Solution Challenge SSTIC 2015

NB : La lecture de ce document nécessite l'utilisation d'un User-Agent spécifique

ţ����A"*=;&#}\HWP����k]"Jx�����#1���?
9#�����Su)w8'[V���^
%��X#����p>AIb�����z�o���k6}WA^��Tq��OS#]#
W2�����(~j'''', o����k# ��+9"#-
6hCMiqNlZF#|#rn-x<eE��T2!D#FO��WM#
%#Nw����OE1��7C��y����#cJ##Ú#���~@:
#���u##Ǵ73"���Ď����#
uM#v'n,]��4��Ġj?S��M��v^V��Z'#{=<#���ə|
ū##".q''##h H ḫ#_0��F��O��v 4?
#&US¥a��Hf#######^G###F##h��l(h���,G����j/#!
���6Atl����5����#rqrA\   ###
#f���x%
#&##H?~��ȅy#8OG#####L9$k#~
E,Zl####MT1######9jɣ#####

###zA#####m#@a####GÒR####8#####O########*
cO####S##W]###h S   GG #

#k>ṢaI x#X##hV(>cQB#####j5Uz#`ON##]Dq
v##Á##m{`+v)
J####+l#qg#4###:####8s_a@##ᴗN#hy##`i
▫#########+B</U#JG##W#o###!
Q##__#N#####55755i#Ꮒ#}2~#|l[K##7

AH C'EST MOINS MARRANT QUAND ON EST DE L'AUTRE COTE, HEIN ?!

NB : L'auteur tient à présenter ses excuses au lecteur, une lecture normale peut reprendre

Introduction

Le cru 2015 du challenge SSTIC consiste à analyser l'image d'une carte MicroSD retrouvée dans une clé USB étrange[1].

L'image de la-dite carte MicroSD est téléchargeable à l'adresse suivante : http://static.sstic.org/challenge2015/challenge.zip

```
$ unzip challenge.zip
Archive:  challenge.zip
  inflating: sdcard.img
$ file sdcard.img
sdcard.img: DOS/MBR boot sector, code offset 0x3c+2, OEM-ID "mkfs.fat",
sectors/cluster 4, root entries 512, Media descriptor 0xf8, sectors/FAT 244,
sectors/track 32, heads 64, sectors 250000 (volumes > 32 MB) , serial number
0xe50d883b, unlabeled, FAT (16 bit)
```

Part 1 : The Duck

Un premier reflexe est de lister les fichiers, effacés ou non, présents sur la carte :

```
$ fls sdcard.img
r/r * 4:    build.sh
r/r 6:      inject.bin
v/v 3992179:      $MBR
v/v 3992180:      $FAT1
v/v 3992181:      $FAT2
d/d 3992182:      $OrphanFiles
```

On remarque la présence de 2 fichiers, dont l'un est effacé. Récupérons les :

```
$ icat sdcard.img 4 > build.sh
$ icat sdcard.img 6 > inject.bin
$ cat build.sh
java -jar encoder.jar -i /tmp/duckyscript.txt
```

Le fichier build.sh contient une ligne de commande permettant l'encodage d'un script destiné à une clé Rubber Ducky.

Une rapide recherche nous amène vers le project ducky-decode[2] permettant d'effectuer le décodage d'un fichier préalablement encodé.

```
$ ./ducky-decode/trunk/ducky-decode.pl inject.bin > decoded.bin ^C
fab@killerwhale ~/challenges/sstic2015 $ file decoded.bin
decoded.bin: ASCII text, with very long lines
```

1 Etrange car il y a peu de périphériques USB nécessitant UN USER-AGENT SPECIFIQUE POUR LIRE UN CERTAIN CONTENU

2 https://code.google.com/p/ducky-decode/

```
$ head decoded.bin
00ff 007d
GUI R

DELAY 500

ENTER

DELAY 1000
 c m d
ENTER
```

Le décodage semble s'être déroulé correctement. L'observation du fichier montre qu'il s'agit d'un enchainement de commandes de type :

```
powershell -enc XXXX
```

où XXXX est une longue chaîne encodée en base64.
De longues recherches nous permettent d'en déduire qu'il va falloir décoder ces chaînes.

```
for x in `grep 00a0 decoded.bin | perl -pe 's/( |00a0)//g'`; do echo -n $x |
base64 -d >> bbb; done
```

Une fois décodées, nous nous retrouvons face à de multiples instances de ce type de code :

```
function write_file_bytes{
      param([Byte[]] $file_bytes, [string] $file_path = ".\stage2.zip");
      $f = [io.file]::OpenWrite($file_path);
      $f.Seek($f.Length,0);
      $f.Write($file_bytes,0,$file_bytes.Length);
      $f.Close();
}
function check_correct_environment{
      $e=[Environment]::CurrentDirectory.split("\");
      $e=$e[$e.Length-1]+[Environment]::UserName;
      $e -eq "challenge2015sstic";
}
if(check_correct_environment){
      write_file_bytes([Convert]::FromBase64String('XXXX'));
}else{
      write_file_bytes([Convert]::FromBase64String('VAByAHkASABhAHIAZABlAHIA'));
}
```

Là encore, XXXX est une longue chaîne encodée en base64, tandis que la chaîne « VAByAHkASABhAHIAZABlAHIA » peut potentiellement être décodée en « TUVASENBOUFFERJUSQUAUBOUTDEMASTEGANO ».

Une fois encore, la décision de décoder ces longues chaînes encodées est prise.

