# *Rootkit : invisibilité sur Windows NT (partie 1/2)*

## I. Introduction aux Rootkit

**Comment ne pas se faire voir sur Windows NT**
**Partie 1 sur 2**
---
Version Originale par: Holy_Father
Traduction par: Damien
Version: 1.1 french
Date: 28.01.2004, updated 26.02.2006
Web: http://www.hxdef.org, http://hxdef.net.ru, http://hxdef.czweb.org, http://rootkit.host.sk
Link: http://www.rootkit.com

## II. Explications

### =====[ 1. Sommaire ]====



### =====[ 2. Introduction ]=====

Ce document traite des techniques de camouflage des objets tels que les fichiers, les services, les processus etc. sur Windows NT. Ces méthodes sont basées sur le hook des fonctions API Windows qui sont plus détaillées dans mon article "Hooking Windows API".

Tout ce qui est évoqué ici vient de ma propre recherche durant l'écriture de code pour rootkits, donc cela certainement être dit de meilleure façon ou plus compréhensible.
Cacher un objet arbitraire dans ce document signifie changer des fonction systèmes qui appellent cet objet de telle manière qu'elles sauteraient son appel. Dans ce cas-là la valeur de retour de la fonction deviendrait notre valeur de retour, l'objet n'existant plus. La méthode de base est ce que nous appellerons ; la fonction d'origine avec les arguments d'origine, dont nous modifierons ensuite la sortie. Dans la version de ce texte, nous verrons plus en détail les moyens de cacher les fichiers, processus, clés et valeurs du registre, les services et pilotes systèmes, ainsi que les mémoires allouées et handles.

## =====[ 3. Les fichiers ]=====

Il y a plusieurs possibilités de cacher les fichiers de telle façon que l'OS ne les voit pas. Notre but sera seulement de modifier les API et de laisser de côté les techniques telles que celles qui jouent sur le système de fichiers. Ce sera aussi plus simple car nous n'aurons pas besoin de savoir comment le système de fichiers fonctionne.

## =====[ 3.1 NtQueryDirectoryFile ]=====

Chercher un fichier dans un répertoire sur WinNT est basé sur la recherche de tous les fichiers de ce répertoire et de ses sous-répertoires.
NtQueryDirectoryFile est utilisé pour lister ces fichiers.

```
NTSTATUS NtQueryDirectoryFile(
IN HANDLE FileHandle,
IN HANDLE Event OPTIONAL,
IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
IN PVOID ApcContext OPTIONAL,
OUT PIO_STATUS_BLOCK IoStatusBlock,
OUT PVOID FileInformation,
IN ULONG FileInformationLength,
IN FILE_INFORMATION_CLASS FileInformationClass,
IN BOOLEAN ReturnSingleEntry,
IN PUNICODE_STRING FileName OPTIONAL,
IN BOOLEAN RestartScan
);
```

Les paramètres qui nous intéressent sont FileHandle, FileInformation et FileInformationClass. FileHandle représente le handle du répertoire de l'objet, il peut être obtenu à l'aide de NtOpenFile. FileInformation est un pointeur sur le bloc mémoire alloué, où la fonction d'écriture veut les données. FileInformationClass renseigne du type d'enregistrement écrit dans FileInformation.
FileInformationClass est un type d'énumération variant, mais nous n'avons besoin que de quatre valeurs, qui sont utilisées pour lister le contenu du répertoire:

```
#define FileDirectoryInformation 1
#define FileFullDirectoryInformation 2
#define FileBothDirectoryInformation 3
#define FileNamesInformation 12
```

La structure de l'enregistrement écrit dans FileInformation pour FileDirectoryInformation:

```
typedef struct _FILE_DIRECTORY_INFORMATION {
ULONG NextEntryOffset;
ULONG Unknown;
LARGE_INTEGER CreationTime;
LARGE_INTEGER LastAccessTime;
LARGE_INTEGER LastWriteTime;
LARGE_INTEGER ChangeTime;
```

```
LARGE_INTEGER EndOfFile;
LARGE_INTEGER AllocationSize;
ULONG FileAttributes;
ULONG FileNameLength;
WCHAR FileName[1];
} FILE_DIRECTORY_INFORMATION, *PFILE_DIRECTORY_INFORMATION;
```

