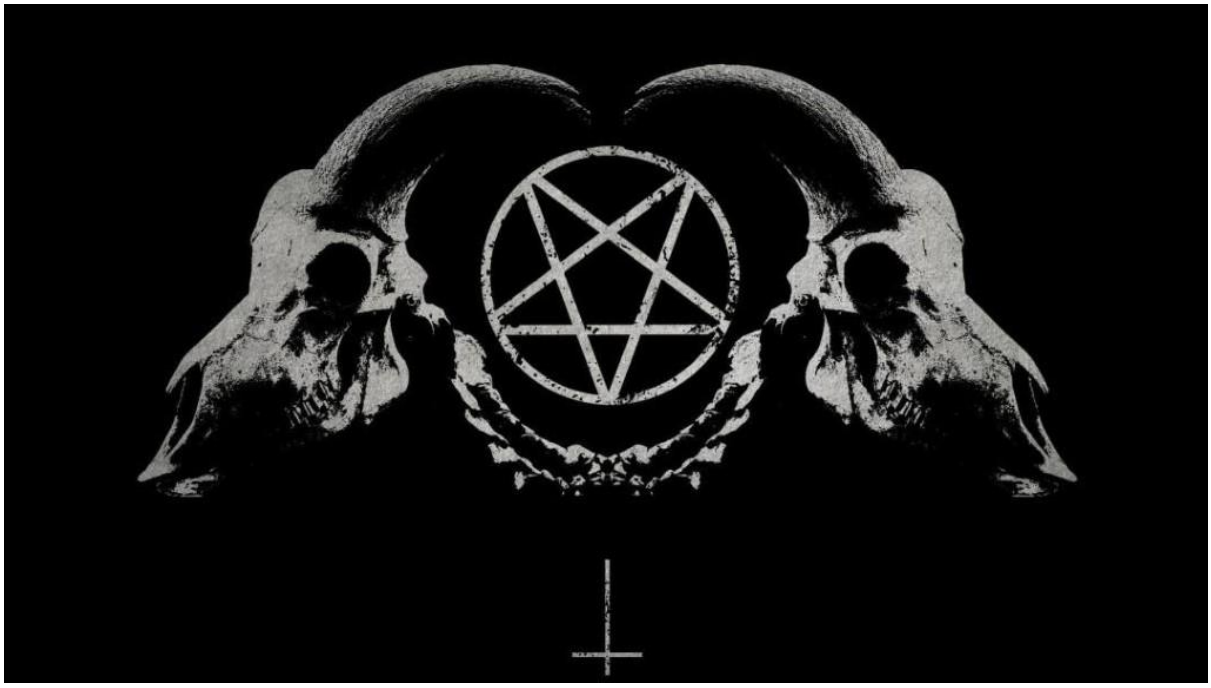
Abuser la Gestion de L’Alimentation de Windows

collection vx-underground // smelly_vx && Paul Laîné (am0nsec)



Introduction par smelly_vx

En Février, am0n et moi-même avons eus une discussion sur Matrix au sujet de la Gestion de l'Alimentation de Windows. Je ne me souviens pas vraiment de comment tout cela a commencé, néanmoins, cela a conduit à l'exploration des interfaces de programmation d'applications (ou API) offertes et documentées par Microsoft. Au cours de nos recherches, nous avons découvert des APIs qui permettaient d'exécuter des applications et du code pendant que le système est en veille. Ce fut vraiment drôle. Peu après, nous avons décidé de nous renseigner sur l'exécution d'applications pendant la veille du système, mais nous n'avons trouvé que très peu d'information. Comme la tradition le veut, nous avons trouvé une publication sur StackOverflow qui affirme que cela est impossible.

La liste ne s'arrête pas là, il y a d'autres publications avec des affirmations incorrectes :

- sleep mode- are programs still running?
- Can a Windows PC do anything while it is sleeping?
- How to keep a c++ code running when the pc is in sleep mode?
- When your computer is in sleep mode, can you make it so python runs?

