Solution du challenge SSTIC 2009

Raphaël Rigo et Simon Marechal

1 Introduction

Cette année, pour la première fois, la conférence SSTIC (http://www.sstic.org) proposait un challenge de *reverse engineering*, à but principalement de compétition amicale. Seuls quelques lots étaient offerts aux cinq premiers.

L'objectif était de découvrir un mot de passe (qui sera désigné sous le terme de *passphrase* par la suite) caché dans un programme, conçu par Stéphane Duverger (EADS) et accessible à l'adresse suivante :

http://communaute.sstic.org/ChallengeSSTIC2009.

2 Ingénierie inverse

2.1 Découverte

La première partie consiste à comprendre le fonctionnement du challenge.

Celui-ci se présente sous la forme d'un fichier nommé `ChallengeSSTIC2009`, d'une taille de 1,44 Mo. La commande `file` permet de déterminer rapidement le type de fichier :

```
ChallengeSSTIC2009: x86 boot sector; GRand Unified Bootloader, (...)
```

On constate donc qu'il s'agit d'une disquette de démarrage pour PC. Une fois montée en *loopback*, on voit que celle-ci contient un fichier nommé `kernel.bin`, chargé par GRUB au démarrage. `file` nous permet une nouvelle fois d'identifier le type de fichier :

```
ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, (...)
```

Quelques indices supplémentaires, comme les valeurs *magiques* au début du fichier ELF, ainsi que les chaînes de caractères et son mode d'exécution nous permettent de dire qu'il s'agit *a priori* d'un programme démarrant en *ring 0*, en mode protégé, et qu'il ne s'agit pas d'un noyau de système d'exploitation connu.

Le premier contact avec le programme fourni est difficile, et laisse présager de quelques surprises à bas niveau : QEMU et certaines versions de Bochs refusent de le faire tourner. Il fonctionne par contre correctement avec VMWare et la version de Bochs supportée par le logiciel IDA. Une fois initialisé, le programme demande à l'utilisateur de saisir la passphrase. Si celle-ci est incorrecte, le message « FAIL » est affiché en ASCII art (capture 1).

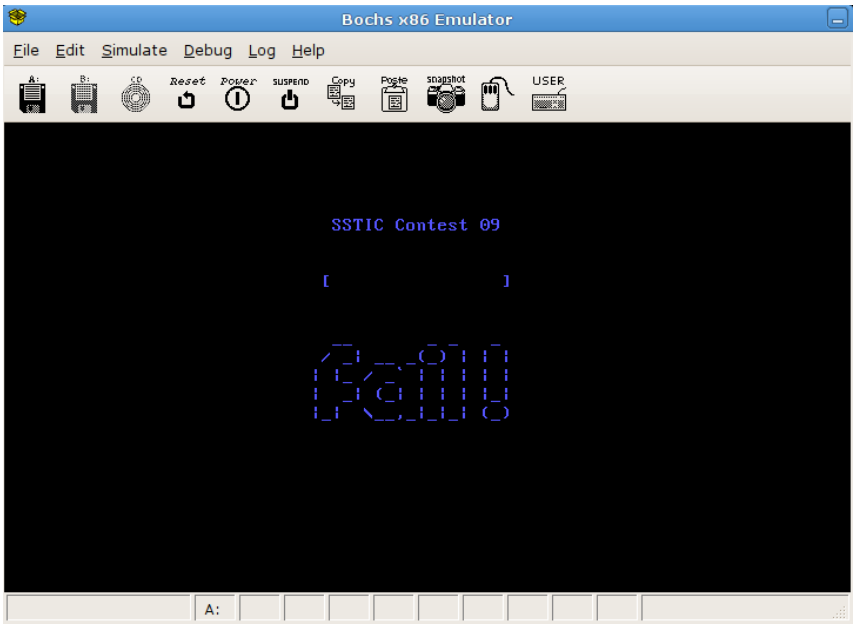

FIG. 1 – Échec

2.2 L'outillage

Les premiers pas concernant l'analyse se sont heurtés à un problème important : comme on le verra par la suite, il s'agit d'un programme complètement inhabituel. Il est en effet assez rare de rencontrer des programmes à reverser qui se comportent comme un OS. En effet, comment analyser dynamiquement la cible afin de mieux comprendre son fonctionnement ? Les outils classiques tels que IDA, OlldyDbg ou GDB sont principalement prévus pour analyser des programmes fonctionnant en mode utilisateur sur un OS déterminé. Heureusement, les machines virtuelles permettent de plus en plus de s'interfacer directement au niveau de l'émulation du processeur avec des outils familiers, souvent en utilisant des mécanismes interopérables : QEMU, Bochs et VMWare proposent ainsi des *stubs* GDB qui permettent de debugger (avec plus ou moins de réussite et de problèmes) les cibles en cours d'exécution.

Les premiers tests n'ont pas permis de réveler d'outils adaptés, d'autant plus que la première version de Bochs testée n'était pas compatible avec le challenge et que les autres implémentations ne semblaient pas satisfaisantes. Ensuite, après avoir installé une version de Bochs fonctionnelle, il s'est avéré que le support *stub* GDB fonctionnait très mal. Finalement, les solutions suivantes ont été utilisées :

- IDA pour l'analyse statique et dynamique avec le support Bochs intégré[1] ;
- un module d'instrumentation spécifique compilé avec Bochs pour les besoins de traçage plus lourds.

