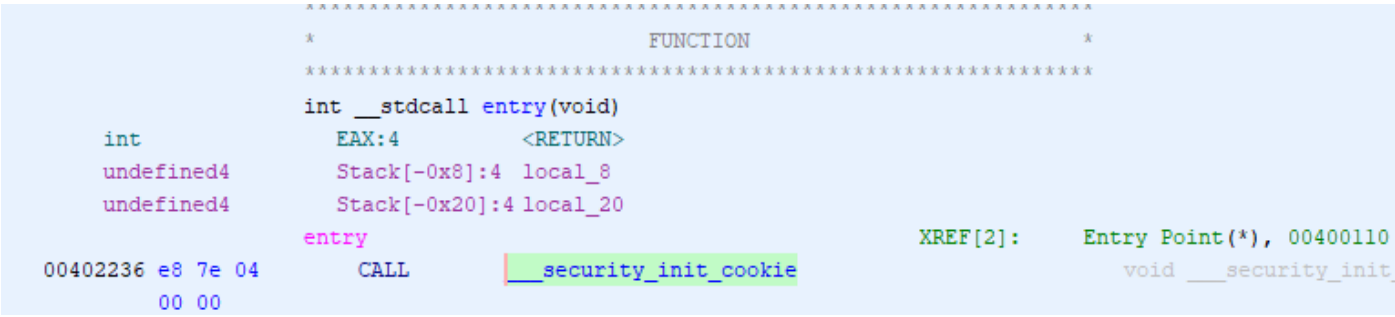
L'appel à __security_init_cookie doit être effectué avant l'entrée d'une fonction protégée contre le dépassement de délai. Dans le cas contraire, un dépassement de mémoire tampon parasite est détecté.
Cette fonction est appelée dans le main.

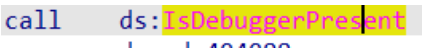


Lors de l'exécution du programme, s'il n'y a pas d'arguments, alors le programme ne fait pas echo (il plante mdr XD).

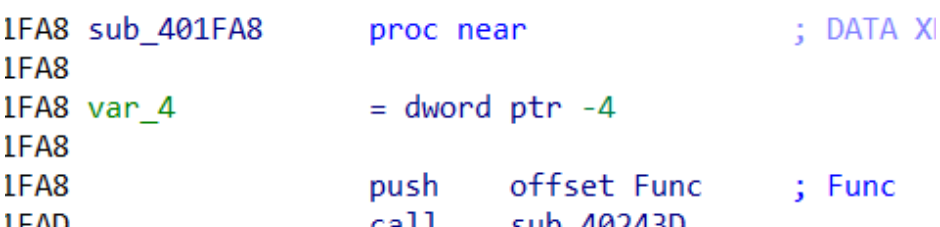
De plus, lorsque nous exécutons le programme avec plusieurs arguments, seul le premier est affiché, donc cela ne fait pas echo.

appel de **HeapEnableTerminateOnCorruption**
Qui vérifie la corruption du tas et renvoie un bool True s'il y a corruption
Setting the HeapEnableTerminateOnCorruption option is strongly recommended because it reduces an application's exposure to security exploits that take advantage of a corrupted heap.

Alors, il y a ce commentaire dans le main "/* WARNING: Globals starting with '_' overlap smaller symbols at the same address */", cela n'est-il pas la preuve qu'ils ont utilisé de l'overlapping ?

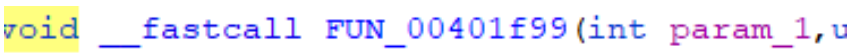


=> vu dans IDA
il appel la fonction IsDebuggerPresent, mais je ne sais pas s'il s'agit du main, je pensais que c'était le main jusqu'à ce que je tombe sur ce qu'il y a en dessous XD

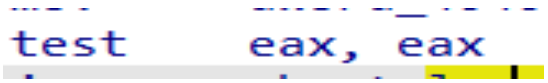


Étant donné que la fonction permettant de récupérer les arguments entrés en paramètres dans le main est utilisée dans cette fonction, je pense qu'il s'agit du main. => a vérifier, mais c'est la seul fonction qui contient _getmainargs

Dans ghidra, je pense que le main est cette fonction, cela correspond un peu XD



Et cette fonction contient bien l'appel à la fonction IsDebuggerPresent.
Il y a également une condition sur un argument qui je suppose est le premier argument entré en paramètres.



