DLL shell game and other misdirections: the Windows's native loader is a magician

Lucas Georges
lucas.georges@synacktiv.com

Synacktiv

Abstract. Windows developers extensively use shared libraries (DLLs) in order to maximize code reuse and update subcomponents independently on deployed systems. However, using shared libraries also opens up a myriad of issues coming from DLL incompatibilities, also known as DLL Hell (or more generically speaking dependency hell). That's why over the years the Windows core team has implemented various magic tricks based on DLL redirection to keep systems up to date while retaining backwards compatibility.

In this article we present several of these sleight of hands as well as other ways to dynamically load libraries, and some vulnerabilities that can be exploited via DLL hijacking still present in modern software.

Finally, this article also present `Dependencies` [7], a tool written by the author to analyze and troubleshoot DLL dependency issues on modern Windows binaries.

1 Introduction

A DLL (Dynamic-link library) is a shared library that has the same file format as Windows EXE files: the PE (Portable Executable). It is usually used to provide code that can be executed by other applications and to allow them to be structured in a modular fashion.

For example, if a programmer wants to write code with registry access, he can use the functions exported by the library `advapi32.dll` that exports code to manipulate the Windows registry.

A DLL implements and provides exported functions for any application that can import and call them. The PE file header includes information about external functions used from a library and functions to be used by other applications. This information is stored respectively in the Import and the Export Directories and are parsed by the Windows loader to resolve the dependencies.

A DLL can be loaded at process creation if it is present in the executable's Import Directory entries, or dynamically through the `LoadLibrary` function.

1.1 Dependency resolution

When a process is created, the kernel performs some operations such as setting up the `EPROCESS` object, initializing the `PEB` and mapping `ntdll.dll` in the process memory space.

A thread is created by the kernel in the process context and the function `LdrInitializeThunk` is executed. It is exported by `ntdll.dll` and is used to initialize the loader. The functions that are part of the loader are easily identified (their names begin with `Ldr*`) and are located in the `ntdll` module.

Among other things, the Windows loader is responsible for parsing the PE File header and resolve the dependencies.

1. The Import Directory from the PE Header is parsed to know which DLLs are needed. For each DLL, a first check is done to know if it is already loaded by checking the structure `PEB_LDR_DATA` from the PEB. This structure contains a list of the loaded modules. If the DLL is a *known DLL*, it has already been loaded at startup and is accessible through the global memory mapped file. Otherwise the DLL is loaded and mapped in the process address space (relocations are also performed if needed by parsing the `.reloc` section).
2. When the DLL is loaded, the `EAT` (*Export Address Table*), which contains the offset (RVA) of the functions exported by the module, is parsed to look for the functions needed by the application. The absolute address of these functions are then computed by adding the module base address. The `IAT` (*Import Address Table*) of the application will be filled with these addresses. The `IAT` is a table of function pointers. It is useful because a static address of a function exported by a DLL cannot be called directly [1]. Indeed, at compile time, the addresses where the modules will be loaded in the process are not known. So a function pointer located in the `IAT` will be called (filled by the loader during dependencies resolution).
3. Last but not least, the main entry function `DllMain` is called for each of the modules loaded in the process address space [2].

There is also a special type of exported functions that only act as forwarder. For example, the function `EnterCriticalSection` exported

1. Except if the flag `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` of the DLLCharacteristics field from the PE Header is not set.

2. It does not happen if the flag `IMAGE_FILE_DLL` is not set in the field `Characteristics` in the PE File Header.

by `Kernel32` acts only as a redirection to `RtlEnterCriticalSection` exported by another DLL. In the case of a forwarded export, the loader perform the exact same steps as above.

1.2 Windows Search folders

When an application does not provide the full path of a DLL to be loaded or does not use other mechanisms such as a manifest or a DLL redirection, Windows attempts to locate the DLL by searching through a list of locations in a fixed order.

1. The directory from which the application is loaded,
2. The system directory (for example, `C:\Windows\System32`),
3. The 16-bit system directory (for example, `C:\Windows\System`),
4. The Windows directory (for example, `C:\Windows`),
5. The current directory,
6. Directories that are listed in the `PATH` environment variable.

This order is the one that is used nowadays with the *Safe DLL search mode* enabled. When it is disabled (by default on Windows XP), Windows will look for the file to be loaded in the current directory before looking at the system directory.

Malware authors have used this search feature for years by placing a malicious DLL in the same folder as an application that loads a DLL without providing the full path. If the DLL has the same name and the same export names as the legit one, it will be used by the application [13].

The standard search order can be changed by calling the function `LoadLibraryEx` with the flag `LOAD_WITH_ALTERED_SEARCH_PATH` or by calling the `SetDllDirectory` function [12]. By using these functions, a specific directory can be specified to look for the DLL. In this case, the system will begin to search in this directory and then (if the DLL was not found), in the other folders in the default search order[3].

1.3 `Dependency Walker`, a tool to analyze loading dependencies

`Dependency Walker` [14] is a program used to list dependent modules of a Windows 32/64-bit Portable Executable file. It displays a recursive

3. The function `Add/SetDllDirectory` or the function `LoadLibraryEx` can be called with other flags such as `LOAD_LIBRARY_SEARCH_DLL`, `LOAD_LIBRARY_SEARCH_SYSTEM32` to tweak the search order.

tree of all dependent modules and can list all the exported and imported functions for each of the modules.

It was included in several Microsoft products such as Visual Studio or the Windows SDK but it was never publicly supported by Microsoft. The project seems to be discontinued as the last version was built in 2006.

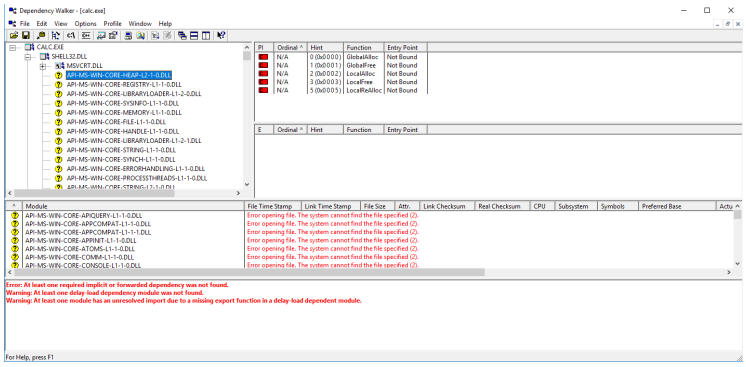


Fig. 1. `Dependency Walker`.

As shown in figure 1, `Dependency Walker` is outdated as it does not handle the API-sets introduced in Windows 7. The main motivation behind writing `Dependencies` [7], presented in figure 2 was to have an open-source alternative that could be maintained and evolve along the Windows DLL loader.

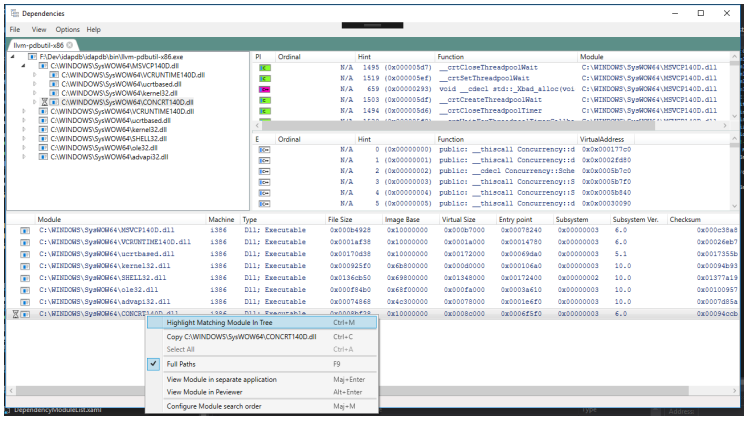


Fig. 2. `Dependencies`.