2.3 Début d'analyse statique

L'analyse statique a été réalisée dans le logiciel IDA. Cependant, un coup d'œil superficiel montre que la logique de validation de la phrase clef n'est pas immédiatement apparente, et qu'il va falloir retrouver les zones de code clefs.

Les deux principaux objectifs sont :

- localiser la lecture de la passphrase ;
- localiser la validation de la passphrase.

La lecture de la passphrase est assez simple à trouver. En effet, la référence à la chaîne de caractères *SSTIC Contest 2009*, qui est affichée à l'écran (capture 1), permet de retrouver le code exécuté après toute la phase d'initialisation du système. En regardant le code exécuté juste après (fonction à l'adresse `0x20006D`), on peut observer des lectures sur le port 0x60, qui correspond au clavier, ainsi qu'une boucle se terminant à la seizième itération, ce qui correspond à la longueur maximale de la passphrase.

À partir de ce point, l'analyse statique devient plus difficile : on peut voir la fonction à 0x200DE4 qui va mo-

[1] On notera l'inélégance de cette solution, qui écrit et lis sur la console de débug pour communiquer.

difier la fréquence du timer en fonction d'un *checksum* calculé à partir de la passphrase. La fonction à 0x200B94 n'est pas claire non plus, avec une instruction `lldt ax` sans utilité évidente. Cette partie est d'autant plus étonnante qu'ensuite le programme boucle sur des appels à la fonction de lecture de la passphrase. La vérification de celle-ci est donc réalisée d'une manière non immédiatement accessible.

2.4 Analyse dynamique

Comme indiqué précedemment, la solution la plus directe d'analyse dynamique est l'utilisation d'IDA 5.4, avec le support intégré de Bochs. En effet, IDA peut directement intéragir avec le simulateur pour débugger et tracer dynamiquement la cible[2]. Ceci correspond parfaitement à nos besoins pour l'analyse du challenge.

Découverte de la fonction de vérification

Vu que le chemin d'exécution ne permet pas facilement de trouver cette vérification, il faut utiliser un moyen alternatif. L'étude des chaînes du programme ne met pas en évidence la présence de l'ASCII art affiché, il doit donc être construit à partir de données de position. Il faut donc trouver un moyen de mettre un point d'arrêt sur les routines d'affichage. Or, l'une d'elle est utilisée dans la boucle de lecture de la passphrase, pour afficher le caractère tapé : 0x201868, qui n'est qu'un wrapper pour 0x2019B3.

Le plus simple est donc de mettre un breakpoint sur cette fonction et d'analyser les fonctions l'appelant, tout en surveillant l'écran de la machine virtuelle. On constate alors que la fonction 0x200A34 est responsable de l'affichage de l'ASCII art et qu'elle est appelée depuis la fonction 0x200AD5, qui semble en effet comparer une zone mémoire (probablement dérivée de la passphrase), située à l'adresse 0x70000, à une autre stockée (mais cachée à l'aide d'un xor avec la valeur 0xA5CC1A27) dans la partie statique du binaire. Nous sommes donc a priori sur la bonne voie.

Handlers d'interruptions

Comme indiqué précédemment, le programme met en place un timer, qui va donc générer des interruptions. Il est donc intéressant d'étudier la liste des handlers d'interruptions.

Heureusement, IDA permet de lister aisément l'ensemble des handlers définis dans Bochs :

```
BOCHS>info idt 0 34
Interrupt Descriptor Table (base=0x00000000003222a8, limit=2047):
IDT[0x00]=32-Bit Interrupt Gate target=0x0008:0x002006f0, DPL=0
IDT[0x01]=32-Bit Interrupt Gate target=0x0008:0x00200700, DPL=0
IDT[0x02]=32-Bit Interrupt Gate target=0x0008:0x00200710, DPL=0
IDT[0x03]=32-Bit Interrupt Gate target=0x0008:0x00200720, DPL=0
IDT[0x04]=32-Bit Interrupt Gate target=0x0008:0x00200730, DPL=0
...
IDT[0x21]=32-Bit Interrupt Gate target=0x0008:0x00200900, DPL=0
IDT[0x22]=??? descriptor hi=0x00000000, lo=0x00000000
```

On voit que 33 handlers sont définis. Une étude rapide montre qu'ils redirigent tous le flot d'exécution vers 0x200694 après avoir mis sur la pile le numéro de l'interruption. Toutes les interruptions sont donc traitées par le même code, en 0x200694.