Il est à noter que la dernière chaîne contenait du code powershell différent, visant à vérifier le bon contenu du fichier « stage2.zip » :

```
function hash_file{
      param([string] $filepath);
      $sha1 = New-Object -TypeName
```

```
System.Security.Cryptography.SHA1CryptoServiceProvider;
      $h =
[System.BitConverter]::ToString($sha1.ComputeHash([System.IO.File]::ReadAllBytes
($filepath)));
      $h
}
$h = hash_file(".\stage2.zip");
if($h -eq "EA-9B-8A-6F-5B-52-7E-72-65-20-19-31-3C-25-B5-6A-D2-7C-7E-C6"){
      echo "You WIN";
}else{
      echo "You LOSE";
}
```

Une fois le décodage des chaînes et leur concaténation effectués avec élégance, nous obtenons un fichier stage2.zip conforme.

```
$ for x in `cat bbb | perl -pe 's/;/;\x0a/g' | perl -pe 's/\x00//g' | grep
FromBase64 | grep -v VAByAHkASABhAHIAZABlAHIA | awk -F "'" '{print $2}'`; do
echo -n $x | base64 -d >> stage2.zip; done

$ sha1sum stage2.zip
ea9b8a6f5b527e72652019313c25b56ad27c7ec6  stage2.zip
$ file stage2.zip
stage2.zip: Zip archive data, at least v1.0 to extract
```

Part 2 : The Rocket Jumper

Voyons le contenu de l'archive obtenue :

```
$ unzip stage2.zip
Archive:  stage2.zip
 extracting: encrypted
  inflating: memo.txt
  inflating: sstic.pk3
```

PK3 … comme dans Quake3 ?

```
$ cat memo.txt
Cipher: AES-OFB
IV: 0x5353544943323031352d537461676532
Key: Damn... I ALWAYS forget it. Fortunately I found a way to hide it into my
favorite game !

SHA256: 91d0a6f55cce427132fc638b6beecf105c2cb0c817a4b7846ddb04e3132ea945 -
encrypted
SHA256: 845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187 -
decrypted
```

Le SHA-256 du fichier encrypted est exact, et l'on peut alors soupçonner que celui du fichier déchiffré le sera également.

```
$ sha256sum encrypted
91d0a6f55cce427132fc638b6beecf105c2cb0c817a4b7846ddb04e3132ea945  encrypted
```

Un détail saute alors aux yeux : si l'on XOR les deux hash entre eux, puis avec la clé

59ee0d8627db7c6ca9c46f9e0c9d5c908738e462a2ef2ce2cac8bd7bd05502eb, un message des auteurs du challenge apparaît :

```
>>>
h1="91d0a6f55cce427132fc638b6beecf105c2cb0c817a4b7846ddb04e3132ea945".decode("he
x")
>>>
h2="845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187".decode("he
x")
>>>
key="59ee0d8627db7c6ca9c46f9e0c9d5c908738e462a2ef2ce2cac8bd7bd05502eb".decode("h
ex")
>>> ''.join(chr(ord(h1[i])^ord(h2[i])^ord(key[i])) for i in xrange(len(h1)))
"La stegano, c'est tres rigolo :)"
```

Sans plus attendre, lançons une partie sur la map sstic.pk3 :

Illustration 1: L'arrivée dans un hangar

En parcourant la map, nous constatons la présence de 8 sprites contenant des chaines hexa de 8 caractères en 3 couleurs différentes, ce qui tombe bien, car mettre bout à bout les chaînes d'une même couleur formerait une potentielle clé de 128 bits. Chaque sprite est associée à un symbole spécifique.

L'une des sprites est cachée et nécessite l'ouverture d'une porte via un bouton :

Illustration 2: Un bouton

Illustration 3: Une sprite cachée derrière une porte

Une autre est située sur une plateforme mouvante ne pouvant être atteinte à l'aide d'un simple saut.
L'auteur ayant autrefois sévit sur les serveurs Quake3 sous le pseudonyme de « P3dr0_L3_g1g0L0 », l'ancéstrale technique du rocket jump ne lui est pas inconnue. La récupération du lance roquettes nécessaire à l'opération a toutefois nécessité de passer le mode de jeu en « Elimination », afin de disposer directement de toutes les armes.

Illustration 4: Une sprite sur une boite se déplaçant

Une fois les sprites trouvées, il est possible de se téléporter dans une autre zone à l'aide d'une roquette bien placée dans le logo SSTIC. L'utilisation du rocket jump est encore une fois nécessaire, pour passer à la fois la lave et le téléporteur.

Illustration 5: De la lave (c'est chaud)

Une fois les obstacles franchis, un interrupteur permet à nouveau de se téléporter dans la salle finale, où apparaît la clé (ou presque).

Illustration 6: Une salle avec des têtes de mort. Pas de doute, c'est un challenge de sécurité.

Illustration 7: La clé ! La clé ! Sort ! Sort ! Sort !

La clé finale est donc :
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac

Le script Python suivant permet de déchiffrer le fichier chiffré :

```
#!/usr/bin/python
from Crypto.Cipher import AES

data=open("encrypted","rb").read()
key =
"9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac".decode("hex")
iv = "53535449433230313352d537461676532".decode("hex")
cipher = AES.new(key, AES.MODE_OFB, iv)
open("out.bin","wb").write(cipher.decrypt(data))
```

Le fichier déchiffré est à nouveau une archive ZIP.

Part3 : The Mouse

Une fois encore, observons le contenu de l'archive obtenue :

```
$ unzip p3.zip
Archive:  ../../p3.zip
 extracting: encrypted
  inflating: memo.txt
  inflating: paint.cap
```

Le fichier memo.txt a le contenu suivant :

```
Cipher: Serpent-1-CBC-With-CTS
IV: 0x53535449433230313352d537461676533
Key: Well, definitely can't remember it... So this time I securely stored it
with Paint.

SHA256: 6b39ac2220e703a48b3de1e8365d9075297c0750e9e4302fc3492f98bdf3a0b0 -
encrypted
SHA256: 7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2 -
decrypted
```

Le fichier paint.cap est la capture USB d'une souris. Le memo nous indique que l'on va devoir l'interpréter pour comprendre ce qui a été dessiné dans Paint.

Une souris USB envoie des informations sur son état par le biais d'interrupts. Le premier byte contient l'état des boutons, et les 2 suivants indiquent les mouvements sur les axes X et Y.

