Rootkit : invisibilité sur Windows NT (partie 2/2)

I. Introduction aux Rootkit

Comment ne pas se faire voir sur Windows NT
Partie 2 sur 2

Version Originale par: Holy_Father

II. Explications

=====[7.3 Les nouveaux processus]=====

L'injection dans tous les processus lancés c'est bien beau, mais il nous reste les processus qui pourraient être lancés plus tard. On pourrait récupérer la liste des processus pour la comparer avec l'ancienne apres un petit moment, de façon à détecter les nouveaux processus et les infecter.
Mais cette méthode n'est pas très fiable. Une meilleure est de hooker les fonctions qui sont toujours appelées quand un nouveau processus démarre. Puisque que nous hookons déjà tous les processus lancés, on ne peut en manquer aucun nouveau avec cette méthode. on pourrait hooker NtCreateThread mais ce n'est pas la manière la plus simple de le faire. On fera ca avec NtResumeThread qui est appelé aussi à chaque fois qu'un nouveau processus est créé. Son appel vient après NtCreateThread.
Le seul problème avec NtResumeThread est que ce n'est pas uniquement appelé quand un nouveau processus démarré. Mais on peut arranger ça.
NtQueryInformationThread nous donnera une information sur l'appartenance d'un thread spécifique à un processus. La dernière chose qu'il nous reste à faire est de vérifier si le processus est contient déjà des hooks ou non. On peut le faire en lisant le premier octet de n'importe quelle fonction que nous hookons.

```
NTSTATUS NtQueryInformationThread(
IN HANDLE ThreadHandle,
IN THREADINFOCLASS ThreadInformationClass,
OUT PVOID ThreadInformation,
IN ULONG ThreadInformationLength,
OUT PULONG ReturnLength OPTIONAL
);
```

ThreadInformationClass est une classe d'informations est doit être sur ThreadBasicinformation dans notre cas. ThreadInformation est le buffer prévu pour le résultat dont la taille est ThreadInformationLength octets.

```
#define ThreadBasicInformation 0
```

Cette structure est renvoyée pour la classe ThreadBasicInformation:

```
typedef struct _THREAD_BASIC_INFORMATION {
NTSTATUS ExitStatus;
PNT_TIB TebBaseAddress;
CLIENT_ID ClientId;
KAFFINITY AffinityMask;
KPRIORITY Priority;
KPRIORITY BasePriority;
} THREAD_BASIC_INFORMATION, *PTHREAD_BASIC_INFORMATION;
```

Dans ClientId on trouve le PID de celui à qui appartient le thread.

Maintenant il nous faut infecter le nouveau processus. Le problème est que le nouveau processus n'a que ntdll.dll dans sa mémoire. Tous les autres modules sont chargés après avoir appelé NtResumeThread. Il y a plusieurs façons de parer à ce problème. Par exemple on peut hooker l'API appelée LdrInitializeThunk qui est appelée pendant l'initialisation du processus.

```
NTSTATUS LdrInitializeThunk(
DWORD Unknown1,
DWORD Unknown2,
DWORD Unknown3
);
```

Tout d'abord on exécutera le code original et ensuite on hookera toutes les fonctions qu'on veut dans ce nouveau processus. Mais il sera mieux de retirer le hook sur LdrInitializeThunk parce qu'elle est appelée plusieurs fois par la suite et on ne vaps tout rehooker encore. Tout cela est fait avant l'exécution de la première instruction de l'application. C'est pourquoi il n'y aucune chance que ca appelle une fonction à hooker avant qu'on place le hook.
Le hook en lui-même fonctionne de la même façon que quand on hookait des processus lancés sauf qu'on ne se préoccupe pas des threads lancés ici.

=====[7.4 DLL]=====

Dans tous les processus du système on trouve une copie de ntdll.dll.
Ce que signifie que l'on peut hooker n'importe quelle fonction de ce module dans l'initialisation d processus. Mais pour les fonctions provenant d'autres modules tels kernel32.dll ou advapi32.dll? Et il y a aussi pas mal de processus qui ont uniquement ntdll.dll. Tous les autres modules peuvent être chargés dynamiquement dans le milieu du code après le hook du processus.
C'est pourquoi nous hookerons LdrLoadDll qui chargent les nouveaux modules.

```
NTSTATUS LdrLoadDll(
PWSTR szcwPath,
PDWORD pdwLdrErr,
PUNICODE_STRING pUniModuleName,
PHINSTANCE pResultInstance
);
```

Le plus important pour nous ici est pUniModuleName qui est le nom du module. pResultInstanc contiendra son adresse si l'appel a réussi.

