NtSetDebugFilterState as Anti-Dbg Trick Reverse Engineering

Author: Giuseppe 'Evilcry' Bonfa'
E-mail: evilcry {AT} gmail {DOT} com
Website: http://evilcry.netsons.org ~
http://evilcodecave.wordpress.com
http://evilfingers.com

The Essay

The following paper will uncover some intersting Undocumented functions relative to Windows Debugging Support. NT is capable of generating and collecting text Debug Messages with an high grade of customization. User-mode and kernel-mode drivers use different routines to send output to the debugger.

User Mode: Uses **OutputDebugString,** that sends a null-terminated string to the debugger of the calling process. In a user-mode driver, **OutputDebugString** displays the string in the Debugger Command window. If a debugger is not running, this routine has no effect. **OutputDebugString** does not support the variable arguments of a **printf** formatted string.

Kernel Mode: Uses **DbgPrint,** that displays output in the debugger window. This routine supports the basic **printf** format parameters. Only kernel-mode drivers can call **DbgPrint.** There is also DbgPrintEx that is similar to **DbgPrint,** but it allows you to "tag" your messages. When running the debugger, *you can permit only those messages with certain tags to be sent*. This allows you to view only those messages that you are interested in.

This operation is called **Filtering Debug Messages,** how it works is a little bit undocumented, to understand how to go inside this aspect, let's start from DbgPrint / DbgPrintEx.

In Windows XP, DbgPrint has been extended by adding **_vDbgPrintExWithPrefix, in this way** DbgPrint and DbgPrintEx became wrappers of this function.

```
ULONG
  vDbgPrintExWithPrefix   (
     IN PCCH    Prefix,
     IN ULONG     ComponentId,
     IN ULONG     Level,
     IN PCCH      Format,
     IN va_list   arglist
     );
```

vDbgPrintExWithPrefix routine sends a string to the kernel debugger if certain conditions are met. This routine can append a prefix to debugger output to help organize debugging results.

Let's see what **ComponentId** means:

The component that is calling this routine. This parameter must be one of the component name filter IDs that are defined in *Dpfilter.h.* Each component is referred to in different ways, depending on the context. In the *ComponentId* parameter of **DbgPrintEx,** the component name is prefixed with **DPFLTR_** and suffixed with **_ID.** In the registry, the component filter mask has the same name

as the component itself. In the debugger, the component filter mask is prefixed with **Kd_** and suffixed with **_Mask.**

Now let's see **Level** parameter:

The severity of the message that is being sent. This parameter can be any 32-bit integer. Values between 0 and 31 (inclusive)) are treated differently than values between 32 and 0xFFFFFFFF.
Filter masks that are created by the debugger take effect immediately and persist until Windows is restarted.

The debugger can override a value that is set in the registry, but the component filter mask returns to the value that is specified in the registry if the computer is restarted. There is also a system-wide mask called WIN2000. By default, this mask is equal to 0x1, but you can change it through the registry or the debugger like all other components. **When filtering is performed, each component filter mask is first combined with the WIN2000 mask by using a bitwise OR.** In particular, this combination means that components whose masks have never been specified default to 0x1.

By inspecting deeply **vDbgPrintExWithPrefix** we can see that it represent a wrap around **NtQueryDebugFilterState** that retrieves the state of the selected Debug Filter Mask. By inspecting xRefs we discover that NtQueryDebugFilterState is also used by **DbgQueryDebugFilterState()**

```
NTSTATUS __stdcall DbgQueryDebugFilterState(ULONG ComponentId, ULONG
Level)
0045000C    ComponentId     = dword ptr  8
0045000C    Level           = dword ptr  0Ch
0045000C
0045000C     mov     edi, edi
0045000E     push    ebp
0045000F     mov     ebp, esp
00450011     pop     ebp
00450012     jmp     NtQueryDebugFilterState
00450012 _DbgQueryDebugFilterState end proc
```