Illustration 8: Une souris USB

Nous pouvons tout d'abord extraire les octets transmis par la souris via l'utilisation intuitive de l'outil tshark :

```
$ tshark -T pdml  -r paint.cap | grep usb.capdata | awk -F '"' '{print $10}' >
outfile
```

Le fichier de sortie est alors traité en Python afin d'afficher sous forme graphique les mouvements de la souris :

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

import struct
from PIL import Image

newpic = Image.new('RGB', (4000, 4000), 'white')
pixels = newpic.load()

x, y = 2000,2000
fff=open("outfile","rb")
for ll in fff:
  ll = ll[:-1]
  lt = ll.split(":")
  data=[]
  for k in lt:
    v=int(k,16)
    if v>127:
      data.append(v-256)
    else:
      data.append(v)
  button = data[0]
  x = x + data[1]
  y = y + data[2]

  if (button == 1):
    pixels[x , y] = (0, 0, 0, 0)
  else:
    pixels[x, y] = (0, 0, 255, 0)
newpic.save('paint.png', 'PNG')
```

Le résultat est immonde, mais néanmoins lisible (en zoomant, sisi) :

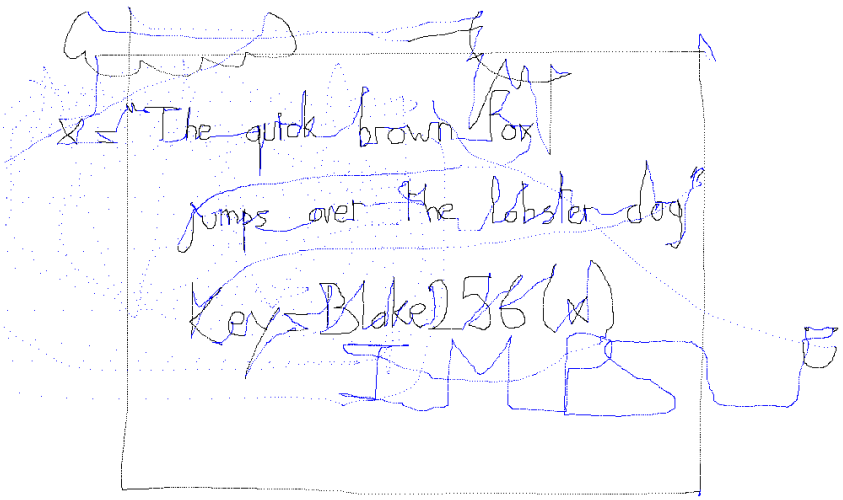


Illustration 9: L'immondice

Contrairement à d'autres participants ne disposant pas d'une vue supersonique, nous

constatons immédiatement que la clé est le hash Blake256 de la phrase « The quick brown fox jumps over the lobster dog ».

Une implémentation de Blake256 est rapidement trouvée. La clé est donc : 66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d

Le problème est maintenant de trouver une implémentation de Serpent-1-CBC-CTS. Une implémentation de Serpent est disponible dans la bibliothèque CryptoPlus[3] pour Python. Il ne nous reste qu'à implémenter le mode CTS pour déchiffrer l'archive.

La page Wikipedia « Mode d'opération (cryptographie) » fournit une représentation graphique du mode CTS permettant de réaliser l'implémentation.

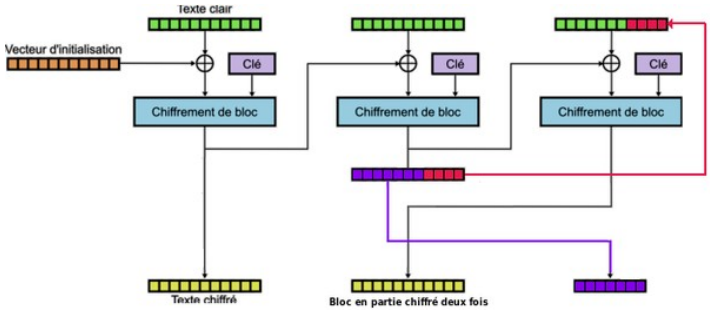


Illustration 10: Le mode CTS

Nous pouvons alors faire l'implémentation dans un script Python :

```
#!/usr/bin/python

from CryptoPlus.Cipher import python_Serpent

key="66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d".decode("h
ex")
iv="535354494323031352d537461676533".decode("hex")

data=open("encrypted","rb").read()
myserp = python_Serpent.new(key,mode=python_Serpent.MODE_CBC,IV=iv)
out = myserp.decrypt(data[:-16-14])
last = data[-14:]
prev_last = data[-16-14:-14]
prev_prev_last = data[-16-16-14:-16-14]
iv = last+'\0\0'
myserp = python_Serpent.new(key,mode=python_Serpent.MODE_CBC,IV=iv)
final = myserp.decrypt(prev_last)
missing = final[14:16]
final = final[:14]
ni = last+missing
myserp = python_Serpent.new(key,mode=python_Serpent.MODE_CBC,IV=prev_prev_last)
finalm1 = myserp.decrypt(ni)

open("final.bin","wb").write(out+finalm1+final)
```

Après une longue exécution, nous disposons du fichier déchiffré.

3 https://github.com/doegox/python-cryptoplus

```
$ sha256sum final.bin

7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2  final.bin
```

Part 4 : The Firefox

L'archive nouvellement obtenue contient un fichier HTML :

```
$ unzip p4_ok.zip
Archive:  p4_ok.zip
  inflating: stage4.html
```

L'auteur a soudainement décidé d'utiliser son MacBook Pro, sur lequel il avait installé Firefox, pour observer le contenu de cette page.

Illustration 11: Le niveau offert aux utilisateurs Macintosh n'effectuant pas leurs mises à jour ...

Il semblerait que cette partie soit résolue …

Mais l'auteur n'a en fait pas de MacBook Pro …

La page contient une variable data contenant des données hex-encodées.

Du code Javascript visiblement obfusqué est également présent. L'obfuscation est effectuée en créant un objet nommé « $ » contenant plusieurs champs, à partir desquels le reste du code sera construit.

Le code fait un moment appel à $.$(). Il suffit alors de remplacer cet appel par document.write() pour voir s'afficher le code final, toujours obfusqué via des noms de variables plutôt illisibles (le code ne sera pas inclut dans la présente solution pour des raisons d'hygiène évidentes).

Un rapide passage de Python avec un dictionnaire pour remplacer les variables par leur contenu, puis l'utilisation du site jsbeautifier.org permettent d'obtenir un code presque lisible :

```
document[write]('<h' + 1 + '>Download manager</h' + 1 + '>');
document[write]('<div id":\"' + status + '\"><i>loading...</i></div>');
document[write]('<div style":\"display:none\"><a target":\"blank\"
href":\"chrome://browser/content/preferences/preferences.xul\">Back to
preferences</a></div>');

function func1(arg1) {
    var1 = [];
    for (counter1 = 0; counter1 < arg1[length]; ++counter1) var1[push]
(arg1[charCodeAt](counter1));
    return new Uint8Array(var1);
}
function func2(arg1) {
    var1 = [];
    for (counter1 = 0; counter1 < arg1[length] / 2; ++counter1) var1[push]
(parseInt(arg1[substr](counter1 * 2, 2), 16));
    return new Uint8Array(var1);
}
function func3(arg1) {
    var1 = "";
    for (counter1 = 0; counter1 < arg1[byteLength]; ++counter1) {
        write = arg1[counter1][toString](16);
        if (write[length] < 2) var1 += 0;
            var1 += write;
    }
    return var1;
}

function func4() {
    IV = func1(window["navigator"]["userAgent"][substr](window["navigator"]
["userAgent"][indexOf]("(") + 1, 16));
    KEY = func1(window["navigator"]["userAgent"][substr](window["navigator"]
["userAgent"][indexOf](")") - 16, 16));
    _$ = {};
    _$[name] = "AES-CBC";
    _$[iv] = IV;
```