=> alors là mdr je ne comprend pas XD

La variable entrée est comparée à DAT_00404000 => voilà l'endroit où est défini ce truc, mais j'arrive pas trop à comprendre quelle valeur il a XD

```
                        // .data
                        // ram:00404000-ram:004041ff
                        //

                        DAT_00404000                                    XREF[10]:   0040023c(*),
                                                                                    FUN_00401260:00401266(R),
                                                                                    FUN_00401f99:00401f99(R),
                                                                                    FUN_00401f99:004022e5(R),
                                                                                    __IsNonwritableInCurrentImage:00...
                                                                                    __SEH_prolog4:0040262d(R),
                                                                                    FUN_00402669:0040267f(*),
                                                                                    ___security_init_cookie:004026c1...
                                                                                    ___security_init_cookie:00402741...
                                                                                    0040319c(*)
    00404000 4e e6 40 bb     undefine...  BB40E64Eh
```

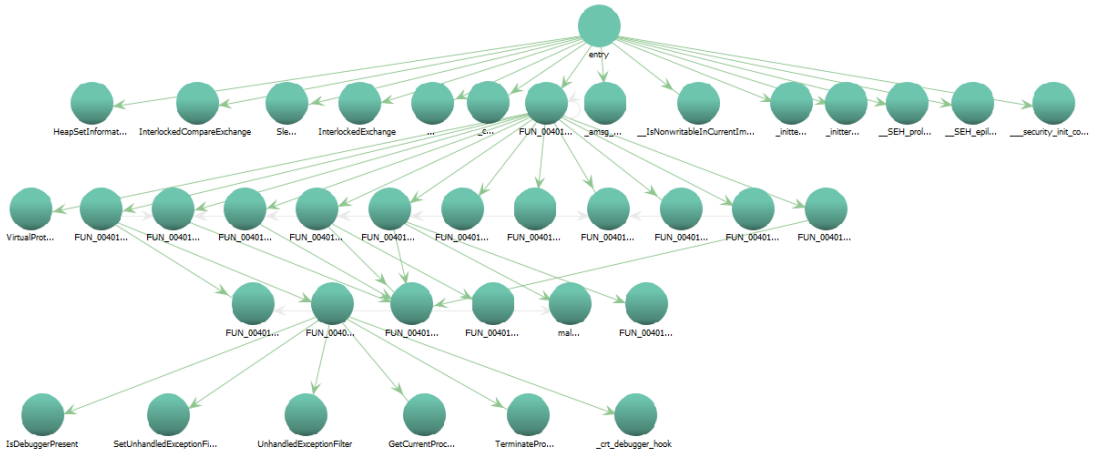
Analyse des strings étranges

IMAGE_SECTION_HEADER	".rsrc"
SectionFlags IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM...	"@.reloc"
?? 6Ch l	"la mere michelle a perdu son chat"
?? 63h c	"courage"
?? 6Fh o	"on t'aime"
?? 62h b	"bg va"

```
:00402236                public start
:00402236 start          proc near
:00402236
:00402236 ; FUNCTION CHUNK AT .text:00401FF3 SIZE 00000139 BYTES
:00402236 ; FUNCTION CHUNK AT .text:0040215C SIZE 00000020 BYTES
:00402236
:00402236                call    sub_4026B9
:0040223B                jmp     loc_401FF3
:0040223B start          endp
:0040223B
```