Le code du traitement lui-même est intéressant :

```
LOAD:002006A8 mov     edx, [esp+3Ch]
LOAD:002006AC mov     [esp-4], esp
LOAD:002006B0 sub     esp, 4
```

[2] http://www.hex-rays.com/idapro/debugger/bochs_tut.pdf

```
LOAD:002006B3 test    edx, 20000h
LOAD:002006B9 jz      short loc_2006CD
```

Après la première ligne, `edx` contient la valeur des *eflags* du processeur et le test suivant vérifie si le mode VM est activé. Le bit concerné a comme valeur 1 lorsque le processeur était en mode virtual 8086 lorsque l'interruption est survenue. Cette éventualité sera étudiée par la suite. La branche « normale », suivie lors des interruptions survenant en mode protégé, va traiter l'interruption à l'aide d'un `switch(ebx)` (`ebx` contenant alors le numero de l'interruption) :

```
LOAD:002006CD lea     edx, off_321E60
LOAD:002006D3 call    dword ptr [edx+ebx*4]
```

mais le tableau de pointeurs en 0x321E60 n'est pas directement exploitable car on ne connait pas la valeur des index utilisés. De plus, les vrais pointeurs sont noyés au milieu d'autres pointant vers des endroits aléatoires du code.

Traitement de la passphrase

Afin de localiser plus exactement les opérations effectuées sur la passphrase, il est intéressant de poser un point d'arrêt déclenché lors d'un accès mémoire en lecture sur celle-ci. La passphrase est stockée à l'adresse 0x1FFFD1. On mettra également un breakpoint à l'accès au pointeur vers la passphrase, à l'adresse 0x322AD0.

Ce breakpoint va être activé en 0x2DA70E (qui fait partie d'une fonction débutant à 0x2DA600) , où l'on peut voir que la passphrase est copiée dans une nouvelle zone mémoire 0x60000. La fonction qui réalise cette copie initialise également d'autres valeurs à diverses adresses. En traçant jusqu'à la fonction appelante, on constate qu'il s'agit d'un handler d'interruption (en mode protégé) et que, `ebx` valant 13, il s'agit d'une exception de type *Global Protection Fault*, qui est déclenchée (comme on le voit en traçant après le `iret`) par l'instruction `lldt ax` observée précedemment (voir 2.3), qui utilise un selecteur de segment invalide (*cf* manuel Intel 3A, 3.4.2) : l'index est de 5 alors que la GDT ne comporte que 3 segments.

```
BOCHS>info gdt
Global Descriptor Table (base=0x0000000000322288, limit=31):
GDT[0x00]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x01]=Code segment, laddr=00000000, limit=fffff * 4Kbytes, X/R, Accessed, 32bit
GDT[0x02]=Data segment, laddr=00000000, limit=fffff * 4Kbytes, R/W, Accessed
GDT[0x03]=32-Bit TSS (Busy) at 0x00322aa8, length 0x02089
```

Si on trace à partir de cette fonction `lldt`, on observe que la fonction suivante est exécutée :

```
LOAD:00200680 sub_200680 proc near                    ; CODE XREF: sub_200B94+8F
LOAD:00200680                                         ; sub_200D12+25
LOAD:00200680 push    ebp
LOAD:00200681 mov     [eax], esp
LOAD:00200683 mov     esp, [edx]
LOAD:00200685 pop     ebp
LOAD:00200686 retn
LOAD:00200686 sub_200680 endp
```

Cette fonction est très inhabituelle : l'instruction `mov esp, [edx]` va en effet modifier la position de la pile, et ce juste avant un `ret`. Le flot d'exécution va donc ici être redirigé. Voici le contenu de la pile juste avant le retour :

```
00326ADC dd    2006D9h ; ihandler:loc_2006D9
```

Le flot d'exécution retourne donc à la fin du handler d'interruption !

```
LOAD:002006D9 add     esp, 2
LOAD:002006DC db      66h
LOAD:002006DC pop     ss
LOAD:002006DE db      66h
LOAD:002006DE pop     gs
LOAD:002006E1
LOAD:002006E1 loc_2006E1:                             ; DATA XREF: LOAD:00300AC4
LOAD:002006E1 db      66h
LOAD:002006E1 pop     fs
LOAD:002006E4 db      66h
LOAD:002006E4 pop     es
LOAD:002006E6 db      66h
LOAD:002006E6 pop     ds
LOAD:002006E8 assume ds:LOAD
LOAD:002006E8 popa
LOAD:002006E9 add     esp, 8
LOAD:002006EC iret
```

On constate que tous les registres sont restaurés depuis la pile, et que l'on sort finalement de l'interruption. Il est donc extrèmement important de découvrir *où* l'on va arriver après l'instruction `iret`.

En l'occurence, après le `iret`, `eip` vaut 0x15D9 et le désassemblage dans IDA semble incorrect. En regardant la valeur des *eflags* on voit :

```
id vip vif ac VM rf nt IOPL=0 of df IF tf sf zf af pf cf
```

Le flag **VM** est activé, sous somme donc en mode Virtual 8086 !

Partie Virtual 8086

Afin d'avoir un désassemblage correct, il est nécessaire de créer un segment 16 bits dans IDA. On pourra ainsi le capturer afin d'y avoir accès lors de l'analyse statique.