Ce qui est intéressant c'est que quelqu'un avait pourtant avancé l'idée sur StackOverflow. Malheureusement, la communauté a préféré se plaindre de son idée.

En tous cas, et comme mon article précédent *Weaponizing Windows Virtualisation* le démontre, je ne suis pas là pour discriminer les réponses sur StackOverflow. Je préfère plutôt démontrer que parfois, les idées que vous avez ou que d'autres ont et qui sont labélisé comme impossible, sont en réalité possibles.

tl;dr nous pouvons exécuter des instructions quand le système est en mode veille léger (c.-à-d. light-sleep S1) et nous pouvons empêcher le système de transitionner dans un mode de veille plus profond en abusant la fonction SeThreadExecutionState.

Les notions suivantes vont-être abordé dans ce document :

L'objectif principal de ce document est d'expliquer comment il est possible de maintenir une application *consciente* malgré le fait que le système soit dans état de veille. De plus, nous aborderons brièvement quelques composants internes des machinismes de gestion de l'alimentation de Windows.

Les notions suivantes ne vont pas être abordées dans ce document :

Nous n'avons pas l'intention d'explorer en profondeur la gestion de l'alimentation de Windows au niveau *firmware* (ACPI) ou des pilotes du noyau Windows qui sont en relation avec ces ensembles d'API.

Exigences

Le code proposé dans ce document a été écrit en C et utilise l'API Windows (c.-à-d. WINAPI). Si vous n'êtes pas familier avec le C ou WINAPI, cet article sera peut-être complexe à comprendre. Néanmoins, si vous persistez, cela ne devrait pas être trop difficile.

L'API de la Gestion de l'Alimentation de Windows

Cette API est assez robuste et contient pas mal de fonctionnalités, interfaces, routines de rappel et structures. Beaucoup de chose peuvent être discutées à propos de cette API, étant donné qu'elle permet à un développeur d'interagir avec le pilot de noyau ACPI tout en restant en mode utilisateur (c.-à-d. user mode).

En résumé : « The Windows operating system provides a comprehensive and system-wide set of power management features. This enables systems to extend battery life and save energy, reduce heat and noise, and help ensure data reliability. The power management functions and messages retrieve the system power status, notify applications of power management events, and notify the system of each application's power requirements. »

Cela peut se traduire de la façon suivante : « le système d'exploitation Windows offre un ensemble de fonctionnalités pour la gestion de l'énergie du système. Ces fonctionnalités se veulent compréhensibles et accessibles par l'ensemble du système afin de prolonger la durée de vie de la batterie, économiser l'énergie, réduire la température et le bruit et de s'assurer de la fiabilité des données proposé. Les fonctions de gestion de l'énergie et les messages permettent de récupérer l'état d'alimentation du système, notifier les applications des événements lié à la gestion de l'alimentation et informer le système des besoins d'alimentation de chaque application. »

En ce qui concerne le mode veille du système, ce n'est que la pointe de l'iceberg. Il y a vraiment beaucoup d'information concernant les différents niveaux de veille du système, hibernation, pilotes ACPI du noyau, etc … Comme expliqué précédemment nous n'allons pas aborder ces sujets. Il existe une norme pour les différents états de veille, et c'est sur cela que nous allons nous concentrer. [1] [2] [3] [4] [5]

Microsoft a documenté les différents états de veille sur MSDN. Ils illustrent cela avec leur structure SYSTEM_POWER_STATE.

```
typedef enum _SYSTEM_POWER_STATE {
  PowerSystemUnspecified,
  PowerSystemWorking,
  PowerSystemSleeping1,
  PowerSystemSleeping2,
  PowerSystemSleeping3,
  PowerSystemHibernate,
  PowerSystemShutdown,
  PowerSystemMaximum
} SYSTEM_POWER_STATE, *PSYSTEM_POWER_STATE;
```

Dans notre cas, les deux états qui nous intéressent sont : *modern standby* (ACPI S0) and *sleep* (ACPI S1).

