Challenge SSTIC 2017

Introduction

Cette solution pour le challenge SSTIC 2017 ne cherche pas à se trouver dans le haut du tableau du classement qualité. L'auteur n'étant ni joueur ni consciencieux, les *LUM* rencontrés au cours de la résolution du challenge ne sont pas mentionnés. La lecture de n'importe quelle autre solution est chaudement recommandée pour de plus amples détails. Enfin, la brièveté de ce document est inversement proportionnelle au temps passé sur chaque épreuve.

Electronic Flash

Le challenge consiste en un fichier intitulé ***Bittendo_email_leak_by_shadowbrokers.zip***. Afin de ne pas partir dans une fausse direction, il semble approprié de vérifier que le fichier n'est pas polyglotte :

```
$ evince Bittendo email leak by shadowbrokers.zip
Unable to open document "Bittendo email leak by shadowbrokers.zip".

$ tar xzvf Bittendo email leak by shadowbrokers.zip
gzip: stdin has more than one entry--rest ignored
tar: Child returned status 2
tar: Error is not recoverable: exiting now

$ eog Bittendo email leak by shadowbrokers.zip
Could not load image 'Bittendo_email_leak_by_shadowbrokers.zip'.
Unrecognized image file format

$ python Bittendo email leak by shadowbrokers.zip
/usr/bin/python: can't find '__main__' module in
'Bittendo_email_leak_by_shadowbrokers.zip'

$ strings Bittendo email leak by shadowbrokers.zip | grep drogue | head -2
https://www.youtube.com/watch?v=pmmRQd4Jd8I
https://www.youtube.com/watch?v=xeDKyl04G74
```

Vérifications faites, le fichier est un bien une archive zip contenant un e-mail avec 2 pièces jointes :

- ***NAND_pinout.jpg*** : photographie de la mémoire flash NAND,
- ***NAND_FLASH_writes_no_OOB_5MHz.sr*** : enregistrement de la programmation de la mémoire flash via un analyseur logique.

Un plugin Python pour le logiciel sigrok a été développé afin d'extraire les données écrites sur la mémoire flash. Aucun détail supplémentaire sur la résolution de cette épreuve ne sera donné, le disque dur de l'ordinateur portable contenant ce plugin Python ayant été *wipé*[1]. C'est regrettable, le script était, de mémoire, un exemple de qualité et de concision.

Deux fichiers ont été écrits sur la mémoire flash : ***challenge.zip*** et ***challenge.zip.md5***.

```
$ cat challenge.zip.md5
3977e2084331bb1c52abb115a332c6cc  challenges.zip
$ md5sum challenges.zip
f4e9e584023e6c3fb627697dae50e11d  challenges.zip
```

Quelques bits semblent être invalides mais l'archive s'extrait sans erreur, le challenge peut donc continuer.

L'archive ***challenge.zip*** contient des fichiers destinés à être servis par un serveur web afin d'évaluer les performances des moteurs JavaScript. Les captures d'écran suivantes illustrent d'ailleurs brillamment l'importance du choix d'un navigateur web moderne.

[1] Véridique.