Le code 16bits commence ainsi :

```
0:15D9 int     0C0h
0:15DB int     21h
0:15DD mov     dx, 364h
0:15E0 out     dx, al
0:15E1 int     88h
0:15E3 int     5Fh
0:15E5 int     98h
0:15E7 mov     dx, 7C46h
0:15EA out     dx, al
0:15EB mov     dx, 5C5Eh
0:15EE out     dx, al
...
0:15F3 mov     dx, 9856h
0:15F6 in      al, dx
```

L'ensemble du code 16bits est constitué des instructions `in`, `out`, `int`, `mov dx`. Il est essentiel pour comprendre la suite de connaître le fonctionnement du traitement des interruptions en mode virtual 8086 (qui était bien entendu inconnus avant ce challenge). Pour le voir de façon pratique, il faut faire suivre le flot d'exécution sur chaque instruction.

On peut alors différencier trois comportements distincts :
- arrivée dans un handler d'interruption 16 bits ;
- arrivée dans le handler d'exception GPF en mode protégé (sur un `int` ou un `in/out`) ;
- exécution normale de l'instruction.

Ici, les choses se compliquent. Le lecteur intéressé pourra étudier le chapitre 15.3.3 du manuel Intel (3A). Ici nous sommes dans le mode où la décision de gestion des interruptions est faite à l'aide de la table de bits présente dans la TSS (`TSS + iomap_offset - 32`) : certaines sont traitées directement en 16 bits, les autres par le handler de GPF en mode protégé.

Afin de savoir quelles interruptions sont traitées en 16 bits, il faut donc dumper à la fois l'IDT et la map correspondante. Pour dumper l'IDT, rien de plus simple : elle est située à l'adresse physique 0 en mémoire et chaque handler correspond à un mot de 32 bits. On dumpera donc la mémoire physique de la VM, par exemple en faisant un suspend dans Bochs. Pour la TSS, la commande `x` a été utilisée dans IDA.

Ensuite un petit script ruby permet de lister les handlers 16 bits qui seront effectivement utilisés :

```
ints = [0x9F, 0x24, 0x41, 0x84, 0x41, 0x0C, 0x54, 0x3, 0xCC, 0x0, 0x58, 0xEC, 0x0D,
        0xCD, 0x42, 0x3A, 0x56, 0x20, 0x84, 0x84, 0x4, 0xC9, 0x48, 0x1A, 0x10, 0x5,
        0x20, 0x11, 0x10, 0x50, 0x12, 0x8]
handlers = [0x00000977, 0x0000078D, 0x00000438, 0x0000057A, 0x0000085E, 0x00000BFB,
            0x0000041C, 0x00000892, 0x0000096D, 0x000004E4, 0x00000BB7, 0x000008BB,
            0x00000AA0, 0x00000B05, 0x000012C3, 0x00001313, 0x0000044D, 0x000009E5,
            ...
            0x00000455, 0x00000496, 0x00000545, 0x0000082A, 0x00000B5C, 0x000004C5]
n = 0;
ints.each do |inter|
        (0..7).each do |s|
                if (inter>>s)&1 == 0
                        printf "int %02X : %04X\n", n, handlers[n]
                end
                n+=1
        end
end
```

Maintenant se pose le problème du handler en mode protégé, dont le traitement est réalisé par la fonction 0x200C60. L'analyse montre que ce dernier va exécuter les instructions suivantes :

```
LOAD:00200CF0 mov     edx, dword ptr table2_cd[eax*4]
...
LOAD:00200CFD call    edx ;
```

Un peu d'analyse dynamique montre que les valeurs de `edx` changent à chaque passage et que les index utilisés pour la table `table2_cd` sont peu nombreux comparés à sa taille. Il faudra donc trouver un moyen de récupérer les adresses des fonctions effectivement appelées.

Le détail de ces fonctions montre qu'elles agissent sur différentes zones mémoire, dont les bases sont les suivantes : 0x60000, 0x70000, 0x80000, 0x90000. On aura pu noter en traçant les handlers 16 bits qu'ils utilisent les registres selecteurs de segments pour atteindre ces adresses, et que des valeurs étonamment similaires sont initialisées dans la fonction 0x2DA600 précedemment évoquée (voir 2.4).

Résumé de la situation

Ce que nous avons pu voir correspond donc à une suite d'instructions en mode virtual 8086, qui, par le biais d'interruptions, déclenchent des routines spécifiques modifiant certaines zones mémoire, qui semblent contenir un contexte d'exécution. Bref, plus ou moins le comportement d'une machine virtuelle.

L'objectif va donc être de reconstruire l'ensemble de la chaîne des appels à ces routines afin de pouvoir comprendre quel algorithme est utilisé pour la vérification de la passphrase.

3 Reconstruction de l'algorithme et inversion

3.1 Identification des routines

Points communs entre les handlers en mode protégé

Une première observation a été que toutes les fonctions 32 bit invoquées prenaient en paramètre un pointeur vers un tableau de DWORDS. Ce dernier étant toujours le même, il est apparu comme le contexte d'exécution d'une machine virtuelle.

