Analyse du challenge SSTIC 2009

Vincent Bénony

Après un rapide coup d'œil dans un éditeur héxadécimal, il apparaît certaines chaînes de caractères plutôt évocatrices (GRUB, …). Le challenge se présente donc sous la forme d'une image d'un support de type disquette. Elle est au format Ext3, et contient un bootloader en charge d'exécuter un fichier kernel.bin.

Mes choix préliminaires se sont portés sur *qemu*, *GDB* et *IDA*. Malheureusement, ma version de *qemu* n'était pas capable d'exécuter dans son intégralité le challenge. Alors, plutôt que de partir dans une mauvaise direction à cause d'un crash quelconque, j'ai décidé d'utiliser *VMWare Fusion*, étant sous MacOS X pour ce challenge. De plus, comme pour *qemu*, il est possible de démarrer un serveur *GDB* en modifiant le fichier de description de la machine virtuelle.

La première phase de l'analyse du challenge sera entièrement statique.

Le premier réflexe à avoir face à ce type de challenge est de rechercher quelques points d'appui : les plus simples à trouver sont en général les endroits dans le code où les différents textes sont affichés. Après un rapide désassemblage dans *IDA*, il apparaît que la chaîne « SSTIC Contest 09 » est présente en clair à l'adresse 0x2E17E0. Afin de s'assurer qu'il ne s'agit pas d'un leurre, il suffit de modifier cette chaîne dans un éditeur quelconque et de valider que ce changement affecte bien le challenge, ce qui est bien le cas. Si *IDA* ne se trompe pas, cette zone de données est référencées une unique fois dans le code, et la procédure l'utilisant semble démarrer en 0x20158C.

Tout au long du challenge, je formulerai certaines hypothèses fortes, et je commence ici par décréter que la plupart des zones de mémoires autour de cette procédure sont vraisemblablement en rapport avec de l'affichage, comme des procédures de gestion de coordonnées de curseur ou ce genre de choses. Il s'avère que cette hypothèse est très souvent vérifiée à cause du fait que tous les programmeurs procèdent de la même manière pour écrire du code : ils écrivent des procédures concernant une tâche précise qu'ils regroupent dans des bibliothèques qui seront ensuite linkées les unes aux autres, conservant ainsi un certaines proximités des méthodes dans le binaire généré.

Je vais donc, petit à petit, renommer certaines parties du code avec des noms indiquant que je soupçonne qu'elles servent à des choses graphiques. Ainsi, si à d'autres moment dans mon analyse je retombe sur ces portions de code alors que j'étais sous la même hypothèse, je saurais que la probabilité que le nom que je lui ai donné la première fois était vraisemblablement correct. Je préfixe généralement les noms des fonctions dont je ne suis pas certain par « maybe_ » pour éviter d'être induit en erreur par la suite. J'appellerai donc la méthode en 0x20158C « maybe_drawTitle ».

En remontant à l'appelant, on remarque quelques appels de méthodes contenant des instruction très bas niveaux (LLDT, LIDT …). Il s'avèrera par la suite que le challenge fait beaucoup appel à la segmentation et à des passages mode protégé / mode réel pour obfusquer le code. J'appelle cette méthode « maybe_someInits ». Elle est elle même appelée depuis la méthode représentant le point d'entrée de l'exécutable.

Une deuxième méthode appelée depuis le point d'entré se trouve à l'adresse 0x200060. Cette méthode attire particulièrement mon attention dans le sens où il s'agit d'une boucle sans fin qui appel toujours une méthode en 0x20006D: j'ai donc vraisemblablement trouvé la boucle principale du challenge, qui doit certainement être de la forme « lecture de la passphrase au clavier » → « vérification du challenge » → « lecture de la passphrase » …

Et en effet, d'après *IDA*, beaucoup de lectures de port d'entré/sortie correspondant au clavier sont détectés dans cette méthode. On retrouve certaines méthodes que je suspectais être relative au tracé de texte et au déplacement de curseur. En remplacent certains des appels par des instructions NOP il m'est possible de confirmer leur rôle dans le code. Ainsi, la méthode à l'adresse 0x20006D se présente sous une forme pseudo-code:

```
        effaceLigne()
        afficheCaractère('[')
boucle:
        affichePassphrase()
        attendTouche()
        si caractère affichable:
                ajoute caractère passphrase
        si touche entrée:
                verifiePassphrase()
        boucler
```

On remarquera au passage une utilisation étrange du registre CR4, et de son bit PGE gérant la pagination.

Le fait que ce challenge soit un exécutable bootable complique beaucoup les choses, puisqu'il a accès à toutes les fonctionnalités du processeur (il tourne en RING 0), mais il souffre aussi d'un gros défaut qui est qu'il ne bénéficie pas de choses proposées par beaucoup d'OS comme la randomisation de l'espace mémoire. Ainsi, on remarquera très vite qu'une adresse revient très souvent dans le code, il s'agit de l'adresse 0x322288. Cette zones marque le début de l'IDT et de la GDT. A un certain offset à partir de cette zone est copié la passphrase telle qu'entrée par l'utilisateur. Certaines zones sont renseignées avec des valeurs que nous ne maîtrisons pas encore, puis un appel à une méthode étrange à lieu : elle change explicitement la valeur du registre ESP juste avant d'exécuter une instruction RET : il s'agit donc d'un appel de méthode caché, où une pile d'exécution est forgée de toute pièce.

A partir de cet instant, je commence à exécuter le challenge dans une machine virtuelle, sous le contrôle de *GDB*. La destination de l'appel de méthode qui est caché se révèle être encore plus étrange : il s'agit de la méthode qui se trouve à l'adresse 0x2006D6. En effet, elle empile la valeur des registres de segments avant d'exécuter cette fois-ci un IRET. La différence avec un RET est que le registre d'état EFLAGS ainsi que le sélecteur de segment CS sont dépilés. Il s'avèrera qu'il s'agit du mécanisme mis en place dans le challenge pour basculer du mode protégé au mode réel très souvent. Cette méthode est en fait un pont par lequel passera souvent le flot d'exécution.

On s'aperçoit vite que le code est très lourdement obfusqué. Je décide donc de poser plusieurs point d'arrêt dans les méthode que je suspecte être des méthodes de tracé de caractères : il y a bien un moment où le flot d'exécution arrivera à cette endroit pour afficher le fait que le challenge est réussit où raté. Et en effet l'exécution s'arrête à l'adresse 0x20019B3 qui sert à afficher une chaîne de caractères de longueur quelconque. En remontant à l'appelant (procédure débutant à l'adresse 0x200a34), on remarque qu'avant d'appeler la méthode de tracé de chaîne, on exécute une boucle qui fait un XOR d'une valeur 32 bits en dur dans le code avec un buffer en mémoire : les chaînes de caractères « Failed ! » (et « Excellent ! ») sont cachées dans le code avec le masque 0xA5CC1A27). J'appelle la méthode en 0x200A34 « drawCipheredText ».

Dans la frame précédente, on trouve la méthode débutant en 0x200AD5, et c'est là qu'on remarque une comparaison dans une boucle entre un buffer situé à l'adresse 0x70000 et un autre buffer chiffré avec le masque 0xA5CC1A27, et qu'en fonction de cette comparaison, la méthode

drawCipheredText est appelée avec un texte ou un autre. Nous avons trouvé le test final, reste à savoir qui à généré les données présentes à l'adresse 0x70000.

