Malware 3/3 (Reverse - 200 points) - bluesheet

Énoncé

Ouf ! Vous avez réussi à récupérer le malware, à vous connecter sur le serveur de l'attaquant et à récupérer la clé privée (fichier `key.priv` ci-joint) ayant servi à chiffrer votre précieux flag.

Le fichier `key.priv` portait initialement le nom : `0fdb0eea57198b3bb69e8267690ede5d5ba95ab791638a610372120b773d4acc_2021-03-15|21:34:41.priv`.

Dechiffrez le fichier `flag.txt` pour valider cette épreuve.

Résolution

Examinons les fichiers à notre disposition :

- Une clé privée RSA,
- Un fichier `flag.txt` chiffré,
- Un exécutable ELF x86, le malware Command & Control.

Ouvrons l'exécutable avec Ghidra afin de comprendre comment le fichier `flag.txt` a été chiffré.

Rétro-ingénierie statique du code côté client

Je cherche la chaîne de caractères `"flag.txt"` dans l'exécutable, pour remonter à ses cross-references (là où elle est utilisée). J'atterrirai probablement proche de l'endroit où le fichier est chiffré.

```
longueur_cle = strlen(cle_recue);
key = BIO_new_mem_buf(cle_recue,(int)longueur_cle);
rsa_ctx = RSA_new();
PEM_read_bio_RSA_PUBKEY(key,&rsa_ctx,(undefined1 *)0x0,(void *)0x0);
BIO_free(key);
username = get_username();
local_28 = concat("/home/",username);
flag_path = concat(local_28,"/Bureau/flag.txt");
flag_content = (uchar *)read_file(flag_path);
if (flag_content == (uchar *)0x0) {
                    /* WARNING: Subroutine does not return */
  exit(1);
}
iVar1 = RSA_size(rsa_ctx);
flag_cipher = (uchar *)malloc((long)iVar1);
rsa_bis = rsa_ctx;
longueur_cle = strlen((char *)flag_content);
len_flag_cipher = RSA_public_encrypt((int)longueur_cle,flag_content,flag_cipher,rsa_bis,4);
RSA_free(rsa_ctx);
longueur_cle = strlen((char *)flag_content);
memset(flag_content,0,longueur_cle);
FUN_0040349e(flag_cipher,flag_path,len_flag_cipher,flag_path);
return;
```

Au cours de la rétro-ingénierie, j'essaie de renommer autant que possible les variables et les fonctions, en leur donnant des noms plus descriptifs que `local_XX` ou `iVarX`. Ici, j'ai procédé de la manière suivante :

1. Regarder la signature de la fonction `PEM_read_bio_RSA_PUBKEY` (dont le symbole était déjà présent dans l'exécutable), et renommer les paramètres de cette fonction de manière cohérente (`key` , `RSA_ctx`).
2. Renommer les paramètres de `BIO_new_mem_buf` correspondant à l'affectation de `key` .
3. Renommer `longueur_cle` de manière cohérente, étant donné l'affectation à `strlen(cle_recue)` .
4. Inspecter et comprendre le contenu de la fonction `get_username` , qui ne s'appelait pas encore `get_username` , mais dont le contenu est assez explicite :

```
char * get_username(void)

{
  char *__name;

  __name = (char *)malloc(0x101);
  getlogin_r(__name,0x101);
  return __name;
}
```

5. Idem avec la fonction `concat` :

```
char * concat(char *param_1,char *param_2)

{
  size_t sVar1;
  size_t sVar2;
  char *__dest;

  sVar1 = strlen(param_1);
  sVar2 = strlen(param_2);
  __dest = (char *)malloc(sVar2 + sVar1 + 1);
  strcpy(__dest,param_1);
  strcat(__dest,param_2);
  return __dest;
}
```

6. Renommer `flag_path` de manière cohérente avec son contenu.
7. Examiner la signature de la fonction `RSA_public_encrypt` et renommer les paramètres de manière cohérente. *Notons ici le dernier paramètre, correspondant au padding, fixé à la constante **4**.* Aussi, la variable `longueur_cle` est réutilisée par le compilateur pour stocker la logueur du flag.
8. Je me rends compte au moment de l'écriture de ce write-up que je n'ai pas renommé la fonction `FUN_0040349e` . Voici son contenu :

```
void FUN_0040349e(void *param_1,char *param_2,int param_3)

{
  FILE *__s;

  __s = fopen(param_2,"w+");
  if (__s == (FILE *)0x0) {
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  fwrite(param_1,1,(long)param_3,__s);
  fclose(__s);
  return;
}
```

Je choisirai donc de la renommer `write_file`.

