esiea
ECOLE D'INGENIEURS
DU MONDE NUMERIQUE

RAPPORT
Reverse Engineering

Vincent Drouin
vdrouin@et.esia.fr

SOMMAIRE

1 – EZ.EXE.....3

2 – HACHOIR.EXE.....6

3 – CYPHERME.EXE.....8

ANNEXE 1 - HACHOIR.C.....10

ANNEXE 2 – CYPHERME_AES_DECRYPT.PY.....12

ANNEXE 3 – CYPHERME_AES_DECRYPT.C.....13

ANNEXE 4 - QUESTIONNAIRE.....15

1 – EZ.exe

Concept général :

Ce programme consiste en une succession de défis sous la forme de questions posées à l'utilisateur, qui doit apporter une réponse correcte à chaque étape pour accéder au niveau suivant.

Dans la fonction _main du programme on constate la présence d'un « switch case » sur une variable nommée var_48.
Ce switch appelle exécute une partie spécifique de code pour les valeurs de 0 à 4.

La variable vaut -1 au premier tour ce qui fait s'exécuter la section par défaut.

A chaque étape passée avec succès, var_48 est incrémentée : il y a donc 6 étapes en tout.

En cas d'échec à une étape, l'exécution du programme s'arrête.

Etape « default » - fonction sub_401000

A cette étape, on constate la présence d'une variable (Str2) contenant la chaîne de caractères « TANK ».
Cette fonction prend une chaîne de caractères en paramètre (Str1).

Par la suite :
1. Vérification que Str1 et Str2 ont la même taille
2. Comparaison la chaîne de caractères entrée par l'utilisateur (Str1)

La valeur à entrer pour valider cette étape est donc « TANK ».

Etape 0 - fonction sub_4010C0

Cette fonction prend une chaîne de caractères en paramètre (Str) et utilise une variable var_18 qui contient la valeur 0x19C, soit 412 en décimal.

Par la suite :
1. Vérification que Str n'est pas vide
2. Conversion de Str en entier en appelant la fonction atoi
3. Comparaison du résultat avec var_18

La valeur à entrer pour valider cette étape est donc 412.

Etape 1 - fonction sub_401130

Cette fonction prend une chaîne de caractères en paramètre (Str) et utilise une variable var_18 qui contient 0x0D1CECA5E.

Par la suite :

1. Vérification que Str n'est pas vide
2. Conversion de Str en entier de 32 bits en appelant la fonction strtol
3. XOR de l'entier calculé avec la valeur 0x0BAADBAAD
4. Comparaison du résultat avec var_18

La valeur à entrer est donc le XOR de 0x0D1CECA5E et 0x0BAADBAAD, soit 0x6B6370F3, sous forme de chaîne de caractères 6B6370F3.

Etape 2 - fonction sub_4011B0

Après avoir vérifié que la chaîne de caractères Str passée en paramètre était non nulle, la fonction procède ainsi :

1. Conversion de Str en entier en utilisant la fonction atoi, le résultat est mis dans var_C
2. Appel de la fonction srand avec la graine 0x6B6370F3
3. Appel de la fonction rand(), le résultat est mis dans la variable var_10
4. Comparaison de var_10 et de var_C

En analysant le contenu de var_10 dans la mémoire à l'exécution, on en déduit que la valeur attendue est 31445.

Etape 3 - fonction sub_401260

La fonction s'assure d'abord que la chaîne de caractères passée en paramètre contient au moins 10 caractères, incluant le « \0 » de fin.

En analysant le contenu de la mémoire à l'exécution on se rend compte que :

- Le premier caractère doit être « N » (comparaison avec la valeur de la variable var_104)
- Il y a un mécanisme de substitution de caractères : lorsqu'on soumet une chaîne de caractères si un même caractère apparaît plusieurs fois, celui-ci apparaîtra une fois inchangé pour la première occurrence, puis sera substitué par l'indice d'apparition de la première occurrence de ce caractère dans la chaîne les fois suivantes.
  Exemple : « NaaaZbcEEEaf » est transformée en la représentation hexadécimale de « Na11ZbcE771f »

On remarque qu'il y a un grand nombre de variables pré-initialisées à des valeurs arbitraires en début de fonction, qui restent inchangées tout au long de l'exécution.