2 Redirection mechanisms

2.1 API Set DLLs

API Set DLLs is a fairly documented feature of Windows (though mainly by third-party researchers [2]) introduced by Microsoft since Windows 7. They were part of a major refactoring necessary to accommodate the fact that Windows uses the "same" NT kernel for a great diversity of platforms (desktops, servers, XBox and historically Windows Phone). You can read more on this, straight from Windows developers themselves: *One Windows Kernel* [11].

API Sets DLLs like `api-ms-win-eventing-provider-l1-1-0.dll` are "virtual" DLLs in the sense they are not actually present on disk. Instead there is a mapping (the API Set schema) which indicates which "host" DLL actually implements the API Set contract (e.g. `kernelbase.dll` for desktop Windows). This mapping is stored as a hash table present in every user process memory mapping and is accessible via the `PEB.ApiSetMap` pointer. Here is how the API Set schema itself is loaded (see also figure 3):

1. `winload.exe` (Windows Bootloader) loads the `ApiSetSchema.dll` from an hard-coded path in `System32`, and extract its `.apiset` section into a member of `KeLoaderBlock`, the loading context used to pass data between boot world and kernel world.
2. `winload.exe` loads and hand over to `ntoskrnl.exe`, Windows NT kernel. `ntoskrnl.exe` is actually compiled as a Windows driver (a special one though) and `winload.exe`'s `KeLoaderBlock` is passed through `ntoskrnl.exe`'s "`DriverEntry`" as its `DriverObject`.
3. On kernel startup, `MiInitializeApiSets` is called, and copies the API Set schema from the `KeLoaderBlock` into an undocumented static variable (called `nt!g_ApiSetSchema` in the drawing).
4. On a new user process creation (`NtCreateProcess`), `PspSetupUserProcessAddressSpace` is called, and calls `MmMapApiSetView` in order to create a new memory mapping and copy the API Set schema into the user process virtual memory. The user process `EPROCESS->Peb->ApiSetMap` variable is then modified to point to this new memory mapping.

Once the `ApiSetMap` is set, the API Set redirection is handled entirely in userland [5], via the use of helpers such as:

— `ntdll!ApiSetQueryApiSetPresence`: high-level API which only checks whether the specified DLL name is associated with an API Set contract;

— `ntdll!ApiSetResolveToHost`: high-level API to lookup the host DLL possibly associated with an API Set contract name;
— `ntdll!ApiSetpSearchForApiSet`: `ApiSetMap` hash table lookup;
— `ntdll!ApiSetpSearchForApiSetHost`: discriminate between hosts DLL in the (rare) case an API Set contract points to several hosts.

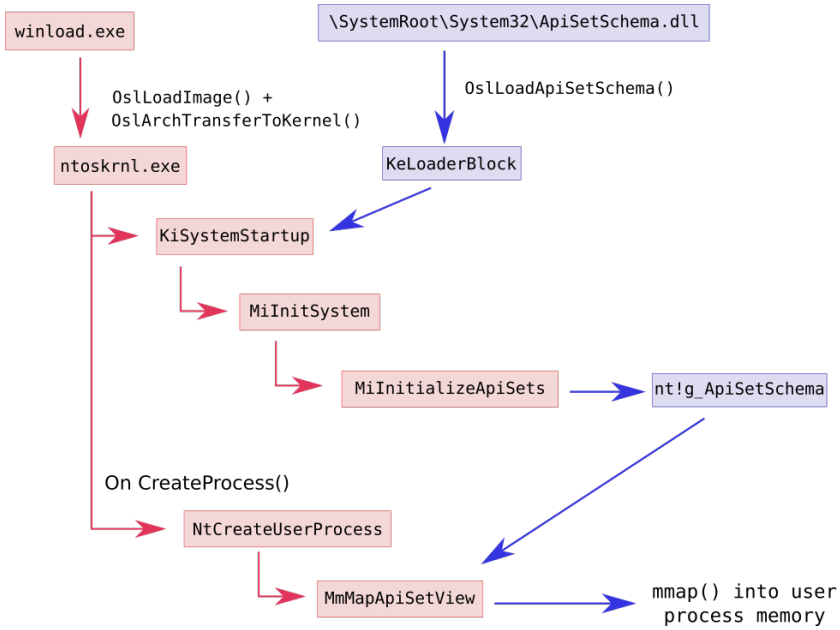


Fig. 3. API Set Schema load mechanism.

2.2 WinSxS

WinSxs (also known as SxS, side-by-side assemblies or *Fusion*) is a file redirection mechanism specifically created to fix the "DLL dependency Hell" issue.

There is a special resource called a *Manifest*, which can be either embedded within the process PE file or in an external file. In this manifest, specific dependencies can be added, and more importantly their "compatibility version" can be specified. This is mainly used to handle `comctl32.dll` dependency (which orchestrates the GUI side and gives applications a Windows "look and feel") in order to keep a program's "theme" consistent across Windows OS versions.

Every SxS dependency is declared using an external file via the `<file>` anchor like in the Chrome (listing 1), or by using the `<dependentAssembly>` XML anchor for a `Publisher` dependency, as `Microsoft.Windows.Common-Controls` for `notepad.exe` (listing 2).

```
<assembly
  xmlns='urn:schemas-microsoft-com:asm.v1' manifestVersion='1.0'>
  <assemblyIdentity
      name='71.0.3578.98'
      version='71.0.3578.98'
      type='win32'/>
  <file name='chrome_elf.dll'/>
</assembly>
```

Listing 1. `C:\Program Files (x86)\Google\Chrome\Application\71.0.3578.98\71.0.3578.98.manifest` external manifest

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="
    1.0">
  <assemblyIdentity name="Microsoft.Windows.Shell.notepad"
      processorArchitecture="amd64" version="5.1.0.0" type="win32" /
      >
  <description>Windows Shell</description>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.Windows.Common-
          Controls" version="6.0.0.0" processorArchitecture="*"
          publicKeyToken="6595b64144ccf1df" language="*" />
    </dependentAssembly>
  </dependency>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="asInvoker" uiAccess="false"
            />
      </requestedPrivileges>
    </security>
  </trustInfo>
  <application xmlns="urn:schemas-microsoft-com:asm.v3">
    <windowsSettings>
    </windowsSettings>
  </application>
</assembly>
```

Listing 2. `C:\Windows\System32\notepad.exe` embedded manifest

WinSxS redirection is handled by `csrss.exe`, another critical process of Windows and part of the "legacy" win32 subsystem (along with `smss.exe`, `lsass.exe`, `winit.exe`, `winlogon.exe` and `services.exe`). On a new process creation, the `csrss.exe` service in charge for the corresponding session is notified by the kernel, and tries to parse the

application's manifest. The manifest is used to create an "activation context" listing every DLL that needs to be side-loaded for this process. This activation context is then injected into the newly created target process PEB, and accessed by `ntdll.dll` on a module load.

WinSxS redirection is extremely complicated and so ancient probably not even most Windows developers know exactly what's going on underneath, but here is a basic searching sequence (taken from the MSDN [1]):

1. Side-by-side searches the WinSxS folder (`\\SystemRoot\WinSxs\`).
2. `$(PWD)\<assemblyname>.DLL`
3. `$(PWD)\<assemblyname>.manifest`
4. `$(PWD)\<assemblyname>\<assemblyname>.DLL`
5. `$(PWD)\<assemblyname>\<assemblyname>.manifest`

This is confirmed by looking at `sxs.dll`, the DLL in charge of probing and parsing application manifests (see figure 4).

