« Balade avec la mémoire virtuelle »

par *Homeostasie*

15/12/2009

(trashomeo at gmail dot com)

1. Introduction

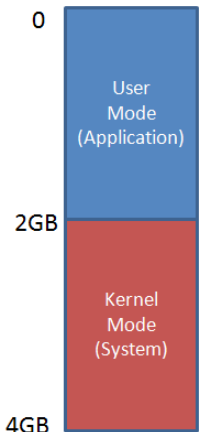
Tout au long de ce "paper", nous allons nous promener dans le monde de la mémoire virtuelle sous une plateforme Windows 32 bits.
Pourquoi ce "paper"? L'idée m'est venue en aidant rapidement un internaute X sur un code d'injection de thread. Puis en y réfléchissant, nombre de programmeurs utilise la fameuse API VirtualAlloc*() pour allouer de l'espace mémoire dans un processus distant ou non. Notamment pour faire de l'injection de DLL ou de code, sans pour autant comprendre le mécanisme qui se cache derrière. D'autant plus que j'ai l'impression que la « toile » n'est pas très riche concernant ce thème.

Dans un premier temps, je présenterais l'utilité de la MV (Mémoire Virtuelle). J'expliquerais ensuite le principe de la pagination et de l'accès à la mémoire physique. Enfin, pour digérer, nous verrons quelques exemples qui montrent l'évolution de la MV lors des phases de réservation et d'engagement. Pour cela, je mettrais à disposition un petit outil que j'ai développé.

2. Présentation et utilité de la mémoire virtuelle

Avec l'augmentation du nombre de processus actifs gourmand en termes de mémoire, la mémoire physique ne peut pas toujours satisfaire l'ensemble des processus. C'est notamment grâce à la MV que ce problème est résolu, ceci en mettant à disposition des programmes plus de mémoire qu'ils en existent.

Dorénavant chaque processus reçoit son propre espace d'adressage de 4Go. Valeur issue du nombre de fils d'adresses du microprocesseur pour accéder à la mémoire physique, soit 2^32. A préciser que sur ces 4Go de mémoire adressable, 2 Go sont destinés au mode utilisateur et les 2 Go restant au noyau.

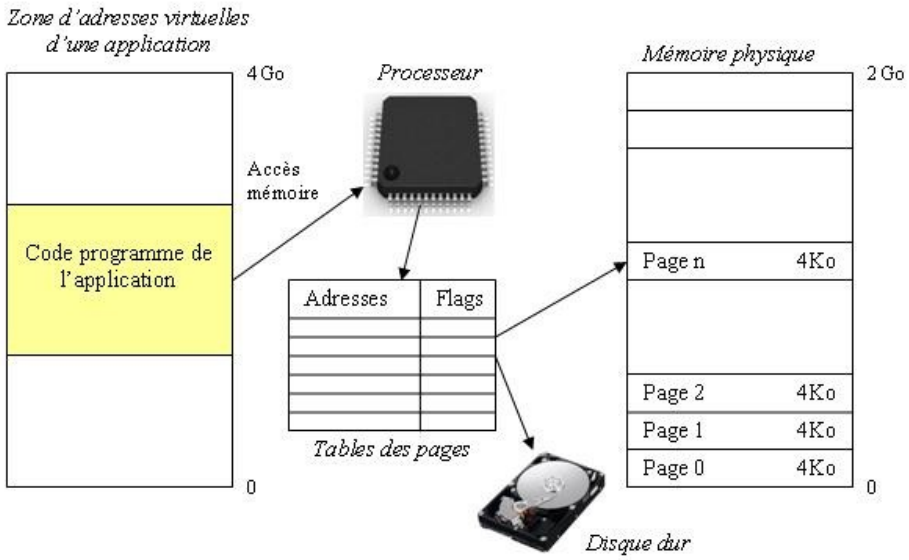


Disposition mémoire utilisateur/noyau

Comme son nom l'indique, cela reste de la mémoire virtuelle et il serait impossible d'accéder et d'avoir autant de mémoire physique disponible pour chaque programme.

Alors quelle est la solution mise en œuvre ? Bien, étant donné que l'intégralité de la mémoire au bon fonctionnement d'un processus n'est pas nécessaire à un instant x, il est possible de stocker sur disque les parties inactives ou temporaires. C'est le fichier d'échange qui fera office "d'extension" de RAM. Ce fichier est nommé « swap file » sur plateforme Windows 95/98/ME et « page file » sur Windows NT, incluant Windows 2000 et XP (pagefile.sys).