```
    _$[length] = _$__[length] * 8;
    window.crypto.subtle[importKey]("raw", KEY, _$, false, [decrypt])[then]
(function(_$write) {
        window.crypto.subtle[decrypt](_$, _$write, func2(data))[then]
(function(_decrypt) {
            getElementById$ = new Uint8Array(_decrypt);
            window.crypto.subtle[digest]({
                name: 'SHA-1'
            }, getElementById$)[then](function(getElementByIdwindow) {
                if (hash == func3(new Uint8Array(getElementByIdwindow))) {
                    hash_ = {};
                    hash_[type] = "application/octet-stream";
                    hash = new Blob([getElementById$], hash_);
                    innerHTML_ = URL[createObjectURL](hash);
                    __[getElementById](status)[innerHTML] = "<a href": "" +
innerHTML_ + \" download\":"
                    stage5.zip ">download stage5</a>;
                } else {
                    __[getElementById](status)[innerHTML] = $;
                }
            });
        }).catch(function() {
            __[getElementById](status)[innerHTML] = $;
        });
    }).catch(function() {
        __[getElementById](status)[innerHTML] = $;
    });
}
window[setTimeout](func4, $_$);
```

Le code se lit plutôt clairement : la partie entre parenthèses du User-Agent est utilisée pour former une clé et un IV, l'IV étant les 16 premiers caractères, et la clé les 16 derniers.

La chaîne « chrome://browser/content/preferences/preferences.xul » nous indique que le navigateur doit être Firefox. Nous allons alors générer un ensemble de User-Agents afin d'effectuer un bruteforce sur le déchiffrement AES, jusqu'à obtenir un ZIP valide. Il est à noter que la clé peut être bruteforcée sans connaissance de l'IV, un mauvais IV n'ayant pour conséquence que l'obtention d'un premier bloc invalide => il est donc toujours possible de reconnaître un fichier ZIP en observant sa fin.

Le bruteforce est implémenté en suivant la documentation de référence concernant le User-Agent des navigateurs Gecko[4].

Le bon User-Agent finit par tomber[5], il s'agit de :

```
Macintosh; Intel Mac OS X 10.6; rv:35.0
```

Le fichier peut donc être correctement déchiffré.

4 https://developer.mozilla.org/en-US/docs/Web/HTTP/Gecko_user_agent_string_reference

5 Dans la solution, cela semble rapide, mais l'auteur s'est arraché les cheveux pendant de longues heures pour trouver le bon User-Agent...

Part 5 : The Transputer

Le contenu de l'archive récupérée est le suivant :

```
$ unzip p5.zip
Archive:  p5.zip
  inflating: input.bin
  inflating: schematic.pdf
```

Le document PDF contient le schéma suivant :

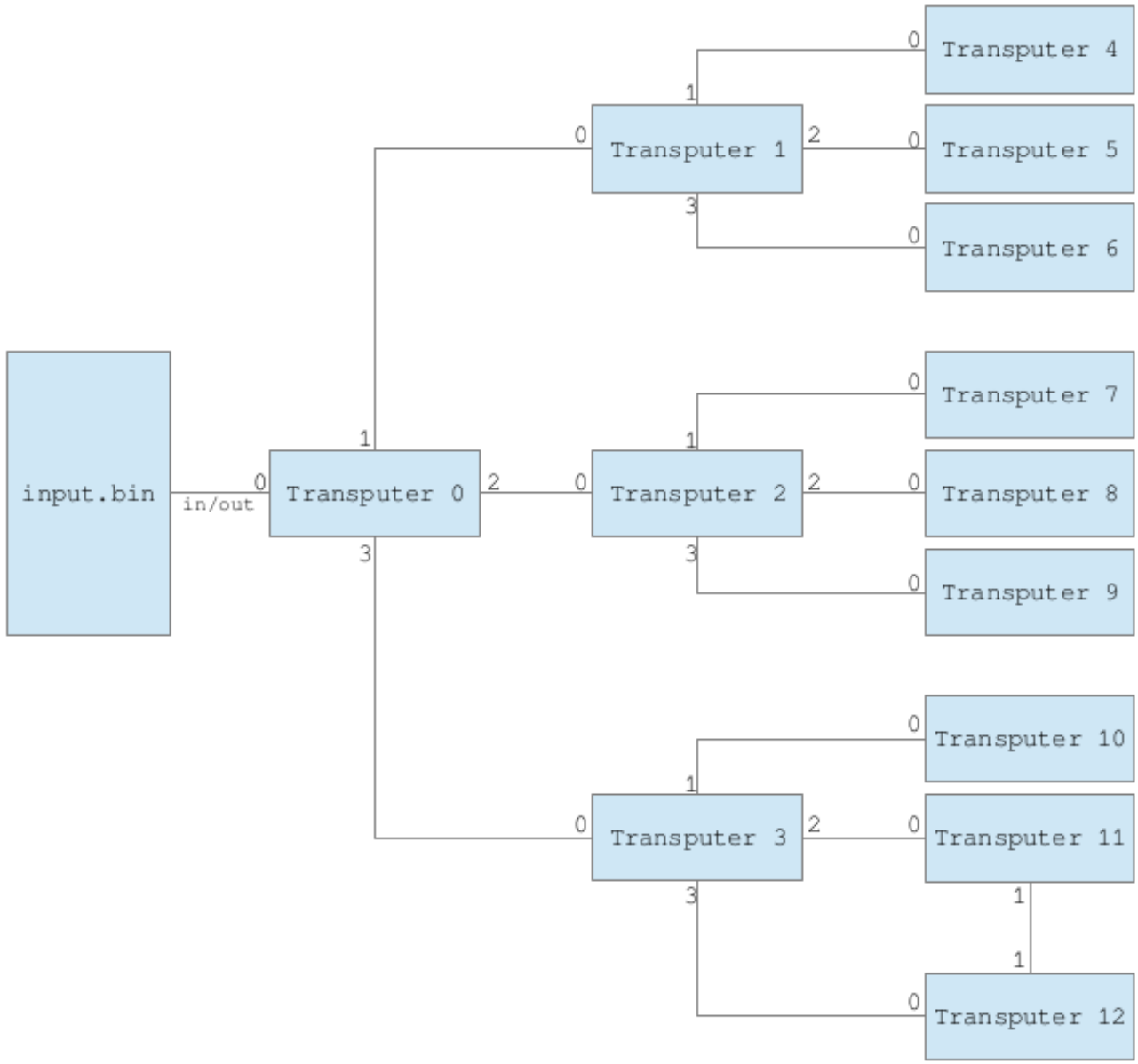


Illustration 12: Schéma d'architecture

Le texte suivant est également présent :

