Soluce

On a un grub et un noyau. On veut de quoi déboguer le noyau.

Bochs reboot en boucle dessus. Simics ne semble pas digérer certaines exceptions et Qemu part en vrille lors de la première interruption en mode *v8086*.
N'ayant pas de temps à perdre pour le Contest il faut trouver une autre alternative. Remplacer le grub 1 par une version 2 avec support d'un gdb serveur était une piste mais il s'est avéré que les dernière version de VMware 6 ont également le support gdb serveur. VMware émulant correctement le Contest de bout en bout s'avère être un bon chois.

Dans un premier temps, la maquette :

- gdb (le magnifique!)
- IDA (is gorgeous)
- Vmware Workstation 6

Vmware ne semble pas supporter les breaks hardwares, on le configure donc pour n'utiliser que les breaks soft avec la commande : *set can-use-hw-watchpoints 0*. En breakant (CTRL-C) lors de l'écran de saisie on tombe facilement sur le code de saisie des touches. En plaçant un watchpoint sur la zone contenant les données saisies on tombe sur un memcopy appelé depuis la fonction d'initialisation du premier contexte *v8086* en sub_2DA600 (appelé par un iret, c'est en ce point que qemu bug). Cette fonction place nos datas en 0x60000 (linéaires) et initialise une zone de 128 bits également comprenant des constantes xorées en run-time par `0x1337C0DE` :

```
key[4]={0x4b9e2731,0xaab1bbe6,0xe4e4cfe3,0x5baa36bd};
```

L'adresse en *v8086* est quand à elle calculée en :

```
mov     eax, offset dword_2DBDA0
mov     edx, eax
mov     eax, offset byte_2DA7C7
mov     ecx, edx
sub     ecx, eax
```

Soit : 0x15D9. On effectue alors un trace complet de l'execflow en partant de :