Instrumentation de Bochs

L'idéal aurait été de pouvoir écrire un script directement dans IDA pour tracer le fonctionnement. Malheureusement il semblerait que l'API IDC ou même Python ne le gèrent pas encore, sans parler des limitation provenant de la méthode qu'utilise IDA pour contrôler Bochs, source de limitations particulièrement frustrantes (tapez deux fois F8 rapidement pour voir). Il a donc fallu instrumenter Bochs. En effet, cette fonctionnalité est directement accessible dans le code de Bochs (fichier instrument.cc et option de configuration appropriée) et permet de tracer tout et n'importe quoi.

La liste des handlers en mode virtual 8086 a été récupérée et utilisée pour tracer l'état des registres lorsqu'ils sont invoqués. De même, pour le mode protégé, l'adresse de la fonction appelée par le `call edx` est journalisée, ainsi que l'état du contexte d'exécution.

```
switch(eip) {
...
case 0x200cfd:
        pcontext = BX_CPU(cpu)->gen_reg[0].dword.erx;
        BX_MEM(cpu)->dbg_fetch_mem(BX_CPU(cpu), pcontext , sizeof(context),
                (Bit8u*) &context);
        fprintf(logfile, "%.4d edx=0x%.8x ",
                count, BX_CPU(cpu)->gen_reg[2].dword.erx);
        fprintf(logfile, "%.8x %.8x %.8x %.8x %.8x %.8x %.8x",
                ((unsigned int *)&context)[10],
                ((unsigned int *)&context)[7],
                ((unsigned int *)&context)[9],
                ((unsigned int *)&context)[8],
                ((unsigned int *)&context)[4],
                ((unsigned int *)&context)[3],
                ((unsigned int *)&context)[5]);
        SHOWMEM;
        count++;
        fprintf(logfile, "\n");
        fflush(logfile);
        break;

case 0x1521:
case 0x455:
```

```
case 0x5c1:
case 0x665:
case 0x82d:
case 0x8bb:
case 0x973:
case 0xa17:
case 0xabc:
...
case 0x4c5:
        fprintf(logfile, "%.4d eip=0x%x\t", count, eip);
        fprintf(logfile, "%.8x %.8x %.8x %.8x %.8x %.8x %.8x",
                        BX_CPU(cpu)->gen_reg[BX_32BIT_REG_EAX].dword.erx,
                        BX_CPU(cpu)->gen_reg[BX_32BIT_REG_EBX].dword.erx,
                        BX_CPU(cpu)->gen_reg[BX_32BIT_REG_ECX].dword.erx,
                        BX_CPU(cpu)->gen_reg[BX_32BIT_REG_EDX].dword.erx,
                        BX_CPU(cpu)->gen_reg[BX_32BIT_REG_ESI].dword.erx,
                        BX_CPU(cpu)->gen_reg[BX_32BIT_REG_EDI].dword.erx,
                        BX_CPU(cpu)->gen_reg[BX_32BIT_REG_EBP].dword.erx);
        SHOWMEM;
        fprintf(logfile, "\n");
        fflush(logfile);
        count++;
        break;
...
}
```

En journalisant tous les appels aux handlers d'interruptions il a été possible d'avoir la liste et l'ordre d'exécution des fonctions concernées. Le code présenté n'est qu'un extrait de la journalisation réelle. Il est entre autre nécessaire de ne pas journaliser les appels avant le passage en mode virtual 8086 pour isoler les portions utiles.

Decompilation des routines de traitement de la passphrase

Maintenant que toutes les adresses des routines, en 16 bits et en 32 bits, sont connues, il est possible d'utiliser une fonctionnalité expérimentale de l'excellent metasm[3] : la décompilation. Hex-Rays aurait pu être envisagé, mais il ne gère pas le code 16 bits et n'est pas scriptable.

Il suffit d'exécuter la commande `ruby samples/disassemble.rb -e 'dasm.cpu.size=16' -decompile -no-data-trace -no-data  /temp/sstic2/memory.ram 0x1521 0x455 ...` pour obtenir le résultat suivant :

```
int entrypoint_1521h(void)
{
        register __int32 edi;
        register __int32 ecx;
        register __int32 *esi;
        static __int32 segment_base_fs;
        register __int32 ebp;
        __int32 var_2;
        __int32 var_6;
        __int32 var_A;
        var_2 = edi;
        *((__int32*)esi) = (((*((__int32*)(segment_base_fs + esi + ebp)) ^ ecx)
        + (edi ^ esi[1])) ^ ((((esi[1] >> 3) & 0x1FFFFFFF) ^ (ecx << 4))
        + ((esi[1] << 2) ^ ((ecx >> 5) & 0x7FFFFFF)))) + *((__int32*)esi);
        var_6 = ecx;
```

[3] http://metasm.cr0.org

```
        var_A = esi[1];
        asm ("iret");
}
...
```

