~~Solution~~ Recueil de poèmes du challenge SSTIC 2019

Pierre Bienaimé

27 avril 2019

Table des matières

Introduction - *J'ai besoin d'anaphores*

J'aime le challenge SSTIC ! Chaque année, j'apprends grâce à lui de nouvelles choses et je prends grand plaisir à le résoudre, même si c'est toujours un moment difficile. C'est un challenge exigeant, qui nécessite un sérieux investissement en temps et une bonne dose de motivation. Après deux semaines passées à ne penser qu'à lui (sans pouvoir pour autant m'y adonner pleinement), le monotâche invétéré que je suis termine généralement dans un triste état et finit par faire n'importe quoi. Petite démonstration en image.

Sur la forme

Lors du challenge SSTIC 2018 j'avais trouvé la dernière épreuve tellement belle qu'au moment de rédiger ma solution *normalement*, j'avais un goût de trop peu. Je voulais créer des choses. Je me suis donc laissé emporter par les envolées lyriques que les épreuves m'inspirèrent ; bravant le ridicule, j'écrivis quelques vers.

En 2018, j'ai découvert et exploré 6 formes fixes de poèmes. Et comme j'avais pris du plaisir à me plier à l'exercice, je renouvelle l'expérience pour cette édition 2019, mais en changeant quelques règles.

Cette année, mes 6 poèmes sont en alexandrins [1], uniquement en rimes suivies et avec un nombre libre de vers. Le thème de chaque poème est inspiré, plus ou moins librement, du niveau du challenge auquel il correspond. De plus, chaque poème s'articule autour d'un type de figure de style différent, sélectionné arbitrairement parmi une liste [2] bien remplie.

Le poème illustré [3] servant d'introduction est basé sur une anaphore, la répétition d'un groupe de mots au début de plusieurs phrases. Je vous laisse découvrir les autres figures de style au fur et à mesure de votre lecture.

Les puristes trouveront certainement des choses à redire dans le comptage des syllabes de certains alexandrins, ou dans le strict respect des contraintes que je me fixe. J'en suis conscient mais ce n'est rien : j'invoque la licence poétique [4] !

Sur le fond

Le challenge 2019 est composé de 4 niveaux. Dans le scénario, un individu suspect a été interpellé par la police car il est soupçonné de vouloir du mal à la communauté de la sécurité informatique française. Son téléphone portable a été analysé mais il contient plusieurs couches de sécurité. Notre aide est donc requise pour examiner ce téléphone et y trouver des preuves.

C'est un challenge qui se déroule intégralement hors ligne. La résolution de chaque niveau permet de déverrouiller un coffre-fort numérique sur le téléphone et d'accéder à l'épreuve suivante.

Comme fichiers de départ, nous disposons du bios (rom.bin) et de la flash chiffrée (flash.bin) d'un téléphone virtuel, ainsi que d'une commande **qemu** *complètement triviale* permettant de le démarrer.

1. Vers de 12 syllabes composés de deux hémistiches de 6 syllabes
2. https://fr.wikipedia.org/wiki/Liste_des_figures_de_style
3. Les mèmes qui illustrent ce poème, d'auteurs difficilement identifiables, proviennent de http://memes.at et de https://knowyourmeme.com
4. La correction des erreurs est *laissée en exercice au lecteur* :)

```
# qemu-system-aarch64 -nographic -machine virt,secure=on -cpu max -smp 1 -m 1024 -bios rom.bin
-semihosting-config enable,target=native -device loader,file=./flash.bin,addr=0x04000000
-netdev user,id=network0,hostfwd=tcp:127.0.0.1:5555-192.168.200.200:22 -net nic,netdev=network0

########################################
#      virtual environment detected      #
#           QEMU 3.1+ is needed          #
########################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: AES Key is still encrypted, need decryption
KEYSTORE: Need RSA key to decrypt
KEYSTORE: RSA private exponent is not set, please set it in the keystore or enter hex value :
```

Le téléphone nous demande de rentrer un exposant privé RSA pour poursuivre son démarrage (et déchiffrer une clé AES qui déchiffrera la flash), car le magasin de clés du suspect n'a pas pu être copié. Cependant nous disposons d'une trace de la consommation électrique du téléphone lors d'un précédent démarrage fonctionnel. Nous allons donc pouvoir nous mettre au travail et commencer le premier niveau.

1 Niveau 1 - *L'appel des zeugmes*

Le but de ce premier niveau est de réaliser une attaque par canal auxiliaire. Nous allons retrouver une clé privée RSA uniquement en analysant la consommation électrique du téléphone au moment où il la manipule. Ça peut sembler intimidant de prime abord, mais nous sommes ici face à un cas d'école, dont la résolution sera très directe.

Pour le poème de ce niveau, chaque couple d'alexandrins va contenir un zeugme. Pour paraphraser wikipédia, il s'agit d'une figure de style consistant à faire dépendre d'un même mot deux termes disparates qui entretiennent avec lui des rapports différents. Si vous voulez des exemples, jetez un œil à #PassionZeugme sur Twitter.

~ L'appel des zeugmes ~

Un vilain criminel a des comptes à régler
Mais avant tout à rendre à la communauté.

C'est un professionnel engagé pour sa cause
Et surtout pour nous nuire et nous voler des choses.

Il a très largement chiffré sa prestation
Mais aussi son smartphone en cas d'arrestation.

Me voilà investi d'une urgente mission
Ainsi que dans mon job pour le mettre en prison.

Je vais devoir prouver sa culpabilité
Et qu'un bon appareil ne reste pas muet.

Consommation d'alcool et d'électricité
Sont les deux seuls moyens de le faire parler.

Une astucieuse attaque par canaux auxiliaires
Vient vite en sauvetage de ce premier mystère.

Mais il faut se serrer les coudes et la ceinture
Car c'est le tout début d'une longue aventure.

Une vidéo [5] YouTube publiée par LiveOverflow explique comment fonctionne une analyse de consommation électrique et pourquoi ça marche tellement bien avec le chiffrement/déchiffrement RSA.

Chiffrer ou déchiffrer avec RSA, c'est simplement faire une exponentiation modulaire. Pour des raisons de performance, c'est un algorithme d'exponentiation rapide qui est utilisé. Cet algorithme travaille de manière séquentielle sur chaque bit de clé [6], et fait des opérations mathématiques différentes selon si le bit est un 1 ou un 0.

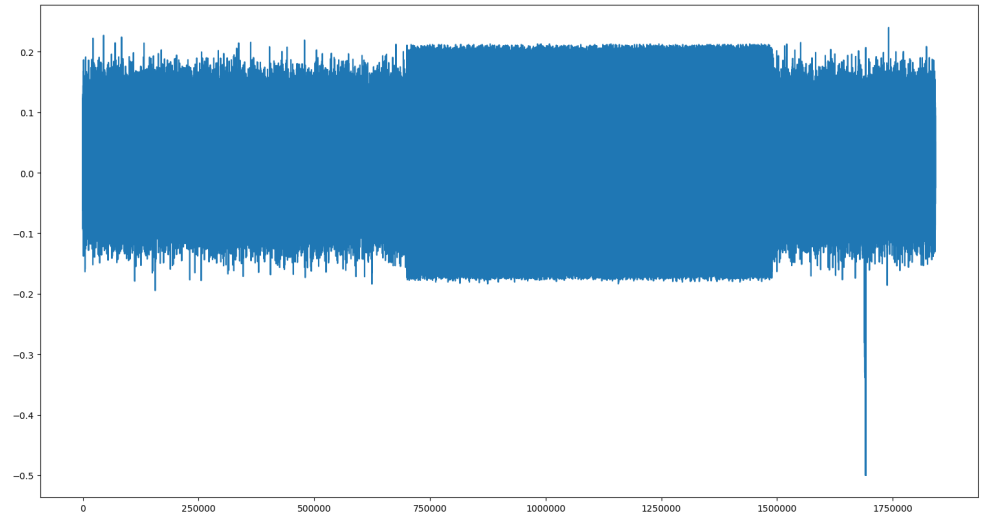
Quand un bit de clé est 0, l'algorithme d'exponentiation rapide calcule un carré. Quand ce bit est un 1, il calcule également un carré mais poursuit par une opération supplémentaire (en l'occurrence une multiplication). La conséquence directe est que le processeur va consommer davantage d'électricité (et pendant plus longtemps) quand il traitera des bits de clé à 1.

La trace de consommation électrique est fournie au format **npz**. C'est une archive zip interprétable par **numpy**. Elle contient une liste de 1842128 valeurs flottantes. On peut utiliser **numpy** et **matplotlib** pour afficher le graphe de consommation.

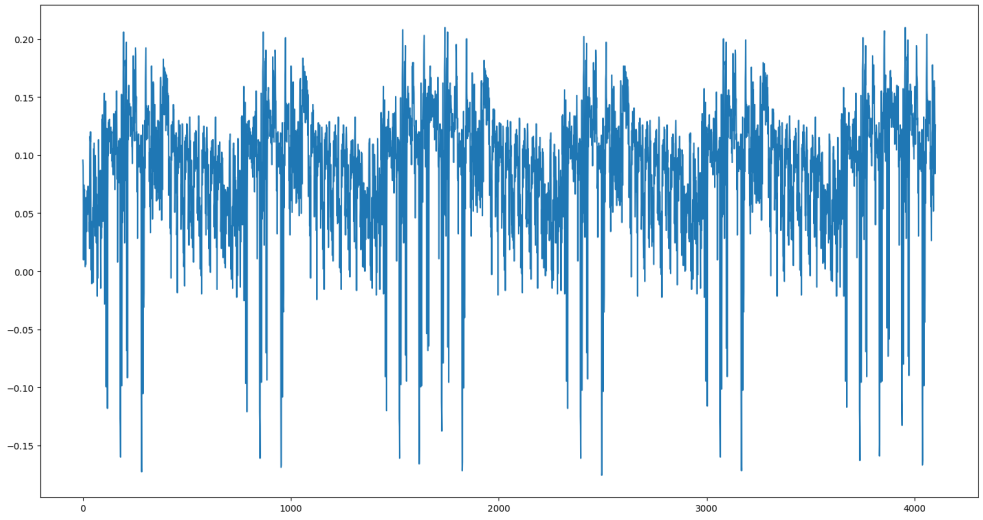
```python
import matplotlib.pyplot as plt
import numpy as np

data = np.load("power_consumption.npz")["arr_0"]
plt.plot(data)
plt.show()
```

5. https://www.youtube.com/watch?v=bFfyROX7V0s
6. Je vais utiliser plusieurs fois le terme *clé* pour ne pas alourdir l'explication, mais il s'agit bien ici d'un exposant privé d (ou d'un exposant public e)



On aperçoit un gros rectangle qui correspond à la consommation durant l'exponentiation modulaire. En zoomant sur une petite zone de ce rectangle, on voit apparaître les bits de clé. Par exemple, ci-dessous on devine que les bits de clé manipulés sont 001001.



