KPCR se remet au sport :

Présentation du ring0 Windows pour un h4x0r-codeur-de-rootkit débutant

Par **KPCR**

Prologue écrit sous stupéfiants.

Au commencement il y avait Dieu, mais il s'emmerdait sévère alors il créa KPCR, ce dernier s'emmerda encore plus sévère car Dieu c'est juste une tantouze avec un corps super secure et sans orifices donc on ne peut pas le fingerprint ou le pénétrer. Alors Dieu créa les hommes qui créèrent les pc pour amuser KPCR. KPCR était censé devenir un 1337 qui coderait sans répit pour s'occuper. Il n'en fut rien, la faute à un jeune homme au corps ectomorphe non sans rappeler celui du serpent, du nom d'0vercl0k qui conduisit KPCR sur irc. Depuis ce jour KPCR ne fit que troller et n'écrivit plus à propos de l'informatique. Jusqu'à ce jour où je me décide enfin à m'armer de mon pénis, de mon cerveau et de mes dix doigts pour réécrire à propos de Windows.

Prologue normal.

On m'a souvent demandé ce que je trouvais intéressant dans le kernel land windows. Peu de gens étaient convaincus que ce soit vraiment passionnant. Je décide donc, profitant de ce projet de zine pour proposer une visite du kernel land de windows et de ses faiblesses. Le but ici n'est pas de faire une surenchère de niveau technique mais de faire découvrir à un plus grand nombre certains éléments du kernel land windows, et montrer que bosser à ce niveau pour des malwares peut s'avérer très intéressant et varié.

Connais l'adversaire et surtout connais toi toi-même et tu seras invincible. **Sun Tzu**

Connais Windows et balance dans leurs dans la gueule un low kick rotatif et tu roxxeras. **KPCR**

Et sur ce, musique !

Introduction : pourquoi s'intéresser au kernel land quand l'user land se fait si friendly ?

Si deux chemins s'offrent à toi, choisis le plus difficile. Proverbe d'un bouddhiste dont j'ai oublié le nom.

Il fut un temps où la programmation de malware était bien moins prise de tête qu'aujourd'hui et la notion d'indétection n'était que survolé car, il faut le dire, les malwares ne couraient pas les rues. Mais tout a augmenté dans le monde de l'informatique, aussi bien en positif qu'en négatif. Les antivirus et tout le tintouin ont donc du devenir de plus en plus attentif au système, aux modifications qui lui sont apportés, aux exécutables lancés …

Inutile de dire que cela n'arrange pas nos affaires. Il parait donc logique d'aller dans des voies plus méconnues et plus profonde dans l'espoir d'être UD.

Mais ne le cachons pas, il y a également une part de curiosité et d'envie de contrôler toujours plus d'éléments. Nous serions vraiment bêtes de nous en priver après tout ! Tout au long de cet article je vais parfois utiliser des outils tels kd ou la wdk, vous retrouverez des liens pour télécharger ces programmes à la fin de l'article.

Commençons la visite !

Ok, les gars, je serai votre guide pour la visite. Nous irons aux lieux permettant de répondre au mieux à vos questions. Vous êtes donc 5 pour faire la randonnée avec moi, tapz1, tapz2, tapz3, tapz4 et tapz5. Commençons, que vous voulez vous savoir, et par conséquent, que visiterons nous ?

tapz1 : On mange quand ?

KPCR : STFU

Tapz3 : Tu suces ?

KPCR : Nan mais c'est quoi ces randonneurs trolleurs ? Du sérieux !

Tapz2 : Bon ok, est ce que beaucoup de choses sont traités dans le kernel land de Windows ?

Enormément, si l'on veut schématiser au maximum, on peut dire que l'user land se contente d'envoyer le boulot au kernel land, comme un chef de projet délègue le travail à différentes personnes suivants leurs qualités.

Tapz4 : Mais pourtant quand on code en VB, ou en C basique avec des api Win, on est bien en user land alors que se passe-t-il, tu nous racontes de la merde dude ?

