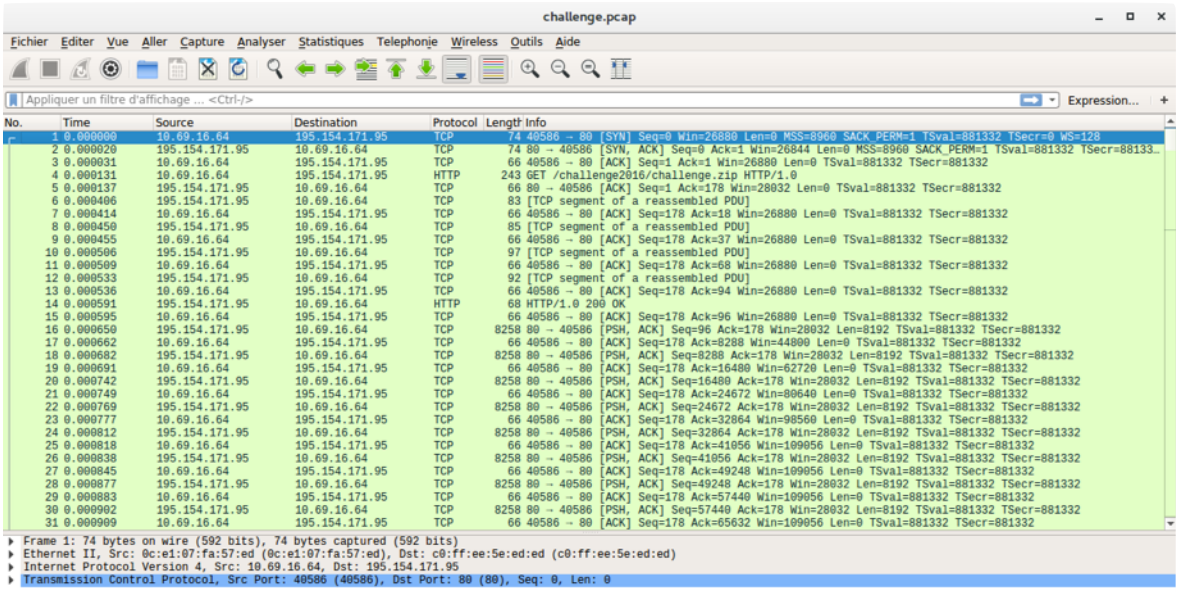
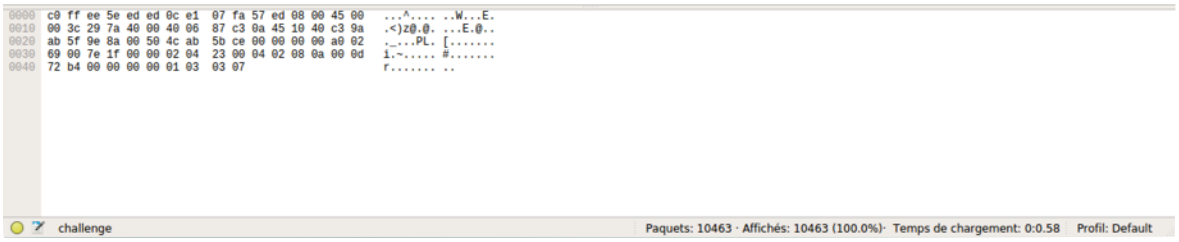
Solution du challenge SSTIC 2016

Vincent Bénony
Cryptic Apps

Le challenge de cette année se présente sous la forme d'un fichier PCAP, qui contient la trace du téléchargement d'un fichier depuis static.sstic.org. Il suffit d'utiliser Wireshark pour extraire l'archive « *challenge.zip* », et passer au niveau 1.





Niveau 1

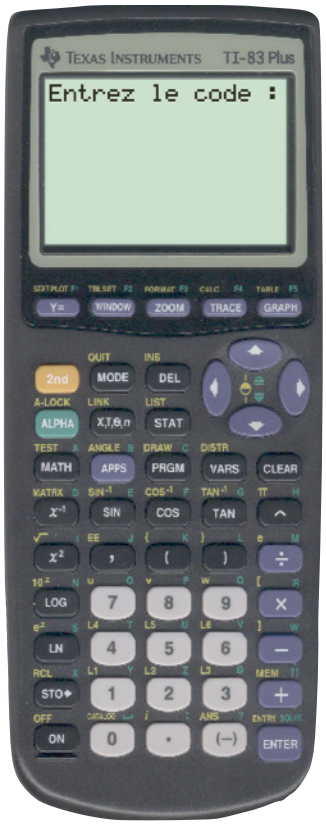
L'archive contient un jeu type RPG, écrit à l'aide de RPG JS (http://rpgjs.com).
Il s'agit de retrouver des clés en résolvant les différents challenges proposés par les personnages du jeu.

Pour ce premier niveau, trois épreuves sont proposées :
- calc
- radio
- SOS-Fant0me

Et en bonus, une grille de mots croisés.

Je me suis intéressé aux challenges « calc » et « SOS-Fant0me ».

calc

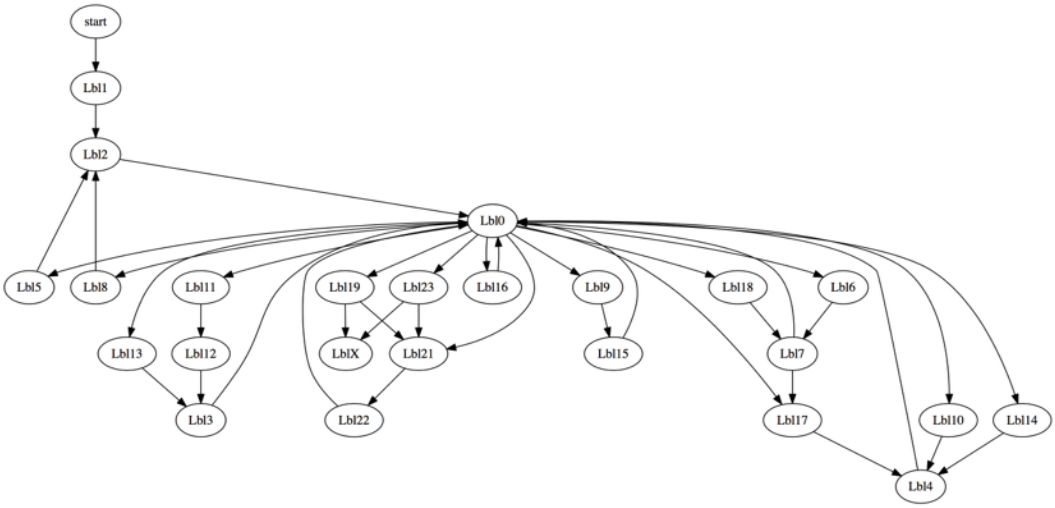


L'archive contient un unique fichier « SSTIC 16.8xp »

Il s'agit d'un programme pour calculatrice TI-83, qu'il est possible d'émuler avec des outils comme Virtual TI (http://www.ticalc.org/archives/files/fileinfo/84/8442.html).

Le programme TI Basic est stocké dans le fichier 8xp sous la forme de numéro de token; c'est une représentation compacte qui évite à la calculatrice de consommer trop de mémoire pour stocker le programme, et de passer trop de temps à parser le programme pendant l'exécution. Pour pouvoir l'analyser, il va tout d'abord falloir le transformer en un fichier texte lisible, grâce, par exemple, à un service comme Cemetech (https://www.cemetech.net/sc/). Le listing est disponible à la page suivante.

À première vue, le programme semble avoir été légèrement obfusqué (*control flow flattening*), mais on aperçoit très rapidement la présence d'un tableau, dont les valeurs font penser aux tables précalculées utilisées pour le calcul des CRC32.



Tous les calculs sur les nombres binaires passent par des manipulations de chaînes de caractères contenant des « 0 » et des « 1 », ce qui complique un peu la lecture. Malgré tout, on retrouve la plupart des primitives assez rapidement.

