Solution Challenge SSTIC 2009 – Fabien Perigaud

1. Premier contact

Bon allez, on va tenter de résoudre ce challenge, qui sait, j'aurai peut-être une nouvelle brosse à dents !

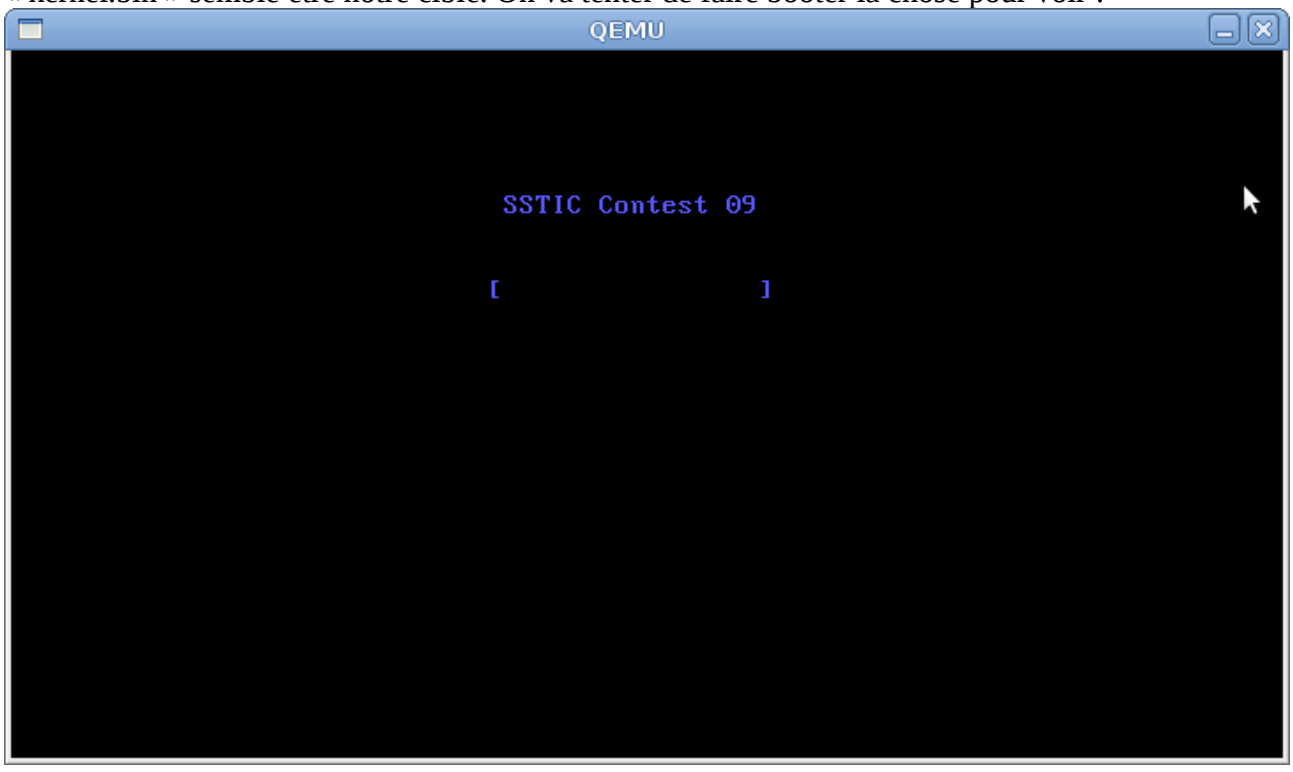
On télécharge, puis on regarde un peu le fichier :

```
fab@sawfish-laptop:~/sstic$ file ChallengeSSTIC2009
ChallengeSSTIC2009: x86 boot sector; GRand Unified Bootloader,
stage1 version 0x3, 1st sector stage2 0x62, code offset 0x48

fab@sawfish-laptop:~/sstic$ ll -h ChallengeSSTIC2009
-rw-r--r-- 1 fab fab 1,5M avr 20 10:55 ChallengeSSTIC2009
```

D'après la taille et le type, on dirait bien une disquette de boot. On va tenter de la monter pour voir.

```
fab@sawfish-laptop:~/sstic$ mount -o loop ChallengeSSTIC2009 /mnt/
sstic/

fab@sawfish-laptop:~/sstic$ ll /mnt/sstic/
total 1188
drwxr-xr-x 4 root root    1024 mar 19 16:15 .
drwxr-xr-x 7 root root    4096 avr 27 10:59 ..
drwxr-xr-x 2 root root    1024 mar 19 16:15 grub
-rwxr-xr-x 1 root root 1191519 mar 19 16:15 kernel.bin
drwx------ 2 root root   12288 mar 19 16:15 lost+found
```

« kernel.bin » semble être notre cible. On va tenter de faire booter la chose pour voir !