Figure 1 : w3m

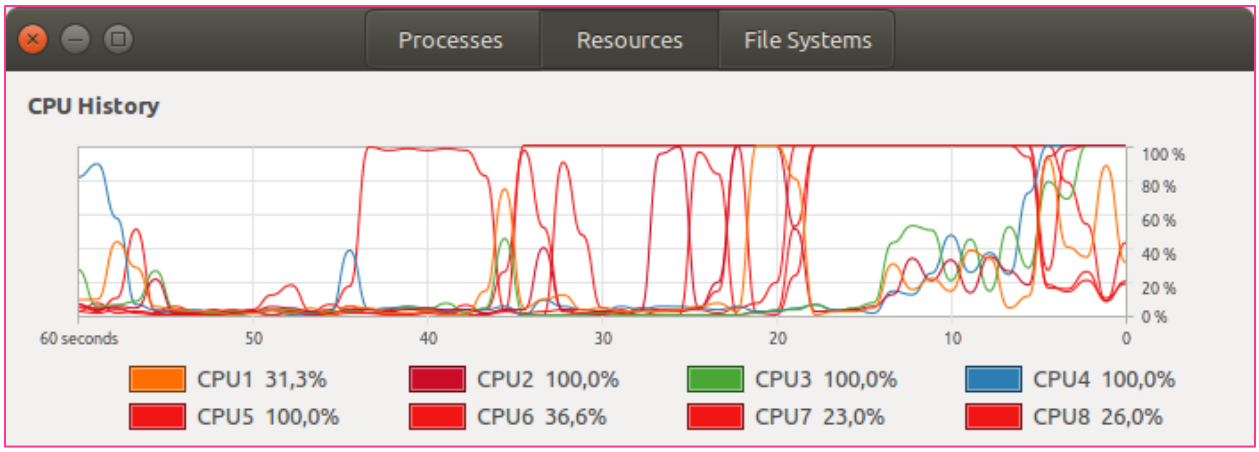

Figure 2 : Mozilla Firefox

L'interface montre par ailleurs que les goûts et les couleurs sous w3m se discutent, et que le challenge comporte 4 épreuves dont la première, Electronic Flash, vient d'être résolue remarquablement. La dernière épreuve reste quant à elle entourée de mystère.

Notons que l'agent de télémétrie SSTICsoft envoie des notifications au serveur de l'organisation lors de la validation d'une épreuve et lors de la découverte d'un LUM. Pour des raisons évidentes de vie privée, il est recommandé de désactiver cet agent (CVSS : 5.2, information disclosure). Le lecteur médisant notera à juste titre que sa désactivation évite de montrer que l'auteur a passé plus de 5 jours sur la résolution de l'épreuve *Unstable Machines*.

Don't Let Him Escape

Le script Python */challenges/dont_let_him_escape/server.py* créé une raw socket UDP à laquelle est attaché un filtre eBPF compilé. Le but de l'épreuve est de comprendre comment sont parsés les paquets réseau par ce filtre eBPF afin de générer un paquet contenant la clé attendue.

Plusieurs outils censés désassembler le programme BPF compilé ont été testés sans succès. Le résultat du noyau Linux lui-même est incorrect. Le code noyau JITé a donc été *dumpé* et *mappé* en espace utilisateur afin d'être débuggé de façon plus conventionnelle avec un débuggeur qui ne marche pas : GDB. Le code dumpé utilise certaines adresses absolues de l'espace noyau, notamment pour faire appel aux fonctions relatives à ***BPF_MAP_CREATE*** et ***BPF_MAP_LOOKUP_ELEM***. Quelques octets ont été modifiées pour faire appel à des fonctions équivalentes en espace utilisateur.

L'analyse et la compréhension de ce code permet de déterminer le paquet UDP attendu pour afficher un LUM ou la clé attendue. Notons que 2 mots de 4 octets sont obtenus par bruteforce.

La clé est ***KEY{ 2d4ceda2fa2a0e08fc360b55291de7c9}*** et sa validation débloque l'épreuve suivante :

```
/home/user # /challenges/tools/add_key 2d4ceda2fa2a0e08fc360b55291de7c9
  [OK] Valid KEY : 2d4ceda2fa2a0e08fc360b55291de7c9 (world 2)
```

Riscy Zones

Plusieurs fichiers sont présent dans le répertoire */challenges/riscy_zones/* :

- ***password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted***
- ***secret.lzma.encrypted***
- ***TA.elf.signed***
- ***trustzone_decrypt***

L'excellent jeu de mot sur le nom de l'épreuve et le des 2 binaires suggèrent de façon subtile la présence d'un *Trusted Execution Environment* (TEE) dont la communication depuis le Normal World est assuré par l'intermédiaire du device */dev/sstic*. Le programme ***trustzone_decrypt*** tire parti de ce device pour interagir avec la Trusted Applet ***TA.elf.signed***. Ainsi, le déchiffrement du fichier ***password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted*** est réalisé dans le Secure World, comme le montrent les commandes suivantes :

```
/challenges/riscy_zones $ ./trustzone_decrypt \
    00112233445566778899AABBCCDDEEFF \
    password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted \
    /tmp/clear.txt

/challenges/riscy_zones $ hd /tmp/clear.txt
00000000  44 65 63 72 79 70 74 65  64 20 62 6c 6f 63 6b 0a  |Decrypted block.|
*
00000100  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
```