Le programme demande à l'utilisateur de saisir un code (un entier). Il en calcule le CRC32, et le compare à une valeur (3298472535). Si le CRC32 est correct, il calcule la clé du challenge en utilisant le générateur pseudo aléatoire du système, dont la graine a été initialisée en fonction du code saisi par l'utilisateur. Impossible de trouver la clé, sans trouver le nombre dont le CRC32 est connu…

Calculer un CRC32 est habituellement très rapide, quand il n'est pas fait de cette manière. De plus, l'espace des entiers 32 bits est très petit : il est donc possible de rechercher le nombre par force brute.

```
:ClrHome
:Goto 1
:
:Lbl 0
:If S=5:Goto 5
:If S=6:Goto 6
:If S=8:Goto 8
:If S=9:Goto 9
:If S=10:Goto 10
:If S=11:Goto 11
:If S=13:Goto 13
:If S=14:Goto 14
:If S=16:Goto 16
:If S=17:Goto 17
:If S=18:Goto 18
:If S=19:Goto 19
:If S=21:Goto 21
:If S=23:Goto 23
:Disp "DISPATCH ERROR"
:Stop
:
:Lbl 3
:sub(Str1,length(Str1)-((A+1)*8)+1,8)->Str1
:Goto 0
:
:Lbl 2
:" "->Str1
:Repeat not(A
: A/2->A
: sub("01",1+not(not(fPart(Ans))),1)+Str1-
>Str1
: iPart(A->A
:End
:sub(Str1,1,length(Str1)-1)->Str1
:While length(Str1)<32
: "0"+Str1->Str1
:End
:Goto 0
:
:
:Lbl 4
:" "->Str3
:For(I,1,length(Str1)
: If "1"=sub(Str1,length(Str1)-I+1,1) xor
"1"=sub(Str2,length(Str1)-I+1,1):Then
: "1"+Str3->Str3
: Else
: "0"+Str3->Str3
: End
:End
:sub(Str3,1,length(Str3)-1)->Str3
:Goto 0
:
:
:Lbl 7
:0->A
:For(I,1,length(Str1)
: If "1"=sub(Str1,length(Str1)-I+1,1):Then
: A+2^(I-1->A
: End
:End
:Goto 0
:
:
:Lbl 15
:"00000000"+sub(Str1,1,24)->Str1
:Goto 0
:
:Lbl 22
:" "->Str1
:0->C
:While A
: iPart(A/16)->C
: fPart(A/16)*16->B
: sub("0123456789ABCDEF",B+1,1)+Str1->Str1
: C->A
:End
:sub(Str1,1,length(Str1)-1)->Str1
:If length(Str1)<2
:"0"+Str1->Str1
:Goto 0
:
:
:Lbl 1
:Input "Entrez le code : ",Z
:4294967295->C
:0->N
:
{0,1996959894,3993919788,2567524794,1246341
37,1886057615,3915621685,2657392035,2492682
74,2044508324,3772115230,2547177864,1629419
95,2125561021,3887607047,2428444049,4985365
48,1789927666,4089016648,2227061214,4505488
61,1843258603,4107580753,2211677639,3258839
90,1684777152,4251122042,2321926636,3356334
87,1661365465,4195302755,2366115317,9970730
96,1281953886,3579855332,2724688242,1006888
145,1258607687,3524101629,2768942443,901097
722,1119000684,3686517206,2898065728,853044
451,1172266101,3705015759,2882616665,651767
980,1373503546,3369554304,3218104598,565507
253,1454621731,3485111705,3099436303,671266
974,1594198024,3322730930,2970347812,795835
527,1483230225,3244367275,3060149565,199414
6192,31158534,2563907772,4023717930,1907459
465,112637215,2680153253,3904427059,2013776
290,251722036,2517215374,3775830040,2137656
763,141376813,2439277719,3865271297,1802195
444,476864866,2238001368,4066508878,1812370
925,453092731,2181625025,4111451223,1706088
902,314042704,2344532202,4240017532,1658658
271,366619977,2362670323,4224994405,1303535
960,984961486,2747007092,3569037538,1256170
817,1037604311,2765210733,3554079995,113101
4506,879679996,2909243462,3663771856,114112
4467,855842277,2852801631,3708648649,134253
3948,654459306,3188396048,3373015174,146647
9909,544179635,3110523913,3462522015,159167
1054,702138776,2966460450,3352799412,150491
8807,783551873,3082640443,3233442989,398829
2384,2596254646,62317068,1957810842,3939845
945,2647816111,81470997,1943803523,38149189
30,2489596804,225274430,2053790376,38261757
55,2466906013,167816743,2097651377,40275525
80,2265490386,503444072,1762050814,41504172
45,2154129355,426522225,1852507879,42753135
26,2312317920,282753626,1742555852,41897081
43,2394877945,397917763,1622183637,36043908
88,2714866558,953729732,1340076626,35187199
85,2797360999,1068828381,1219638859,3624741
850,2936675148,906185462,1090812512,3747672
003,2825379669,829329135,1181335161,3412177
804,3160834842,628085408,1382605366,3423369
109,3138078467,570562233,1426400815,3317316
542,2998733608,733239954,1555261956,3268935
591,3050360625,752459403,1541320221,2607071
920,3965973030,1969922972,40735498,26178372
25,3943577151,1913087877,83908371,251234163
4,3803740692,2075208622,213261112,246327260
3,3855990285,2094854071,198958881,226202901
2,4057260610,1759359992,534414190,217671854
1,4139329115,1873836001,414664567,228224893
4,4279200368,1711684554,285281116,240580172
7,4167216745,1634467795,376229701,268506789
6,3608007406,1308918612,956543938,280855510
5,3495958263,1231636301,1047427035,29329598
18,3654703836,1088359270,936918000,28477148
99,3736837829,1202900863,817233897,31833421
08,3401237130,1404277552,615818150,31342074
93,3453421203,1423857449,601450431,30098376
14,3294710456,1567103746,711928724,30206684
71,3272380065,1510334235,755167117}->L1
:Z->A
:5->S
:Goto 2
:
:
:Lbl 5
:Str1->Str5
:C->A
:11->S
:Goto 2
:
:
:Lbl 11
:Str1->Str6
:Goto 12
:
:
:Lbl 12
:Str6->Str1
:0->A
:13->S
:Goto 3
:
:
:Lbl 13
:Str1->Str7
:Str5->Str1
:N->A
:14->S
:Goto 3
:
:
:Lbl 14
:Str1->Str1
:Str7->Str2
:6->S
:Goto 4
:
:
:Lbl 6
:Str3->Str1
:8->S
:Goto 7
:
:
:Lbl 8
:L1(A+1)->A
:9->S
:Goto 2
:
:
:Lbl 9
:Str1->Str8
:Str6->Str1
:10->S
:Goto 15
:
:
:Lbl 10
:Str8->Str2
:16->S
:Goto 4
:
:
:Lbl 16
:Str3->Str1
:11->S
:N+1->N
:If N=4
:Goto 17
:Goto 0
:
:
:Lbl 17
:"11111111111111111111111111111111"->Str2
:18->S
:Goto 4
:
:
:Lbl 18
:Str3->Str1
:19->S
:Goto 7
:
:
:Lbl 19
:ClrHome
:If A=3298472535:Then
: Z->rand
: 21->S
: 1->X
: 4->Y
: Output(3,7,"KEY:")
: Goto 21
: Else
: Output(3,6,"PERDU")
:End
:Goto X
:
:
:Lbl 21
:iPart(rand*256)->A
:23->S
:Goto 22
:
:
:Lbl 23
:If X=17:Then
: If Y=5:Then
: Goto X
: End
: 1->X
: 5->Y
:End
:Output(Y,X,Str1)
:X+2->X
:Goto 21
:
:Lbl X
:Stop
```