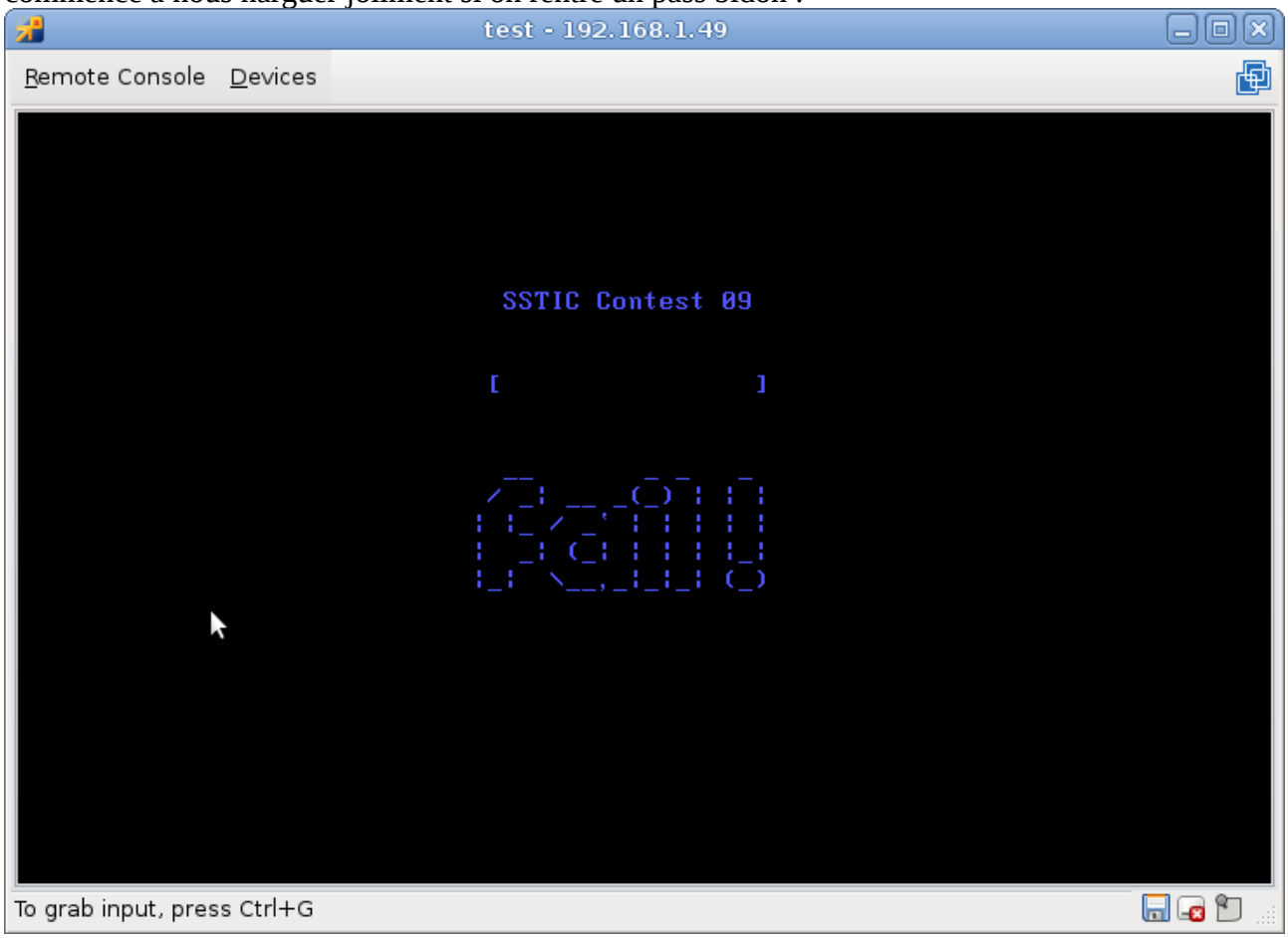
C'est bien ça, mais si on tente un password, surprise :

```
qemu: fatal: Trying to execute code outside RAM or ROM at
0xe000200c

EAX=0026d9b5 EBX=0000000d ECX=64646483 EDX=d4000000
ESI=00000000 EDI=00000000 EBP=4600326a ESP=00326acd
EIP=e000200c EFL=00000002 [-------] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300
CS =0008 00000000 ffffffff 00cf9b00
SS =0010 00000000 ffffffff 00cf9300
DS =0010 00000000 ffffffff 00cf9300
FS =0010 00000000 ffffffff 00cf9300
GS =0010 00000000 ffffffff 00cf9300
LDT=0000 00000000 0000ffff 00008000
TR =0018 00322aa8 00002089 00008932
GDT=     00322288 0000001f
IDT=     003222a8 000007ff
CR0=60000011 CR2=00000000 CR3=00000000 CR4=000001d5
CCS=00000001 CCD=0026d9b5 CCO=LOGICL
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000
XMM01=00000000000000000000000000000000
XMM02=00000000000000000000000000000000
XMM03=00000000000000000000000000000000
XMM04=00000000000000000000000000000000
XMM05=00000000000000000000000000000000
XMM06=00000000000000000000000000000000
XMM07=00000000000000000000000000000000
Abandon
```