As is obvious, DbgQueryDebugFilterState asks for the actual state of Debug Filters. Near the Query function we can see **DbgSetFilterState()**

```
NTSTATUS __stdcall DbgSetDebugFilterState(ULONG ComponentId, ULONG Level,
BOOLEAN State)
0045001C     mov     edi, edi
0045001E     push    ebp
0045001F     mov     ebp, esp
00450021     pop     ebp
00450022     jmp     NtSetDebugFilterState
00450022     DbgSetDebugFilterState endp
```

DbgSetDebugFilterState is a wrapper of a native **NtSetDebugFilterState(ULONG ComponentId, ULONG Level, BOOLEAN State)**

As you can understand this is an intersting API cause attempts to modify the Debug Filter Mask:

0056384C **NtSetDebugFilterState(ULONG ComponentId, unsigned int Level, char State)**

```
0056384C      mov     edi, edi
0056384E      push    ebp
0056384F      mov     ebp, esp
00563851      mov     eax, large fs:124h       ;KTHREAD
00563857      movsx   eax, byte ptr [eax+140h] ;KTHREAD->PreviousMode
0056385E      push    eax
0056385F      push    ds:_SeDebugPrivilege.HighPart
00563865      push    ds:_SeDebugPrivilege.LowPart ;PrivilegeValue
0056386B      call    SeSinglePrivilegeCheck
00563870      test    al, al
00563872      jz      short loc_5638BE ;If PrivilegeValue does not match, exit
                                         and return 0xC00000022 error
```

SeSinglePrivilegeCheck checks for the passed privilege value in the context of the current thread, If *PreviousMode* is **KernelMode**, the privilege check always succeeds. Otherwise, this routine uses the token of the user-mode thread to determine whether the current (user-mode) thread has been granted the given privilege.

Here the rest of the function

```
v3 = &Kd_WIN2000_Mask;
 if ( ComponentId >= KdComponentTableSize )
    {
      if ( ComponentId != 0xFFFFFFFF )
        return 0xC00000EF;
    }
else
    {
      v3 = (int *)*(&KdComponentTable + ComponentId);
    }
    if ( Level <= 0x1F )
```

```
      v4 = 1 << (char)Level;
    else
      v4 = Level;
    v6 = v4;
    if ( !State )
      v6 = 0;
    *v3 = v6 | *v3 & ~v4;
    result = STATUS_SUCCESS;
  }
```

Now we can implement a little Anti-Debug trick based on Debug State Awareness, indeed with NtSetDebugFilterState we are able to determine if the process is debugged or not:

```
#define WIN32_LEAN_AND_MEAN
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>
#include "ntDefs.h"

#pragma comment(lib,"ntdll.lib")

int main(void)
{
      NTSTATUS ntStatus;

      ntStatus = NtSetDebugFilterState(0,0,TRUE);

      if (ntStatus != STATUS_SUCCESS)
            MessageBoxA(NULL,"Not Debugged","Warning",MB_OK);

      else
            MessageBoxA(NULL,"Debugged","Warning",MB_OK);

      return (EXIT_SUCCESS);
}
```

→ ntDefs.h

```
typedef LONG NTSTATUS;
#define STATUS_SUCCESS ((NTSTATUS)0x00000000L)
```

```
extern "C"
__declspec(dllimport)
ULONG __stdcall
NtSetDebugFilterState(
                                ULONG ComponentId,
                                ULONG Level,
                                BOOLEAN State
                                );
```

Trick is really basilar if the Process is Debugged NtSetDebugFilterState returns STATUS_SUCCESS else returns 0xC00000022 Error Code. May be that this trick is already used, but for sure I haven's seen nothing about NtQueryDebugFilterState/NtSetDebugFilterState =)

Refs:

http://msdn.microsoft.com/en-us/library/ms792789.aspx
http://msdn.microsoft.com/en-us/library/ms804344.aspx

Thanks to #bug channel especially ratsoul 'n swirl

Regards,
Giuseppe 'Evilcry' Bonfa'