Le code étant très obfusqué, je décide de logger toutes les instructions qui seront exécutées depuis l'adresse 0x200B94 (qui me semble être le début des hostilités) jusqu'à ladresse 0x200AD5, et d'enregistrer aussi toutes les modifications de valeur dans les registres qui se produisent. Pour cela, un simple script GDB pourra suffire.

Après une première exécution complète, je remarque que les adresses qu'emprunte EIP sont regroupables en deux catégories : celles en dessous de 0x200000 et celles au dessus. Je remarque aussi assez vite que les instructions dont l'adresse est inférieure à 0x200000 semblent assez incohérentes, et l'explication sera vite trouvée : ces instructions sont en réalité exécutées en mode 16 bits. Je modifie donc mon script afin d'intégrer dans le désassemblage cette information (voir en annexe A le script final utilisé). J'obtiens au final un log cohérent d'une vingtaine de méga octets de cette forme :

```
0x200b94:       push   ebp
0x00200b95 in ?? ()
0x200b95:       mov    ebp,esp
0x00200b97 in ?? ()
 EBP=0x1fff9c
0x200b97:       sub    esp,0x18
0x00200b9a in ?? ()
0x200b9a:       mov    eax,DWORD PTR [ebp+0xc]
0x00200b9d in ?? ()
 EAX=0x10
0x200b9d:       mov    BYTE PTR [ebp-0x14],al
0x00200ba0 in ?? ()
0x200ba0:       mov    DWORD PTR [ebp-
0xc],0x322288
0x00200ba7 in ?? ()
0x200ba7:       mov    eax,DWORD PTR [ebp-0xc]
0x00200baa in ?? ()
 EAX=0x322288
0x200baa:       add    eax,0x820
0x00200baf in ?? ()
 EAX=0x322aa8
0x200baf:       mov    DWORD PTR [ebp-0x8],eax
0x00200bb2 in ?? ()
0x200bb2:       mov    eax,DWORD PTR [ebp-0xc]
0x00200bb5 in ?? ()
 EAX=0x322288
...
```

Il s'agit maintenant de tenter de comprendre ce qu'il s'y passe. Pour cela, je vais écrire un petit programme en C dont le but sera de filtrer ce log au fur et à mesure de ma compréhension.

La première chose qui saute aux yeux est que ce début est une boucle de copie de données. En fait, en observant l'évolution des registres en ce début de log, on remarque que la passphrase est copiée à l'adresse 0x60000. J'intègre alors quelques informations dans mon filtreur concernant certaines adresses semblant être un équivalent des fonctions memset et memcpy de la libc.

Une autre chose qui saute très vite aux yeux dans ce log est que beaucoup d'instructions étranges sont utilisées, comme IN et OUT, et qu'à chaque fois qu'elle le sont, l'exécution se poursuit à une adresse qui n'est pas directement à la suite dans le code. En fait, il s'agit d'une astuce pour repasser du mode réel au mode protégé : ces instructions étant privilégiées, elles déclenchent une exception qui est récupérée par un gestionnaire qui continue l'exécution à un endroit qui dépend de la valeur des registres au moment où l'appel à IN, OUT ou INT s'est produit. Dans ce gestionnaire d'exception, le calcul est continué, à la différence que ce ne sont plus les registres du processeur qui

sont utilisés, mais que toutes les étapes transitent vers une zone mémoire où ont été sauvegardés les valeurs des registres en mode réel juste avant l'exception. Ainsi, au retour au mode réel, les registres contiendront les valeurs nécessaires à la poursuite du calcul. Cette méthode est très astucieuse, par contre, le compilateur utilisé à laissé trop de régularité dans ces parties de code. Ainsi, beaucoup de motifs se retrouvent, et je décide d'exploiter cette information en modifiant à nouveau mon filtre de fichier log.

L'idée est alors de dire qu'à chaque fois que je rencontre dans du code dont l'EIP est supérieur à 0x200000 (donc en mode protégé) une instruction « MOV EAX, DWORD PTR [EBP + 8] », alors je considérerai que EAX contient ensuite un pointeur vers une structure contenant les valeurs des registres du mode réel, et que quand on accédera à cette zone mémoire avec une instruction du type « MOV xxx, DWORD PTR [EAX + off] », alors ces instructions seront réécrite sous la forme « MOV xxx, VIRTUAL_REG_yyy » où yyy dépend de l'offset de départ.

Je maintiens donc simplement un état m'indiquant si EAX pointe sur la zone mémoire qui m'intéresse ou pas pour savoir si les instructions doivent être réécrites autrement.

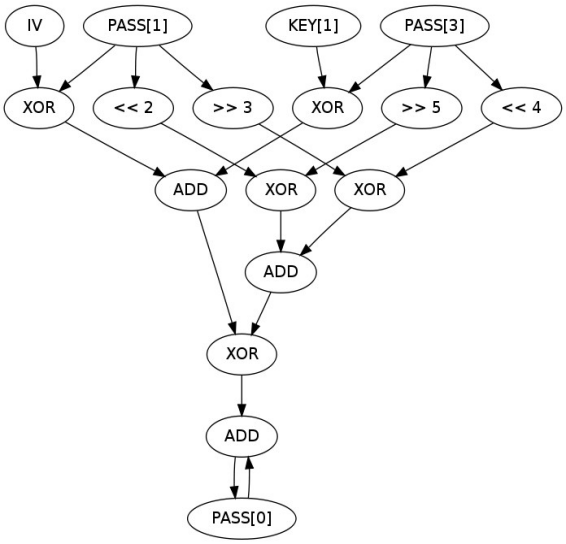
En évitant d'afficher aussi les instructions du type « PUSH EBP » ou encore « IRET », « RET », « CALL » ou « JMP », on clarifie encore un peu plus les choses. Le programme du filtreur final est donnée en annexe B.

Malgré ces efforts, la compréhension reste délicate. Je décide alors de poser quelques points d'arrêt matériels à l'écriture de certaines zones mémoire. Les zones qui m'intéressent sont 0x60000 - 0x60010 et 0x70000 – 0x70040, puisqu'il s'agit des zones où se trouvent la passphrase ainsi que celle représentant le buffer final comparé à une données en dur.

J'écris donc un nouveau script GDB afin de loguer tous ces accès jusqu'à l'arrivée en 0x200AD5.

On remarque alors très vite que tout le flot d'exécution se découpe en deux phases : une modification de la passphrase en 0x60000 à l'aide de données en 0x80000 (une sorte de clé), puis une génération de données en 0x70000 en fonction des données crées en 0x60000.

La première phase opère une diffusion des bits de la passphrase en 19 passes, sur chacun des 4 mots de 32 bits de la passphrase. En observant dans le log du flot d'exécution les endroit où on été écrites les valeurs en 0x60000-0x60010 on remarque que le schéma est toujours similaire. Un log est donnée en Annexe C ; Une opération est exécutée 19 x 4 fois sur la passphrase, uniquement avec des indexes différents. Le schéma ressemble pour la première itération à ceci :



Le problème est alors de trouver les différents IV servant aux 19 passes, ainsi que les indexes dans la clé et dans la passphrase utilisés. Je n'ai pas pu généraliser facilement cette opération et écrire un filtre pour le faire, j'ai donc décider d'aller regarder dans le log les instructions proches de chaque écriture. Une fois la rétine bien imprégnée du motif du code, on arrive assez rapidement à extraire les 19 vecteurs d'initialisations, ainsi que les différents offsets, c'est uniquement l'histoire d'une heure et d'un peu de courage...