Voilà, cette fonction est maintenant plus propre, et on commence à avoir une idée de ce qui arrive à notre flag : il est chiffré avec une clé publique RSA, qui est passé en paramètre de la fonction sur laquelle nous travaillons, que je vais nommer `encrypt_flag`.

Regardons maintenant les cross-references de notre fonction `encrypt_flag`, pour trouver l'origine de la clé publique RSA passée en paramètre.

```
username = get_username();
hostname = get_hostname();
username_at = concat(username,&at);
username_at_hostname = (char *)concat(username_at,hostname,hostname);
SHA256_Init(&sha_ctx);
sVar1 = strlen(username_at_hostname);
SHA256_Update(&sha_ctx,username_at_hostname,sVar1);
SHA256_Final(sha_u_at_h,&sha_ctx);
```

[…]

```
temps = time((time_t *)0x0);
srand((uint)temps);
random_seeded = rand();
rand_mod_1000 = random_seeded % 1000;
log_rand_mod_1000 = log10((double)rand_mod_1000);
local_58 = (int)(log_rand_mod_1000 + 1.0);
rand_mod_1000_ascii_int = (char *)malloc((long)local_58);
sprintf(rand_mod_1000_ascii_int,"%d",(ulong)rand_mod_1000);
local_68 = concat(sha_hex,&point_virgule);
sha_hex_pv_rand_mod_1000 =
     (char *)concat(local_68,rand_mod_1000_ascii_int,rand_mod_1000_ascii_int);
sVar1 = strlen(sha_hex_pv_rand_mod_1000);
sVar2 = send(sock,sha_hex_pv_rand_mod_1000,sVar1,0);
local_74 = (int)sVar2;
```

[…]

```
sVar2 = recv(sock,local_488,0x3ff,0);
local_74 = (int)sVar2;
if (local_74 < 1) {
                    /* WARNING: Subroutine does not return */
  exit(1);
}
local_488[local_74] = 0;
encrypt_flag();
close(sock);
sleep(10000);
return;
```

Extraits de la fonction parente de `encrypt_flag`, variables et fonctions renommées par une méthode similaire à celle détaillée plus haut.

On remarque une erreur de désassemblage lors de l'appel à `encrypt_flag`, mais `local_488` (qui pourrait en réalité s'appeler `donnees_recues`, voire `cle_recue`) semble un bon candidat pour être son paramètre. Ainsi, la fonction parente effectue les actions suivantes :

1. Hacher la chaîne `username@password` avec `SHA256` (on a vu dans `Malware 1/3` qu'il s'agissait de `forensics@fcsc2021`), concatène le digest avec un `;`, puis avec un entier entre **0** et **1000** généré par une fonction `rand()`, seedée par le timestamp de l'instant d'exécution.
2. Envoyer cette chaîne à travers la socket (vers la machine de l'attaquant donc)
3. Recevoir par la socket la clé publique qui servira à chiffrer le flag.

Logiquement, ma prochaine étape va consister en l'examen du code côté serveur, pour déterminer comment la paire de clés RSA est générée.
Pour cela, je vais regarder les cross-references de la fonction `send`. Cette dernière est référencée deux fois, donc une que l'on vient d'examiner. Allons jeter un oeil sur cette seconde cross-reference !