Programme TI-Basic du challenge calc

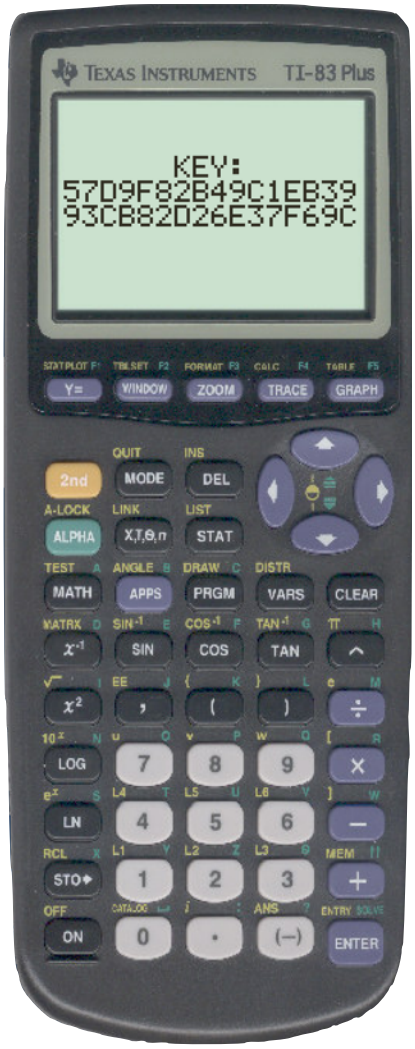
Afin d'accélérer les calculs, j'ai parallélisé la boucle à l'aide de GCD, mais je pense que ce n'était pas vraiment nécessaire au final… j'ai laissé le programme en l'état.

```
#include <iostream>
#include <cstdint>
#include <dispatch/dispatch.h>

static uint32_t poly8_lookup[256] =
{
    0x00000000, 0x77073096, 0xEE0E612C, 0x990951BA,
    0x076DC419, 0x706AF48F, 0xE963A535, 0x9E6495A3,
    0x0EDB8832, 0x79DCB8A4, 0xE0D5E91E, 0x97D2D988,
    0x09B64C2B, 0x7EB17CBD, 0xE7B82D07, 0x90BF1D91,
    0x1DB71064, 0x6AB020F2, 0xF3B97148, 0x84BE41DE,
    0x1ADAD47D, 0x6DDDE4EB, 0xF4D4B551, 0x83D385C7,
    0x136C9856, 0x646BA8C0, 0xFD62F97A, 0x8A65C9EC,
    0x14015C4F, 0x63066CD9, 0xFA0F3D63, 0x8D080DF5,
    0x3B6E20C8, 0x4C69105E, 0xD56041E4, 0xA2677172,
    0x3C03E4D1, 0x4B04D447, 0xD20D85FD, 0xA50AB56B,
    0x35B5A8FA, 0x42B2986C, 0xDBBBC9D6, 0xACBCF940,
    0x32D86CE3, 0x45DF5C75, 0xDCD60DCF, 0xABD13D59,
    0x26D930AC, 0x51DE003A, 0xC8D75180, 0xBFD06116,
    0x21B4F4B5, 0x56B3C423, 0xCFBA9599, 0xB8BDA50F,
    0x2802B89E, 0x5F058808, 0xC60CD9B2, 0xB10BE924,
    0x2F6F7C87, 0x58684C11, 0xC1611DAB, 0xB6662D3D,
    0x76DC4190, 0x01DB7106, 0x98D220BC, 0xEFD5102A,
    0x71B18589, 0x06B6B51F, 0x9FBFE4A5, 0xE8B8D433,
    0x7807C9A2, 0x0F00F934, 0x9609A88E, 0xE10E9818,
    0x7F6A0DBB, 0x086D3D2D, 0x91646C97, 0xE6635C01,
    0x6B6B51F4, 0x1C6C6162, 0x856530D8, 0xF262004E,
    0x6C0695ED, 0x1B01A57B, 0x8208F4C1, 0xF50FC457,
    0x65B0D9C6, 0x12B7E950, 0x8BBEB8EA, 0xFCB9887C,
    0x62DD1DDF, 0x15DA2D49, 0x8CD37CF3, 0xFBD44C65,
    0x4DB26158, 0x3AB551CE, 0xA3BC0074, 0xD4BB30E2,
    0x4ADFA541, 0x3DD895D7, 0xA4D1C46D, 0xD3D6F4FB,
    0x4369E96A, 0x346ED9FC, 0xAD678846, 0xDA60B8D0,
    0x44042D73, 0x33031DE5, 0xAA0A4C5F, 0xDD0D7CC9,
    0x5005713C, 0x270241AA, 0xBE0B1010, 0xC90C2086,
    0x5768B525, 0x206F85B3, 0xB966D409, 0xCE61E49F,
    0x5EDEF90E, 0x29D9C998, 0xB0D09822, 0xC7D7A8B4,
    0x59B33D17, 0x2EB40D81, 0xB7BD5C3B, 0xC0BA6CAD,
    0xEDB88320, 0x9ABFB3B6, 0x03B6E20C, 0x74B1D29A,
    0xEAD54739, 0x9DD277AF, 0x04DB2615, 0x73DC1683,
    0xE3630B12, 0x94643B84, 0x0D6D6A3E, 0x7A6A5AA8,
    0xE40ECF0B, 0x9309FF9D, 0x0A00AE27, 0x7D079EB1,
    0xF00F9344, 0x8708A3D2, 0x1E01F268, 0x6906C2FE,
    0xF762575D, 0x806567CB, 0x196C3671, 0x6E6B06E7,
    0xFED41B76, 0x89D32BE0, 0x10DA7A5A, 0x67DD4ACC,
    0xF9B9DF6F, 0x8EBEEFF9, 0x17B7BE43, 0x60B08ED5,
    0xD6D6A3E8, 0xA1D1937E, 0x38D8C2C4, 0x4FDFF252,
    0xD1BB67F1, 0xA6BC5767, 0x3FB506DD, 0x48B2364B,
    0xD80D2BDA, 0xAF0A1B4C, 0x36034AF6, 0x41047A60,
    0xDF60EFC3, 0xA867DF55, 0x316E8EEF, 0x4669BE79,
    0xCB61B38C, 0xBC66831A, 0x256FD2A0, 0x5268E236,
    0xCC0C7795, 0xBB0B4703, 0x220216B9, 0x5505262F,
    0xC5BA3BBE, 0xB2BD0B28, 0x2BB45A92, 0x5CB36A04,
    0xC2D7FFA7, 0xB5D0CF31, 0x2CD99E8B, 0x5BDEAE1D,
    0x9B64C2B0, 0xEC63F226, 0x756AA39C, 0x026D930A,
    0x9C0906A9, 0xEB0E363F, 0x72076785, 0x05005713,
    0x95BF4A82, 0xE2B87A14, 0x7BB12BAE, 0x0CB61B38,
    0x92D28E9B, 0xE5D5BE0D, 0x7CDCEFB7, 0x0BDBDF21,
    0x86D3D2D4, 0xF1D4E242, 0x68DDB3F8, 0x1FDA836E,
    0x81BE16CD, 0xF6B9265B, 0x6FB077E1, 0x18B74777,
    0x88085AE6, 0xFF0F6A70, 0x66063BCA, 0x11010B5C,
    0x8F659EFF, 0xF862AE69, 0x616BFFD3, 0x166CCF45,
    0xA00AE278, 0xD70DD2EE, 0x4E048354, 0x3903B3C2,
    0xA7672661, 0xD06016F7, 0x4969474D, 0x3E6E77DB,
    0xAED16A4A, 0xD9D65ADC, 0x40DF0B66, 0x37D83BF0,
    0xA9BCAE53, 0xDEBB9EC5, 0x47B2CF7F, 0x30B5FFE9,
    0xBDBDF21C, 0xCABAC28A, 0x53B39330, 0x24B4A3A6,
    0xBAD03605, 0xCDD70693, 0x54DE5729, 0x23D967BF,
    0xB3667A2E, 0xC4614AB8, 0x5D681B02, 0x2A6F2B94,
    0xB40BBE37, 0xC30C8EA1, 0x5A05DF1B, 0x2D02EF8D
};

inline uint32_t crc32(uint32_t x) {
    uint32_t v = 0xFFFFFFFF;
    int length = 4;
    uint8_t *bytes = (uint8_t *)&x;
    while (length-- != 0) {
        v = poly8_lookup[((uint8_t) v ^ *(bytes++))]
^ (v >> 8);
    }
    return v ^= 0xFFFFFFFF;
}

int main (int argc, char const *argv[]) {
    dispatch_queue_t queue =
dispatch_queue_create("parallel",
                      DISPATCH_QUEUE_CONCURRENT);
    int split = 64;
    dispatch_apply(split, queue, ^(size_t index) {
        uint64_t block_size = 0x100000000 / split;
        uint64_t from = index * block_size;
        uint64_t to = from + block_size;
        printf("worker %d: from %#08x to %#08x\n",
index, from, to);
        for (uint64_t x=from; x<to; x++) {
            uint32_t c = crc32(x);

            if ((x & 0x3FFFF) == 0) {
                printf("... %#08x\n", x);
                fflush(stdout);
            }

            if (c == 3298472535) {
                printf("Trouvé: %lld\n", x);
                exit(0);
            }
        }
    });
    return 0;
}
```

Recherche d'un entier dont le CRC32 est connu

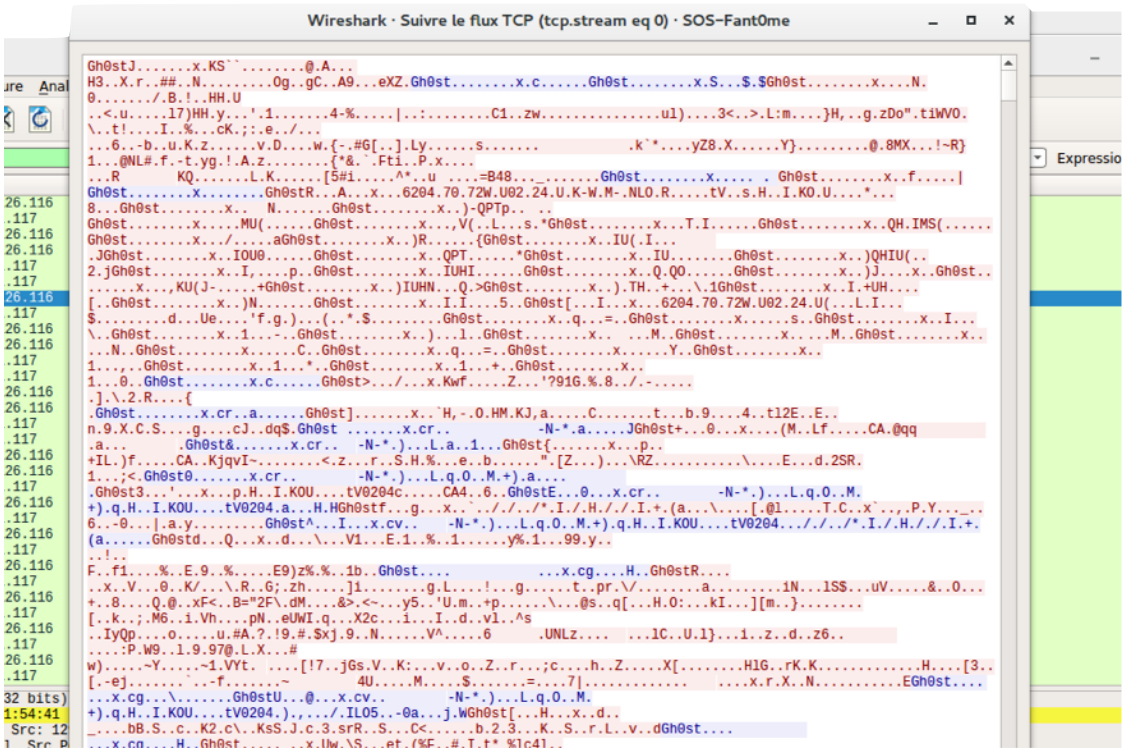
Après quelques secondes, nous obtenons la valeur : 89594902.