Le gros rectangle contient 1024 séquences de ce type qui correspondent aux 1024 bits de la clé privée RSA. On va donc écrire un petit script pour extraire ces séquences et ainsi convertir les valeurs de consommation en clé. On ne va pas sortir l'artillerie lourde, un petit script ~~crade~~ simpliste fera très bien l'affaire.

On va parcourir les valeurs et compter les pics de consommation (on choisit arbitrairement les valeurs qui plongent sous -0.13). Quand il y en a beaucoup, le bit de clé est un 1. Quand il y en a peu c'est un 0. Quand il n'y en a pas pendant longtemps, c'est la

séparation entre deux bits de clé.

Dernière subtilité, l'algorithme commence par traiter les bits de poids faible de la clé, il faudra donc penser à les remettre dans le bon sens.

```python
import matplotlib.pyplot as plt
import numpy as np

data = np.load("power_consumption.npz")["arr_0"]
rsa = data[700000:1500000]
r = []
last = count = 0
for i, v in enumerate(rsa):
    if v < -0.13:
        if i - last > 400:
            # new bit
            b = "0"
            if count >= 7:
                b = "1"
            r.append(b)
            count = 1
            if len(r) == 1024:
                break
        else:
            # same bit
            count += 1
        last = i

k = "".join(r)[::-1]
key = bytearray(int(k[i:i+8], 2) for i in range(0, len(k), 8))
print("RSA key: %s" % str(key).encode("hex"))
```

```
# python stage1.py
RSA key: 23d87cdf97bb95abe6273c384190c765f552ab86f6de30a8db74435c95e6e3138f54af689812d8f9359cf0f4d453
a0c11ec68ce470216c09e74c8947adaf23e902415d61ddf2c0ffe459cbb40f7de42bdb7cd14093100a570e8c29819765e2d8d
276f86471b52ac29aa2ce2bb72cd45006279e82bec253ae9675fe45824f6001
```

Lorsque la clé est acceptée, un message nous indique d'envoyer un premier flag à challenge2019@sstic.org, puis le téléphone démarre sur un linux sur lequel on trouve l'énoncé du niveau 2 (et des coffres-forts contenant les niveaux suivants).

```
SSTIC{a947d6980ccf7b87cb8d7c246}
```

2 Niveau 2 - *Triple chiasme*

Pour le 2ème niveau, on dispose du schéma d'un composant électronique (*secure element*) en charge de contrôler l'ouverture du premier coffre-fort. Il faut suivre les fils et les portes logiques pour comprendre comment ouvrir le coffre sans connaître le code d'accès.

Le poème que m'a inspiré ce niveau contient un chiasme par quatrain. C'est une construction en croix qui consiste à utiliser un groupe de mots (ou d'idées) dans un ordre puis dans l'autre.

∼ Triple chiasme ∼

Notre point de départ est une belle image ;
Schéma d'un composant pratiquant l'enfumage,
Il crie sécurité mais c'est du cinéma ;
Voyons les éléments composant ce schéma.

Un enchevêtrement de ces portes logiques
Brasse l'information d'un procédé magique.
Vu la sécurité que ce gadget apporte
C'est fort logiquement qu'il doit prendre la porte.

Il nous faudra penser, pour résoudre l'épreuve,
A bien suivre les fils comme on suivrait des preuves.
Restons bien attentifs, car à s'en dispenser,
On risquerait de perdre le fil de nos pensées.

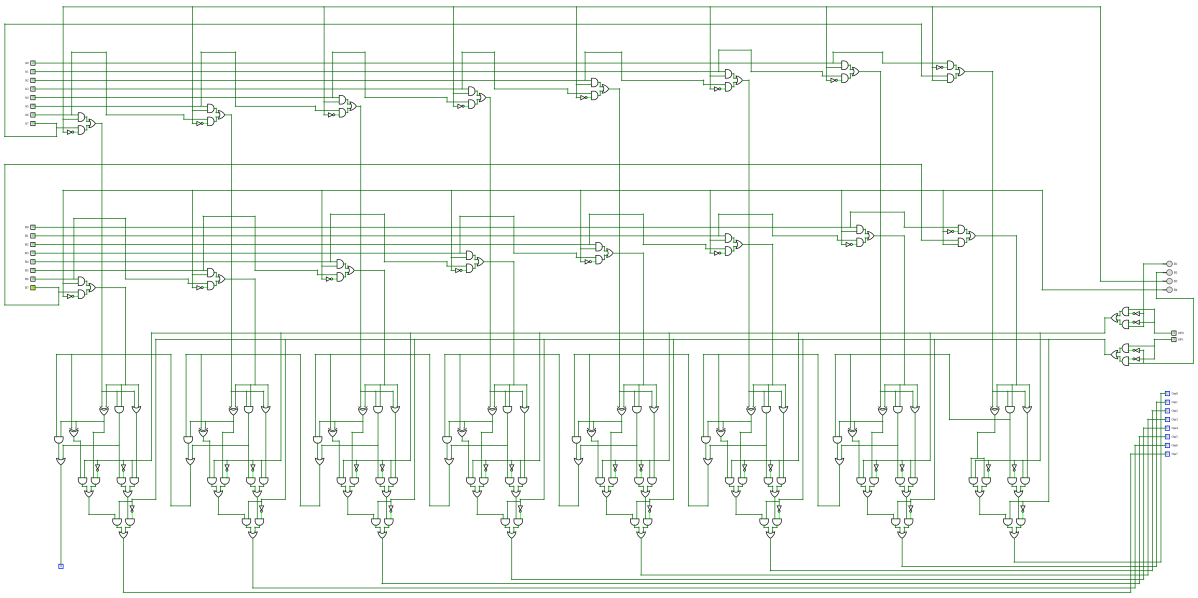
Les trois chiasmes sont sur *schéma d'un composant*, *portes logiques* et *penser à suivre les fils*. Par ailleurs, la Maison tient à vous signaler qu'elle décline toute responsabilité dans l'éventualité où la lecture de **Triple chiasme** vous aurait donné la **Triple chias␣e**. Adressez-vous plutôt à votre traiteur.

Voici ci-dessous le schéma du secure element. Un fichier python nous est également fourni. C'est un squelette qui va effectuer des actions sur le secure element... lorsque la communication vers ce dernier sera implémentée.

Une première analyse du script et du schéma nous permet déjà de tirer plusieurs conclusions. Le secure element est prévu pour prendre deux octets en entrée (A et B) ainsi qu'un opcode (OP) codé sur deux bits. Il y a donc 4 opcodes possibles.

Il va produire un octet en sortie (Out), qui correspondra à un mélange des deux entrées A et B, le type du mélange étant sélectionné par l'opcode.

Le secure element possède une autre entrée : 4 boutons qui servent d'interface avec l'utilisateur et qui lui permettront d'entrer son code d'accès. Chaque bouton va venir (légèrement) perturber les mélanges effectués par le secure element.



Le script python est en charge de construire une clé de 8 octets qui sera ensuite dérivée en clé AES pour déverrouiller le coffre. Pour chacun des 8 octets de clé, le script va effectuer une suite d'appels au secure element avec des entrées A, B et OP fixées d'avance. La seule chose qui va varier (et que nous ne connaissons pas), c'est la façon dont les boutons seront appuyés par l'utilisateur au moment de ces appels.

Il n'y a que 4 boutons, le calcul de chaque octet de clé ne pourra donc être perturbé que de 16 manières différentes. La clé de 8 octets va donc subir une drastique réduction d'entropie, puisqu'on passe de 256^8 (2^{64}) clés possibles à uniquement 16^8 (2^{32}).

Notre travail est donc de réimplémenter de manière logicielle les algorithmes de mélange du secure element. Ensuite, un bruteforce sur 4 octets permettra de trouver la clé du coffre.

Le script python contient deux séquences d'initialisation du secure element qui vont nous aider dans notre tâche. En effet, nous connaissons la valeur que doit renvoyer le secure element dans le cas où ses boutons sont tous appuyés, puis tous relâchés. Il sera donc possible de vérifier que notre implémentation logicielle est correcte avant de se lancer dans le bruteforce.

Il est temps d'analyser plus en détail le schéma et de le traduire en code python. En fonction de la valeur de l'opcode, le secure element va effectuer une opération bit à bit entre les entrées A et B. Soit une opération logique (AND, OR, XOR), soit une opération arithmétique (ADD). Cette dernière se traduit sur le schéma par un fil rétroactif chargé de propager la retenue.

On constate que les groupes de portes logiques qui s'occupent de traiter chaque bit sont dupliqués 8 fois. Il n'y a pas de piège. Si le secure element travaille au niveau des bits, notre implémentation pourra directement travailler au niveau des octets. L'implémentation tient ainsi en quelques lignes.

```python
def secure_device(a,b,op):
    B1 = 1
    B2 = 2
    B3 = 4
    B4 = 8

    # Select A and B
    if buttons & B3:
        a = rol(a, 1)
    if buttons & B4:
        b = rol(b, 1)

    # Select OP
    if buttons & B1:
        op = op ^ 1
    if buttons & B2:
        op = op ^ 2

    # Apply OP
    if op == 0:
        out = a & b
    elif op == 1:
        out = a | b
    elif op == 2:
        out = a ^ b
    elif op == 3:
        out = (a + b) & 0xff
    return out
```

Pour réaliser le bruteforce, on va s'insérer dans le script python fourni (pour pouvoir bénéficier des fonctions *init()* et *step*()*), et remplacer *main()* par *solve()*. Il va falloir calculer jusqu'à 2^{32} hashes sha256. Pour gagner un peu de temps, on va paralléliser le traitement en lançant plusieurs processus.

L'extrait de code rajouté au script get_safe1_key.py est disponible en annexe A.

En utilisant 4 processus, le bruteforce a pris environ 20 minutes.

```
# python3 get_safe1_key.py
[i] Sanity check ok
[i] Key found: 8fa4dfa9d4edbbf0
[i] Safe key: 5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a
```

On peut maintenant utiliser l'outil *tools/add_key.py* pour ajouter cette clé à notre magasin et ouvrir le premier coffre-fort. Il contient le fichier de départ du niveau 3.

```
SSTIC{5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a}
```

3 Niveau 3 - *Le goût de l'aventure*

Ce 3ème niveau est un crackme. C'est un exécutable ELF pour architecture AArch64 (ARM 64 bits) à qui on doit fournir un flag en argument, et qui nous répond *Not good*.

Ce fichier ELF joue avec les exceptions C++ pour finalement faire exécuter de manière très sournoise du bytecode DWARF. Ce bytecode implémente un algorithme cryptographique *relou*, qu'il faudra inverser pour trouver le flag.

Du DWARF caché *dans* un ELF... Il n'en faut pas plus pour trouver l'inspiration du poème de ce niveau. Mais pour que ce rapport reste tous publics, je vais plutôt m'en tenir à ma seconde idée. Laissez-moi donc vous conter l'aventure épique d'un elfe et d'un nain partis explorer une crypte-aux-graphies terrifiantes.