On appelera le LdrLoadDll original puis on hookera toutes les fonctions dans le module chargé.

=====[8. La mémoire]=====

Lorsque nous hookons une fonction, nous modifions ses premiers octets.
En appelant NtReadVirtualMemory n'importe qui peut détecter qu'une fonction est hookée. Donc nous allons hooker NtReadVirtualMemory à son tour pour empêcher la détection.

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

Nous avons modifier des octets dans le début de toutes les fonctions que nous avons hookées et nous avons aussi alloué de la mémoire pour notre nouveau morceau de code. On doit vérifier si l'appelant lit une partie de ces octets. Si nous avons nos octets dans une partie allant de BaseAddress à BaseAddress + BufferLength nous devons changer quelques octets dans Buffer.
Si des octets de notre mémoire allouée sont demandés, nous devons renvoyer un Buffer vide et une erreur STATUS_PARTIAL_COPY. Cette valeur dit que tous les octets demandés n'ont pas été copiés dans le buffer. C'est aussi utilisé quand une demande de mémoire non allouée est faite. ReturnLength doit être égal à 0 dans ce cas là.

```
#define STATUS_PARTIAL_COPY 0x8000000D
```

Si l'on demande les premiers octets d'une fonction hookée, nous devons appelé le code original et ensuite on doit copier les octets originaux (on les a sauvegardés pour les appels originaux) vers le Buffer.
Maintenant le processus est incapable de détecter qu'il est hooké en lisant sa mémoire. Aussi, si v débugguez des processus hookés, le débuggueur aura un lèger problème. Il va montrer les octets originaux mais exécuter notre code.

Pour cacher cela parfaitement on peut aussi hooker NtQueryVirtualMemory. Cette fonction est utilisée pour récupérer des infos sur la mémoire virtuelle. On peut la hooker pour empêcher la détection de notre mémoire allouée.

```
NTSTATUS NtQueryVirtualMemory(
IN HANDLE ProcessHandle,
IN PVOID BaseAddress,
IN MEMORY_INFORMATION_CLASS MemoryInformationClass,
OUT PVOID MemoryInformation,
IN ULONG MemoryInformationLength,
OUT PULONG ReturnLength OPTIONAL
);
```

MemoryInformationClass spécifie la classe de données retournée.
Les deux premiers types sont intéressants pour nous.

```
#define MemoryBasicInformation 0
#define MemoryWorkingSetList 1
```

Cette structure est renvoyée pour la classe MemoryBasicInformation:

```
typedef struct _MEMORY_BASIC_INFORMATION {
PVOID BaseAddress;
PVOID AllocationBase;
ULONG AllocationProtect;
ULONG RegionSize;
ULONG State;
ULONG Protect;
ULONG Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

Chaque partie de mémoire possède sa taille RegionSize ses types Type. La mémoire libre est d MEM_FREE.

```
#define MEM_FREE 0x10000
```

Si une section a le type MEM_FREE avant la nôtre, on devra ajouter la taille de notre section à RegionSize. Si la section suivante est aussi de type MEM_FREE on devra ajouter la taille de la section suivante encore une fois à cette RegionSize.
Si une section avant la nôtre est d'un autre type, on renverra MEM_FREE pour notre section. Sa taille est encore comptée suivant les sections suivantes.

Voilà la structure renvoyée pour la classe MemoryWorkingSetList:

```
typedef struct _MEMORY_WORKING_SET_LIST {
ULONG NumberOfPages;
ULONG WorkingSetList[1];
} MEMORY_WORKING_SET_LIST, *PMEMORY_WORKING_SET_LIST;
```

NumberOfPages est le nombre de valeurs dans WorkingSetList. Ce nombre doit être décrémenté. On devrait trouver nos sections dans WorkingSetList et déplacer les enregistrements suivants après les nôtres.
WorkingSetList est un tableau de DWORDs où les 20 octets de poid fort (les plus grands) représentent les 20 octets de poid fort de la section adresse et les 12 octets de poid faible spécifient les flags (J'utilise flags aussi en français par abus de language personnellement mais c'est "drapeaux" au sens littéraire).

=====[9. Handle]=====

Appeler NtQuerySystemInformation avec une classe de type SystemHandleInformation nous donne un tableau de tous les handles ouverts dans la structure _SYSTEM_HANDLE_INFORMATION_EX.

