Memory Editing

« C++ »

By Yttrium

Bonjour à tous.
Dans ce tutoriel, je vais essayer de vous expliquer les rudiments de l'édition de mémoire en C++.
Ce cours sera dispatché en 3 grands chapitres.

Commençons par le commencement, les choses nécessaires à la compréhension et à l'utilisation du cours :

- Un Environnement de Développement Intégrée tel que Code::Blocks
- Des connaissances basiques en C++ et dans le fonctionnement de la mémoire et des dll

Sommaire :

Commençons dès maintenant avec le premier chapitre !

I. L'édition de mémoire en théorie

Tout d'abord, la première question que vous pouvez vous posez est la suivante :

Qu'est-ce que le « Memory Editing » ?
J'ai choisie d'utiliser la version anglaise du terme car elle est bien plus rependue.

Cela consiste à modifier le contenu des variables utilisées par un programme.

Pour ce faire, il nous faut tout d'abord trouver l'adresse de ces variables, afin de les repéré permit les centaines qui ne nous intéresse pas.

Pour cela, j'ai créé un petit programme qui sera notre cible tout le long de ce cours, et qui nous permettra de nous exercer sans aucune difficulté.

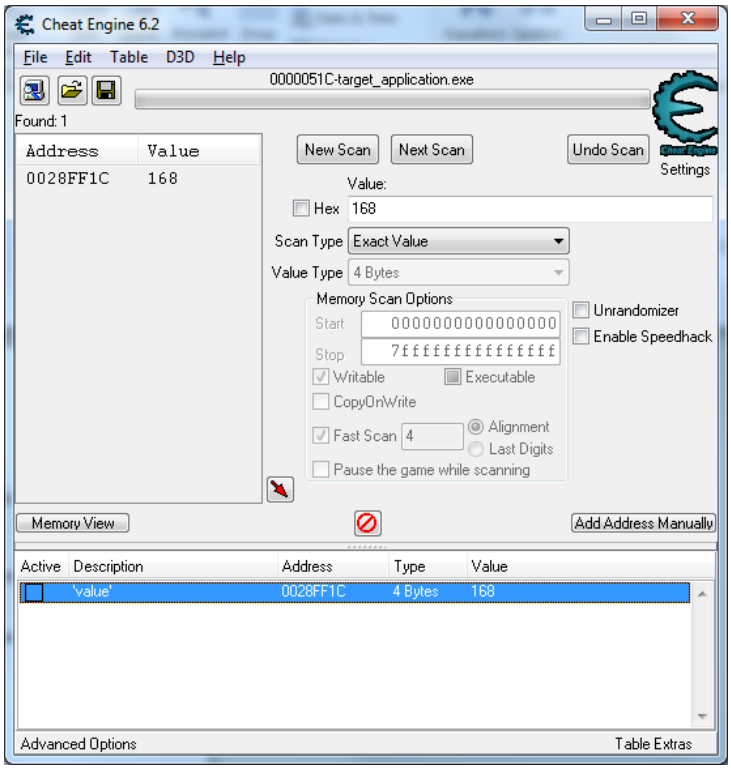
Il se trouve normalement dans les ressources du tutoriel sous le nom « target_application.exe », mais si vous souhaitez vous-même le compilé, en voici le code source :

```
#include <iostream>

using namespace std;

int main()
{
   cout << "\tTarget Application 1" << endl;
   cout << "\t  Tuto Memory C++   " << endl;
   cout << "\t   Yttrium - 2012   " << endl;
   cout << "\t  BufferOverflow.fr " << endl;

   int value = 42;

   while(value != 0)
   {
      value = value + 42;
      cout << value << endl;
      cin.get();
   }

   return 0;
}
```

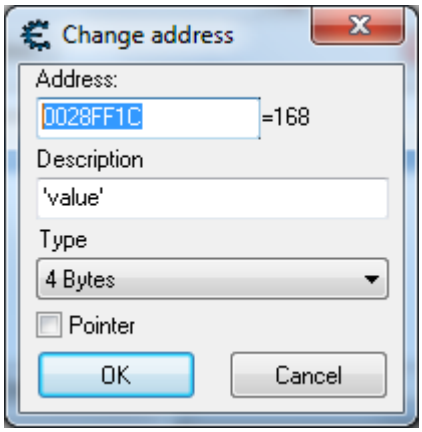
Maintenant que nous connaissons notre cible, notre but va être de trouver puis de modifier de toutes les façons possibles la valeur de la variable « value ».