Quand un programme tentera d'accéder à une adresse qui n'est pas couramment en RAM, cela générera une interruption, appelée « page fault ». Celle-ci demandera au système de récupérer à partir du « page file », la page de 4 Ko contenant l'adresse attendue.
A noter qu'au regard du processus, l'ensemble de la mémoire est toujours disponible. Ce dernier ne remarquera nullement que c'est le processeur qui intercepte l'accès et recharge la partie utile en RAM.



Concept de la gestion de la mémoire virtuelle

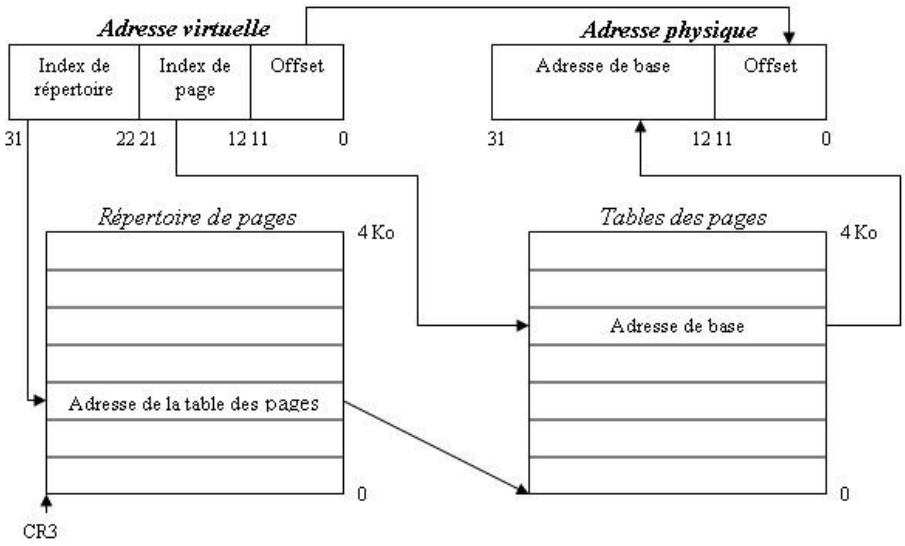
Le processeur divise la mémoire virtuelle en blocs de 4Ko appelés "page". Lorsque le code d'une application souhaite accéder à son espace d'adressage, le processeur effectue un calcul sur l'adresse virtuelle pour déterminer le numéro de la page concernée à partir de la table des pages (« page directories »).

Un avantage qui est apparu grâce au principe de pagination est la possibilité de projeter une même page dans plusieurs espaces d'adressage. Ainsi plusieurs processus accèderont sans le savoir à la même « page-frame » (emplacement physique) à partir d'endroits différents (emplacement dans la mémoire virtuelle). C'est notamment le cas lorsque plusieurs applications utilisent une DLL commune.

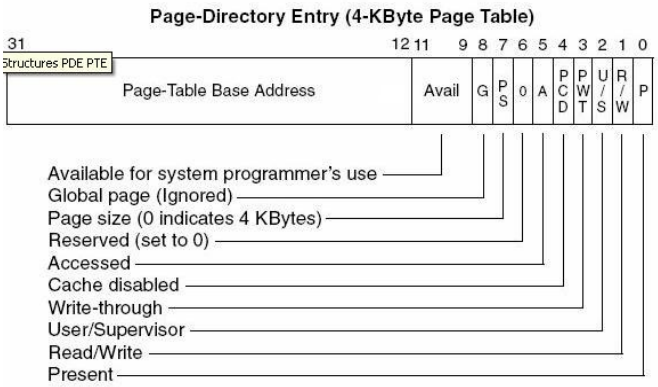
3. Fonctionnement de la pagination et d'accès de la mémoire

Comme dit précédemment, lorsqu'un processus souhaite accéder à une page mémoire, l'adresse virtuelle doit être traduite en adresse physique. Cette transformation est réalisée via la table dite "répertoire de tables de pages", souvent abrégée en "répertoire de pages".
Cette table représente en fait un tableau de 1024 entrées de 32 bits, chaque entrée pointant sur l'adresse de base d'une table de page. De plus, une entrée fournit diverses informations sur l'état de la table de page concernée. C'est ainsi que le processeur découvre via un flag, si une page se trouve en mémoire physique ou au contraire, si il doit effectuer son chargement en RAM au préalable avant de fournir l'accès au processus. De même, il lui saura possible de déterminer si une table de pages est en lecture seule ou non, si celle-ci est disponible pour le monde utilisateur ou noyau.
A savoir que le point d'entrée du répertoire de page est stocké dans le registre CR3.