Une fois saisie dans l'émulateur, le programme nous retourne la valeur de la clé :



SOS-Fant0me

Le challenge se présente à nouveau sous la forme d'un fichier PCAP. En l'analysant avec Wireshark, on aperçoit que le motif « *Gh0st* » apparaît très fréquemment.



Il s'agit de la signature d'un malware connu, Gh0st RAT (https://en.wikipedia.org/wiki/Gh0st_RAT), qui permet de prendre le contrôle d'une machine à distance en lui envoyant des commandes à exécuter.

Le protocole réseau utilisé est très simple. Chaque message se compose ainsi :
- un en-tête de 5 caractères : « Gh0st »,
- un entier 32 bits correspondant à la taille en octets du paquet compressé,
- un entier 32 bits correspondant à la taille du paquet une fois décompressé,
- des données compressées avec la bibliothèque *zlib*.

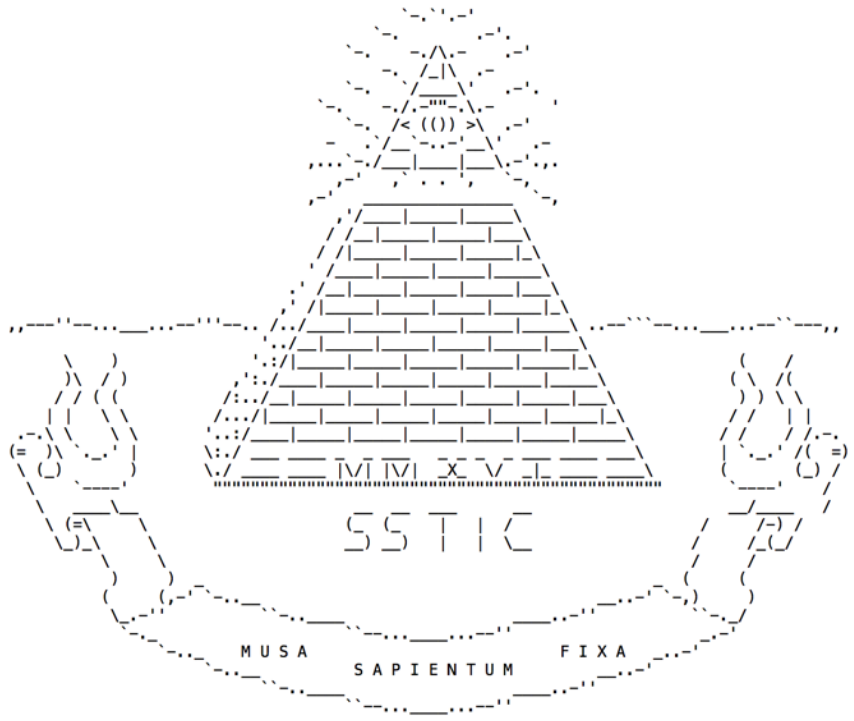
Une fois décompressé, le premier octet du paquet correspond à une commande, ou un évènement. Ceux qui nous intéressent le plus sont *COMMAND_FILE_DATA* (`0x05`) et *TOKEN_KEYBOARD_DATA* (`0x7C`). Ils permettent respectivement de récupérer un fichier depuis la machine distante, et d'intercepter des touches du clavier.

À l'aide d'un petit programme en C++, nous pouvons reconstruire les fichiers échangés, ainsi que le message saisi au clavier, et intercepté par le malware.

```
#include <cstdint>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <zlib.h>

using namespace std;

void exportFile(int from, int to, const char *packet_base_path, const char *filename) {
    void *data = 0;
    size_t total_data = 0;
    for (int i=from; i<=to; i+=2) {
        stringstream str;
        str << packet_base_path << "packet_" << i << ".bin";
        auto stream = ifstream(str.str(), ios_base::binary);
        stream.seekg(0, ios_base::end);
        size_t total = stream.tellg();
        total -= 9;
        stream.seekg(9, ios_base::beg);
        total_data += total;
        data = realloc(data, total_data);
        char *where = (char *)data + total_data - total;
        stream.read(where, total);
    }
    std::stringstream output_path;
    output_path << packet_base_path << filename;
    auto output = ofstream(output_path.str().c_str(), ios_base::binary);
    output.write((char *)data, total_data);
    free(data);
}

int main(int argc, const char * argv[]) {
    const char *packet_base_pth = "SSTIC2016/stage-1/data/Gh0st/packets/";
    auto stream = ifstream("SSTIC2016/stage-1/data/Gh0st/Gh0st.raw", ios::binary);

    stream.seekg(0, ios_base::end);
    auto total_len = stream.tellg();
    stream.seekg(0, ios_base::beg);

    char *buffer = (char *) malloc(total_len);
    stream.read(buffer, total_len);
    char *ptr = buffer;
    char *end = buffer + total_len;
    const char *header_sig = "Gh0st";

    int packet_cnt = 0;
    string msg;

    while (ptr < end) {
        if (memcmp(header_sig, ptr, 5)) {
            cerr << "Bad header at 0x" << hex << (ptr - buffer) << "\n";
            exit(1);
        }

        ptr += 5;
        uint32_t packed_length = *(uint32_t *) ptr; ptr += 4;
        uint32_t unpacked_length = *(uint32_t *) ptr; ptr += 4;

        char *dec = (char *) malloc(unpacked_length);
        memset(dec, 0, unpacked_length);

        z_stream strm;
        memset(&strm, 0, sizeof(z_stream));
        strm.avail_in = packed_length;
        strm.next_in = (Bytef *) ptr;
        strm.avail_out = unpacked_length;
        strm.next_out = (Bytef *) dec;
        if (inflateInit(&strm) != Z_OK) {
            cerr << "Cannot inflate" << "\n";
            exit(2);
        }
        inflate(&strm, Z_SYNC_FLUSH);
        inflateEnd(&strm);

        stringstream str;
        str << packet_base_pth << "packet_" << (packet_cnt++) << ".bin";
        auto output = ofstream(str.str(), ios_base::binary);
        output.write(dec, unpacked_length);

        unsigned char cmd = dec[0];
        if (cmd == 0x7b || cmd == 0x7c) {
            for (int i=1; i<unpacked_length; i++) {
                msg += dec[i];
            }
        }
        else if (cmd == 0x05) {
            cout << "file in " << str.str() << endl;
        }

        ptr += packed_length - 13;
        free(dec);
    }

    cout << "Message:\n" << msg << endl;

    exportFile(62, 62, packet_base_pth, "file.txt");
    exportFile(67, 113, packet_base_pth, "mov.mp4");
    exportFile(120, 120, packet_base_pth, "archive.zip");

    return 0;
}
```

On récupère trois fichiers :

- un fichier texte contenant ceci



- une vidéo célèbre

- une archive ZIP chiffrée,
- un message saisi au clavier :

```
[2016/02/27 - 23:14] New message: [SSTIC 2016/Challenge] Stage 1Salut !
Comme pis voici la clef pour le stage 1 ! Le mot de passe de l'archive
reste ceui convenu ensemble.[2016/02/27 - 23:15] sstic2016-stage1-
solution.zip - Saisir mot de passeCyb3rSSTIC_2016
```

Il semble y avoir quelques lettres manquantes, et j'avoue que je n'ai pas compris la raison… ce n'est pas faute d'avoir cherché. Mais après tout, le plus important est là, le mot de passe pour déchiffrer l'archive ZIP : « *Cyb3rSSTIC_2016* »

À l'intérieur, on trouve un fichier contenant la clé du challenge :

36 8B E8 C1 CC 7C C7 0C 22 45 03 09 34 30 1C 15.

Nous pouvons passer au niveau 2.

Niveau 2

Ce niveau propose 3 nouveaux challenges :

- EFI
- huge
- loader

J'ai décidé de m'intéresser aux deux premiers.

EFI

L'archive ZIP contient un unique fichier « *foo.efi* ». La commande FILE indique qu'il s'agit d'une application EFI, et plus précisément, d'une application EFI écrite en byte code EBC.

L'application est assez petite, et la fonction qui nous intéresse le plus se trouve en `0x10000530`. Elle reçoit en paramètre la clé de 16 octets saisie par l'utilisateur.