Pour trouver son adresse, la façon la plus rapide, et la plus simple va être d'utilisé un programme tel que « Cheat Engine »
Je vous laisse vous débrouiller avec ce dernier pour trouver la valeur en question, de nombreux tutoriels sont disponible en ligne afin d'apprendre à s'en servir !

Une fois la valeur identifié comme ci-dessous



Nous allons récupérer son adresse.



Dans mon cas, l'adresse en question sera : 0028FF1C

II. L'édition Externe

Maintenant que nous savons où se trouve notre variable nous allons apprendre à la modifier grâce à un programme tierce.

Voici les include nécessaire à nos programmes :

```
#include <iostream>
#include <windows.h>
#include <tlhelp32.h>
```

En premier lieu, il va falloir récupérer le handle du processus. Pour cela nous avons deux possibilités. Tout d'abord, nous pouvons essayer le récupéré grâce à la fenêtre du processus en utilisant FindWindow().

```
HWND hWnd = NULL;
DWORD dwID;
HANDLE hProcess;

hWnd = FindWindow(NULL, "Ma Fenetre");
   if( hWnd == NULL )
   {
      cout << "ERROR: FindWindow" << endl;
      cin.get();
      return 0;
   }

hProcess = OpenProcess(PROCESS_ALL_ACCESS, NULL, dwID);
   if( hProcess == NULL )
   {
      cout << "ERROR: OpenProcess" << endl;
      cin.get();
      return 0;
   }
```

Mais il existe une autre solution (que je trouve bien meilleurs) qui est de parcourir la liste des processus à la recherche du notre.

```
PROCESSENTRY32 entry;
entry.dwSize = sizeof(PROCESSENTRY32);
HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);

if (Process32First(snapshot, &entry) == TRUE)
{
   while (Process32Next(snapshot, &entry) == TRUE)
   {
      if (stricmp(entry.szExeFile, "target_application.exe") == 0)
      {
         HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, entry.th32ProcessID);
         CloseHandle(hProcess);
      }
   }
}

CloseHandle(snapshot);
cin.get();
return 0;
```

Maintenant que nous avons notre programme en main, ainsi que notre adresse, il ne nous reste plus qu'à accéder à ces dernières grâce a notre programme et les fonctions :

ReadProcessMemory() et WriteProcessMemory()

Voici un exemple d'utilisation avec notre « target_application » :

```
#include <iostream>
#include <windows.h>
#include <tlhelp32.h>

using namespace std;

int main()
{
   PROCESSENTRY32 entry;
   entry.dwSize = sizeof(PROCESSENTRY32);
   HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);

   if (Process32First(snapshot, &entry) == TRUE)
   {
      while (Process32Next(snapshot, &entry) == TRUE)
      {
         if (stricmp(entry.szExeFile, "target_application.exe") == 0)
         {
            HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, entry.th32ProcessID);

            int addr = 0x0028FF1C;
            int value = 50;
            int buf = 0;

            ReadProcessMemory   (hProcess,  (LPVOID)addr,   &buf,4,NULL);
            cout << buf << endl;
            cin.get();

            WriteProcessMemory  (hProcess,  (LPVOID)addr,   (LPVOID) &value, sizeof(&value), NULL);

            ReadProcessMemory   (hProcess,  (LPVOID)addr,   &buf,4,NULL);
            cout << buf << endl;
            cin.get();

            CloseHandle(hProcess);
         }
      }
   }

   CloseHandle(snapshot);
   return 0;
}
```

Voilà, nous en avons fini pour ce chapitre, si vous cherchez a en apprendre plus, vous pouvez aussi lire l'annexe, qui concerne les adresse relatives (adresse sous la forme de module.dll+00FF par exemple)

III. L'édition Interne

Maintenant, que nous savons modifier nos variables à partir d'un programme externe, nous allons le faire, mais à partir d'une dll qui sera injecté dans notre processus.