Eprenant en considération qu'on utilise une de ces variables (var_104) pour faire des tests sur le buffer calculé à partir de l'entrée utilisateur, on en déduit qu'il est

intéressant d’analyser l’ensemble des variables pré-initialisées, et d’y appliquer l’algorithme de substitution « inverse », en partant de var_104, supposée être la première lettre attendue.

En faisant une traduction ASCII et en inversant l’algorithme de substitution, on obtient rapidement « Nebuchadnezzar »

Etape 4 - fonction sub_4019E0

Dans cette dernière étape on revient à un fonctionnement assez classique de conversion d’une chaîne de caractères passée en paramètre en un entier qu’on va comparer avec une valeur attendue.

La subtilité réside dans le calcul de la fameuse « valeur attendue » : on peut remarquer qu’on a 17 (0x11) tours d’une boucle correspondant à la suite de Fibonacci.

La valeur attendue est donc Fibonacci(16), soit 987.

2 – HACHOIR.exe

Concept général :

Il s'agit d'un algorithme de hachage, qui attend une chaîne de caractères en entrée, calcule le hash et le vérifie.

Pour retrouver quelle chaîne de caractères est attendue, il faut inverser les fonctions utiles au hachage.

La fonction sub_40120 se limite à initialiser le buffer qui sera utilisé par la suite à partir de l'entrée utilisateur au moyen d'un fgets : il est inutile de l'inverser.

Le buffer généré contiendra 3 entiers de 32 bits, qui seront la représentation hexadécimale de la chaîne de caractères donnée par l'utilisateur.

La valeur du hash final est 0E8C01B7 2FB067A8 69E0BBE7.

Pour le codage en C du déchiffement du hash et l'exécution sous linux, il faut prendre garde au format qui est « little endian », et donc l'ordre des octets de chacun de ces entiers sera inversé pour exécuter le programme du reverse proprement.

Fonction sub_401000 :

Cette fonction prend en argument un entier de 32 bits (arg1) et un index (arg2) et effectue les étapes suivantes :

- Rotation à gauche de 12 bits de arg1
- Arg1 = arg1 + arg2
- Inversion de arg1
- Rotation à droite de 7 bits de arg1
- Arg1 = arg1 – 0x453B698E
- Arg1 = arg1 XOR 0x1337

Fonction sub_401110 :

Cette fonction prend pour paramètre le buffer, et le modifie par ses 3 appels à la fonction sub_401000 :

1. Buffer[0] = sub_401000(buffer[0], 0)
2. Buffer[1] = sub_401000(buffer[1], 1)
3. Buffer[2] = sub_401000(buffer[2], 2)

Fonction sub_401190 :

Cette fonction prend pour paramètre le buffer, et effectue un calcul sur chacune de ses valeurs :

1. Buffer[0] = buffer[0] XOR 0x444E4152
4. Buffer[1] = buffer[1] XOR 0x52304D4F
5. Buffer[2] = buffer[2] XOR 0x21444545

_main :

Après initialisation du buffer, le programme appelle 3 fois de suite la fonction sub_401110, puis une fois la fonction sub_401190 pour calculer le hash à partir du clair.

Après inversion de ces fonctions et du main, l'exécution du code C correspondant nous permet de déduire que le message clair à fournir en entrée doit être le suivant : 87A125F5DB5C

3 – CYPHERME.exe

Concept général :

Il s'agit d'un programme de chiffrement, prenant en entrée un mot de passe ainsi qu'un fichier à chiffrer, et construisant le fichier chiffré final à partir des données entrées.

L'utilisation de l'utilitaire signsrch a permis d'identifier divers types d'algorithmes cryptographiques dont les signatures apparaissent dans le programme :

- AES est l'algorithme utilisé pour le chiffrement
- MD5 et RIPEMD-128 semblent être utilisés, probablement pour le calcul du hash.