Rétro-ingénierie statique du code côté serveur

```
pv_pos = memrchr(param_2,0x3b,param_3);
pv_pos_int = (long)(int)pv_pos;
if (pv_pos_int == 0) {
  uVar1 = 0;
}
else {
  pv_ind = (int)pv_pos - (int)param_2;
```

[…] *(0x3b correspond au caractère ';'… `param_2` est certainement la chaîne envoyée par le client.)*

```
sha_hex[pv_ind] = '\0';
len_nombre_apres_pv = strlen((char *)((long)param_2 + (long)pv_ind + 1));
nb_apres_pv = (char *)malloc(len_nombre_apres_pv);
strcpy(nb_apres_pv,(char *)((long)pv_ind + 1 + (long)param_2));
__isoc99_sscanf(nb_apres_pv,&DAT_0040420a,&int_nb_apres_pv);
if (int_nb_apres_pv % 1000 == int_nb_apres_pv) {
  local_40 = time((time_t *)0x0);
  time_decale_de_UTC = localtime(&local_40);
  strftime(str_time_decale_de_utc,0x1a,"%Y-%m-%d|%H:%M:%S",time_decale_de_UTC);
  key_password = generate_password(sha_hex,str_time_decale_de_utc,int_nb_apres_pv,
                                   str_time_decale_de_utc);
  public_key = (char *)generate_RSA_keys(key_password,sha_hex,str_time_decale_de_utc,sha_hex);
  if (public_key == (char *)0x0) {
    uVar1 = 0;
  }
  else {
    len_nombre_apres_pv = strlen(public_key);
    sVar2 = send(socket,public_key,len_nombre_apres_pv,0);
```

Extraits de la fonction qui référence `send`.

A noter : à ce stade lors de ma résolution, la fonction `generate_password` n'était pas encore nommée. La fonction `generate_RSA_keys` est par contre très lisible, et son utilité comme ses paramètres sont facilement déterminables.