```
#define SystemHandleInformation 0x10
```

```
typedef struct _SYSTEM_HANDLE_INFORMATION {
ULONG ProcessId;
UCHAR ObjectTypeNumber;
UCHAR Flags;
USHORT Handle;
PVOID Object;
ACCESS_MASK GrantedAccess;
```

```
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;

typedef struct _SYSTEM_HANDLE_INFORMATION_EX {
ULONG NumberOfHandles;
SYSTEM_HANDLE_INFORMATION Information[1];
} SYSTEM_HANDLE_INFORMATION_EX, *PSYSTEM_HANDLE_INFORMATION_EX;
```

ProcessId désigne le processus auquel apparatient le handle.
ObjectTypeNumber est le type du handle. NumberOfHandles est le nombre d'enregistrements dans le tableau Information. Cacher une valeur est enfantin. Nous devons supprimer tous les enregistrements suivants un par un et décrémenter NumberOfHandles. La suppression de tous les suivants est requise car les handles sont groupés par ProcessId dans le tableau.
Ce qui signifie que tous les handles d'un simple processus sont ensembles. Et pour un processus le nombre Handle croît.
Maintenant souvenez vous de la structure _SYSTEM_PROCESSES qui est renvoyée par cette fonction avec la classe SystemProcessesAndThreadsInformation.
On peut voir ici que chaque processus connaît sont nombre de handles via HandleCount. Si nous voulons faire parfait, nous devons modifier HandleCount suivant le nombre de handles cachés en appelant cette fonction avec la classe SystemProcessesAndThreadsInformation. Mais cette correction prendrait pas mal de temps processeur. Il y a pas mal de handles ouverts et fermés dans un très court temps pendant une exécution normale du système. Donc il peut se produire que le nombre de handles est changé entre deux appels de cette fonction et nous n'avons pas besoin de changer HandleCount.

=====[9.1 Nommer le handle et obtenir le type]=====

Cacher un handle est simple mais trouver quel handle cacher est un peu plus compliqué. Si nous avons, par exemple des processus cachés, nous devons cacher tous ses handles et tous les handles reliés à lui. Cacher les handles d'un processus est encore une fois assez simple. On va juste comparer le ProcessId du handle et le PID de notre processus, et cacher ça quand ils sont égaux. Mais les handles des autres processus doivent être nommés avant que l'on puisse comparer quelque chose. Le nombre de handles dans le système est en général énorme, donc le meilleur moyen est de comparer le type du handle avant de le nommer. Nommer les types fera gagner pas mal de temps pour les handles qui ne nous intéressent pas.
On peut nommer le handle et son type en appelant NtqueryObject.

```
NTSTATUS ZwQueryObject(
IN HANDLE ObjectHandle,
IN OBJECT_INFORMATION_CLASS ObjectInformationClass,
OUT PVOID ObjectInformation,
IN ULONG ObjectInformationLength,
OUT PULONG ReturnLength OPTIONAL
);
```

ObjectHandle est le handle dont nous voulons récupérer des infos, ObjectInformationClass est le type d'informations qui sera stocké dans le buffer ObjectInformation qui est long de ObjectInformationLength octets.
On utilisera ObjectNameInformation et ObjectAllTypesInformation.
La classe ObjectNameInformation remplira le buffer avec une structure OBJECT_NAME_INFORMATION, et la classe ObjectAllTypesInformation avec une structure de type OBJECT_ALL_TYPES_INFORMATION.

```
#define ObjectNameInformation 1
```

```
#define ObjectAllTypesInformation 3

typedef struct _OBJECT_NAME_INFORMATION {
UNICODE_STRING Name;
} OBJECT_NAME_INFORMATION, *POBJECT_NAME_INFORMATION;
```

Name représente le nom du handle.