```
.rdata:00000001650632A0 __manifest_paths dq offset aLNDll       ; DATA XREF: SxspGenerateManifestPathForProbing(ulong,ulong,
.rdata:00000001650632A0                                         ; "$.$L$N.DLL"
.rdata:00000001650632A8                 dq 13h
.rdata:00000001650632B0                 dq offset aLNManifest   ; "$.$L$N.MANIFEST"
.rdata:00000001650632B8                 dq 3
.rdata:00000001650632C0                 dq offset aLNNDll       ; "$.$L$N\\$N.DLL"
.rdata:00000001650632C8                 dq 1Bh
.rdata:00000001650632D0                 dq offset aLNNManifest  ; "$.$L$N\\$N.MANIFEST"
.rdata:00000001650632D8                 dq 0Bh
```

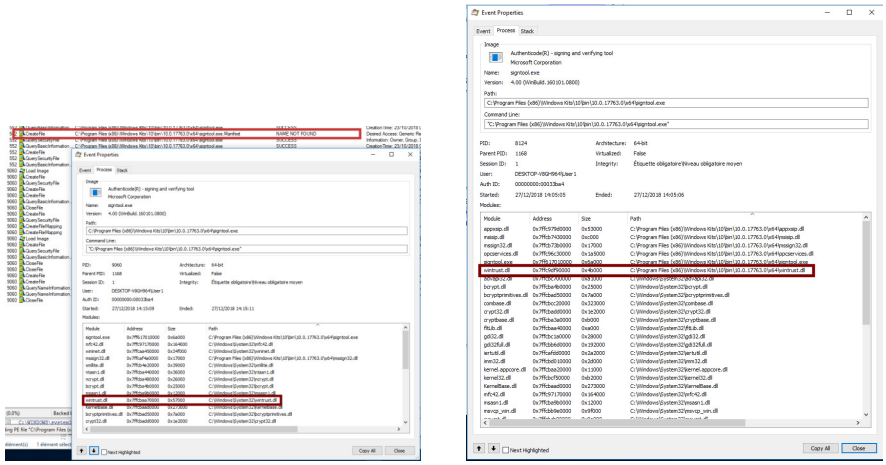
Fig. 4. `sxs.dll!SxspGenerateManifestPathForProbing`

As an example of external manifest redirection, figure 5 describes `signtool.exe`, an executable distributed with the WDK for developers to sign their PE binaries with a valid Authenticode signature, loading `wintrust.dll`, `mssign32.dll` and `appxsip.dll` via WinSxS side-loading.

This example is pretty egregious since `wintrust.dll` is actually a `KnownDll`, and should not be subject to DLL redirection.

WinSxS has also a fairly special (and fairly dangerous) redirection mechanism since it respects the old `.LOCAL` resolution order, where a DLL located in a `.local` folder (in the same current directory) can have precedence over a DLL present in a system path. This is the equivalent of the `LD_PRELOAD` macro in Linux, and has pretty much disappeared in modern Windows systems since it has been widely abused for `UAC` bypasses and privilege escalations.

As a rule of thumb, do not run trusted or privileged code from an untrusted location (e.g. a world writable folder) if WinSxS is involved since it allows an attacker a good variety of DLL redirections, based on the primitive he has.



(a) without an external manifest (b) with an external manifest

Fig. 5. Impact of an external manifest on `signtool`'s modules dependency resolution: `wintrust.dll` which is usually loaded from `C:\Windows\System32` is now loaded from the current directory.

2.3 KnownDlls

`KnownDlls` is an old Windows "trick" used to speed up process initialization by caching "hot" system DLLs that are pretty much always required (e.g. `ntdll.dll`, `kernel32.dll`, `kernelbase.dll`, etc.). The `KnownDlls` feature is implemented inside `smss.exe`, Windows Session Manager.

When a new process is launched, instead of loading `ntdll.dll` from the disk, the NT loader first checks if the module name is present in a special section called `\KnownDlls` (for x64 binaries) and, if present, maps it directly into the process memory using a *Copy-On-Write* (COW) mechanism. This caching feature has the big advantage to reduce disk I/O by compensating with a larger memory footprint (which is plentiful on modern systems anyway).

The DLLs to load as `KnownDlls` are listed under the registry key `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\KnownDLLs`.

```
PS C:\Users\User> cd HKLM:\
PS HKLM:\> cd SYSTEM\CurrentControlSet\Control\Session Manager\
    KnownDLLs
PS HKLM:\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs>
     Get-ItemProperty .
_wow64          : wow64.dll
_wow64cpu       : wow64cpu.dll
```

```
_wow64win     : wow64win.dll
_wowarmhw     : wowarmhw.dll
advapi32      : advapi32.dll
clbcatq       : clbcatq.dll
combase       : combase.dll
COMDLG32      : COMDLG32.dll
coml2         : coml2.dll
DifxApi       : difxapi.dll
gdi32         : gdi32.dll
gdiplus       : gdiplus.dll
IMAGEHLP      : IMAGEHLP.dll
IMM32         : IMM32.dll
kernel32      : kernel32.dll
MSCTF         : MSCTF.dll
MSVCRT        : MSVCRT.dll
NORMALIZ      : NORMALIZ.dll
NSI           : NSI.dll
ole32         : ole32.dll
OLEAUT32      : OLEAUT32.dll
PSAPI         : PSAPI.DLL
rpcrt4        : rpcrt4.dll
sechost       : sechost.dll
Setupapi      : Setupapi.dll
SHCORE        : SHCORE.dll
SHELL32       : SHELL32.dll
SHLWAPI       : SHLWAPI.dll
user32        : user32.dll
WLDAP32       : WLDAP32.dll
WS2_32        : WS2_32.dll
```

Listing 3. `KnownDlls` in registry.

However, there are sightly more `KnownDlls` listed in the `\KnownDlls` section than in the registry key, since the NT loader will cache every DLLs in the registry key but also their own DLL dependencies.

By listing the DLLs present in the `\KnownDLLs` section (listing 4), one can see that `ntdll.dll` is present in this section while not being part of the values in the registry (listing 3), since it's probably loaded by almost every DLL registered in the registry key.

```
PS C:\Users\User> .\Dependencies.exe -knowndll
  C:\WINDOWS\system32\advapi32.dll
  C:\WINDOWS\system32\bcryptPrimitives.dll
  C:\WINDOWS\system32\cfgmgr32.dll
  C:\WINDOWS\system32\clbcatq.dll
  C:\WINDOWS\system32\combase.dll
  C:\WINDOWS\system32\COMCTL32.dll
  C:\WINDOWS\system32\COMDLG32.dll
  C:\WINDOWS\system32\coml2.dll
  C:\WINDOWS\system32\CRYPT32.dll
  C:\WINDOWS\system32\difxapi.dll
  C:\WINDOWS\system32\FLTLIB.DLL
  C:\WINDOWS\system32\gdi32.dll
  C:\WINDOWS\system32\gdi32full.dll
```

```
C:\WINDOWS\system32\gdiplus.dll
C:\WINDOWS\system32\IMAGEHLP.dll
C:\WINDOWS\system32\IMM32.dll
C:\WINDOWS\system32\kernel.appcore.dll
C:\WINDOWS\system32\kernel32.dll
C:\WINDOWS\system32\KERNELBASE.dll
C:\WINDOWS\system32\MSASN1.dll
C:\WINDOWS\system32\MSCTF.dll
C:\WINDOWS\system32\msvcp_win.dll
C:\WINDOWS\system32\MSVCRT.dll
C:\WINDOWS\system32\NORMALIZ.dll
C:\WINDOWS\system32\NSI.dll
C:\WINDOWS\system32\ntdll.dll
C:\WINDOWS\system32\ole32.dll
C:\WINDOWS\system32\OLEAUT32.dll
C:\WINDOWS\system32\powrprof.dll
C:\WINDOWS\system32\profapi.dll
C:\WINDOWS\system32\PSAPI.DLL
C:\WINDOWS\system32\rpcrt4.dll
C:\WINDOWS\system32\sechost.dll
C:\WINDOWS\system32\Setupapi.dll
C:\WINDOWS\system32\SHCORE.dll
C:\WINDOWS\system32\SHELL32.dll
C:\WINDOWS\system32\SHLWAPI.dll
C:\WINDOWS\system32\ucrtbase.dll
C:\WINDOWS\system32\user32.dll
C:\WINDOWS\system32\win32u.dll
C:\WINDOWS\system32\windows.storage.dll
C:\WINDOWS\system32\WINTRUST.dll
C:\WINDOWS\system32\WLDAP32.dll
C:\WINDOWS\system32\wow64.dll
C:\WINDOWS\system32\wow64cpu.dll
C:\WINDOWS\system32\wow64win.dll
C:\WINDOWS\system32\WS2_32.dll
```

Listing 4. List of DLLs present in the `\KnownDLLs` section.

The `KnownDlls` load mechanism also doubles down as a security feature since its has the most precedence over regular search folders, and the loader is assured to load a "good" image.

The folder where the loader searches for `KnownDlls` used to be under the registry key `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\KnownDLLs\DllDirectory`. Instead, it's hard-coded using the undocumented `ntdll.dll` API `RtlGetNtSystemRoot` (which points to `C:\Windows`) and the values in figure 6

The `KnownDlls` registry key can only be updated by the `TrustedInstaller` user in order to prevent attackers from backdooring the `KnownDLLs` feature (although getting `TrustedInstaller` is not that big of an hassle for an attacker with a remote access, as shown by J.Forshaw [3]).