Ce poème-aventure en alexandrins contient des homophonies culinaires. C'est à dire que le **son** d'une recette de cuisine est caché, plus ou moins subtilement, dans chaque quatrain. Il y a donc 8 plats ou desserts à retrouver, tels que *cassoulet* ou *tiramisu*. Mais... pourquoi ? ! Heu... Eh bien... Bon, arrêtez de me juger.

Pour ceux qui se sont pris au jeu, vous trouverez la solution à la dernière page de ce rapport (en annexe H).

~ Le goût de l'aventure ~

Sous son air innocent, l'elfe est un grand guerrier.
Son arc est si mortel que nul ne peut crier.
Bien qu'il soit courageux, la douleur l'horripile ;
Affirmant son audace quelquefois il s'épile.

Gladiateur sanguinaire, le nain est un boucher.
A l'arène et ailleurs des têtes sont tranchées.
Ceux qui défient sa hache ou sa cupidité
regrettent amèrement tant de stupidité.

Duo complémentaire en quête de richesse,
Nos deux aventuriers étaient plein d'allégresse
Car des sommes d'argent, douillettes mais futures,
Attendaient sagement au fond des sépultures.

Après un long chemin les amis pénétrèrent
Dans la crypte aux graphies terriblement austères.
Le nain pourtant si fort tomba dans le coma,
Car on ne sait comment, un gourdin l'assomma.

Bienvenue au palais d'un mort-vivant frivole
Qui plus que tout le reste a horreur qu'on le vole.
Aux vendeurs et vandales il réservait ce sort :
Déguster leur cervelle, conserver leur trésor.

Voyant son compagnon en fâcheuse posture,
L'elfe bandant son arc visa la créature.
Grâce à ce tir ami soutirant un cri rauque,
Le nain eut un répit avant qu'on ne le croque.

Le mort-vivant furieux tapa contre les murs,
Ce qui eut pour effet d'effondrer la structure.
L'elfe et le nain périrent sous des rochers en nombre ;
Il prit donc son encas sous les nouveaux décombres.

Dans les yeux du zombie brillait un éclat fou ;
Titubant hors des ruines il jura sans tabou ;
Son festin fut si bon, divine nourriture,
qu'il retrouva enfin le goût de l'aventure.

En définitive, le goût de l'aventure était surtout le goût des aventuriers. L'histoire ne le dit pas, mais ça ne m'étonnerait qu'à moitié que notre ami mort-vivant était en réalité un participant du challenge SSTIC qui a essayé pendant trop longtemps d'inverser des algos de crypto qui ne sont pas inversibles.

L'exécutable ELF du niveau 3 a été analysé de manière statique avec Ghidra et de manière dynamique avec gdb. On remarque que le volume de code est extrêmement petit. A première vue, le crackme se contente de lire le flag en entrée, et de faire un *throw Exception*, au milieu d'un *try catch*, en donnant le flag comme paramètre.

Ensuite, le handler d'exception va comparer le flag avec SSTIC{congolexicomatisation}. Sauf que dans la pratique, ce code est un leurre et n'est jamais exécuté. Une fois que l'exception est lancée, on ne sait pas trop ce qui se passe. Il y a de la magie qui opère. Puis un message (d'origine inconnue) est affiché pour nous signifier que le flag est mauvais.

Pour comprendre ce qui se passe sous le capot quand l'exception C++ est levée, on cherche à se documenter et on constate que les ressources sur le sujet son assez pauvres. Il y a juste un blog qui explique absolument tout, dans une très longue et complète suite

d'articles intitulée **C++ exception handling internals / C++ exceptions under the hood** [7]. Je pense que tous les participants du challenge sont tombés sur ce blog, et que son propriétaire s'est peut-être questionné sur la raison du pic de trafic et du soudain regain d'intérêt de la communauté pour les exceptions C++.

En tout cas, le moins que l'on puisse en dire c'est que la gestion interne des exceptions C++ , c'est sacrément compliqué ! On va zapper toutes les étapes intermédiaires pour se consacrer à ce qui nous intéresse (et m'a le plus surpris). À un moment de la gestion de l'exception, il y a quelques octets de bytecode DWARF qui sont exécutés par une machine virtuelle située dans la libgcc. Et c'est le fonctionnement normal.

Dans ce challenge, les octets de bytecode DWARF qui gèrent notre exception ont été modifiés pour y introduire un jump vers la section *.gnu.hash* de l'ELF. Cette section ne contient donc pas des hashes, mais du bytecode DWARF. C'est ce code DWARF qui va implémenter un algorithme de cryptographie et se charger de la vérification du flag.

Pour résoudre cette épreuve, je suis passé par 4 étapes. Tout d'abord, comme je n'ai pas réussi à trouver un désassembleur DWARF qui fonctionne sur du bytecode brut, j'ai dû en écrire un minimaliste.

Ensuite, constatant que l'algorithme était trop compliqué pour être compris de manière statique [8], j'ai transformé mon désassembler en émulateur. En mettant des points d'arrêt au bon endroit de la machine virtuelle DWARF de la libgcc, on peut suivre le flot d'exécution et récupérer l'évolution de la pile afin de valider que les résultats de l'émulateur sont corrects.

Une fois l'émulateur DWARF fonctionnel, on l'utilise pour générer des traces d'exécution, exécuter des sous-parties de l'algorithme de manière indépendante, vérifier des hypothèses sur l'inutilité de certaines parties, poser des points d'arrêt, exécuter pas à pas, etc. Le but final est de réécrire l'algorithme cryptographique en python, dans la version la plus simplifiée possible. Cette tâche a été relativement longue car il est compliqué d'identifier les fonctions et leurs arguments d'entrée et de sortie. En effet, le DWARF est un langage uniquement à pile, et la pile n'est pas toujours gérée proprement par l'algorithme. Certaines valeurs intermédiaires devenues inutiles restent dans la pile (elle grossit donc légèrement). Et parfois une valeur située au milieu de la pile est récupérée et sert d'argument à une fonction.

Avant d'arriver à une implémentation fonctionnelle, j'ai d'abord modifié mon émulateur pour qu'il utilise le langage intermédiaire de Miasm et ses bindings z3, dans l'espoir de trouver le flag à l'aide d'un solveur SMT. Mais le solveur n'a rien trouvé dans un temps respectable.

Quand l'algorithme de crypto a été réimplémenté avec succès en python, il nous reste à écrire l'algorithme inverse. Je n'étais pas familier avec l'inversion d'algos de crypto, et j'ai donc dû me taper la tête contre le clavier un bon moment avant d'intégrer la logique de l'exercice. Un conseil pour gagner du temps : utiliser un nom de variable différent pour chaque calcul intermédiaire.

7. https://monoinfinito.wordpress.com/series/exception-handling-in-c/
8. En tout cas par moi. Je ne m'appelle pas Eddy Malou.

Le code de l'émulateur DWARF est disponible en annexe B et celui de l'implémentation python de la crypto est fourni en annexe C.

Je n'ai pas pu retrouver le nom de l'algorithme cryptographique car toutes les constantes qu'il manipule ne renvoient strictement à rien. Je ne sais pas s'il s'agit d'un algorithme connu ou non. Mais cela n'a pas empêché de l'inverser.

Dans mon implémentation, j'ai découpé le code en fonctions que j'ai nommé A(), B(), C(), D(), E(), H(), deriv() et cipher().

Inverser l'algorithme de chiffrement, c'est écrire la fonction uncipher(), ainsi que chaque fonction inverse dont elle aura besoin. En l'occurrence unA(), unB(), unD() et unE(). Les fonctions C(), H() et deriv() n'ont pas besoin d'être inversées.

Un fonction interne de deriv() s'appelle big_small_crypto() car dans la version DWARF elle est gigantesque (plus de 2000 instructions) alors qu'en réalité, elle ne fait que retourner deux constantes.

Un fois toutes les fonctions inverses écrites et vérifiées, on peut appeler uncipher() sur la valeur qui est comparée en sortie d'algo, et ainsi retrouver le flag du niveau.

```
# python solve3.py
A sanity... OK
B sanity... OK
D sanity... OK
E sanity... OK
Cipher sanity... OK
Flag: SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}
```

```
SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}
```

4 Niveau 4 - *Peau pourrie*

Ce 4ème et dernier niveau est lui aussi un crackme. Tout comme pour le niveau 3, on commence avec un exécutable ELF AArch64 qui attend un flag en entrée. Sauf que cette fois il répond *Loose* au bout de plusieurs (!!) secondes. On peut donc déjà craindre le volume et la complexité des traitements qui vérifieront ce flag.

En pratique, ce niveau renferme une machine virtuelle cachée dans le Secure World du téléphone (ARM TrustZone), coupée en plusieurs morceaux, à cheval entre le monitor et le secure OS. Le niveau est semé d'embûches. Plusieurs astuces sont utilisées pour tenter de nous perdre.

Le poème de ce niveau s'appelle **Peau Pourrie** pour deux raisons. Tout d'abord, c'est un pot-pourri de figures de style. On y trouvera en vrac allitération, métaphore, antanaclase, hyperbole, antilogie, oxymore, pléonasme, litote, ... Ensuite, parce que je

n'ai pas résisté à l'envie d'écrire une suite à l'aventure du zombie. Et la trustzone piégée de ce niveau m'a évoqué une quête dans un labyrinthe.

Non, le rapport entre le thème de ce poème et celui du niveau n'est pas capillotracté. Le zombie n'ayant plus de cheveux, je ne vois vraiment pas comment ça serait possible.

~ Peau pourrie ~

Un triste aventurier et bien piètre adversaire
Procura au zombie un délicieux dessert.

Ce dernier sursauta en trouvant sur son corps
Le tatouage encore frais d'une carte au trésor.

Il sautilla de joie en déchiffrant l'image,
On crut apercevoir un cabri nécrophage,

Car ce plan indiquait où retrouver intacte
La puissance infinie d'un divin artefact.

Depuis qu'il est un mort, il se sent si vivant
Qu'il entama de suite ce défi captivant.

Il se rendit sur place et vit monter sa crainte
Car ce qui l'attendait était un labyrinthe.

Il n'avait vraiment pas confiance en cette zone
Mais il y pénétra comme on prend l'octogone.

Il progressa ainsi, de sa démarche lente,
Dans de nombreux couloirs aux noirceurs aveuglantes.

Perdu dans les entrailles du tortueux dédale,
Il posa son pied nu sur la mauvaise dalle.

Il comprit que l'endroit était truffé de pièges
Dont certains provoquaient d'horribles sortilèges.

Une flamme infernale jaillit entre ses jambes ;
Déjà bien putréfiées désormais elles flambent.

Si sa situation semble ici si sensible,
Il su se surpasser et cesser d'être cible.

Il reprit ses esprits, rampa jusqu'au virage
Où des esprits maudits l'attaquèrent au visage.

Pour fuir ces ennemis il glissa vers le Nord
Mais tomba nez-à-nez avec un minotaure.

L'hybride le toisa d'un regard imposant ;
Voilà qui n'était pas un piteux opposant.

