

# Wrapping & Speedhack

Mathieu SCHROETER

8 décembre 2008

révisé le 19 février 2010

## Résumé

Le but de cette article est de présenter le wrapping de fonctions avec GCC/LD sous Linux ainsi que le fonctionnement des Speedhacks. Les informations délivrées par ce document sont avant tout dans un but pédagogique. Les manipulations ne sont pas simplifiées pour le commun des mortel et demandent certaines connaissances en programmation.

## Première partie

# Le speedhack

## 1 Généralités

Les tricheurs (ou cheaters) sur les jeux vidéos en ligne le connaissent bien. Le speedhack permet de changer dynamiquement la vitesse d'un programme. Par exemple avec un jeu vidéo, il devient facile d'accélérer ses mouvements pour se déplacer plus vite, de recharger plus vite une arme, etc,.. Ce cheat<sup>1</sup> est (était ?) particulièrement utilisé avec des jeux tel que Counter-Strike<sup>2</sup> couplé entre autre à un cheat très connu à l'époque qui se nommait OGC<sup>3</sup>. Si on ne connaît pas très bien l'informatique, on peut légitimement se demander comment est-ce que c'est possible de changer la vitesse d'un programme. Il faut tout d'abord comprendre comment est-ce qu'on développe une application pour lui donner un rythme bien précis. En principe, il y a toujours une grande boucle principale quelque part qui va plus ou moins vite. Les très vieux jeu utilisaient la vitesse du processeur. C'est à dire que la boucle allait aussi vite que le processeur pouvait fonctionner. Avec un processeur plus rapide, le jeu devenait plus rapide. Sur un système multi-tâche c'est particulièrement problématique car le programme utilise le maximum de ressource possible. Sans compter que le changement de matériel change la vitesse et peut rendre un jeu complètement inutilisable. Ainsi il faut temporiser d'une manière ou d'une autre. Une temporisation va endormir le processus pendant un temps donné, ce qui permet de garder un rythme précis tout en libérant le processeur pour les autres applications.

Le speedhack m'avait longtemps intrigué alors j'avais décidé de comprendre son fonctionnement interne. Pour l'utiliser il fallait simplement exécuter le jeu, exécuter ensuite le speedhack, puis le jeu apparaissait dans une liste de processus. Il suffisait alors de le choisir et de lui appliquer le speedhack. Il était possible d'agir sur la vitesse en étant dans le jeu à l'aide de deux touches (ralentir/accélérer). Le fait d'augmenter la vitesse utilisait beaucoup plus de charge processeur ce qui peut sembler évident. Une vitesse trop élevée pouvait faire planter complètement le programme ou alors lui donner des comportements imprévisibles.

## 2 Fonctionnement

L'idée est assez simple. Pour varier la vitesse perçue par le jeu, il faut lui modifier sa perception du temps. Cela veut simplement dire qu'il faut lui faire croire qu'une seconde dure moins de temps que dans la réalité (par exemple 1 s représenterait 500 ms ce qui doublerait la vitesse), et inversement pour le ralentir. Pour ce faire, le speedhack va faire ce qu'on appelle du "wrapping"<sup>4</sup>. Le but est donc de réécrire une fonction de l'API<sup>5</sup> Windows et de s'interposer entre le jeu

<sup>1</sup>Moyen de tricher (en général dans un jeu vidéo).

<sup>2</sup>Jeux vidéo de tir subjectif à la première personne qui se joue en équipe sur internet.

<sup>3</sup>Abréviation de Online Game Cheat qui représentait une communauté de cheaters (tricheurs) et de logiciels de triche (parfois libres).

<sup>4</sup>Moyen de s'interposer entre deux systèmes afin de manipuler l'information de manière transparente (ou non).

<sup>5</sup>Abréviation de Application Programming Interface : <http://en.wikipedia.org/wiki/Api>

et cette fonction. Ainsi ce n'est plus la fonction de Windows qui est exécutée, mais celle du speedhack. Pour cela il faut injecter une *DLL*<sup>6</sup> dans le processus en cours qui contient la fonction "wrappée". Je ne vais pas vous expliquer comment on fait une injection de *DLL* dans cet article (un peu de recherche sur internet vous donne suffisamment d'exemples), mais je vais vous dire quelle fonction de l'API est "wrappée" par les speedhacks que l'on trouve sur la toile. C'est la fonction *QueryPerformanceCounter()* de *kernel32.dll*.