Les figures ci-dessous illustrent la conversion d'une adresse linéaire en adresse physique et la structure d'une entrée du répertoire de pages :



Conversion d'une adresse physique en adresse physique



Structure d'une entrée du répertoire de page

Comme l'illustration le montre, les 10 bits supérieurs de l'adresse virtuelle donne l'indice d'accès à l'intérieur du répertoire de page pour obtenir l'adresse de la table de page concernée. On notera que l'on retrouve bien nos 2^10 (1024) entrées pour une taille finale du répertoire de page de 4 octets * 1024 entrées, soit 4Ko.
Les 10 bits suivants (12 à 21) de l'adresse virtuelle constituent un index dans la table de page. Comme précédemment une table de page à 1024 entrées et chaque éléments de la table a pour valeur la partie haute de l'emplacement de l'adresse physique (de 12 à 31). La valeur de cette entrée est aussi appelée "Page-Frame". La structure d'une entrée de la table de page est quasiment identique à celle d'une entrée du répertoire de page.
Enfin les 12 bits de poids faible restant de l'adresse virtuelle représentent un offset à coupler avec l'adresse de base pour obtenir l'adresse physique finale et ainsi récupérer les données voulues. L'offset peut atteindre une valeur de 2^12 octets soir 4096.

Je ne sais pas vous, mais moi cela me retourne le cerveau! Mais pourquoi donc tant de manipulation avec des index de table et des offsets? Eh bien, tout simplement pour économiser de la mémoire.

Imaginons deux cas possibles:

- Dans le premier cas, nous avons seulement 2 étages avec une table de pages sur 10 bits et un offset pour l'adresse de base sur 22 bits. Nous aurions donc 1024 entrées pointant chacune sur une page de 4Mb. Comme dit précédemment, la taille d'une page sur la plupart des systèmes est fixée à 4Kb. Alors pourquoi 4Mb n'est-il pas envisageable? A cause du gaspillage de mémoire dû à la fragmentation. Il est nettement préférable de perdre une page de 4Kb qu'une page de 4Mb. D'autant plus si le programme utilise seulement quelques dizaines octets d'une page.
  Mais alors, pourquoi pas une page de 1Kb, voire 512 octets? Parce qu'en fait la MMU (Memory Management Unit) utilise une partie de la mémoire cache du processeur appelée TLB (Translation Lookaside Buffer) pour accélérer la traduction des adresses virtuelles en adresses physiques. Pour résumer brièvement, le TLB mémorise les derniers couples (page, offset) correspondant aux dernières pages physiques auxquelles le processeur a dû accéder, ainsi on évite la perte de temps pour effectuer la conversion d'une adresse virtuelle en adresse physique. En fait, avec l'augmentation de la taille mémoire RAM qu'un système possède de nos jours, il s'avère qu'une page de 4Kb n'est pas le bon compromis en terme de gaspillage mémoire versus temps de processeur pour effectuer la translation. En effet des pages de 16Kb seraient plutôt recommandées.

- Dans le second cas, oui oui, j'avais dit qu'il y en avait deux, nous aurons aussi deux 2 étages mais cette fois-ci une table de page sur 20 bits et un offset sur 12 bits afin d'obtenir des pages de 4Kb. Nous obtenons finalement une table qui prendra en mémoire pour chaque processus 4 octets * 2^20, soit 4Mb. Ceci n'est pas non plus raisonnable. Ainsi donc un étage supplémentaire est rajouté pour éviter de consommer inutilement de la mémoire pour chaque processus.

Mais maintenant, comment cela ce passe t'il dans un environnement multi-processus ?

Pourquoi des processus peuvent employer la même adresse virtuelle dans leur espace adressable de 4Go sans pour autant qu'il y ait de collision en mémoire physique ?
Tout simplement, parce qu'à chaque processus actif est associé un répertoire de page distinct. Et c'est à chaque commutation de processus qu'un changement de contexte a lieu et qu'une nouvelle valeur du registre CR3 est chargée afin de pointer sur le bon répertoire de page.

Bon, c'est fait, j'ai grillé un neurone! Vite un ti-punch pour rétablir la connexion... Bizarrement, mon crâne résonne, martelé par les effluves de citron vert et de sucre de canne, la prochaine fois je prendrais du rhum sec.