Le zombie estropié prit ses jambes à son cou ;
Comme il n'en avait plus, il récolta un coup.

Après un vol plané par dessus les cloisons,
Il heurta violemment l'entrée d'une maison.

Cette fois le hasard lui fut très bénéfique,
Il venait de trouver la suprême relique !

Heureux comme un poison dans l'eau d'un Lannister,
Il exultait d'avoir résolu ce mystère.

Le trésor était sien mais il gardait en tête
Que son corps en charpie devra payer ses dettes.

Il fit donc la promesse, auprès de ses amours,
De suspendre ses quêtes pour un an et un jour.

Puis il saisit enfin son butin fantastique
Et porta fièrement ce t-shirt du sstic !

Il s'agit d'un niveau très axé sur la rétroconception, ce qui m'a donné l'opportunité de bien me familiariser avec Ghidra. Par ailleurs, une retranscription fidèle de la méthodologie utilisée pour résoudre ce niveau sera difficile à réaliser. Comme souvent, la compréhension a été très progressive et empirique. Il y a eu un certain nombre d'essais/erreurs, de fausses pistes, avant d'avoir les bons déclics.

Le binaire ELF du niveau 4 est gros (1.5 Mo), mais pourtant il ne contient presque pas de code. C'est une coquille vide. Tout ce qu'il fait est d'appeler 4 ioctls. L'un pour envoyer le flag. Un autre pour envoyer un gros fichier (contenu dans le binaire) de 0x101000 octets. Le 3ème et le 4ème sont appelés sans argument dans une boucle, jusqu'à temps que le 4ème ait une certaine valeur de retour nous indiquant si le flag est valide ou non. Ce code nous fait rapidement penser à une machine virtuelle. Le gros fichier contient certainement des opcodes maison et un algo chargé de vérifier le flag. Mais une analyse d'entropie sur ce fichier nous montre qu'il est chiffré... ce qui n'est pas cool.

La filouterie du DWARF furtif du niveau précédent a laissé des traces encore fraîches et douloureuses dans ma mémoire. J'ai peur de me faire à nouveau mystifier. La première chose que j'ai faite, avant d'aller chercher où atterrissent ces 4 ioctls, c'est de recoder en python l'équivalent de ce que fait le fichier ELF. Pour valider que les ioctls existent vraiment et me répondent, ou qu'il n'y a rien de caché dans l'ELF. Le code du client python est disponible en annexe D.

Avec le recul, je constate que ce petit travail préliminaire m'a beaucoup aidé pour la suite du niveau. Ce client python m'a permis de faire une première analyse en boite noire avant de devoir aller mettre les mains dans le secure world. Il m'a ensuite aidé à identifier un des mécanismes d'antidebug.

Pour mon analyse en boite noire, j'ai observé et fait varier le peu de paramètres qui sont sous notre contrôle :

- Compter le nombre de tours, c'est à dire le nombre de fois où les ioctls 3 et 4 sont exécutés (9366 tours quand tout se passe bien).
- Faire varier le flag et constater que ça n'a pas d'influence visible depuis ici.
- Faire varier le gros fichier. Il suffit de corrompre un octet au début, et le programme sera arrêté prématurément. Plus l'octet corrompu est loin dans le fichier, plus le nombre de tours sera élevé. On en déduit que le gros fichier est traité de manière séquentielle (mais pas octet par octet, plutôt par blocs).
- Faire varier le temps entre l'appel des ioctls 3 et 4. Le nombre de tours diminue alors légèrement (9260, 9280, 9305, ...) selon le moment où la temporisation a été insérée. On ne le sait pas encore, mais on vient de mettre le doigt sur un mécanisme d'antidebug.

Toutes ces observations vont bien dans le sens de l'existence d'une machine virtuelle à l'autre bout de nos ioctls, et dont le bytecode sera contenu dans le gros fichier. Ce fichier est chiffré, mais pas par le flag. Il sera donc probablement déchiffré par une clé en dur.

Bon, allons chercher cette VM. On trouve que les ioctls sont traités par le driver sstic.ko, mais ce dernier fait juste office de passe-plat entre nous et le secure world. Bon... il se trouve où, le code qui tourne dans le secure world ? Vu qu'on est dans qemu, je peux y accéder ?

Après une phase de documentation, on démarre qemu avec l'option **-gdb**, ce qui va permettre de connecter un gdb-multiarch en remote sur le téléphone virtuel. En déroulant pas à pas les étapes de boot, j'ai dumpé en mémoire le code du monitor [9] et du secure OS.

A la suite de longues heures de reverse dans Ghidra, j'ai pu trouver et comprendre la VM. Voici ses principales caractéristiques :

— La VM est à cheval entre le monitor et le secure OS, qui passent leur temps à jouer au ping-pong.
— Les registres de la VM sont stockés de manière chiffrée dans le monitor, ils sont déchiffrés et rechiffrés à la volée grâce à des instructions AES (aese, aesd) en utilisant une partie de notre flag comme clé. C'est pénible, mais on va pouvoir faire abstraction de tout ça, par exemple en mettant des tracepoints aux endroits où les registres sont temporairement déchiffrés.
— Le code et les données du programme (i.e. le gros fichier chiffré qu'on passe en entrée) sont stockés du coté du secure OS. Le fichier n'est jamais déchiffré intégralement en mémoire, tout est fait à la volée en utilisant des exceptions. Un accès mémoire invalide provoque une exception qui réalise un déchiffrement SM4 d'un bloc du fichier.
— Les opcodes sont traités à moitié par le monitor et à moitié par le secure OS.

Par exemple, l'instruction `shr r0, 10` est traitée coté secure OS. Il va envoyer un **smc** au monitor pour récupérer la valeur de r0, faire son calcul chez lui, puis envoyer un autre **smc** pour stocker la nouvelle valeur de r0.

A l'inverse, une instruction comme `xor r1, r2` est traitée coté monitor. Les valeurs des registres ne vont donc pas transiter en clair (vu qu'elles ne vont pas transiter du tout).

Nous avons donc de quoi écrire un désassembleur (partiel) et réaliser des traces d'exécution (incorrectes) afin de commencer à analyser l'algorithme de crypto qui vérifie notre flag. Le code du désassembleur est disponible en annexe E.

Les traces d'exécution ne sont pas toutes cohérentes entre elles car des mécanismes d'antidebug viennent nous compliquer la tâche. Le désassembleur est partiel car quelques opcodes sont pénibles à suivre. Notamment, le secure OS a parfois le bon goût de passer temporairement en 32 bits, ce que ni gdb ni Ghidra n'apprécie vraiment.

Tout d'abord, dans les strings du secure OS, on trouve ce genre de chose :

```
0e20528d        (ps -ef |egrep -q 'qemu-system.* -[s] |qemu-system.* -[g]db ') && exit 42

0e2052d8        /proc/self/cmdline
```

J'ai eu de la chance, car comme j'ai dumpé le secure OS directement en mémoire avec gdb, j'ai pu voir apparaître ces chaines en clair (car sinon, dans le blob binaire du secure OS, elles sont stockées encodées). C'était gros, donc au début je pensais que c'était juste un troll. Car à quel moment le secure OS a le droit d'exécuter du code sur mon host à

9. Le machin qui tourne avec le niveau de privilège maximum, EL3

travers qemu ? ? ! Oops. En fait il peut, à cause de l'option **-semihosting**. Ça calme. J'ai un peu peur. Et si on enlève l'option, le téléphone ne démarre plus.

Cet antidebug liste les processus de mon host pour savoir si qemu a été lancé avec l'option **-gdb** ou **-s**. Pour le contourner, on peut donc commencer par changer le nom de notre binaire qemu-system-aarch64. Mais ça n'est pas suffisant car l'antidebug vérifie également /proc/self/cmdline. La solution pour s'en débarrasser a donc été de patcher un octet dans qemu-system-aarch64 pour que l'option **-gdb** devienne **-xdb**.

Si jamais cet antidebug nous détecte, alors un petit morceau du gros fichier chiffré ne sera pas mappé en mémoire. La conséquence est que la table d'indirections utilisée par l'algo de crypto ne sera pas bonne, et qu'on ne pourra donc jamais retrouver le bon flag.

Le second antidebug est celui basé sur des timers, que nous avions déjà effleuré en boite noire. J'ai compris ce qu'il faisait, mais j'ai terminé le challenge sans avoir besoin de trouver où il était implémenté, ni comment le contourner. L'un des opcodes de la VM est un jump conditionnel, dont la condition est assez opaque. Il s'avère que c'est en fait un *jump if debugger*. Si on prend le jump, on va ignorer quelques instructions de l'algo de crypto. Ça explique d'une part pourquoi le nombre d'instructions exécutées (i.e. le nombre de tours d'ioctl du début) n'est pas 9366 quand on est trop lent. Et d'autre part pourquoi l'algo de crypto n'est pas inversible si l'antidebug nous a vu.

Pour ne pas être dérangé par cet antidebug, il suffit de faire des traces d'exécution qui ne soient pas trop gourmandes en temps [10]. Par exemple, tracer la valeur des registres lors de chaque **xor** n'est pas très long (avec un **dprintf**) et permet d'accéder à presque toutes les valeurs intermédiaires intéressantes de l'algo. Si on trace l'opcode **sub**, on va directement accéder à la valeur que devra avoir notre flag en sortie d'algo.

D'ailleurs, au passage, cette valeur c'est pr.a.rfg.cnf.fv.snpvyr@ffgvp.bet (ce qui donne, en ROT13, ce.n.est.pas.si.facile@sstic.org).

La dernière brique manquante pour pouvoir travailler dans de bonnes conditions, c'est le fichier déchiffré. Pour le dumper, j'ai utilisé une petite boucle dans gdb à l'endroit où le secure OS fait ses accès mémoire invalides (pour lever des exceptions, se faire déchiffrer à la volée et retomber sur ses pattes).

```
b *0x0e200b98

define ssticdump
    set $count=0
    while ($count < 0x101010)
        set $pc=0x0e200b90
        set $x0=$count
        c
        set $count=$count+4
        printf "%08X SSTIC %08X\n", $count, $x0
    end
end
```

10. Et si jamais on a été trop lent, on va tout de suite le savoir car notre script python aura fait moins de 9366 tours

On peut maintenant désassembler directement l'algo de crypto situé au début du fichier en clair, plutôt que les traces d'exécution corrompues, avec toutes les boucles à plat. Le code désassemblé est disponible en annexe F.

Désormais, le plan de bataille est un peu le même que pour la fin du niveau 3 : recoder l'algo de crypto en python, puis écrire l'algo inverse. Le code qui calcule le flag est disponible en annexe G.

```
# python solve4.py
A sanity... OK
B sanity... OK
Cipher sanity... OK
Flag:  acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e
```

```
SSTIC{acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e}
```

Nous pouvons ainsi accéder au contenu du dernier coffre-fort, qui contient les données des applications d'un téléphone Android. On y trouve le fichier mmssms.db qui est une base sqlite contenant les SMS.