```
typedef struct _OBJECT_TYPE_INFORMATION {
UNICODE_STRING Name;
ULONG ObjectCount;
ULONG HandleCount;
ULONG Reserved1[4];
ULONG PeakObjectCount;
ULONG PeakHandleCount;
ULONG Reserved2[4];
ULONG InvalidAttributes;
GENERIC_MAPPING GenericMapping;
ULONG ValidAccess;
UCHAR Unknown;
BOOLEAN MaintainHandleDatabase;
POOL_TYPE PoolType;
ULONG PagedPoolUsage;
ULONG NonPagedPoolUsage;
} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;

typedef struct _OBJECT_ALL_TYPES_INFORMATION {
ULONG NumberOfTypes;
OBJECT_TYPE_INFORMATION TypeInformation;
} OBJECT_ALL_TYPES_INFORMATION, *POBJECT_ALL_TYPES_INFORMATION;
```

Name représente le nom du type d'objet qui suit immédiatement chaque OBJECT_TYPE_INFORMATION. La structure OBJECT_TYPE_INFORMATION suivante succède à Name, commence sur le premier paquet de quatre octets.

ObjectTypeNumber de la structure SYSTEM_HANDLE_INFORMATION est un index sur un tableau de TypeInformation.

Le plus dur est de récupérer un handle à partir d'un autre processus.
Il y a deux manières de le nommer. La première est de copier le handle via NtDuplicateObject vers notre processus et puis de le nommer. Cette méthode échouera pour certains types de handles. Ma seulement pour un petit nombre d'entre eux, donc pas de panique on peut l'utiliser.

```
NtDuplicateObject(
IN HANDLE SourceProcessHandle,
IN HANDLE SourceHandle,
IN HANDLE TargetProcessHandle,
OUT PHANDLE TargetHandle OPTIONAL,
IN ACCESS_MASK DesiredAccess,
IN ULONG Attributes,
IN ULONG Options
);
```

SourceProcessHandle est un handle du processus auquel appartient SourceHandle, qui lui-même est le handle que nous voulons copier.
TargetProcessHandle est le handle du processus de destination où l'on va copier. Ce sera le handle de notre processus dans notre cas.
TargetHandle est le pointeur sur le handle où l'on va garder une copie du handle original. DesiredAccess doit être sur PROCESS_QUERY_INFORMATION, et Attributes et Options sur 0.

La seconde méthode pour nommer qui marche avec tous les handles est d'utiliser un pilote système. Le code source est dispo dans le projet OpHandle sur le site http://www.hxdef.org.

=====[10. Ports]=====

Le manière la plus simple de lister les ports est d'utiliser les fonctions à AllocateAndGetTcpTableFromStack et AllocateAndGetUdpTableFromStack, AllocateAndGetTcpExTableFromStack et AllocateAndGetUdpExTableFromStack de iphlpapi.dll. Les fonctions Ex sont disponibles depuis Windows XP.

```
typedef struct _MIB_TCPROW {
DWORD dwState;
DWORD dwLocalAddr;
DWORD dwLocalPort;
DWORD dwRemoteAddr;
DWORD dwRemotePort;
} MIB_TCPROW, *PMIB_TCPROW;

typedef struct _MIB_TCPTABLE {
DWORD dwNumEntries;
MIB_TCPROW table[ANY_SIZE];
} MIB_TCPTABLE, *PMIB_TCPTABLE;

typedef struct _MIB_UDPROW {
DWORD dwLocalAddr;
DWORD dwLocalPort;
} MIB_UDPROW, *PMIB_UDPROW;

typedef struct _MIB_UDPTABLE {
DWORD dwNumEntries;
MIB_UDPROW table[ANY_SIZE];
} MIB_UDPTABLE, *PMIB_UDPTABLE;

typedef struct _MIB_TCPROW_EX
{
DWORD dwState;
DWORD dwLocalAddr;
DWORD dwLocalPort;
DWORD dwRemoteAddr;
DWORD dwRemotePort;
DWORD dwProcessId;
} MIB_TCPROW_EX, *PMIB_TCPROW_EX;

typedef struct _MIB_TCPTABLE_EX
{
DWORD dwNumEntries;
```

```
MIB_TCPROW_EX table[ANY_SIZE];
} MIB_TCPTABLE_EX, *PMIB_TCPTABLE_EX;

typedef struct _MIB_UDPROW_EX
{
DWORD dwLocalAddr;
DWORD dwLocalPort;
DWORD dwProcessId;
} MIB_UDPROW_EX, *PMIB_UDPROW_EX;

typedef struct _MIB_UDPTABLE_EX
{
DWORD dwNumEntries;
MIB_UDPROW_EX table[ANY_SIZE];
} MIB_UDPTABLE_EX, *PMIB_UDPTABLE_EX;