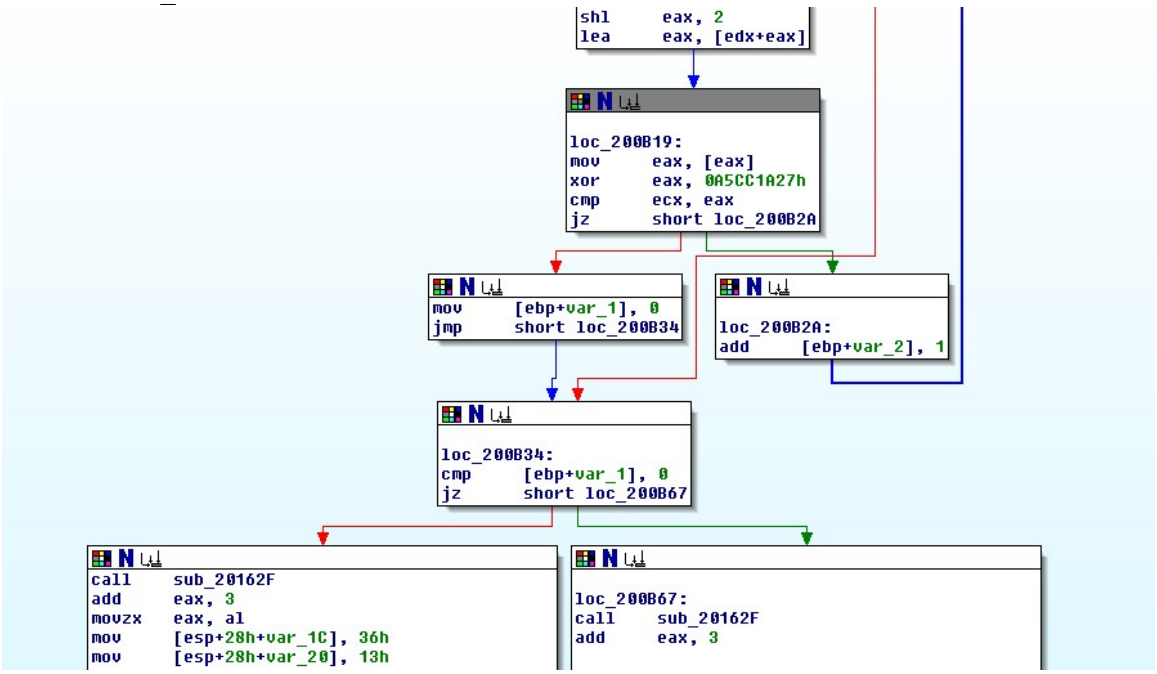
```
0x15d9: int     0xc0
```

Qui est la première instruction *v8086* exécutée.
On stoppe le tracer des les premiers affichages des caractères du message d'échec (FAIL).

Dans l'IDA on peut voir le message d'accueille :

```
002E17E0 aSsticContest09 db 'SSTIC_Contest_09',0
```

Servant d'argument à l'appelle "call sub_2018AC" effectuant cet affichage. En greppant la trace d'exécution précédemment obtenue avec l'adresse 2018AC on n'obtient rien. En revanche si l'on prend le seul appelle effectué dans la fonction sub_2018AC : call sub_2019B3 En greppant 2019B3 dans la trace d'exécution on y voie un appelle depuis la fonction sub_200A34 elle même appelée par la fonction "sub_200AD5".



La morphologie de "sub_200AD5" montre une comparaison sur 384 Bits de la zone linéaire générée en 0x70000 avec la zone constante en dword_2E11C0 xorée par `0A5CC1A27h`. En chargeant une version modifiée du kernel.bin inversant le saut de cette comparaison on valide rapidement qu'il s'agit du contrôle de validité du mot de passe (ou des données en découlant.) Le reste ne repose plus que sur les watchpoints afin d'observer la génération des données en 0x70000. Cette génération est lourdement noyée dans un flux de junk code (généralement annulant des zones inutiles en 0x90000). De plus, le code initialement en *v8086* a en parti été converti sous forme de VM. Le code *v8086* se contentant alors d'être invalide, tel qu'un (ou une int parfois) :

```
0x4231: mov     dx,0x666
0x4234: out     [dx],al
```

Le dx valant alors l'opcode de la VM, celle-ci seras alors émulé en mode protégé dans le handler d'exception /texttt« general protection fault » (13). Il est à noter qu'il ne s'agit pas réellement d'une VM mais plus d'un découpage de l'execflow

au sens où l'on ne retrouvera pas deux fois la même opcode.

On observe alors rapidement que les données en 0x70000 sont directement générée a partir de celle générées en 0x60000 et ceux a l'aide d'équations modulaire sur 16 Bits.

Nous avons deux types de calculs :

```
Avec :
0x60000:        0x36586a21      0x83801b69      0xf746490f      0x3554efb2
                A       B       C       D       E       F       G       H
```

```
0x1275: mov     ebx,0x0
0x127b: xor     dx,dx
0x127d: mov     ax,WORD PTR [si+2]
0x1280: mov     cx,0x23bf
0x1283: div     cx
0x1285: addr32 mov WORD PTR es:[edi+ebx*8+2],dx
0x128b: iret
```

