

Contourner l'utilisation de la CRT avec MinGW (hack GCC)

Gabriel Linder | Gabriel@wargan.com

Préambule

Bien que cet article soit très fortement axé sur GCC et sa suite d'outils, il est parfaitement applicable à Visual C++ et d'autres compilateurs. En effet on retrouve des concepts communs, et les exemples fournis sont compilables et fonctionnent sur Visual C++ 6. Pour ce qui est de leur compréhension, quelques notions d'assembleur et une certaine expérience de la programmation Windows sont requises. Bonne lecture :)

Introduction

La plupart des développeurs C/C++ sérieux sous Windows n'utilisent pas la CRT (pour *C Runtime Library*, une couche d'émulation pour les fonctions standard du C) mais écrivent du code Win32 natif. Cependant, GCC lie *toujours* les exécutables avec la CRT, car le code de démarrage (celui qui a le contrôle avant que la fonction WinMain ne soit appelée) utilise des fonctions présentes dans la CRT... Nous allons donc voir dans cet article comment contourner les limitations du compilateur, et créer notre propre routine de démarrage en lieu et place de celle fournie par défaut. Le but est d'obtenir des exécutables plus compacts, aussi bien en terme d'espace disque que d'occupation mémoire. Notez toutefois que ce genre d'optimisations est vraiment pour les passionnés qui cherchent un résultat optimal, et/ou un contrôle total sur le code généré ;)

Il est également à noter l'emploi du mot *hack*. Contrairement aux bêtises répandues par les médias, un *hacker* (dans le sens original et noble du terme) n'est pas un pirate, ni même forcément une personne douée dans le domaine des réseaux. Un *hacker* est avant tout une personne passionnée par l'informatique, et dont le but principal est de chercher (et trouver) comment optimiser au maximum le code qu'il écrit (pour un développeur), son système et/ou son réseau. Bien sûr, il doit souvent recourir à des procédés d'intrusion pour parvenir à ses fins, mais il ne le fait que sur sa propre machine – à moins d'y être invité. Il a également le sens du partage, mettant à disposition du reste de la communauté les connaissances parfois durement acquises, ce que je fais d'ailleurs en ce moment même :p

Assez de blabla idéologique, je vais à présent vous montrer comment j'ai constaté l'existence de ce code « caché » et la manière que j'ai trouvée pour y remédier.

Exemple pratique

Considérons le célèbre programme « Hello, world ! » :

```
#include <windows.h>
INT WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine,
                  INT nCmdShow)
{
    MessageBox(NULL,
               "Hello, world !",
               "Information",
               MB_ICONINFORMATION | MB_OK);
    return 0;
}
```

Simple, hein ? Bon, compilons-le (avec les symboles) :

```
$ gcc -g WinMain.c -o WinMain.exe -mwindows -luser32
```

On peut gager que le seul import de notre exécutable sera MessageBox@user32. Que nenni ! Un dump nous confirmera rapidement notre grossière erreur :

```
$ objdump -x WinMain.exe
start address 0x00401240
DLL Name: KERNEL32.dll
  vma:  Hint/Ord Member-Name Bound-To
  5130      1   AddAtomA
  513c     155  ExitProcess
  514c     175  FindAtomA
  5158     220  GetAtomNameA
  5168     236  GetCommandLineA
  517c     335  GetModuleHandleA
  5190     384  GetStartupInfoA
  51a4     735  SetUnhandledExceptionFilter
DLL Name: msvcrt.dll
  vma:  Hint/Ord Member-Name Bound-To
  51c4      39  __getmainargs
  51d4      60  __p__environ
  51e4      62  __p__fmode
  51f4      80  __set_app_type
  5208     121  _cexit
  5214     233  _iob
  521c     350  _onexit
  5228     388  _setmode
  5234     533  abort
  523c     540  atexit
  5248     575  free
  5250     626  malloc
  525c     656  signal
DLL Name: USER32.dll
  vma:  Hint/Ord Member-Name Bound-To
  5268     430  MessageBoxA
```

J'ai zappé ce qui n'avait rien à voir avec les imports... MessageBox est bien présente, mais qu'est-ce que c'est que toutes ces fonctions ?