On remarquera au passage que cette opération est facilement inversible, en effet, à chaque itération, on calcul une quantité en fonction de trois des quatre mots de la passphrase pour l'ajouter au quatrième mot. Il suffit donc d'effectuer les passes dans l'autre sens pour calculer ce delta, et de le retrancher à chaque fois.

La seconde phase est plus obscure. Je décide d'analyser le code autour des zones effectuant des modifications dans le buffer à l'adresse 0x70000-0x70040. Je remarque qu'il y a 48 étapes, regroupables en trois types d'opérations atomiques. Chacune de ces opérations opère sur un mot de 16 bits de la passphrase modifiée pour générer un mot de 16 bits à destination du buffer visé.

Le premier type d'opération atomique effectue très simplement une rotation du mot de 16 bits (échange des deux octets) et complémente le résultat à 1.

Le deuxième type calcul X *mod* Y, avec X le mot de 16 bits lu de la passphrase, et Y une constante dans le code.

Le dernier type effectue une opération un peu plus complexe de la forme :

$$X - ((X * F1) >> S) * F2$$

avec F1, F2 et S des constantes 16 bits dans le code.

Il suffit alors d'analyser le log d'exécution pour extraire les différentes constantes utilisées. On remarquera alors que ces opérations sont regroupables, et qu'en fait on calcul les fonctions de type 2 et 3, puis que l'intégralité du buffer passe par des opération du type 1.

Un texte dans l'exécutable nous met sur la voie du théorème des restes Chinois pour résoudre cette phase. Dans les faits, s'agissant de mots de 16 bits, une simple résolution par recherche systématique sera suffisante. Il suffit donc de regrouper toutes les fonctions prenant en entrée le premier mots de 16 bits de la passphrase, de résoudre tester toutes les valeurs de 0 à 65536, et de conserver la seule qui vérifie les trois équations. On répète l'opération pour tous les mots de 16 bits de la passphrase.

On retrouve alors la valeur du buffer à la fin de la phase 1, qui est ensuite lui même inversé par la méthode décrite plus haut. Le programme en C permettant de calculer la passphrase en fonction du buffer chiffré découvert dans le code est donné en annexe D.

La passphrase retrouvée est alors : Z7aw?gW\=HL|Np_9

En conclusion, ce challenge présente beaucoup de chose très intéressantes, malheureusement non accessible à un programme s'exécutant en RING 3. Il sera intéressant de remarquer que certaines petites modifications simple permettrait de compliquer fortement ce challenge, à commencer par le fait que la structure du code généré par le compilateur utilisé est trop répétitive,

ce qui a permis de simplifier de manière drastique le log du flot d'exécution. Il est aussi possible de compliquer l'inversion de la phase 1 en faisant en sorte que l'algorithme de la phase 2 n'utilise pas l'intégralité de la passphrase modifiée : en effet, si 64 bits par exemple n'étaient pas utilisés, il aurait alors fallu trouver parmi les 2^64 solutions possibles de passphrase modifiée laquelle ne contenait que des caractères affichables, et saisissable au clavier. Une autre faiblesse concerne le fait que la primitive utilisée dans la première phase est toujours la même, aux paramètres près, ce qui simplifie une fois encore l'extraction d'information du code obfusqué.

Ce challenge a été une expérience fort agréable, et a bien occupé la dizaine d'heure que je lui ai consacré. Je remercie vivement l'équipe à l'origine de ce challenge fort enrichissant, et j'espère pouvoir m'exercer sur un nouveau pour la prochaine édition de SSTIC !