=> fonction main

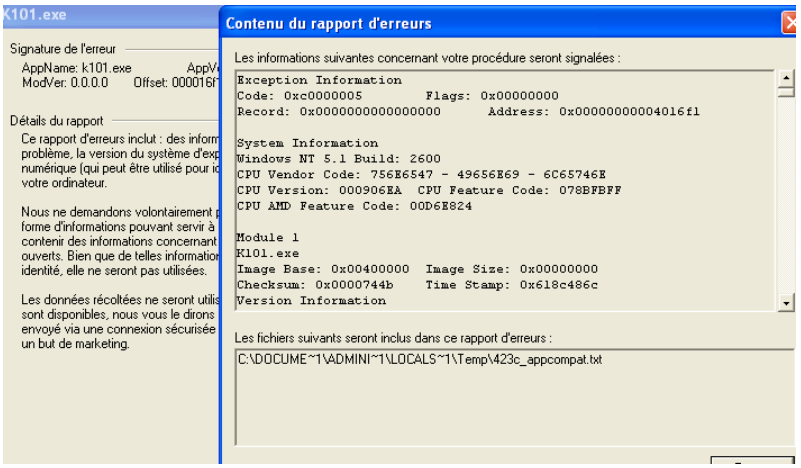
Entry function call graph:




```
                    FUN_00401030                                    XREF[1]:     FUN_0040
00401030 55             PUSH        EBP
00401031 8b ec          MOV         EBP,ESP
00401033 83 ec 14       SUB         ESP,0x14
00401036 a1 a4 30       MOV         EAX,[->MSVCR100.DLL::fscanf]                  = 000
         40 00
0040103b 89 45 fc       MOV         dword ptr [EBP + local_8],EAX
0040103e c7 45 f4       MOV         dword ptr [EBP + local_10],0x1
         01 00 00 00
00401045 c7 45 f8       MOV         dword ptr [EBP + local_c],0x0
         00 00 00 00

                    LAB_0040104c                                    XREF[1]:     00401226
0040104c 83 7d f4 00    CMP         dword ptr [EBP + local_10],0x0
00401050 0f 84 d5       JZ          LAB_0040122b
         01 00 00
00401056 8b 4d fc       MOV         ECX,dword ptr [EBP + local_8]
00401059 03 4d f8       ADD         ECX,dword ptr [EBP + local_c]
0040105c 0f be 11       MOVSX       EDX,byte ptr [ECX]
0040105f 69 d2 ff       IMUL        EDX,EDX,0xff
         00 00 00
```

```
bVar2 = true;
local_c = 0;
while (bVar2) {
  if (((((((char)fscanf_exref[local_c] * 0xff == -0x748b) &&
          ((char)fscanf_exref[local_c + 1] * 0xff == -0xff)) &&
         ((char)fscanf_exref[local_c + 2] * 0xff == 0x54ab)) &&
        (((char)fscanf_exref[local_c + 3] * 0xff == -0x748b &&
         ((char)fscanf_exref[local_c + 4] * 0xff == -0x13ec)))) &&
       (((char)fscanf_exref[local_c + 5] * 0xff == -0x728d &&
        ((char)fscanf_exref[local_c + 6] * 0xff == 0x44bb &&
         ((char)fscanf_exref[local_c + 7] * 0xff == 0xbf4)))))) &&
      ((char)fscanf_exref[local_c + 8] * 0xff == 0x4fb0)) &&
     (((((char)fscanf_exref[local_c + 9] * 0xff == 0x6996 &&
        ((char)fscanf_exref[local_c + 10] * 0xff == 0)) &&
       ((char)fscanf_exref[local_c + 0xb] * 0xff == -0xff)) &&
      ((((char)fscanf_exref[local_c + 0xc] * 0xff == 0x748b &&
        ((char)fscanf_exref[local_c + 0xd] * 0xff == 0x7f8)) &&
       (((char)fscanf_exref[local_c + 0xe] * 0xff == 0x6798 &&
        (((char)fscanf_exref[local_c + 0xf] * 0xff == -0x649b &&
         ((char)fscanf_exref[local_c + 0x10] * 0xff == -0x48b7)))))) {
    bVar2 = false;
  }
```



```
XREF[2]:     FUN_00401260:00401290(c),
             FUN_00401d70:00401daf(c)
```

dans la fonction FUN_00401f99
_DAT_00404154 = &param_3;
 _DAT_00404138 = param_2;
 _DAT_0040413c = param_1;
Ces 3 paramètres servent à rien car on les utilise juste pour les stocker (commande search all)

4016e0
Fonction qui retourne le nb de caractères.

```
1
2  int __cdecl FUN_004016e0(int param_1)
3
4  {
5    int local_8;
6
7    for (local_8 = 0; *(char *)(param_1 + local_8) != '\0'; local_8 = local_8 + 1) {
8    }
9    return local_8;
10 }
```

Fonction de la condition peut être ?: 0x401260 avec argv[1] en argument de la fonction

Toutes les fonctions (avec en arg: local_54 et &param_1):

- case 1: 0x401710
- case 2: 0x401780
- case 3: 0x401780
- case 4: 0x401900
- case 5: 0x401960
- case 6: 0x401A90
- case 7: 0x401B40
- case 8: 0x401900
- case 9: 0x401C10
- case 10: 0x401C70

- case 11: 0x401C60

```
for (local_14 = 0; local_14 < 4; local_14 = local_14 + 1) {
  if (param_1 != -1) {
    for (local_18 = 0; local_18 < param_1 + 1; local_18 = local_18 + 1) {
      FUN_00401000();
      *param_2 = *param_2 + 1;
    }
  }
```

```
sub_401000      proc near               ; CODE
                                        ; sub_

var_4           = dword ptr -4
arg_0           = byte ptr  8

                push    ebp
                mov     ebp, esp
                push    ecx
                push    ebx
                call    sub_401030
                mov     [ebp+var_4], eax
                movsx   eax, [ebp+arg_0]
                test    eax, eax
                jz      short loc_401024
                pop     eax
                pop     ebx
                pop     ecx
                mov     ebx, [ebx]
                push    ebx
                push    ecx
                push    ebx
                push    eax
                sub     ebp, 4
                mov     eax, [ebp+var_4]
```

var_4 est une variable locale qui contient la valeur de retour de la fonction sub_401030. Compare la valeur entrée en argument de la fonction, si elle vaut 0, alors on jump au label loc_401024 qui permet de retourner 2, sinon la valeur de retour de la fonction vaut var_4.