De la même manière, pour les handlers 32bits (en enlevant l'option `dasm.cpu.size` bien sûr) :

```
__int32 *entrypoint_263219h(__int32 *arg_0)
{
        __int32 var_4;
        __int32 *var_8;
        var_8 = (__int32*)((arg_0[4] & 0xFFFF) + (arg_0[19] << 4));
        var_4 = (arg_0[20] << 4) + (arg_0[4] & 0xFFFF);
        arg_0[7] = var_8[2];
        var_8[1] += ((((arg_0[7] >> 3) & 0x1FFFFFFF) ^ (arg_0[9] << 4)) +
                    (((arg_0[9] >> 5) & 0x7FFFFFF) ^ (4 * arg_0[7]))) ^
                    ((*((__int32*)(((arg_0[5] << 2) ^ 4) + var_4)) ^ arg_0[9])
                      + (arg_0[7] ^ arg_0[3]));
        arg_0[9] = var_8[1];
        return arg_0;
}
```

3.2 Simplification

Plusieurs observation sont alors possibles :
- comme indiqué précedemment, les handlers 32 bits travaillent avec un tableau de DWORDS, noté `arg_0` dans l'extrait précédent ;
- les fonctions 32 bits et 16 bits semblent très proches, comme on peut le voir dans les deux extraits précédents ;
- l'état des registres en mode 16 bits est semblable à l'état de ce tableau de DWORDS lorsqu'on passe d'un mode à l'autre.

Ces fonctions travaillent donc toutes sur les même données, et il existe un système pour passer de la représentation « registres » à la représentation « contexte ». Tout se passe en fait de la façon suivante :
- lorsqu'on entre dans le handler 32 bits (0x200694), la pile pointe vers une zone du contexte qui contient la valeur des registres génériques ;
- l'instruction `pusha` copie la valeur des registres dans le contexte ;
- l'adresse d'une fonction de traitement est calculée et celle ci est appelée (`call edx`) ;
- cette fonction travaille sur le contexte ;
- a la fin du handler d'interuption, l'instruction `popa` copie la valeur du contexte dans les registres.

Il est donc possible d'associer directement une zone mémoire du contexte en mode 32 bits à un registre en mode 16 bits.

Il faut donc maintenant analyser des centaines de fonctions qui modifient l'état des registres et de la mémoire. Il est donc nécessaire de les simplifier. Une petite étude rapide nous montre que beaucoup de ces fonctions sont très semblables, seuls quelques paramètres changent. Il est donc intéressant d'automatiser la création de fonctions paramétrées.

De plus, le traçage du contenu des zones mémoire permet de mettre en évidence le fait qu'à partir d'un certain moment les fonctions appelées ne modifient plus la zone mémoire utilisée pour la comparaison finale. Les fonctions suivantes seront ignorées.

Handlers 16bits

La version décompilée n'étant pas très exploitable, un script Perl a été écrit pour transformer l'assembleur en fonctions utilisables dans un programme en C. Il donne par exemple :

```
void handler_1521h(unsigned int * ctx)
{
        EBX = ((unsigned int *)PASS)[1];
        EAX = ((unsigned int *)FSMEM)[0^EBP];
        EAX ^= ECX;
        EAX += (EDI ^ EBX);
        EAX ^= ((EBX<<2) ^ (ECX>>5)) + ((EBX>>3) ^ (ECX<<4));
        ECX = EAX;
        EAX = ((unsigned int *)PASS)[0];
        ECX += EAX;
        ((unsigned int *)PASS)[0] = ECX;
}
```

Les valeurs telles que `EAX` ou `PASS` sont des macros pointant vers les zones appropriées du contexte. Cette version est nettement plus lisible que l'originale.

Handlers 32bits

Les handlers 32bits décompilés par metasm sont simplifiés à l'aide de remplacements par expressions rationnelles afin d'utiliser les mêmes macros que les handlers 16 bits. Par exemple :

```
int entrypoint_263219h(unsigned int *ctx)
{
        int var_4;
        int *var_8;

        var_8 = (((unsigned int *)PASS));
        var_4 = (((unsigned int *)FSMEM));
        EBX = var_8[2];
        var_8[1] += ((((EBX >> 3)) ^ (ECX << 4))
                     + (((ECX >> 5)) ^ (4 * EBX)))
                     ^ ((*((int*)((EBP ^ 1)*4 + var_4)) ^ ECX)
                       + (EBX ^ EDI));
        ECX = var_8[1];
        return ctx;
}
```

On remarquera que les deux fonctions ainsi obtenues sont très semblables.

Reconstitution de la logique

Un source en C émulant de bout en bout le calcul de la valeur utilisée pour la comparaison finale a ainsi généré. La routine principale contient l'initialisation des valeurs mais également la succession des appels aux fonctions décompilées :

```
handler_b29h(ctx); // int c0
handler_1521h(ctx); // int 21h
entrypoint_263219h(ctx);
inFF(ctx);  // int 88h
entrypoint_2778b1h(ctx); // int 7f
handler_f7ch(ctx);
entrypoint_22015eh(ctx);
entrypoint_2591f8h(ctx);
entrypoint_2587aeh(ctx);
```

L'étude des simillitudes entre les fonctions a permis de simplifier largement cette succession d'appels. On constate également que les handlers 16 bits et les 32 bits implémentent différemment les mêmes opérations, ce qui permet d'unifier plusieurs fonctions.

```
EDI = 0x7f434625; EBP = 1;
rotB(ctx, 0); //change EBX,EAX,ECX,PASS[0]
rotB32(ctx,2);
inFF(ctx);  // int 88h
rotB32(ctx,3);
rotA(ctx); //change EBX, EAX, ECX
incPass32(ctx);
EDI = 0xA1882186; EBP = 1;
```

Toute cette liste a ensuite été simplifiée manuellement, en supprimant les fonctions n'influant pas sur les calculs finaux,et en groupant les autres. Toute cette étape de reconstitution de la logique, bien que conceptuellement simple, a été particulièrement laborieuse.