Elle commence par déchiffrer un GUID qui se trouve à l'adresse `0x100013D8`, grâce à un masque présent à l'adresse `0x10001698`. À l'issue du processus, on obtient le GUID `D8117CFE-94A6-11D4-9A-3A-00-90-27-3F-C1-4D` qui correspond au service EFI `gEfiDecompressProtocolGuid`, qui comme son nom l'indique, est un simple service de décompression de données.

Il est utilisé pour décompresser le buffer qui se trouve à l'adresse `0x10001470`. L'algorithme de compression étant public, il est facile de décompresser le buffer, et nous obtenons la chaîne unicode :

```
secret data: cb41dcb1d89746705a7fe998f11acce7
```

À partir de l'adresse `0x10000730`, la fonction transforme la partie hexadécimale de la chaîne décompressée en tableau d'octets. La méthode vérifie alors pour chaque octet de la clé saisie par l'utilisateur la propriété suivante :

$$\forall i \in [0;16[,\ s_i \oplus f(k_i, i) = 0$$

avec s le secret décompressé, k la clé saisie par l'utilisateur, et f la fonction définie par :

```
uint8_t f(uint8_t x, uint8_t s) {
	uint8_t t = x;

	s %= 8;
	x >>= s;
	t <<= 8 - s;

	return ~(x | t) & 0xFF;
}
```

Il s'agit de la rotation à droite de x par s, suivit du complement à 1.

Il est donc possible de retrouver reconstruire une clé qui vérifie cette propriété octet par octet par force brute. Nous obtenons :

34 7D 8C 72 72 0D 6E C7 A5 01 58 3B E0 BC CC 0C

Huge

L'archive contient un unique fichier TAR, qui contient lui-même un fichier ELF. Malheureusement, il n'est pas possible de décompresser ce dernier à cause de sa taille gigantesque (environ 116 To).

Il s'agit en fait d'un fichier très « creux », où les très longues plages de zéros ont été retirées par l'algorithme utilisé par TAR.

En analysant le contenu du fichier dans un éditeur hexadécimal, on trouve à l'offset `0x600` la table qui renseigne sur les parties non nulles du fichier ELF. Il est constitué de 24 segments de 4Ko, et 1 segment de 8Ko.

J'ai décidé d'extraire les 25 segments dans un seul fichier binaire, simplement en retirant les `0x800` premiers octets du fichier TAR. J'ai ensuite ouvert ce fichier en mode « RAW » dans Hopper.

L'en-tête ELF nous renseigne sur la manière dont le fichier va être mappé en mémoire :

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr           FileSiz            MemSiz             Flags  Align
  LOAD           0x0000000000001000 0x00002b0000000000 0x00002b0000000000 0x00001ef000000000 0x00001ef000000000  R E    1000
  LOAD           0x00002affffe1000 0x000049f000000000 0x000049f000000000 0x0000161000000000 0x0000161000000000  R E    1000
  LOAD           0x000049effffe1000 0x0000000000020000 0x0000000000020000 0x00002affffe0000 0x00002affffe0000  R E    1000
```

Afin de pouvoir suivre le flot de contrôle du programme, il faut être capable de transformer un offset dans ce fichier brut en une adresse en mémoire. Il suffit de procéder aux deux mappings à la suite, et pour cela, ce petit script Python nous sera très utile :

```
def physMemAddrToSegmentIndex(a):
    if a >= 0x2b0000000000 and a < 0x49f000000000:
        return 0
    elif a >= 0x49f000000000 and a < 0x600000000000:
        return 1
    elif a >= 0x20000 and a < 0x2b0000000000:
        return 2
    else:
        raise ArithmeticError

def physMemAddrToFileOffset(a):
    # Offset dans le fichier ou se trouve l'adresse
    file_offset=0
    if a >= 0x2b0000000000 and a < 0x49f000000000:
        file_offset = a - 0x2b0000000000 + 0x1000
    elif a >= 0x49f000000000 and a < 0x600000000000:
        file_offset = a - 0x000049f000000000 + 0x2affffe1000
    elif a >= 0x20000 and a < 0x2b0000000000:
        file_offset = a - 0x20000 + 0x49effffe1000
    else:
        raise ArithmeticError
    return file_offset

sparse_areas = [0x0, 0x017affef1000, 0x017affef7000,
0x0a2845ab1000, 0x0a2845ab4000, 0x17021da91000,
0x17021da9d000, 0x18abdb4a1000, 0x1ee7e5411000,
0x1ee7e541c000, 0x1ee7e541d000, 0x2b2706801000,
0x2b270680f000, 0x32566a40f000, 0x3adb440a1000,
0x3adb440a5000, 0x3b5815631000, 0x50e4b0dc1000,
0x50e4b0dd0000, 0x538a38011000, 0x538a38018000,
0x5ae338cb1000, 0x5ae338cb8000, 0x746ee2461000,
0x746ee246b000, 0x74effffc0000]
```

```
def fileOffsetToPartIndex(offset):
    for index in xrange(len(sparse_areas)):
        start = sparse_areas[index]
        stop = start + 4096
        if offset >= start and offset < stop:
            return index
    raise ArithmeticError

def fileOffsetToAddress(offset):
    part = offset / 4096
    offset_in_part = offset % 4096
    elf_offset = sparse_areas[part] + offset_in_part
    if elf_offset >= 0x1000 and elf_offset < 0x1ef000001000:
        return elf_offset - 0x1000 + 0x2b0000000000
    elif elf_offset >= 0x2affffe1000 and elf_offset < 0x410ffffe1000:
        return elf_offset - 0x2affffe1000 + 0x49f000000000
    elif elf_offset >= 0x49effffe1000 and elf_offset < 0x74effffc1000:
        return elf_offset - 0x49effffe1000 + 0x20000
    else:
        print "part: %d" % part
        print "offset_in_part: %d" % offset_in_part
        print "elf_offset: %d" % elf_offset
        raise ArithmeticError


def partIndexToSparseOffset(index):
    return index * 4096

def addressToSparseOffset(a):
    elf_offset = physMemAddrToFileOffset(a)
    pi = fileOffsetToPartIndex(elf_offset)
    return partIndexToSparseOffset(pi) + (elf_offset % 4096)
```

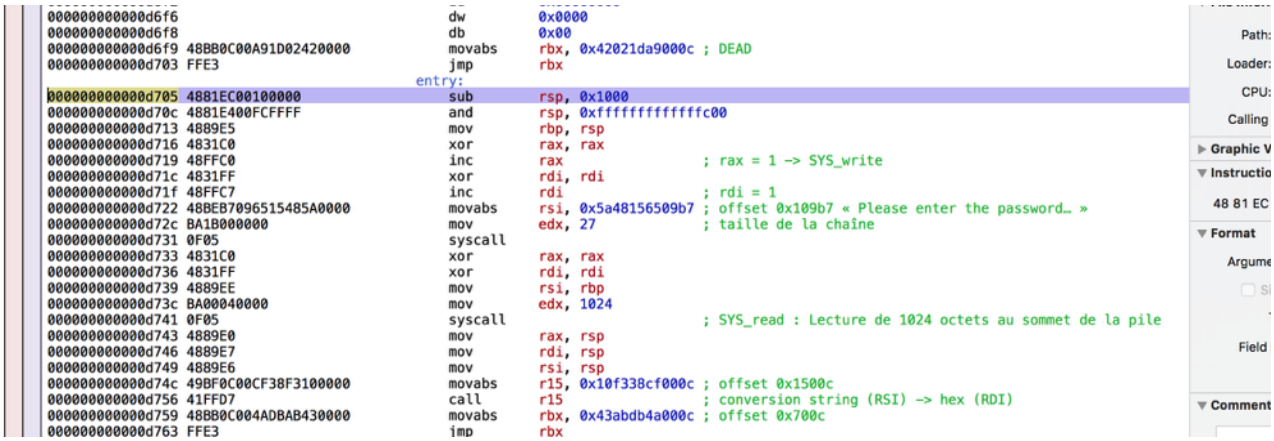
Ainsi, pour connaître l'offset dans Hopper de l'adresse A, il suffit de faire :

```
print hex(physMemAddrToFileOffset(A))
```

Et pour connaître l'adresse en mémoire d'un offset O du fichier brut :

```
print hex(fileOffsetToAddress(0))
```

Le point d'entrée du programme se trouve à l'adresse `0x51466a42e705`, ce qui correspond à l'offset `0xd705` dans Hopper.



Le programme affiche le message « *Please enter the password* » en utilisant un `syscall`, lit une chaîne au clavier, puis la transforme en un tableau d'octet à l'aide de la fonction à l'adresse `0x10f338cf000c`. Ensuite, il vérifie la structure de la clé morceau par morceau.

On trouve une grande quantité de « `00` » dans le programme, qui ont été vraisemblablement introduits pour brouiller les pistes, laissant penser que l'adresse ne pointe pas sur du code valide. Toutefois, « `00 00` » se désassemble en « `add byte [ds:rax], al` », et cette instruction n'a pas d'effet si le registre RAX pointe vers une adresse valide multiple de 256. Et c'est précisément ce qui est fait au tout début du programme.

La première propriété vérifiée par le programme est triviale : il vérifie que le premier octet de la clé est bien 0x29, ce qui nous donne une première information. Puis, il vérifie que le mot de 16 bits formé par les octets 2 et 3 est bien égal à `0xd17e`. Et ensuite, que l'octet 11 est bien 0x8C

Pour l'instant, nous savons que la clé est de la forme :