(De plus, ils modifient le pointeur EBP pour obtenir un accès à l'argument param_1 sans l'ajouter une nouvelle fois dans la pile et donc sans la passer en argument de la fonction. => pas sûr)

```
void __cdecl FUN_00401710(int param_1,int *param_2)

{
  int local_c;
  int local_8;

  for (local_8 = 0; local_8 < 1; local_8 = local_8 + 1) {
    if (param_1 != -1) {
      for (local_c = 0; local_c < param_1 + 1; local_c = local_c + 1)
        _fun_retour_2(*(undefined *)*param_2);
        *param_2 = *param_2 + 1;
      }
    }
  }
  return;
}
```

Cette fonction retourne le param_1 +1 ième caractère de param_2.

```
undefined ** FUN_00401030(void)

{
  code *pcVar1;
  bool bVar2;
  code **ppcVar3;
  int local_c;

  bVar2 = true;
  local_c = 0;
  while (bVar2) {
    if (((((((char)fscanf_exref[local_c] * 0xff == -0x748b) &&
            ((char)fscanf_exref[local_c + 1] * 0xff == -0xff)) &&
           ((char)fscanf_exref[local_c + 2] * 0xff == 0x54ab)) &&
          (((char)fscanf_exref[local_c + 3] * 0xff == -0x748b &&
           ((char)fscanf_exref[local_c + 4] * 0xff == -0x13ec)))) &&
         (((char)fscanf_exref[local_c + 5] * 0xff == -0x728d &&
          ((char)fscanf_exref[local_c + 6] * 0xff == 0x44bb &&
           ((char)fscanf_exref[local_c + 7] * 0xff == 0xbf4)))))) &&
       ((char)fscanf_exref[local_c + 8] * 0xff == 0x4fb0)) &&
      (((((char)fscanf_exref[local_c + 9] * 0xff == 0x6996 &&
         ((char)fscanf_exref[local_c + 10] * 0xff == 0)) &&
        ((char)fscanf_exref[local_c + 0xb] * 0xff == -0xff)) &&
       ((((char)fscanf_exref[local_c + 0xc] * 0xff == 0x748b &&
         ((char)fscanf_exref[local_c + 0xd] * 0xff == 0x7f8)) &&
        (((char)fscanf_exref[local_c + 0xe] * 0xff == 0x6798 &&
         (((char)fscanf_exref[local_c + 0xf] * 0xff == -0x649b &&
          ((char)fscanf_exref[local_c + 0x10] * 0xff == -0x48b7))))
    bVar2 = false;
  }
  else {
    local_c = local_c + 1;
  }
```

```
00401030 55             PUSH    EBP
00401031 8b ec          MOV     EBP,ESP
00401033 83 ec 14       SUB     ESP,0x14
00401036 a1 a4 30       MOV     EAX,[->MSVCR100.DLL::fscanf]
         40 00
0040103b 89 45 fc       MOV     dword ptr [EBP + local_8],EAX
0040103e c7 45 f4       MOV     dword ptr [EBP + local_10],0x1
         01 00 00 00
00401045 c7 45 f8       MOV     dword ptr [EBP + local_c],0x0
         00 00 00 00

                    LAB_0040104c                                    XREF[1]
0040104c 83 7d f4 00    CMP     dword ptr [EBP + local_10],0x0
00401050 0f 84 d5       JZ      LAB_0040122b
         01 00 00
00401056 8b 4d fc       MOV     ECX,dword ptr [EBP + local_8]
00401059 03 4d f8       ADD     ECX,dword ptr [EBP + local_c]
0040105c 0f be 11       MOVSX   EDX,byte ptr [ECX]
0040105f 69 d2 ff       IMUL    EDX,EDX,0xff
         00 00 00
00401065 81 fa 75       CMP     EDX,0xffff8b75
         8b ff ff
0040106b 0f 85 ac       JNZ     LAB_0040121d
         01 00 00
00401071 8b 45 fc       MOV     EAX,dword ptr [EBP + local_8]
00401074 03 45 f8       ADD     EAX,dword ptr [EBP + local_c]
00401077 0f be 48 01    MOVSX   ECX,byte ptr [EAX + 0x1]
0040107b 69 c9 ff       IMUL    ECX,ECX,0xff
         00 00 00
00401081 81 f9 01       CMP     ECX,0xffffff01
         ff ff ff
```

Dans cette fonction, on peut voir qu'elle retourne l'adresse d'une fonction en partant de l'adresse de fscanf, puis en effectuant des opérations afin de trouver l'adresse à renvoyer.

Michael: slide 1 à 6
Julie:7 à 9 et 16 à 18
Léo: 10 à 15