pour FileFullDirectoryInformation:

```
typedef struct _FILE_FULL_DIRECTORY_INFORMATION {
ULONG NextEntryOffset;
ULONG Unknown;
LARGE_INTEGER CreationTime;
LARGE_INTEGER LastAccessTime;
LARGE_INTEGER LastWriteTime;
LARGE_INTEGER ChangeTime;
LARGE_INTEGER EndOfFile;
LARGE_INTEGER AllocationSize;
ULONG FileAttributes;
ULONG FileNameLength;
ULONG EaInformationLength;
WCHAR FileName[1];
} FILE_FULL_DIRECTORY_INFORMATION, *PFILE_FULL_DIRECTORY_INFORMATION;
```

pour FileBothDirectoryInformation:

```
typedef struct _FILE_BOTH_DIRECTORY_INFORMATION {
ULONG NextEntryOffset;
ULONG Unknown;
LARGE_INTEGER CreationTime;
LARGE_INTEGER LastAccessTime;
LARGE_INTEGER LastWriteTime;
LARGE_INTEGER ChangeTime;
LARGE_INTEGER EndOfFile;
LARGE_INTEGER AllocationSize;
ULONG FileAttributes;
ULONG FileNameLength;
ULONG EaInformationLength;
UCHAR AlternateNameLength;
WCHAR AlternateName[12];
WCHAR FileName[1];
} FILE_BOTH_DIRECTORY_INFORMATION, *PFILE_BOTH_DIRECTORY_INFORMATION;
```

et pour FileNamesInformation:

```
typedef struct _FILE_NAMES_INFORMATION {
ULONG NextEntryOffset;
ULONG Unknown;
ULONG FileNameLength;
WCHAR FileName[1];
} FILE_NAMES_INFORMATION, *PFILE_NAMES_INFORMATION;
```

Ces fonctions écrivent une liste de ces structures dans FileInformation. Seulement trois variables sont importantes pour nous dans toutes ces types de structures.
NextEntryOffset représente la longueur d'une valeur particulière de la liste. La première valeur peut être trouvée à l'adresse FileInformation + 0. Et donc la deuxième sera à l'adresse FileInformation + NextEntryOffset de la première. Et la dernière valeur 0 comme valeur de NextEntryOffset.
FileName est le nom complet du fichier.

FileNameLength est la longueur du nom du fichier.

Si nous voulons cacher un fichier, nous devons prendre à part ces quatre types et, pour chaque enregistrement retourné, nous devons comparer son nom avec celui que nous souhaitons cacher. Si nous voulons cacher le premier enregistrement, nous devrons déplacer les structures suivantes de la taille de la première. Ceci entraînera que le premier enregistrement serait réecrit. Maintenant si nous voulons cacher un autre enregistrement, nous pouvons changer simplement la valeur de NextEntryOffset de l'enregistrement précédent. La nouvelle valeur serait 0 si nous voulons cacher le dernier enregistrement, sinon elle serait égale à la somme de NextEntryOffset de l'enregistrement que nous voulons caché et de l'enregistrement précédent.
Ensuite nous devons modifier la valeur du Unknown de l'enregistrement précédent qui doit être un index pour la recherche suivante. La valeur de Unknown de l'enregistrement précédent doit contenir la valeur de Unknown de l'enregistrement que nous voulons cacher.
Si aucun enregistrement qui devrait être trouvé l'a été, nous retournerons une erreur STATUS_NO_SUCH_FILE.

```
#define STATUS_NO_SUCH_FILE 0xC000000F
```

## =====[ 3.2 NtVdmControl ]=====

Pour une raison inconnue l'émulation DOS de NTVDM peut aussi obtenir une liste des fichiers avec la NtVdmContol.

```
NTSTATUS NtVdmControl(
IN ULONG ControlCode,
IN PVOID ControlData
);
```

ControlCode spécifie la sous-routine qui est appliquée sur les données du buffer ControlData. Si ControlCode est égal à VdmDirectoryFile, cette fonction fait le même que NtQueryDirectoryFile avec FileInformationClasségal à FileBothDirectoryInformation.

```
#define VdmDirectoryFile 6
```

Ensuite ControlData est utilisé comme FileInformation. la seule différence notable ici est que nous ne connaissons pas la longueur de ce buffer. Donc nous allons devoir la calculer manuellement. Nous devons ajouter NextEntryOffset de tous les enregistrements avec FileNameLength du dernier, ainsi que 0x5E comme longueur du dernier enregistrement (nom du fichier exclu). Les méthodes de camouflage deviennent alors les mêmes que pour NtQueryDirectoryFile.