3.3 Logique simplifiée, en C

Une fois tout simplifié, l'algorithme est le suivant :

```
void do_crypt(unsigned int * ctx, unsigned int edi, unsigned int ebp)
{
        unsigned int v, sum, i;
        unsigned int *password = (unsigned int *) PASS;
        unsigned int *k = (unsigned int *) FSMEM;

        sum = ECX;
        for(i = 0; i < 4; i++) {
                v = password[(i+1)&3];
                password[i]+=((k[i^ebp]^sum)+(edi^v))^((v<<2)^(sum>>5))+((v>>3)^(sum<<4));
                sum = password[i];
        }
        ECX = sum;
}

void rot16_32(unsigned int * ctx, unsigned int dec)
{
        unsigned short * ptr;
        unsigned short tmp;

        ptr = ESMEM + dec;
        tmp =  ~ ( (*ptr>>8) | (*ptr<<8) );
        *ptr = tmp;
}

void rem32(unsigned int *ctx, unsigned int pass, unsigned int esmem, unsigned short num)
{
        unsigned short *esptr = esmem + ESMEM;

        *esptr = *((unsigned short *)(PASS + pass)) % num;
        return;
}

do_crypt(ctx, 0x7f434625, 1);
do_crypt(ctx, 0xA1882186, 1);
```

```
do_crypt(ctx, 0xA882A3FA, 2);
do_crypt(ctx, 0xE52D841A, 2);
do_crypt(ctx, 0xce2684cb, 2);
do_crypt(ctx, 0xD82420E6, 1);
do_crypt(ctx, 0xA4E8F9FE, 3);
do_crypt(ctx, 0xb45692eb, 2);
do_crypt(ctx, 0xA1010FEA, 2);
do_crypt(ctx, 0xBF6470C8, 2);
do_crypt(ctx, 0x2E36BA18, 2);
do_crypt(ctx, 0xDFFB67A8, 2);
do_crypt(ctx, 0x436A9322, 0);
do_crypt(ctx, 0x79E3AD8E, 3);
do_crypt(ctx, 0x6A5F7A29, 2);
do_crypt(ctx, 0x1BCCC67A, 2);
do_crypt(ctx, 0xa1b52732, 0);
do_crypt(ctx, 0x11A35424, 1);
do_crypt(ctx, 0x693fa863, 0);

rem32(ctx,  0 ,  0 , 13558);
rem32(ctx,  0 , 16 , 16043);
rem32(ctx,  0 , 32 , 16323);
rem32(ctx,  2 ,  2 , 9151);
rem32(ctx,  2 , 18 , 16916);
rem32(ctx,  2 , 34 , 19933);
rem32(ctx,  4 ,  4 , 5584);
rem32(ctx,  4 , 20 , 9875);
rem32(ctx,  4 , 36 , 13031);
rem32(ctx,  6 ,  6 , 25635);
rem32(ctx,  6 , 22 , 13508);
rem32(ctx,  6 , 38 , 18691);
rem32(ctx,  8 ,  8 , 3782);
rem32(ctx,  8 , 24 , 4489);
rem32(ctx,  8 , 40 , 293);
rem32(ctx, 10 , 10 , 2417);
rem32(ctx, 10 , 26 , 22671);
rem32(ctx, 10 , 42 , 17821);
rem32(ctx, 12 , 12 , 2568);
rem32(ctx, 12 , 28 , 11297);
rem32(ctx, 12 , 44 , 3797);
rem32(ctx, 14 , 14 , 467);
rem32(ctx, 14 , 30 , 1413);
rem32(ctx, 14 , 46 , 802);


for (j = 0; j < 48; j += 2) {
        rot16_32(ctx, j);
}
```

3.4 Inversion de la logique

Une fois la logique décrite comme précédemment, il devient très simple de l'inverser pour partir de la valeur finale attendue et retrouver la passphrase.

Finalement, l'ensemble des opérations est assez simple, et l'inversion est également facile, à partir de la valeur dechiffrée du binaire :

- inverser la fonction `rot16_32`;
- inverser la fonction `rem32`;
- inverser la fonction `do_crypt`.

La fonction `rot16_32`, lorsqu'on l'applique deux fois sur une même zone mémoire, ne modifie pas cette

dernière. Elle est donc très facilement inversible, puisqu'elle est son propre inverse.