Trêve de plaisanteries, passons plutôt à des exemples d’illustration.

4. Illustration par des exemples

Pour l'ensemble de ces tests, les logiciels utilisés sont :

- VirtualMem.exe, un petit outil que j'ai codé pour s'amuser avec la MV.
- Le logiciel "Process Explorer" de Sysinternals pour visualiser l'état de la MV, de la mémoire engagée et physique.

A noter que je suis sur un système avec 3Go de RAM et un « page file » de 5Go.

Astuce : Lorsqu'il sera nécessaire d'allouer des milliers de régions, je vous conseille de réduire la console afin d'obtenir le résultat plus rapidement.

4.1. Visualiser les informations concernant la MV sur son système et sur le processus VirtualMem.exe.

Pour cela lancez la commande suivante:
VirtualMem.exe --blocknum 8 où 8 représente le nombre de page allouée.

Ci-dessous le résultat obtenu:

```
Information about Virtual Memory
--------------------------------
Granularity for the starting address at which virtual memory can be allocated: 65536 (0x10000).
Computer page size: 4096 (0x1000).
Lowest memory address accessible: 65536 (0x10000).
Highest memory address accessible: 2147418111 (0x7ffeffff).

/!\ Please, press key to reserve and commit virtual memory...

Reserve and commit Virtual Memory
---------------------------------
(00000) VirtualAlloc succeed to reserve a block. (Virtual Base Address of Block = 0x00500000)
        -> (00) VirtualAlloc succeed to commit a size of 4096 bytes.(Virtual Base Address of Page = 0x00500000)
        -> (01) VirtualAlloc succeed to commit a size of 4096 bytes.(Virtual Base Address of Page = 0x00501000)
        -> (02) VirtualAlloc succeed to commit a size of 4096 bytes.(Virtual Base Address of Page = 0x00502000)
        -> (03) VirtualAlloc succeed to commit a size of 4096 bytes.(Virtual Base Address of Page = 0x00503000)
        -> (04) VirtualAlloc succeed to commit a size of 4096 bytes.(Virtual Base Address of Page = 0x00504000)
        -> (05) VirtualAlloc succeed to commit a size of 4096 bytes.(Virtual Base Address of Page = 0x00505000)
        -> (06) VirtualAlloc succeed to commit a size of 4096 bytes.(Virtual Base Address of Page = 0x00506000)
        -> (07) VirtualAlloc succeed to commit a size of 4096 bytes.(Virtual Base Address of Page = 0x00507000)
```

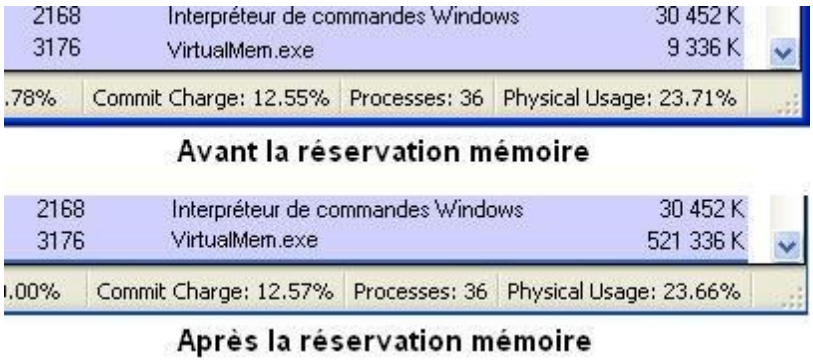
On constate ici que la granularité d'allocation pour une région est de 64Ko (65536).
La taille d'une page sur mon système est bien de 4Ko. En effet, à chaque appel de VirtualAlloc*(), l'adresse de base est multiple de 0x1000 soit 4Ko.
Je ne m'attarde pas sur l'adresse la plus basse. Enfin l'adresse la plus haute signifie que la dernière région qui peut être utilisée débute à l'adresse 0x7ffe0000. Je n'en dirais pas plus !

4.1. Comprendre la signification des différents états de la mémoire (libre, réservé ou engagé).

Un processus démarre avec l'ensemble de son espace adressable libre. Avant de pouvoir utiliser de la mémoire virtuelle, il est nécessaire de la réserver afin d'éviter que tout autre appel à VirtualAlloc*() alloue la même région. Cette opération n'engendre aucun cout en termes de mémoire. Ni la mémoire physique, ni le "page file" n'est sollicité. Vérifions donc cela à l'aide de nos outils.
Nous allons réserver 512 Mo de MV, pour cela saisissez le commande suivante : *VirtualMem.exe --regionnum 8000*".