```
typedef struct _KNOWN_DLLS_INIT_STRUCT {
  UNICODE_STRING Folder;
  UNICODE_STRING SectionName;
  bool bCreateIfNotFound;

} KNOWN_DLLS_INIT_STRUCT, *PKNOWN_DLLS_INIT_STRUCT;

void SmpInitializeKnownDlls()
{
  KNOWN_DLLS_INIT_STRUCT KnownDlls[3] = {
    [0] = { // System32 : x64 dlls
      .Folder = RTL_CONSTANT_STRING("\\System32"),
      .SectionName = RTL_CONSTANT_STRING("\\KnownDlls"),
      .bCreateIfNotFound = true
    },
    [1] = { // SysWow64 : x86 dlls on x64 systems
      .Folder = RTL_CONSTANT_STRING("\\SysWOW64"),
      .SectionName = RTL_CONSTANT_STRING("\\KnownDlls32"),
      .bCreateIfNotFound = false
    },
    [2] = { // SysArm32 : x86 dlls on ARM systems ?
      .Folder = RTL_CONSTANT_STRING("\\SysArm32"),
      .SectionName = RTL_CONSTANT_STRING("\\KnownDllsArm32"),
      .bCreateIfNotFound = false
    },
  };

  // [...]
}
```

Fig. 6. `sms!SmpInitializeKnownDlls` is the function responsible to initialize KnownDlls sections based on the constants shown above.

`smss.exe` actually "abuses" the fact that `LdrVerifyImageMatchesChecksumEx` provides a callback feature which is called on every import found in the checked `Image` in order to recursively add every DLL dependencies found in the `KnownDLLs` list, as shown on listing 5 [4].

```
NTSTATUS NTAPI LdrVerifyImageMatchesChecksumEx(HANDLE Image,
    LDR_VERIFY_IMAGE_INFO *VerifyInfo)
{
  // Load and map DLL Image
  status = NtCreateSection();
  status = ZwMapViewOfSection();
  /* [...] */

  // actually check the checksum
```

4. Take note that this transitive dependency load only applies to "direct" imports but not for delay-load imports, this will be of significance later on in this article.

```
  if ( !LdrVerifyMappedImageMatchesChecksum(NULL,
      _ImageSectionOffset, _ImageInformation.EndOfFile.LowPart) )
    status = STATUS_IMAGE_CHECKSUM_MISMATCH;
    /* [...] */

  // Retrieve IMPORT_DATA_DIRECTORY
  status = RtlpImageDirectoryEntryToDataEx(
               NULL, NULL,
               IMAGE_DIRECTORY_ENTRY_IMPORT,
               &_LastRvaSection,
               &Import
  );

  // Iterate over IMAGE_IMPORT_DESCRIPTOR entries
  while ( 1 )
  {
    ImportNameRVA = Import->Name;
    if ( !ImportNameRVA)
      break;

    ImportNameAscii = RtlImageRvaToVa(NtHeaders, NULL, ImportNameRVA
        , &_LastRvaSection);

    // VerifyInfo->CallbackInfo.Callback is in reality sms.exe!
        SmpProcessModuleImports(HANDLE SmpContext, char *ImportName)
    // which add every import dependencies to the KnownDlls list.
    VerifyInfo->CallbackInfo.Callback(
      VerifyInfo->CallbackInfo.CallbackParameter,
      ImportNameAscii
    );

    ++Import;
  }
  return status;
}
```

Listing 5. Recursive processing of `KnownDlls` imports.

2.4 Delay-load DLL

Delay-loading is an hybrid way to load DLL at runtime. The idea behind it is to speed up process initialization by loading some dependencies in a lazy way: the actual DLL load (and associated cost from disk I/O) will be done the first time the parent process calls the import API. This is the same lazy loading idea that have been applied to various resources: virtual memory commits (via `#PF` interrupts), modern websites “infinite scrolling”, etc.

This can be used via the link directive `DELAYLOAD:\$(dll_name)` and, instead of creating an `IMAGE_IMPORT_DESCRIPTOR` entry in the assembled PE file import data directory, a similar structure entry called `IMAGE_DELAYLOADIMPORT_DESCRIPTOR` will be written in the delay-load

data directory. More interesting, the linker will also redirect every call to the imported APIs that are now delay-loaded by a resolver stub, usually called `__tailMerge_XXXX_dll`.

The "new" version of resolving delay-loading rely on calling `kernelbase.dll!ResolveDelayLoadedAPI`'s API (which relies on `ntdll.dll!LdrResolveDelayLoadedAPI`) since it has the advantage of being always compatible with the current OS the binary is running on.

However, older binaries used the previous version which embed a full DLL resolver inside the stub [5].

Both versions used `LoadLibrary` underneath for resolving DLLs location, but the older helper does not handle IAT export suppression [9] and that's probably the reason why it's not used anymore.

2.5 System32 redirection for 32-bit binaries

With the introduction of 64-bit architectures, most OSes need to support running 32-bit as well as 64-bit application (more commonly known as "multiarch"). While most Linux distributions, like Debian, chose to create a new folder for 64-bit system shared libs (`/lib64/`, while `/lib/` is for 32-bit binaries), Windows curiously chose to do the opposite.

`%windir%\System32\` which used to host system DLLs for 32-bit Windows (also called `x86` DLLs) now host 64-bits DLLs on 64-bit architectures (also called `x64`) while 32-bits DLLs (also called `WoW64`) are located in a new folder called `%windir%\SysWow64\`. The reason behind this philosophy is not frankly clear, but it was probably to ensure a smoother transition from 32- to 64-bit OS architecture. It's pretty frequent to see hard-coded `%windir%\System32\` paths in binaries that need to start services and initialize drivers.

However this decision is a major pain point for Windows: it breaks backwards compatibility. Thanks to WoW64 (Windows on Windows64) emulation, a previously compiled 32-bit executable can still run in 64-bit OSes, but tries to access `%windir%\System32\` in order to load additional DLLs or other files. In order to prevent the legacy programs from breaking, Windows developers have implemented a file system redirection which symlink `%windir%\System32\` to `%windir%\SysWow64\` for WoW64 binaries.

This redirection is implemented in userland at WoW emulation level, handled by `wow64.dll`, `wow64cpu.dll` and `wow64win.dll` binaries, usually when translating 32-bit system calls into native syscalls. For example,

5. Here an example of a full import resolver: `https://gist.github.com/lucasg/f3168c24615a9852963ae6c762a65926#file-delayload_helper_full-c`.

`wow64.dll!whNtCreateFile` is in charge of emulating `NtCreateFile` for 32-bit binaries, and calls `wow64.dll!RedirectPath` to actually redirect file operations on `%windir%\System32\` to `%windir%\SysWow64\`. Table 1 describes these redirections.

Original Path	Redirected Path
`C:\Windows\System32\ntdll.dll`	`C:\Windows\SysWOW64\ntdll.dll`
`C:\Windows\Sysnative\ntdll.dll`	`C:\Windows\System32\ntdll.dll`
`C:\Windows\System32\spool \prtprocs\x64\winprint.dll`	`C:\Windows\System32\spool \prtprocs\x64\winprint.dll`

Table 1. WoW64 folder redirection.

If a WoW64 binary still want to access `%windir%\System32\` resources, it has three ways to do it:

— Disable the file system redirection by calling `kernel32!Wow64DisableWow64FsRedirection`. However this only disable the redirection on the calling thread (by updating a flag on the current `TEB` structure) and should be quickly restored, lest to have unintended consequences.

— Use `%windir%\Sysnative\` which is a "virtual" folder (which is not present on disk) which points to `%windir%\System32\` on x64 systems. Wow64 exes can access native system file using this path.

— If the program has admin level privileges, place the resource in a subfolder which is exempt from wow64 folder redirection (see listing 6).