Le problème des restes est normalement résolu à l'aide du théorème des restes chinois (ce qui explique l'indice visible lorsqu'on liste directement les chaînes de caractères présentes dans le programme[4]). Il est cependant bien plus simple de retrouver le résultat en force brute :

```
void solve_pass(unsigned int *ctx, unsigned int pass,
                unsigned short un, unsigned short deux, unsigned short trois)
{
        int brute,i ;
        unsigned short reste1, reste2, reste3;

        reste1 = *((unsigned short *)(ESMEM + (pass + 0)));
        reste2 = *((unsigned short *)(ESMEM + (pass + 16)));
        reste3 = *((unsigned short *)(ESMEM + (pass + 32)));

        while( (brute%un != reste1)
            || (brute%deux != reste2)
            || (brute%trois != reste3)) {
                brute++;
        }

        ((unsigned short *)(PASS + pass))[0] = brute;
        return;
}
```

Quant à la fonction `do_crypt`, il suffit d'inverser l'ordre d'application des opérations, et de transformer l'addition en soustraction :

```
void do_decrypt(unsigned int * ctx, unsigned int edi, unsigned int ebp)
{
    unsigned int v, sum;
    int i;
    unsigned int *password = (unsigned int *) PASS;
    unsigned int *k = (unsigned int *) FSMEM;

#define FUNC(SUM,V,I) (((k[I^ebp]^SUM)+(edi^V))^((V<<2)^(SUM>>5))+((V>>3)^(SUM<<4)))

    password[3] -= FUNC(password[2], password[0], 3) ;
    password[2] -= FUNC(password[1], password[3], 2) ;
    password[1] -= FUNC(password[0], password[2], 1) ;
    password[0] -= FUNC(password[3], password[1], 0) ;
    ECX = password[3];
}
```

Le code final permettant d'obtenir la solution est alors :

```
for (j = 0; j < 48; j += 2) {
        rot16_32(ctx, j);
}

solve_pass(ctx, 0, 13558, 16043, 16323);
solve_pass(ctx, 2, 9151, 16916, 19933);
solve_pass(ctx, 4, 5584, 9875, 13031);
```

[4] Il semblerait que pour de nombreux joueurs le challenge se soit arrêté là.

```
solve_pass(ctx, 6, 25635, 13508, 18691);
solve_pass(ctx, 8, 3782, 4489, 293);
solve_pass(ctx, 10, 2417, 22671, 17821);
solve_pass(ctx, 12, 2568, 11297, 3797);
solve_pass(ctx, 14, 467, 1413, 802);

do_decrypt(ctx, 0x693fa863, 0);
do_decrypt(ctx, 0x11A35424, 1);
do_decrypt(ctx, 0xa1b52732, 0);
do_decrypt(ctx, 0x1BCCC67A, 2);
do_decrypt(ctx, 0x6A5F7A29, 2);
do_decrypt(ctx, 0x79E3AD8E, 3);
do_decrypt(ctx, 0x436A9322, 0);
do_decrypt(ctx, 0xDFFB67A8, 2);
do_decrypt(ctx, 0x2E36BA18, 2);
do_decrypt(ctx, 0xBF6470C8, 2);
do_decrypt(ctx, 0xA1010FEA, 2);
do_decrypt(ctx, 0xb45692eb, 2);
do_decrypt(ctx, 0xA4E8F9FE, 3);
do_decrypt(ctx, 0xD82420E6, 1);
do_decrypt(ctx, 0xce2684cb, 2);
do_decrypt(ctx, 0xE52D841A, 2);
do_decrypt(ctx, 0xA882A3FA, 2);
do_decrypt(ctx, 0xA1882186, 1);
do_decrypt(ctx, 0x7f434625, 1);
```

La passphrase était donc : `Z7aw ?gW\=HL|Np_9` (capture 2).

FIG. 2 – C'est gagné !

4 CONCLUSION

4.1 *a posteriori*

Après avoir résolu le challenge, le principal reproche que l'on peut faire à la méthode utilisée est que la partie d'analyse et de simplification des fonctions a été en très grande partie manuelle et laborieuse. Il aurait été beaucoup plus intéressant, bien que potentiellement plus complexe, d'instrumenter metasm afin de faire une reconnaissance beaucoup plus automatisée des différentes fonctions.

Une autre approche aurait été de travailler sur les handlers 16 bits avant de s'attaquer aux handlers 32 bits. Ceci aurait permis de faire des simplifications immédiates, comme par exemple d'évacuer l'émulation des instructions `in` ne déclenchant pas d'interruptions, et ne modifiant que le registre `EAX`, qui n'est pas crucial pour la suite.

4.2 Intérêts du challenge

Ce challenge était intéressant sur plusieurs points. Il permet d'abord de découvrir ou de revoir certains aspects du fonctionnement des processeurs x86 : interruptions, mode virtual 8086, segmentation, TSS, ...

Il permet également de prendre en main les outils nécessaires à l'analyse de systèmes complets, ce qui peut être utile par exemple pour l'analyse de rootkits ou de systèmes embarqués (via l'utilisation de stubs GDB de QEMU par exemple), et surtout d'identifier les faiblesses de ces outils (le stub GDB de Bochs).

Enfin, l'analyse des handlers de la VM permet de développer des outils de reconnaissance et de se familiariser avec metasm[5].

[5] Cet outil ayant d'ailleurs été mis à jour à de nombreuses reprises durant le challenge, en particulier sur la partie 16 bits ...