

# Trouver kernel32.dll

Jerome Athias

[HTTP://WWW.ATHIAS.FR](http://www.athias.fr)

Du fait que parler directement au kernel n'est pas une option, une solution alternative est nécessaire. La seule autre manière de parler au noyau est de passer par une API existante sur la machine Windows. Sur Windows, comme les variantes Unix, les APIs standards du mode utilisateur sont exportées sous la forme d'objets chargés dynamiquement qui sont mappés dans l'espace d'un processus durant son exécution. Ces types d'objets sont appelés Shared Objects (.so) soit Objets Partagés, ou dans le cas de Windows Dynamically Linked Library (.dll) (Bibliothèques de Liaison Dynamique). Le choix d'une DLL est très simple sur Windows car il n'y en a qu'une seule dont on soit sûr qu'elle soit partagée dans l'espace d'un processus, si le binaire n'est pas lié statiquement ; kernel32.dll.

*ntdll.dll n'est pas traitée par souci de simplicité*

Kernel32.dll offre la possibilité, requise pour assurer la fiabilité et portabilité d'un shellcode, de réaliser des actions comme l'accès à un port, le téléchargement de fichier... et ce, via les fonctions LoadLibraryA et GetProcAddress.

La fonction LoadLibraryA, comme son nom l'indique, permet de charger une DLL.

```
WINBASEAPI HMODULE WINAPI LoadLibraryA(LPCSTR lpLibFileName);
```

LoadLibraryA utilise la convention d'appel et accepte une pointeur chaîne constant sur le nom de fichier du module à charger, retournant l'adresse de base (sous la forme d'un pointeur void) du module chargé en cas de succès.

Nous discuterons plus tard la résolution des adresses de symboles.

Malheureusement, il existe un problème récurrent à l'utilisation de kernel32.dll, c'est qu'elle n'est pas chargée à la même adresse suivant la version de Windows. (cette adresse peut être paramétrée en utilisant l'utilitaire rebase.exe).

Cela implique que l'on ne peut pas inclure les adresses des fonctions de kernel32.dll en dur dans le shellcode sans perdre en portabilité.

Nous allons aborder trois méthodes pour pallier à ce problème.

# PEB

Cibles: Windows 95/98/ME/NT/2K/XP

Taille: 34 octets

Cette première technique du PEB (Process Environment Block ) est documentée dans l'excellent document de The Last Stage of Delerium (<http://www.lsd-pl.net/documents/winasm-1.0.1.pdf>).

C'est de loin la technique la plus fiable à utiliser pour déterminer l'adresse de base de kernel32.dll. Le seul désavantage est que c'est également la plus gourmande vis à vis de la taille du code requis ; 34 octets pour une version fonctionnant avec Windows 9x et Windows NT.

Le système d'exploitation alloue une structure pour chaque processus s'exécutant qui peut toujours être trouvée en fs:[0x30] dans le processus.

La structure PEB contient des informations sur les tas du processus, l'image binaire, et, le plus important, trois listes chaînées sur les modules chargés qui ont été mappés dans l'espace du processus.

Les listes chaînées elles-mêmes diffèrent dans les fonctionnalités à montrer l'ordre dans lequel les modules ont été chargés, à l'ordre dans lequel les modules ont été initialisés. La liste chaînée de l'ordre d'initialisation est la plus intéressante car l'ordre dans lequel kernel32.dll est initialisée est toujours constant ; c'est toujours le second module à être initialisé. En parcourant cette liste, l'on peut donc extraire l'adresse de base de kernel32.dll de la seconde entrée.

## Code assembleur:

find\_kernel32:

```
    push esi
    xor eax, eax
    mov eax, fs:[eax+0x30]
    test eax, eax
    js find_kernel32_9x
```

find\_kernel32\_nt:

```
    mov eax, [eax + 0x0c]
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    jmp find_kernel32_finished
```

find\_kernel32\_9x:

```
    mov eax, [eax + 0x34]
    lea eax, [eax + 0x7c]
    mov eax, [eax + 0x3c]
```

find\_kernel32\_finished:

```
    pop esi
    ret
```

## Description du Code:

### **push esi**

Préserve le registre ESI

### **xor eax, eax**

Met EAX à zéro

### **mov eax, fs:[eax+0x30]**

Stocke l'adresse du PEB dans EAX

*(eax+0x30 élimine les zéros (nulls) et sauve un octet en même temps)*

### **test eax, eax**

Compare EAX avec lui même

### **js find\_kernel32\_9x**

Si SF vaut 1 alors c'est un Windows 9x

Sinon c'est un NT

*(La logique ici est que l'adresse où se charge kernel32.dll sur un Windows 9x est trop grande pour tenir dans un entier signé (plus grand que 0x7fffffff).)*