En l'examinant (avec strings puis sqlitebrowser) on découvre les preuves que l'on recherchait, et le terrible complot dont le SSTIC a été victime.

L'intoxication alimentaire du SSTIC 2018 n'était pas un hasard, c'était un coup monté !

L'adresse email de victoire est présente dans l'un des SMS échangé.

```
9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org
```

Comme autres fichiers d'intérêt, on trouve la base sqlite contenant les contacts, ainsi que 4 photos, dont le très traditionnel Lobster Dog [11].

Conclusion - *La prison métonyme*

Le challenge est terminé ! Dans ce court poème de conclusion, chaque couple d'alexandrins se termine pour une métonymie signifiant toujours la même chose : prison.

11. Information inutile : la police d'écriture cursive des poèmes a été choisie uniquement car elle s'appelle *LobsterTwo*

~ La prison métonyme ~

Grâce à tous ces efforts, le plaisir m'est offert
De dénoncer l'auteur pour qu'il soit mis aux fers.

La police est chez lui, tout le monde est paré
A défoncer sa porte pour le mettre aux arrêts.

Il aurait vraiment dû se tenir à carreau,
Il va finir sa vie derrière les barreaux.

Vous pouvez à nouveau manger votre ragoût,
L'empoisonneur véreux croupit sous les verrous

L'édition 2019 du challenge SSTIC était encore un très bon cru ! Cette année, j'ai appris beaucoup de nouvelles choses : m'initier à Ghidra, appréhender plus en détail le fonctionnement de la technologie TrustZone, inverser des algos de crypto, ou encore découvrir la magie des exceptions C++.

Je suis également content d'avoir pu recycler et entretenir quelques connaissances acquises lors des précédentes éditions. Le niveau 2 a été facile à résoudre grâce au reverse de FPGA du challenge SSTIC 2013. J'avais déjà cotoyé l'architecture AArch64 grâce au 2ème niveau du challenge SSTIC 2014. Les entourloupes discrètes du niveau 4 pouvaient être soupçonnées depuis le dernier niveau de l'édition 2017.

Je tiens à remercier ma femme, ma fille et mes collègues, qui ont eu à m'endurer pendant la résolution du challenge puis la rédaction de ce rapport.

Un grand merci également aux concepteurs, qui ont été à la hauteur de la réputation du challenge. Trouver des épreuves qui soient intéressantes à résoudre, variées, avec une difficulté croissante et bien dosée, ce n'est vraiment pas facile. Surtout pour un challenge 100% hors ligne. Ici, le défi est relevé haut la main.

Au moment de la rédaction de ce rapport, 121 personnes ont validé [12] le niveau 1, 106 le niveau 2, 26 le niveau 3 et 11 le challenge. Ce sont de bons chiffres. Les deux premiers niveaux étaient accessibles sans être triviaux pour autant. Une grande majorité des participants qui ont résolu le premier niveau ont également résolu le 2ème. Le 3ème a bien calmé les ardeurs, mais un quart des participants a malgré tout persévéré jusqu'à le résoudre. Si le challenge commençait directement au niveau 3, je pense qu'il y aurait bien

12. Publiquement, car il faut garder à l'esprit que l'envoi des flags est facultatif

moins de 26 personnes à l'avoir résolu. La tentation de l'ouvrir, de ne rien comprendre, et de baisser immédiatement les bras serait trop grande.

De mon point de vue, un bon challenge, c'est quelque chose qui doit entretenir suffisamment notre motivation, qui doit nous permettre d'acquérir des connaissances pointues, de résoudre des problèmes complexes, sans nous décourager pour autant devant la tâche à accomplir. L'accroche est importante. Ici, après avoir résolu les deux premiers niveaux (sur 4), il y a un biais cognitif qui peut nous amener à penser : *« Je suis déjà arrivé à la moitié du challenge, j'ai bien avancé, ça serait dommage d'abandonner maintenant »*. Et donc on persévère (alors qu'en réalité on a parcouru que 10% de la route). Et plus on avance, plus c'est dur mais plus c'est dommage d'abandonner si près du but.

On se retrouve ainsi coincé dans une spirale bénéfique et douloureuse à la fois. Bénéfique car elle nous stimule, elle nous donne l'énergie de résoudre des problèmes complexes, elle nous fait avancer. Douloureuse car plus la difficulté (et le sadisme des concepteurs) va croître, plus on va sortir de sa zone de confort et se sentir nul. Jusqu'au moment où on va enfin trouver le dernier flag et être libéré.

Au final, un bon challenge, c'est une suite d'épreuves suffisamment captivantes pour nous emmener jusqu'au bout et pouvoir se dire ensuite *« Je l'ai fait. J'ai appris 1000 trucs ! »*. Alors que si on nous avait directement présenté à l'avance l'ampleur de tout ce qu'il faudrait accomplir, notre réaction aurait juste été *« No way. Pas le temps. Pas intéressant. Pas pour moi. »*.

L'année dernière, pour le 1er niveau du challenge SSTIC 2018, j'avais écrit un poème qui résume les idées que je viens de développer. Je me permets donc de le recycler. C'est un Madrigal, un poème d'amour adressé à une femme – ici incarnée par le challenge SSTIC – avec qui j'entretiens une relation ambivalente. Il est en vers de 12 syllabes mais pas en alexandrins (il n'y a pas de césure à la 6ème syllabe). Chers lecteurs, à défaut de pots de vin [13], je vous présente le **pot de miel**.

~ Pot de miel ~

Cette première épreuve est un préliminaire ;
Un baiser dans le cou de notre tortionnaire
Qui nous appâte par ses lèvres au goût de miel
Avant de nous forcer à manger tout le pot.
Mais mon appétit eu raison de ce suppôt
Qui m'emmena fortuitement au septième ciel.

Merci pour tout et à l'année prochaine !

13. https://twitter.com/Karion_/status/1116766665721749505

A Niveau 2 : get_safe1_key.py (extrait)

```python
import multiprocessing
import itertools
import time

global buttons

def step(i):
    return eval("step%i()" % i)

def split(l, n):
    for i in range(n):
        yield l[i::n]

def _bruteforce(k0, keys):
    for key in itertools.product(k0, *keys[1:]):
        k = bytearray(key)
        if hashlib.sha256(k).hexdigest() == "00c8bb35d44dcbb2712a11799d8e1316045d64404f337f4ff653c27607f436ea":
            return k

def bruteforce(keys, processes=4):
    pool = multiprocessing.Pool(processes)
    # split keys
    k = list(keys[0])
    r = []
    for k0 in split(k, processes):
        p = pool.apply_async(_bruteforce, (k0, keys))
        r.append(p)
    pool.close()

    while r:
        time.sleep(0.2)
        for p in r:
            if p.ready():
                result = p.get()
                if result is not None:
                    return result
                r.remove(p)

def solve():
    # sanity checks
    global buttons
    buttons = 0
    assert init() == 0xE0
    buttons = 0xF
    assert init() == 0xA1
    info("Sanity check ok")

    # get possible values of each key byte
    keys = []
    for i in range(1,9):
        k = set()
        for b in range(0x10):
            buttons = b
            k.add(step(i))
        keys.append(k)

    # bruteforce key
    key = bruteforce(keys, processes=4)
    info("Key found: %s" % key.hex())

    # derive key to get key of the safe
    aes_key = hashlib.scrypt(key,salt =b"sup3r_s3cur3_k3y_d3r1v4t10n_s4lt",n=1<<0xd,r=1<<3,p=1<<1,dklen=32)
    info("Safe key: %s" % aes_key.hex())
```