Gestion des Modifications des Etats d'Alimentation

En fonction de l'état de l'alimentation dans lequel le système entrera, Windows suspendra les différents *threads* en cours d'exécution et enregistrera la mémoire volatile sur le disque. Ceci est important parce que si les différents processus et les pilotes du noyau ne sont pas avertis, il y a beaucoup de chance pour que des évènements incontrôlés et incontrôlable surviennent, ce qui va nécessairement amener au crash des application, ou pire du système.

Windows offre plusieurs solutions pour palier à ce problème. Premièrement, le module User32 expose une fonction appelée RegisterPowerSettingNotification, qui nécessite un destinataire. Le destinataire est typiquement une routine de rappel créée avec une fonction comme RegisterClassEx en conjonction avec CreateWindowEx.

Alternativement, il est possible d'utiliser une fonction vraiment similaire exposée par le module Powrprof, appelé PowerSettingRegisterNotification. Dans ce cas, au lieu d'invoquer les fonctions *RegisterClassEx* et *CreateWindowEx*, la routine de rappel est définie par le troisième paramètre (c.-à-d. *Recipient*) de la fonction. Ce paramètre est un pointeur vers une structure appelée DEVICE_NOTIFY_SUBSCRIBE_PARAMETERS.

Dans les deux cas, les routines de rappel vont permettre de recevoir un message quand le système d'exploitation va s'apprêter à entrer en état de veille. Dans le cas où le module User32 est utilisé, le nom de l'évènement va être WM_POWERBROADCAST. Sinon, cela va être PBT_POWERSETTINGCHANGE.

Il est important de noter que RegisterPowerSettingNotification n'est autre qu'une abstraction de la fonction PowerSettingRegisterNotification. *In fine* les deux fonctions accomplissent le même objective.

Finalement, pour conclure cette section, Windows offre depuis longtemps aux développeurs un ensemble de fonctions et structures permettant de détecter et d'empêcher le système d'entrer en état de veille. Pour être plus précis, il est possible d'abuser de la fonction SetThreadExecutionState afin de mettre le système en état de veille légère (c.-à-d. light-sleep) et d'empêcher un état de veille plus profond, décrit comme état S2 et S3. Ce qui peut donc être utilisé comme technique d'évasion de *sandbox* et de *debugger* et pour conduire des opérations que l'utilisateur ne pourra pas voir.

Le Code

Le code ci-dessous démontre comment il est possible de s'assurer que l'exécution de code n'est possible uniquement quand le système est en état de veille S1 et comment il est possible d'empêcher la transition dans des états de veille plus profonds. Le code utilise les en-têtes *powrprof* et *powersettings*. Toutefois, pour empêcher les potentiels problèmes de compilation, le code importe dynamiquement les fonctions exportées par le module *powrprof.dll*.

Voici un rapide aperçu du code :

1. Définition des deux fonctions qui vont être utilisées.

```
typedef DWORD(WINAPI* POWERSETTINGREGISTERNOTIFICATION)(LPCGUID, DWORD, HANDLE, PHPOWERNOTIFY);
typedef DWORD(WINAPI* POWERSETTINGUNREGISTERNOTIFICATION)(HPOWERNOTIFY)
```

2. Chargement dynamique des fonctions.

```
HMODULE hLibrary;
POWERSETTINGREGISTERNOTIFICATION _PowerSettingRegisterNotification = NULL;
POWERSETTINGUNREGISTERNOTIFICATION _PowerSettingUnregisterNotification = NULL;

hLibrary = LoadLibrary(L"powrprof.dll");
if (hLibrary == NULL)
    goto FAILURE;

_PowerSettingRegisterNotification = (POWERSETTINGREGISTERNOTIFICATION)GetProcAddress(hLibrary,
"PowerSettingRegisterNotification");
 _PowerSettingUnregisterNotification = (POWERSETTINGUNREGISTERNOTIFICATION)GetProcAddress(hLibrary,
"PowerSettingUnregisterNotification");
if (!_PowerSettingRegisterNotification || !_PowerSettingUnregisterNotification)
    goto FAILURE;
```