Pas du tout. C'est ça la bogossitude de Windows, si on utilise par exemple WriteFile, contenue dans kernel32.dll, et que l'on désassemble cette dll, avec Ida que j'apprécie pour ce qui est disas de dll par exemple, on peut voir que cela mène à un call qui lui-même nous emmène à une autre dll, ntdll.dll, pour appeler non plus WriteFile mais NtWriteFile, qui elle est une API Native. Bref, on arrive à l'endroit de la dll pour NtOpenFile mais ce n'est encore pas fini, en effet une instruction ASM va être utilisée, un SYSENTER, c'est une instruction assez importante donc je vais détailler son fonctionnement et son but. Elle a pour but d'amener à une routine ring0, KiSystemService, qui permet, elle de retrouver l'adresse de la fonction que l'on veut faire utiliser, il va aller chercher dans la SSDT. Nous reviendrons à cela en détail plus tard mais là il est l'heure du premier point h4x0r de cet article en expliquant le fonctionnement de SYSENTER.

Le point h4x0r n°1 : SYSENTER HOOKING (modification de MSR)

Je disais donc que SYSENTER était une instruction assembleur clé car elle permet la liaison entre l'user land et le kernel land, ce qui est essentiel pour touts nos chers programmes et leurs fonctions, APIs, toussa toussa. Reprenons l'exemple de WriteFile, il a donc besoin pour être lancé d'aller jusqu'à la SSDT qui est en ring0, mais on ne passe pas en ring0 comme ça, car des changements sont effectués et encore faut-il savoir où aller ! Pour cela SYSENTER va utiliser une notion x86 assez connue, les MSR, diminutif de Model Specific Register, qui contiennent des adresses, ces MSR sont utilisés pour des manipulations au niveau du proco.

En l'occurrence SYSENTER utilise trois MSR : MSR_SYSENTER_ESP, MSR_SYSENTER_CS et MSR_SYSENTER_EIP. Ceux qui connaissent l'assembleur devineront vite que ces msr contiennent des adresses de registres. Rappelons donc que le registre Cs contient l'adresse de l'endroit où sont les instructions asm à lancer, que le registre esp avec sp pour Stack Pointer indique donc le sommet de la pile et enfin l'eip IP signifiant Instruction pointer contient l'adresse de la prochaine instruction à exécuter, dans notre cas KiFastCallEntry. Et là vous vous dîtes, « Pitin ça serait bien si on pouvait modifier le MSR_SYSENTER_EIP pour que ça pointe vers une autre instruction que KiFastCallEntry, une instruction à moi comme ça on exécuterai notre code en ring0 ». Et bien on peut. Grâce à NtDebugSystemControl :

```
NtSystemDebugControl(
DEBUG_CONTROL_CODE ControlCode,
PVOID InputBuffer,
ULONG InputBufferLength,
PVOID OutputBuffer,
ULONG OutputBufferLength,
PULONG ReturnLength
);
```

Cette API a de nombreuses utilités, elle permet à la base le debugging d'éléments kernel land, et donc de lire et agir sur de nombreuses données, dont les msr. A noter que cette API peut être intéressante à manipuler dans des optiques de forensic mais c'est un autre sujet.

Pour votre coding vous aurez besoin du typedef de la structure MSR

```
typedef struct _MSR_STRUCT {
        DWORD MsrNum;                   // MSR number
        DWORD NotUsed;                  // Never accessed by the kernel
        DWORD MsrLo;                    // IN (write) or OUT (read): Low 32
bits of MSR
        DWORD MsrHi;                    // IN (write) or OUT (read): High 32
bits of MSR
} MSR_STRUCT;
```

Que j'ai trouvé dans ce code http://archives.neohapsis.com/archives/vulnwatch/2004-q1/att-0041/xploit_dbg.cpp

C'est une manipulation en somme très intéressante et où il suffit d'utiliser NtSystemDebugControl. Et vu que nous sommes dans un esprit « white h4t 4 ev4 » je donne le moyen de détecter cette action, il suffit avec le kernel debugger d'effectuer une lecture du MSR_SYSENTER_EIP (rdmsr à l'adresse du msr) pour voir si l'eip pointe toujours vers KiFastCallSystemEntry ou non.