Les informations de journalisation confirment ce comportement :

```
[i] load TA.elf.signed in TrustedOS
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD GET TA VERSION
[+] OS return code = 0x00000000, TA return code = 0x00000000
retreived version : SSTIC Trusted APP v0.0.1
[i] check password in TEE
[+] OS return code = 0x00000000, TA return code = 0x00000000
Good password !
[i] Send command to Trusted App CMD DECRYPT BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
...
```

Le but du challenge est de trouver la clé de déchiffrement du fichier ***secret.lzma.encrypted***.

L'avantage de commencer le challenge en cours de route est que la probabilité qu'un participant ait déjà réalisé une partie du travail fastidieux est loin d'être nulle. Merci à Guillaume Jeanne d'avoir implémenté rapidement le support des CPU RISC-V [2] et OpenRISC [3] dans IDA Pro. Quel regret que de ne pas avoir à compiler objdump pour supporter ces architectures exotiques.

[2] https://github.com/0xDeva/ida-cpu-RISC-V
[3] https://github.com/0xDeva/ida-cpu-OpenRisc

Le *reverse* de la Trusted App montre que plusieurs commandes sont acceptées (numéro de version, déchiffrement d'un bloc, etc.). Les fichiers chiffrés comportent un en-tête de 112 octets de la forme suivante, chiffré en AES-ECB avec la clé ***___SSTIC_2017___***:

```
==BEGIN PASSWORD HMAC==
Password HMAC en hexa
==END PASSWORD HMAC==
```

Il est possible (en théorie, cela n'a pas été vérifié), de déterminer si le mot de passe du fichier est valide en calculant son HMAC-SHA256 avec la clé ***7fdcb986058767ec47f417efbe85a10c*** hardcodée dans la Trusted App, et en comparant le condensat avec celui présent dans l'en-tête.

Les données suivant l'en-tête sont chiffrées avec un algorithme custom de type CBC, dont la longueur de clé ainsi que la taille de bloc sont chacune de 16 octets. Les données en clair sont *paddées* avec ***n*** octets dont la valeur ASCII est ***n***.

Le reverse de la Trusted App aboutit au développement du code suivant Python pour déterminer les 14 premiers octets de la clé :

```
def solve(n, output):
    key = ''
    for i in range(0, 16):
        c = output[i] - ((output[i + 1] + 170 - i * 7))
        key += chr(c & 0xff)

    a = struct.unpack('<I', key[0:4])[0]
    a = ((a << ((n + 13) & 31)) | (a >> ((32 - n - 13) & 31))) & 0xffffffff

    b = struct.unpack('<I', key[4:8])[0]
    b = ((b << ((n + 17) & 31)) | (b >> ((32 - n - 17) & 31))) & 0xffffffff

    c = struct.unpack('<I', key[8:12])[0]
    c = ((c << ((n + 19) & 31)) | (c >> ((32 - n - 19) & 31))) & 0xffffffff

    d = struct.unpack('<I', key[12:16])[0]
    d = ((d << ((n + 23) & 31)) | (d >> ((32 - n - 23) & 31))) & 0xffffffff

    part0 = ((c & ~0xadaaa9aa) & 0x52555655) | ((a & ~0x52555655) & 0xadaaa9aa)
    part1 = ((d & ~0xadaaa9aa) & 0x52555655) | ((b & ~0x52555655) & 0xadaaa9aa)
    part2 = ((a & ~0xadaaa9aa) & 0x52555655) | ((c & ~0x52555655) & 0xadaaa9aa)
    part3 = ((b & ~0xadaaa9aa) & 0x52555655) | ((d & ~0x52555655) & 0xadaaa9aa)

    result  = struct.pack('<I', part0)
    result += struct.pack('<I', part1)
    result += struct.pack('<I', part2)
    result += struct.pack('<I', part3)

    return result
```

Les 2 derniers octets de la clé ont été obtenus par bruteforce. Une technique plus élégante (bien que non testée) serait de déterminer ces 2 octets directement en utilisant les 16 valeurs possibles de padding.