Annexe A

Script de log des instructions exécutées

```
target remote localhost:8832
tb *0x00200b94
c

set $old_eax = $eax
set $old_ebx = $ebx
set $old_ecx = $ecx
set $old_edx = $edx
set $old_esi = $esi
set $old_edi = $edi
set $old_ebp = $ebp

set $finalEIP = 0x200ad5
set $virtualRegs = 0x326ae0

set $old_r0 = 0
set $old_r1 = 0
set $old_r2 = 0
set $old_r3 = 0
set $old_r4 = 0
set $old_r5 = 0
set $old_r6 = 0
set $old_r7 = 0
set $old_r8 = 0
set $old_r9 = 0
set $old_r10 = 0
set $old_r11 = 0
set $old_r12 = 0
set $old_prg_ptr = 0
set $old_prg_sel = 0
set $old_sel0 = 0
set $old_sel1 = 0
set $old_sel2 = 0
set $old_sel3 = 0
set $old_sel4 = 0
set $old_sel5 = 0
set $old_sel6 = 0
set $old_sel7 = 0
set $old_sel8 = 0
set $old_sel9 = 0
set $old_sel10 = 0
set $old_sel11 = 0
set $old_sel12 = 0
set $old_sel13 = 0

set $c = 32

while $eip != $finalEIP
      if $eip < 0x200000
            if $c == 32
                  set $c = 16
                  set architecture i8086
            end
      else
            if $c == 16
                  set $c = 32
                  set architecture i386
            end
      end

      if *(char *)$eip == 0x66
            if *(char *)($eip+1) == 0x17 || *(short *)($eip+1) == 0xa90f ||
*(short *)($eip+1) == 0xa10f || *(char *)($eip+1) == 0x07 || *(char *)($eip+1)
== 0x1f || *(char *)($eip+1) == 0x16 || *(short *)($eip+1) == 0xa80f || *(short
*)($eip+1) == 0xa00f || *(char *)($eip+1) == 0x06 || *(char *)($eip+1) == 0x1e
```

```
            x/2i $eip
      else
            x/1i $eip
      end
else
      x/1i $eip
end

si

set $prt = 0

if $old_eax != $eax
      set $old_eax = $eax
      printf " EAX=0x%x",$eax
      set $prt = 1
end
if $old_ebx != $ebx
      set $old_ebx = $ebx
      printf " EBX=0x%x",$ebx
      set $prt = 1
end
if $old_ecx != $ecx
      set $old_ecx = $ecx
      printf " ECX=0x%x",$ecx
      set $prt = 1
end
if $old_edx != $edx
      set $old_edx = $edx
      printf " EDX=0x%x",$edx
      set $prt = 1
end
if $old_esi != $esi
      set $old_esi = $esi
      printf " ESI=0x%x",$esi
      set $prt = 1
end
if $old_edi != $edi
      set $old_edi = $edi
      printf " EDI=0x%x",$edi
      set $prt = 1
end
if $old_ebp != $ebp
      set $old_ebp = $ebp
      printf " EBP=0x%x",$ebp
      set $prt = 1
end

if ((int *)$virtualRegs)[0] != $old_r0
      set $old_r0 = ((int *)$virtualRegs)[0]
      printf " R0=0x%x", $old_r0
      set $prt = 1
end
if ((int *)$virtualRegs)[1] != $old_r1
      set $old_r1 = ((int *)$virtualRegs)[1]
      printf " R1=0x%x", $old_r1
      set $prt = 1
end
if ((int *)$virtualRegs)[2] != $old_r2
      set $old_r2 = ((int *)$virtualRegs)[2]
      printf " R2=0x%x", $old_r2
      set $prt = 1
end
if ((int *)$virtualRegs)[3] != $old_r3
      set $old_r3 = ((int *)$virtualRegs)[3]
      printf " R3=0x%x", $old_r3
      set $prt = 1
end
if ((int *)$virtualRegs)[4] != $old_r4
      set $old_r4 = ((int *)$virtualRegs)[4]
      printf " R4=0x%x", $old_r4
      set $prt = 1
```

```
end
if ((int *)$virtualRegs)[5] != $old_r5
      set $old_r5 = ((int *)$virtualRegs)[5]
      printf " R5=0x%x", $old_r5
      set $prt = 1
end
if ((int *)$virtualRegs)[6] != $old_r6
      set $old_r6 = ((int *)$virtualRegs)[6]
      printf " R6=0x%x", $old_r6
      set $prt = 1
end
if ((int *)$virtualRegs)[7] != $old_r7
      set $old_r7 = ((int *)$virtualRegs)[7]
      printf " R7=0x%x", $old_r7
      set $prt = 1
end
if ((int *)$virtualRegs)[8] != $old_r8
      set $old_r8 = ((int *)$virtualRegs)[8]
      printf " R8=0x%x", $old_r8
      set $prt = 1
end
if ((int *)$virtualRegs)[9] != $old_r9
      set $old_r9 = ((int *)$virtualRegs)[9]
      printf " R9=0x%x", $old_r9
      set $prt = 1
end
if ((int *)$virtualRegs)[10] != $old_r10
      set $old_r10 = ((int *)$virtualRegs)[10]
      printf " R10=0x%x", $old_r10
      set $prt = 1
end
if ((int *)$virtualRegs)[11] != $old_r11
      set $old_r11 = ((int *)$virtualRegs)[11]
      printf " R11=0x%x", $old_r11
      set $prt = 1
end
if ((int *)$virtualRegs)[12] != $old_r12
      set $old_r12 = ((int *)$virtualRegs)[12]
      printf " R12=0x%x", $old_r12
      set $prt = 1
end
if ((int *)$virtualRegs)[13] != $old_prg_ptr
      set $old_prg_ptr = ((int *)$virtualRegs)[13]
      printf " PRG_PTR=0x%x", $old_prg_ptr
      set $prt = 1
end
if ((int *)$virtualRegs)[14] != $old_prg_sel
      set $old_prg_sel = ((int *)$virtualRegs)[14]
      printf " PRG_SEL=0x%x", $old_prg_sel
      set $prt = 1
end
if ((int *)$virtualRegs)[15] != $old_sel1
      set $old_sel1 = ((int *)$virtualRegs)[15]
      printf " SEL1=0x%x", $old_sel1
      set $prt = 1
end
if ((int *)$virtualRegs)[16] != $old_sel2
      set $old_sel2 = ((int *)$virtualRegs)[16]
      printf " SEL2=0x%x", $old_sel2
      set $prt = 1
end
if ((int *)$virtualRegs)[17] != $old_sel3
      set $old_sel3 = ((int *)$virtualRegs)[17]
      printf " SEL3=0x%x", $old_sel3
      set $prt = 1
end
if ((int *)$virtualRegs)[18] != $old_sel4
      set $old_sel4 = ((int *)$virtualRegs)[18]
      printf " SEL4=0x%x", $old_sel4
      set $prt = 1
end
if ((int *)$virtualRegs)[19] != $old_sel5
```

```
        set $old_sel5 = ((int *)$virtualRegs)[19]
        printf " SEL5=0x%x", $old_sel5
        set $prt = 1
    end
    if ((int *)$virtualRegs)[20] != $old_sel6
        set $old_sel6 = ((int *)$virtualRegs)[20]
        printf " SEL6=0x%x", $old_sel6
        set $prt = 1
    end
    if ((int *)$virtualRegs)[21] != $old_sel7
        set $old_sel7 = ((int *)$virtualRegs)[21]
        printf " SEL7=0x%x", $old_sel7
        set $prt = 1
    end
    if ((int *)$virtualRegs)[22] != $old_sel8
        set $old_sel8 = ((int *)$virtualRegs)[22]
        printf " SEL8=0x%x", $old_sel8
        set $prt = 1
    end
    if ((int *)$virtualRegs)[23] != $old_sel9
        set $old_sel9 = ((int *)$virtualRegs)[23]
        printf " SEL9=0x%x", $old_sel9
        set $prt = 1
    end
    if ((int *)$virtualRegs)[24] != $old_sel10
        set $old_sel10 = ((int *)$virtualRegs)[24]
        printf " SEL10=0x%x", $old_sel10
        set $prt = 1
    end
    if ((int *)$virtualRegs)[25] != $old_sel11
        set $old_sel11 = ((int *)$virtualRegs)[25]
        printf " SEL11=0x%x", $old_sel11
        set $prt = 1
    end
    if ((int *)$virtualRegs)[26] != $old_sel12
        set $old_sel12 = ((int *)$virtualRegs)[26]
        printf " SEL12=0x%x", $old_sel12
        set $prt = 1
    end
    if ((int *)$virtualRegs)[27] != $old_sel13
        set $old_sel13 = ((int *)$virtualRegs)[27]
        printf " SEL13=0x%x", $old_sel13
        set $prt = 1
    end

    if $prt == 1
        printf "\n"
    end
end
```

Annexe B

Filtreur de log

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum reg
{
	reg_none,
	reg_eax,
	reg_ebx,
	reg_ecx,
	reg_edx,
	reg_esi,
	reg_edi,
	reg_ebp,
	reg_esp,

	nbRegs
};

enum regState
{
	state_unknown,

	state_statePointer,
	state_constant,
	state_selector,
	state_offset,
	state_vmReg
};

enum reg textToReg(char * args, int index)
{
	while (index)
	{
		args = strstr(args, ",");
		args++;
		index--;
	}
	if (!strncmp(args, "eax", 3)) return reg_eax;
	if (!strncmp(args, "ebx", 3)) return reg_ebx;
	if (!strncmp(args, "ecx", 3)) return reg_ecx;
	if (!strncmp(args, "edx", 3)) return reg_edx;
	if (!strncmp(args, "esi", 3)) return reg_esi;
	if (!strncmp(args, "edi", 3)) return reg_edi;
	if (!strncmp(args, "ebp", 3)) return reg_ebp;
	if (!strncmp(args, "esp", 3)) return reg_esp;
	return reg_none;
}

const char * vmRegsNames[] = {
	"R0", "R1", "R2", "R3",
	"R4", "R5", "R6", "R7",
	"INDIRECT_OPCODE",
	"R9", "R10", "R11", "R12",
	"PROGRAM_PTR",
	"PRG_SEL",
	"FLAG",
	"SEL1", "SEL2", "SEL3", "SEL4",
	"SEL5", "SEL6", "SEL7", "SEL8",
	"SEL9", "SEL10", "SEL11", "SEL12",
	"SEL13"
};

enum Drawn
{
	drawn_none,
```

```
	drawn_memset,
	drawn_memcpy,
	drawn_exchangeESP
};

int main(int argc, char **argv)
{
	FILE * f = fopen("newlog", "r");
	int lastDrawn = drawn_none;

	enum regState states[nbRegs];
	int i;
	for (i=0; i<nbRegs; i++) states[i] = state_unknown;

	int adr = 0;

	while (!feof(f))
	{
		char line[256];
		fgets(line, 256, f);

		if (strstr(line, "??")) continue;
		if (strstr(line, "The target")) continue;
		if (!strncmp(line, "0x", 2)) sscanf(line, "0x%x:", &adr);