Le langage de programmation utilisé semble être C++, car de nombreuses librairies et fonctions C++ issues de la WINAPI sont utilisées dans le programme.

Des exécutions successives permettent de se rendre compte qu'un même fichier clair aboutira à un chiffré différent.

Fonction sub_403600 :

Cette fonction est le point d'entrée du processus de chiffrement.
En analysant la mémoire à l'exécution, on peut récupérer le hash du mot de passe, et le vecteur d'initialisation utilisés lors de l'exécution en cours.

Mot de passe :

Un calcul du hash du mot de passe passé en clair permet de confirmé que l'algorithme de hash utilisé est MD5.

Vecteur d'initialisation :

Après plusieurs exécutions, on constate que le vecteur d'initialisation change à chaque fois.
La fonction GenerateRandom semble être utilisée dans une fonction appelée par le programme principal pour initialiser cette valeur de manière aléatoire.

Analyse du fichier chiffré généré :

Par la suite, en analysant le fichier créé avec l'utilitaire hexdump, on constate que le hash du mot de passe se trouve sur les 16 premiers octets, et le vecteur d'initialisation se trouve sur les 16 octets suivants.

Conclusion :

Étant donné qu'on a le vecteur d'initialisation, le hash du mot de passe (qui est la clé de chiffrement et déchiffrement), et l'algorithme utilisé (AES), on peut tester de déchiffrer l'image en mode AES CBC, et si cela ne fonctionne pas, avec le mode CTR.
Le mode CBC permet d'obtenir l'image déchiffrée.

Note : l'image déchiffrée n'est pas entière, car il manque les blocs résiduels, AES n'acceptant que des blocs de taille 128.

Un algorithme complémentaire doit être présent dans le code pour obtenir ces blocs clairs résiduels.

ANNEXE 1 - hachoir.c

```
#include <stdio.h>
#include <stdlib.h>

#define N 3

unsigned int _rotl(const unsigned int value, int shift) {
    return (value << shift) | (value >> (32 - shift));
}

unsigned int _rotr(const unsigned int value, int shift) {
    return (value >> shift) | (value << (32 - shift));
}

void sub_401190_reverse(unsigned int *buf){
    buf[0] ^= 0x444E4152;
    buf[1] ^= 0x53204D4F;
    buf[2] ^= 0x21444545;
}

unsigned int sub_401000_reverse(unsigned int input, int iter){
    unsigned int res = input;
    res ^= 0x1337;
    res += 0x453B698E;
    res = _rotl(res, 7);
    res = ~res;
    res -= iter;
    res = _rotr(res, 13);

    return res;
}

void sub_401110_reverse(unsigned int *buf, int iter){
    int i;
    for(i = 0; i < 3; i++)
    {
        buf[i] = sub_401000_reverse(buf[i], iter);
    }
}

int main(int argc, char **argv)
{
    int i = 0;
//    unsigned int hash[N] = {0xE7BBE069,0xA867B02F,0xB7018C0E};
    unsigned int hash[N] = {0xB7018C0E,0xA867B02F,0xE7BBE069};
    //unsigned int hash[N] = {0x69E0BBE7,0x2FB067A8,0x0E8C01B7};
    //unsigned int hash[N] = {0x0E8C01B7,0x2FB067A8,0x69E0BBE7};
```

```
    for(i = 0; i < N; i++)
        printf("0x%08x\n", hash[i]);

    sub_401190_reverse(hash);
    for(i = N-1; i >= 0; i--)
        sub_401110_reverse(hash, i);


    for(i = 0; i < N; i++)
        printf("0x%08x\n", hash[i]);
    return 0;
}
```