```
SHA256:
a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81 - encrypted
```

```
9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321 - decrypted

Test vector:
key = "*SSTIC-2015*"
data = "1d87c4c4e0ee40383c59447f23798d9fefe74fb82480766e".decode("hex")
decrypt(key, data) == "I love ST20 architecture"
```

Les concepteurs ne sont pas trop vicieux, ils nous ont fournit un vecteur de test pour pouvoir vérifier que l'on a bien compris l'architecture ! La chaîne déchiffrée nous donne également un indice sur les transputers utilisés : il s'agit de ST20.

Fort heureusement, IDA Pro permet de gérer cette architecture. Un coup d'oeil à la page wikipedia Transputer nous indique[6] que lorsque les transputers sont configurés pour démarrer depuis leur lien réseau, le premier octet envoyé correspond à la taille du code qui sera envoyé ensuite.

Le premier octet du fichier « input.bin » étant 0xf8, on en déduit que le code de démarrage du transputer 0 (les transputers seront nommés Tx, 0 ≤ x ≤ 12 dans la suite du document) fait 248 octets. L'aperçu dans IDA Pro nous donne confirmation :

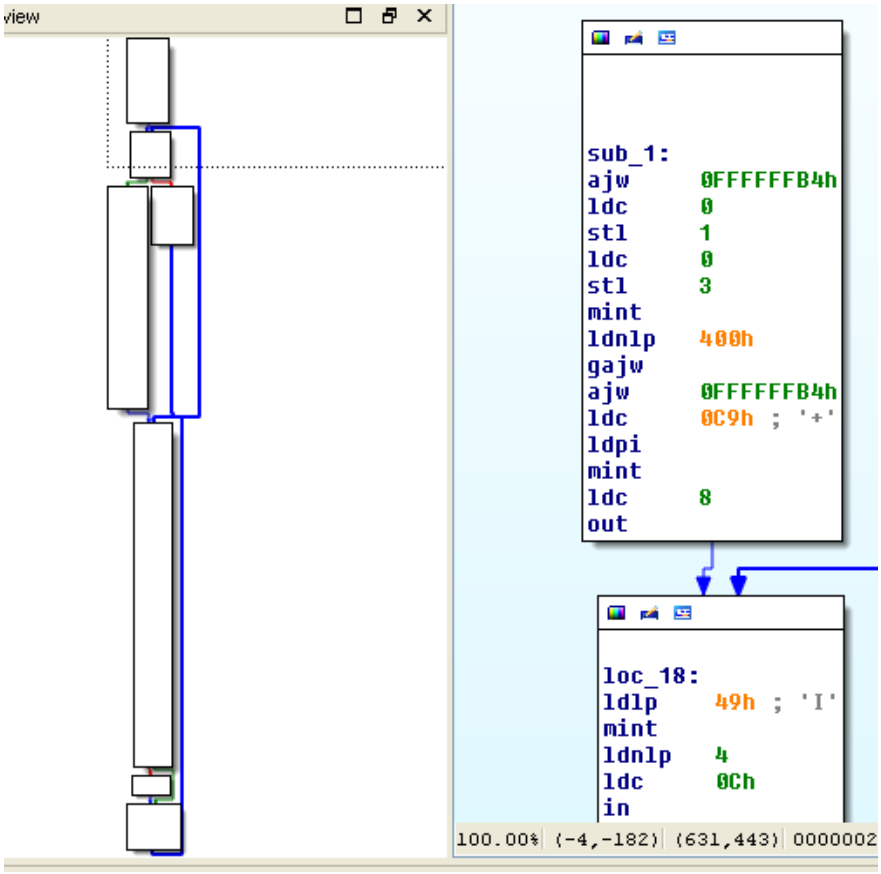


Illustration 13: Un bilboquet

Nous pouvons commencer à étudier le code de T0. Celui-ci effectue les actions suivantes :

- Envoie sur output0 de « Boot ok »

6 http://en.wikipedia.org/wiki/Transputer#Booting

- Entre dans une boucle qui :
    - Lit 3 dword sur input0 (un structure contenant un champ taille (dword1), un champ output (dword2) et un champ offset (dword3))
    - Sort si dword1 vaut 0
    - Lit dword1 octets sur input0
    - Envoie ces dword1 octets dans l'output spécifié par dword2
- Envoie un structure avec taille 0 à toutes les outputs
- Envoie « Code ok » sur output0
- Lit 4 octets sur input0 (« KEY: »)
- Lit 12 octets sur input0 (que des 0xff), stockés dans une variable que l'on nomme « key »
- Envoie « Decrypt » sur output0
- Lit une chaîne précédée par sa taille sur input0 (« congratulations.tar.bz2 »)
- Enfin, entre dans la boucle finale, dont le compteur (« i ») est initialisé à 0 :
    - Lit un octet sur input0 (« c »)
    - Transmet 12 octets de key sur les outputs 1 à 3
    - Lit un octet de chaque input (1 à 3), puis les xor entre eux (« x »)
    - Xor l'octet « c » avec (i+key[i]*2) (« d »)
    - Remplace key[i] par « x »
    - Envoie « d » sur output0

Nous comprenons que la première boucle sert de dispatcher pour transmettre le code des autres transputers. La deuxième boucle correspond, elle, à une partie de l'algorithme de chiffrement.

Les transputers T1, T2 et T3 reçoivent tous le même code :

- Une boucle de dispatch de code vers les 3 transputers auxquels ils sont reliés
- Une deuxième boucle qui :
    - Lit les 12 octets de key de input0
    - Transmet ces 12 octets sur les outputs 1 à 3
    - Lit un octet de chaque input (1 à 3), puis les xor entre eux (« x »)

    - Renvoie « x » sur output0

On comprend alors que l'octet « x » du transputer T0 sera un XOR entre les octets transmis par les transputers T4 à T12.

Il ne reste alors qu'à étudier le code de ces 9 transputers pour comprendre comment sont générés les octets qu'ils envoient.

Chaque transputer peut être vu comme une sorte de LFSR, qui renvoit un octet en sortie pour 12 octets en entrée. Voici en vue macro le fonctionnement de chaque transputer :

- T4 : met à jour une variable interne avec la somme des 12 octets reçus, et renvoie cette variable
- T5 : met à jour une variable interne avec le XOR des 12 octets reçus, et renvoie cette variable
- T6 : initialise une variable interne de 16 bits avec la première clé, puis renvoie le byte de poids faible après plusieurs opérations logiques
- T7 : fait séparément les sommes des 6 premiers et 6 derniers octets reçus, puis renvoie leur XOR
- T8 : maintient 4 tableaux de 12 octets qu'il remplit en tournant avec la nouvelle valeur de la clé, puis renvoie le XOR de la somme des octets de chaque tableau
- T9 : renvoie le XOR des 12 octets reçus, chacun ayant subit un SHR en fonction du compteur de boucle
- T10 : maintient 4 tableaux de 12 octets qu'il remplit en tournant avec la nouvelle valeur de la clé, puis renvoie un octet parmis ces tableaux en fonction de la somme des premiers octets de chaque tableau
- T11/T12 : ces deux transputers communiquent entre eux. T11 transmet à T12 un XOR de 3 octets parmis les 12 reçus (indices 0, 3 et 7). T12 transmet à T11 un XOR de 3 octets parmis les 12 octets de la clé précédente (indices 1, 5 et 9) et chacun renvoie un octet de la clé en fonction de l'octet qu'il a reçu

Pour plus de détails, le lecteur est invité à se référer à « input.bin ».

Une implémentation du fonctionnement de ces 9 transputers est réalisée en Python, et ceux-ci sont exécutés séquentiellement (T11 et T12 pouvant également être gérés de la sorte, leur interaction étant très basique). Il est alors possible de déchiffrer le vecteur de test. Le code est présent en annexe du présent document.

Toutefois, lorsque vient le moment de déchiffrer la véritable payload, le fichier de sortie ne répond pas au format attendu (fichier BZ2). L'auteur comprend alors que la tâche n'est pas terminée, et que la clé fournie (0xff * 12) n'est effectivement pas la bonne.

L'algorithme identifié pour effectuer le déchiffrement dans T0 nous indique que les 12

premiers octets du fichier déchiffré ne dépendent que de la clé d'entrée (pour rappel, c=x^(i+key[i]*2)). L'entête d'un fichier BZ2 est constitué de[7] :

- 2 octets fixes : « BZ »
- 1 octet de version : « h » pour BZ2
- 1 octet de blocksize : « 1 » à « 9 »
- 6 octets fixes : « 1AY&SY »

Le blocksize par défaut de la commande bzip2 étant « 9 », considérons que c'est ici aussi le cas. D'après la formule pour le calcul des octets du fichier déchiffré, nous pouvons rapidement voir que plusieurs valeurs d'entrée peuvent donner une même valeur de sortie (à cause de la multiplication par 2). Pour chacun des octets de sortie fixes vus précédemment, nous pouvons calculer les octets d'entrée en inversant la formule :