Le 8000 corresponds au nombre de régions à allouer. Soit la réservation mémoire attendue divisée par la granularité de l'adresse de départ d'une région (512Mo/64Ko).

Ci-dessous un « screen shot » de l'état initial avant et après réservation:



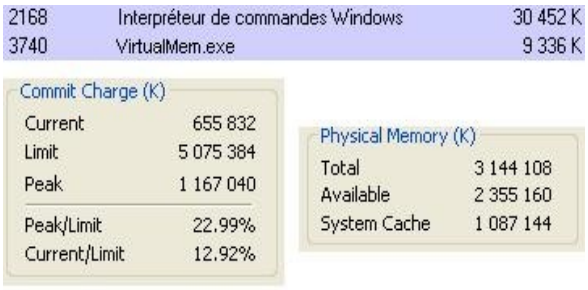
Nous constatons bien que la mémoire engagée et la mémoire physique restent identiques. Par contre, la mémoire virtuelle allouée au processus VirtualMem.exe a bien augmenté de 512 Mo.

Afin d'utiliser un espace réservé, la mémoire doit être au préalable engagée. Il est possible d'engager une page à la fois ou plusieurs pages. Ceci en fonction du nombre d'octets demandé.
Nous restons dans le même cas de figure que précédemment mais cette fois nous allons engager toute la mémoire réservée pour nos 8000 régions afin de constater que l'opération est cette fois-ci couteuse en ressources.

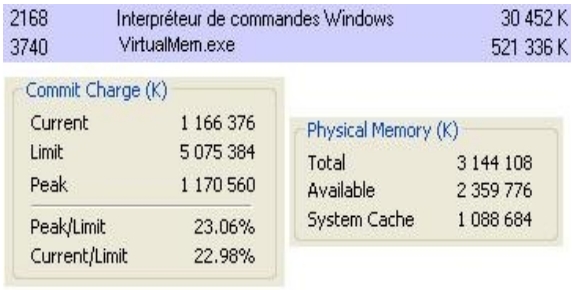
La ligne de commande à saisir est la suivante:
VirtualMem.exe --regionnum 8000 --blocknum 16 où le 16 représente le nombre de blocs (ici équivalent à une page) dans une région.

En effet, 16*4Ko équivaut bien à 64Ko. Étant donné que je possède sur mon PC 3Gb de RAM et un « page file » de 5Gb, je devrais augmenter réciproquement les pourcentages de 17% et 10.2%.

Ci-dessous un « screen shot » de l'état initial avant et après réservation/engagement:



Avant l'engagement de la mémoire



Après l'engagement de la mémoire

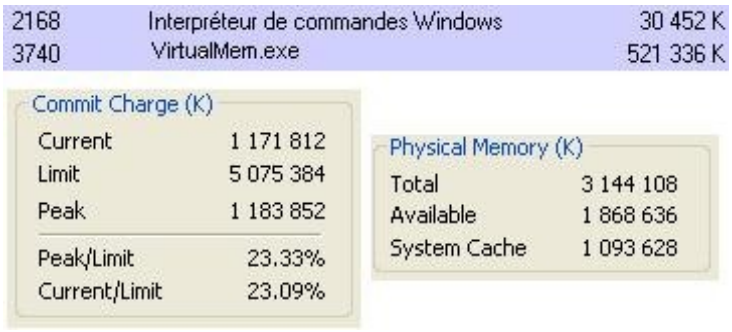
Sacrebleu... Vous ne voyez pas un truc de bizarre. La mémoire engagée a bien augmenté comme attendue mais la mémoire physique n'a pas bougé. Pourquoi?
Tout simplement parce que la mémoire engagée n'est pas utilisée par l'application, donc il n'y a pas lieu de charger les pages en mémoire physique.

Une façon simple de forcer ce « swapping » est de tout simplement écrire quelques données dans chaque page engagée.

La ligne de commande à saisir est la suivante:
VirtualMem.exe --regionnum 8000 --blocknum 16 --write 128 où 128 représente le nombre d'octets à écrire dans chaque page.

Je précise que l'on aurait pu très bien écrire uniquement 1 octet ou 4Ko, le résultat aurait été le même en terme d'occupation de RAM. La granularité du système pour charger une page en mémoire physique étant de 4Ko.