Merci qemu :(

2. Looking for tools !

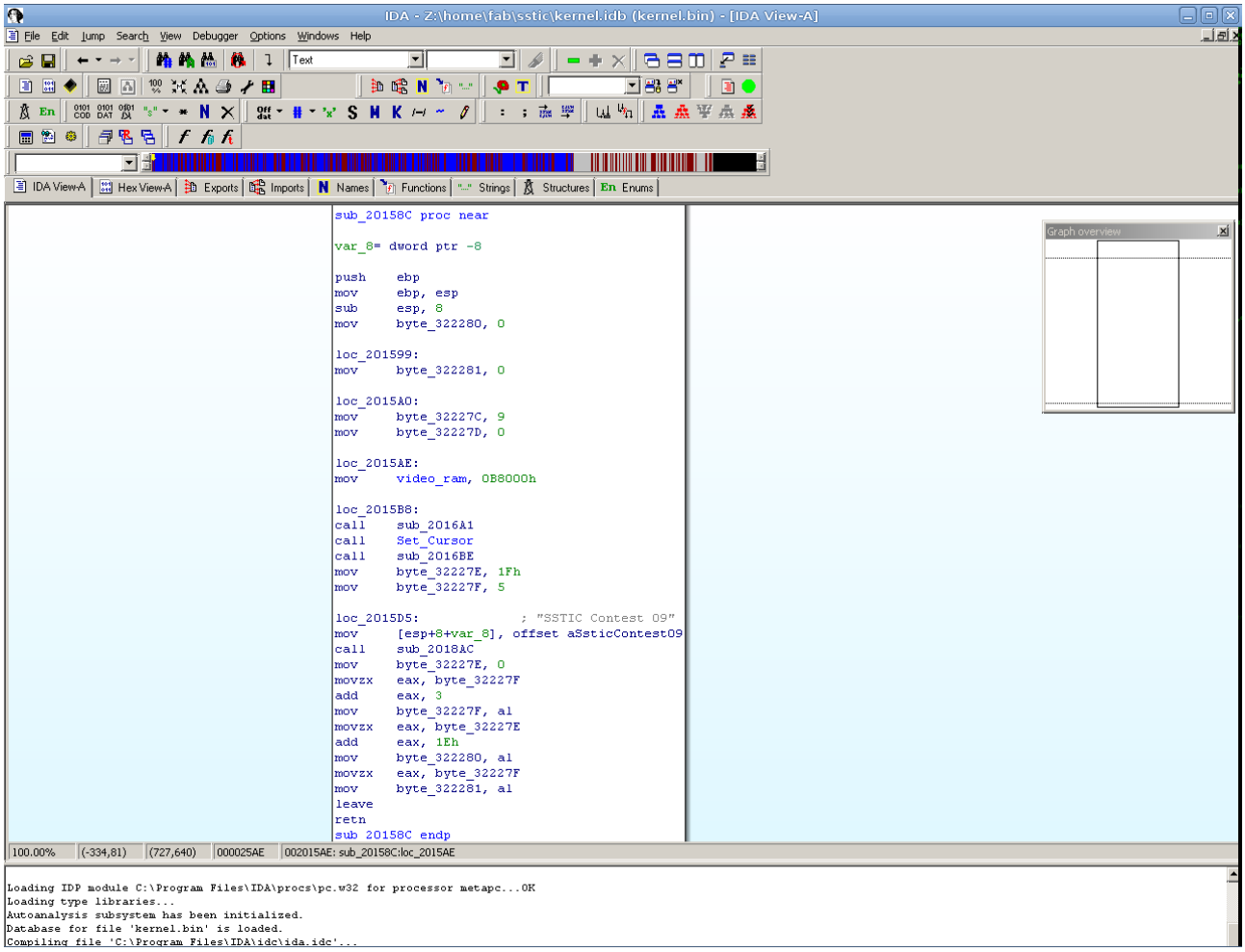
Bon qemu, n'est pas d'accord, essayons VMWare. Là, ça marche déjà mieux, et le challenge commence à nous narguer joliment si on rentre un pass bidon :