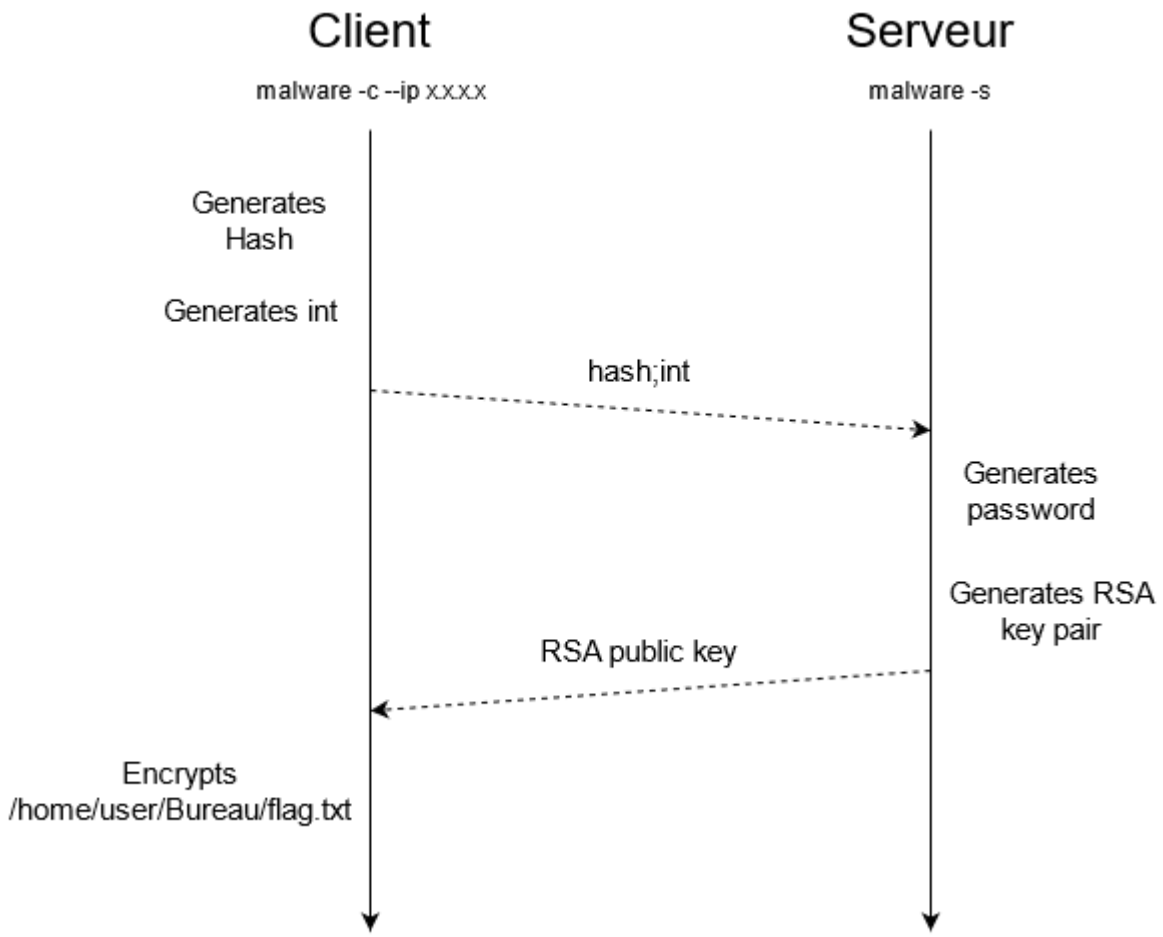
```
undefined8 generate_RSA_keys(undefined8 password,undefined8 sha_hex,undefined8 time_string)

{
  local_10 = concat("/keys/",sha_hex);
  local_18 = concat(local_10,&underscore);
  local_20 = (char *)concat(local_18,time_string,time_string);
  chemin_cle_privee = (char *)concat(local_20,".priv");
  chemin_cle_publique = (char *)concat(local_20,&DAT_00404laf);
  pass_pass = concat("pass:",password);
  iVar1 = access(chemin_cle_privee,0);
  if (iVar1 != 0) {
    local_f8 = "openssl";
    local_f0 = "genrsa";
    local_e8 = "-aes256";
    local_e0 = &dash_out;
    local_d8 = chemin_cle_privee;
    local_d0 = "-passout";
    local_c8 = pass_pass;
    local_c0 = &DAT_4096;
    local_b8 = 0;
    exec(&local_f8);
    local_a8 = "openssl";
    local_a0 = &DAT_00404le4;
    local_98 = &DAT_00404le8;
    local_90 = chemin_cle_privee;
    local_88 = "-passin";
    local_80 = pass_pass;
    local_78 = "-pubout";
    local_70 = &dash_out;
    local_68 = chemin_cle_publique;
    local_60 = 0;
    exec(&local_a8);
  }
  local_58.actime = 0;
  local_58.modtime = 0;
  utime(local_20,&local_58);
  utime(chemin_cle_privee,&local_58);
  utime(chemin_cle_publique,&local_58);
  local_40 = read_file(chemin_cle_publique);
  remove(chemin_cle_publique);
  return local_40;
}
```

Je sais maintenant d'où provient le nom de la clé privée donné dans l'énoncé ! Une rapide vérification confirme que `SHA256("forensics@fcsc2021") = 0fdb0eea57198b3bb69e8267690ede5d5ba95ab791638a610372120b773d4acc`
Attention toutefois, la date indiquée dans le nom de la clé est générée par le **serveur**, et le petit entier envoyé par le client est généré par le **client**, il y a donc un délai entre les deux (principalement causé par la latence sur le réseau). De plus, la date affichée dans le nom de la clé passe par une fonction adaptant la date au fuseau horaire du serveur, là où le timestamp du client est fixé sur UTC. Aussi, l'entier envoyé par le client n'est pas très grand (entre 0 et 999). Pour cela, j'ai décidé de ne pas me fier à la date générée par le serveur, et de plutôt tester tous les entiers possibles (plutôt que de tenter de seeder le random côté client à date - 3600, puis -7200, puis -3601, puis …)

Reste à décortiquer la fonction `generate_password`, que j'ai pu nommer en remarquant que sa sortie était utilisée comme mot de passe pour la clé privée RSA.

Mais avant cela, il me semble intéressant de réaliser un bilan sur l'architecture de ce malware.