```
static UNICODE_STRING System32Exempts[] = {
  RTL_CONSTANT_STRING("\\catroot"),
  RTL_CONSTANT_STRING("\\catroot2"),
  RTL_CONSTANT_STRING("\\driverstore"),
  RTL_CONSTANT_STRING("\\drivers\\etc"),
  RTL_CONSTANT_STRING("\\hostdriverstore"),
  RTL_CONSTANT_STRING("\\logfiles"),
  RTL_CONSTANT_STRING("\\spool")
};
```

Listing 6. Wow64 folder redirection exemptions.

In the end this is a pretty known file redirection, but it can break WoW64 executables in really subtle ways (for example by mmap-ing the wrong system DLL).

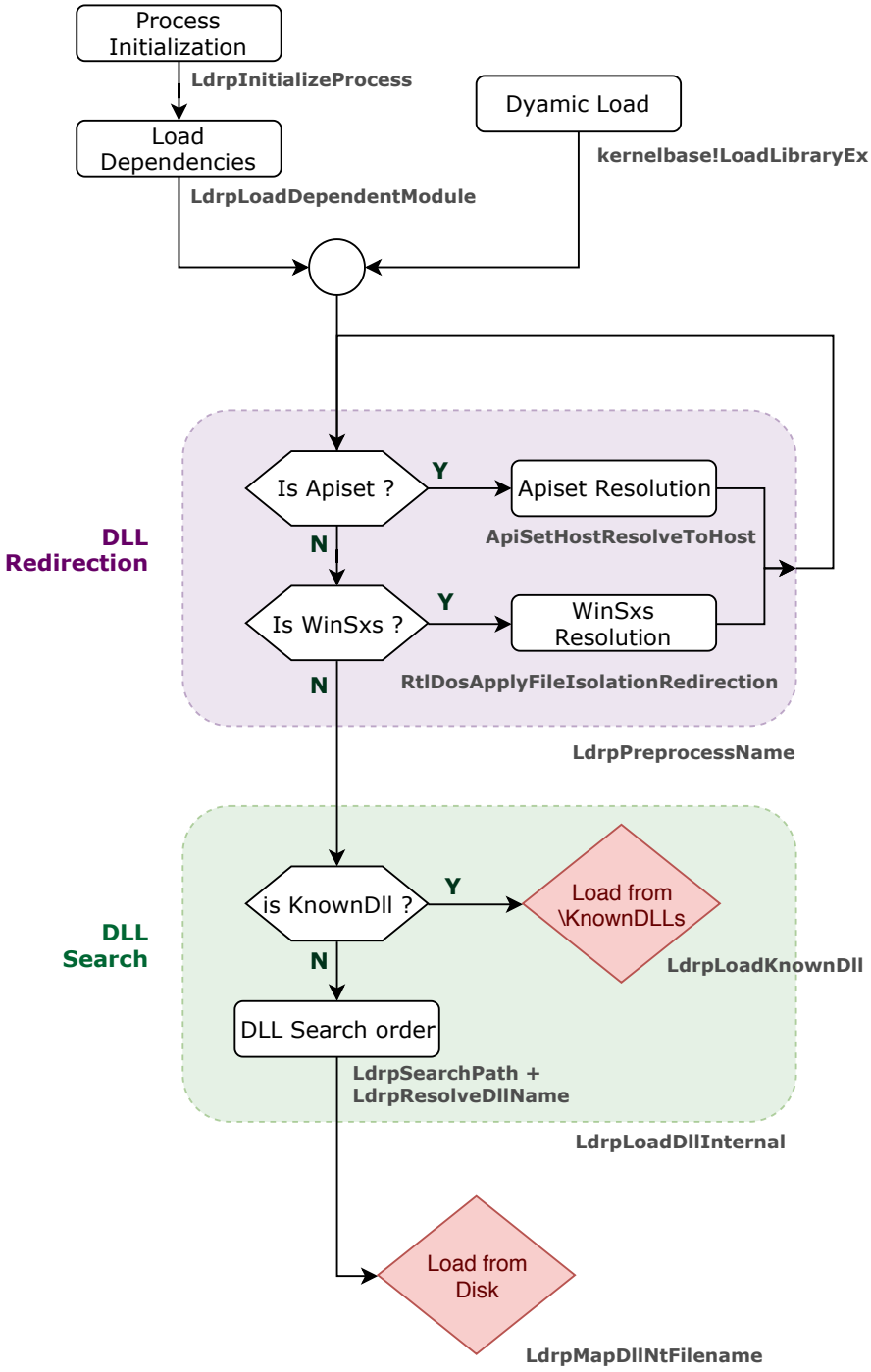


Fig. 7. Schematic flow chart of a DLL Dependency resolution and load.

2.6 Flow Chart

Figure 7 is a diagram summing up the various redirection mechanisms shown previously. Whether the module is loaded at process initialization (`ntdll!MapAndSnapDependencies`) or dynamically at a later time (`kernel32!LoadLibrary`), the control flow is the same [6]. The names below each step in the flow match the `ntdll` function responsible for the step in question, however those names are internal (coming from PDB files) and might be subject to future changes.

3 Vulnerabilities

In this part we present two vulnerabilities exhibiting features from DLL redirection mechanisms presented before:

— a *User to Admin* local privilege escalation affecting certain ASUS Zenbook models;

— a *User to SYSTEM* local privilege escalation affecting Opera, a major web browser.

The bugs are explained by the fact that privileged processes are executing code from world-writable folders. Both vulnerabilities have now been patched, but many others similar issues still probably lurks within third-party software.

3.1 Delay-load DLL hijack

On some ASUS Zenbook laptops, there is a scheduled task installed by default which launches an executable running with High Integrity level on user logon: `C:\Program Data\AsTouchPanel\AsPatchTouchPanel.exe`.

At this point, what this process does is not exactly obvious, but it seems to be a software stub for a "faulty" touch panel hardware feature on some laptops. However, something is sure: this is a **bad** idea to run privileged applications inside the `Program Data` folder, as shown in listing !

```
  PS C:\ProgramData\AsTouchPanel> (Get-Acl C:\ProgramData\
      AsTouchPanel\AsPatchTouchPanel.exe).Access

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
```

6. Actually there is a minor difference: `ntdll!LdrpLoadDependentModule` (called by `ntdll!MapAndSnapDependencies`) is heavily inlined while `ntdll!LdrpLoadDLL` (called by `kernel32!LoadLibrary`) is not, probably a byproduct of PGO (Profile-Guided Optimization.)