La clé de déchiffrement (et de validation de l'épreuve) est ***5921cd9fd3a82bd9244ece5328c6c95f*** et le fichier déchiffré est une image JPEG compressée avec LZMA, contenant un LUM encodé avec 3 itérations de ROT13.

Unstable Machines

Le fichier de cette épreuve est un binaire Windows 64-bit, ***unstable.machines.exe***. Son exécution affiche une fenêtre contenant 7 *checkbox*. Quelques F5 sous IDA Pro montrent que le clic des checkbox dans un ordre précis (5, 4, 6, 2, 7, 4, 1, 3, 1, 7, 3) modifie la fenêtre et affiche une entrée de texte. Un *serial* de 32 octets est attendu, et si le résultat de sa modification par un algorithme restant à déterminer correspond à une valeur fixe, l'épreuve est résolue.

Metasm

Metasm peut être utilisé afin de trouver semi-automatiquement le serial de l'épreuve, comme illustré par la capture d'écran suivante. La solution trouvée étant semi-complète, une analyse manuelle du binaire reste néanmoins nécessaire. Afin d'être moderne, x64dbg a été choisi pour le débugger. Cette erreur ne sera pas reproduite dans le futur, ce débuggeur étant lent à mourir.

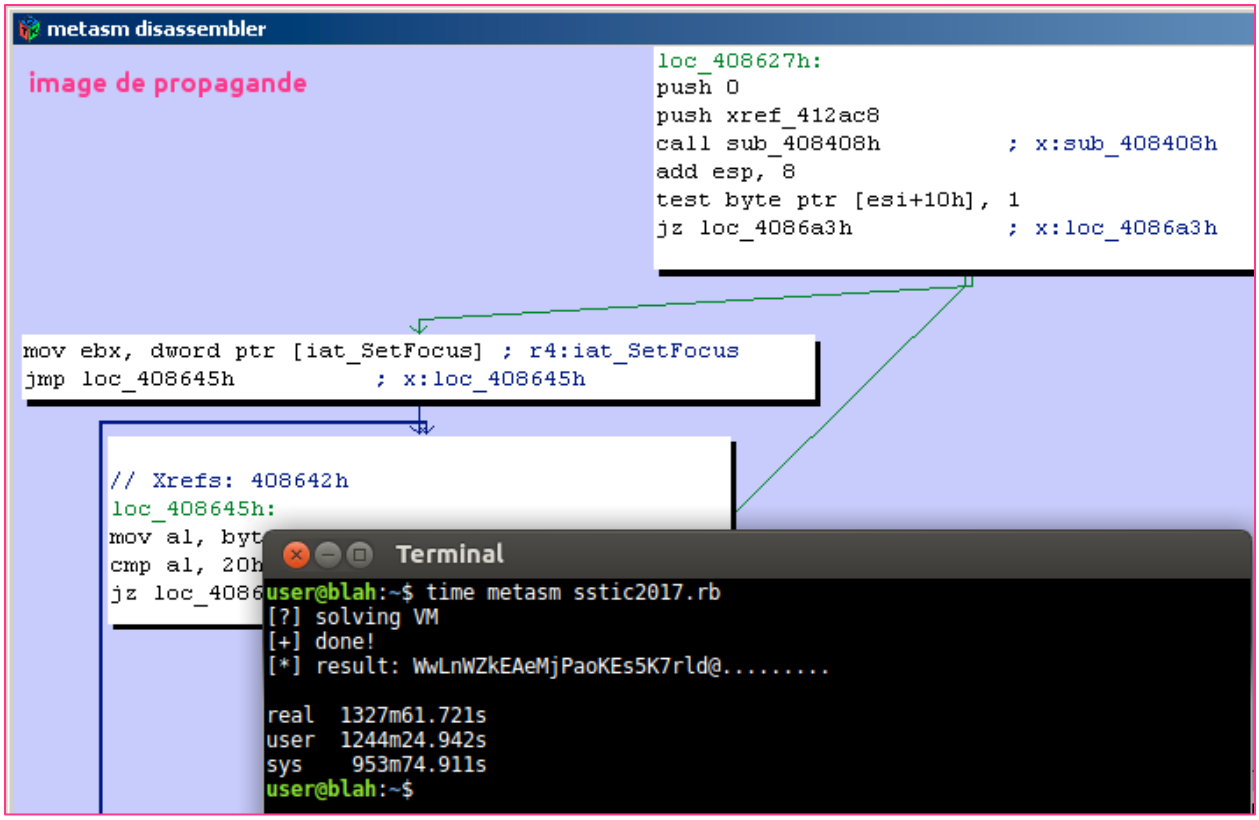

Figure 3 : Metasm en action

Trucs relous

Le binaire est truffé d'astuces (quelques exemples sont présentés par l'image suivante) destinées à contrarier le reverser. Notons qu'il n'existe pas de poudre de perlimpinpin pour en venir à bout magiquement.

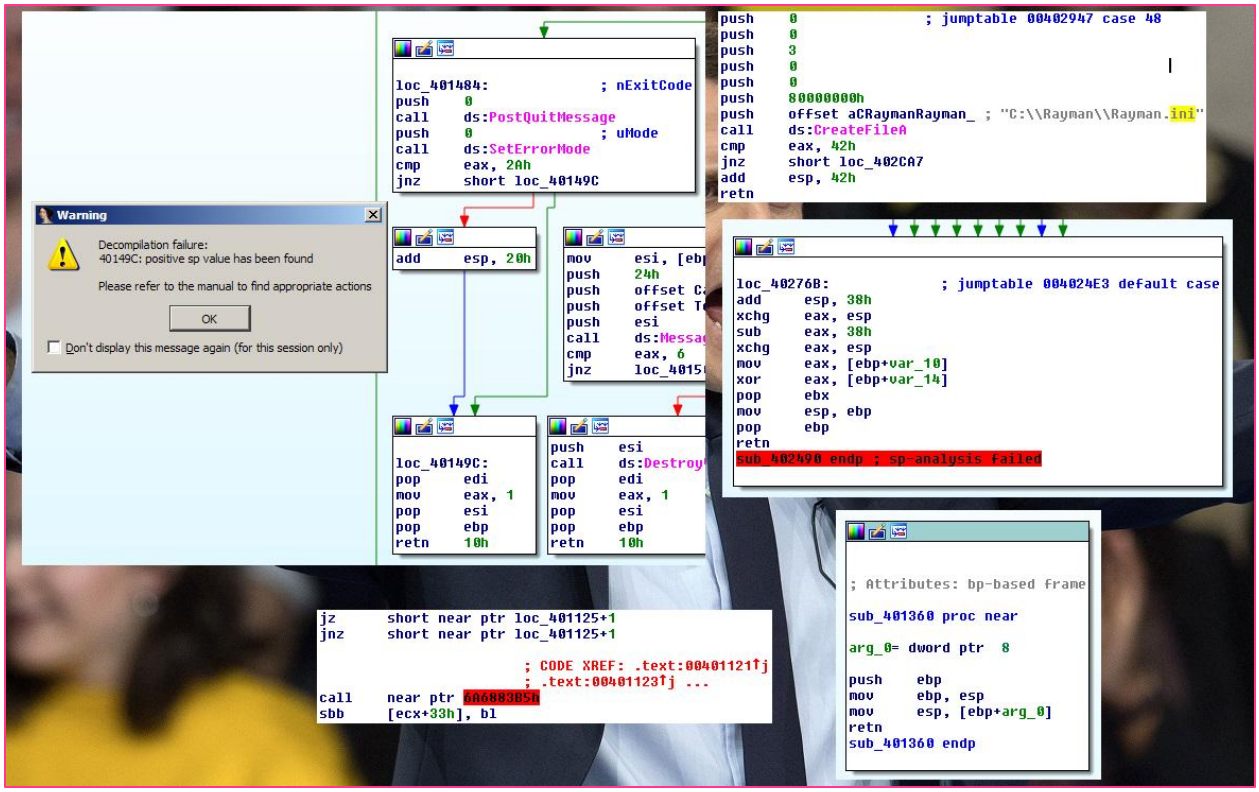

Figure 4 : exemples de trucs relous dont on se passerait bien

Notons qu'un anti-debug reposant sur la valeur du champ ***BeingDebugged*** du PEB donne des constantes fausses, et donc un serial incorrect, si un débuggeur est attaché au programme durant l'analyse.