## =====[ 4. Les processus ]=====

Des informations systèmes variées sont disponibles en utilisant NtQuerySystemInformation.

```
NTSTATUS NtQuerySystemInformation(
IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
IN OUT PVOID SystemInformation,
IN ULONG SystemInformationLength,
OUT PULONG ReturnLength OPTIONAL
);
```

SystemInformationClass spécifie le type d'information que nous voulons obtenir, SystemInformation est un pointeur sur la fonction du buffer de sortie, SystemInformationLength spécifie la longueur de ce buffer e ReturnLength est le nombre d'octets écrits.
Pour la liste de processus en cours d'exécution nous utilisons SystemInformationCl SystemProcessesAndThreadsInformation.

```
#define SystemInformationClass 5
```

La structure retournée dans le buffer SystemInformation est:

```
typedef struct _SYSTEM_PROCESSES {
ULONG NextEntryDelta;
ULONG ThreadCount;
ULONG Reserved1[6];
LARGE_INTEGER CreateTime;
LARGE_INTEGER UserTime;
LARGE_INTEGER KernelTime;
UNICODE_STRING ProcessName;
KPRIORITY BasePriority;
ULONG ProcessId;
ULONG InheritedFromProcessId;
ULONG HandleCount;
ULONG Reserved2[2];
VM_COUNTERS VmCounters;
IO_COUNTERS IoCounters; // Windows 2000 uniquement
SYSTEM_THREADS Threads[1];
} SYSTEM_PROCESSES, *PSYSTEM_PROCESSES;
```

Cacher des processus est similaire à cacher des fichiers.
Nous devons modifier NextEntryDelta de l'enregistrement précédent celui que nous souhaitons cacher. En général nous ne souhaiterons pas cacher le premier puisqu'il s'agit du processus Idle.

## =====[ 5. La base de Registre ]=====

La base de registre Windows est composé d'une arborescence assez grande et contenant deux types d'enregistrement intéressants pour nous dans le but de les cacher. Le premier est les clés registre, et le second les valeurs. Les structures du registre pour cacher des clés n'est pas aussi simple que pour cacher des fichiers processus comme nous allons le voir.

## =====[ 5.1 NtEnumerateKey ]=====

De part sa structure nous ne pourrons pas demander une liste de toutes les clés dans une partie spécifique de la base de registre. Nous pouvons seulement obtenir des informations sur une clé spécifique par son index dans une partie du registre. Ceci est possible avec NtEnumerateKey.

```
NTSTATUS NtEnumerateKey(
IN HANDLE KeyHandle,
IN ULONG Index,
IN KEY_INFORMATION_CLASS KeyInformationClass,
OUT PVOID KeyInformation,
IN ULONG KeyInformationLength,
OUT PULONG ResultLength
);
```

KeyHandle est un handle vers une clé dont nous voulons obetnir des informations sur une sous-clé spécifiée par Index. L'information retournée est d'un type spécifié par KeyInformationClass. Les données sont écrites dans le buffer KeyInformation dont la longueur est KeyInformationLength.
Le nombre d'octets écrits est renvoyé par ResultLength.
La chose la plus importante à bien percevoir est que si nous cachons une clé, les index de toutes les clés suivantes seront faux. Et puisque nous sommes capables d'obtenir des informations sur une clé qui a un index plus haut en demandant à une clé dont l'index est plus bas, nous devons toujours compter combien d'enregistrements ont été cachés avant puis retourner le bon.
Prenons un petit exemple. Supposons que nous avons des clés appelées A, B, C, D, E et F à n'importe quel endroit du registre. L'indexation commence à zero ce qui signifie que le numéro d'index 4 correspond à la clé E.
Maintenant si nous voulons cacher la clé B et que l'application hookée appelle NtEnumerateKey avec l'Index 4,