```
>>> out="BZh91AY&SY"
>>> ciph="fef350dc81bc972789ac".decode("hex")
>>> for i in xrange(len(out)):
...   v=ord(out[i])^ord(ciph[i])
...   v1=((v-i)/2)&0xff
...   v2=(v1+0x80)&0xff
...   print "%02x %02x" % (v1,v2)
...
5e de
54 d4
1b 9b
71 f1
56 d6
7c fc
64 e4
fd 7d
69 e9
76 f6
```

Nous disposons alors de 1024 combinaisons possibles pour les 10 premiers octets de la clé. Les 2 restants étant totalement inconnus, cela multiplie les possibilités par 65536.

Le nombre de clés à bruteforcer étant élevé, et Python n'étant pas reconnu pour sa grande rapidité, l'algorithme de bruteforce doit être optimisé de façon à ce que les mauvaises clés soient éliminées très rapidement.

Une observation empirique de fichiers BZ2 a permis de déterminer une condition de sortie ne nécessitant le calcul que de 9 octets par les transputers. En effet, cela suffit à obtenir 21 octets du clair.

Nous considérons alors que la clé fera partie des clés valides si les 3 derniers octets de ces 21 valent 0xff.

7 http://en.wikipedia.org/wiki/Bzip2

Même en ne faisant calculer que peu d'octets par les transputers, ce bruteforce est lent, et quelques tests montrent qu'il prendrait environ **3 heures** sur un seul coeur. L'auteur a donc choisi de mettre à profit son i7 et réparti l'espace du bruteforce sur 16 processus Python en parallèle.

```
for k in `seq 0 15`; do python bf20.py $k & done
```

Ceci a permis de réduire la durée totale du bruteforce à **40 minutes**, la bonne clé étant tombée en environ 20 minutes.

L'auteur a par la suite pu constater que l'utilisation de pypy à la place de python faisait tomber le temps de bruteforce total à un peu moins de **7 minutes**.

La clé finale est donc : 5ed49b7156fce47de976dac5.

Nous pouvons alors déchiffrer l'ensemble du chiffré :

```
$ sha256sum decrypted
9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321  decrypted

$ tar tvjf decrypted
-rw-r--r-- test/test    252569 2015-03-23 10:34 congratulations.jpg
```

Ca sent bon !

Part 6 : The Mega Rage

Ouvrons l'image …

Illustration 14: Un premier troll

A ce moment précis, l'auteur se retrouve dans la situation dite du Lapin de Pâques le lundi soir.

Illustration 15: Des lapins de Pâques discutant à l'issue du week-end de Pâques

Vérifions que la solution ne soit pas une simple concaténation de fichiers :

```
# hachoir-subfile congratulations.jpg
[+] Start search on 252569 bytes (246.6 KB)

[+] File at 0 size=55248 (54.0 KB): JPEG picture
[+] File at 55248: bzip2 archive

[+] End of search -- offset=252569 (246.6 KB)
```

Bingo ! Extrayons ce fichier compressé et voyons son contenu.

