
            iVar1 = RSA_check_key((RSA *)local_24e0);
            if (iVar1 == 0) {
              uVar3 = 0xffffffff;
            }
            else {
              iVar1 = RSA_public_encrypt((int)sVar2,(uchar *)local_1018,(uchar
  *)local_2028,
                                         (RSA *)local_24e0,1);
              local_24f0 = fopen("encrypted.txt","wb");
              fwrite(local_2028,1,(long)iVar1,local_24f0);
              fclose(local_24f0);
              RSA_free((RSA *)local_24e0);
              EVP_PKEY_free(local_24f8);
              BIO_free_all(local_24e8);
              uVar3 = 0;
            }
          }
        }
```

```
      }
    }
    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
      __stack_chk_fail();
    }
    return uVar3;
  }
```

Okay, so it has to do with RSA.

The functions are those from OpenSSL and so we will assume they are secure. Also, a useful thing to check : parameters orders. After checking the doc of each function, the dev who wrote this chall didn't mess up any param order.

So let's see how the main part of this program works, from the bottom (and ignoring the error cases) :

- `local_2028` is stored in the file `encrypted.txt`
- `local_1018` is encrypted with RSA, the ciphertext is put in `local_2028` and the RSA key is located at `local_24e0`
- The program checks if `local_24e0` is a valid RSA key
- `local_24e0` is initialized as the publicKey of `local_24f8`
- `local_24f8` is initialized as a privateKey from the BIO `local_24e8`
- `local_24e8` is initialized as a new BIO from `local_24d8` of length 0x4a9

So now we know `local_24d8` must contain the private key we need to decrypt the message. Let's look where it is initialized :

```
lVar4 = 0x95;
puVar5 = &DAT_00100e20;
puVar6 = local_24d8;
while (lVar4 != 0) {
  lVar4 = lVar4 + -1;
  *puVar6 = *puVar5;
  puVar5 = puVar5 + 1;
  puVar6 = puVar6 + 1;
}
*(undefined *)puVar6 = *(undefined *)puVar5;
```

This part of the code is a memory copy from `&DAT_00100e20` to `local_24d8` (of size 0x95, which I don't explain as clearly in memory you can see all of `DAT_00100e20` copied to `local_24d8`). Moreover, let's check the size of `DAT_00100e20` : 0x4a9. It is the same size as our BIO !

So... We might have found our key ! Let's extract it (using HxD), and google the first bytes : `30 82 04` (google : `"30 82 04" rsa`). It looks like it is indeed our key ! 😃 (see this article). A last check we can do in order to check the validity of our key is `openssl rsa -in rsakey.key -inform DER -text`, which validates our key.

The last step is just to decrypt the `encrypted2.txt` file, which is done via the python script attached.