```
29 xx 7E D1 xx xx xx xx xx xx xx 8C xx xx xx xx
```

Viennent ensuite des propriétés plus complexes. La partie suivant se déroule à l'adresse `0x49e7e541000c` (offset `0x800c`). Cette fois-ci, les instructions « `add byte [ds:rax], al` » vont avoir un intérêt. Le programme commence par faire pointer RAX vers une adresse dont l'octet de poids faible est 0x03. Puis, il lit l'octet numéro 2 de la clé, et s'en sert pour exécuter un certain nombre d'instructions « `add byte [ds:rax], al` ». Ceci aura pour effet de calculer $3 \times (256 - x) \mod 256$, avec x l'octet lu de la clé. Il vérifie que le résultat est 0x65, ce qui n'est possible que pour x = 0x89. La clé est donc de la forme :

```
29 89 7E D1 xx xx xx xx xx xx xx 8C xx xx xx xx
```

Le test suivant (adresse `0x352845ab000c`) est plus trivial : il vérifie que les octets 12 à 15 forment le mot 32 bits `0xA9B00F5C` ⊕ `0x45ab3bd5`, soit `0xEC1B3489`. La clé est donc :

```
29 89 7E D1 xx xx xx xx xx xx xx 8C 89 34 1B EC
```

Puis que les octets 9 à 12 vérifient `0x59cb440c4556` ⊕ `0x59cbc8cc0b83` = `0x8cc04ed5`.

À ce stade, la clé est donc :

```
29 89 7E D1 xx xx xx xx D5 4E C0 8C 89 34 1B EC
```

Nous arrivons au dernier test, à l'adresse `0x2a7ee24a000c`. Celui-ci est un peu plus compliqué. Il lit les octets de 4 à 7 de la clé, et procède au XOR avec un mot de 80 bits :

```
3F FE 24 B8 78 38 1E 71 6D CB
```

$\oplus$

```
00 00 K4 K5 K6 K7 00 00 00 00
```

Le résultat est lu comme un flottant IEEE754, que j'appelle x par la suite. Le programme vérifie :

$$x = cos(x)$$

La solution de cette équation est le nombre de Dottie, $x \approx 0{,}739085$. Mais pour en déterminer l'encodage IEEE754 utilisé, j'ai à nouveau essayé tous les mots de 32 bits. L'encodage est :

```
3F FE BD 34 AE EC 1E 71 6D CB
```

ce qui implique que la partie lue depuis la clé entre les octets 4 à 7 est `0x998cd6d4`.

La clé finale est :

```
29 89 7E D1 D4 D6 8C 99 D5 4E C0 8C 89 34 1B EC
```

Une toute dernière étape masque cette clé avec la valeur CC FD CB C5 B2 A9 E6 2B 0D 7E 87 37 0E 2E 4F 19. Nous obtenons la clé du challenge :

E5 74 B5 14 66 7F 6A B2 D8 30 47 BB 87 1A 54 F5

Niveau 3

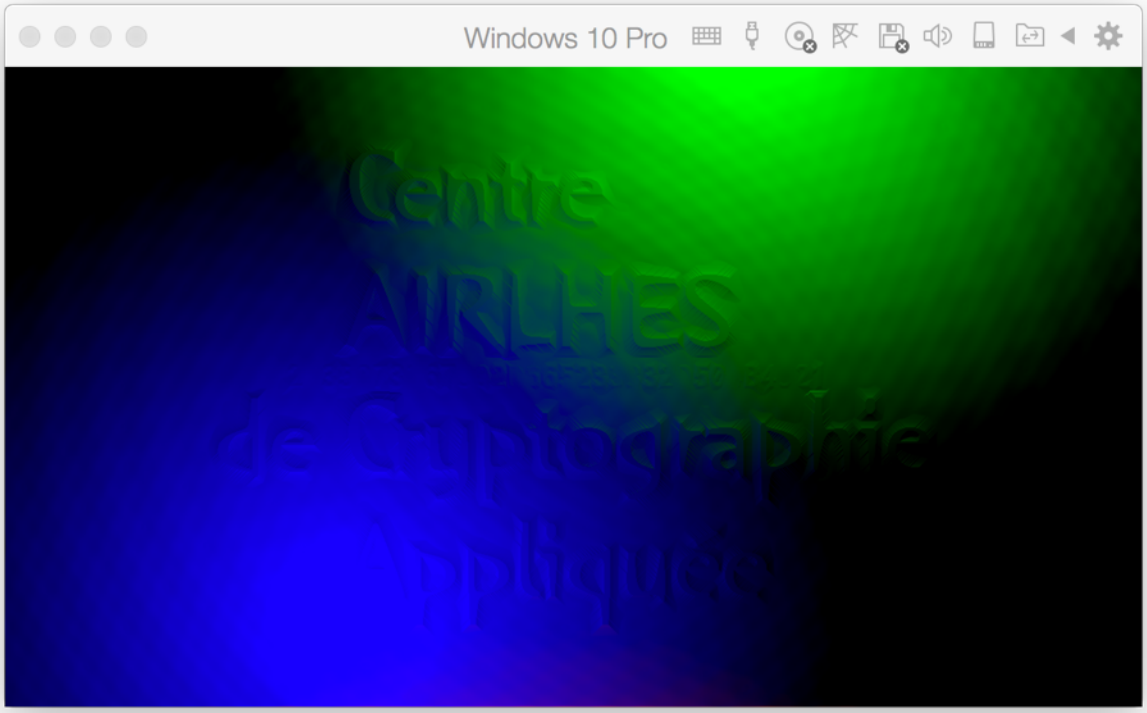
Pour ce niveau, nous avons 4 challenges :

- Ring
- Strange
- USB
- Video

Je me suis occupé des challenges Video, et Strange.

Video

L'archive de ce challenge contient 3 fichiers : un exécutable d'installation pour un économiseur d'écran, un fichier texte qui explique le but du challenge, et une vidéo.



Dans le fichier texte, nous apprenons que le but du challenge est de découvrir comment l'économiseur d'écran est capable d'exfiltrer des clés secrètes, et d'utiliser cette information pour retrouver la clé depuis la vidéo.

En observant la vidéo, on remarque des flashs de couleurs à partir de la 10e seconde. Il y a fort à parier qu'il s'agit de la méthode utilisée pour exfiltrer les clés, et qu'il va falloir déterminer le protocole exact.

J'ai décidé d'utiliser l'outil « *Inno Setup Unpacker* » (http://innounp.sourceforge.net) pour examiner le contenu de l'installeur. Après extraction, on trouve l'exécutable de l'économiseur d'écran, et un script pour l'installation.

La première chose à remarquer dans ce script d'installation, est l'inscription dans la base de registre d'une clé « *trajectories* », dont la valeur est un tableau d'octets correspondant à la chaîne de caractères « *Hello friend :)* » compressée à l'aide de *zlib*.

En recherchant la chaîne correspondant à la clé de registre dans l'exécutable, on trouve rapidement l'endroit où la valeur est lue (méthode débutant à l'adresse `0x4032D0`). Cette méthode énumère toutes les sous-clés de la base de registre, et concatène leurs valeurs, en les préfixant par un mot de 32 bits correspondant à la longueur des données. Cette méthode est utilisée à un seul endroit : dans la méthode débutant à l'adresse `0x403870`.

Il s'agit de la méthode principale, qui s'occupe d'initialiser l'économiseur d'écran. Il commence par lire les clés de la base de registre, initialiser DirectDraw, créer les images des cercles, *etc*.

La première chose intéressante à remarquer est la présence d'un tableau de mots de 32 bits à l'adresse `0x405020` dont les valeurs sont systématiquement utilisées après avoir été déchiffrées à l'aide d'un XOR et de la valeur `0x4FB78EB3`. En décodant le contenu, on remarque que les valeurs ressemblent aux différentes couleurs utilisées pour les flashs (`0xFF0000`, `0x00FF00`, `0xFFFF00`, … pour rouge, vert, jaune, bleu).

La méthode principale semble avoir un comportement qui dépend du calcul effectué par la méthode à l'adresse `0x403750`. Cette méthode détermine l'heure actuelle, en la déduisant de la date retournée par le couple *GetLocalTime* / *SystemTimeToFileTime*, modulo $8{,}64 \times 10^{13}$ (le nombre de nanosecondes dans une journée). Visiblement, en fin de journée, le comportement de l'économiseur d'écran change…

À ce moment là, la couleur utilisée pour dessiner les cercles change. Elle est calculée en fonction du tableau construit à la lecture de la base de registre, et suivant le protocole suivant :

- On choisit 8 couleurs, et on leur affecte une valeur :

rouge	0
vert	1
jaune	2
bleu	3
rose	4
cyan	5
blanc	6
noir	7

- on démarre le protocole en sélectionnant la couleur blanche,
- la suite à envoyer est convertie en un nombre en base 7, où chaque chiffre sera envoyé l'un après l'autre,
- pour chacun d'entre eux, on ajoute sa valeur à la couleur courante + 1, modulo 8 . Par exemple, si le chiffre est 5, on passe de la couleur blanche = 6, à la couleur 11 = 3 mod 8 = bleu. Le fait d'ajouter 1 permet de ne pas se retrouver deux fois de suite avec la même couleur, et donc, de détecter plus facilement le changement de chiffre.

En regardant la vidéo à partir de la dixième seconde, nous pouvons noter les couleurs des flashs successifs :

cyan	rose	cyan	rose	bleu	rose
rouge	bleu	noir	rouge	bleu	cyan
blanc	jaune	bleu	rouge	noir	rose
blanc	vert	jaune	noir	rouge	jaune
cyan	blanc	noir	cyan	noir	vert
blanc	jaune	bleu	jaune	bleu	rose
bleu	rose	noir	blanc	bleu	cyan

bleu	vert	bleu	blanc	bleu	blanc
rouge	blanc	vert	cyan	rouge	jaune
cyan	vert	bleu	rouge	vert	cyan
rouge	jaune	rose	blanc	vert	rose
cyan	jaune	blanc	noir	rouge	cyan
blanc	noir	cyan	noir	bleu	rouge
blanc	rose	noir	vert	rouge	rose

Pour reconstruire la donnée envoyée, nous allons calculer la distance entre deux couleurs successives, et en déduire la valeur du prochain chiffre en base 7. Il suffit ensuite de convertir le nombre en base 16.

La donnée reconstruite est :