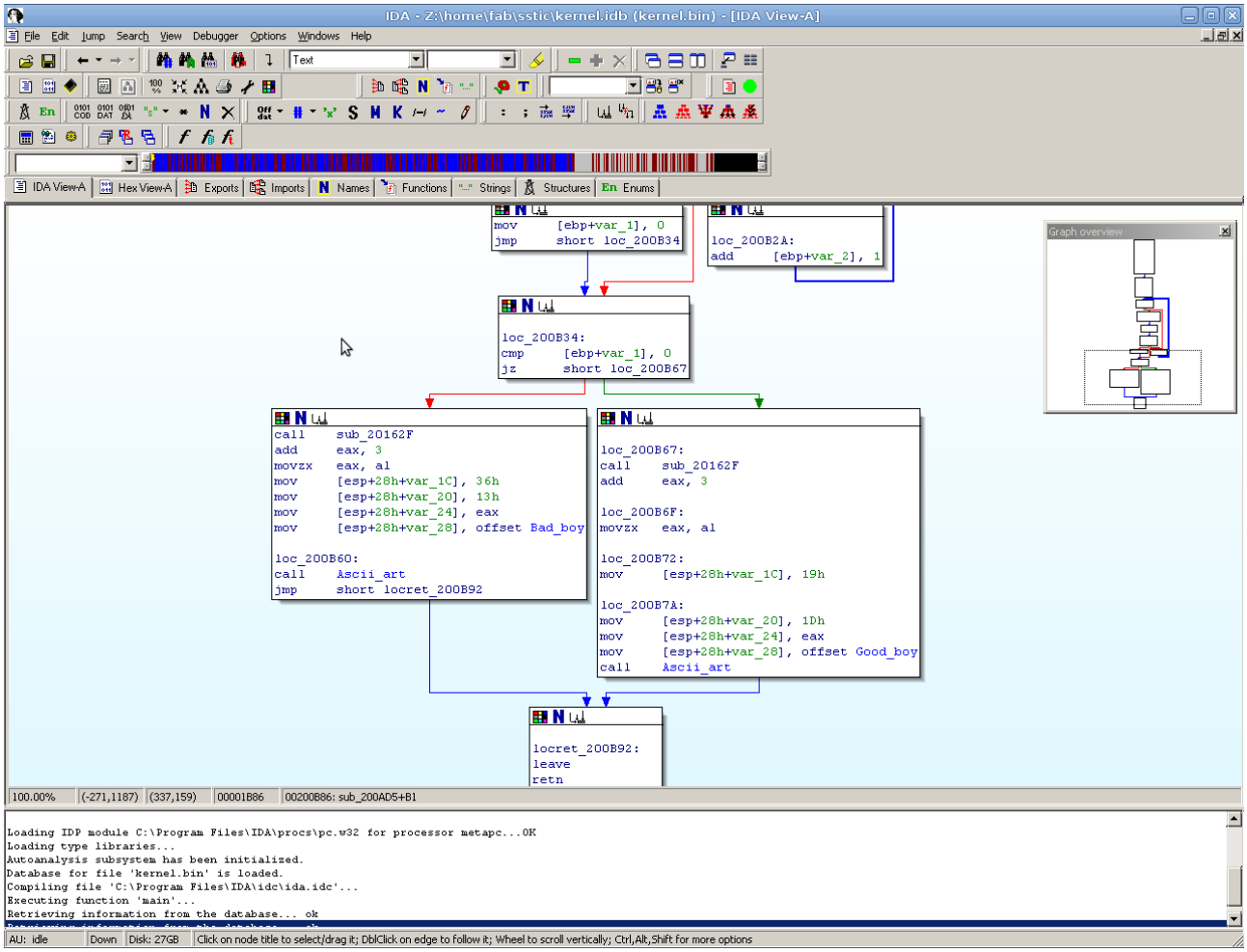
Après quelques recherches j'ai trouvé comment attacher un gdb à VMWare, on peut donc commencer les choses sérieuses.

3. Point de départ

Une fois le noyau désassemblé dans IDA, j'ai commencé par regarder les strings.
Première chose qui attire l'oeil, une énigme dont la résolution se fait par modulos.
On tente « 785 » comme passphrase, mais bien sûr, ça ne marche pas :)
L'autre string est « SSTIC Contest 09 », qui nous amène à la fonction suivante :



L'objectif étant de trouver la partie du code affichant le « fail! » en ascii-art, on repère la variable qui reçoit l'adresse de la RAM vidéo, puis de là on regarde les fonctions qui y font appel.
On tombe alors rapidement sur la fonction qui va nous indiquer si l'on est un good boy ou un bad boy :