		// Filtering by address
		if (adr >= 0x200694 && adr <= 0x200cfd) continue;
		if (adr >= 0x201b4c && adr <= 0x201b76)
		{
			if (lastDrawn != drawn_memset)
			{
				printf("\t\tMEMSET\n");
			}
			lastDrawn = drawn_memset;
			continue;
		}
		else if (adr >= 0x201b77 && adr <= 0x201bac)
		{
			if (lastDrawn != drawn_memcpy)
			{
				printf("\t\tMEMCPY\n");
			}
			lastDrawn = drawn_memcpy;
			continue;
		}
		else if (adr >= 0x200680 && adr <= 0x200686)
		{
			if (lastDrawn != drawn_exchangeESP)
			{
				printf("\t\tEXCHANGE_ESP\n");
			}
			lastDrawn = drawn_exchangeESP;
			continue;
		}
		else
		{
			lastDrawn = drawn_none;
		}

		if (line[0] == ' ')
		{
			line[strlen(line) - 1] = 0;
			printf("\t\t(%s)\n", line + 1);
			continue;
		}

		char opcode[64];
		char args[256];
		int i = 0, j = 0;
		while (line[i] != 9 && line[i] != 32) i++;
		while (line[i] == 32 || line[i] == 9) i++;
		while (line[i] != 32 && line[i] != 9 && line[i] != 10) opcode[j++] =
```

```
line[i++];
            opcode[j] = 0;
            j = 0;
            args[j] = 0;
            while (line[i] == 32 || line[i] == 9) i++;
            while (line[i] != 10) args[j++] = line[i++];
            args[j] = 0;

            if (!strcmp(opcode, "push") && !strcmp(args, "esp")) continue;
            if (!strcmp(opcode, "pop") && !strcmp(args, "esp")) continue;
            if (!strcmp(opcode, "jmp")) continue;
            if (!strcmp(opcode, "call")) continue;
            if (!strcmp(opcode, "ret")) continue;
            if (!strcmp(opcode, "iret")) continue;
            if (!strcmp(opcode, "leave")) continue;
            if (!strcmp(opcode, "mov"))
            {
                  if (!strcmp(args, "ebp,esp")) continue;
            }
            if (!strcmp(opcode, "sub"))
            {
                  if (!strncmp(args, "esp,0x", 6)) continue;
            }

            if (!strcmp(opcode, "mov"))
            {
                  if (!strncmp(args, "eax,", 4))
                  {
                        if (!strcmp(args, "eax,DWORD PTR [ebp+0x8]"))
                        {
                              states[reg_eax] = state_statePointer;
                              continue;
                        }
                        else if (!strncmp(args, "eax,DWORD PTR [eax+", 19) &&
(states[reg_eax] == state_statePointer))
                        {
                              int offset;
                              sscanf(args + 19, "0x%x", &offset);
                              states[reg_eax] = state_vmReg;
                              printf("0x%x\tVIRTUAL_MOV\teax,%s\n", adr,
vmRegsNames[offset >> 2]);
                              continue;
                        }
                        else
                        {
                              states[reg_eax] = state_unknown;
                        }
                  }
                  else if (!strncmp(args, "DWORD PTR [eax+", 15) &&
(states[reg_eax] == state_statePointer))
                  {
                        int offset;
                        sscanf(args + 15, "0x%x", &offset);
                        printf("0x%x\tVIRTUAL_MOV\t%s,%s\n", adr,
vmRegsNames[offset >> 2], strstr(args, ",") + 1);
                        continue;
                  }
            }

            printf("%s", line);
      }
      return 0;
}
```

Annexe C

Log des modifications en 0x60000

```
Initial passphrase : abcdefghijklmnop

pass 0
 (EIP 0x0000158d): 29 7b 7d c0 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70    part 0 : 0x64636261 -> 0xc07d7b29 (+ 0x5c1a18c8)
 (EIP 0x002632e8): 29 7b 7d c0 a3 10 6a c5 69 6a 6b 6c 6d 6e 6f 70    part 1 : 0x68676665 -> 0xc56a10a3 (+ 0x5d02aa3e)
 (EIP 0x00277980): 29 7b 7d c0 a3 10 6a c5 b1 09 1b fa 6d 6e 6f 70    part 2 : 0x6c6b6a69 -> 0xfa1b09b1 (+ 0x8daf9f48)
 (EIP 0x002201a4): 29 7b 7d c0 a3 10 6a c5 b1 09 1b fa 6d d8 49 d2    part 3 : 0x706f6e6d -> 0xd249d86d (+ 0x61da6a00)

pass 1
 (EIP 0x002c359a): cc c0 2d 53 a3 10 6a c5 b1 09 1b fa 6d d8 49 d2    part 0 : 0xc07d7b29 -> 0x532dc0cc (+ 0x92b045a3)
 (EIP 0x002a1cb3): cc c0 2d 53 2f b7 3d 32 b1 09 1b fa 6d d8 49 d2    part 1 : 0xc56a10a3 -> 0x323db72f (+ 0x6cd3a68c)
 (EIP 0x0024399c): cc c0 2d 53 2f b7 3d 32 28 b7 2c 59 6d d8 49 d2    part 2 : 0xfa1b09b1 -> 0x592cb728 (+ 0x5f11ad77)
 (EIP 0x00000a13): cc c0 2d 53 2f b7 3d 32 28 b7 2c 59 a4 43 8d 29    part 3 : 0xd249d86d -> 0x298d43a4 (+ 0x57436b37)

pass 2
 (EIP 0x0026ec13): 26 a3 94 53 2f b7 3d 32 28 b7 2c 59 a4 43 8d 29    part 0 : 0x532dc0cc -> 0x5394a326 (+ 0x0066e25a)
 (EIP 0x0027a8c5): 26 a3 94 53 82 6b b0 93 28 b7 2c 59 a4 43 8d 29    part 1 : 0x323db72f -> 0x93b06b82 (+ 0x6172b453)
 (EIP 0x00232d18): 26 a3 94 53 82 6b b0 93 59 6a 0e 13 a4 43 8d 29    part 2 : 0x592cb728 -> 0x130e6a59 (+ 0xb9e1b331)
 (EIP 0x0020c501): 26 a3 94 53 82 6b b0 93 59 6a 0e 13 c9 26 18 67    part 3 : 0x298d43a4 -> 0x671826c9 (+ 0x3d8ae325)

pass 3
 (EIP 0x00000bfa): 02 ad 0b 9f 82 6b b0 93 59 6a 0e 13 c9 26 18 67    part 0 : 0x5394a326 -> 0x9f0bad02 (+ 0x4b7709dc)
 (EIP 0x002c77b8): 02 ad 0b 9f f7 06 09 15 59 6a 0e 13 c9 26 18 67    part 1 : 0x93b06b82 -> 0x150906f7 (+ 0x81589b75)
 (EIP 0x002d8278): 02 ad 0b 9f f7 06 09 15 7e 64 ff 2c c9 26 18 67    part 2 : 0x130e6a59 -> 0x2cff647e (+ 0x19f0fa25)
 (EIP 0x0000093f): 02 ad 0b 9f f7 06 09 15 7e 64 ff 2c a4 b0 2d c0    part 3 : 0x671826c9 -> 0xc02db0a4 (+ 0x591589db)

pass 4
 (EIP 0x002675c9): 76 4e 72 4c f7 06 09 15 7e 64 ff 2c a4 b0 2d c0    part 0 : 0x9f0bad02 -> 0x4c724e76 (+ 0xad66a174)
 (EIP 0x00253646): 76 4e 72 4c f1 5d f4 a3 7e 64 ff 2c a4 b0 2d c0    part 1 : 0x150906f7 -> 0xa3f45df1 (+ 0x8eeb56fa)
 (EIP 0x00242f52): 76 4e 72 4c f1 5d f4 a3 2a cb 9b 07 a4 b0 2d c0    part 2 : 0x2cff647e -> 0x079bcb2a (+ 0xda9c66ac)
 (EIP 0x002108c3): 76 4e 72 4c f1 5d f4 a3 2a cb 9b 07 0a 8a 85 4d    part 3 : 0xc02db0a4 -> 0x4d858a0a (+ 0x8d57d966)