Ci-dessous un « screen shot » de l'état initial après réservation/engagement et écriture:



Après l'engagement de la mémoire et écriture dans les pages

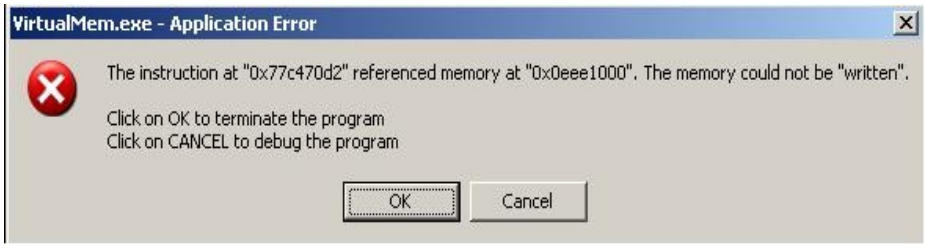
Cette fois-ci, la mémoire physique grimpe en flèche et nous avons bien consommé nos 512 Mo.
Vous pourrez remarquer qu'après libération de la mémoire à la dernière étape de mon « tool », tout redevient à l'état initial.

4.3. Pour terminer, utilisons l'ensemble des commandes pour mettre en évidence la granularité d'une page et l'allocation de zone mémoire contigüe.

Dans un premier te temps, la commande à saisir :
VirtualMem.exe --startaddr 0eee0000 --regionnum 1 --regionsize 65536 --blocknum 1 --blocksize 1 --write 4096

Ici, en écrivant 4096 octets dans 1 bloc de 1 octets de mémoire engagée à partir de la région d'adresse de départ 0x0eee0000, on constate que aucune erreur à lieu. Pourtant seulement un octet a été alloué. Encore une fois, ce comportement est dû à la granularité d'allocation d'une page qui est de 4Ko.

Saisissons maintenant la commande suivante :
VirtualMem.exe --startaddr 0eee0000 --regionnum 1 --regionsize 65536 --blocknum 1 --blocksize 1 --write 4097



Pop-up avertissant l'accès à une adresse invalide

En écrivant maintenant 4097 octets, cette fois-ci le dernier octet sort de l'espace engagée et provoque une erreur fatale. D'ailleurs grâce à la pop-up, nous constatons bien que l'on effectue une écriture à l'adresse 0x0eee1000. Adresse qui correspond au début de la page suivante non engagée.

Enfin pour terminer, utilisons la commande suivante :
VirtualMem.exe --startaddr 0eee0000 --regionnum 1 --regionsize 65536 --blocknum 1 --blocksize 4097 --write 8192

En engageant un bloc de 4097 octet, on réserve implicitement 2 pages de 4Ko de mémoire contigüe. Donc l'écriture de 1 à 8192 octets ne produit aucune erreur. Par contre, avec 8193, c'est le drame !

5. Conclusion

Je souhaite que cet article vous ait apporté les informations attendues. En tout cas, j'ai essayé d'expliquer avec précision, les points essentiels de la mémoire virtuelle.
Afin de bien comprendre le principe de région, de pages, de blocs de mémoire contigüe, je vous conseille de jouer avec les différents paramètres de mon outil. D'autre part, pour que vous puissiez manipuler à votre guise, je ne fais aucune vérification sur la pertinence des données utilisateurs saisies.

Pour d'éventuelles questions, remarques, ou précisions sur le contenu de ce « paper »: «trashomeo at gmail dot com».

6. Salutations

599eme Man, Electricdr3ke, int_0x80, PHPLizardo, Xylitol, p3lo & All EuropA|SecuriTy.