```
18 00 00 00 78 DA 0B 16 34 17 2B DA C5 58 F9 CB 6E 62 57 F3 BE 7B 5B 00 32 50 07 7E
```

Elle commence par la valeur `0x00000018`, qui correspond à la taille de ce qui suit, en rouge. Le reste est, comme au début, un tableau d'octets compressé grâce à la *zlib*, et contient la clé du niveau :

53 11 37 16 72 BA 01 79 FA 3E 91 8A 83 BE DE B4

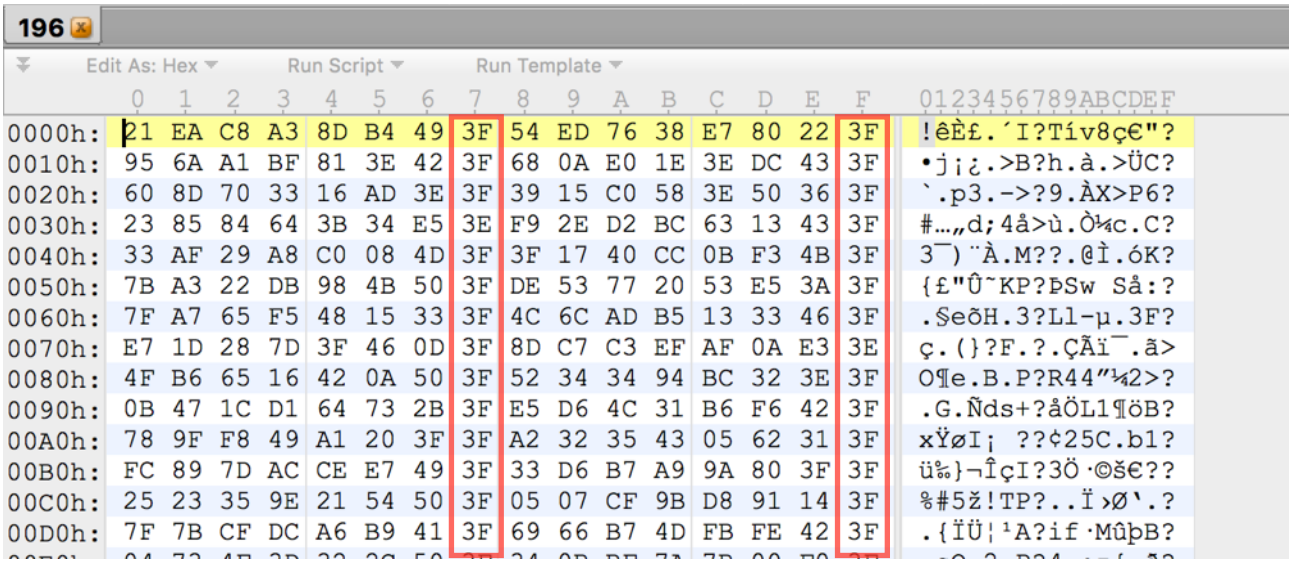
Strange

Il s'agit du challenge le plus délicat, et certainement celui qui m'a pris le plus de temps.

Il se présente sous la forme de deux fichiers, nommés « *196* » et « *a.out* ».

D'après la commande « file », le fichier « *a.out* » est un fichier exécutable ELF, pour processeur Itanium IA64. Par contre, aucune information quant à la nature du fichier « *196* ».

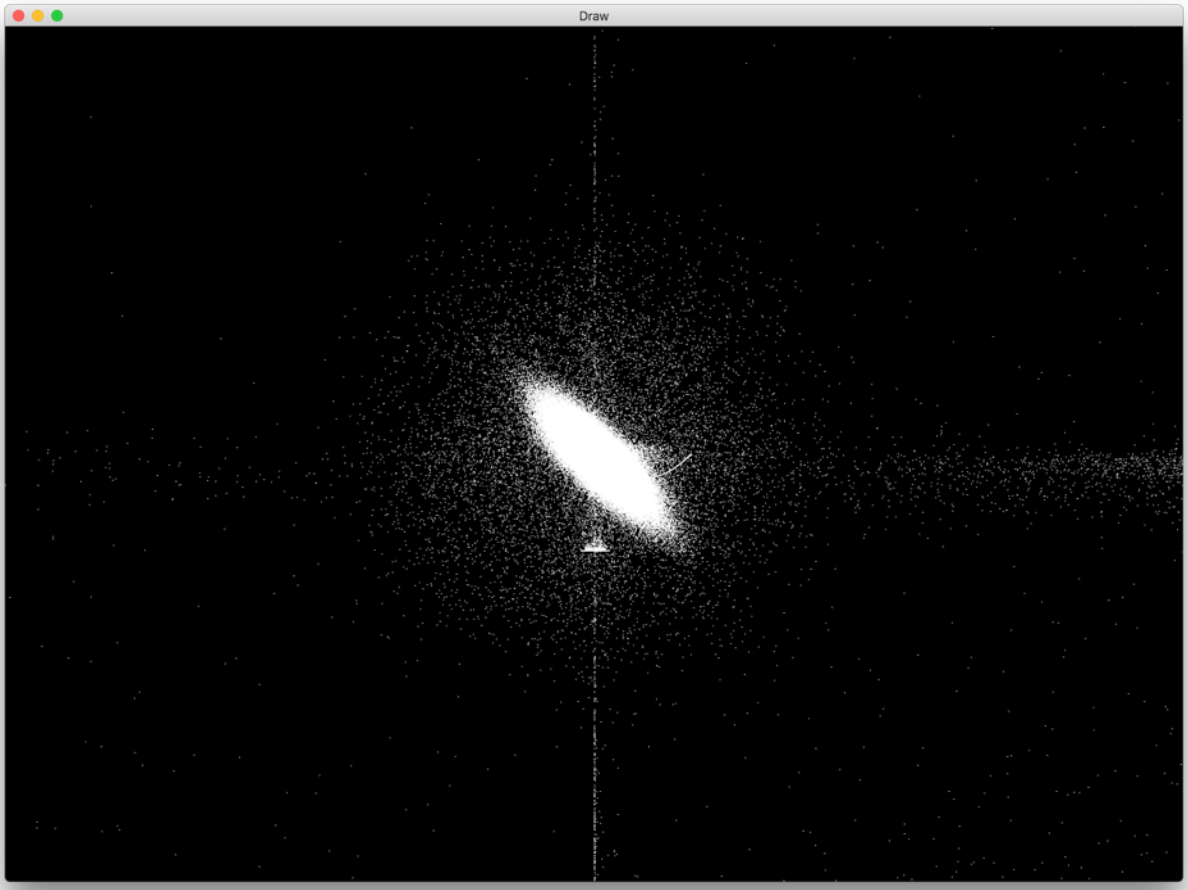
En y regardant de plus près, on peut remarquer un motif récurrent : la valeur `0x3F` qui apparaît tous les 8 octets.



Le fichier est en fait constitué de nombres flottants 64 bits, dont les valeurs sont souvent proches de 0. N'ayant pour l'instant aucune idée de l'utilité de ces nombres, j'ai tenté différentes approches pour en comprendre le sens, et j'ai entrepris de les représenter sous différentes formes.

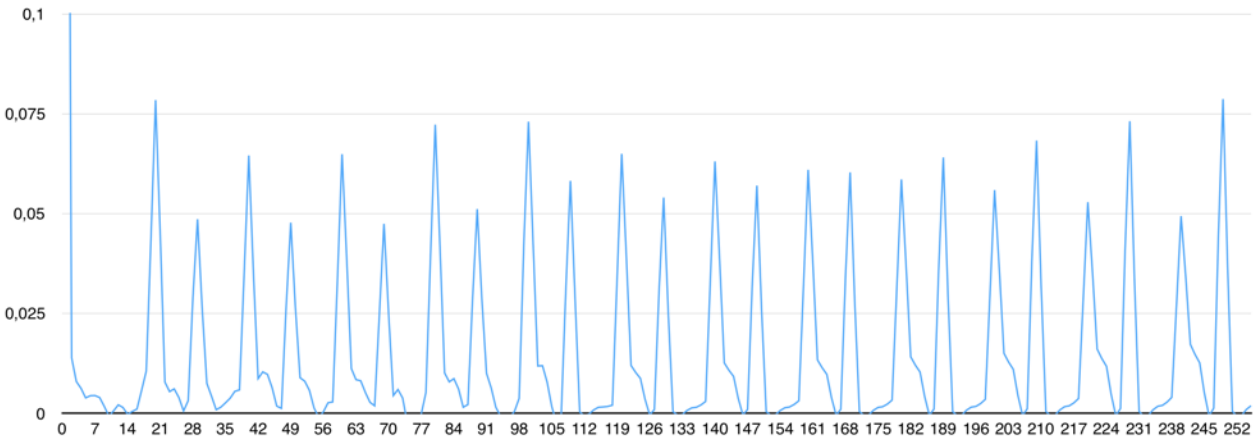
La première approche fut de considérer le fichier comme des couples de flottants, correspondant à des coordonnées 2D, et de tracer les points correspondants.

Malgré une jolie image, j'en ai retiré que très peu d'information… on aperçoit quelques singularités, mais pas assez pour réussir à déterminer la nature réelle de ces données.

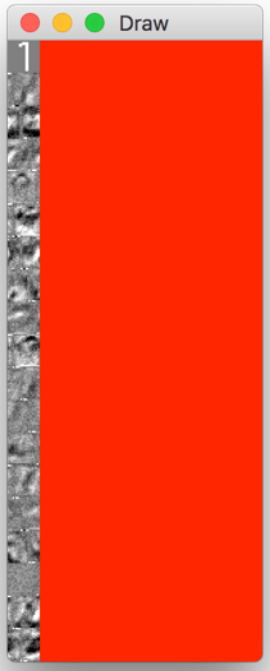


Fichier « 196 » représenté sous forme de couples (x, y)

En calculant l'autocorrélation du signal, nous voyons émerger quelque chose :



Il semblerait qu'il y ait un signal de période 20. J'ai donc décidé de visualiser le fichier, cette fois-ci, sous la forme de points plus ou moins clairs, en fonction de leur valeur absolue, et avec un retour à la ligne tous les 20 points. Bingo !



Représentation du fichier « 196 » sous forme de points dont la couleur dépend de la valeur

On voit clairement apparaître des motifs, et même le dessin d'un chiffre au tout début. En creusant un peu plus, on trouve l'image de tous les chiffres de 0 à 9 à intervalle régulier (tous les 526 880 octets).

Ne pouvant aller beaucoup plus loin, je suis passé à l'analyse du fichier « *a.out* ».

J'ai tout d'abord cru qu'il était lui aussi constitué en grande partie de données, qui auraient servi d'index dans le fichier « *196* » afin de reconstituer la clé. Mais non, il s'agissait de pur code IA64. Malgré tout, l'analyse m'a permis de mettre en évidence un motif dans le code : il y a un grand nombre de fonctions de 32 320 octets, assez similaires.

Au milieu de toutes ces fonctions, on en trouve une très différente à l'adresse `0x400000000019C700`. Mon analyse va se concentrer sur celle-ci. Le code IA64 n'est pas évident à lire quand on ne connaît pas, mais il finit par devenir assez clair après quelques heures…

La fonction commence par ouvrir le fichier « *196* », et en copie le contenu en mémoire. Elle ouvre ensuite un fichier texte (car ouvert avec un « *fopen* » en mode « *r* »), dont le nom est donné en paramètre du programme. Ce fichier est lu ligne par ligne.

La méthode vérifie qu'il commence par la chaîne « P2 ». Après une rapide recherche sur Internet, il apparaît que cet en-tête correspond à un fichier PGM, ce qui sera validé par le reste de l'analyse.

Après avoir vérifié que la ligne suivante est un commentaire (*i.e.*, commençant par un dièse), elle lit la largeur et la hauteur de l'image, et vérifie que le produit est égal à 12 800. Cette taille peut correspondre à de nombreux couples largeur / hauteur, mais si l'on se souvient que nous avons découvert des images de chiffres dans le fichier « *196* » dont la taille était de 20×20, et que 12 800 / (20×20) = 32, il semblerait assez logique que l'image soit de taille 640×20 ou 20×640, et qu'elle contienne des caractères identiques à ceux que nous avons découverts; d'autant plus que 32 correspond au nombre de chiffres qu'il y a dans une clé de 128 bits…

En analysant un peu plus la méthode, il apparaît qu'elle va tester certaines propriétés sur le fichier PGM lu, et qu'elle va afficher le mot « *fail* » ou « *pass* » en fonction de ce qu'elle a calculé.

Pour chaque bloc de 20×20 du PGM, la méthode va vérifier que cette partie de l'image n'est pas vide, et que le motif qu'elle contient occupe un certain espace, en déterminant quelles sont les premières et dernières lignes / colonne non vides.

Ensuite, la méthode appelle en cascade toute une série de méthodes, qui vont retourner 4 valeurs flottantes. En fonction de ces valeurs, l'algorithme passera au bloc suivant, ou s'arrêtera. Si les 32 blocs sont corrects, l'algorithme affichera « *pass* ».

Je dois avouer que la nature exacte de ces fonctions reste encore un peu floue; toutefois, je pense qu'il s'agit d'une sorte de réseau neuronal, dont les paramètres sont fournis par le fichier « *196* », juste après les dessins des chiffres, et qui a été entraîné pour reconnaître les caractères de la clé.

L'algorithme utiliserait donc 32 réseaux, pour vérifier les 32 caractères.

Une chose importante à remarquer est que l'algorithme affiche un point avant chaque test, et qu'il s'arrête immédiatement si le caractère n'est pas reconnu. Cette « faille » va être exploitée pour retrouver la clé.

J'ai donc écrit un programme en C qui, à partir d'une clé donnée en paramètre, construit un fichier PGM de 640×20 avec les images que j'ai trouvées dans le fichier « 196 »:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>
#include <string.h>

int main(int argc, char **argv) {
    FILE *f = fopen("196", "rb");
    char *key = argv[1];
    double pgm[12800];
    double d;
    for (int i=0; i<32; i++) {
        const char *lut = "1234567890";
        int v = strchr(lut, key[i]) - lut;
        fseek(f, 526880 * v, SEEK_SET);
        for (int y=0; y<20; y++) {
            for (int x=0; x<20; x++) {
                fread(&d, 8, 1, f);
                pgm[x + i * 20 + y * 640] = (fabs(d) > 0.5) ? 1 : 0;
            }
        }
    }

    printf("P2\n");
    printf("# %s\n", key);
    printf("640 20\n");
    printf("255\n");

    for (int y=0; y<20; y++) {
      for (int x=0; x<640; x++) {
        printf("  %d\n", pgm[x + y * 640] ? 0 : 255);
      }
    }
    return 0;
}
```