En xorant avec la clé « 0xA5CC1A27 » les octets situés aux 2 offsets passés en paramètre de la fonction sub_200A34 (ici renommée Ascii_art), on retrouve notre « fail! », mais aussi un « excellent! ». On est donc sur la bonne voie :)
Un peu plus haut, on constate que pour afficher le good_boy, les 12 entiers situés en 0x2E11C0 sont xorés avec la clé « 0xA5CC1A27 » puis comparés à des valeurs situées à une adresse passée en argument de la fonction.
On va pouvoir sortir le débugueur pour en savoir plus …

4. Analyse dynamique

Si l'on pose un breakpoint au niveau de la comparaison, on constate que les données comparées sont situées en 0x70000. Pour comprendre ce qui leur arrive, posons un watchpoint en 0x70000.
On constate alors 4 modifications, les deux premières positionnant les deux « short » qui composent l'entier à une valeur calculée à partir de valeurs en 0x60000, et les secondes consistant à inverser les octets de chaque « short » avant de faire un « not ».
Si l'on observe d'où viennent les données en 0x60000 en positionnant un autre watchpoint, on constate que la première modification de la zone consiste en la copie du mot de passe que l'on vient d'entrer. Toutefois, au moment de la génération des données en 0x70000, le mot de passe a subit quelques modifications …
On peut d'ores et déjà imaginer que le mécanisme est le suivant :

<Mot de passe> ----fonction1---> valeurs en 0x60000 ----fonction2---> valeurs en 0x70000

Ayant constaté que la fonction2 semble moins longue que la fonction1, et en bon flemmard, j'ai commencé par essayer de remonter des valeurs en 0x2E11C0 à ce qu'on devrait avoir en 0x60000 si

le mot de passe entré est le bon.

5. **Reversing Part 2**

On commence par prendre les 12 entiers en 0x2E11C0, à les xorer avec « 0xA5CC1A27 », puis un NOT, et enfin inverser les octets des deux « short » composant un entier, ce qui se résume à :

```
int main(int argc, char *argv[]) {
     int fd=open(argv[1], O_RDONLY);
     int p;
     int q;
     while(read(fd, &p, 4)==4) {
          p=p^0xA5CC1A27;
          q = ((p&0xFF)<<8) | ((p&0xFF00)>>8) |
((p&0xFF000000)>>8) | ((p&0x00FF0000)<<8);
          printf("%08x ", ~q);
     }
     printf("\n");
     close(fd);
     return 0;
}
```

On a alors notre première base de travail. Pour plus de commodité par la suite, j'adopterai la notation suivante pour les zones 0x60000 et 0x70000 :

0x60000 : A1A2 A3A4 A5A6 A7A8

0x70000 : C1C2 C3C4 C5C6 C7C8
D1D2 D3D4 D5D6 D7D8
E1E2 E3E4 E5E6 E7E8

Toujours en posant des watchpoints sur les entiers en 0x70000, on constate que les modifications sont effectuées soit par des fonctions en mode réel, qui sont des modulos, soit par des fonctions en mode protégé, qui sont une suite de multiplications et de décalages, qui au final sont peut être aussi des modulos, va savoir ! :)
Ces fonctions utilisent les valeurs en 0x60000 pour générer celles en 0x70000. On se retrouve donc pour chaque « short » en 0x60000 avec un ensemble de 3 équations nous donnant la solution.

Par exemple, pour A6, on a :
C6 = A6 % 0xec6 = 0x7b0
D6 = A6 % 0x1189 = 0x22a
E6 = A6 % 0x125 = 0x9c

Ce qui nous donne A6 = 0x253C ! Tiens tiens, ça rappelle le problème des pirates vu plus haut ;)

Au final, la résolution de toutes les équations nous indique que les valeurs suivantes doivent être en 0x60000 pour générer le bon ensemble de 12 octets en 0x70000 :

```
0x60000: 0x9e8db86a 0xce6d6c69 0xbcc7253c 0x196cd5f6
```

6. Reversing Part 1

Flemmard ou pas, il faut bien maintenant se coller à la fonction1 :(
Les watchpoints ont indiqué que pour chaque entier en 0x60000, 19 passes de fonctions étaient effectuées. J'ai commencé par tracer les modifications en 0x60000 pour récupérer les adresses des fonctions agissant dessus. 76 fonctions … ça fait beaucoup !