ANNEXE 2 – cypherme_aes_decrypt.py

```
#!/usr/bin/env python

from Crypto.Cipher import AES
import Crypto.Cipher.AES


if __name__ == '__main__':

    crypt_file = open('secret.jpg.bin','rb')
    clear_file = open('secret.jpg','wb')

    key = crypt_file.read(16)
    IV = crypt_file.read(16)
    print key
    print IV
    decipher = AES.new(key, AES.MODE_CBC, IV)
    ciphertext = crypt_file.read(16)
    while ciphertext != "":
        plaintext = decipher.decrypt(ciphertext)
        (plaintext).encode('hex')
        clear_file.write(plaintext)
        ciphertext = crypt_file.read(16)

    crypt_file.close()
    clear_file.close()
```

ANNEXE 3 – cypherme_aes_decrypt.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/evp.h>
#include <openssl/aes.h>

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif


/**
 *  * Encrypt or decrypt, depending on flag 'should_encrypt'
 *   */
void en_de_crypt(int should_encrypt, FILE *ifp, FILE *ofp, unsigned char *ckey,
unsigned char *ivec) {

    const unsigned BUFSIZE=4096;
    unsigned char *read_buf = malloc(BUFSIZE);
    unsigned char *cipher_buf;
    unsigned blocksize;
    int out_len;
    EVP_CIPHER_CTX ctx;

    EVP_CipherInit(&ctx, EVP_aes_128_cbc(), ckey, ivec, should_encrypt);
    blocksize = EVP_CIPHER_CTX_block_size(&ctx);
    cipher_buf = malloc(BUFSIZE + blocksize);

    fseek(ifp,32,SEEK_SET);
    while (1) {
        // Read in data in blocks until EOF. Update the ciphering with each read.
        int numRead = fread(read_buf, sizeof(unsigned char), BUFSIZE, ifp);
        EVP_CipherUpdate(&ctx, cipher_buf, &out_len, read_buf, numRead);
        fwrite(cipher_buf, sizeof(unsigned char), out_len, ofp);
        if (numRead < BUFSIZE) { // EOF
            break;
        }
    }

    // Now cipher the final block and write it out.
```

```
    EVP_CipherFinal(&ctx, cipher_buf, &out_len);
    fwrite(cipher_buf, sizeof(unsigned char), out_len, ofp);
    free(cipher_buf);
    free(read_buf);
}

int main(int argc, char *argv[]) {

    unsigned char ckey[16];
    unsigned char ivec[16];
    FILE *fIN, *fOUT;
    int i;

    if (argc != 2) {
        printf("Usage: <executable> <filename>");
        return -1;
    }

    //Decrypt file now

    fIN = fopen(argv[1], "rb"); //File to be read; cipher text
    for(i = 0; i < 16; i++)
    {
        fread(ckey+i, sizeof(char), 1, fIN);
    }

    fseek(fIN,16,SEEK_SET);
    for(i = 0; i < 16; i++)
    {
        fread(ivec+i, sizeof(char), 1, fIN);
    }
    fOUT = fopen("decrypted", "wb"); //File to be written; cipher text

    en_de_crypt(FALSE, fIN, fOUT, ckey, ivec);

    fclose(fIN);
    fclose(fOUT);

    return 0;
}
```

ANNEXE 4 - Questionnaire

- Avez-vous déjà fait quelques crackmes ?
Non

- Avez-vous déjà quelques connaissances en assembleur ?
Les bases apprises lors du cours d'introduction à l'assembleur.

- Combien de temps avez-vous passé à faire le TP ?
Environ 30h.

- Le TP le plus facile? Le plus difficile?
Le TP le plus facile était hachoir, le plus difficile était cypherme.

- Avez-vous trouvé l'intervention intéressante?
J'étais absent le jour du cours.

- Est-ce que les cours étaient adaptés? Trop court? Trop long? Manque de clarté etc.?
J'étais absent le jour du cours.

- Vos remarques/commentaires
Merci pour votre aide et vos conseils lors du TP.

- Avez-vous kiffé votre race?
C'était pas mal ouais :)

- Allez-vous continuer à en faire à titre perso?
Si j'arrive à dégager du temps, certainement.