```
$ tar tvjf xxx.bz2
-rw-r--r-- test/test    197557 2015-03-23 10:34 congratulations.png
```

Ouvrons l'image …

Illustration 16: Un second troll ...

C'est à peu près le moment ou l'auteur commence à rager. Voyons ce que donne l'outil pngcheck sur cette image png …

```
$ pngcheck congratulations.png
congratulations.png  illegal reserved-bit-set chunk sTic
ERROR: congratulations.png
```

Il y a donc des chunks de type « sTic » dans l'image … Il est possible d'extraire leur contenu à l'aide d'un script python.

```
#!/usr/bin/python
import struct

data=open("congratulations.png","rb").read()
data=data[8:]
out=""
while len(data)>0:
    x=struct.unpack(">2L",data[:8])
    ll=x[0]
    ty=x[1]
    data = data[8:]
    tmp=data[:ll]
    data = data[ll:]
    (crc,) = struct.unpack("<L",data[:4])
    data=data[4:]
    if ty == 0x73546963: #sTic
        out += tmp
    print "%x (%d bytes) %x" % (ty,ll,crc)
open("extract.bin","wb").write(out)
```

Le fichier de sortie ne correspond pas à un type de fichier connu. Une analyse statistique des différents octets qu'il contient nous indique que les données sont compressées ou chiffrées. Vu qu'aucune clé n'est fournie, la compression semble être la bonne voie.

En ajoutant un appel à zlib.decompress() à notre script précédent, nous obtenons en sortie un nouveau fichier compressé en bz2.

```
# tar tvjf extract.bin
-rw-r--r-- test/test    904520 2015-03-23 10:34 congratulations.tiff
```

Ouvrons l'image, on ne sait jamais …

Illustration 17: Encore un troll

…

A ce stade, l'auteur a l'impression que cette inception de stéganographie ne s'arrêtera jamais …

Voici l'ensemble des hashs de cette image :

```
$ md5sum congratulations.tiff
f35daecaa038124523e46660460430da  congratulations.tiff
$ sha1sum congratulations.tiff
c31197c43d210482ed85e6d369a8e62b13f33bae  congratulations.tiff
$ sha256sum congratulations.tiff
122fe4291cfe290b742b4d464fb6dbc745448e7659113c2cd2786921ecd9ebb5
congratulations.tiff
```

Détail curieux, cette image contenant 2 petits diables a la chaîne « 666 » dans son MD5. Ceci nous prouve que des informations pourraient être cachées dans les bits de poids faible de certains pixels.

Examinons cette image à l'aide de l'outil privé megasteganotequilaboomboom.py :

```
$ python megasteganotequilaboomboom.py congratulations.tiff
$ eog x.png
```

Voici l'image produite :


Illustration 18: TEQUILA BOOM BOOM

Comme nous l'avions soupçonné, il semble y avoir de l'information cachée dans les bits de poids faibles des pixels de l'image. Plus particulièrement, l'outil nous montre que les informations seraient cachées dans les valeurs codant le rouge et le vert.

Nous extrayons alors ces bits de poids faible pour les composantes rouge et verte, et nous nous trouvons face à un nouveau fichier compressé en BZ2.

```
$ tar tvjf outz.bin

bzip2: (stdin): trailing garbage after EOF ignored
-rw-r--r-- test/test     28755 2015-03-23 10:34 congratulations.gif
```

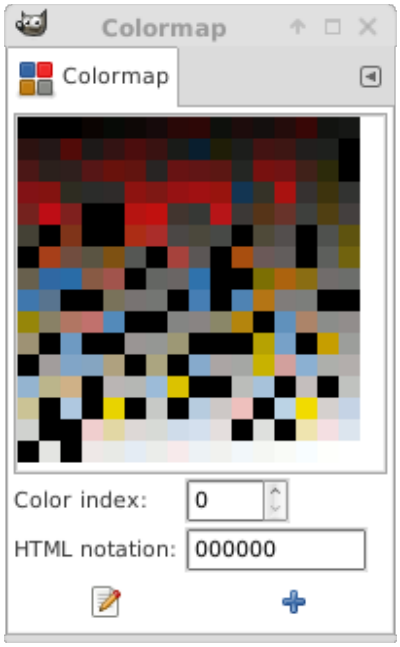
Vérifions le contenu de l'image …

Illustration 19: Le troll final

… Toujours pas … Le vice a été poussé loin par le(s) concepteur(s) de cette année.

Une façon de cacher de l'information dans une image au format GIF est de jouer avec la palette de couleur.

Voici la palette de l'image telle qu'observée dans Gimp :



La palette semble contenir de trop nombreuses nuances de noir. Les bords de l'image étant noirs, il est possible que de l'information s'y cache.

Si nous essayons de passer successivement les différentes nuances de noir vers du blanc, l'image suivante finit par apparaître :

Illustration 20: Le salut

ENFIN !

Remerciements

Merci aux concepteurs de ce challenge riche et varié. Même s'il m'a beaucoup fait rager …
La partie ST20 était vraiment sympa ;)

Annexes

Déchiffrement du vecteur de test :

```
#!/usr/bin/python

import sys

key = "*SSTIC-2015*"
kb = [ord(k) for k in key]

data = "1d87c4c4e0ee40383c59447f23798d9fefe74fb82480766e".decode("hex")

t4 = 0
def trans4():
    global t4
    for k in kb:
        t4 = (t4+k)&0xff
    return t4

t5 = 0
def trans5():
    global t5
    for k in kb:
        t5 = (t5^k)&0xff
    return t5

t6 = 0
def trans6_init():
    global t6
    for k in kb:
        t6 = (t6+k)&0xffff

trans6_init()

def trans6():
    global t6
    v1 = (t6&0x8000)>>0xf
    v2 = (t6&0x4000)>>0xe
    v1 = (v1^v2)&0xffff
    v3 = (t6<<1)&0xffff
    v1 = (v1^v3)&0xffff
    t6 = v1
    return v1&0xff
```