DWORD WINAPI AllocateAndGetTcpTableFromStack(
OUT PMIB_TCPTABLE *pTcpTable,
IN BOOL bOrder,
IN HANDLE hAllocHeap,
IN DWORD dwAllocFlags,
IN DWORD dwProtocolVersion;
);

DWORD WINAPI AllocateAndGetUdpTableFromStack(
OUT PMIB_UDPTABLE *pUdpTable,
IN BOOL bOrder,
IN HANDLE hAllocHeap,
IN DWORD dwAllocFlags,
IN DWORD dwProtocolVersion;
);

DWORD WINAPI AllocateAndGetTcpExTableFromStack(
OUT PMIB_TCPTABLE_EX *pTcpTableEx,
IN BOOL bOrder,
IN HANDLE hAllocHeap,
IN DWORD dwAllocFlags,
IN DWORD dwProtocolVersion;
);

DWORD WINAPI AllocateAndGetUdpExTableFromStack(
OUT PMIB_UDPTABLE_EX *pUdpTableEx,
IN BOOL bOrder,
IN HANDLE hAllocHeap,
IN DWORD dwAllocFlags,
IN DWORD dwProtocolVersion;
);
```

Il y a une autre façon de faire le même. Quand un programme créé une socket et la met en écoute, il a un handle de la socket et du port ouverts pour sûr. On peut lister tous les handles ouverts du système et leur envoyer un buffer spécial via NtDeviceIoControlFile pour voir si c'est un handle pour un port ouvert ou non. Cela nous donnera aussi des informations sur le port.
Vu qu'il y a beaucoup de handles ouverts, nous testerons seulement ceux dont le type est File et le

nom \Device\Tcp ou \Device\Udp. Les ports ouverts ont seulement ce type et ce nom.

Quand on regarde au code des fonctions iphlpapi.dll ci-dessus, on voit que ces fonctions appellents aussi NtDeviceIoControlFile et envoient un buffer spécial pour obtenir une liste de tous les ports ouverts du système. Ce qui signifie que la seule fonction à hooker pour cacher les po NtDeviceIoControlFile.

```
NTSTATUS NtDeviceIoControlFile(
IN HANDLE FileHandle
IN HANDLE Event OPTIONAL,
IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
IN PVOID ApcContext OPTIONAL,
OUT PIO_STATUS_BLOCK IoStatusBlock,
IN ULONG IoControlCode,
IN PVOID InputBuffer OPTIONAL,
IN ULONG InputBufferLength,
OUT PVOID OutputBuffer OPTIONAL,
IN ULONG OutputBufferLength
);
```

Les arguments intéressants pour nous sont FileHandle qui spécifie un handle du périphérique avec lequel communiquer, IoStatusBlock qui pointe sur une variable qui reçoit l'état d'avancement final et les informations sur l'opération demandée, IoControlCode qui est un nombre spécifiant le type périphérique, la méthode, l'accès aux fichiers et une fonction.
InputBuffer contient les données d'entrée qui sont longues de InputBufferLength octets et idem OutputBuffer et OutputbufferLength.

=====[10.1 Netstat, OpPorts sur WinXP, FPort sur WinXP]=====

Récupérer une liste de tous les ports ouverts est la première chose utilisée par OpPorts, FPort sur Windows XP mais aussi netstat par exemple.
Les programmes appellent NtDeviceIoControlFile deux fois avec IoControlCode 0x0001200(OutputBuffer est rempli après un deuxième appel.
Le nom de FileHandle est toujours \Device\Tcp ici. InputBuffer diffère pour différents types d'appels:

1) Pour récupérer un tableau de MIB_TCPROW InputBuffer ressemble à ça:

premier appel:
0x00 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01 0x00 0x00
0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00

deuxième appel:
0x00 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01 0x00 0x00
0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00

2) Pour récupérer un tableau de MIB_UDPROW:

premier appel:

0x01 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01 0x00 0x00
0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00

deuxième appel:
0x01 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01 0x00 0x00
0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00

3) Pour récupérer un tableau de MIB_TCPROW_EX:

premier appel:
0x00 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01 0x00 0x00
0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00

deuxième appel:
0x00 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01 0x00 0x00
0x02 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00

4) Pour récupérer un tableau de MIB_UDPROW_EX:

premier appel:
0x01 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01 0x00 0x00
0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00

deuxième appel:
0x01 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01 0x00 0x00
0x02 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00

Vous pouvez voir que ces buffers diffèrent seulement sur quelques octets. Avec un peu de lucidité on peut récapituler comme ca:

Les appels qui nous intéressent ont InputBuffer[1] égal à 0x04 et généralement InputBuffer[17] sur 0x01. Seulement après ces données d'entrée nous remplirons le OutputBuffer avec les tables souhaitées. Si nous voulons avoir des infos sur des ports TCP on mettra InputBuffer[0] sur 0x00, ou sur 0x01 pour des infos sur UDP. Si nous voulons des tables de sortie étendues (MIB_TCPROW_EX or MIB_UDPROW_EX) on utilisera Inputbuffer[16] sur 0x02 dans le deuxième appel.

Si nous trouvons l'appel avec ces paramètres, nous pouvons modifier le buffer de sortie. Pour obtenir le nombre de colonnes dans le buffer de sortie, il faut simplement diviser Information de IoStatusBlock par la taille de la colonne. Cacher une colonne devient alors facile. Il faut juste la réécrire avec les colonnes suivantes et supprimer la dernière. N'oubliez pas de changer OutputBufferLength et IoStatusBlock.

=====[10.2 OpPorts sur Win2k et NT4, FPort sur Win2k]=====

On utilise NtDeviceIoControlFile avec IoControlCode 0x00210012 pour déterminer si le handle du

type File ainsi que les noms \Device\Tcp ou \Device\Udp est le handle du port ouvert.

Donc en premier nous comparerons IoControlCode et puis un type et le nom du handle. si c'est intéressant alors on compare la longueur du buffer d'entrée qui doit être égale à la longueur de la structure TDI_CONNECTION_IN.
Cette longueur est 0x18. OutputBuffer est sur TDI_CONNECTION_OUT.