Revenons à nos moutons.

Avant d'expliquer le principe du SYSENTER hooking je vous expliquais que le long trajet d'une fonction s'arrêtait à la SSDT.
Un peu de cours sur la SSDT ne fait pas de mal.
Et le cours est contenu dans

Le point h4x0r n°2: SSDT Hooking

Il y a quelques temps j'avais déjà écrit un article sur le SSDT Hooking qui était complet et expliquait bien ce qu'est la SSDT et comment la soumettre à nous. Je ne vais pas copier/coller l'article entier qui est relativement long mais vous invite plutôt à aller voir ce lien :
http://kpcr.blogspot.com/2008/03/prsentation-du-ssdt-hooking.html

Cependant il peut être intéressant de rajouter que certains anti-rootkits peuvent apparemment être bernés en utilisant un trick supplémentaire, je vous laisse juge en allant ici http://www.rootkit.com/newsread.php?newsid=922

Bon les randonneurs, maintenant que j'ai répondu à votre question tout en dérivant à mort du sujet pour vous apportez un maximum de connaissances avec vous d'autres questions ?

Tapz5 : Il y a beaucoup d'informations contenues dans le ring0 je suppose ?

Oh que oui. Sur absolument tout ce que tu peux imaginer. Prenons un exemple tout bête, lorsqu'un programmeur ou un logiciel comme le gestionnaire de fin de tâche veut obtenir diverses informations, il utilise l'api NtQuerySystemInformation ou (son équivalent ring0 en Zw*). Et bien cette api native n'échappe pas à ce que je vous ai expliqué plus haut, avec ntdll, les sysenter et tout le tintouin, à la différence que le trajet ici ne s'arrête pas à la SSDT, on va ensuite interroger ExpGetProcessInformation qui va ensuite aller parcourir des listes sur divers sujets pour vous ramener les informations demandées.
Et c'est avec ce sujet que j'introduis

Le point h4x0r n°3: le DKOM

Il y a deux lignes je vous expliquais que des systèmes de listes existaient en kernel land pour diverses informations. Il serait donc utile de pouvoir modifier ces listes pour ensuite pouvoir cacher des informations aux yeux des utilisateurs et logiciels. Pour cela on va utiliser le DKOM, abréviation de Direct Kernel Object Manipulation. J'avais rédigé pour la section réservée d'europa security un mini article/tutoriel sur le concept de DKOM avec un exemple. Il vous aidera à comprendre le DKOM.
Le voici :
«
I-Le DKOM c'est quoi ?

"Tout d'abord DKOM ça veut dire quoi ?"
-DKOM est l'abréviation de Direct Kernel Object Manipulation.

"Ok, mais à quoi ça sert père Castor ?"
-Cela consiste à manipuler directement les objets du kernel. Un monde sans loi où règne la terreur car on manipule de nombreuses choses non documentées inutile donc de dire qu'il faut agir sur une machine virtuelle et non sur son os principal. et de bien observer ce que l'on va modifier pour ne pas avoir de mauvaises surprises.

"Encore plus concrètement ça sert à quoi ?"
On s'en sert principalement pour cacher des choses :
L'exemple le plus simple et également le connu est de cacher un processus (nous verrons cela en détail tout à l'heure source à l'appui) aux yeux de la plupart des utilisateurs, des connaissances étant ensuite nécessaire pour trouver le processus.
-On peut également envisager de cacher des LKM en modifiant la double liste chainée PsLoadedModuleList. : On trouve le MODULE_ENTRY du LKM que l'on veut hide et on modifie les pointeurs des MODULE_ENTRY qui précèdent et succèdent et c'est good car quand on veut récupérer la liste des mod chargés on utilise NtQuerySystemInformation ou ZwQuerySystemInformation avec un SYSTEM_INFOMATION_CLASS de valeur SystemModuleInformation qui va aller utiliser la PsLoadedModuleList !
-On peut également donner des privilèges à un thread (ou à un simple processus) et ce sans pour autant avoir des privilèges comme TOKEN_ADJUST_GROUPS et TOKEN_ADJUST_PRIVILEGES.