J'ai ensuite utilisé l'émulateur IA64 Ski (http://ski.sourceforge.net) pour simuler le programme sur ma machine.

Un simple script Bash m'a permis de tester tous les caractères possibles, jusqu'à trouver la clé complète :

```
#!/bin/bash
found=""
for n in $(seq 32); do
    for i in 0 1 2 3 4 5 6 7 8 9; do
        KEY="${found}${i}00000000000000000000000000000"
        ./gen_pgm $KEY >output.pgm

        CNT=$(bskinc -simroot $(pwd) a.out output.pgm | grep "\." | wc -l)

        if [ $CNT -ge $(( $n + 1 )) ]; then
            found="${found}${i}"
            break;
        fi
    done
    echo $found
done
```

Après quelques minutes, nous obtenons la clé complète :

23 42 50 38 47 25 08 28 73 35 77 20 85 54 40 35

Niveau 4

Ce sera le niveau le plus court. Nous obtenons dès le début un fichier « final.txt » qui contient :

> Coucou !
>
> Tu as presque réussi le challenge !
> l01p1 y'4qe3553 z41y : 8Y6d5j9Vy88HUGHfGSKsJvqA@ffgvp.bet

Une partie du fichier est tout simplement codé en ROT13, et le décodage nous donne l'adresse finale du challenge :

> V01c1 l'4dr3553 m41l : 8L6q5w9Il88UHTUsTFXfWidN@sstic.org