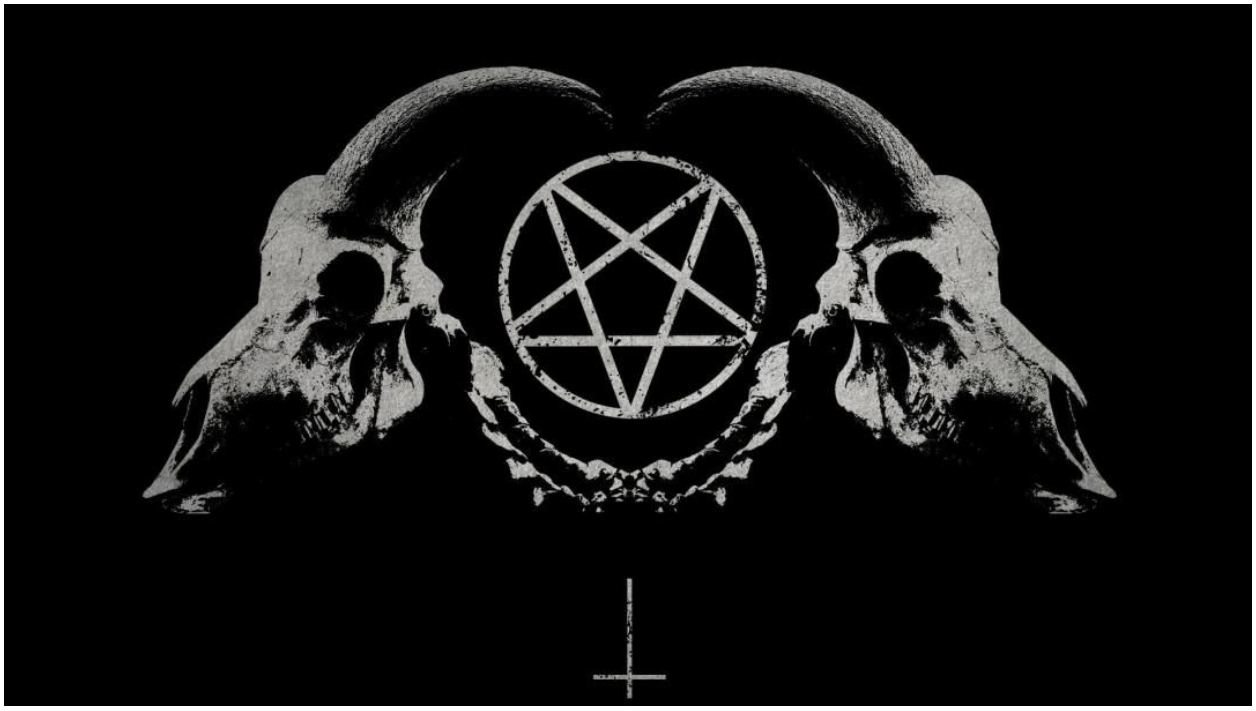
Abusing the Windows Power Management API

vx-underground collection // smelly__vx && am0nsec



Introduction by smelly__vx

Sometime in February am0n and I were discussing the Windows Power Management API on Matrix. I cannot recall the exact details on how the conversation started. Nevertheless this resulted in us both exploring the Windows Power Management API set and what it had in store for us. Through this we discovered some APIs which *allowed us to run an executable while the machine was asleep*. This was fun. We decided to do some internet detective research and we found very little regarding binaries running while the machine is asleep. As is tradition, we found a StackOverflow question and answer saying this is impossible.

And other questions and answers which were also wrong..
Microsoft Question: Sleep mode - are programs still running?
Superuser.com: Can a Windows PC do anything while it is sleeping?
Stackoverflow: How to keep a C++ code running when the PC is in sleep mode?
Stackoverflow: When your computer is in sleep mode, can you make it so python runs?

Interestingly, one guy got it right but StackOverflow complained to him.

Anyway, as per my previous paper *Weaponizing Windows Virtualisation*, I am not here to discriminate against StackOverflow answers, rather I am doing this to point out that sometimes ideas you have, which others may label as impossible, are actually indeed possible. Don't give up, kids.

tl;dr we can still execute instructions in light-sleep, S1, and prevent the system from transitioning into deeper stages of sleep by abusing SetThreadExecutionState

What this paper will discuss:
This paper will show how to ensure an application remains *conscious* despite the machine being asleep. Additionally, we will briefly review some internal components regarding Windows sleeping mechanisms.

What this paper will NOT discuss:
We do not intend for this paper to dive deep into the Windows Power Management API at a firmware level (ACPI) - or any other kernel mode driver which may have a relationship to this API set.

Requirements:
The code used in this paper will be using C WINAPI. If you're unfamiliar with C or the WINAPI this paper may be hard to follow. However, if you're persistent it shouldn't be too bad.