B Niveau 3 : dwarf_emul.py

```python
import sys
import struct
import random


def read2s(x):
    return struct.unpack("<h", x[:2])[0]

def read4u(x):
    return struct.unpack("<I", x[:4])[0]

def read8u(x):
    return struct.unpack("<Q", x[:8])[0]

def rnd(size):
    return "".join([chr(random.randint(30, 0x80)) for i in range(size)])

class Mem(object):
    def __init__(self):
        self.d = {}

    def _get_base_addr(self, addr):
        off = abs(max(x - addr for x in self.d if x <= addr))
        if addr-off in self.d and len(self.d[addr-off]) > off:
            return addr-off, off
        else:
            raise Exception("Memory address 0x%x is not mapped" % addr)

    def __setitem__(self, k, v):
        self.d[k] = bytearray(v)

    def __getitem__(self, k):
        key, off = self._get_base_addr(k)
        return self.d[key][off]

    def __getslice__(self, begin, end):
        addr, off = self._get_base_addr(begin)
        return self.d[addr][off:off+(end-begin)]

    def __str__(self):
        return str(self.d)

    def __repr__(self):
        return repr(self.d)

class Stack(list):
    def push(self, it):
        self.append(it)

    def __str__(self):
        return str([hex(x) for x in self])

class EmulDwarf(object):
    def __init__(self):
        self.verbose = False
        self.pause = False
        self.single_pause = False
        self.flow = False
        self.mem = Mem()
        self.stack = Stack()
        self.regs = [None]*32
        self.start = 0x400258
        self.pc = self.start
        self.dbg = set()
        self.init_mem()

    def init_mem(self):
        key = "00112233445566778899AABBCCDDEEFF"
        #key = rnd(32)
```

```
70          print "key used:", repr(key)
71          with open("dwarf.bin") as f:
72              self.mem[self.start] = f.read()
73          self.mem[0xfffffffffb78] = struct.pack("<Q", 0xfffffffffc68)
74          self.mem[0xfffffffffc70] = struct.pack("<Q", 0xfffffffffe90)
75          self.mem[0xfffffffffe90] = key + "\x00"
76          self.regs[31] = 0xfffffffffad0
77
78      def custom_end(self):
79          print "End"
80          print self.stack
81          string_ptr = self.stack.pop()
82          if string_ptr == 0x4030b8:
83              print "Bad flag"
84          elif string_ptr == 0x403098:
85              print "Good flag"
86          else:
87              raise Exception("Bad string_ptr value %r !?" % string_ptr)
88          sys.exit(0)
89
90      def step(self):
91          op = self.mem[self.pc]
92          msg = ""
93          addr = self.pc
94          self.pc += 1
95          MASK = (1 << 64) - 1
96
97          if op == 0:
98              pass
99
100         if op == 0x6:
101             a = self.stack.pop()
102             v = read8u(self.mem[a:a+8])
103             self.stack.push(v)
104             msg = "DW_OP_deref"
105
106         elif op == 0x8:
107             arg = self.mem[self.pc]
108             self.pc += 1
109             self.stack.push(arg)
110             msg = "DW_OP_const1u  0x%x" % arg
111
112         elif op == 0xc:
113             arg = read4u(self.mem[self.pc:self.pc+4])
114             self.pc += 4
115             self.stack.push(arg)
116             msg = "DW_OP_const8u  0x%08x" % arg
117
118         elif op == 0xe:
119             arg = read8u(self.mem[self.pc:self.pc+8])
120             self.pc += 8
121             self.stack.push(arg)
122             msg = "DW_OP_const8u  0x%016x" % arg
123
124         elif op == 0x12:
125             self.stack.push(self.stack[-1])
126             msg = "DW_OP_dup"
127
128         elif op == 0x13:
129             self.stack.pop()
130             msg = "DW_OP_drop"
131
132         elif op == 0x15:
133             arg = self.mem[self.pc]
134             self.pc += 1
135             self.stack.push(self.stack[-(arg+1)])
136             msg = "DW_OP_pick  0x%x" % arg
137
138         elif op == 0x16:
139             a = self.stack.pop()
140             b = self.stack.pop()
141             self.stack.push(a)
142             self.stack.push(b)
143             msg = "DW_OP_swap"
144
```

```
145             elif op == 0x17:
146                 a = self.stack.pop()
147                 b = self.stack.pop()
148                 c = self.stack.pop()
149                 self.stack.push(a)
150                 self.stack.push(c)
151                 self.stack.push(b)
152                 msg = "DW_OP_rot"
153 
154             elif op == 0x1a:
155                 a = self.stack.pop()
156                 b = self.stack.pop()
157                 self.stack.push(a & b)
158                 msg = "DW_OP_and"
159 
160             elif op == 0x1c:
161                 a = self.stack.pop()
162                 b = self.stack.pop()
163                 self.stack.push((b - a) & MASK)
164                 msg = "DW_OP_minus"
165 
166             elif op == 0x1e:
167                 a = self.stack.pop()
168                 b = self.stack.pop()
169                 self.stack.push((a * b) & MASK)
170                 msg = "DW_OP_mul"
171 
172             elif op == 0x21:
173                 a = self.stack.pop()
174                 b = self.stack.pop()
175                 self.stack.push(a | b)
176                 msg = "DW_OP_or"
177 
178             elif op == 0x22:
179                 a = self.stack.pop()
180                 b = self.stack.pop()
181                 self.stack.push((a + b) & MASK)
182                 msg = "DW_OP_plus"
183 
184             elif op == 0x24:
185                 a = self.stack.pop()
186                 b = self.stack.pop()
187                 self.stack.push((b << a) & MASK)
188                 msg = "DW_OP_shl"
189 
190             elif op == 0x25:
191                 a = self.stack.pop()
192                 b = self.stack.pop()
193                 self.stack.push(b >> a)
194                 msg = "DW_OP_shr"
195 
196             elif op == 0x27:
197                 a = self.stack.pop()
198                 b = self.stack.pop()
199                 self.stack.push(a ^ b)
200                 msg = "DW_OP_xor"
201 
202             elif op == 0x28:
203                 # BRANCH if not 0
204                 a = self.stack.pop()
205                 arg = read2s(self.mem[self.pc:self.pc+2])
206                 self.pc += 2
207                 if a == 0:
208                     msg = "DW_OP_bra  %s  (not taken)" % hex(arg)
209                 else:
210                     self.pc += arg
211                     msg = "DW_OP_bra  %s  (%s)" % (hex(arg), hex(self.pc))
212                     if self.flow:
213                         print hex(self.pc)[-3:], "b"
214 
215             elif op == 0x2f:
216                 arg = read2s(self.mem[self.pc:self.pc+2])
217                 self.pc += 2
218                 self.pc += arg
219                 msg = "DW_OP_skip  %s  (%s)" % (hex(arg), hex(self.pc))
```

```
220             if self.flow:
221                 print hex(self.pc)[-3:], "s"
222             if arg == 0x7fff:
223                 self.custom_end()
224 
225         elif 0x30 <= op <= 0x4f:
226             r = op - 0x30
227             self.stack.push(r)
228             msg = "DW_OP_lit%i" % r
229 
230         elif 0x50 <= op <= 0x6f:
231             r = op - 0x50
232             reg = self.regs[r]
233             if reg is None:
234                 raise Exception("reg%i not initialized" % r)
235             self.stack.push(reg)
236             msg = "DW_OP_reg%i" % r
237 
238         elif op == 0x94:
239             arg = self.mem[self.pc]
240             self.pc += 1
241             a = self.stack.pop()
242             if arg == 1:
243                 v = self.mem[a]
244             elif arg == 4:
245                 v = read4u(self.mem[a:a+4])
246             else:
247                 raise Exception("Deref_size not handled: %i" % arg)
248             self.stack.push(v)
249             msg = "DW_OP_deref_size  0x%x" % arg
250 
251         else:
252             raise Exception("Unknown OP 0x%x" % op)
253 
254         if self.verbose or self.single_pause:
255             print "0x%x %s" % (addr, msg)
256             print self.stack
257             print
258 
259         if self.pause or self.single_pause:
260             x = raw_input()
261             if x == "s":
262                 self.verbose = True
263                 self.pause = True
264             elif x == "q":
265                 self.verbose = False
266                 self.pause = False
267             if self.single_pause:
268                 self.single_pause = False
269 
270 
271     def run(self):
272         while True:
273             self.step()
274 
275     def custom_run(self):
276         # put your breakpoints here
277         while True:
278             if self.pc == 0x400438:
279                 print "Algo5 START"
280                 self.single_pause = True
281             if self.pc == 0x400285:
282                 print "Checking result"
283                 a = self.stack[-1]
284                 print hex(a)
285                 self.verbose = True
286                 self.pause = True
287             self.step()
288 
289 
290 def main():
291     e = EmulDwarf()
292 #    e.custom_run()
293     e.run()
294 
```

```python
if __name__ == "__main__":
    main()
```