7. Source

http://www.mirorii.com/fichier/83/324133/VirtualMem-zip.html

```
/*****************************************************************/
/* VirtualMem.exe: Small tool for playing with Virtual Memory.*/
/*      With the user input parameters, you can specify       */
/*      address starting, number and size of region. It will  */
/*      be also possible to set number and size block.        */
/*      Finally, to force RAM using, you can write n bytes in */
/*      each commited blocks.                                 */
/*                                                            */
/* This tool is developped for the 50-1337 eMagazine 1,       */
/* "Balade avec la mémoire virtuelle" paper.                  */
/*                                                            */
/* 12/11/2009 by Homeostasie (trashomeo at gmail dot com)     */
/*****************************************************************/

#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#define BLOCKLIMIT 32768    // Max Number of block to ask
#define HELP    "Help menu, below list of available arguments:\n" \
                "-----------------------------------------------\n" \
                "--startaddr:  Starting HEXADECIMAL address of region to reserve (by default NULL)\n" \
                "--regionnum:  Number of region to reserve (by default 1)\n" \
                "--regionsize: Size of region to reserve (by default frame granularity, 64k on Win32)\n" \
                "--blocknum:   Number of block to commit (by default 0)\n" \
                "--blocksize:  Size of block to commit (by default page size, 4k on Win32\n" \
                "--write:      Write n bytes into each commited memory block to force physical memory usage\n"

#define USED_PARAMETERS "Start Address: %p\n" \
                        "Region Number: %lu\n" \
                        "RegionSize:    %lu bytes\n" \
                        "Page Number:   %lu\n" \
                        "Commited size: %lu bytes\n"
```

```
void myGetLastError(void)
{
   LPVOID lpMsgBuf;
   FormatMessage(
      FORMAT_MESSAGE_ALLOCATE_BUFFER |
      FORMAT_MESSAGE_FROM_SYSTEM |
      FORMAT_MESSAGE_IGNORE_INSERTS,
      NULL,
      GetLastError(),
      0, // Default language
      (LPTSTR) &lpMsgBuf,
      0,
      NULL
   );

   printf("%s\n", (char*)lpMsgBuf);

   LocalFree( lpMsgBuf );
}


int main(int argc, char *argv[])
{
        LPVOID lpvBase, lpvNext, lpvResult; // Base address of memory
        LPTSTR lpTabPage[BLOCKLIMIT];       // Address array of virtual memory page
        BOOL bSuccess;                      // Flag
        DWORD i,j;                      // Generic counters
        LPVOID lpvStartAddress,lpvStartBase;// Starting address of region
   DWORD dwRegionNumber;              // Region number to rerserve/allocate
   DWORD dwSizeToAllocate;            // Virtual memory size to allocate by Block
   DWORD dwBlockNum;                  // Number of commited blocks
   DWORD dwPageSize;                  // Page size value
   DWORD dwNumToWrite;                // Number of byte to write for each blocks
   char *strToWrite = NULL;           // String to write
        SYSTEM_INFO sSysInfo;              // Useful information about the system

        // Initialize local variables
        GetSystemInfo(&sSysInfo);
        dwPageSize = sSysInfo.dwPageSize;
        dwSizeToAllocate = sSysInfo.dwAllocationGranularity;
        dwRegionNumber = 1;
        dwBlockNum = 0;
        dwNumToWrite = 0;
        lpvStartBase = lpvStartAddress = NULL;

   system("CLS");
   printf ("------------------------------------------------------------\n");
   printf ("*-----------------------------------------------------------*\n");
        printf ("*                                                           *\n");
        printf ("*      Walking with the Virtual Memory by Homeostasie       *\n");
        printf ("*                                                           *\n");
        printf ("*            (501337 eZine1 & Europa|Security)              *\n");
        printf ("*                                                           *\n");
        printf ("*                                                           *\n");
        printf ("*-----------------------------------------------------------*\n");
        printf ("------------------------------------------------------------\n\n\n");

        if(argc > 13)
        {
      printf("Invalid number of arguments.\n--help to get information.\n");
      return EXIT_FAILURE;
        }
        else
        {
      for(i = 1 ; i < argc; i++)
      {
         if(strcmp(argv[i],"--startaddr")==0){
            i++;
            lpvStartBase = (LPVOID)strtoul(argv[i], NULL, 16);
```

```
        }
        else if(strcmp(argv[i],"--regionnum")==0){
            i++;
            dwRegionNumber = strtoul(argv[i], NULL, 10);
        }
        else if(strcmp(argv[i],"--regionsize")==0){
            i++;
            dwSizeToAllocate = strtoul(argv[i], NULL, 10);