En bref c'est super utile et ça roxx la choucroute william saurin !

II- Cacher un processus grâce au DKOM.

C'est à mes yeux la manipulation la plus simple et "marrante" à réaliser quand on débute avec le DKOM donc on va se pencher là dessus.
Tout d'abord un point de cours.
On ne pourra pas réaliser tout en kernel land.
En effet il faut un tout Pitti programme user land qui s'occupera de communiquer à notre driver kernel land le nom du process à hide.
Evidemment notre Exe ne va pas communiquer le nom du processus au .Sys avec des signaux de fumée, il va nous falloir utiliser les IOCTLs (diminutif de I/O Control Codes) : en effet on enverra un IOCTL à notre .Sys avec en argument le nom du processus à cacher en bref on fera une chaine de caractères.
Une fois que notre driver aura le nom du processus, il faudra bien savoir quoi modifier.
Et c'est là qu'il faut un peu de reverse pour comprendre où les informations sont trouvées par le système.
Pour obtenir la liste des processus ainsi que d'autres détails sur eux, il faut faire appel à NtQuerySystemInformation() en mettant à InformationClass une valeur SystemProcessInformation.
A partir de ça la demande sera gérée par ExpGetProcessInformation() qui va lui même regarder dans la PsActiveProcessHead.
En toute logique il faut donc modifier la PsActiveProcessHead pour cacher notre processus.
Mais pour savoir comment faire il faut tout d'abord un point de cours sur la PsActiveProcessHead.

La PsActiveProcessHead est une double liste chainée référençant des structures EPROCESS, chaque structure EPROCESS représente un processus.
Une structure EPROCESS contient une structure LIST_ENTRY.
On jette à coup d'œil au type def de cette structure

```
typedef struct _LIST_ENTRY {
   struct _LIST_ENTRY *Flink;
   struct _LIST_ENTRY *Blink;
} LIST_ENTRY,*PLIST_ENTRY;
```

On voit que la structure LIST_ENTRY contient d'étrange chose : FLINK et BLINK.
FLINK et BLINK sont des membres qui sont des pointeurs menant respectivement au processus précédant et au processus suivant.
Il faudra donc modifier les membres FLINK et BLINK de sorte à ce que lorsque ExpGetProcessInformation cherche dans la liste il saute notre processus et ça sera tout bon.

Bon maintenant que toute la marche à suivre est expliquée il ne me reste plus qu'à vous donner diverses informations sur comment programmer tout ça.

Commençons par le programme user land qui est simplissime.
Il vous faudra bien entendu inclure les librairies pour utiliser tout ce qui a attrait à windows et aux IOCTLs.
C'est à dire :
windows.h et winioctl.h.
Il faut également string.h pour gérer les chaines de caractères.
Tout d'abord, on prépare notre chaine de caractère (celle que j'avais précédemment expliquée).
On crée un handle de type hDevice sur notre driver avec CreateFile (on n'oublie pas la définition préalable "HANDLE hDevice;")
Puis on envoie un IOCTL contenant notre chaine de caractère à hDevice avec l'api DeviceIoControl.
Et c'est tout pour le programme ring3

Maintenant la partie kernel land, déjà plus complexe :
Il faut la wdk évidemment donc les librairies
wdm.h et string.h pour manipuler les chaines de caractères
On commence notre driver par la routine d'initialisation, un DriverEntry, point essentiel pour les drivers, mon ami tr00ps qui aime les trojans appelleraient ça le point d'entrée du driver, en effet lorsque l'IOCTL est envoyé à notre driver qui est dans notre cas un Device il va chercher à contacter le DriverEntry.
Pour ceux qui ne sauraient pas comment écrire ce DriverEntry voici la routine écrite ici :