pass 5
 (EIP 0x00299e68): 27 78 59 87 f1 5d f4 a3 2a cb 9b 07 0a 8a 85 4d    part 0 : 0x4c724e76 -> 0x87597827 (+ 0x3ae729b1)
 (EIP 0x000010b2): 27 78 59 87 8d 06 32 c7 2a cb 9b 07 0a 8a 85 4d    part 1 : 0xa3f45df1 -> 0xc732068d (+ 0x233da89c)
 (EIP 0x002217f8): 27 78 59 87 8d 06 32 c7 db 15 95 a0 0a 8a 85 4d    part 2 : 0x079bcb2a -> 0xa09515db (+ 0x98f94ab1)
 (EIP 0x002d50bc): 27 78 59 87 8d 06 32 c7 db 15 95 a0 29 12 7a df    part 3 : 0x4d858a0a -> 0xdf7a1229 (+ 0x91f4881f)

pass 6
 (EIP 0x000013bc): 08 81 29 69 8d 06 32 c7 db 15 95 a0 29 12 7a df    part 0 : 0x87597827 -> 0x69298108 (+ 0xe1d008e1)
 (EIP 0x0000114e): 08 81 29 69 1c fd 1e 5d db 15 95 a0 29 12 7a df    part 1 : 0xc732068d -> 0x5d1efd1c (+ 0x95ecf68f)
 (EIP 0x0029d7e7): 08 81 29 69 1c fd 1e 5d db 75 d8 da 29 12 7a df    part 2 : 0xa09515db -> 0xdad875db (+ 0x3a436000)
 (EIP 0x0028ee26): 08 81 29 69 1c fd 1e 5d db 75 d8 da 28 11 8f fb    part 3 : 0xdf7a1229 -> 0xfb8f1128 (+ 0x1c14feff)

pass 7
 (EIP 0x002d7072): e6 95 77 97 1c fd 1e 5d db 75 d8 da 28 11 8f fb    part 0 : 0x69298108 -> 0x977795e6 (+ 0x2e4e14de)
 (EIP 0x0028cabf): e6 95 77 97 31 33 af 3d db 75 d8 da 28 11 8f fb    part 1 : 0x5d1efd1c -> 0x3daf3331 (+ 0xe0903615)
 (EIP 0x0020c870): e6 95 77 97 31 33 af 3d 88 0f 37 4e 28 11 8f fb    part 2 : 0xdad875db -> 0x4e370f88 (+ 0x735e99ad)
 (EIP 0x0029ef68): e6 95 77 97 31 33 af 3d 88 0f 37 4e 83 52 79 52    part 3 : 0xfb8f1128 -> 0x52795283 (+ 0x56ea415b)

pass 8
 (EIP 0x0028277e): 83 a4 7b de 31 33 af 3d 88 0f 37 4e 83 52 79 52    part 0 : 0x977795e6 -> 0xde7ba483 (+ 0x47040e9d)
 (EIP 0x00293a77): 83 a4 7b de 96 51 5a 97 88 0f 37 4e 83 52 79 52    part 1 : 0x3daf3331 -> 0x975a5196 (+ 0x59ab1e65)
 (EIP 0x002a7555): 83 a4 7b de 96 51 5a 97 28 2f ac 6b 83 52 79 52    part 2 : 0x4e370f88 -> 0x6bac2f28 (+ 0x1d751fa0)
 (EIP 0x00268aad): 83 a4 7b de 96 51 5a 97 28 2f ac 6b 35 8c d2 ad    part 3 : 0x52795283 -> 0xadd28c35 (+ 0x5b5939b2)

pass 9
 (EIP 0x00000899): 32 e3 3a 35 96 51 5a 97 28 2f ac 6b 35 8c d2 ad    part 0 : 0xde7ba483 -> 0x353ae332 (+ 0x56bf3eaf)
 (EIP 0x0029eac9): 32 e3 3a 35 a7 68 06 e6 28 2f ac 6b 35 8c d2 ad    part 1 : 0x975a5196 -> 0xe60668a7 (+ 0x4eac1711)
 (EIP 0x00220cc9): 32 e3 3a 35 a7 68 06 e6 3c c2 c5 51 35 8c d2 ad    part 2 : 0x6bac2f28 -> 0x51c5c23c (+ 0xe6199314)
 (EIP 0x0025ba17): 32 e3 3a 35 a7 68 06 e6 3c c2 c5 51 e0 bb 84 22    part 3 : 0xadd28c35 -> 0x2284bbe0 (+ 0x74b22fab)

pass 10
 (EIP 0x00288029): c7 9f 44 78 a7 68 06 e6 3c c2 c5 51 e0 bb 84 22    part 0 : 0x353ae332 -> 0x78449fc7 (+ 0x4309bc95)
 (EIP 0x0024e925): c7 9f 44 78 82 ba aa 56 3c c2 c5 51 e0 bb 84 22    part 1 : 0xe60668a7 -> 0x56aaba82 (+ 0x70a451db)
 (EIP 0x00000927): c7 9f 44 78 82 ba aa 56 57 28 0a 30 e0 bb 84 22    part 2 : 0x51c5c23c -> 0x300a2857 (+ 0xde44661b)
 (EIP 0x0024daa3): c7 9f 44 78 82 ba aa 56 57 28 0a 30 56 b9 96 23    part 3 : 0x2284bbe0 -> 0x2396b956 (+ 0x0111fd76)

pass 11
 (EIP 0x002c5dd9): f4 11 f6 57 82 ba aa 56 57 28 0a 30 56 b9 96 23    part 0 : 0x78449fc7 -> 0x57f611f4 (+ 0xdfb1722d)
 (EIP 0x002b1584): f4 11 f6 57 d7 d5 64 1e 57 28 0a 30 56 b9 96 23    part 1 : 0x56aaba82 -> 0x1e64d5d7 (+ 0xc7ba1b55)
 (EIP 0x00000e62): f4 11 f6 57 d7 d5 64 1e 0b c8 8b 52 56 b9 96 23    part 2 : 0x300a2857 -> 0x528bc80b (+ 0x22819fb4)
 (EIP 0x0022b83f): f4 11 f6 57 d7 d5 64 1e 0b c8 8b 52 ad 6f 60 23    part 3 : 0x2396b956 -> 0x23606fad (+ 0xffc9b657)

pass 12
 (EIP 0x000004c1): 0e 4d 54 c0 d7 d5 64 1e 0b c8 8b 52 ad 6f 60 23    part 0 : 0x57f611f4 -> 0xc0544d0e (+ 0x685e3b1a)
 (EIP 0x002449e4): 0e 4d 54 c0 0b 54 e9 45 0b c8 8b 52 ad 6f 60 23    part 1 : 0x1e64d5d7 -> 0x45e9540b (+ 0x27847e34)
 (EIP 0x00000f26): 0e 4d 54 c0 0b 54 e9 45 39 a1 3c 3e ad 6f 60 23    part 2 : 0x528bc80b -> 0x3e3ca139 (+ 0xebb0d92e)
 (EIP 0x002c1f29): 0e 4d 54 c0 0b 54 e9 45 39 a1 3c 3e 7f 8e 14 38    part 3 : 0x23606fad -> 0x38148e7f (+ 0x14b41ed2)