C Niveau 3 : solve3.py

```python
import struct
import random

# 4006b4
H_TAB = [1499706267, 821517021, 272624828, 3747450, 903838431, 3785014547, 3673731932, 4168712724, 3003917061,
108626153, 1761663296, 56526703, 2743431188, 1456005474, 3916730198, 2446648329, 3704052233, 2892363055, 2238831879,
27188470, 1372518891, 514961789, 180241519, 992443646, 2218289596, 2990449304, 2083090257, 3447298418, 1645742692,
180376277, 4027015685, 320352659, 3104361364, 1274334524, 3739604106, 4132412020, 3595179472, 2999637044, 1692991803,
4100768210, 2641947099, 293114390, 861309414, 122453723, 3874185605, 3491763979, 4077169173, 3170023070, 1320407786,
3616361820, 2127740868, 1491857651, 2337872959, 3540262763, 6073992, 3127471964, 3245117150, 2596059943, 1456967031,
1985691168, 3476496685, 422456783, 2360289286, 3164231515, 2003050958, 1384102379, 1777877824, 2914810839, 1945289513,
1937738367, 3808547812, 92192434, 1482274904, 751049081, 3966346809, 1404097827, 966980914, 2334778205, 1155714689,
3423641965, 1155137368, 1997548564, 4216815997, 1022871752, 304252554, 1098110417, 891940402, 2057172366, 1362437548,
2174045100, 3461598453, 2031519528, 2731958520, 4077701582, 4038630518, 144683945, 3373915048, 530468291, 2723761676,
889184632, 2957251835, 3542070074, 407700088, 746678938, 3113480946, 3839385628, 1239609054, 2631208161, 3225296503,
2835449502, 1970921040, 1919599787, 953174080, 3938325104, 2353037613, 3897128631, 388695213, 1969620676, 1713734801,
3047210744, 1808555251, 2771758740, 3380250538, 2432936182, 3426459231, 3139485215, 4291817367, 2425704214, 1343938982,
1859488205, 1747365931, 1449128547, 2391553691, 1771093626, 891664448, 407361467, 895107271, 2740428560, 3924683950,
3493034673, 1666573754, 2291618645, 1589703631, 2698387773, 1966642055, 3825342192, 3649096928, 1820903036, 2191344028,
1005995291, 300973050, 3300988238, 3171366379, 2633555869, 3830955637, 1671871768, 4024926163, 2202147683, 2139293485,
1150553953, 4173188599, 1640497866, 170981800, 2146450231, 2725037481, 3775813389, 2515143702, 2690542338, 3357365124,
2005261061, 2462600287, 89671191, 2193424013, 1436459482, 1339411888, 2642878445, 3715061049, 1322381094, 4268233668,
486175189, 3579919082, 253533068, 734708829, 1708134209, 1755046104, 1756369152, 1666962048, 2504875339, 1921548153,
3456268917, 4243594342, 6953939, 429530254, 2279009829, 2475597576, 3634437453, 731532673, 3069563621, 3512480346,
1693102017, 4248024604, 1136770391, 934867016, 1555725554, 1921060048, 2885612628, 2808317114, 3632665664, 920541239,
258606065, 282365857, 1485040687, 2831158427, 1636826788, 2690490886, 4290374010, 2318886785, 827765881, 3836698317,
2279202927, 3033846873, 3952770341, 1507851946, 3700159827, 1842232378, 560109326, 3905524381, 1815780340, 1171936146,
2632902576, 1063834859, 3442033487, 239990546, 2980628122, 471404358, 725976892, 1185931209, 1642629183, 2542868679,
1895403953, 528638029, 3247174799, 3005740287, 1369077000, 4200700685, 646780048, 1451112386, 631273105, 1111710144,
322047059, 4025959165, 1220101562, 3483952913, 2405414169, 728969893, 279606908]

# 400678
deriv_TAB = [3593965546, 3795626180, 2229514684, 3462271167, 2729665754, 1106399472, 266956864, 482366054, 385384011,
1365124791, 4003761611, 1655920612, 968695873, 1926051962, 4084903051, 1499706267]

C_TAB = [0x489dddde, 0x67990f1, 0x95bf74a9, 2006195943, 242057443, 770551691, 4220702018, 3504891840, 4071859195,
1815727687, 3880753632, 1512370721]

Mask4 = (1<<32) - 1

def ror(x, n, bits=8):
    mask = (2**n) - 1
    mask_bits = x & mask
    return (x >> n) | (mask_bits << (bits - n))

def rol(x, n, bits=8):
    return ror(x, bits - n, bits)

def split_key(k):
    return [struct.unpack("<Q", k[i:i+8])[0] for i in range(0, 32, 8)]

# cut a qword into two dwords
def cut(a):
    lo = a & Mask4
    hi = a >> 32
    return lo, hi

def H(a, b):
    for i in range(4):
```

```
        loa, hia = cut(a)
        lob, hib = cut(b)
        q = (hia + lob) & Mask4
        r = q ^ loa
        s = hia & lob
        t = (lob - r) & Mask4
        idx = (hib & 0xFF)
        u = H_TAB[idx]
        v = (r + u) & Mask4
        w = (hib >> 8) ^ v
        x = (w << 32) | t
        y = (s << 32) | v
        a = y
        b = x
    return y, x

def deriv(k3, k4, idx=0):
    lo3 = k3 & Mask4
    hi3 = k3 >> 32
    lo4 = k4 & Mask4
    hi4 = k4 >> 32
    a = deriv_TAB[idx] ^ lo3
    b = (0x45786532 + hi3) &  Mask4
    c = b ^ lo4
    d = rol(hi4, 4, 32)
    e = (a - b) & Mask4
    f = big_small_crypto(c)
    g = b ^ f
    h = d ^ g
    i = (d << 32) | c
    j = (h << 32) | e
    return i, j

def big_small_crypto(c):
    if c & 0x80000000:
        return 0x60bf080f
    else:
        return 0x818f694a

# 0x4004ea
def C(a, b):
    """
    Size:
     input(4,4)
     output(4,4)
    """
    for idx in range(6):
        x = C_TAB[idx*2]
        y = C_TAB[(idx*2)+1]
        c = (x + b) & Mask4
        d = c ^ a
        e = d | y
        f = (e ^ b)
        a = d
        b = f
    return d, f

# 0x4003f2
def A(a, b):
    """
    Size:
     input(8,8)
     output(8)
    """
    loa, hia = cut(a)
    lob, hib = cut(b)
    lob, hib = C(lob, hib)
    c = hia ^ hib
    d = loa ^ lob
    e = rol(d, 4, 32)
    f = e ^ hib
    g = ror(c, 18, 32)
    h = g ^ lob
    i = (h << 32) | f
    return i
```

```
135
136 def unA(i, b):
137     f, h = cut(i)
138     lob, hib = cut(b)
139     lob, hib = C(lob, hib)
140     g = h ^ lob
141     c = rol(g, 18, 32)
142     e = f ^ hib
143     d = ror(e, 4, 32)
144     loa = d ^ lob
145     hia = c ^ hib
146     a = (hia << 32) | loa
147     return a
148
149 def B(a, b):
150     loa, hia = cut(a)
151     loa, hia = C(loa, hia)
152     lob, hib = cut(b)
153     c = hia ^ lob
154     d = rol(c, 26, 32)
155     e = loa ^ hib
156     f = e ^ hia
157     g = ror(f, 14, 32)
158     h = (g << 32) | d
159     return h
160
161 def unB(a, h):
162     loa, hia = cut(a)
163     loa, hia = C(loa, hia)
164     d, g = cut(h)
165     f = rol(g, 14, 32)
166     e = f ^ hia
167     hib =e ^ loa
168     c = ror(d, 26, 32)
169     lob = c ^ hia
170     b = (hib << 32) | lob
171     return b
172
173 def D(a, b, c, d, i):
174     r1, r2 = deriv(c, d, i)
175     r3 = A(a, b)
176     r4 = r2 ^ r3
177     r5 = B(r4, b)
178     r6 = r1 ^ r5
179     return r4, r6
180
181 def unD(r4, r6, c, d, i):
182     r1, r2 = deriv(c, d, i)
183     r5 = r6 ^ r1
184     r3 = r4 ^ r2
185     b = unB(r4, r5)
186     a = unA(r3, b)
187     return a, b
188
189 def E(p1, p2, p3, p4):
190     s3 = p3
191     s4 = p4
192     for i in range(15):
193         r1, r2 = D(p1, p2, p3, p4, i)
194         p1 = r1
195         p2 = r2
196         if i == 14:
197             break
198         r3, r4 = (s3, s4)
199         for j in range(i+1):
200             r4, r3 = deriv(r3, r4, j)
201         p3 = r3
202         p4 = r4
203     return p1, p2
204
205 def unE(r1, r2, p3, p4):
206     s3 = p3
207     s4 = p4
208     for i in range(14, -1, -1):
209         r3, r4 = (s3, s4)
```

```
        for j in range(i):
            r4, r3 = deriv(r3, r4, j)
        p3 = r3
        p4 = r4
        p1, p2 = unD(r1, r2, p3, p4, i)
        r1 = p1
        r2 = p2
    return p1, p2

def cipher(p1, p2, p3, p4):
    for i in range(4):
        h3, h4 = H(p3, p4)
        p1, p2 = E(p1, p2, h3, h4)
        h1, h2 = H(p1, p2)
        p3, p4 = E(p3, p4, h1, h2)
    return p1, p2, p3, p4

def uncipher(p1, p2, p3, p4):
    for i in range(4):
        h1, h2 = H(p1, p2)
        p3, p4 = unE(p3, p4, h1, h2)
        h3, h4 = H(p3, p4)
        p1, p2 = unE(p1, p2, h3, h4)
    return p1, p2, p3, p4

def sanity_check():
    flag = "00112233445566778899AABBCCDDEEFF"
    f1, f2, f3, f4 = split_key(flag)
    assert(cipher(f1, f2, f3, f4) == (
        0x5d7df1a93f31b826,
        0x302ccd27ed9cd844,
        0xf0bb476762999b56,
        0x43482469a28bf4ac))

    x = random.randint(0, (1<<64)-1)
    y = random.randint(0, (1<<64)-1)
    print "A sanity...",
    r = A(x, y)
    assert(unA(r, y) == x)
    print "OK"

    print "B sanity...",
    r = B(x, y)
    assert(unB(x, r) == y)
    print "OK"

    print "D sanity...",
    a, b, c, d = [random.randint(0, (1<<64)-1) for x in range(4)]
    r1, r2 = D(a, b, c, d, 4)
    assert(unD(r1, r2, c, d, 4) == (a,b))
    print "OK"

    print "E sanity...",
    a, b, c, d = [random.randint(0, (1<<64)-1) for x in range(4)]
    x, y = E(a, b, c, d)
    assert(unE(x, y, c, d) == (a,b))
    print "OK"

    print "Cipher sanity...",
    a, b, c, d = [random.randint(0, (1<<64)-1) for x in range(4)]
    p1, p2, p3, p4 = cipher(a, b, c, d)
    assert(uncipher(p1, p2, p3, p4) == (a, b, c, d))
    print "OK"

def main():
    sanity_check()

    # hardcoded result to decrypt
    a = 0x65850b36e76aaed5
    b = 0xd9c69b74a86ec613
    c = 0xdc7564f1612e5347
    d = 0x658302a68e8e1c24

    f1, f2, f3, f4 = uncipher(a, b, c, d)
    flag = "".join(struct.pack("<Q", f) for f in [f1, f2, f3, f4])
```

```
285     print "Flag:", flag
286
287 if __name__ == "__main__":
288     main()
```

D Niveau 4 : ioctl.py

```
1  import fcntl
2  import ctypes
3
4  IOCTL_SEND_FILE  = 0xc0105300
5  IOCTL_SEND_FLAG  = 0xc0105301
6  IOCTL_VM_GO      = 0xc0105302
7  IOCTL_GET_RESULT = 0xc0105303
8
9  class Buffer(ctypes.Structure):
10     _fields_ = [
11         ("pData", ctypes.c_void_p),
12         ("size",  ctypes.c_uint64),
13     ]
14
15 def init_buf(data, flag):
16     encrypted = Buffer()
17     encrypted.size = len(data)
18     x = (ctypes.c_char * len(data)).from_buffer(data)
19     encrypted.pData = ctypes.c_void_p(ctypes.addressof(x))
20
21     flag_t = Buffer()
22     flag_t.size = len(flag)
23     x = (ctypes.c_char * len(flag)).from_buffer(flag)
24     flag_t.pData = ctypes.c_void_p(ctypes.addressof(x))
25     return encrypted, flag_t
26
27 def run(encrypted, flag):
28     with open("/dev/sstic", "r") as fd:
29         print("Sending file (0xc0105300)")
30         fcntl.ioctl(fd, IOCTL_SEND_FILE, encrypted)
31
32         print("Sending flag (0xc0105301)")
33         fcntl.ioctl(fd, IOCTL_SEND_FLAG, flag)
34
35         count = 0
36         r = 0
37         while (r & 0xffff) != 0xffff:
38             fcntl.ioctl(fd, IOCTL_VM_GO, 0)
39             r = fcntl.ioctl(fd, IOCTL_GET_RESULT, 0)
40             count += 1
41             if r & 0xffff == 1:
42                 print("Finished")
43                 print(r)
44                 if r >> 16 == 0:
45                     print("Win")
46                 else:
47                     print("Loose")
48                 break
49         if r == 0xffff:
50             print("Failure")
51         print(count)
52
53 def main():
54     with open("code.bin", "rb") as f:
55         data = bytearray(f.read())
56     flag = bytearray(b"ABCDEFGHIJKLMNOPQRSTUVWXYZ012345")
57     c, f = init_buf(data, flag)
```

```
58      run(c, f)
59
60  if __name__ == "__main__":
61      main()
```

E Niveau 4 : disass.py

```
1   import sys
2   import struct
3
4   def decode(instr):
5       op = instr >> 20
6       optype = (instr >> 18) & 3
7       r0 = (instr >> 14) & 0xf
8       r1 = (instr >> 10) & 0xf
9       imm = instr & 0x3fff
10
11      if optype == 0: # REG, REG
12          if op == 0:
13              return "mov r%i, r%i" % (r0, r1)
14
15          elif op == 1:
16              return "dec r%i" % r0
17
18          elif op == 2:
19              return "add r%i, r%i" % (r0, r1)
20
21          elif op == 3:
22              return "sub r%i, r%i" % (r0, r1)
23
24          elif op == 6:
25              return "xor r%i, r%i" % (r0, r1)
26
27          elif op == 11:
28              return "swp r%i" % r0
29
30      elif optype == 1: # REG, MEM
31          if op == 0:
32              return "mov r%i, [r%i]" % (r0, r1)
33
34      elif optype == 2: # MEM, REG
35          if op == 0:
36              return "mov [r%i], r%i" % (r0, r1)
37
38      elif optype == 3: # REG, IMM
39          if op == 0:
40              return "mov r%i, 0x%x" % (r0, imm)
41
42          elif op == 2:
43              return "add r%i, 0x%x" % (r0, imm)
44
45          elif op == 3:
46              return "sub r%i, 0x%x" % (r0, imm)
47
48          elif op == 4:
49              return "shl r%i, 0x%x" % (r0, imm)
50
51          elif op == 5:
52              return "shr r%i, 0x%x" % (r0, imm)
53
54          elif op == 7:
55              return "and r%i, 0x%x" % (r0, imm)
56
57          elif op == 8:
```

```
58              return "jmp 0x%x" % (imm)
59
60          elif op == 9:
61              return "jnz r%i, 0x%x" % (r0, imm)
62
63          elif op == 12:
64              return "!!! r%i" % r0
65
66          elif op == 13:
67              return "??? r%i" % r0
68
69          elif op == 14:
70              return "jdb 0x%x" % imm
71
72      return Exception("UNSUPPORTED (op %i, optype %i, r%i, r%i, imm 0x%x)" % (op, optype, r0, r1, imm))
73
74  def main():
75      if len(sys.argv) == 2:
76          a = sys.argv[1]
77          if a.startswith("0x"):
78              a = a[2:]
79          print decode(int(a, 16))
80      else:
81          with open("decrypted2.bin", "rb") as f:
82              data = f.read()
83
84          for i in range(0, len(data), 3):
85              instr = struct.unpack("<I", data[i:i+3] + "\x00")[0]
86              if instr == 0:
87                  print "END"
88                  break
89              disass = decode(instr)
90              print "%08X %08X  %s" % (i, instr, disass)
91
92  if __name__ == "__main__":
93      main()
```