### **mov eax, [eax + 0x0c]**

Extrait le pointeur vers la structure de données du loader

### **mov esi, [eax + 0x1c]**

Extrait la première entrée dans la liste d'ordre d'initialisation des modules

### **lodsd**

Récupère la prochaine entrée dans la liste qui pointe sur kernel32.dll

### **mov eax, [eax + 0x8]**

Récupère l'adresse de base du module et la stocke dans EAX

### **jmp find\_kernel32\_finished**

Saute vers la fin comme kernel32.dll a été fait

### **mov eax, [eax + 0x34]**

Stocke le pointeur en offset 0x34 dans EAX (non documenté)

### **lea eax, [eax + 0x7c]**

Charge l'adresse effective dans EAX plus 0x7c pour rester dans la limite d'un octet signé pour éviter les zéros (nulls)

### **mov eax, [eax + 0x3c]**

Extrait l'adresse de base de kernel32.dll

### **pop esi**

Restaure ESI à sa valeur initiale.

### **ret**

Retourne à l'appelant.

# SEH

Cibles: 95/98/ME/NT/2K/XP

Taille: 33 octets

La technique du SEH (Structured Exception Handling – gestionnaire structuré d’exception) est la deuxième technique la plus fiable pour obtenir l’adresse de base de kernel32.dll.

Cette méthode est également mentionnée dans le document du The Last Stage of Delerium mais n’est pas couverte en détail. Le shellcode en lui-même est de 33 octets et fonctionne à la fois sur Windows 9x et Windows NT.

Le procédé pour déterminer l’adresse de base de kernel32.dll avec cette technique repose sur le fait que le SEH par défaut est configuré pour utiliser une fonction qui réside dans kernel32.dll. A la fois sur Windows 9x et Windows NT, la première entrée dans la liste SEH est toujours trouvée en fs:[0] dans le processus. L’on peut donc parcourir la liste des gestionnaires d’exception installés jusqu’à la dernière. Quand la dernière est atteinte, l’adresse du pointeur de fonction peut être utilisée comme point d’entrée pour revenir en arrière par pas de 64Ko, ou pages de 16 x 4096 octets.

Dans Windows, les DLLs vont s’aligner seulement par limites de 64Ko. A chaque limite de 64Ko, une vérification peut être réalisée pour voir si les 2 caractères sont ‘MZ’. Ces deux caractères marquent l’entête MSDOS des exécutables portables. Quand une correspondance est trouvée, il est fiable de dire que l’adresse de base de kernel32.dll a été trouvée.

Le problème que l’on peut rencontrer avec cette technique est que le UEH (Unhandled Exception Handler – gestionnaire d’exception non traitée) peut ne pas pointer dans kernel32.dll. Il est possible pour une application de retirer complètement le gestionnaire standard de l’image et d’utiliser le sien propre. Dans ce cas, cette méthode ne peut pas être utilisée. Mais cela n’arrive que rarement.

## Code assembleur:

```
find_kernel32:
    push esi
    push ecx
    xor ecx, ecx
    mov esi, fs:[ecx]
    not ecx
find_kernel32_seh_loop:
    lodsd
    mov esi, eax
    cmp [eax], ecx
    jne find_kernel32_seh_loop
find_kernel32_seh_loop_done:
    mov eax, [eax + 0x04]

find_kernel32_base:
find_kernel32_base_loop:
    dec eax
    xor ax, ax
    cmp word ptr [eax], 0x5a4d
    jne find_kernel32_base_loop

find_kernel32_base_finished:
    pop ecx
    pop esi
    ret
```

## Description du Code :

### **push esi**

Préserve le registre ESI

### **push ecx**

Préserve le registre ECX

### **xor ecx, ecx**

Met ECX à zéro, ainsi il peut être utilisé comme l'offset pour obtenir la première entrée dans la liste SEH

### **mov esi, fs:[ecx]**

Capture la première entrée dans la liste SEH et la stocke dans ESI

### **not ecx**

Inverse tous les bits dans ECX pour l'utiliser dans la comparaison plus tard pour déterminer si le dernier gestionnaire (handler) d'exception a été atteint

### **lodsd**

Charge l'entrée suivante dans la liste SEH et la stocke dans EAX

### **mov esi, eax**

Initialise ESI à la prochaine entrée dans la liste

### **cmp [eax], ecx**

Compare la valeur de EAX pour voir s'il est à 0xffffffff. Si oui, la dernière entrée de la liste est atteinte et le pointeur de fonction doit être dans kernel32.dll.