```
IsInherited       : True

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : True

FileSystemRights  : ReadAndExecute, Synchronize
AccessControlType : Allow
IdentityReference : BUILTIN\Users
IsInherited       : True

PS C:\ProgramData\AsTouchPanel> (Get-Acl C:\ProgramData\AsTouchPanel
    ).Access

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
IsInherited       : True

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : True

FileSystemRights  : 268435456
AccessControlType : Allow
IdentityReference : CREATOR OWNER
IsInherited       : True

FileSystemRights  : ReadAndExecute, Synchronize
AccessControlType : Allow
IdentityReference : BUILTIN\Users
IsInherited       : True

FileSystemRights  : Write
AccessControlType : Allow
IdentityReference : BUILTIN\Users
IsInherited       : True
```

Listing 7. ACLs sets on the executable and the parent directory

Since `C:\Program Data\AsTouchPanel\AsPatchTouchPanel.exe` was created by the `Admin` user, regular users can't simply rewrite it. However, by looking at the last ACL entry shown in listing 7, users have a `Write` privilege on the parent folder. Indeed, `%PROGRAM_DATA%` is a folder created for applications to store user-independent data (instead of using the registry or local `AppData` folders and thus this directory (and any subfolder that inherit its ACL from it) is User R/W by default.

An authenticated user have several privileges at his disposal. Specifically he can create files and folders in the same directory. With this primitive, it may be possible to plant a malicious DLL somewhere in the DLL search path. A first look at the dependency tree in figure 9 obtained

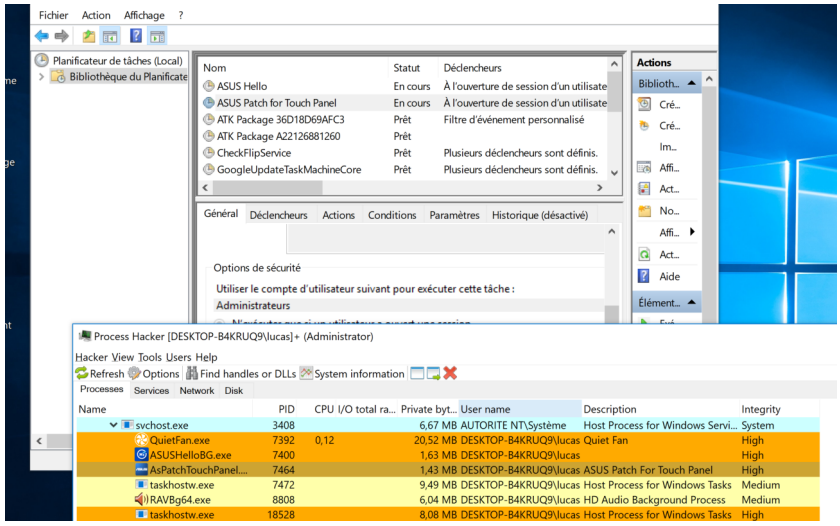


Fig. 8. There is actually quite a number of processes running as High Integrity...

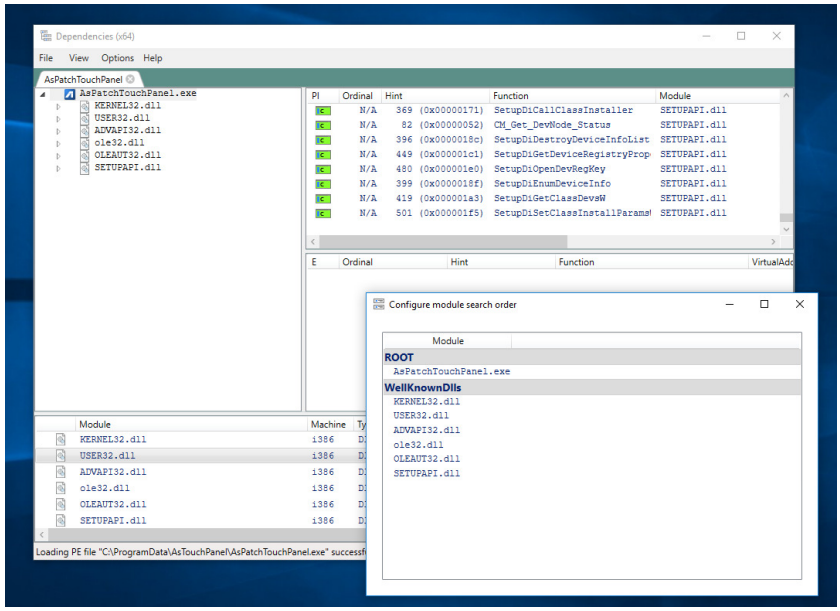


Fig. 9. It's `KnownDlls` all the way down.

with `Dependencies` [7] is not very encouraging: `AsPatchTouchPanel.exe` imports only `KnownDlls` that are not hijackable (except from WinSxS redirection, but this card can't be played here). And we've seen previously that `KnownDlls` dependencies are also `KnownDlls` (as shown on figure 9), which is recursively unhijackable... but is it really ?

During its main routine, `AsPatchTouchPanel.exe` calls `Setupapi.dll!SetupDiGetClassDevsW` in order to enumerate PnP nodes on the machine. Underneath, `Setupapi.dll` relays the call to `devobj.dll!DevObjCreateDeviceInfoList` and that's where it get interesting. Since `devobj.dll` is a delay-load DLL loaded by `Setupapi.dll`, it's not part of `KnownDlls`. This means the NT loader will try to load the DLL from the current directory before loading it from `C:\Windows\SysWoW64` (see figure 10).

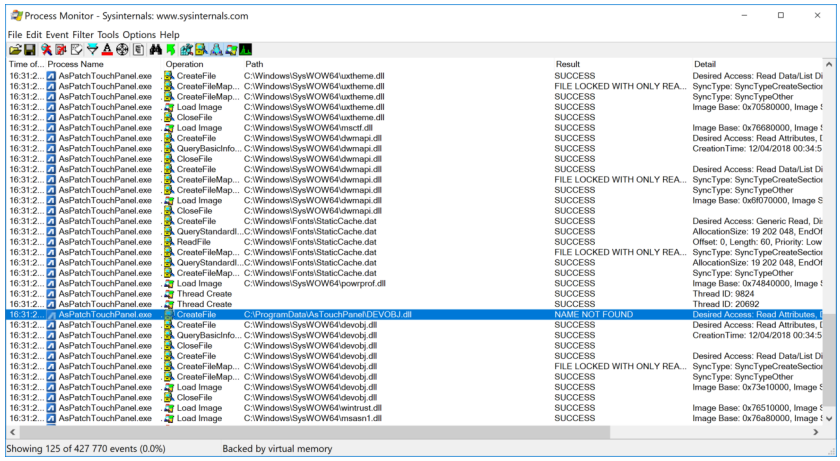


Fig. 10. `Procmon` trace showing that DLL search hijack is possible.

From this point on this is a walk in the park: plant a custom `devobj.dll` in `C:\Program Data\AsTouchPanel` and gain Admin privileges next time the user logs on.

This vulnerability has been reported to the Asus Security team in January 2019 and they pushed an update on vulnerable models in February. Since the TPIC patch v4.0, the ACL on `C:\Program Data\AsTouchPanel` has been fixed and is not accessible to users anymore.

3.2 WinSxS binary planting

When you install `Opera`, it sets up a scheduled task for its autoupdate that runs every day (and at every startup), the binary `C:\Program Files\Opera\Launcher.exe`, as `NT AUTHORITY\SYSTEM`. This task has an interesting trace on *ProcMon*, as shown in figure 11.

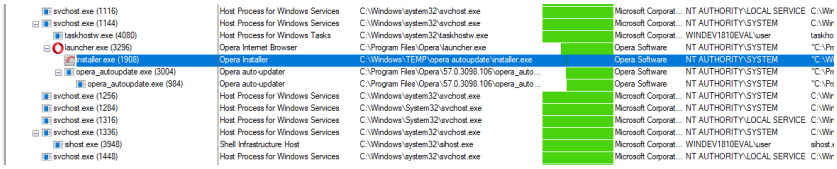


Fig. 11. Opera autoupdater task procmon trace, running as SYSTEM in %TEMP%.

Without really reversing `launcher.exe`, we can get the gist of it:

1. `launcher.exe` copy `installer.exe` from `C:\Program Files\Opera\$(version)\installer.exe` into a temporary directory, `C:\Windows\Temp\opera autoupdate`
2. `launcher.exe` calls `CreateProcess` on the temporary executable
3. `installer.exe` is executed and *also* drops a temporary DLL `C:\Windows\Temp\Opera_installer_$(timestamp).dll` which it then loaded.
4. `C:\Windows\Temp\opera autoupdate\installer.exe` is automatically deleted when the process exits.

The real issue here is to use `%TEMP%` (which still point to a world writable folder even for `SYSTEM` processes) to run elevated processes.

At this point, we can attack this vulnerability from several points: you can try to win the race between the moment where `launcher.exe` drops `installer.exe` (TOC) and the moment where it launches the installer (TOU) locking the executable from overwriting it. You can also try to win the race on the DLL dropped, since the "random" part of the name is pretty predictable. Or you can take advantage of the existing WinSxS redirection (see figure 12).

`installer.exe` usually loads `comctl32.dll` from the WinSxS publisher folder, but you can force it to load it from the current directory by planting a particular path (see figure 13).

The exploit code for the fake proxy DLL that creates a custom ScheduledTask using `NT AUTHORITY\SYSTEM` privileges is provided online [6].

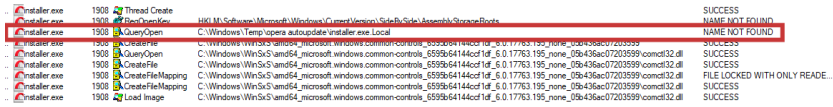


Fig. 12. WinSxS .local redirection is possible here.

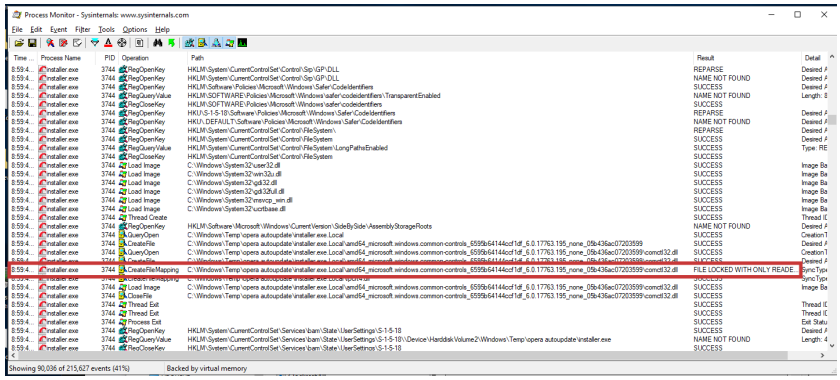


Fig. 13. You need to craft a correct proxy DLL if you don't want to crash the process.

Since Opera 58.0.3135.118, this vulnerability is fixed by setting correct ACL on the `opera autoupdate` folder, making it only accessible from admin level, as shown in listing 8.

