# Any sufficiently advanced uninstaller is indistinguishable from malware

devblogs.microsoft.com/oldnewthing/20230911-00

September 11, 2023

Raymond Chen

There was a spike in Explorer crashes that resulted in the instruction pointer out in the middle of nowhere.

```
0:000> r
eax=00000001 ebx=008bf8aa ecx=77231cf3 edx=00000000 esi=008bf680 edi=008bf8a8
eip=7077c100 esp=008bf664 ebp=008bf678 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
7077c100 ??              ???
```

Maybe the return address tells us something.

```
0:000> u poi esp
008bf6d4 test    eax,eax
008bf6d6 je      008bf6b9
008bf6d8 xor     edi,edi
008bf6da cmp     dword ptr [esi+430h],edi
```

It's strange that we're executing from someplace that has no name. If you look closely, you'll see that we are executing code *from the stack*: `esp` is `008bf664`, so the code that went haywire is on the stack.

Who executes code from the stack?

Malware, that's who.

Let's see what this malware is trying to do.

Disassembling around the last known good code address gives us this:

```
008bf6c4 call    dword ptr [esi+214h]
008bf6ca inc     dword ptr [ebp+8]
008bf6cd push    edi
008bf6ce call    dword ptr [esi+210h]   ; this called into space
008bf6d4 test    eax,eax
008bf6d6 je      008bf6b9
008bf6d8 xor     edi,edi
008bf6da cmp     dword ptr [esi+430h],edi
008bf6e0 je      008bf70d
```

It looks like the payload stored function pointers at `esi+210` and `esi+214`. Let's see what's there. This is probably where the payload stashed all its call targets.

```
0:000> dps @esi+200
008bf880  1475ff71
008bf884  00000004
008bf888  76daecf0 kernel32!WaitForSingleObject
008bf88c  76daeb00 kernel32!CloseHandle
008bf890  7077c100
008bf894  76dada90 kernel32!SleepStub
008bf898  76db6a40 kernel32!ExitProcessImplementation
008bf89c  76daf140 kernel32!RemoveDirectoryW
008bf8a0  76da6e30 kernel32!GetLastErrorStub
008bf8a4  770d53f0 user32!ExitWindowsEx
008bf8a8  003a0043
008bf8ac  0050005c
008bf8b0  006f0072
008bf8b4  00720067
008bf8b8  006d0061
```

Yup, there's a payload of function pointers here. It looks like this malware is going to wait for something, and then exit the process, or remove a directory, or exit Windows. Those bytes after `user32!ExitWindowsEx` look like a Unicode string, so let's dump them as a string:

```
0:000> du 008bf8a8
008bf8a8  "C:\Program Files\Contoso\contoso_update.exe"
```

Wait, what? It is trying to mess around with Contoso's auto-updater?

Let's take a look at more of the malware payload. Maybe we can figure out what it's doing. It looks like it's using `esi` as its base of operations, so let's disassemble from `esi`.

```
008bf684 push    ebp                        ; build stack frame
008bf685 mov     ebp,esp
008bf687 push    ebx                        ; save ebx
008bf688 push    esi                        ; save esi
008bf689 mov     esi,dword ptr [ebp+8]      ; parameter
008bf68c push    edi                        ; save edi
008bf68d push    0FFFFFFFFh                 ; INFINITE
008bf68f push    dword ptr [esi+204h]       ; data->hProcess
008bf695 lea     ebx,[esi+22Ah]             ; address of path + 2
008bf69b call    dword ptr [esi+208h]       ; WaitForSingleObject
008bf6a1 push    dword ptr [esi+204h]       ; data->hProcess
008bf6a7 call    dword ptr [esi+20Ch]       ; CloseHandle

008bf6ad and     dword ptr [ebp+8],0        ; count = 0
008bf6b1 lea     edi,[esi+228h]             ; address of path
008bf6b7 jmp     008bf6cd                   ; enter loop
008bf6b9 cmp     dword ptr [ebp+8],28h      ; waited too long?
008bf6bd jge     008bf6d8                   ; then stop
008bf6bf push    1F4h                       ; 500
008bf6c4 call    dword ptr [esi+214h]       ; Sleep
008bf6ca inc     dword ptr [ebp+8]          ; count++
008bf6cd push    edi                        ; path
008bf6ce call    dword ptr [esi+210h]       ; DeleteFile
008bf6d4 test    eax,eax                    ; Q: Did it delete?
008bf6d6 je      008bf6b9                   ; N: Loop and try again

008bf6d8 xor     edi,edi
008bf6da cmp     dword ptr [esi+430h],edi   ; data->fRemoveDirectory?
008bf6e0 je      008bf70d                   ; N: Skip
008bf6e2 jmp     008bf6f0                   ; Enter loop for trimming file name
008bf6e4 cmp     ax,5Ch                     ; Q: Backslash?
008bf6e8 jne     008bf6ed                   ; N: Ignore
008bf6ea mov     dword ptr [ebp+8],ebx      ; Remember location of last backslash
008bf6ed add     ebx,2                      ; Move to character
008bf6f0 movzx   eax,word ptr [ebx]         ; Fetch next character
008bf6f3 cmp     ax,di                      ; Q: End of string?
008bf6f6 jne     008bf6e4                   ; N: Keep looking

008bf6f8 mov     ecx,dword ptr [ebp+8]      ; Get location of last backslash
008bf6fb xor     eax,eax                    ; eax = 0
008bf6fd mov     word ptr [ecx],ax          ; Terminate string at last backslash
008bf700 lea     eax,[esi+228h]             ; Get path (now without file name)
008bf706 push    eax                        ; Push address
008bf707 call    dword ptr [esi+21Ch]       ; RemoveDirectory

008bf70d cmp     dword ptr [esi+434h],edi   ; data->fExitWindows?
008bf713 je      008bf71e                   ; N: Skip
008bf715 push    edi                        ; dwReason = 0
008bf716 push    12h                        ; EWX_REBOOT | EWX_FORCEIFHUNG
008bf718 call    dword ptr [esi+224h]       ; ExitWindowsEx

008bf71e push    edi                        ; dwExitCode = 0
```

```
008bf71f call    dword ptr [esi+218h]       ; ExitProcess
008bf725 pop     edi
008bf726 pop     esi
008bf727 pop     ebx
008bf728 pop     ebp
008bf729 ret

; This code appears to be unused
008bf72a push    ebp
008bf72b mov     ebp,esp
008bf72d push    esi
008bf72e mov     esi,dword ptr [ebp+10h]
008bf731 test    esi,esi
008bf733 jle     008bf746
...
```