L'étude de la première fonction, donc celle qui effectue la première transformation en 0x60000, montre que l'opération est la suivante :

```
   0:     66 ba 01 00 00 00        mov    edx,0x1
   6:     67 66 8b 1c 96           addr32 mov ebx,DWORD PTR
[esi+edx*4]
   b:     66 4a                    dec    edx
   d:     66 89 d0                 mov    eax,edx
  10:     66 83 e0 03              and    eax,0x3
  14:     66 31 e8                 xor    eax,ebp
  17:     64 67 66 8b 04 86        addr32 mov eax,DWORD PTR fs:
[esi+eax*4]
  1d:     66 31 c8                 xor    eax,ecx
  20:     66 57                    push   edi
  22:     66 51                    push   ecx
  24:     66 53                    push   ebx
  26:     66 31 df                 xor    edi,ebx
  29:     66 01 f8                 add    eax,edi
  2c:     66 c1 eb 03              shr    ebx,0x3
  30:     66 c1 e1 04              shl    ecx,0x4
  34:     66 31 cb                 xor    ebx,ecx
  37:     66 89 df                 mov    edi,ebx
  3a:     67 66 8b 1c 24           addr32 mov ebx,DWORD PTR [esp]
  3f:     67 66 8b 4c 24 04        addr32 mov ecx,DWORD PTR
[esp+0x4]
  45:     66 c1 e9 05              shr    ecx,0x5
  49:     66 c1 e3 02              shl    ebx,0x2
  4d:     66 31 cb                 xor    ebx,ecx
  50:     66 01 df                 add    edi,ebx
  53:     66 31 f8                 xor    eax,edi
  56:     66 5b                    pop    ebx
  58:     66 59                    pop    ecx
  5a:     66 5f                    pop    edi
  5c:     66 89 c1                 mov    ecx,eax
  5f:     67 66 8b 04 96           addr32 mov eax,DWORD PTR
[esi+edx*4]
  64:     66 01 c1                 add    ecx,eax
  67:     67 66 89 0c 96           addr32 mov DWORD PTR
[esi+edx*4],ecx
  6c:     cf                       iret
```

Ce qui donne :
ebx=(0x60000+4)*
eax=(0x80000+4)*
ecx=(0x60000+C)*

cste=0x7f434625

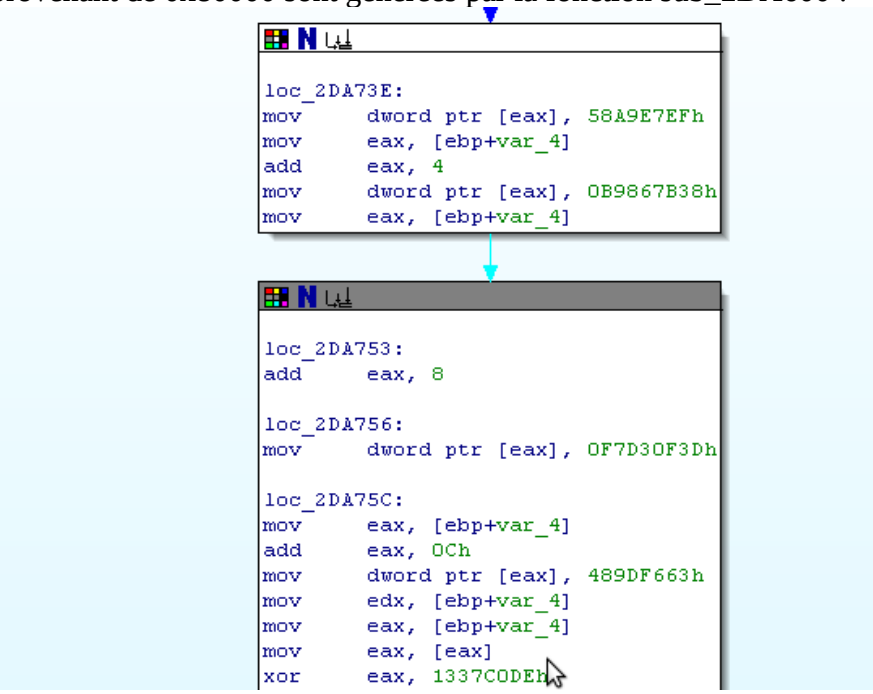
eax=eax^ecx
edi=cste^ebx
eax+=edi
ebx=ebx>>3
ecx=ecx<<4
edi=ebx^ecx
reinit(ebx,ecx)
ecx=ecx>>5
ebx=ebx<<2
ebx=ebx^ecx
edi+=ebx
eax=eax^edi
**(0x60000)=*(0x60000)+eax*

Soit :
**(0x60000)=*(0x60000) + ((cste^*(0x60004) + *(0x80004)^*(0x6000c)) ^ ((*(0x60004)>>3 ^ *(0x6000c)<<4) + (*(0x6000c)>>5 ^ *(0x60004)<<2)))*

On constate alors 2 choses qui viennent prendre part au calcul, en plus des valeurs en 0x60000 :

- une valeur constante provenant de la zone 0x80000
- une autre constante

Les constantes provenant de 0x80000 sont générées par la fonction sub_2DA600 :