### **jne find\_kernel32\_seh\_loop**

Si les valeurs sont différentes, alors l'adresse de base n'a pas encore été trouvée. On boucle encore

### **mov eax, [eax + 0x04]**

Si l'entrée suivante dans la liste est égale à 0xffffffff, on sait que la fin est atteinte. On peut donc extraire le pointeur pour cette entrée et le stocker dans EAX.

### **dec eax**

Décrémente EAX. Si la valeur précédente était alignée sur une limite de 64Ko, cela va mettre les 16 bits inférieurs de EAX à 0xffff. Sinon, cela va juste décrémente EAX à une valeur indéterminée

### **xor ax, ax**

Met à zéro les 16 bits inférieurs d'EAX pour aligner l'adresse sur une limite de 64Ko

### **cmp word ptr [eax], 0x5a4d**

Vérifie si la valeur des 2 octets d'EAX est 'MZ'

### **jne find\_kernel32\_base\_loop**

Si les valeurs ne correspondent pas, boucle encore et va à la prochaine limite de 64Ko inférieure. Sinon, saute plus bas comme l'adresse de base de kernel32.dll a été trouvée.

### **pop ecx**

Restaure ECX à sa valeur initiale

### **pop esi**

Restaure ESI à sa valeur initiale

### **ret**

Retourne à l'appelant

# TOPSTACK

Cibles: NT/2K/XP

Taille: 25 octets

La dernière technique décrite dans ce document est relativement nouvelle.

Elle n'implique que 25 octets et ne fonctionne actuellement que pour les versions basées sur Windows NT.

Elle consiste à extraire le dessus du tas en utilisant un pointeur enregistré dans le TEB (Thread Environment Block). Chaque thread s'exécutant possède son propre TEB correspondant avec des informations uniques à ce thread. Le TEB pour le thread en cours peut être accédé en fs:[0x18] dans le processus. Le pointeur sur le dessus du tas pour le thread courant peut être trouvé 0x4 octets dans le TEB. A partir de là, 0x1c octets dans le tas depuis le dessus réside un pointeur qui existe quelque part dans kernel32.dll. Finalement, nous pouvons utiliser la même approche que la technique du SEH en parcourant les limites de 64Ko jusqu'à trouver un 'MZ'.

## Code assembleur:

```
find_kernel32:
    push esi
    xor esi, esi
    mov esi, fs:[esi + 0x18]
    lodsd
    lodsd
    mov eax, [eax 0x1c]

find_kernel32_base:
find_kernel32_base_loop:
    dec eax
    xor ax, ax
    cmp word ptr [eax], 0x5a4d
    jne find_kernel32_base_loop

find_kernel32_base_finished:
    pop esi
    ret
```

## Description du Code :

### **push esi**

Préserve le register ESI

### **xor esi, esi**

Met ESI à zéro pour l'utiliser comme base pour l'index dans le segment FS

### **mov esi, fs:[esi + 0x18]**

Capture le TEB et le stocke dans ESI

### **lods**

Ajoute 4 à ESI, la valeur actuelle n'a pas d'importance

### **lods**

Capture le haut du tas et le stocke dans EAX

### **mov eax, [eax 0x1c]**

Capture le pointeur qui est 0x1c octets dans la pile et le stocke dans EAX. Cela sera l'adresse qui est dans kernel32.dll

### **dec eax**

Décrémente EAX. Si la valeur précédente était alignée sur une limite de 64Ko, cela va mettre les 16 bits de poids faible d'EAX à 0xffff. Sinon, cela va juste décrémente EAX pour une valeur indéterminée

### **xor ax, ax**

Met à zéro les 16 bits de poids faible d'EAX pour aligner l'adresse sur une limite de 64Ko

### **cmp word ptr [eax], 0x5a4d**

Vérifie si la valeur des 2 octets de EAX est 'MZ'

### **jne find\_kernel32\_base\_loop**

Si les valeurs ne correspondent pas, boucle encore et va à la prochaine limite inférieure de 64Ko. Sinon, saute plus bas puisque l'adresse de base de kernel32.dll a été trouvée.

### **pop esi**

Restaure ESI à sa valeur initiale.

### **ret**

Retour à l'appelant