## 2.1 Mais encore...

Certain sont en train de se dire que ça leur fait une belle jambe. Je suis d'accord qu'il y a encore un pas non négligeable à franchir pour faire croire au programme, que la fonction qu'il doit appeler est celle du wrapper et non pas celle de Windows (ou plutôt celle de *kernel32.dll*). Les tricheurs sont des vrais plaies dans les jeux en lignes, ainsi les éditeurs se battent très fermement contre ces gens et donc leurs produits ont plusieurs anti-cheats intégrés plus ou moins sophistiqués. Attaquer un processus directement pour lui injecter notre wrapper se serait ouvrir la porte grande ouverte à l'anti-cheat (pour autant qu'il y en ait un, bien entendu).

Alors revenons à cette fonction *QueryPerformanceCounter()*, mais d'abord.. à quoi sert-elle? Elle sert à retourner une valeur relative au temps écoulé pour un clock de multiple de 1.193182 MHz (haute résolution). Et un autre point intéressant est de constater qu'en réalité la fonction de *kernel32.dll* fait appelle à une fonction du même type mais de plus bas niveau dans *ntdll.dll*. Le nom change légèrement en *NtQueryPerformanceCounter()*.

## 2.2 QueryPerformanceCounter

La fonction "wrappée" dans les speedhacks que j'ai rencontré n'est pas idéale. Et oui, il arrive souvent qu'un programme ne soit pas très affecté par le speedhack. Et comme rien n'est magique, j'ai donc cherché la vraie raison. Et celle-ci est simplement la conséquence de "wrapper" une fonction de trop haut niveau *QueryPerformanceCounter()* qui semble juste utiliser *NtQueryPerformanceCounter()*. Et certains programmes ne l'utilisent pas tout en restant basés sur un temps donné. Ce temps là est rendu par *NtQuerySystemTime()* qui est également situé dans *ntdll.dll*. Et dans le cas de Wine (j'y reviendrais plus tard), la fonction *NtQueryPerformanceCounter()* ne fait qu'utiliser cette seconde fonction en y convertissant la fréquence du clock. Passer uniquement par *QueryPerformanceCounter()* ou par *NtQueryPerformanceCounter()* peut être suffisant dans certains cas (ou avec Windows directement), voir même n'influencer que certaines parties dans d'autres cas et c'est donc beaucoup moins intéressant (ce que font les speedhacks que j'ai pu expérimenter). Il suffit donc de faire un hack<sup>7</sup> sur le *NtQuerySystemTime()* pour intégrer le speedhack de manière globale (au moins dans le cas de Wine).

Si vous n'avez pas tout saisi ce que je voulais dire dans le paragraphe précédent, j'y reviens plus en détails dans la suite du document.

## 2.3 Un hack

Comme je l'ai dis avant, je n'ai pas l'intention de vous expliquer comment injecter une *DLL* sous Windows. Premièrement je travail sous Linux, et deuxièmement il existe un fabuleux projet libre qui permet d'exécuter des applications Windows sous Linux. Je parle bien sûr du projet Wine<sup>8</sup>. Mais avant de s'intéresser à Wine, je vous propose un petit peu de théorie sur le "wrapping".

---

<sup>6</sup>Abréviation de Dynamic Link Library : [http://en.wikipedia.org/wiki/Dynamic-link\\_library](http://en.wikipedia.org/wiki/Dynamic-link_library)

<sup>7</sup>Un moyen rapide de contourner un problème, quel qu'il soit.

<sup>8</sup>Wine Is Not an Emulator : <http://www.winehq.org>

## Deuxième partie

# Wrapping

Les explications et les codes ci-dessous concernent uniquement Linux et GCC, il y a également des possibilités sous Windows mais je ne tiens pas à les aborder ici.

### 3 The GNU linker

Pour autant que l'on ne souhaite pas "wrapper" une fonction d'un exécutable déjà "relocaté"<sup>9</sup>, il est facile de le faire avec le *linker*. Pour l'exemple nous allons créer un programme très simple qui sera uniquement compilé en un fichier objet. Le but va être de "wrapper" la fonction *malloc()* et d'afficher un texte avant de réellement allouer la mémoire et cela sans modifier ce fichier source.