Soit :

```
B%0x23bf = 0x0F91
```

Ou encore :

```
0x25c59b:       push    ebp
0x25c59c:       mov     ebp,esp
0x25c59e:       push    ebx
0x25c59f:       sub     esp,0x14
0x25c5a2:       mov     eax,DWORD PTR [ebp+8]
0x25c5a5:       mov     eax,DWORD PTR [eax+76]
0x25c5a8:       mov     edx,eax
0x25c5aa:       shl     edx,0x4
0x25c5ad:       mov     eax,DWORD PTR [ebp+8]
0x25c5b0:       mov     eax,DWORD PTR [eax+16]
0x25c5b3:       and     eax,0xffff
0x25c5b8:       lea     eax,[edx+eax]
0x25c5bb:       mov     DWORD PTR [ebp-12],eax
0x25c5be:       mov     eax,DWORD PTR [ebp+8]
0x25c5c1:       mov     eax,DWORD PTR [eax+72]
0x25c5c4:       mov     edx,eax
0x25c5c6:       shl     edx,0x4
0x25c5c9:       mov     eax,DWORD PTR [ebp+8]
0x25c5cc:       mov     eax,DWORD PTR_EDI [eax+12]
```

```
0x25c5cf:       and     eax,0xffff
0x25c5d4:       lea     eax,[edx+eax]
0x25c5d7:       add     eax,0x10
0x25c5da:       mov     DWORD PTR [ebp-8],eax
0x25c5dd:       mov     ecx,DWORD PTR [ebp-8]
0x25c5e0:       mov     eax,DWORD PTR [ebp-12]
0x25c5e3:       movzx   edx,WORD PTR [eax]
0x25c5e6:       movzx   eax,dx
0x25c5e9:       imul    eax,eax,0x82b9
0x25c5ef:       shr     eax,0x10
0x25c5f2:       mov     ebx,eax
0x25c5f4:       shr     bx,0xd
0x25c5f8:       mov     WORD PTR [ebp-22],bx
0x25c5fc:       imul    ax,WORD PTR [ebp-22],0x3eab
0x25c602:       mov     ebx,edx
0x25c604:       sub     bx,ax
0x25c607:       mov     WORD PTR [ebp-22],bx
0x25c60b:       movzx   eax,WORD PTR [ebp-22]
0x25c60f:       mov     WORD PTR [ecx],ax
0x25c612:       add     esp,0x14
0x25c615:       pop     ebx
0x25c616:       pop     ebp
0x25c617:       ret
```

Soit :

```
A-((((A*0x82b9)>16)&0xFFFF > 0xD)0x3eab)=3B14
```

Ce dernier type d'équation est visiblement un équivalent d'une équation modulaire.

Le tout subit ensuite une négation ainsi que l'inversion de l'octet de poids faible avec celui de poids fort :

```
0x25f055:       and     eax,0xff
0x25f05a:       shl     eax,0x8
0x25f05d:       mov     edx,eax
0x25f05f:       movzx   eax,WORD PTR [ebp-6]
0x25f063:       and     eax,0xff00
0x25f068:       sar     eax,0x8
0x25f06b:       or      eax,edx
0x25f06d:       not     eax
0x25f06f:       mov     WORD PTR [ebp-6],ax
```

Connaissant la zonne finale de 384 Bits, en la xorant puis en inversant le changement d'ordre et la précédente négation on obtient les résultats des équations modulaires, chacune au nombre de trois sur les 8 valeurs 16Bits que constitue

la zone en 0x60000. Le théorème des restes chinois devrait garantir une solution unique pour chacun de ces ensemble de trois équations. Dans le cas présent je me suis contenté de brute-forcer sans plus amples calcules.

On obtient alors la valeur en 0x60000 désirée. Celle-ci est dérivée du mots de passe ainsi que la clé précédemment xorée par 0x1337C0DE dans 19 itération d'un algorithme réversible ayant une constante d'initialisation à chaque cycle. On peut obtenir ces constantes a partir de la trace d'exécution ou les extraire en monitorant un dword de la zone 0x60000 et en affichant l'EDI.

Cela donne :