        }
        else if(strcmp(argv[i],"--blocknum")==0){
            i++;
            dwBlockNum = strtoul(argv[i], NULL, 10);
        }
        else if(strcmp(argv[i],"--blocksize")==0){
            i++;
            dwPageSize = strtoul(argv[i], NULL, 10);
        }
        else if(strcmp(argv[i],"--write")==0){
            i++;
            dwNumToWrite = strtoul(argv[i], NULL, 10);
            if(dwNumToWrite) {
                strToWrite = malloc(dwNumToWrite);
                if(strToWrite == NULL){
                    printf("\t-> malloc FAILED: ");
                    myGetLastError();
                    return EXIT_FAILURE;
                }
                else{
                    DWORD k = 0;
                    for(k = 0; k < dwNumToWrite; k++) {strToWrite[k] = 'a';}
                }
            }
        }
        else{
            printf("%s\n", HELP);
            return EXIT_FAILURE;
        }
    }
        }

printf ("Used Parameters for this test\n");
        printf ("-----------------------------------\n");
        printf (USED_PARAMETERS, lpvStartBase, dwRegionNumber, dwSizeToAllocate, dwBlockNum, dwPageSize);

        printf ("\nInformation about Virtual Memory\n");
        printf ("-----------------------------------\n");
        printf ("Granularity for the starting address at which virtual memory can be allocated: %lu (0x%x).\n",
                                            sSysInfo.dwAllocationGranularity,
                                            sSysInfo.dwAllocationGranularity);
        printf ("Computer page size: %lu (0x%x).\n", sSysInfo.dwPageSize, sSysInfo.dwPageSize);
printf ("Lowest memory address accessible: %lu (0x%x).\n", sSysInfo.lpMinimumApplicationAddress,
                                            sSysInfo.lpMinimumApplicationAddress);
        printf ("Highest memory address accessible: %lu (0x%x).\n", sSysInfo.lpMaximumApplicationAddress,
                                            sSysInfo.lpMaximumApplicationAddress);

printf("\n/!\\ Please, press key to reserve and commit virtual memory...\n"); _getch();

        printf ("\nReserve and commit Virtual Memory\n");
        printf ("-----------------------------------\n");
        for(i = 0; i < dwRegionNumber; i++)
        {
    if(lpvStartBase != NULL)
        lpvStartAddress = lpvStartBase + i*(dwSizeToAllocate);

    // Reserve pages in the virtual address space of the process.
    lpvBase = VirtualAlloc(
                    lpvStartAddress,      // System by default or user selects address
                    dwSizeToAllocate,     // Size of allocation
                    MEM_RESERVE,          // Allocate reserved pages
                    PAGE_NOACCESS);       // Protection = no access
```

```
      if (lpvBase == NULL )
      {
         printf("(%05lu) VirtualAlloc reserve failed: ", i);
         myGetLastError();
      }
      else
      {
         lpTabPage[i] = lpvBase;
         printf("(%05lu) VirtualAlloc succeed to reserve a block. (Virtual Base Address of Block = 0x%p)\n", i, lpvBase);

         for(j = 0; j < dwBlockNum; j++)
         {
            lpvNext = lpvBase + j*dwPageSize;
            lpvResult = VirtualAlloc(
                        (LPVOID) lpvNext,  // Next memory location to commit
                        dwPageSize,        // Page size, in bytes
                        MEM_COMMIT,        // Allocate a committed page
                        PAGE_READWRITE);   // Read/write access
            if (lpvResult == NULL )
            {
               printf("\t-> (%02lu) VirtualAlloc commit FAILED: ", j);
               myGetLastError();
            }
            else
            {
               printf("\t-> (%02lu) VirtualAlloc succeed to commit a size of %lu bytes.(Virtual Base Address of Page = 0x%p)\n", j,
dwPageSize,lpvResult);
               if(dwNumToWrite){
                  if(memcpy(lpvNext, strToWrite, dwNumToWrite) != lpvNext){
                     printf("\t-> memcpy FAILED: ");
                     myGetLastError();
                  }
               }
            }
         }
      }
         }

   printf("\n/!\\ Please, press key to release blocks...\n"); _getch();

         printf ("\nRelease blocks of Virtual Memory page\n");
         printf ("----------------------------------------\n");
         // Release the block of pages when you are finished using them.
         for(i = 0; i < dwRegionNumber; i++)
         {
      lpvBase = lpTabPage[i];
      if(lpvBase != NULL)
      {
         bSuccess = VirtualFree(
                        lpvBase,       // Base address of block
                        0,             // Bytes of committed pages
                        MEM_RELEASE);  // Decommit the pages
         printf ("Release (0x%p) %s", lpTabPage[i], bSuccess ? "succeeded.\n" : "failed: " );
         if(bSuccess == FALSE) myGetLastError();
      }
         }

   printf("\n/!\\ Please, press key to exit...\n"); _getch();
   if(strToWrite != NULL) free(strToWrite);

   return EXIT_SUCCESS;
}
```