Etude du code généré

Pour comprendre ce qui se passe, désassemblons notre fonction WinMain : en toute logique, on devrait y retrouver notre code (en assembleur, évidemment) :

```
(gdb) disassemble WinMain
Dump of assembler code for function WinMain:
0x401290 <WinMain>:      push  ebp
0x401291 <WinMain+1>:      mov   ebp,esp
0x401293 <WinMain+3>:      sub   esp,0x18
0x401296 <WinMain+6>:      mov   DWORD PTR [esp+12],0x40
0x40129e <WinMain+14>:     mov   DWORD PTR [esp+8],0x403000
0x4012a6 <WinMain+22>:     mov   DWORD PTR [esp+4],0x40300c
0x4012ae <WinMain+30>:     mov   DWORD PTR [esp],0x0
0x4012b5 <WinMain+37>:     mov   eax,ds:0x405128
0x4012ba <WinMain+42>:     call  eax
0x4012bc <WinMain+44>:     sub   esp,0x10
0x4012bf <WinMain+47>:     mov   eax,0x0
0x4012c4 <WinMain+52>:     leave
0x4012c5 <WinMain+53>:     ret   0x10
End of assembler dump.
```

Rien d'anormal, ça correspond à notre code en C. Mais on constate que WinMain est à l'adresse 0x401290, ce qui ne correspond pas avec la valeur du point d'entrée obtenue lors du dump précédent... Qu'y a t'il donc à l'adresse 0x00401240 ?

```
(gdb) disassemble 0x00401240
Dump of assembler code for function WinMainCRTStartup:
0x401240 <WinMainCRTStartup>:      push  ebp
0x401241 <WinMainCRTStartup+1>:      mov   ebp,esp
0x401243 <WinMainCRTStartup+3>:      sub   esp,0x8
0x401246 <WinMainCRTStartup+6>:      mov   DWORD PTR [esp],0x2
```

```

0x40124d <WinMainCRTStartup+13>:    call    ds:0x4050f8
0x401253 <WinMainCRTStartup+19>:    call    0x401100 <__mingw_CRTStartup>
0x401258 <WinMainCRTStartup+24>:    nop
0x401259 <WinMainCRTStartup+25>:    lea    esi, [esi*1]
End of assembler dump.

```

Bingo ! Voilà la fameuse fonction d'initialisation... En désassemblant la fonction `__mingw_CRTStartup`, on se rend compte qu'elle accomplit diverses initialisations avant de passer le contrôle du programme à la fonction `main`, qui récupère les arguments passés au programme avant d'appeler elle-même `WinMain`... Ouf ! Pas très optimal tout ça... De plus, une fois `WinMain` terminée, `main` fait un peu de ménage avant de se terminer, et `__mingw_CRTStartup` termine le nettoyage avant de terminer le programme par `ExitProcess`.

Recherche d'une solution

Puisque le véritable point d'entrée de l'exécutable est `WinMainCRTStartup`, pourquoi ne pas déclarer notre propre fonction `WinMainCRTStartup` :

```

#include <windows.h>
VOID WinMainCRTStartup(VOID)
{
    MessageBox(NULL,
               "Hello, world !",
               "Information",
               MB_ICONINFORMATION | MB_OK);
    ExitProcess(0);
}

```

Le prototype de la fonction, qui peut sembler surprenant de prime abord, correspond à celui deviné lors de l'étude du code assembleur (pas d'arguments apparents ni de retour, ce qui oblige l'emploi de `ExitProcess@kernel32` pour terminer le processus). De toute manière, ce n'est pas important, nous verrons plus tard comment obtenir `hInstance`, `lpCmdLine` et `nCmdShow`, ainsi que les `sdtinput/output/error` pour les applications console.

Compilons plutôt notre code :

```

$ gcc -g WinMainCRTStartup.c -o WinMainCRTStartup.exe -mwindows -lkernel32 -luser32
cckXcaaa.o(.text+0x0): In function `WinMainCRTStartup':
WinMainCRTStartup.c:2: multiple definition of `WinMainCRTStartup'
crt2.o(.text+0x240):crt1.c: first defined here
libmingw32.a(main.o)(.text+0x106):main.c: undefined reference to `WinMain@16'
collect2: ld returned 1 exit status

```

Aïe, on dirait bien que quelque chose s'est mal passé... Et oui, la fonction est déclarée en double : dans le code d'initialisation de la CRT et dans notre propre code, ce qui perturbe évidemment l'éditeur de lien ! Sans compter l'appel non résolu à `WinMain`... Allez, ne perdons pas courage, GCC est un compilateur tellement flexible, il doit bien y avoir une option permettant de zapper tout code autre que le nôtre propre.... `ld --help` ?

Solution

Une des options de `ld`, `--allow-multiple-definition`, permet de définir plusieurs fois une fonction. Une telle solution est inapplicable dans notre cas, puisque nous souhaitons vraiment retirer tout ce code inutile de notre exécutable... Oh, mais ! A peine plus loin, on trouve `-nostdlib` : « Only use library directories specified on the command line ».