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

Il ne reste qu'as écrire l'algorithme inverse de celui-ci pour obtenir aux derniers cycles la clé de validation. Le code utilisé :

```
#include "stdio.h"
void main(){
unsigned int key[4]={0x4b9e2731,0xaab1bbe6,0xe4e4cfe3,0x5baa36bd};
unsigned int input[4]={0x9e8db86a, 0xce6d6c69, 0xbcc7253c, 0x196cd5f6};

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

unsigned int i,u,val;
unsigned int tmp1,tmp2,edi,ecx,key_;
for(i=1;i<0xffff; i++)
{
        u=(i*0x82b9);
        u=u>>0x1D;

        if ((i%0x3fc3==0x38E4)&&(i%0x34f6==0x1988)&&((i-(u*0x3eab))==0x3B14))
printf("A_%x\n",i);

        if ((i%0x23bf == 0x0F91)&&(i-(((i*0x349b)>>16>>12)*0x4ddd)==0x02D3)
&& (i-(((i*0xf7f3)>>16>> 0xe)*0x4214)==0x1A65)) printf("B_%x\n",i);

        if ((i%0x2693==0x1F43) && (i%0x15d0 == 0x1529)
&& (i-(((i*0xa0f)>>16>> 9)*0x32e7)==0x069B)) printf("C_%x\n",i);

        if ((i%0x6423 == 0x0627 ) && (i%0x4903==0x3C67)&&
(i-(((i*0x9b41)>>16>> 0xD)*0x34c4)==0x3021)) printf("D_%x\n",i);

        if ((i%0xec6==0x7B0)&&(i%0x125==0x09C)&&(i%0x1189==0x22A))
printf("E_%x\n",i);

        if ((i%0x459d==0x318D)&&(i%0x971==0x0964)&&
(i-(((i*0x5c81)>>16>> 0xD)*0x588f)==0x0BA9)) printf("F_%x\n",i);

        if ((i%0xed5==0x650)&&(i-(((i*0x2e69)>>16>> 0xB)*0x2c21)==0x2572)&&
(i-(((i*0x6615)>>16>> 10)*0xa08)==0x034E)) printf("G_%x\n",i);

        if ((i-(((i*0x462b)>>16>>7)*0x1d3)==0x01B5)&&(i-(((i*0xa36f)>>16>>9)*0x322)==0x005C)&&
```

```
(i-(((i*0x5cc3)>>16>> 9)*0x585)==0x0358)) printf("H %x\n",i);

}
printf("\n\n");
for(i=1;i<=19; i++)
{
edi=EDI[19-i];

        for(u=4;u>0; u--)
        {

        key_=key[(u-1)^((edi>>2)&3)];
        if(u==1) {ecx=input[3];} else {ecx=input[u-2];}
        if(u==4) {val=input[0];} else {val=input[u];}

        tmp1=((key_^ecx)+(edi^val))&0xFFFFFFFF;
        tmp2=((((ecx<<4)&0xFFFFFFFF)^(val>>3))+((ecx>>5)^((4*val)&0xFFFFFFFF)))&0xFFFFFFFF;
        input[u-1]=(input[u-1]-(tmp2^tmp1))&0xFFFFFFFF;

        }

        printf("%x %x %x %x\n",input[0],input[1],input[2],input[3]);
}
}
```

Et son résultat :

```
# ./a.out
H 196c
E 253c
C 6c69
B 9e8d
A b86a
F bcc7
D ce6d
G d5f6
```

```
f9ac7c75 3b1e3b4b 7a204a42 99013732
4ce643cd 6c8d5e0e 22d44fa9 74bf846c
9fbb9b57 f2889dcd 7e008046 4309464b
17645068 938d2883 5b65af44 1ad3cda6
ec0824dd c6aa1a82 f0547e5f 9fc3f7e2
6db1c11d fc2dcf24 e56dbded 1de704b2
a2da84e5 64a4adc3 c399b5c4 3cee77b1
6b4c8f22 104d3af8 c3a8691d df6f5d38
618b804d d9016c89 7e04171c 91e4e577
42d4688b 994f6cc2 bdd9f528 cff9274d
f7aa35ae df0e44b6 10fdf9ff b0402c14
756f2d82 9144b9aa 73bb53cc 9f2d3032
eddf82bd f035ae44 68db759e 9c9620b3
9a30e9e3 be53641b 50365d30 1617f8dc
9f6d65d4 3b1169e5 f0a4d0bd e11c487d
```

```
a46a47e6 a8567dde bad63bf 99953467
251715f4 3b4c7b2d 2bc4d505 ab4ad883
2f78daa1 ec1069cf d1e868ef 8cfd6d45
7761375a 5c57673f 7c4c483d 395f704e
```

```
'Z7aw?gW\=HL|Np_9'
```

Merci à Stéphane DUVERGER pour ce joli chalenge.
Flo florent.marceau@gmail.com