pass 13
 (EIP 0x00000d4d): a5 c0 e6 ff 0b 54 e9 45 39 a1 3c 3e 7f 8e 14 38    part 0 : 0xc0544d0e -> 0xffe6c0a5 (+ 0x3f927397)
 (EIP 0x000009df): a5 c0 e6 ff b0 9e 41 e0 39 a1 3c 3e 7f 8e 14 38    part 1 : 0x45e9540b -> 0xe0419eb0 (+ 0x9a584aa5)
 (EIP 0x0024976f): a5 c0 e6 ff b0 9e 41 e0 d8 87 c9 a4 7f 8e 14 38    part 2 : 0x3e3ca139 -> 0xa4c987d8 (+ 0x668ce69f)
 (EIP 0x002069a6): a5 c0 e6 ff b0 9e 41 e0 d8 87 c9 a4 a9 88 91 73    part 3 : 0x38148e7f -> 0x739188a9 (+ 0x3b7cfa2a)

pass 14
 (EIP 0x000006d1): cd 44 26 86 b0 9e 41 e0 d8 87 c9 a4 a9 88 91 73    part 0 : 0xffe6c0a5 -> 0x862644cd (+ 0x863f8428)
 (EIP 0x002aee8a): cd 44 26 86 c0 77 79 82 d8 87 c9 a4 a9 88 91 73    part 1 : 0xe0419eb0 -> 0x827977c0 (+ 0xa237d910)
 (EIP 0x002a935a): cd 44 26 86 c0 77 79 82 36 fd 56 bc a9 88 91 73    part 2 : 0xa4c987d8 -> 0xbc56fd36 (+ 0x178d755e)
 (EIP 0x0024087d): cd 44 26 86 c0 77 79 82 36 fd 56 bc 0b ce d7 63    part 3 : 0x739188a9 -> 0x63d7ce0b (+ 0xf0464562)

pass 15
 (EIP 0x00001239): e7 e0 ed 9e c0 77 79 82 36 fd 56 bc 0b ce d7 63    part 0 : 0x862644cd -> 0x9eede0e7 (+ 0x18c79c1a)
 (EIP 0x00000a83): e7 e0 ed 9e d3 f3 5c 06 36 fd 56 bc 0b ce d7 63    part 1 : 0x827977c0 -> 0x065cf3d3 (+ 0x83e37c13)
 (EIP 0x0026889f): e7 e0 ed 9e d3 f3 5c 06 26 77 55 f9 0b ce d7 63    part 2 : 0xbc56fd36 -> 0xf9557726 (+ 0x3cfe79f0)
 (EIP 0x0025bb45): e7 e0 ed 9e d3 f3 5c 06 26 77 55 f9 07 d4 d9 3d    part 3 : 0x63d7ce0b -> 0x3dd9d407 (+ 0xda0205fc)

pass 16
 (EIP 0x002bb637): c8 f8 af 8a d3 f3 5c 06 26 77 55 f9 07 d4 d9 3d    part 0 : 0x9eede0e7 -> 0x8aaff8c8 (+ 0xebc217e1)
 (EIP 0x0000062d): c8 f8 af 8a 53 4a 88 f4 26 77 55 f9 07 d4 d9 3d    part 1 : 0x065cf3d3 -> 0xf4884a53 (+ 0xee2b5680)
 (EIP 0x002a066d): c8 f8 af 8a 53 4a 88 f4 41 40 31 e6 07 d4 d9 3d    part 2 : 0xf9557726 -> 0xe6314041 (+ 0xecdbc91b)
 (EIP 0x0027fd81): c8 f8 af 8a 53 4a 88 f4 41 40 31 e6 e4 06 40 b5    part 3 : 0x3dd9d407 -> 0xb54006e4 (+ 0x776632dd)

pass 17
 (EIP 0x000014c3): c4 44 b0 b1 53 4a 88 f4 41 40 31 e6 e4 06 40 b5    part 0 : 0x8aaff8c8 -> 0xb1b044c4 (+ 0x27004bfc)
 (EIP 0x00000b28): c4 44 b0 b1 83 e9 52 49 41 40 31 e6 e4 06 40 b5    part 1 : 0xf4884a53 -> 0x4952e983 (+ 0x54ca9f30)
 (EIP 0x00225d7c): c4 44 b0 b1 83 e9 52 49 77 54 3e d3 e4 06 40 b5    part 2 : 0xe6314041 -> 0xd33e5477 (+ 0xed0d1436)
 (EIP 0x00293c85): c4 44 b0 b1 83 e9 52 49 77 54 3e d3 d3 a4 01 e7    part 3 : 0xb54006e4 -> 0xe701a4d3 (+ 0x31c19def)

pass 18
 (EIP 0x00001042): ac c3 5f 08 83 e9 52 49 77 54 3e d3 d3 a4 01 e7    part 0 : 0xb1b044c4 -> 0x085fc3ac (+ 0x56af7ee8)
 (EIP 0x00000dba): ac c3 5f 08 d4 bd fa f9 77 54 3e d3 d3 a4 01 e7    part 1 : 0x4952e983 -> 0xf9fabdd4 (+ 0xb0a7d451)
```