Il s'agit de constantes xorées avec « 0x1337C0DE ».
On a donc en 0x80000 les constantes suivantes :

```
0x80000: 0x4b9e2731 0xaab1bbe6 0xe4e4cfe3 0x5baa36bd
```

L'étude d'autres fonctions parmi les 76 nous donne une bonne surprise : le mécanisme est le même :)

On peut sortir de cette étude la formule suivante pour le calcul des entiers en 0x60000 :

A(i)=A(i) + ((A((i+1)%4)^constante) + (B(ind) ^ A((i-1)%4))) ^ (((A((i-1)%4)>>5)^(A((i+1)%4)<<2)) + ((A((i-1)%4)<<4)^(A((i+1)%)>>3)))
si l'on note A le tableau des entiers en 0x60000 et B celui des entiers en 0x80000.

```
sub_2C5F9E      proc near               ; DATA XREF: LOA

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     dword ptr [eax+0Ch], 0DFFB67A8h
                mov     eax, [ebp+arg_0]
                mov     eax, [eax+0Ch]
                shr     eax, 2
                mov     edx, eax
                and     edx, 3
                mov     eax, [ebp+arg_0]
                mov     [eax+14h], edx
                pop     ebp
                retn
sub_2C5F9E      endp
```

On peut constater que « ind », l'indice pour la valeur à utiliser en 0x80000, est dérivé de la constante de la façon suivante :
ind = ((constante>>2)&3)^i

```
mov     eax, [ebp+arg_0]
mov     eax, [eax+14h]
xor     eax, 1

loc_2535CC:
shl     eax, 2
mov     edx, eax
mov     eax, [ebp+var_4]
lea     eax, [edx+eax]
```

Exemple pour i=1

On peut donc commencer à écrire un petit programme effectuant ces opérations à l'envers, à partir de ce qu'on a pu trouver lors de la résolution de la partie 2, et en utilisant la formule :

Old(A(i)) = A(i) - ((A((i+1)%4)^constante) + (B(((constante>>2)&3)^i)^A((i-1)%4))) ^ (((A((i-1)%4)>>5)^(A((i+1)%4)<<2)) + ((A((i-1)%4)<<4)^(A((i+1)%4)>>3)))

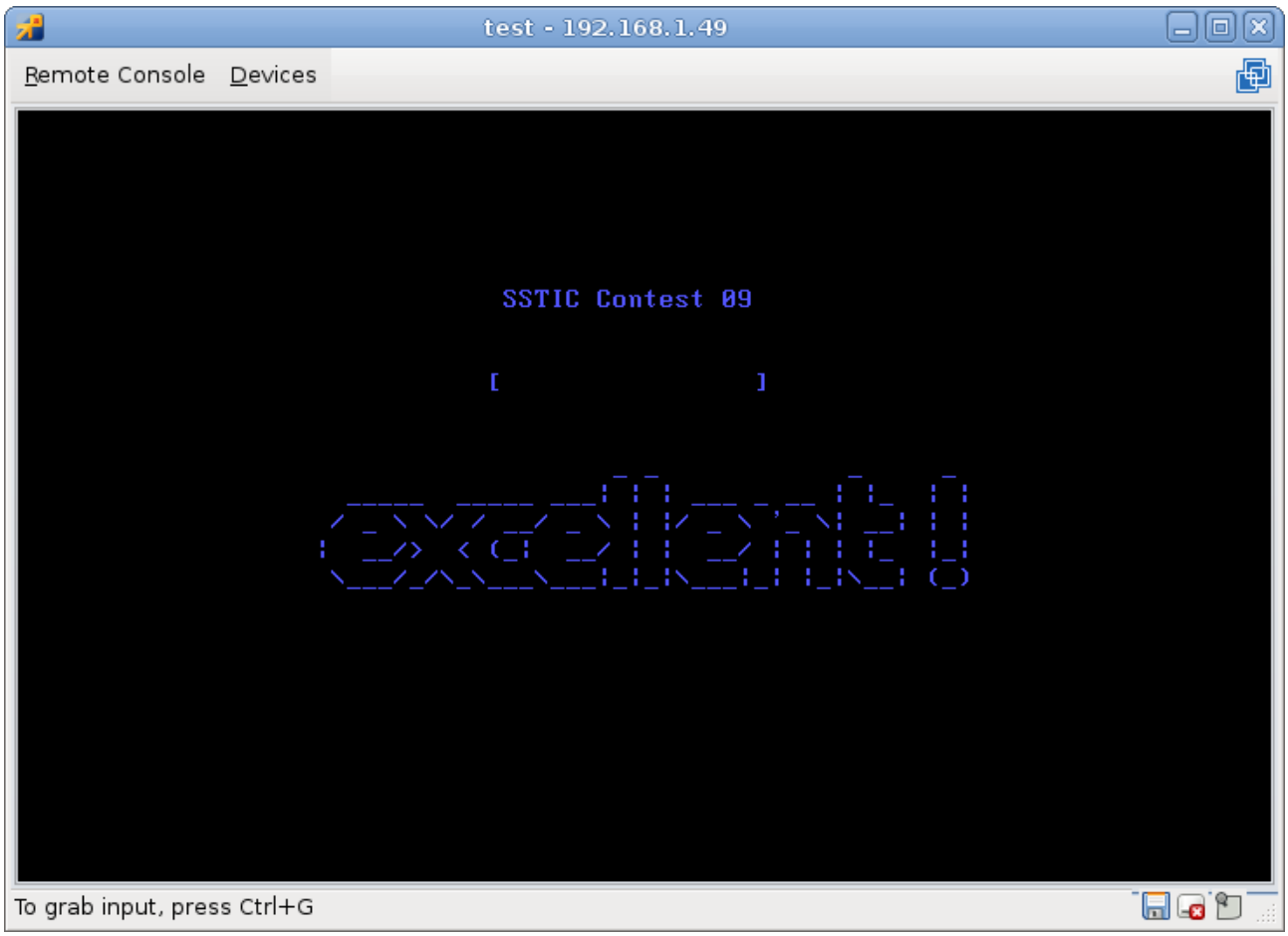
qui va nous permettre de remonter les 19 passes et trouver la passphrase :)