Exactement ce qu'il nous faut, testons vite :

```

$ gcc -g WinMainCRTStartup.c -o WinMainCRTStartup.exe -nostdlib -mwindows -lkernel32 -luser32

```

Pas d'erreur à la compilation, l'exécution fonctionne...

Mais est-ce que les imports ont bien été épurés ? Vérifions le :

```

$ objdump -x WinMainCRTStartup.exe
start address 0x00401000

```

```

DLL Name: KERNEL32.dll
    vma:  Hint/Ord Member-Name Bound-To
    306c  155  ExitProcess
DLL Name: USER32.dll
    vma:  Hint/Ord Member-Name Bound-To
    307c  430  MessageBoxA

```

C'est bien mieux, non ? On va quand même vérifier que le point d'entrée du programme soit bien notre fonction personnalisée :

```

(gdb) disassemble 0x00401000
Dump of assembler code for function WinMainCRTStartup:
0x401000 <WinMainCRTStartup>:      push  ebp
0x401001 <WinMainCRTStartup+1>:     mov   ebp,esp
0x401003 <WinMainCRTStartup+3>:     sub   esp,0x18
0x401006 <WinMainCRTStartup+6>:     mov   DWORD PTR [esp+12],0x40
0x40100e <WinMainCRTStartup+14>:    mov   DWORD PTR [esp+8],0x402000
0x401016 <WinMainCRTStartup+22>:    mov   DWORD PTR [esp+4],0x40200c
0x40101e <WinMainCRTStartup+30>:    mov   DWORD PTR [esp],0x0
0x401025 <WinMainCRTStartup+37>:    mov   eax,ds:0x403064
0x40102a <WinMainCRTStartup+42>:    call  eax
0x40102c <WinMainCRTStartup+44>:    sub   esp,0x10
0x40102f <WinMainCRTStartup+47>:    mov   DWORD PTR [esp],0x0
0x401036 <WinMainCRTStartup+54>:    mov   eax,ds:0x403058
0x40103b <WinMainCRTStartup+59>:    call  eax
End of assembler dump.

```

Impeccable, mais tout ça pour quoi, me direz-vous ?

Comparons la taille de nos deux exécutables :

```

$ dir *.exe /o:n
17/09/2004  05:06                779.950 WinMain.exe
17/09/2004  05:18                376.140 WinMainCRTStartup.exe

```

Une diminution de 207% sans compression, sans compter l'optimisation du code, vous ne trouvez pas ça bien, vous ? Et ne perdons pas de vue que ces deux exécutables ont été compilés avec symboles, donc sont obèses :)

Voyons à présent la taille des fichiers sans les symboles :

```

$ strip -s *.exe
$ dir *.exe /o:n
17/09/2004  05:20                5.632 WinMain.exe
17/09/2004  05:20                 2.048 WinMainCRTStartup.exe

```

2.048 octets pour un Hello World, vous l'avez fait en assembleur ou quoi ? :)

Limitations

Ben oui, y'en a... Ce serait trop beau sinon :)

Considérons le code suivant :

```

#include <windows.h>
VOID WINAPI WinMainCRTStartup(HINSTANCE hInstance,
                              HINSTANCE hPrevInstance,
                              LPSTR lpCmdLine,
                              INT nCmdShow)
{
    MessageBox(NULL,
               lpCmdLine,
               "Hello, world !",
               MB_ICONINFORMATION | MB_OK);
    ExitProcess(0);
}

```

Quand on le compile, on obtient le message « Warning: resolving _WinMainCRTStartup by linking to _WinMainCRTStartup@16 ». Souvenez-vous : WinMainCRTStartup n'accepte pas d'arguments... Mais alors comment récupérer la ligne de commande et les autres informations passées à WinMain ?

Ben oui, c'est çà de zapper le code d'initialisation, faut le recoder ensuite :p

Recoder le code d'initialisation

L'API Win32 fournit quelques fonctions que nous allons utiliser, à savoir :
GetCurrentProcess@kernel32, qui nous permettra de récupérer hInstance
GetCommandLine@kernel32, qui nous donnera la ligne de commande « brute »
GetStartupInfo@kernel32 pour récupérer nCmdShow (pour les applications graphiques) et
hStdInput/Output/Error (pour les applications console)