---

**Algorithm 1** test.c

---

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int
5  main (void)
6  {
7      char *c;
8
9      c = malloc (1);
10     free (c);
11
12     return 0;
13 }
```

---

Sauvez cette source dans un fichier nommé `test.c`. Vous pouvez compiler ce code de la manière suivante :

```
gcc -c test.c -o test.o
```

Ce programme va allouer un espace mémoire dynamique de 1 octet puis le libérer avant de se terminer. Pour notre exemple, on va tenter d'intercepter l'appel à la fonction *malloc()* à l'aide du *linker*. Pour se faire, il va nous falloir écrire un deuxième programme qui fera office de wrapper pour *malloc()*. Nous allons le sauver dans une source nommée `wrap_test.c`.

---

**Algorithm 2** wrap\_test.c

---

```
1  #include <stdio.h>
2
3  void *
4  __wrap_malloc (size_t c)
5  {
6     printf ("WRAPPER\n");
7     return __real_malloc (c);
8 }
```

---

Les noms utilisés sont importants car il vont permettre au *linker* de retrouver cette fonction afin de l'utiliser à la place du vrai *malloc()* au moment du "link". Il suffit maintenant de compiler ce wrapper de la même manière que pour le programme précédent :

```
gcc -c wrap_test.c -o wrap_test.o
```

On peut désormais linker et créer notre exécutable avec les commandes suivantes :

---

<sup>9</sup>Lier les symboles avec les adresses : [http://en.wikipedia.org/wiki/Relocation\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Relocation_(computer_science))

```
ld -r -x -wrap malloc wrap_test.o test.o -o foobar.o
gcc foobar.o -o foobar
```

On aurait pu le faire directement avec `gcc` sans utiliser `ld`, mais le but de l'exemple est de visualiser les étapes. L'argument `-wrap` du linker va automatiquement retrouver `__wrap_malloc()` et l'appeler à la place de `malloc()` en prenant soins de mettre le vrai `malloc` à la place de `__real_malloc()`.

Pour tester le résultat il suffit d'exécuter simplement le programme `foobar` de cette manière :

```
~$ ./foobar
WRAPPER
~$
```

### 3.1 Conclusion

L'intérêt ici est évident pour autant qu'on ait des objets non "relocatés" à disposition ou alors les sources. Il peut être très pratique de vouloir tracer certains appels de fonctions sans faire de modifications dans le programme original. Par contre, cette manière de faire ne pourra pas être possible avec un exécutable "relocaté" tel qu'un `ELF`<sup>10</sup> (`foobar` dans notre exemple).

## 4 Variable LD\_PRELOAD

Par le biais de `LD_PRELOAD`<sup>11</sup>, l'idée ici est de "wrapper" la fonction `free()` d'un exécutable `ELF` pour lequel nous considérerons que les sources sont indisponibles. Cet exemple est bien plus simple que le précédent car il va suffire d'écrire uniquement un fichier source avec la fonction à "wrapper". La seule contrainte est que le prototype de la fonction doit correspondre à celui de la fonction originale. Le code ci-dessous va réécrire la fonction `free()` en prenant soins de charger l'original afin que le "wrapping" n'influence pas directement le programme qui fera appel à `free()`.

---

#### Algorithm 3 wrap\_free.c

---

```
1  #include <stdio.h>
2  #include <dlfcn.h>
3
4  void
5  free (void *ptr)
6  {
7      static void *handle = NULL;
8      static void (*real_free) (void *) = NULL;
9
10     if (!handle)
11     {
12         handle = dlopen ("/lib/libc.so.6", RTLD_LAZY);
13         real_free = dlsym (handle, "free");
14     }
15
16     printf ("INTERCEPT FREE\n");
17     return real_free (ptr);
18 }
```

---

Sauvez ce code dans un fichier nommé `wrap_free.c`.

Comme vous pouvez le voir, notre fonction s'appelle exactement comme la vraie. Dans le cas contraire il ne serait pas possible de l'intercepter car les symboles doivent correspondre. Il est également nécessaire de faire appel à quelques fonctions de `dlfcn.h` afin de pouvoir retrouver le pointeur de fonction du vrai `free()`, pour ne pas influencer directement le comportement de l'exécutable qui servira de cobaye (dans ce cas, pour ne pas créer bêtement des fuites de mémoires).

Il faut maintenant compiler notre code en tant que bibliothèque partagée de cette manière :

---

<sup>10</sup>Un format d'exécutable : [http://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](http://en.wikipedia.org/wiki/Executable_and_Linkable_Format)

<sup>11</sup>Variable d'environnement : [http://en.wikipedia.org/wiki/LD\\_PRELOAD#Unix\\_and\\_Unix-likes](http://en.wikipedia.org/wiki/LD_PRELOAD#Unix_and_Unix-likes)