Reverse-compiling back to C, we have

```
struct Data
{
    char code[0x0204];
    HANDLE hProcess;
    DWORD (CALLBACK* WaitForSingleObject)(HANDLE, DWORD);
    BOOL (CALLBACK* CloseHandle)(HANDLE);
    DWORD (CALLBACK* MysteryFunction)(PCWSTR);
    void (CALLBACK* Sleep)(DWORD);
    void (CALLBACK* ExitProcess)(UINT);
    BOOL (CALLBACK* RemoveDirectoryW)(PCWSTR);
    DWORD (CALLBACK* GetLastError)();
    BOOL (CALLBACK* ExitWindowsEx)(UINT, DWORD);
    wchar_t path[MAX_PATH];
    BOOL fRemoveDirectory;
    BOOL fExitWindows;
};
void Payload(Data* data)
{
    // Wait for the process to exit
    data->WaitForSingleObject(data->hProcess, INFINITE);
    data->CloseHandle(data->hProcess);

    // Try up to 20 seconds to do something with the file
    for (int count = 0;
        !data->MysteryFunction(data->path) && count < 40;
        count++) {
        Sleep(500);
    }

    if (data->fRemoveDirectory) {
        PWSTR p = &data->path[1];
        PWSTR lastBackslash = p;
        while (*p != L'\0') {
            if (*p == L'\\') lastBackslash = p;
            p++;
        }
        *lastBackslash = L'\0';
        RemoveDirectoryW(data->path);
    }

    if (data->fExitWindows) {
        ExitWindowsEx(EWX_REBOOT | EWX_FORCEIFHUNG, 0);
    }
}
```

Aha, this isn't malware. This is an uninstaller!

The mystery function is almost certainly `DeleteFileW`. It's waiting for the main uninstaller to exit, so it can delete the binary.

There is a page on CodeProject that shows how to write a self-deleting file, and it seems that multiple companies have decided to use that code to implement their own uninstallers. (Whether they follow the licensing terms for that code I do not know.)

Okay, so why did we crash? What went wrong with `DeleteFileW`?

According to the dump file, the spot where `DeleteFileW` was supposed to be instead holds `7077c100`. This is a function pointer into some mystery DLL that isn't loaded. How did that happen?

My guess is that the `DeleteFileW` function was detoured in the Contoso uninstaller. When the uninstaller tried to built its table of useful functions, it ended up getting not the address of `DeleteFileW` but the address of a detour. It then tried to call that detour from its payload, but since the detour is not installed in Explorer (or if it is, the detour is in some other location), it ended up calling into space.

Neither code injection nor detouring is officially supported. I can't tell who did the detouring. Maybe somebody added a detour to the uninstaller, unaware that the uninstaller is going to inject a call to the detour into Explorer. Or maybe the detour was injected by anti-malware software. Or maybe the detour was injected by Windows' own application compatibility layer. Whatever the reason, the result was a crash in Explorer.

Which means that people like me spend a lot of time studying these crashes to figure out what is going on, only to conclude that they were caused by other people abusing the system.

If you want to create a self-deleting binary, please don't use code injection into somebody else's process. Here's a way to delete a binary and leave no trace:

Create a temporary file called `cleanup.js` that goes like this:

```
var fso = new ActiveXObject("Scripting.FileSystemObject");
fso.DeleteFile("C:\\Users\\Name\\AppData\\Local\\Temp\\cleanup.js");

var path = "C:\\Program Files\\Contoso\\contoso_update.exe";
for (var count = 0; fso.FileExists(path) && count < 40; count++) {
    try { fso.DeleteFile(path); break; } catch (e) { }
    WSH.Sleep(500);
}
```

This script deletes itself and then tries to delete `contoso_update.exe` for 20 seconds. Run it with `wscript cleanup.js` and let it do its thing. No code injection, no detours, all documented.