Tout d'abord, il va falloir injecter cette dll. Vu que l'objet de ce cours n'est pas de travailler sur l'injection de dll en elle-même, mais plus sur l'édition de mémoire via l'injection, je ne détaillerais pas énormément le code source.

```
#include <iostream>
#include <windows.h>

using namespace std;

int main()
{

char* dllPath = "lib.dll";
void* pLoadLibrary = (void*)GetProcAddress(GetModuleHandle("kernel32"),"LoadLibraryA");
cout << "LoadLibrary:0x" << hex << pLoadLibrary << dec << "\nCreating process ' target_application.exe' ... \n";

STARTUPINFOA startupInfo;
PROCESS_INFORMATION processInformation;
ZeroMemory(&startupInfo,sizeof(startupInfo));

if(!CreateProcessA(0,"target_application.exe",0,0,1,CREATE_NEW_CONSOLE,0,0,
   &startupInfo,&processInformation))
   {
   cout << "Could not run 'target_application.exe'. GetLastError() = " << GetLastError();
   return 0;
   }

cout << "Allocating virtual memory ...\n";
void* pReservedSpace =
VirtualAllocEx(processInformation.hProcess,NULL,strlen(dllPath),MEM_COMMIT,PAGE_EXECUTE_READWRITE)
;
if(!pReservedSpace)
   {
   cout << "Could not allocate virtual memory. GetLastError() = " << GetLastError();
   return 0;
   }
```

```
cout << "Writing process memory ...\n";
if(!WriteProcessMemory(processInformation.hProcess,pReservedSpace,dllPath,strlen(dllPath),NULL))
   {
   cout << "Error while calling WriteProcessMemory(). GetLastError() = " << GetLastError();
   return 0;
   }

cout << "Creating remote thread ...\n";
HANDLE hThread = CreateRemoteThread(processInformation.hProcess,
NULL,
0,
(LPTHREAD_START_ROUTINE)pLoadLibrary,
pReservedSpace,
0,
NULL);
if(!hThread)
   {
   cout << "Unable to create the remote thread. GetLastError() = " << GetLastError();
   return 0;
   }

cout << "Thread created.\n";

WaitForSingleObject(hThread,INFINITE);
VirtualFreeEx(processInformation.hProcess,pReservedSpace,strlen(dllPath),MEM_COMMIT);

cout << "Done.";
return 0;
}
```

Pour faire simple, le programme charge une dll « lib.dll » et l'injecte dans le processus target_application.exe
Ce code n'est pas du tout de moi, il provient du site : http://xevia.webege.com

Maintenant que nous possédons un injecteur, tout ce qu'il nous reste à faire, c'est notre dll !

Pour faire simple, lorsque la dll sera accrochée au processus, nous appelleront la fonction « Func » qui créera un pointeur vers une variable de type int aillant pour adresse 0x0028FF1C. Il nous suffira ensuite de modifier la valeur pointé par le pointeur pour accéder la valeur de notre variable.