```
gcc -fPIC -c wrap_free.c -o wrap_free.o
gcc -shared -Wl,-soname,free.so wrap_free.o -o free.so -ldl
```

Nous avons désormais un fichier `free.so` qui va permettre d'intercepter n'importe quel `free()` de n'importe quel programme. Pour ce faire nous allons utiliser la variable d'environnement `LD_PRELOAD` et y indiquer notre fichier `so`. Le symbole de notre `free()` sera donc pris en premier lorsqu'un programme essaiera d'utiliser cette fonction.

Essayons d'intercepter le `free()` de notre programme précédant nommé `foobar`.

```
~$ LD_PRELOAD=./free.so ./foobar
WRAPPER
INTERCEPT FREE
~$
```

Vous voyez explicitement que `free()` a correctement été "wrappé", vous pouvez tester avec n'importe quel exécutable de votre système, tous les `free()` devraient retourner un "INTERCEPT FREE".

## 4.1 Conclusion

Cette méthode est plus puissante que la précédente car elle permet de s'abstraire des sources et même des `.o`. La seule limitation vient uniquement des fonctions que l'on peut intercepter. Elles doivent forcément être non statics tel que `free()` ou `malloc()` qui sont exportés par `libc.so.6`. En ce qui concerne Windows, il n'existe pas d'équivalent à `LD_PRELOAD` et il est alors nécessaire d'écrire un code quelque peu intrusif pour effectuer une injection de `DLL`.

## Troisième partie

# Wine et le speedhack

Le fait de chercher à comprendre le speedhack m'a rapidement amené à Wine. L'intérêt de ce projet est l'accès à son code-source et donc à tous ses mécanismes internes. Du fait que le code-source de Windows ne soit pas disponible pour le public, il suffit alors de regarder le code-source de Wine qui met au grands jours les secrets de *NtQuerySystemTime()* et de toutes les autres fonctions de l'API Windows qui ont été citées dans la première partie de ce document.

Les explications précédentes sur le "wrapping" ne sont pas toutes valables directement pour Windows c'est pour cela que la suite concerne uniquement Wine. Ce document aurait aussi pu concerner un speedhack pour Linux, mais ce n'était pas mon but. L'accès au code-source de Wine permet justement de s'abstenir d'écrire un wrapper, en modifiant directement le comportement en interne.

## 5 QueryPerformanceCounter

C'est la fonction la plus évidente car c'est elle qui est utilisée dans les speedhacks habituellement rencontrés sur internet. En fouillant dans le code-source de Wine il est facile de la retrouver dans le fichier `dlls/kernel/cpu.c` (qui correspond au `kernel32.dll`), elle se présente ainsi (wine-1.1.10) :

---

### Algorithm 4 `dlls/kernel/cpu.c`

---

```
299  /*****
300  *           QueryPerformanceCounter (KERNEL32.@)
301  *
302  * Get the current value of the performance counter.
303  *
304  * PARAMS
305  * counter [0] Destination for the current counter reading
306  *
307  * RETURNS
308  * Success: TRUE. counter contains the current reading
309  * Failure: FALSE.
310  *
311  * SEE ALSO
312  * See QueryPerformanceFrequency.
313  */
314  BOOL WINAPI QueryPerformanceCounter(PLARGE_INTEGER counter)
315  {
316     NtQueryPerformanceCounter( counter, NULL );
317     return TRUE;
318 }
```

---

Il est évident au premier coup d'oeil que cette fonction n'a presque aucun intérêt. Il serait possible de se baser uniquement sur celle-ci et de manipuler la valeur de *counter*, mais dans ce cas, tout ceux qui feraient appel à *NtQueryPerformanceCounter()* à la place de *QueryPerformanceCounter()* ne seraient nullement influencés. C'est la raison principale qui fait qu'avec certains speedhacks, seules certaines parties d'un programme sont "accélérées" et d'autres non.

## 6 NtQueryPerformanceCounter

Cette fonction se situe dans le fichier `dlls/ntdll/time.c` (qui correspond au `ntdll.dll`) et se présente ainsi (wine-1.1.10) :

---

**Algorithm 5** dlls/ntdll/time.c

---