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPath)
{
code
}
```

Comme son nom l'indique pDriverObject est un pointeur, pointant sur une structure DRIVER_OBJECT qui n'est autre que l'image de notre driver chargé (si vous voulez créer une fonction de déchargement du driver il vous faudra utiliser DriverUnload de cette structure).
pRegristryPath est lui aussi un pointeur, pointant sur une string contenant le chemin du fichier, si ma mémoire est bonne c'est une chaine unicode .
On récupère grâce à l'IOCTL le nom du processus à cacher.
Pour cela on utilise IoGetCurrentIrpStackLocation() (PIO_STACK_LOCATION, donc un pointeur vers la structure IO_STACK_LOCATION. On va ensuite voir dans cette structure le membre Parameters.DeviceIoControl.IoControlCode et hop c'est bon)

Le "nom" processus est sous forme d'une chaine de caractère.
On la converti en Unicode à l'aide de l'api RtlAnsiStringToUnicodeString()
Il va nous falloir ensuite accéder à la PsActiveProcessHead.
Pour cela on va utiliser IoGetCurrentProcess qui est un api qui renvoie un pointeur sur le processus et donc sur une struct EPROCESS !
Ensuite on utilise ActiveProcessLinks qui est un membre permettant de "se balader" dans les différents EPROCESS.
On compare notre chaine de caractère aux ImageFileName des EPROCESS :
Si les chaines sont identiques on a plus qu'à faire une simple modification de pointeurs : on modifie les flinks et blinks des processus entourant celui que l'on veut cacher et c'est bon le processus sera caché.
Sinon on continue de parcourir les EPROCESS, une simple condition if else quoi.

III-Méthode miracle ?

Non bien sur, un œil averti peu se rendre compte de la mascarade en scannant la table des PID, la PspCidTable, mais là encore on peut modifier cette table pour enlever le pid de notre process, comme c'est le cas dans le rootkit FuTo.
Mais on peut aussi voir le processus au niveau du thread scheduler ou tout simplement en scannant la HandleTable. La encore des méthodes existent pour cacher à cet endroit.
Tout est question d'imagination et de patience. »

Revenons à nos moutons.
Bon, changeons un peu de musique.
Nous en étions donc à visiter le kernel land dans le but de présenter les différentes informations qu'il contient, nous avons donc vu qu'il contenait différentes listes chainées contenant des informations diverses et variées, sur les processus, modules etc.
Mais il ne s'arrête pas là pour ce qui est stockage de l'information, en effet il existe aussi des structures qui servent uniquement au stockage d'informations et d'adresses, on citera bien

évidemment le KPCR. C'est d'ailleurs l'occasion pour moi d'expliquer le choix de mon pseudo, à l'époque j'étais passionné par Windows (ça n'a pas trop changé me direz vous) et j'apprenais petit à petit des trucs sur le kernel land en lisant différents forums. Et forcément un jour j'ai entendu parler du KPCR, qui en plus d'être une structure relativement importante du ring0 est également le point d'entrée kernel land (à l'adresse 0xFFDFF000 ou encore FS :0x01c qui emmène au champ SelfPcr du KPCR) donc je me suis dis, ce pseudo sera mon entrée in d4 w0rld de l'informatique. Fin de l'anecdote et regardons ce que contient ce fameux KPCR en utilisant l'instruction dt_KPCR. Voilà ce que l'on obtient :

```
lkd> dt_KPCR
nt!_KPCR
   +0x000 NtTib            : _NT_TIB
   +0x01c SelfPcr          : Ptr32 _KPCR
   +0x020 Prcb             : Ptr32 _KPRCB
   +0x024 Irql             : UChar
   +0x028 IRR              : Uint4B
   +0x02c IrrActive        : Uint4B
   +0x030 IDR              : Uint4B
   +0x034 KdVersionBlock   : Ptr32 Void
   +0x038 IDT              : Ptr32 _KIDTENTRY
   +0x03c GDT              : Ptr32 _KGDTENTRY
   +0x040 TSS              : Ptr32 _KTSS
   +0x044 MajorVersion     : Uint2B
   +0x046 MinorVersion     : Uint2B
   +0x048 SetMember        : Uint4B
   +0x04c StallScaleFactor : Uint4B
   +0x050 DebugActive      : UChar
   +0x051 Number           : UChar
   +0x052 Spare0           : UChar
   +0x053 SecondLevelCacheAssociativity : UChar
   +0x054 VdmAlert         : Uint4B
   +0x058 KernelReserved   : [14] Uint4B
   +0x090 SecondLevelCacheSize : Uint4B
   +0x094 HalReserved      : [16] Uint4B
   +0x0d4 InterruptMode    : Uint4B
   +0x0d8 Spare1           : UChar
   +0x0dc KernelReserved2  : [17] Uint4B
   +0x120 PrcbData         : _KPRCB
```

On voit dans le premier champ une struct NT_TIB. Un petit coup de DT nous permet de voir ce qu'elle contient, les noms sont plutôt évocateurs et nous montre un lien clair avec la pile ring0, jugez par vous-même :