```
PS C:\Windows\Temp> (Get-Acl "C:\Windows\Temp\opera autoupdate").
    Access

FileSystemRights  : 268435456
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
IsInherited       : False

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
IsInherited       : False

FileSystemRights  : 268435456
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : False

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : False
```

Listing 8. Opera temporary folder's ACL for autoupdate.

3.3 API Set Extensions abuse

There are two ways to extend the current API Set scheme:
— using a DynamicSchema extension;
— using an API Set Extension.

The dynamic method is a way to add optionals entries to the schema based on the value of an associated predicate. The extension list is currently hardcoded within the bootloader and used to load `traceext.sys` only if an undocumented boot flag is set (this is checked by `winload!OslpDtraceExtensionEnabled`):

```
.rdata:000000018012CA48 OslpDynamicSchemaExtensions dq offset ext_ms_win_ntos_trace_l1_1_0; Apiset
.rdata:000000018012CA48                                         ; DATA XREF: OslLoadApiSetSchema+1A1↑r
.rdata:000000018012CA48                                         ; OslLoadApiSetSchema+1A8↑o
.rdata:000000018012CA48                 dq offset _traceext_sys ; host ; "ext-ms-win-ntos-trace-l1-1-0" ...
.rdata:000000018012CA48                 dq offset OslpDtraceExtensionEnabled; callback
.rdata:000000018012CA60                 align 40h
```

More interestingly, the other method to extend the API Set schema is done via an API Set Extension. API Set Extension are additional files which "override" the default API Set schema that is also loaded at boot time by `winload.exe`.

First things first, `winload.exe` checks if the current schema is "sealed" (`winload!ApiSetIsSchemaSealed`): every API Set schema starts with an `API_SET_NAMESPACE` struct entry which has a flag member describing if the current schema is sealed. If set to true, the current schema cannot be modified. However, the schema under `C:\Windows\System32\ApiSetSchema.dll` is not currently sealed (but maybe in the future).

`winload!ApiSetpLoadSchemaExtensions` enumerates every subkeys within `ApiSetSchemaExtensions` registry key and tries to load the pointed API Set file under the `Filename` key. Listing 9 shows an example of a correct key set that will trigger a load.

```
PS C:\Users\user> cd HKLM:\
PS HKLM:\> cd "SYSTEM\CurrentControlSet\Control\Session Manager\
    ApiSetSchemaExtensions\CustomExt"
PS HKLM:\SYSTEM\CurrentControlSet\Control\Session Manager\
    ApiSetSchemaExtensions\CustomExt> Get-ItemProperty .

Filename      : apisetschema-mylittleextensions.dll
```

Listing 9. Registering a new API Set extension;

The loaded API Set file must respect the same file format as the original `apisetschema.dll`, and must be present in `SystemRoot` folder (usually `C:\Windows\System32`). The extension API Set schema must be

located at the start of a custom PE section called `.apiset`, and must respect the file format described in figure 14.

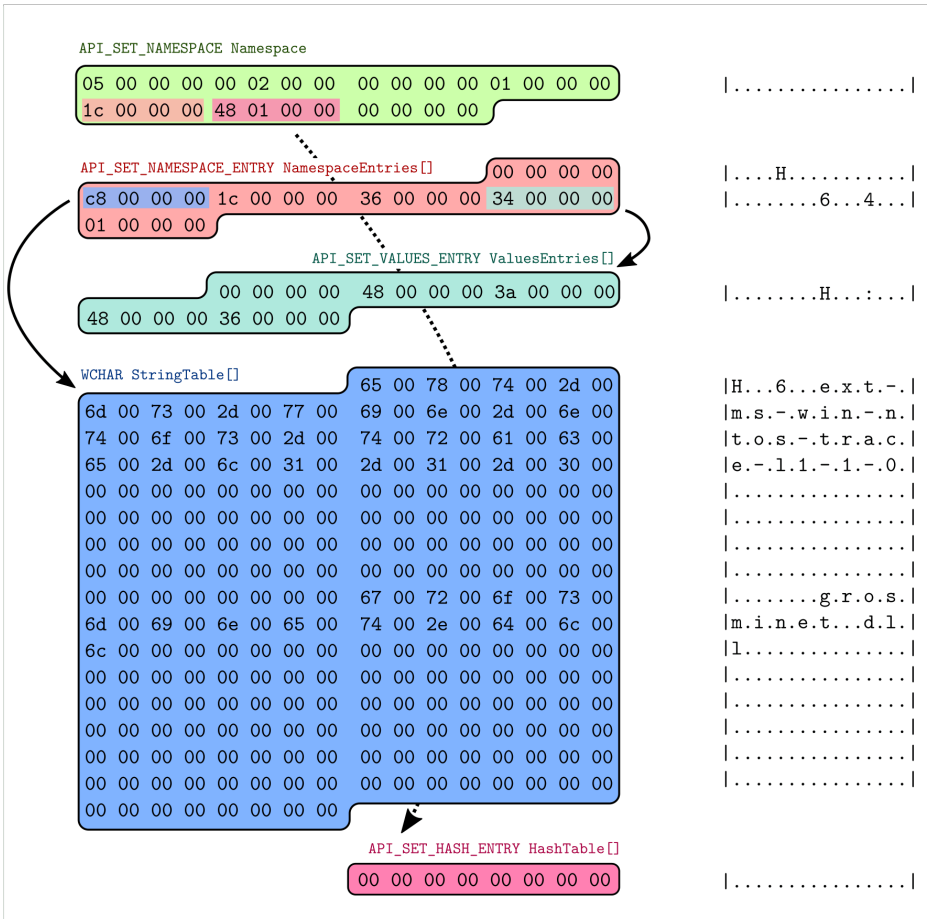


Fig. 14. File format for an apiset extension schema. `API_SET*` structures are defined in the Github repository for Windows internals [8]

This is a pretty annoying flat structure to craft since every field is accessed by its offset from the start of the structure, and the whole schema must be contained in a valid PE. [7]

Here's below the implementation of `ApiSetComposeSchema` which is called to merge the default API Set schema and the extension we provide.

7. There are examples of how to make a single redirection schema using only C code. [4]

There are several constraints we need to respect if we want `winload.exe` to properly load our extension:

- The API Set contract must be of the form `XXXXXXXXXXX-YY.dll`. `ApiSetComposeSchema` will rstrip everything after the last dash char found: this is because API Set DLLs are versioned in the following form: `api-min-win-XXXXXXX-lM-m-p.dll` with `M`, `m` and `p` standing for major version, minor and patch. The rstrip operation effectively ignores patch version.
- More importantly, the API Set contract *must already be present* in the current API Set schema. Unfortunately, we can't extend the schema by adding new entries.
- Lastly, the API Set `API_SET_VALUE_ENTRY` that will be updated must not be sealed.

After several hours debugging the windows bootloader, I managed to extend `ext-ms-win-hyperv-hvplatform-l1-1` which originally points to `winhvplatform.dll` to make it point towards a controlled binary in `system32` called `grosminet.dll`:

```
<#
    Before loading apiset extensions
#>
PS> .\Dependencies.exe -apisets | Select-String hyper
ext-ms-win-hyperv-compute-l1-1 -> [ vmcompute.dll ]
ext-ms-win-hyperv-hgs-l1-1 -> [ vmhgs.dll ]
ext-ms-win-hyperv-hvemulation-l1-1 -> [ winhvemulation.dll ]
ext-ms-win-hyperv-hvplatform-l1-1 -> [ winhvplatform.dll ]

<#
    After loading apiset extensions
#>
PS> .\Dependencies.exe -apisets | Select-String hyper
ext-ms-win-hyperv-compute-l1-1 -> [ vmcompute.dll ]
ext-ms-win-hyperv-hgs-l1-1 -> [ vmhgs.dll ]
ext-ms-win-hyperv-hvemulation-l1-1 -> [ winhvemulation.dll ]
ext-ms-win-hyperv-hvplatform-l1-1 -> [ grosminet.dll, winhvplatform.
    dll, grosminet.dll ]
```

Listing 10. Apiset redirection customization using Apiset extension.

This API Set schema extension is potentially "dangerous" in a backdooring scenario since an attacker with admin privileges can use it as a non-obvious rootkit mechanism, where a legit API Set contract can point to attacker's binaries only on specific hosts. The API Set schema is probably never verified on an organization level since cross-examining redirections to look for discrepancies is not the easiest thing to do (and it's contrary to the API Set design which is to allow custom DLL redirections).

Fortunately, Microsoft enforces the API Set extension to be "correctly" signed to be loaded (same as `hal.sys` or other kernel drivers). And thanks for that since there are bound checks missing in `winload.exe` parsing routines for the API Set schema extension, as shown in figure 15.

```
Windows Boot Debugger Kernel Version 17763 UP Free x64
Machine Name:
Primary image base = 0x00000000`00844000 Loaded module list = 0x00000000`00997a68
System Uptime: not available
winload!DebugService2+0x5:
00000000`00961c75 cc              int     3
kd> g
*** Windows is unable to verify the signature of
    the file \Windows\system32\apiset-crash.dll.  It will be allowed to load
    because the boot debugger is enabled.
Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
winload!ApiSetComposeSchema+0x90:
00000000`0093d1e8 428b44e104      mov     eax,dword ptr [rcx+r12*8+4]
kd> r rcx
rcx=fffff80164802fff
kd> r r12
r12=0000000000000000
kd> db rcx
fffff801`64802fff  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ??  ???????????????
fffff801`6480300f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ??  ???????????????
fffff801`6480301f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ??  ???????????????
fffff801`6480302f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ??  ???????????????
fffff801`6480303f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ??  ???????????????
fffff801`6480304f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ??  ???????????????
fffff801`6480305f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ??  ???????????????
fffff801`6480306f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ??  ???????????????
```

Fig. 15. `winload.exe` is dereferencing an API Set value entry way outside the API Set extension's bounds

4 Conclusion

In conclusion, while most of the redirection mechanisms shown previously were already known and for the most part somewhat documented, there is no previous work [8] which attempts to find the order in which they are applied (to the author's knowledge). In the end the way these redirections are layered is pretty straightforward, but that's with the benefit of hindsight.

This article also showcased vulnerabilities impacting well-known mainstream products, indicating that DLL hijacking vulnerabilities are still moderately present in third-party software (and this study didn't ever cover the class of issue that is binary planting for persistence where the attacker already has admin level privileges).

That is maybe the result of poor tooling: apart from Dependency Walker, there are few system tools, whether it's antivirus software, pentesting frameworks or auditing software like the Sysinternals, that take

8. Even the ever great Windows Internals series is pretty unequal when describing DLL loading and DLL name redirection [10].

DLL name redirections into account and try to automate away the issue of DLL hijacking. [9]

Hopefully, this public study as well as the open source software `Dependencies` [7] that was built upon it may help security software developers understand better the different aspects of DLL loading, and in turn build better tooling on top of it.

In the end, I would like to thanks my colleague Nicolas Correia for helping me write this article, as well as Tristan, Bruno and Fabien for reviewing it.

References

1. Microsoft Dev Center. Assembly Searching Sequence. `https://docs.microsoft.com/en-gb/windows/desktop/SbsCs/assembly-searching-sequence`.
2. Geoff Chapell. The API Set Schema. `https://www.geoffchappell.com/studies/windows/win32/apisetschema/index.htm`, 2016.
3. James Forshaw. The Art of Becoming TrustedInstaller. `https://tyranidslair.blogspot.com/2017/08/the-art-of-becoming-trustedinstaller.html`.
4. Lucas Georges. Apiset extension implementation. `https://gist.github.com/lucasg/f3168c24615a9852963ae6c762a65926#file-apiset_extension-h`.
5. Lucas Georges. Apiset Resolution. `https://lucasg.github.io/2017/10/15/Api-set-resolution/`.
6. Lucas Georges. COMCTL32 proxy dll. `https://gist.github.com/lucasg/f3168c24615a9852963ae6c762a65926#file-comctl32_proxy_dll-c`.
7. Lucas Georges. Dependencies. `https://www.github.com/lucasg/Dependencies`.
8. Pavel Yosifovich; Alex Ionescu. Windows Internals Github repository. `https://github.com/zodiacon/WindowsInternals/blob/master/APISetMap/ApiSet.h`.
9. Pavel Yosifovich; David A. Solomon; Alex Ionescu. Windows Internals, Part 1: System architecture, processes, threads, memory management, and more. `https://books.google.com/books?id=y83LDgAAQBAJ&pg=PT1062`, 2015.
10. Pavel Yosifovich; David A. Solomon; Alex Ionescu. Windows Internals, Part 1: System architecture, processes, threads, memory management, and more. `https://books.google.com/books?id=y83LDgAAQBAJ&pg=PT281`, 2015.
11. The Windows kernel team. One Windows Kernel. `https://techcommunity.microsoft.com/t5/Windows-Kernel-Internals/One-Windows-Kernel/ba-p/267142`, 2018.
12. Microsoft. Dynamic-Link Library Search Order. `https://docs.microsoft.com/en-us/windows/desktop/dlls/dynamic-link-library-search-order`.
13. Palo Alto Networks. PlugX Uses Legitimate Samsung Application for DLL Side-Loading. `https://unit42.paloaltonetworks.com/plugx-uses-legitimate-samsung-application-for-dll-side-loading/`.
14. Dependency Walker. Dependency Walker 2.2. `http://www.dependencywalker.com`.

9. All the vulnerabilities in this article were found “by hand” by the author.