Architecture du Malware



L'exécutable dispose des fonctionnnalités de client et de serveur. Lorsqu'il est lancé en mode client (avec le flag `-c`), il envoie à l'IP spécifiée (par le flag `--ip`) une chaîne de caractère composée d'un hash (qui sert d'identifiant unique) et un entier (qui servira à générer le mot de passe de la clé privée). Il attend ensuite la réception d'une clé publique RSA, puis chiffre le fichier `flag.txt` à l'aide de cette clé.
Lorsqu'il est en mode serveur (flag `-s`), il écoute l'arrivée de nouveaux clients, génère le couple de clés RSA en fonction de la chaine envoyée par le client, et renvoie la clé publique RSA au client.

Rétro-ingénierie statique de `generate_password`

```
i = 0xfa;
puVar2 = &chaine_bizarre;
copy_bizarre = &chaine_bizarre_xor_1;
while (i != 0) {
  i = i + -1;
  *copy_bizarre = *puVar2;
  puVar2 = puVar2 + 1;
  copy_bizarre = copy_bizarre + 1;
}
*(undefined2 *)copy_bizarre = *(undefined2 *)puVar2;
*(undefined *)((long)copy_bizarre + 2) = *(undefined *)((long)puVar2 + 2);
j = 0;
while( true ) {
  len_bizarre = strlen((char *)&chaine_bizarre_xor_1);
  if (len_bizarre <= j) break;
  *(byte *)((long)&chaine_bizarre_xor_1 + j) = *(byte *)((long)&chaine_bizarre_xor_1 + j) ^ 1;
  j = j + 1;
}
buffer_taille_0xAB = (char *)malloc(0xab);
```

Début de la fonction `generate_password`

Pour l'instant rien de très compliqué, une chaîne de caractère étrange est copiée depuis la section `.data` dans la stack, et tous les caractères de cette chaîne sont XORés avec **0x1**.
S'en suit un appel à la fonction `FUN_004037c0`, qui est une sorte de machine à états qui avance sur cette chaîne de caractères. Je la mets de côté pour l'instant et continue la lecture de `generate_password`.

```
FUN_004037c0(&chaine_bizarre_xor_1,buffer_taille_0xAB,buffer_taille_0xAB);
local_38 = malloc(0xab);
local_14 = 0;
k = 0;
while( true ) {
  len_bizarre = strlen(buffer_taille_0xAB);
  if (len_bizarre - 8 <= k) break;
  premier_bit_de_chaque_case =
       buffer_taille_0xAB[k + 7] & 1U |
       (byte)((int)buffer_taille_0xAB[k] << 7) |
       (byte)(((int)buffer_taille_0xAB[k + 1] & 1U) << 6) |
       (byte)(((int)buffer_taille_0xAB[k + 2] & 1U) << 5) |
       (byte)(((int)buffer_taille_0xAB[k + 3] & 1U) << 4) |
       (byte)(((int)buffer_taille_0xAB[k + 4] & 1U) << 3) |
       (byte)(((int)buffer_taille_0xAB[k + 5] & 1U) << 2) |
       buffer_taille_0xAB[k + 6] * '\x02' & 2U;
  local_24 = SEXT14((char)premier_bit_de_chaque_case);
  if (0x7e < local_24) {
    local_24 = local_24 + ((local_24 >> 1) / 0x3f) * -0x7e;
  }
  if (local_24 < 0x20) {
    local_24 = local_24 + 0x20;
  }
  *(char *)((long)local_38 + (long)local_14) = (char)local_24;
  local_14 = local_14 + 1;
  k = k + 8;
}
```