```
lkd> dt_NT_TIB
nt!_NT_TIB
   +0x000 ExceptionList    : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 StackBase        : Ptr32 Void
   +0x008 StackLimit       : Ptr32 Void
   +0x00c SubSystemTib     : Ptr32 Void
   +0x010 FiberData        : Ptr32 Void
   +0x010 Version          : Uint4B
   +0x014 ArbitraryUserPointer : Ptr32 Void
   +0x018 Self             : Ptr32 _NT_TIB
```

On voit ensuite SelfPCR, qui contient l'adresse du KPCR, c'est ici que vous arrivez avec un FS:0x01c. Passons à des notions plus intéressantes qui se matérialisent sous nos yeux avec Irql, IRR, IrrActive, IDR et IDT.

Les notions d'IRQL, IRR, IrrActive, IDR et IDT gravitent autour d'une même notion, la notion d'interruptions, aussi bien matérielles que logicielles.

Bon là je vois déjà vos têtes en train de vous dire « ah gné wtf bbq ». Calmez-vous, l'objectif de ce tutoriel est de vous introduire dans ce monde en douceur, je vais donc expliquer tout cela.

Une interruption matérielle est provoqué comme son nom l'indique par du matériel. Pour vous expliquer de manière la plus claire prenons un exemple, le cas d'école étant le clavier.

Vous ne le savez peut-être pas, mais dès que vous appuyez sur une touche de votre clavier, un signal électrique se déclenche en direction du processeur, plus précisément une broche du nom d'IRQ, diminutif d'interrupt request qui va provoquer une interruption
L'OS va détecter cela et en fonction du type d'interruption provoquée, l'os va déléguer la tache d'aller retrouver le SCANCODE en questionnant le clavier. Le clavier possède un système de buffer aussi bien en INPUT qu'OUPUT, les INPUT_BUFFER permettent de recevoir des commandes, au contraire les OUTPUT_BUFFER permettent d'envoyer des informations, en l'occurrence il va envoyer le SCANCODE. Mais à quoi servira le SCANCODE ? Et bien à être « traduit » pour savoir sur quelle touche l'utilisateur a appuyé. Et tout ça presque instantanément. Balaise hein ?
Une interruption logicielle, quand à elle est provoquée par un programme et non un périphérique mais a le même but qu'une interruption matérielle, l'exécution du code du logiciel sera stoppé un court instant, pour exécuter une routine exactement comme c'est le cas dans l'exemple du clavier, pour ensuite reprendre le cours de l'exécution du code du logiciel.
Il est également important de préciser qu'il existe une « hiérarchie » des IRQ, en effet chaque IRQ a ce qu'on appelle un IRQL, ça vous rappelle quelque chose vu dans le KPCR hein ? ^^ IRQL signifie évidemment Interrupt Request Level. Le niveau le plus bas est le PASSIVE_LEVEL, c'est celui alloué pour les programmes users vient ensuite l'APC_LEVEL, APC signifiant ici Asynchronous Procedure Call, l'APC est un mécanisme très utilisé dans la programmation nécessitant une communication entre threads. Cet IRQL est destiné pour ce genre de logiciels. A niveau supérieur vient ensuite le DISPATCH_LEVEL qui a pour avantage de permettre un accès à la mémoire utilisateur et paginée. Ensuite vient les DIRQL_LEVEL qui sont destiné aux interruptions matérielles. Sans surprise on voit donc que la priorité est donnée au hardware.
Et c'est là que vient

Le point h4x0r n°4: IDT Hooking

Vous savez maintenant ce qu'est une interruption matérielle ou logicielle et leurs priorités. Il est indéniable qu'il serait très utile de pouvoir les maitriser.
Encore faut-il savoir où attaquer ? Et bien au niveau de l'IDT, l'IDT étant l'endroit où sont centralisées les handlers des fonctions servant à gérer ces interruptions.