nous devons retourner les informations sur la clé F puisqu'il va y avoir un décalage de l'indexation. Le problème est que nous ne savons pas qu'il y a un décalage. Et si nous laissons le décalage de côté et retournons E à la place de F quand nous demandons la clé d'index 4 nous ne retournerions rien en demandant la clé d'index 1 ou nous retournerions C. De toute façon, dans les deux cas c'est une erreur. C'est pourquoi nous devons faire attention au décalage.
Maintenant si nous comptons les décalages en appelant à nouveau la fonction pour chaque index à partir de 0 jusqu'à Index nous attenderions pas mal de temps (sur un processeur 1Ghz ca pourrait prendre 10 secondes ce qui est déjà beaucoup trop). Donc il nous faut trouver une méthode plus évoluée.
Nous savons que les clés sont (sauf les références) classées alphabétiquement. Si nous laissons de côté les références (que nous n'avons pas besoin de cacher) nous pouvons compter le décalage avec la méthode suivante.
Nous classerons alphabétiquement notre liste des noms de clés que nous voulons cacher (RtlCompareUnicodeString peut être employer), puis quand l'application appelera NtEnumerateKey nous l'appellerons pas à nouveau avec des arguments inchangés mais nous trouverons le nom de l'enregistrement spécifié par Index.

```
NTSTATUS RtlCompareUnicodeString(
IN PUNICODE_STRING String1,
IN PUNICODE_STRING String2,
IN BOOLEAN CaseInSensitive
);
```

String1 et String2 sont les chaînes qui seront comparées, CaseInSensitive est sur True si voulons comparer sa s'occuper de la casse des caractères (minuscules et majuscules). Le résultat de la fonction nous donnera la relation entre String1 et String2:

résultat > 0: String1 > String2
résultat = 0: String1 = String2
résultat < 0: String1 < String2

Maintenant nous devons trouver les frontières limites. Nous comparerons alphabétiquement le nom de la clé spécifiée par Index avec les noms de notre liste. Les limites seront les derniers noms de notre liste. Nous savons que le décalage est au plus égal au nombre de frontières de notre liste. Mais tous les objets de notre liste n'ont pas besoin d'être des clés
valides dans la partie du registre où nous sommes. Donc, pour toutes les valeurs de notre liste jusqu'à la limite, nous allons devoir demander si elles sont dans cette partie du registre. Cela est faisable en utlisant NtOpenKey.

```
NTSTATUS NtOpenKey(
OUT PHANDLE KeyHandle,
IN ACCESS_MASK DesiredAccess,
IN POBJECT_ATTRIBUTES ObjectAttributes
);
```

KeyHandle est un handle de clé ordinaire. Nous utiliserons la valeur renvoyée par NtEnumerateKey pou DesiredAccess sont les droits d'accès.
KEY_ENUMERATE_SUB_KEYS est la bonne valeur à utiliser. ObjectAttributes décrit les sous-clés que nou voulons ouvrir (avec leurs noms).

```
#define KEY_ENUMERATE_SUB_KEYS 8
```

Si le résultat de NtOpenKey est 0, l'ouverture a réussi, ce qui signifie que cette clé existe bien dans notre liste.
La clé ouverte doit être refermée avec NtClose.

```
NTSTATUS NtClose(
IN HANDLE Handle
);
```

Pour chaque appel de NtEnumareteKey, nous compterons le décalage comme un nombre de clésde notre liste

qui existe dans une partie donnée du registre.
Puis nous ajouterons le décalage de l'argument Index et finalement appellerons la fonction NtEnumerateKey originale.
Pour obtenir le nom d'une clé spécifiée par Index nous utiliserons la valeur de KeyBasicInformation comme une info de type KeyInformationClass.

```
#define KeyBasicInformation 0
```

NtEnumerateKey retourne cette structure dans KeyInformation:

```
typedef struct _KEY_BASIC_INFORMATION {
LARGE_INTEGER LastWriteTime;
ULONG TitleIndex;
ULONG NameLength;
WCHAR Name[1];
} KEY_BASIC_INFORMATION, *PKEY_BASIC_INFORMATION;
```

La seule chose dont nous avons besoin ici est le nom et la longueur (Name et NamLength).
S'il n'y pas d'entrée pour l'Index décalé nous retournerons l'erreur STATUS_EA_LIST_INCONSISTENT.

```
#define STATUS_EA_LIST_INCONSISTENT 0x80000014
```