Les complications…

Les choses se compliquent, mais sont encore faisables. Je continue de lire la fonction jusqu'au bout, hisoire de m'aider dans mon renommage futur.

```
sprintf(local_878,"%d",(ulong)randint_apres_pv);
local_40 = concat(local_38,sha_hex,sha_hex);
local_48 = concat(local_40,strtme_decale_de_utc,strtme_decale_de_utc);
local_38_sha_hex_strtme_decale_utc_randint = (char *)concat(local_48,local_878,local_878);
len_bizarre = strlen(local_38_sha_hex_strtme_decale_utc_randint);
len_string_magique = (int)len_bizarre;
buffer_string_magique = calloc((long)len_string_magique,4);
len_buffer_charcount =
     fonction_python_1(local_38_sha_hex_strtme_decale_utc_randint,buffer_string_magique,
                       len_string_magique,buffer_string_magique);
problemes_divisions(buffer_string_magique,len_buffer_charcount,len_string_magique,
                    len_buffer_charcount);
local_70 = extraout_XMM0_Qa;
local_78 = malloc(0xb);
snprintf(local_70,(size_t)local_78,(char *)0xb,&DAT_00404238);
uVar1 = concat(local_38_sha_hex_strtme_decale_utc_randint,local_78,local_78);
return uVar1;
```

C'est ici que j'ai réalisé que je n'arriverai pas au bout du reversing de cette fonction… On peut voir `fonction_python`, une fonction dont j'ai tenté de répliquer le fonctionnement en python, mais surtout `problemes_divisions`, une fonction qui mène au bout de code suivant :

```
float problemes_divisions_bis(uint param_1)

{
  ulong local_10;

  local_10 = (ulong)param_1;
  local_10._0_4_ = (float)((param_1 & 0x807fffff) + 0x3f800000);
  return ((float)local_10 * ((float)local_10 * -0.3448484 + 2.024666) - 0.6748776) +
         (float)(((long)local_10 >> 0x17 & 0xffU) - 0x80);
}
```

Cette fonction, je n'arriverai pas à la reverse… En somme, je ne suis pas capable de reverse la fin de la fonction `generate_password`.

MAIS

Je sais qu'elle n'a pas de side-effects, et je connais ses paramètres ! Je peux donc directement l'appeler dans `gdb` pour obtenir le password généré !

`generate_password` - la revanche…

Scripting time !

```
import gdb
gdb.execute("b *0x402687", False, False) # breakpoint à l'entrée de main
gdb.execute("b *0x403cb7", False, False) # breakpoint au return de generate_password
gdb.execute("r", False, False)

passlist = ""

begin = 0
end = 1000
```

```
for i in range(begin, end):
    passlist += str(i) + " : "
    for k in range(2): # jsp pourquoi, mais appeler la fonction une seule fois ne marche pas
        # On doit faire un try catch parce que breakpoint = exception
        try:
            gdb.execute("p ((char* (*) (char*, char*, int)) (0x40396d)) (\"0fdb0eea57198b3bb69e8267690ede5d5ba95ab79163
        except:
            pass
    cle = gdb.execute("print (char*)$r8", False, True)
    passlist += cle + "\n"


f = open("gdb_passwords_"+str(begin)+"-"+str(end)+".txt", "w")
f.write(passlist)
f.close()
```

Comme dit plus haut, quitte à tester un entier, autant tester les 1000 ! Ce n'est pas beaucoup plus long, et m'assure que le mot de passe sera dans le fichier généré.

Récupération de la clé privée

Ainsi, j'enchaîne sur un one-liner bash, en travaillant sur une version épurée du fichier généré :

```
while read -r line; do echo $line && openssl rsa -in key.priv -passin pass:"$line" -check 2> /dev/null; done < password
```

J'obtiens un retour pour un unique password (`n = 578`). La clé privée est déchiffrée !!

Déchiffrement de `flag.txt` avec la clé prifée

Il ne reste plus qu'à déchiffrer le flag avec la clé privée, en se rappelant bien de la constante de padding rencontrée dans l'exécutable : **4**. Celle-ci correspond, d'après les headers, à la constante nommée `RSA_PKCS1_OAEP_PADDING`. Je rédige donc un rapide dernier script afin de déchiffrer le fichier `flag.txt` :

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

key = RSA.importKey(open('key.decrypt').read())
cipher = PKCS1_OAEP.new(key)

f = open("flag.txt", "rb")
raw = f.read()
f.close()
message = cipher.decrypt(raw)
f = open('decoded_flag.txt', 'wb+')
f.write(message)
f.close()
```

Et **le tour est joué !** Le flag apparaît, en caractères gothiques.