```
(EIP 0x002093cc): ac c3 5f 08 d4 bd fa f9 12 a3 87 87 d3 a4 01 e7    part 2 : 0xd33e5477 -> 0x8787a312 (+ 0xb4494e9b)
(EIP 0x00277459): ac c3 5f 08 d4 bd fa f9 12 a3 87 87 56 a2 3c 8a    part 3 : 0xe701a4d3 -> 0x8a3ca256 (+ 0xa33afd83)
```

Annexe D

Script de log des instructions exécutées

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

#define MASK(X) ((1<<(32-(X))) - 1)
#define P_MOD(X, Y) ((X) % (Y))
#define P_MUL(X, F1, F2, S) (((X) - (((((X) * (F1)) >> (S)) & MASK(S)) * (F2)))
& 0xffff)

unsigned int fct(unsigned int IV, unsigned int k, unsigned int p1, unsigned int
p2, unsigned int s1, unsigned int s2, unsigned int s3, unsigned int s4)
{
	unsigned int t1 = (p1 << s1) ^ (p2 >> s3);
	unsigned int t2 = (p1 >> s2) ^ (p2 << s4);
	unsigned int t3 = t1 + t2;
	unsigned int t4 = (IV ^ p1) + (k ^ p2);
	unsigned int t5 = t4 ^ t3;
	return t5;
}

void invertPart1(char * passphrase)
{
	unsigned int key[] = {0x4b9e2731, 0xaab1bbe6, 0xe4e4cfe3, 0x5baa36bd};
	unsigned int pass[] = {0x64636261, 0x68676665, 0x6c6b6a69, 0x706f6e6d};

	memcpy(pass, passphrase, 16);

	unsigned int IVs[19] = {
		0x7f434625, 0xa1882186, 0xa882a3fa, 0xe52d841a,
		0xce2684cb, 0xd82420e6, 0xa4e8f9fe, 0xb45692eb,
		0xa1010fea, 0xbf6470c8, 0x2e36ba18, 0xdffb67a8,
		0x436a9322, 0x79e3ad8e, 0x6a5f7a29, 0x1bccc67a,
		0xa1b52732, 0x11a35424, 0x693fa863};

	unsigned int keyOffset[] = {
		1, 0, 3, 2,		// 0
		1, 0, 3, 2,		// 1
		2, 3, 0, 1,		// 2
		2, 3, 0, 1,		// 3
		2, 3, 0, 1,		// 4
		1, 0, 3, 2,		// 5
		3, 2, 1, 0,		// 6
		2, 3, 0, 1,		// 7
		2, 3, 0, 1,		// 8
		2, 3, 0, 1,		// 9
		2, 3, 0, 1,		// 10
		2, 3, 0, 1,		// 11
		0, 1, 2, 3,		// 12
		3, 2, 1, 0,		// 13
		2, 3, 0, 1,		// 14
		2, 3, 0, 1,		// 15
		0, 1, 2, 3,		// 16
		1, 0, 3, 2,		// 17
		0, 1, 2, 3		// 18
	};

	unsigned int keyParts[] = {
		1, 3, // 0
		1, 3, // 1
		1, 3, // 2
		1, 3, // 3
		1, 3, // 4
		1, 3, // 5
		1, 3, // 6
		1, 3, // 7
		1, 3, // 8
		1, 3, // 9
```

```
                1, 3, // 10
                1, 3, // 11
                1, 3, // 12
                1, 3, // 13
                1, 3, // 14
                1, 3, // 15
                1, 3, // 16
                1, 3, // 17
                1, 3, // 18
        };

        int p;
        for (p=18; p>=0; p--)
        {
                unsigned int IV = IVs[p];

                pass[3] -= fct(IV, key[keyOffset[p * 4 + 3]], pass[(keyParts[p * 2 +
0] + 3) & 3], pass[(keyParts[p * 2 + 1] + 3) & 3], 2, 3, 5, 4);
                pass[2] -= fct(IV, key[keyOffset[p * 4 + 2]], pass[(keyParts[p * 2 +
0] + 2) & 3], pass[(keyParts[p * 2 + 1] + 2) & 3], 2, 3, 5, 4);
                pass[1] -= fct(IV, key[keyOffset[p * 4 + 1]], pass[(keyParts[p * 2 +
0] + 1) & 3], pass[(keyParts[p * 2 + 1] + 1) & 3], 2, 3, 5, 4);
                pass[0] -= fct(IV, key[keyOffset[p * 4 + 0]], pass[(keyParts[p * 2 +
0] + 0) & 3], pass[(keyParts[p * 2 + 1] & 3)], 2, 3, 5, 4);
        }

        memcpy(passphrase, pass, 16);
}

int main(int argc, char **argv)
{
        unsigned int expected[] = {
                0xcb3c6dc1, 0x7d35cccd, 0x3e3a55df, 0xef32abdb,
                0x3f29f1e3, 0x7b03a6c7, 0xf338cfda, 0x23097fd,
                0x893101e0, 0x3d0f7ede, 0xd70279d8, 0x633b5de
        };
        char pass[16] = {
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
        };

        int i;
        for (i=0; i<12; i++) expected[i] ^= 0xa5cc1a27;

        unsigned char *p = (unsigned char *)expected;
        for (i=0; i<48; i+=2)
        {
                unsigned char a = p[0];
                unsigned char b = p[1];
                *p++ = ~b;
                *p++ = ~a;
        }

        unsigned int x;
        unsigned short s1, s2, s3;

        unsigned short * p_s = (unsigned short *)pass;

        // part 0
        s1 = ((unsigned short *)expected)[8];
        s2 = ((unsigned short *)expected)[16];
        s3 = ((unsigned short *)expected)[0];

        for (x=0; x<0x10000; x++)
        {
                if (P_MUL(x, 0x82b9, 0x3eab, 29) == s1 &&
                        P_MOD(x, 0x3fc3) == s2 &&
                        P_MOD(x, 0x34f6) == s3)
                {
                        printf("part 0 : %4x\n", x);
                        *p_s++ = x;
                        break;
                }
```

```
}

// part 1
s1 = ((unsigned short *)expected)[17];
s2 = ((unsigned short *)expected)[9];
s3 = ((unsigned short *)expected)[1];

for (x=0; x<0x10000; x++)
{
	if (P_MUL(x, 0x349b, 0x4ddd, 28) == s1 &&
		P_MUL(x, 0xf7f3, 0x4214, 30) == s2 &&
		P_MOD(x, 0x23bf) == s3)
	{
		printf("part 1 : %4x\n", x);
		*p_s++ = x;
		break;
	}
}

// part 2
s1 = ((unsigned short *)expected)[10];
s2 = ((unsigned short *)expected)[2];
s3 = ((unsigned short *)expected)[18];

for (x=0; x<0x10000; x++)
{
	if (P_MOD(x, 0x2693) == s1 &&
		P_MOD(x, 0x15d0) == s2 &&
		P_MUL(x, 0xa0f, 0x32e7, 25) == s3)
	{
		printf("part 2 : %4x\n", x);
		*p_s++ = x;
		break;
	}
}

// part 3
s1 = ((unsigned short *)expected)[19];
s2 = ((unsigned short *)expected)[3];
s3 = ((unsigned short *)expected)[11];

for (x=0; x<0x10000; x++)
{
	if (P_MOD(x, 0x4903) == s1 &&
		P_MOD(x, 0x6423) == s2 &&
		P_MUL(x, 0x9b41, 0x34c4, 29) == s3)
	{
		printf("part 3 : %4x\n", x);
		*p_s++ = x;
		break;
	}
}

// part 4
s1 = ((unsigned short *)expected)[12];
s2 = ((unsigned short *)expected)[4];
s3 = ((unsigned short *)expected)[20];

for (x=0; x<0x10000; x++)
{
	if (P_MOD(x, 0x1189) == s1 &&
		P_MOD(x, 0xec6) == s2 &&
		P_MOD(x, 0x125) == s3)
	{
		printf("part 4 : %4x\n", x);
		*p_s++ = x;
		break;
	}
}

// part 5
s1 = ((unsigned short *)expected)[13];
```

```
	s2 = ((unsigned short *)expected)[21];
	s3 = ((unsigned short *)expected)[5];

	for (x=0; x<0x10000; x++)
	{
		if (P_MUL(x, 0x5c81, 0x588f, 29) == s1 &&
			P_MOD(x, 0x459d) == s2 &&
			P_MOD(x, 0x971) == s3)
		{
			printf("part 5 : %4x\n", x);
			*p_s++ = x;
			break;
		}
	}

	// part 6
	s1 = ((unsigned short *)expected)[14];
	s2 = ((unsigned short *)expected)[22];
	s3 = ((unsigned short *)expected)[6];

	for (x=0; x<0x10000; x++)
	{
		if (P_MUL(x, 0x2e69, 0x2c21, 27) == s1 &&
			P_MOD(x, 0xed5) == s2 &&
			P_MUL(x, 0x6612, 0xa08, 26) == s3)
		{
			printf("part 6 : %4x\n", x);
			*p_s++ = x;
			break;
		}
	}

	// part 6
	s1 = ((unsigned short *)expected)[15];
	s2 = ((unsigned short *)expected)[23];
	s3 = ((unsigned short *)expected)[7];

	for (x=0; x<0x10000; x++)
	{
		if (P_MUL(x, 0x5cc3, 0x585, 25) == s1 &&
			P_MUL(x, 0xa36f, 0x322, 25) == s2 &&
			P_MUL(x, 0x462b, 0x1d3, 23) == s3)
		{
			printf("part 7 : %4x\n", x);
			*p_s++ = x;
			break;
		}
	}

	invertPart1(pass);
	puts(pass);
	return 0;
}
```