## =====[ 5.2 NtEnumerateValueKey ]=====

Les valeurs du registre ne sont pas classées alphabétiquement. Par chance le nombre de valeurs d'une clé est assez petit, donc nous pouvons utiliser la méthode de rappel pour obtenir le décalage. L'API pour obtenir des infos sur une valeur est appelée NtEnumerateValueKey.

```
NTSTATUS NtEnumerateValueKey(
IN HANDLE KeyHandle,
IN ULONG Index,
IN KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,
OUT PVOID KeyValueInformation,
IN ULONG KeyValueInformationLength,
OUT PULONG ResultLength
);
```

KeyHandle est encore le handle de la clé ordinaire. Index est un index de la liste de valeurs de KeyValueInformationClass représente un type d'information qui sera stocké dans le buffer KeyValueInformation qui est long de KeyValueInformationLength octets. Le nombre d'octets écrits est renvoyé par ResultLength.
Encore une fois nous allons devoir compter le décalage mais suivant le nombre de valeurs dans une clé nous pouvons rappeler cette fonction pour tous les index de 0 jusqu'à Index. Le nom de la valeur peut être obtenu lorsque KeyValueInformationClass pointe sur KeyValueBasicInformation.

```
#define KeyValueBasicInformation 0
```

Nous obtiendrons alors la structure suivante dans le buffer KeyValueInformation:

```
typedef struct _KEY_VALUE_BASIC_INFORMATION {
ULONG TitleIndex;
ULONG Type;
ULONG NameLength;
WCHAR Name[1];
} KEY_VALUE_BASIC_INFORMATION, *PKEY_VALUE_BASIC_INFORMATION;
```

Nous sommes de nouveau uniquement intéressés par Name et NameLength.

S'il n'y pas d'entées pour l'Index décalé nous retournerons l'erreur STATUS_NO_MORE_ENTRIES.

#define STATUS_NO_MORE_ENTRIES 0x8000001A

## =====[ 6. Les services et pilotes système ]=====

Les services systèmes et pilotes sont listés par quatre API indépendantes. Leur utilisation est différente dans chaque version de Windows. C'est pour cela qu'il va falloir hooker les quatre fonctions.

```
BOOL EnumServicesStatusA(
SC_HANDLE hSCManager,
DWORD dwServiceType,
DWORD dwServiceState,
LPENUM_SERVICE_STATUS lpServices,
DWORD cbBufSize,
LPDWORD pcbBytesNeeded,
LPDWORD lpServicesReturned,
LPDWORD lpResumeHandle
);

BOOL EnumServiceGroupW(
SC_HANDLE hSCManager,
DWORD dwServiceType,
DWORD dwServiceState,
LPBYTE lpServices,
DWORD cbBufSize,
LPDWORD pcbBytesNeeded,
LPDWORD lpServicesReturned,
LPDWORD lpResumeHandle,
DWORD dwUnknown
);

BOOL EnumServicesStatusExA(
SC_HANDLE hSCManager,
SC_ENUM_TYPE InfoLevel,
DWORD dwServiceType,
DWORD dwServiceState,
LPBYTE lpServices,
DWORD cbBufSize,
LPDWORD pcbBytesNeeded,
LPDWORD lpServicesReturned,
LPDWORD lpResumeHandle,
LPCTSTR pszGroupName
);

BOOL EnumServicesStatusExW(
SC_HANDLE hSCManager,
SC_ENUM_TYPE InfoLevel,
DWORD dwServiceType,
DWORD dwServiceState,
LPBYTE lpServices,
DWORD cbBufSize,
LPDWORD pcbBytesNeeded,
LPDWORD lpServicesReturned,
LPDWORD lpResumeHandle,
LPCTSTR pszGroupName
```

);

Le plus important ici est lpServices qui pointe sur le buffer où la liste de services sera stockée. Et a lpServicesReturned qui pointe sur le nombre d'enregistrements retournés. La structure des données dans le buffer de sortie dépend du type de la fonction. Pour les fonctions EnumServicesStatusA et EnumServicesGroupW la structure renvoyée:

```
typedef struct _ENUM_SERVICE_STATUS {
LPTSTR lpServiceName;
LPTSTR lpDisplayName;
SERVICE_STATUS ServiceStatus;
} ENUM_SERVICE_STATUS, *LPENUM_SERVICE_STATUS;

typedef struct _SERVICE_STATUS {
DWORD dwServiceType;
DWORD dwCurrentState;
DWORD dwControlsAccepted;
DWORD dwWin32ExitCode;
DWORD dwServiceSpecificExitCode;
DWORD dwCheckPoint;
DWORD dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

pour EnumServicesStatusExA et EnumServicesStatusExW ce sera:

```
typedef struct _ENUM_SERVICE_STATUS_PROCESS {
LPTSTR lpServiceName;
LPTSTR lpDisplayName;
SERVICE_STATUS_PROCESS ServiceStatusProcess;
} ENUM_SERVICE_STATUS_PROCESS, *LPENUM_SERVICE_STATUS_PROCESS;

typedef struct _SERVICE_STATUS_PROCESS {
DWORD dwServiceType;
DWORD dwCurrentState;
DWORD dwControlsAccepted;
DWORD dwWin32ExitCode;
DWORD dwServiceSpecificExitCode;
DWORD dwCheckPoint;
DWORD dwWaitHint;
DWORD dwProcessId;
DWORD dwServiceFlags;
} SERVICE_STATUS_PROCESS, *LPSERVICE_STATUS_PROCESS;
```

Nous sommes intéressés uniquement par lpServiceName qui est le nom du service NT. Les enregistrements ont des tailles statiques, donc si nous voulons en cacher un, nous déplacerons tous les enregistrements suivants par leurs tailles. Nous devons différencier les tailles de SERVICE_STATUS et SERVICE_STATUS_PROCESS.

## =====[ 7. Hooker et propager ]=====

Pour obtenir l'effet désiré nous devons hooker tous les processus en cours d'exécution mais aussi tous les processus qui seront créés plus tard.
Les nouveaux processus devront être hookés avant même de lancer leur première instruction de leur propre code sinon ils seraient capables de voir nos objets cachés dans le temps avant d'être hookés.

## =====[ 7.1 Les droits ]=====

Premièrement il est bon de savoir que nous avons besoin des droits administrateurs au minimum pour avoir

accès à tous les processus en cours d'exécution. La meilleure façon est de lancer notre processus en service l'exécutera avec les droits SYSTEM. Pour installer le service nous avons Obtenir le SeDebugPrivilege est aussi assez pratique. Cela peut être réalisé en utilisant les API OpenProcessToken, LookupPrivilegeVa AdjustTokenPrivileges.

```
BOOL OpenProcessToken(
HANDLE ProcessHandle,
DWORD DesiredAccess,
PHANDLE TokenHandle
);

BOOL LookupPrivilegeValue(
LPCTSTR lpSystemName,
LPCTSTR lpName,
PLUID lpLuid
);

BOOL AdjustTokenPrivileges(
HANDLE TokenHandle,
BOOL DisableAllPrivileges,
PTOKEN_PRIVILEGES NewState,
DWORD BufferLength,
PTOKEN_PRIVILEGES PreviousState,
PDWORD ReturnLength
);
```

En négligeant les erreurs le code ressemble à ça:

```
#define SE_PRIVILEGE_ENABLED 0x0002
#define TOKEN_QUERY 0x0008
#define TOKEN_ADJUST_PRIVILEGES 0x0020

HANDLE hToken;
LUID DebugNameValue;
TOKEN_PRIVILEGES Privileges;
DWORD dwRet;

OpenProcessToken(GetCurrentProcess(),
TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,hToken);
LookupPrivilegeValue(NULL,"SeDebugPrivilege",&DebugNameValue);
Privileges.PrivilegeCount=1;
Privileges.Privileges[0].Luid=DebugNameValue;
Privileges.Privileges[0].Attributes=SE_PRIVILEGE_ENABLED;
AdjustTokenPrivileges(hToken,FALSE,&Privileges,sizeof(Privileges),
NULL,&dwRet);
CloseHandle(hToken);
```

## =====[ 7.2 le hook global ]=====

L'énumération des processus est réalisée par l'API NtQuerySystemInformation comme nous l'avons déjà vu. Il y a quelques processus natifs dans le système, donc nous utiliserons la méthode qui consiste à réécrire les premières instructions de la fonction pour les hooker. Nous ferons de même pour chaque processus lancé. N allouerons une partie de mémoire dans le processus cible où nous écrirons notre nouveau code pour les fonctions que nous voulons hooker. Puis nous changerons les cinq premiers octets de ces fonctions en instruction jmp. Ce saut redirigera l'exécution du code.
Donc l'instruction jmp sera exécutée immédiatement quand la fonction hookée sera appelée. Nous devons sauvegarder les premières instructions pour chaque fonction écrasée. Nous en avons besoin pour appeler le code original des fonctions hookées. La sauvegarde des instructions est détaillée dans le chapitre 3.2.3 du docume

"Hooking Windows API".
This jump will redirect the execution to our code. En premier, nous devons ouvrir le processus cible vi NtOpenProcess et obtenir un handle. Ceci ne marchera pas si nous n'avons pas les droits suffisants.