De même, il est important de souligner qu'après avoir été modifié par l'algorithme de la VM, le serial est vicieusement encodé par blocs de 8 octets par un second thread. Le thread effectue un *ROP* sur les adresses d'instructions situées sur la pile, en commençant par changer le mode du processeur en long mode depuis le mode 32-bit [4]. Ce changement de mode suffit à confirmer la médiocrité de la plupart des débuggeurs. Les 4 blocs de 8 octets sont encodés différemment. Le code Python suivant inverse cet encoding :

```
def decode(output):
    a = struct.unpack('<Q', output[0:8])[0]
    a -= 0x8A9507C21E001D9C
    if a < 0:
        a += 0x10000000000000000
    a ^= 0x897B540210AE34FA
    a &= 0xffffffffffffffff
    a = struct.pack('<Q', a)

    b = struct.unpack('<Q', output[8:16])[0]
    b += 0x897B540210AE34FA
    b &= 0xffffffffffffffff
    b ^= 0x9C1D001EC207958A
    b = struct.pack('>Q', b)

    c = struct.unpack('<Q', output[16:24])[0]
    c = ror(c, 64 - 0x2f, 64)
    c &= 0xffffffffffffffff
    c = struct.pack('>Q', c)
```

[4] http://esec-lab.sogeti.com/posts/2011/02/16/x64-spoon.html

```
    d = struct.unpack('<Q', output[24:32])[0]
    d -= 0x6A804215C69F512F
    if d < 0:
        d += 0x10000000000000000
    d = ror(d, 64 - 48, 64)
    d ^= 0x6A804215C69F512F
    d &= 0xffffffffffffffff
    d = struct.pack('<Q', d)

    return a + b + c + d
```

Analyse de la VM

La fonction principale qui manipule le serial est une machine virtuelle. Le terme *virtuel* n'est pas utilisé ici dans le sens noble de la virtualisation style Intel VT-x ou Ramooflax, mais dans son aspect le plus ingrat, du genre faisons perdre du temps à qui m'analysera de toute façon le bougre n'a que ça à faire pour s'occuper au travail. Quelques dizaines longues heures de reverse permettent néanmoins de comprendre l'architecture et le fonctionnement de la VM :

- 3 zones mémoires : une pile, un tas et le code.
- 9 registres dont un pointeur d'instruction et un pointeur de pile.
- 16 opcodes.

Tous les accès à la mémoire et aux registres de la VM sont chiffrés et déchiffrés par l'algorithme suivant afin qu'aucune valeur en clair ne se retrouve dans la mémoire du processus :

```
#define RELOU           0xfde7f446
#define ENCRYPT(value)  ((RELOU ^ (RELOU + (value))) - RELOU)
```

Un décompilateur a été développé afin de générer un code C représentant le programme virtualisé

Analyse du programme virtualisé

L'analyse du programme virtualisé est beaucoup plus rapide. Le code C d'environ 2000 lignes produit par le décompilateur est simplifié manuellement afin de déterminer l'algorithme appliqué sur chaque bloc de 8 octets du serial. Cet algorithme peut être inversé afin de trouver le serial original, comme le montre le code C suivant :

```
static unsigned int prev r3, prev r4;

static unsigned int helper(unsigned int a, unsigned int b, unsigned int c)
{
        unsigned int x;

        b *= 0xd2d413b3;
        x = func 5571fed(b, c, stuff);

        return (sub 402490(a, 4) + a) ^ (x + b);
}

static void reverse one step(int i, unsigned int r3, unsigned int r4)
{
        unsigned int prev;

        prev r4 = r4 - helper(r3, i + 1, 0xb);
        prev r3 = r3 - helper(prev r4, i, 0);
```

```
}

static int go(unsigned int a, unsigned int b)
{
        int i;

        serial[0] = a;
        serial[1] = b;

        r3 = serial[0];
        r4 = serial[1];

        for (i = 63; i >= 0; i--) {
                reverse_one_step(i, r3, r4);
                r3 = prev_r3;
                r4 = prev_r4;
        }

        write(1, (char *)&prev_r3, 4);
        write(1, (char *)&prev_r4, 4);
}

int main(void)
{

        init();


        stuff[0] = 0x9aacc69b;
        stuff[1] = 0x89a31d8a;
        stuff[2] = 0xa5b2cd1c;
        stuff[3] = 0xb0a5ce54;

        go(0x1b2f7d73, 0xc33437b1);
        go(0x363c8420, 0x51244a87);
        go(0x323d47b2, 0x7162b3c6);
        go(0x1e3571eb, 0xfc713e49);

        return 0;
}
```