```
458 /*****
459 * NtQueryPerformanceCounter [NTDLL.@]
460 *
461 * Note: Windows uses a timer clocked at a multiple of 1193182 Hz. There is a
462 * good number of applications that crash when the returned frequency is either
463 * lower or higher than what Windows gives. Also too high counter values are
464 * reported to give problems.
465 */
466 NTSTATUS WINAPI NtQueryPerformanceCounter( PLARGE_INTEGER Counter, PLARGE_INTEGER
    Frequency )
467 {
468     LARGE_INTEGER now;
469
470     if (!Counter) return STATUS_ACCESS_VIOLATION;
471
472     /* convert a counter that increments at a rate of 10 MHz
473      * to one of 1.193182 MHz, with some care for arithmetic
474      * overflow and good accuracy (21/176 = 0.11931818) */
475     NtQuerySystemTime( &now );
476     Counter->QuadPart = ((now.QuadPart - server_start_time) * 21) / 176;
477     if (Frequency) Frequency->QuadPart = 1193182;
478     return STATUS_SUCCESS;
479 }
```

---

Cette fonction est nettement plus intéressante car elle manipule le temps. J'ai donc tenté d'y intégrer un speedhack à ce niveau, mais les résultats étaient mitigés. Certains programmes n'étaient pas du tout influencés et d'autres que partiellement. Le résultat n'était donc pas satisfaisant, je vous épargne mes tentatives de patches et je me suis alors intéressé à la fonction *NtQuerySystemTime()* qui se trouve être dans le même fichier source.

## 7 NtQuerySystemTime

Cette fonction est appelée à différents endroits et peut donc expliquer les résultats peu satisfaisants avec la fonction précédente. Elle se présente donc ainsi (wine-1.1.10) :

---

**Algorithm 6** dlls/ntdll/time.c

---

```
435 /*****
436 * NtQuerySystemTime [NTDLL.@]
437 * ZwQuerySystemTime [NTDLL.@]
438 *
439 * Get the current system time.
440 *
441 * PARAMS
442 * Time [0] Destination for the current system time.
443 *
444 * RETURNS
445 * Success: STATUS_SUCCESS.
446 * Failure: An NTSTATUS error code indicating the problem.
447 */
448 NTSTATUS WINAPI NtQuerySystemTime( PLARGE_INTEGER Time )
449 {
450     struct timeval now;
451
452     gettimeofday( &now, 0 );
453     Time->QuadPart = now.tv_sec * (ULONGLONG)TICKSPERSEC + TICKS_1601_TO_1970;
454     Time->QuadPart += now.tv_usec * 10;
455     return STATUS_SUCCESS;
456 }
```

---

*gettimeofday()* est une fonction qui appartient à `sys/time.h` et est donc le niveau le plus bas possible que l'on peut atteindre à travers Wine. Il existe alors plusieurs possibilités pour tricher sur le temps. On pourrait imaginer par exemple écrire un wrapper par dessus *gettimeofday()* et exécuter Wine à travers celui-ci (en appliquant la théorie vu dans la partie sur le "wrapping"), ou alors écrire un hack directement dans cette fonction. Par souci de simplicité, je me suis contenté de faire les modifications directement à ce niveau et sans possibilité de modifier la vitesse depuis l'extérieur. Cela demande donc de recompiler Wine à chaque fois que l'on veut adapter la vitesse, mais le gros du travail est fait. Donner la possibilité d'influencer les valeurs depuis l'extérieur est selon moi uniquement du bonus (au moins par rapport à ce document).

## 8 Le Wine'speedhack

Le patch ci-dessous devrait s'appliquer sur n'importe quelle version de Wine. Il se peut qu'il y ait un peu d'offset par rapport à wine-1.1.10 car je l'ai écrit il y a déjà passablement de temps. Si pour différentes raisons vous n'arrivez pas à l'appliquer avec la commande *patch*, je vous invite alors à refaire les modifications à la main dans `dlls/ntdll/time.c`.

**L'utilisation de ce hack est entièrement à vos risques et périls.  
(Notez également qu'il n'est pas MT-Safe !)**

---

### Algorithm 7 speedhack.patch

---