```
typedef struct _TDI_CONNECTION_IN
{
ULONG UserDataLength,
PVOID UserData,
ULONG OptionsLength,
PVOID Options,
ULONG RemoteAddressLength,
PVOID RemoteAddress
} TDI_CONNECTION_IN, *PTDI_CONNECTION_IN;

typedef struct _TDI_CONNECTION_OUT
{
ULONG State,
ULONG Event,
ULONG TransmittedTsdus,
ULONG ReceivedTsdus,
ULONG TransmissionErrors,
ULONG ReceiveErrors,
LARGE_INTEGER Throughput
LARGE_INTEGER Delay,
ULONG SendBufferSize,
ULONG ReceiveBufferSize,
ULONG Unreliable,
ULONG Unknown1[5],
USHORT Unknown2
} TDI_CONNECTION_OUT, *PTDI_CONNECTION_OUT;
```

Une implémentation concrète de la façon de déterminer le handle d'un port ouvert est disponible dans le code source de OpPorts sur http://www.hxdef.org. Maintenant ce qui nous intéresse c'est de cacher un port spécifique. On a déjà comparé InputBufferLength et IoControlCode.
Maintenant on va comparer RemoteAddressLength. On obtiendra toujours 3 ou 4 pour un port ouvert. La dernière chose que nous avons à faire est de comparer ReceivedTsdus du OutputBuffer qui contient le port en écriture réseay et notre liste de ports que nous voulons cacher. La différence entre TCP et UDP se fait en fonction du nom du handle. En supprimant OutputBuffer, en mo IoStatusBlock et en renvoyant la valeur STATUS_INVALID_ADDRESS ce port sera caché.

III. Conclusion du tutorial

L'implémentation complète des techniques décrites sera disponible avec le code source du rootkit hacker defender dans sa version 1.0.0 sur sa page http://www.hxdef.org et aussi sur http://www.rootkit.com.
Il est possible que je rajoute plus d'informations sur l'invisibilité sur Windows NT dans le futur. L nouvelles versions de ce document pourrait aussi contenir des améliorations à propos des méthodes

décrites ou des nouveaux commentaires.
Remerciements spécial à Ratter qui m'a fourni beaucoup de connaissances qui ont été nécessaires pour écrire ce document et pour programmer le projet de Hacker defender.
N'hésitez pas à envoyez toutes vos remarques holy_father@phreaker.net ou sur le forum de http://www.hxdef.org.
De même pour les remarques sur la traduction française du texte à damien @simagorad.com ou sur le forum de http://www.kachouri.com.

Ajouté le 26-02-2006
Lu 657 fois
Tutoriel réalisé par : Holy_Father & Damien
Pas de support par email, veuillez utiliser le forum informatique pour toute question.