Il est surement possible de le simplifier afin d'obtenir un algorithme plus clair. Cela suffit cependant à déterminer que le serial est ***3f691f3d6eb60b343c931c22e0baa92f***, et débloque la dernière épreuve.

LabyQRinth

La dernière épreuve nécessite des compétences particulières pour être résolue : patience, abnégation et maitrise de soi. Le texte lui-même est un labyrinthe dont l'entrée se situe après le texte ***deQRypt(*** et la sortie avant ***);***. Les caractères présents sur le chemin le plus court pour sortir du labyrinthe donnent la chaîne de caractères ***ace80e6fcca1776558babe2c51d6b657658a8abcf973d1bb5c938ac9da252fd4c973***.

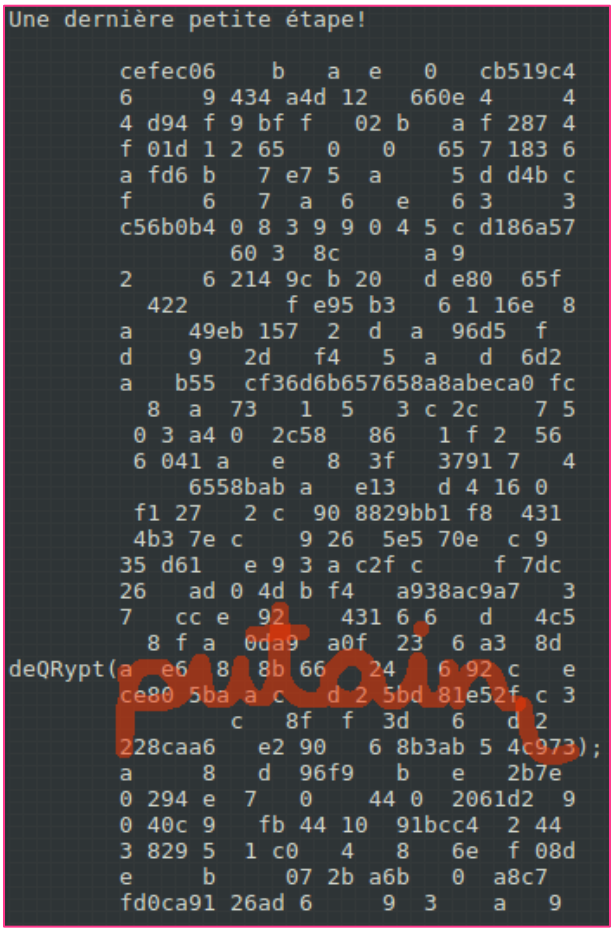

Figure 5 : tracé approximatif du chemin de sortie

Mais le fichier texte donné est aussi, avec un peu d'imagination et de bons yeux, un QR code dont l'information stockée peut-être récupérée une fois transformé en une image plus conventionnelle :

```
Please use this Nibble ADD key: 5571C2017
```

Le code Python suivant donne l'adresse e-mail signifiant la fin du calvaire, ***WwLnWZkEAeMjPaoKEs5K7rld@sstic.org*** :

```
def deQRypt(path):
    key = ('5571C2017' * 100)[:len(path)]
    ret = ''
    for i in range(len(path)):
        k = int(key[i], 16)
```

```
        z = int(code[i], 16) - k
        z += int(z < 0) * 0x10
        ret += '%x' % z
  return ret.decode('hex')

deQRypt('ace80e6fcca1776558babe2c51d6b657658a8abcf973d1bb5c938ac9da252fd4c973')
```

Conclusion

Sincères félicitations aux auteurs du challenge pour avoir réussi avec brio à concevoir un challenge original et techniquement intéressant, dont les épreuves sont variées et enrichissantes.