```
NTSTATUS NtOpenProcess(
OUT PHANDLE ProcessHandle,
IN ACCESS_MASK DesiredAccess,
IN POBJECT_ATTRIBUTES ObjectAttributes,
IN PCLIENT_ID ClientId OPTIONAL
);
```

ProcessHandle est un pointeur sur un handle où le résultat est stocké.
DesiredAccess doit être sur PROCESS_ALL_ACCESS. Nous mettrons le PID du processus cible sur la UniqueProcess de la structure ClientId, et UniqueThread doit être sur 0.
Le handle ouvert peut être fermé dans tous les cas par NtClose.

```
#define PROCESS_ALL_ACCESS 0x001F0FFF
```

Maintenant nous allons allouer la partie de mémoire pour notre code. On peut faire cela en NtAllocateVirtualMemory.

```
NTSTATUS NtAllocateVirtualMemory(
IN HANDLE ProcessHandle,
IN OUT PVOID BaseAddress,
IN ULONG ZeroBits,
IN OUT PULONG AllocationSize,
IN ULONG AllocationType,
IN ULONG Protect
);
```

ProcessHandle est celui de NtOpenProcess. BaseAddress est un pointeur sur un pointeur du début où nous voulons allouer la mémoire. L'adresse de la mémoire allouée sera stockée dedans. La valeur Input peut être NULL.
AllocationSize est le pointeur sur le nombre d'octets que nous voulons allouer. Et, encore une fois, c'est aussi utilisé comme valeur de sortie pour le nombre d'octets alloués réels. Il est bien de mettre Allocatio MEM_TOP_DOWN en plus de MEM_COMMIT car la mémoire serait allouée sur l'adresse la plus grande possible à côté des DLLs.

```
#define MEM_COMMIT 0x00001000
#define MEM_TOP_DOWN 0x00100000
```

Puis nous pouvons écrire notre code avec NtWriteVirtualMemory.

```
NTSTATUS NtWriteVirtualMemory(
IN HANDLE ProcessHandle,
IN PVOID BaseAddress,
IN PVOID Buffer,
IN ULONG BufferLength,
OUT PULONG ReturnLength OPTIONAL
);
```

BaseAddress sera l'adresse renvoyée par NtAllocateVirtualMemory.
Buffer pointe sur les octets que nous voulons écrire, BufferLength est le nombre d'octets que nous voulons écrire.

Maintenant il nous faut hooker les fonctions. La seule librairie chargée dans tous les processus est ntdll.dll. Donc nous devons vérifier si la fonction que nous voulons hooker est importée par le processus si ce n'est pas une provenant de ntdll.dll. Mais la mémoire où serait cette fonction (d'une autre dll) pourrait être allouée, donc

le fait de réécrire des octets sur son adresse pourrait facilement être la cause d'erreurs dans le processus cible. C'est pourquoi nous devons bien vérifier si la librairie (où la fonction que nous désirons hooker se trouve) est chargée dans le processus cible.
Nous avons besoin du PEB (Process Environment Block) du processus cible via NtQueryInformationProcess.

```
NTSTATUS NtQueryInformationProcess(
IN HANDLE ProcessHandle,
IN PROCESSINFOCLASS ProcessInformationClass,
OUT PVOID ProcessInformation,
IN ULONG ProcessInformationLength,
OUT PULONG ReturnLength OPTIONAL
);
```

Nous mettrons ProcessInfromationClass sur la valeur ProcessBasicInformation. Puis PROCESS_BASIC_INFORMATION sera renvoyée sur le buffer ProcessInformation dont la taille est donnée par ProcessInformationLength.