On compile, on lance, on pleure :'(Ca marche pas …

Le seul doute qui me restait encore était au niveau de la constante, étant donné que j'ai fait toute l'analyse dynamique sur la première passe uniquement.

J'ai donc tracé les 19 passes en affichant les différentes constantes utilisées, et modifié mon « décodeur » en conséquence, pour retrouver la passephrase d'origine :

```
fab@sawfish-laptop:~/sstic$ ./dec | tail -n 1 | sed -e 's/ //g' |
perl -pe 's/(..)(..)(..)
(..)/chr(hex($4)).chr(hex($3)).chr(hex($2)).chr(hex($1))/ge'
Z7aw?gW\=HL|Np_9

fab@sawfish-laptop:~/sstic$ echo -n 'Z7aw?gW\=HL|Np_9' | md5sum |
sed 's/ .*/@sstic.org/'
33e96dc842b082d2339b51177ee25ee5@sstic.org
```


test - 192.168.1.49
Remote Console
Devices
SSTIC Contest 09
[ ]
To grab input, press Ctrl+G

7. Code annexe – décodeur

Voici la source en code goret pour le décodeur de la partie 1 :

```
#include <stdio.h>
#include <sys/types.h>

int main() {
     u_int32_t consts[19] = { 0x7f434625, 0xa1882186, 0xa882a3fa,
0xe52d841a, 0xce2684cb, 0xd82420e6, 0xa4e8f9fe, 0xb45692eb,
0xa1010fea, 0xbf6470c8, 0x2e36ba18, 0xdffb67a8, 0x436a9322,
0x79e3ad8e, 0x6a5f7a29, 0x1bccc67a, 0xa1b52732, 0x11a35424,
0x693fa863 };
     u_int32_t vals[4] = { 0x9e8db86a, 0xce6d6c69, 0xbcc7253c,
0x196cd5f6 };
     u_int32_t const2[4] = { 0x4b9e2731, 0xaab1bbe6, 0xe4e4cfe3,
0x5baa36bd };
     int i, j;

     for(j=18; j>=0; j--) {
          for(i=3; i>=0; i--) {
            int ind=((consts[j]>>2)&3)^i;
            vals[i]=vals[i] - ( ( (vals[i==3?0:i+1]^consts[j]) +
(const2[ind]^vals[i==0?3:i-1]) ) ^ ( ( (vals[i==0?3:i-1]>>5) ^
(vals[i==3?0:i+1]<<2) ) + ( (vals[i==0?3:i-1]<<4) ^ (vals[i==3?
0:i+1]>>3) )));
          }
     }

     printf("%08x %08x %08x %08x\n", vals[0], vals[1], vals[2],
vals[3]);

     return 0;
}
```

8. Conclusion

Ma solution est peut-être un peu décevante, dans le sens où elle va directement à l'essentiel, à savoir la résolution du challenge, sans chercher à comprendre tous les mécanismes sous-jacents, mais elle a le mérite d'amener la solution :) Ca sent la VM là dessous, avec les interruptions en tant qu'instructions, mais j'ai pas creusé plus que ça !

Outils utilisés : IDA, gdb, VMWare Server, mais aussi gnome-calculator, gedit, grep, sed, perl ...