```
def trans1():
    return trans4()^trans5()^trans6()

def trans7():
    v1=0
    v2=0
    for i in xrange(6):
        v1 += kb[i]
        v2 += kb[i+6]
    v3= (v1^v2)&0xff
    return v3

t8 = [0]*12*4
t8_ptr4 = 0

def trans8():
    global t8, t8_ptr4
    i=0
    for k in kb:
        t8[12*t8_ptr4+i] = k
        i+=1
    t8_ptr4 = (t8_ptr4+1)%4
    v2=0
    for i in xrange(4):
        v=0
        for j in xrange(12):
            v = (v+t8[12*i+j])&0xff
        v2=v2^v
    return v2

def trans9():
    v=0
    i=0
    for k in kb:
        mm = i&7
        v = (v^(k<<mm))&0xff
        i+=1
    return v

def trans2():
    return trans7()^trans8()^trans9()

t10 = [0]*12*4
t10_ptr4 = 0

def trans10():
    global t10, t10_ptr4
    i=0
    for k in kb:
        t10[12*t10_ptr4+i] = k
```

```
        i+=1
    t10_ptr4 = (t10_ptr4+1)%4
    v2=0
    for i in xrange(4):
        v2 = (v2+t10[12*i])&0xff
    mm = (v2&3)*12
    xx = (v2>>4)%12
    return t10[mm+xx]

t11out=0
def trans11():
    global t11out
    v=kb[0]^kb[3]^kb[7]
    t11out=v
    v2 = trans12_1()

    return kb[v2%0xc]

t12 = [0]*12
def trans12_1():
    global t12
    return t12[1]^t12[5]^t12[9]

def trans12():
    global t12,t11out
    i=0
    for k in kb:
        t12[i] = k
        i += 1
    return kb[t11out%0xc]

def trans3():
    v1=trans10()
    v2=trans11()
    v3=trans12()
    return v1^v2^v3

i=0
dede=""
for u in data:
    v1 = trans1()
    v2 = trans2()
    v3 = trans3()
    v4 = v1^v2^v3
    x=((i+kb[i]*2)^ord(u))&0xff
    kb[i] = v4
    dede += chr(x)
    i = (i+1)%0xc
open("decrypted","wb").write(dede)
```

Code permettant le bruteforce de la clé :

```
#!/usr/bin/python

# Bruteforce goret style

import sys

valuz = int(sys.argv[1])*16

fff=open("bf%d.log"%valuz,"wb")
fff.write("Start %d\n"%valuz)
fff.close()

data = open("encrypted","rb").read(0x20)

t4 = 0
def trans4():
    global t4
    for k in kb:
        t4 = (t4+k)&0xff
    return t4

t5 = 0
def trans5():
    global t5
    for k in kb:
        t5 = (t5^k)&0xff
    return t5

t6 = 0
def trans6_init():
    global t6
    for k in kb:
        t6 = (t6+k)&0xffff

def trans6():
    global t6
    v1 = (t6&0x8000)>>0xf
    v2 = (t6&0x4000)>>0xe
    v1 = (v1^v2)&0xffff
    v3 = (t6<<1)&0xffff
    v1 = (v1^v3)&0xffff
    t6 = v1
    return v1&0xff

def trans1():
    return trans4()^trans5()^trans6()


def trans7():
```

```
    v1=0
    v2=0
    for i in xrange(6):
        v1 += kb[i]
        v2 += kb[i+6]
    v3= (v1^v2)&0xff
    return v3

t8 = [0]*12*4
t8_ptr4 = 0

def trans8():
    global t8, t8_ptr4
    i=0
    for k in kb:
        t8[12*t8_ptr4+i] = k
        i+=1
    t8_ptr4 = (t8_ptr4+1)%4
    v2=0
    for i in xrange(4):
        v=0
        for j in xrange(12):
            v = (v+t8[12*i+j])&0xff
        v2=v2^v
    return v2

def trans9():
    v=0
    i=0
    for k in kb:
        mm = i&7
        v = (v^(k<<mm))&0xff
        i+=1
    return v

def trans2():
    return trans7()^trans8()^trans9()

t10 = [0]*12*4
t10_ptr4 = 0

def trans10():
    global t10, t10_ptr4
    i=0
    for k in kb:
        t10[12*t10_ptr4+i] = k
        i+=1
    t10_ptr4 = (t10_ptr4+1)%4
    v2=0
    for i in xrange(4):
        v2 = (v2+t10[12*i])&0xff
```

```
    mm = (v2&3)*12
    xx = (v2>>4)%12
    return t10[mm+xx]

t11out=0
def trans11():
    global t11out
    v=kb[0]^kb[3]^kb[7]
    t11out=v
    v2 = trans12_1()

    return kb[v2%0xc]

t12 = [0]*12
def trans12_1():
    global t12
    return t12[1]^t12[5]^t12[9]

def trans12():
    global t12,t11out
    i=0
    for k in kb:
        t12[i] = k
        i += 1
    return kb[t11out%0xc]

def trans3():
    v1=trans10()
    v2=trans11()
    v3=trans12()
    return v1^v2^v3

for k0 in (94,222):
    for k1 in (84,212):
        for k2 in (27,155):
            for k3 in (113,241):
                for k4 in (86,214):
                    for k5 in (124,252):
                        for k6 in (100,228):
                            for k7 in (125,253):
                                for k8 in (105,233):
                                    for k9 in (118,246):
                                        for k10 in xrange(valuz,valuz+16,1):
                                            for k11 in xrange(256):
                                                kb =
[k0,k1,k2,k3,k4,k5,k6,k7,k8,k9,k10,k11]
                                                kbo =
[k0,k1,k2,k3,k4,k5,k6,k7,k8,k9,k10,k11]
                                                t4=0
                                                t5=0
                                                t6=0
```

```
                                                trans6_init()
                                                t8 = [0]*12*4
                                                t8_ptr4 = 0
                                                t10 = [0]*12*4
                                                t10_ptr4 = 0
                                                t11out = 0
                                                t12 = [0]*12
                                                i=0
                                                for zzz in xrange(9):
                                                    v1 = trans1()
                                                    v2 = trans2()
                                                    v3 = trans3()
                                                    v4 = v1^v2^v3
                                                    kb[i] = v4
                                                    i = (i+1)%0xc
                                                if ((i-
1+v4*2)&0xff)^ord(data[0x14]) == 0xff and ((i-2+kb[i-2]*2)&0xff)^ord(data[0x13])
== 0xff and ((i-3+kb[i-3]*2)&0xff)^ord(data[0x12]) == 0xff:
                                                    print "FOUND"
                                                    print kbo
                                            fff=open("bf%d.log"%valuz,"ab")
                                            fff.write(str(kbo))
                                            fff.write("\n")
                                            fff.close()
```