The Windows Power Management API

This API set is fairly robust and contains quite a bit of functionality, interfaces, callback routines and structures. A lot can be said about this API, as it allows developers to work from user-mode while interoping with ACPI drivers. As a tl;dr

Quoting MSDN: "*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.*"

That's a pretty good summary. In regards to the machine sleeping though this is just the tip of the iceberg. There is a lot of information regarding machine sleep states, hibernation, ACPI drivers, so on and so forth. We're going to skip all of that. There is a standard in which sleep states operate and that is where we're going to focus. [1] [2] [3] [4] [5]

Following the standard of other OS's Microsoft has documented the differences between its various sleep states on MSDN. They illustrate this further with their SYSTEM_POWER_STATE enum.

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

In our case, the two power states that we will focus on are: sleep modern standby (ACPI S0) and sleep (ACPI S1).

Power States Modification Handling

Depending on the power state the system will enter on, Windows will suspend the threads of running processes and save volatile memory onto the disk. If running applications and drivers are not alerted of the power state transition there is a high probability of uncontrolled and unintended behaviours which might lead to the crash of applications, or worse, the system.

Windows offers different solutions to this issue. The first being a User32 implementation using RegisterPowerSettingNotification, which requires a recipient, typically in the form of a user created callback routine via RegisterClassEx in conjunction with CreateWindowEx.

Alternatively, you can use the eerily similar Powrprof implementation PowerSettingRegisterNotification which does not require an invocation of RegisterClassEx and CreateWindowEx. Instead, you must define your callback routine via the 3rd parameter *Recipient* which will be a pointer to a DEVICE_NOTIFY_SUBSCRIBE_PARAMETERS structure.

Either way, both callback routines will receive a message event from the OS when the OS power state has changed. In the event if you use the User32 RegisterPowerSettingNotification the event will be WM_POWERBROADCAST. If you go the PowerSettingRegisterNotification route the event will be PBT_POWERSETTINGCHANGE.

It should probably be noted that the User32 function RegisterPowerSettingNotification is actually an API forward to PowerSettingRegisterNotification. Both achieve the same result.

Finally, to conclude this segment, for quite some time Windows has provided options to developers to not only detect the machine entering sleep, but also *prevent it from entering sleep*. To elaborate on this further: It is possible to abuse the function SetThreadExecutionState to allow the machine to enter light-sleep but not the sleep state traditionally described such as the processor core and bus stopping and system contexts lost. This can be used for sandbox evasion, debugger evasion, and performing operations users would be unable to see.

The code

This proof-of-concept demonstrates how to ensure your executable remains active while in sleep state S1 and prevent the machine from transitioning into deeper states of sleep (S2 or greater). This code also uses the *powrprof* and *powersetting* headers. However, to avoid making compilation a hassle the code dynamically imports the functionality from powrprof.dll. Here is a quick high-level overview of the code:

1. Define both the powrprof functions we will need to dynamically import. In this case we're importing PowerSettingRegisterNotification and its sister function PowerSettingUnregisterNotification

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

2. Attempt to dynamically load both functions. If we are unable to, go to our failure routine

```
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. Initialise a DEVICE_NOTIFY_SUBSCRIBE_PARAMETERS with an established callback routine to handle system power change notifications. Our callback routine is a DEVICE_NOTIFY_CALLBACK_ROUTINE callback function.

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

```
DEVICE_NOTIFY_SUBSCRIBE_PARAMETERS NotificationsParameters;

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

(Continued below)

4. Create a callback routine to handle incoming power change notifications. Our callback will typecast the incoming system message to type PPOWERBROADCAST_SETTING. Additionally, if the message is of *Type* PBT_POWERSETTINGCHANGE and our typecasted PPOWERBROADCAST_SETTING member *PowerSetting* is equal to GUID_CONSOLE_DISPLAY_STATE then we further evaluate the PPOWERBROADCAST_SETTING member *Data* to determine which S-level we're transitioning toward. When the PowerSetting member is of type GUID_CONSOLE_DISPLAY_STATE our Data member will be one of the follow:

1. 0x0 - Display is off
2. 0x1 - Display is on
3. 0x2 - Display is dimmed

Finally, in the event our Data member is 0 or 2 we wait 10 seconds for padding. Each application is given 2 seconds to take appropriate action on power setting notifications.

Note: per MSDN spec Windows 8 and higher should use GUID_CONSOLE_DISPLAY_STATE not MONITOR_DISPLAY_STATE

```
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. Register our callback routine with the OS. Specify we're wanting to receive messages of type GUID_CONSOLE_DISPLAY_STATE per bulletin 4

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

6. Specify our applications thread execution state so it remains active despite power setting changes

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

7. Infinitely loop to receive system notifications. If in the event our application escapes the loop unregister our application from power setting notifications.

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

if (hNotificationRegister)
         _PowerSettingUnregisterNotification(hNotificationRegister);

return ERROR_SUCCESS;
```

8. Our failure routine

```
FAILURE:

       dwError = GetLastError();

       if (hNotificationRegister)
              _PowerSettingUnregisterNotification(hNotificationRegister);

       return dwError;
```