Là encore, dans un esprit similaire au SSDT Hooking, il suffit de changer un handler pour un handler vers une fonction à nous et cela nous permettra d'avoir notre code en kernel land.
Le microblog de nibbles propose un petit code sympathique fait par mon ami 0vercl0k qui montre la marche à suivre très bien

Sh0ck : Bon, KPCR la tapz, déjà que tu utilises mon pseudo pour introduire des questions débiles, tu pourrais te magner de finir ton article.
KPCR : OK, dernière question.
Tapz1 : Comment ça se passe pour communiquer avec l'extérieur en kernel land ?

Très bonne question. Contrairement en user land où 90% des gens utilisent la même méthode pour communiquer avec l'extérieur, c'est-à-dire utiliser winsock, en kernel land il existe plusieurs interfaces pour communiquer vers l'extérieur.
Il en existe actuellement trois.
-Les Kernels sockets, uniquement depuis vista, système comparable à winsock et tdi.
-TDI, diminutif de transport driver interface. Qui disparaitra à partir de windows 7.
-NDIS, diminutif de network driver interface **s**pécification.
Le classement ci-dessus classe les différentes interfaces par niveau de difficulté du plus facile à utiliser au plus dur, c'est également l'ordre du plus haut niveau au plus bas niveau.
Mais qu'en est-il au niveau de la transparence au niveau des pare-feux ?

Le pare feu intégré à windows ne tarde pas à être largué dès que l'interface tdi entre dans la partie, le firewall windows utilise ipfilter or tdi fonctionnant avec tcpip.sys, situé plus bas qu'ipfilter, tdi bypass le firewall windows et la plupart des firewalls programmés par des amateurs, qui ont tendance à utiliser le concept de filter hook drivers.
Cependant les pare-feux « stars »comme look n' stop, ZA ou kerio ne s'arrêtent pas là.
Pour contrecarrer les malheureux h4x0rs qui auraient l'idée d'utiliser tdi ou ndis les firewalls vont aller se nicher encore plus bas de sorte à contrôler tout ce qui se passe.
Mais où ?
Pour répondre à cette question, un russe au doux pseudo de MaD a écrit un immense article à ce sujet nous expliquant que la plupart des firewalls se placent au niveau d'NDIS pour ensuite hooker des handlers contenus dans les structures NDIS_PROTOCOL_BLOCK et NDIS_OPEN_BLOCK.
En bref, ils sont combatifs ces braves firewalls et une indectabilité à 100% est impossible acquérir.
Mais je suppose qu'il est possible d'outrepasser ce problème au niveau des NDIS_PROTOCOL_BLOCK et NDIS_OPEN_BLOCK. Comment ? Surement en reproduisant leur comportement et donc changer à notre tour les handlers ! Même si il me semble que les firewalls renouvellent ses handlers….

Conclusion de l'article :

Si vous ne connaissiez pas le monde du kernel land windows avant cette article et que vous êtes intéressés en ce moment même, alors l'objectif de cet article est rempli.
Si vous voulez vous lancer dans ce domaine après cela, l'objectif est DOUBLEMENT rempli.
Cet article est évidemment incomplet tant le monde du kernel land est vaste.
Je voudrais remercier quelques personnes :
-Xylitol
-v00d00chile
-0vercl0k
-PHPLizardo
-Dorian
-Tr00ps
-Sh0ck
-fyury

Pour ceux qui voudraient continuer l'aventure kernel land voici quelques liens de qualités :
http://www.rootkit.com/
http://www.ivanlef0u.fr
http://0vercl0k.fr/
https://www.openrce.org/
http://www.uninformed.org/
http://www.phrack.com/

Et une pensée toute particulière pour quelqu'un qui m'a quitté moi et ce monde bien trop tôt.
Puisse-t-il vivre mieux là où il est.