```
#define ProcessBasicInformation 0

typedef struct _PROCESS_BASIC_INFORMATION {
NTSTATUS ExitStatus;
PPEB PebBaseAddress;
KAFFINITY AffinityMask;
KPRIORITY BasePriority;
ULONG UniqueProcessId;
ULONG InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION, *PPROCESS_BASIC_INFORMATION;
```

PebBaseAddress correspond à ce que nous cherchions. On trouve l'adresse de PPEB_LDR_DAT PebBaseAddress+0x0C. On obtiendrait cela en appelant NtReadVirtualMemory.

```
NTSTATUS NtReadVirtualMemory(
IN HANDLE ProcessHandle,
IN PVOID BaseAddress,
OUT PVOID Buffer,
IN ULONG BufferLength,
OUT PULONG ReturnLength OPTIONAL
);
```

Les paramètres sont similaires à NtWriteVirtualMemory.
On a l'adresse de InInitializationOrderModuleList en PPEB_LDR_DATA+0x1C.
C'est une liste de librairies chargées par le processus. Une seule partie de cette structure va nous intéresser.

```
typedef struct _IN_INITIALIZATION_ORDER_MODULE_LIST {
PVOID Next,
PVOID Prev,
DWORD ImageBase,
DWORD ImageEntry,
DWORD ImageSize,
...
);
```

Next est un pointeur sur l'enregistrement suivant, Prev sur le précédent, le dernier enregistrement pointe sur le premier. ImageBase est est une adresse du module en mémoire, ImageEntry est le point d'entrée du module, ImageSize sa taille.

Pour toutes les librairies dans lesquelles nous voulons placer des hooks (je dirai aussi "hooker" par facilité), nous aurons besoin de leur ImageBase (en utilisant GetModuleHandle et LoadLibrary). Nous comparerons cette

ImageBase avec celle de chaque entrée dans InInitializationOrderModuleList.
Nous sommes maintenant prêts à hooker. Puisque nous placons des hooks dans des processus en cours d'exécution, il y a une chance pour que le code soit exécuté au moment où nous allons le réécrire. Cela peut causer des erreurs, donc nous allons arrêter tous les threads du processus cible tout d'abord. La liste de ses threads peut être obtenue via NtQuerySystemInformation avec la classe SystemProcessesAndThreadsInformation. Le résultat de cette fonction est décrit dans le chapître 4. Mais nous devons encore ajouter la description de la structure SYSTEM_THREADS où est l'information sur les threads.

```
typedef struct _SYSTEM_THREADS {
LARGE_INTEGER KernelTime;
LARGE_INTEGER UserTime;
LARGE_INTEGER CreateTime;
ULONG WaitTime;
PVOID StartAddress;
CLIENT_ID ClientId;
KPRIORITY Priority;
KPRIORITY BasePriority;
ULONG ContextSwitchCount;
THREAD_STATE State;
KWAIT_REASON WaitReason;
} SYSTEM_THREADS, *PSYSTEM_THREADS;
```

Pour chaque thread nous devons obtenir son handle avec NtOpenThread.
Nous utiliserons ClientId pour ça.

```
NTSTATUS NtOpenThread(
OUT PHANDLE ThreadHandle,
IN ACCESS_MASK DesiredAccess,
IN POBJECT_ATTRIBUTES ObjectAttributes,
IN PCLIENT_ID ClientId
);
```

Le handle qu'on veut sera stocké dans ThreadHandle. On mettra DesiredAccess sur THREAD_SUSPEND_RESUME.

```
#define THREAD_SUSPEND_RESUME 2
```

ThreadHandle sera utilisé pour appeler NtSuspendThread.

```
NTSTATUS NtSuspendThread(
IN HANDLE ThreadHandle,
OUT PULONG PreviousSuspendCount OPTIONAL
);
```

Le processus suspendu est prêt pour la réécriture. Nous procéderons comme expliqué dans le chapître 3.2.2 d "Hooking Windows API". La seule différence sera d'utiliser des fonctions pour d'autres processus.

Après un hook nous rétablirons tous les threads du processus en appelant NtResumeThread.

```
NTSTATUS NtResumeThread(
IN HANDLE ThreadHandle,
OUT PULONG PreviousSuspendCount OPTIONAL
);
```

## III. Conclusion du tutorial

Ce tutoriel comporte deux parties :
voir la deuxième partie du tutoriel sur les rootkits et l'invisibilité sur Windows NT

Ajouté le 26-02-2006
Lu 789 fois
Tutoriel réalisé par : Holy_Father & Damien
Pas de support par email, veuillez utiliser le forum informatique pour toute question.