3. Initialisation de la structure DEVICE_NOTIFY_SUBSCRIBE_PARAMETERS avec une routine de rappel pour intercepter la transition en état de veille du système.

```
ULONG CALLBACK HandlePowerNotifications(PVOID Context, ULONG Type, PVOID Setting);

DEVICE_NOTIFY_SUBSCRIBE_PARAMETERS NotificationsParameters;
NotificationsParameters.Callback = HandlePowerNotifications;
NotificationsParameters.Context = NULL;
```

4. Création d'une routine de rappel pour intercepter la transition d'état de veille du système.

Dans notre cas le pointeur vers le contenu du message intercepté sera de type PPOWERBROADCAST_SETTING. De plus, si le type de message est PBT_POWERSETTINGCHANGE et que le membre *PowerSetting* de la structure est égal à GUID_CONSOLE_DISPLAY_STATE nous évaluons le membre *Data* afin de déterminer dans quel état de veille le système va transitionner.

Le membre *Data* doit avoir une de ces valeurs :

- 0x0 – Affichage suspendu
- 0x1 – Affichage allumé
- 0x2 – Affichage grisé

Finalement, dans notre exemple, si le membre *Data* est égal à 0x0 ou 0x2 nous faisons une pause de 10 secondes. Notez bien que toutes les applications ont un maximum de 2 seconds pour agir quand la notification de transition d'état de veille est envoyée par le système.

```
ULONG CALLBACK HandlePowerNotifications(PVOID Context, ULONG Type, PVOID Setting) {
PPOWERBROADCAST_SETTING PowerSettings = (PPOWERBROADCAST_SETTING)Setting;

if (Type == PBT_POWERSETTINGCHANGE && PowerSettings->PowerSetting == GUID_CONSOLE_DISPLAY_STATE) {
    switch (*PowerSettings->Data) {
        case 0:
        case 2: {
            Sleep(10000);
            MessageBoxW(NULL, L"Spooky Payload", L"", MB_OK);
            break;
        }
        case 1: {
            Sleep(1);
            break;
        }
        default: {
            break;
        }
    }
}

return ERROR_SUCCESS;
}
```

5. Enregistrement de notre routine de rappel et on spécifie que nous voulons recevoir les messages de type GUID_CONSOLE_DISPLAY_STATE.

```
if (_PowerSettingRegisterNotification(&GUID_CONSOLE_DISPLAY_STATE, DEVICE_NOTIFY_CALLBACK,
(HANDLE)&NotificationsParameters, &hNotificationRegister) != ERROR_SUCCESS) {
    goto FAILURE;
}
```

6. Modification de l'état d'exécution de notre *thread* afin de pouvoir continuer à exécuter notre code.

```
if (SetThreadExecutionState(ES_AWAYMODE_REQUIRED | ES_CONTINUOUS | ES_SYSTEM_REQUIRED) == NULL)
    goto FAILURE;
```

7. Attend indéfiniment le message de notification. Si le programme quitte la boucle infinie, nous dé-enregistrons les notifications.

```
while (1){ Sleep(100); }

if (hNotificationRegister)
    _PowerSettingUnregisterNotification(hNotificationRegister);
return ERROR_SUCCESS;
```

8. Finalement, voici la routine qui sera exécutée en cas d'erreur.

```
FAILURE:

    dwError = GetLastError();

    if (hNotificationRegister)
        _PowerSettingUnregisterNotification(hNotificationRegister);
    return dwError;
```