F Niveau 4 : disass.txt

```
00000000 000D0010  mov r4, 0x10
00000003 004D0010  shl r4, 0x10
00000006 002D0020  add r4, 0x20
00000009 000F4010  mov r13, 0x10
0000000C 004F4010  shl r13, 0x10
0000000F 002F4020  add r13, 0x20
00000012 000F0004  mov r12, 0x4
00000015 00041000  mov r0, [r4]
00000018 004C0010  shl r0, 0x10
0000001B 005C0010  shr r0, 0x10
0000001E 00B00000  swp r0
00000021 002D0002  add r4, 0x2
00000024 00045000  mov r1, [r4]
00000027 004C4010  shl r1, 0x10
0000002A 005C4010  shr r1, 0x10
0000002D 00B04000  swp r1
00000030 002D0002  add r4, 0x2
00000033 00049000  mov r2, [r4]
00000036 004C8010  shl r2, 0x10
00000039 005C8010  shr r2, 0x10
0000003C 00B08000  swp r2
0000003F 002D0002  add r4, 0x2
00000042 0004D000  mov r3, [r4]
```

```
00000045 004CC010  shl r3, 0x10
00000048 005CC010  shr r3, 0x10
0000004B 00B0C000  swp r3
0000004E 000F8020  mov r14, 0x20
00000051 000DC007  mov r7, 0x7
00000054 00DD0000  ??? r4
00000057 00138000  dec r14
0000005A 00010400  mov r4, r1
0000005D 00015000  mov r5, r4
00000060 005D0008  shr r4, 0x8
00000063 007D00FF  and r4, 0xff
00000066 007D40FF  and r5, 0xff
00000069 0002D400  mov r11, r5
0000006C 004EC008  shl r11, 0x8
0000006F 00029C00  mov r10, r7
00000072 004E8010  shl r10, 0x10
00000075 0022AC00  add r10, r11
00000078 00229000  add r10, r4
0000007B 002E9000  add r10, 0x1000
0000007E 0005A800  mov r6, [r10]
00000081 007D80FF  and r6, 0xff
00000084 009DC08A  jnz r7, 0x8a
00000087 000DC00A  mov r7, 0xa
0000008A 0011C000  dec r7
0000008D 0002D000  mov r11, r4
00000090 004EC008  shl r11, 0x8
00000093 00029C00  mov r10, r7
00000096 004E8010  shl r10, 0x10
00000099 0022AC00  add r10, r11
0000009C 00229800  add r10, r6
0000009F 002E9000  add r10, 0x1000
000000A2 00056800  mov r5, [r10]
000000A5 007D40FF  and r5, 0xff
000000A8 009DC0AE  jnz r7, 0xae
000000AB 000DC00A  mov r7, 0xa
000000AE 0011C000  dec r7
000000B1 0002D800  mov r11, r6
000000B4 004EC008  shl r11, 0x8
000000B7 00029C00  mov r10, r7
000000BA 004E8010  shl r10, 0x10
000000BD 0022AC00  add r10, r11
000000C0 00229400  add r10, r5
000000C3 002E9000  add r10, 0x1000
000000C6 00052800  mov r4, [r10]
000000C9 007D00FF  and r4, 0xff
000000CC 009DC0D2  jnz r7, 0xd2
000000CF 000DC00A  mov r7, 0xa
000000D2 0011C000  dec r7
000000D5 0002D400  mov r11, r5
000000D8 004EC008  shl r11, 0x8
000000DB 00029C00  mov r10, r7
000000DE 004E8010  shl r10, 0x10
000000E1 0022AC00  add r10, r11
000000E4 00229000  add r10, r4
000000E7 002E9000  add r10, 0x1000
000000EA 0005A800  mov r6, [r10]
000000ED 007D80FF  and r6, 0xff
000000F0 009DC0F6  jnz r7, 0xf6
000000F3 000DC00A  mov r7, 0xa
000000F6 0011C000  dec r7
000000F9 00025800  mov r9, r6
000000FC 004E4008  shl r9, 0x8
000000FF 00225000  add r9, r4
00000102 00023800  mov r8, r14
00000105 005E0003  shr r8, 0x3
00000108 007E0001  and r8, 0x1
0000010B 00CF8000  !!! r14
0000010E 009E012F  jnz r8, 0x12f
00000111 00020C00  mov r8, r3
00000114 0000F800  mov r3, r14
00000117 002CC001  add r3, 0x1
0000011A 0060C000  xor r3, r0
0000011D 00ECC14A  jdb 0x14a
```

```
00000120 0060C400  xor r3, r1
00000123 00002400  mov r0, r9
00000126 00004800  mov r1, r2
00000129 0000A000  mov r2, r8
0000012C 008C014A  jmp 0x14a
0000012F 00020000  mov r8, r0
00000132 00002400  mov r0, r9
00000135 00007800  mov r1, r14
00000138 002C4001  add r1, 0x1
0000013B 00604000  xor r1, r0
0000013E 00604800  xor r1, r2
00000141 00EC414A  jdb 0x14a
00000144 00008C00  mov r2, r3
00000147 0000E000  mov r3, r8
0000014A 009F8057  jnz r14, 0x57
0000014D 00B00000  swp r0
00000150 00B04000  swp r1
00000153 004C4010  shl r1, 0x10
00000156 00200400  add r0, r1
00000159 000B4000  mov [r13], r0
0000015C 002F4004  add r13, 0x4
0000015F 00B08000  swp r2
00000162 00B0C000  swp r3
00000165 004CC010  shl r3, 0x10
00000168 00208C00  add r2, r3
0000016B 000B4800  mov [r13], r2
0000016E 002F4004  add r13, 0x4
00000171 00013400  mov r4, r13
00000174 00130000  dec r12
00000177 009F0015  jnz r12, 0x15
0000017A 000F0010  mov r12, 0x10
0000017D 004F0010  shl r12, 0x10
00000180 000EC020  mov r11, 0x20
00000183 003F4020  sub r13, 0x20
00000186 000D0000  mov r4, 0x0
00000189 00043400  mov r0, [r13]
0000018C 007C00FF  and r0, 0xff
0000018F 00047000  mov r1, [r12]
00000192 007C40FF  and r1, 0xff
00000195 00300400  sub r0, r1
00000198 009C01A1  jnz r0, 0x1a1
0000019B 00011000  mov r4, r4
0000019E 008C01A7  jmp 0x1a7
000001A1 000D0001  mov r4, 0x1
000001A4 008C01A7  jmp 0x1a7
000001A7 002F4001  add r13, 0x1
000001AA 002F0001  add r12, 0x1
000001AD 0012C000  dec r11
000001B0 009EC189  jnz r11, 0x189
000001B3 00001000  mov r0, r4
000001B6 00A54000  exit
```

G Niveau 4 : solve4.py

```python
import struct

# target == pr.a.rfg.cnf.fv.snpvyr@ffgvp.bet
#        == rot13(ce.n.est.pas.si.facile@sstic.org)
TARGET = [0x7072, 0x2e61, 0x2e72, 0x6667,
          0x2e63, 0x6e66, 0x2e66, 0x762e,
          0x736e, 0x7076, 0x7972, 0x4066,
          0x6667, 0x7670, 0x2e62, 0x6574]

with open("decrypted_code.bin", "rb") as f:
```

```
    DATA = f.read()

def A(a, b, x):
    addr = 0x1000 + (x << 16) + (b << 8) + a
    dword = struct.unpack("<I", DATA[addr:addr+4])[0]
    c = dword & 0xff
    return c, a

def unA(c, a, x):
    for i in range(0x100):
        addr = 0x1000 + (x << 16) + (i << 8) + a
        dword = struct.unpack("<I", DATA[addr:addr+4])[0]
        if dword & 0xff == c:
            b = i
    return a, b

def B(c, x):
    a = c >> 8
    b = c & 0xff
    c, a = A(a, b, x)
    x = (x - 1) % 10
    d, c = A(c, a, x)
    x = (x - 1) % 10
    e, d = A(d, c, x)
    x = (x - 1) % 10
    f, e = A(e, d, x)
    x = (x - 1) % 10
    r = (f << 8) + e
    return r, x

def unB(r, x):
    f = r >> 8
    e = r & 0xff
    x = (x+1) % 10
    e, d = unA(f, e, x)
    x = (x+1) % 10
    d, c = unA(e, d, x)
    x = (x+1) % 10
    c, a = unA(d, c, x)
    x = (x+1) % 10
    a, b = unA(c, a, x)
    c = (a << 8) + b
    return c, x

def cipher(c0, c1, c2, c3):
    x = 7
    for i in range(31, -1, -1):
        a, x = B(c1, x)
        if i & 8:
            t = c0
            c0 = a
            c1 = c0 ^ c2 ^ (i+1)
            c2 = c3
            c3 = t
        else:
            u = c3
            c3 = c0 ^ (i+1)
            c3 = c3 ^ c1
            c0 = a
            c1 = c2
            c2 = u
    d0 = c0
    d1 = c1
    d2 = c2
    d3 = c3
    return d0, d1, d2, d3

def uncipher(d0, d1, d2, d3):
    x = 9
    for i in range(32):
        c1, x = unB(d0, x)
        if i & 8:
            t = c3
            c3 = c2
            c2 = d0 ^ d1 ^ (i+1)
```

```
            c0 = t
        else:
            c3 = d2
            c2 = d1
            c0 = c1 ^ d3 ^ (i+1)
        d0 = c0
        d1 = c1
        d2 = c2
        d3 = c3
    c0 = d0
    c1 = d1
    c2 = d2
    c3 = d3
    return c0, c1, c2, c3

def main():
    # sanity checks
    print "A sanity...",
    a, b, x = (0x78, 0x12, 9)
    c, a = A(a, b, x)
    assert(unA(c, a, x) == (a, b))
    print "OK"

    print "B sanity...",
    assert(unB( *B(0x4689, 8)) == (0x4689, 8))
    print "OK"

    print "Cipher sanity...",
    c0, c1, c2, c3 = (0x4142, 0x4344, 0x4546, 0x4748)
    d0, d1, d2, d3 = cipher(c0, c1, c2, c3)
    assert(uncipher(d0, d1, d2, d3) == (c0, c1, c2, c3))
    print "OK"

    # compute flag
    r = []
    for idx in range(3, -1, -1):
        i = idx*4
        r = list(uncipher(TARGET[i], TARGET[i+1], TARGET[i+2], TARGET[i+3])) + r
    flag = "".join("%04x" % x for x in r)
    print "Flag: ", flag

if __name__ == "__main__":
    main()
```

H Miam ?

Voici les 8 recettes qu'il fallait trouver dans **le goût de l'aventure** :

— Riz pilaf
— Bouchées à la reine
— Andouillette
— Macarons
— Vol-au-vent
— Tiramisu
— Cassoulet
— Clafoutis