Le code sera composé du header basique créer par Code::Blocks et d'une fonction d'écriture.

```
#ifndef __MAIN_H__
#define __MAIN_H__

#include <windows.h>
#include <iostream>

#ifdef BUILD_DLL
   #define DLL_EXPORT __declspec(dllexport)
#else
   #define DLL_EXPORT __declspec(dllimport)
#endif

#ifdef __cplusplus
extern "C"
{
#endif

void DLL_EXPORT Func();
BOOL DLL_EXPORT WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpvReserved);

#ifdef __cplusplus
}
#endif

#endif // __MAIN_H__
```

```
#include "main.h"

void DLL_EXPORT Func()
{
   MessageBoxA(0, "Start", "DLL Message", MB_OK | MB_ICONINFORMATION);
   int *memory = (int*)0x0028FF1C;

   MessageBoxA(0, "*memory = 150", "DLL Message", MB_OK | MB_ICONINFORMATION);
   *memory = 150;
   std::cout << memory << "-" << *memory << std::endl;

   MessageBoxA(0, "*memory = 0", "DLL Message", MB_OK | MB_ICONINFORMATION);
   *memory = 0;
   std::cout << memory << "-" << *memory << std::endl;
}

BOOL DLL_EXPORT WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
   switch (fdwReason)
   {
      case DLL_PROCESS_ATTACH:
      MessageBoxA(0, "DLL_PROCESS_ATTACH", "DLL", MB_OK | MB_ICONINFORMATION);
      Func();
         break;

      case DLL_PROCESS_DETACH:
         break;
      case DLL_THREAD_ATTACH:
         break;
      case DLL_THREAD_DETACH:
         break;
   }
   return TRUE;
}
```

Attention, si vous travailler sous codeblock, n'oublier pas d'exporter la fonction « DllMain » en précisant le « DLL_EXPORT »

A. Les adresses relatives

Nous voilà dans l'annexe qui va nous permettre de clarifier un point, celui des adresses relatives. Dans certains cas, lorsque vous rechercherez une adresse, vous vous retrouverez avec une adresse de type module.dll+00FF

Pour faire simple, sela signifie que l'adresse ne prend pas pour adresse de base le début de la zone mémoire alloué au programme, mais celle du module en question.

Pour résoudre ce problème, il existe une solution très simple, parcourir tous les modules a la recherche de celui en question !

Voici le code source de notre éditeur externe modifié pour prendre en compte un module !

Dans notre cas, il n'y a pas de module, donc aucune différence, mais cela pourra toujours vous servir à l'avenir !

```
PROCESSENTRY32 entry;
   entry.dwSize = sizeof(PROCESSENTRY32);
   HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);

   if (Process32First(snapshot, &entry) == TRUE)
   {
      while (Process32Next(snapshot, &entry) == TRUE)
      {
         if (stricmp(entry.szExeFile, "target_application.exe") == 0)
         {
            HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, entry.th32ProcessID);

            HWND hWnd = NULL;

            cout << "Module32 Search ..." << endl;
            BYTE*      modul_addr = 0;
            HANDLE hModuleSnap = INVALID_HANDLE_VALUE;
            MODULEENTRY32 me32;
            hModuleSnap = CreateToolhelp32Snapshot( TH32CS_SNAPMODULE, entry.th32ProcessID );
            if( hModuleSnap == INVALID_HANDLE_VALUE )
            {
               cout << "ERROR: CreateToolhelp32Snapshot" << endl;
               cin.get();
               return 0;
            }

            me32.dwSize = sizeof( MODULEENTRY32 );
```

```
        me32.dwSize = sizeof( MODULEENTRY32 );

        if( !Module32First( hModuleSnap, &me32 ) )
        {
           cout << "ERROR: Module32First" << endl;
           cin.get();
           CloseHandle( hModuleSnap );
           return 0;
        }
        do
        {
           if (stricmp(me32.szModule, "module.dll") == 0)
           {
           cout << "Get the module : " << me32.szModule << endl;
           modul_addr = me32.modBaseAddr;
           }

        }
        while(Module32Next( hModuleSnap, &me32 ));
        CloseHandle( hModuleSnap );

        int addr = 0x0028FF1C + (int)modul_addr;
        int value = 50;
        int buf = 0;

        ReadProcessMemory  (hProcess,  (LPVOID)addr,   &buf,4,NULL);
        cout << buf << endl;
        cin.get();

        WriteProcessMemory  (hProcess,  (LPVOID)addr,   (LPVOID) &value, sizeof(&value), NULL);

        ReadProcessMemory  (hProcess,  (LPVOID)addr,   &buf,4,NULL);
        cout << buf << endl;
        cin.get();


        CloseHandle(hProcess);
     }
   }
}

CloseHandle(snapshot);
return 0;
```