```
diff --git a/dlls/ntdll/time.c b/dlls/ntdll/time.c
index e6e4ed2..a975b9a 100644
--- a/dlls/ntdll/time.c
+++ b/dlls/ntdll/time.c
@@ -448,10 +448,32 @@
 NTSTATUS WINAPI NtQuerySystemTime( PLARGE_INTEGER Time )
 {
     struct timeval now;
+    static ULONGLONG last_real = 0;
+    static ULONGLONG last_fake = 0;
+    ULONGLONG diff_real, new_real;
+    ULONGLONG diff_fake;
+    double factor = 2.0; /* factor on the speed of the time which is elapsing */

    gettimeofday( &now, 0 );
    Time->QuadPart = now.tv_sec * (ULONGLONG)TICKSPERSEC + TICKS_1601_TO_1970;
    Time->QuadPart += now.tv_usec * 10;
+
+    new_real = Time->QuadPart;
+
+    if (!last_real)
+    {
+        last_real = last_fake = new_real;
+        return STATUS_SUCCESS;
+    }
+
+    diff_real = (ULONGLONG)(new_real - last_real);
+    diff_fake = (ULONGLONG)(factor * (double)diff_real);
+
+    Time->QuadPart = last_fake + diff_fake;
+
+    last_fake += diff_fake;
+    last_real += diff_real;
+
     return STATUS_SUCCESS;
 }
```

---

Je me suis basé sur un code partiel relatif à un speedhack pour Windows qui "wrappait" la fonction *QueryPerformanceCounter()* et qui n'était alors pas toujours efficace. J'ai fait les adaptations nécessaires pour que le calcul se fasse

par rapport à `Time->QuadPart` et non à `Time->LowPart` comme il semble que se soit le cas avec de nombreux speedhack pour Windows. Si vous êtes intéressé, vous trouverez de la documentation sur cette structure directement dans le MSDN de Microsoft. Pour varier la vitesse de perception du temps, il suffit de modifier la variable *factor* avec une valeur supérieure à 1.0 pour augmenter la vitesse et inférieure à 1.0 pour la diminuer. Veillez à ne pas mettre n'importe quoi, au risque de rendre les applications exécutées par Wine complètement incontrôlables et imprévisibles. La logique ci-dessus est relativement simple, notez bien que *last\_real* et *last\_fake* sont `statics`, dans le cas contraire ça fonctionnerait nettement moins bien. Je vous épargne alors les explications sur ce code, je suppose que vous maîtrisez le C.

Les variables étant déclarées en `ULONGLONG` il y a peu de chance d'y avoir un dépassement à moins d'exagérer la valeur de *factor*. Je n'ai pas testé de cas de dépassement et je ne vous conseil pas de le faire.

## 9 Conclusion

Je tiens tout d'abord à rappeler quelques règles de bonne conduite. Il est interdit d'utiliser un speedhack en ligne avant tout pour une question de respect envers les autres joueurs et également conformément à de nombreuses licences d'utilisation des jeux.

Ceci étant dit, j'ai effectué quelques tests sur différents jeux en local et ce hack fonctionne parfaitement. Mes essais se sont portés sur une partie solo de `DIABLO II : LORD OF DESTRUCTION`, une partie solo de `HELLBENDER` ainsi qu'une partie sur `TOMB RAIDER : ANNIVERSARY`. Bien avant j'avais effectués des tests en modifiant uniquement la fonction `NtQueryPerformanceCounter()` et les résultats étaient mitigés. Dans le cas de `DIABLO II`, énormément d'éléments n'étaient pas du tout accélérés ou partiellement. Dans le cas du patch précédent (donc sur `NtQuerySystemTime()`), la variation de vitesse semblait uniforme pour tous les éléments avec tous les jeux. A noter que dans le cas de Windows, le comportement est très probablement différent car `NtQueryPerformanceCounter()` n'est pas sensé dépendre de `NtQuerySystemTime()`.