Voici donc un code basique, simulant en partie l'ancienne CRT :

```
#include <windows.h>
/* fonction WinMain classique */
INT WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPTSTR lpCmdLine,
                  INT nCmdShow)
{
    MessageBox(NULL,
              lpCmdLine,
              "Information",
              MB_ICONINFORMATION | MB_OK);
    return 0;
}
/* CRT personnalisée */
VOID WinMainCRTStartup(VOID)
{
    /* variables WinMain-like */
    HINSTANCE hInstance;
    HINSTANCE hPrevInstance;
    LPTSTR lpCmdLine;
    INT nCmdShow;
    /* récupération des informations du process */
    STARTUPINFO si;
    GetStartupInfo(&si);
    /* assignation des variables */
    hInstance = GetCurrentProcess();
    hPrevInstance = NULL;
    lpCmdLine = GetCommandLine();
    nCmdShow = si.wShowWindow;
    /* appel de WinMain */
    ExitProcess(WinMain(hInstance,
                      hPrevInstance,
                      lpCmdLine,
                      nCmdShow));
}
```

Pourquoi hPrevInstance == NULL ? Cette variable est un vestige des systèmes 16 bits, et n'est plus utilisée depuis Windows 95... Evidemment, la version que je vous propose est uniquement didactique, une bonne optimisation serait d'inclure directement le code du programme dans WinMainCRTStartup et de zapper totalement WinMain ;)

Le code une fois compilé et débarassé des symboles fait 2.560 octets, et importe uniquement les fonctions utilisées :

```
$ objdump -x WinMainCustomCRT.exe
start address 0x00401037
DLL Name: KERNEL32.dll
    vma:  Hint/Ord Member-Name Bound-To
    3084   155  ExitProcess
    3094   236  GetCommandLineA
    30a8   281  GetCurrentProcess
    30bc   384  GetStartupInfoA
DLL Name: USER32.dll
    vma:  Hint/Ord Member-Name Bound-To
    30d0   430  MessageBoxA
```

Pour simuler totalement l'ancienne CRT, il faudrait retirer le nom de l'exécutable des paramètres, car il est retourné par GetCommandLine@kernel32... Ce n'est pas vraiment compliqué (une simple incrémentation de pointeur), je ne vais donc pas en parler ici :)

La récupération des hStd* se fait également par le biais de si, la structure initialisée par GetStartupInfo@kernel32. Pour remplacer printf, il faudra utiliser FormatMessage@kernel32 ou wsprintf@user32 pour la mise en forme, et WriteConsole@kernel32 ou WriteFile@kernel32 pour afficher le texte. Là aussi rien d'insurmontable, le plus dur étant déjà fait :)

Conclusion

Hé bien, si vous êtes arrivés jusqu'ici, j'espère que vous avez apprécié cette astuce peu connue/documentée. Quand j'ai cherché sur Google, je ne suis arrivé à rien et j'ai donc du plonger mes délicates petites mimines dans le cambouis ;)

Enfin, ce hack s'applique également aux applications console, avec `void mainCRTStartup(void)` remplaçant `main()` ou à des DLLs, avec `BOOL WINAPI DllMainCRTStartup(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)` à la place de la classique `DllMain`. Notez que les DLLs sont les seules à conserver un prototype assez proche de la version originale, ceci est dû au fait que la CRT est normalement déjà initialisée par le processus appelant... Ceci peut être une cause de bug avec certaines DLLs mal écrites et s'appuyant encore sur des fonctions de la CRT, mais je n'ai encore eu aucun problème avec celles de l'API Win32. Si d'aventure vous aviez un bug, je vous saurais gré de me le faire savoir :)

Un petit tour dans les sources de GCC (sur le site de MinGW, pour avoir les extensions Windows du compilateur) confirmera les prototypes que je vous ai donnés, et l'examen de la CRT originale sera très instructif. Cette CRT est implémentée dans le paquet MinGW-Runtime.

A noter que si on applique ce hack à Visual C++ 6.0, la taille des exécutables générés passe de 24Ko (avec initialisations) à 16Ko (sans)... Je n'ai pas testé avec d'autres compilateurs, mais je serais ravi de recevoir les résultats de vos tests, ainsi que toute remarque sur ce document ;)

Amusez-vous bien, et si vous avez des astuces dans ce genre... Partagez ! :)

Gabriel Linder
Gabriel@wargan.com

Quelques liens utiles

<http://www.mingw.org/> - la version de GCC pour Windows par excellence
<http://www.msdn.com/> - la documentation Windows de référence
<http://upx.sourceforge.net/> - un compresseur d'exécutables versatile et efficace
<http://flatassembler.net/> - un assembleur excellent pour DOS/Windows/*nix
<http://www.wargan.org/> - le site préliminaire de Wargan

Ce document est fourni sous licence GNU/FDL. Pour plus d'informations, rendez-vous sur le site de la Free Software Foundation : <http://www.fsf.org/licenses/fdl.html>

Les sources de ce document (au format OpenOffice.org v1.1.2) sont disponibles sur Wargan.org dans mon dossier personnel, accompagnées des divers codes sources didactiques qui ont parsemés ce document.