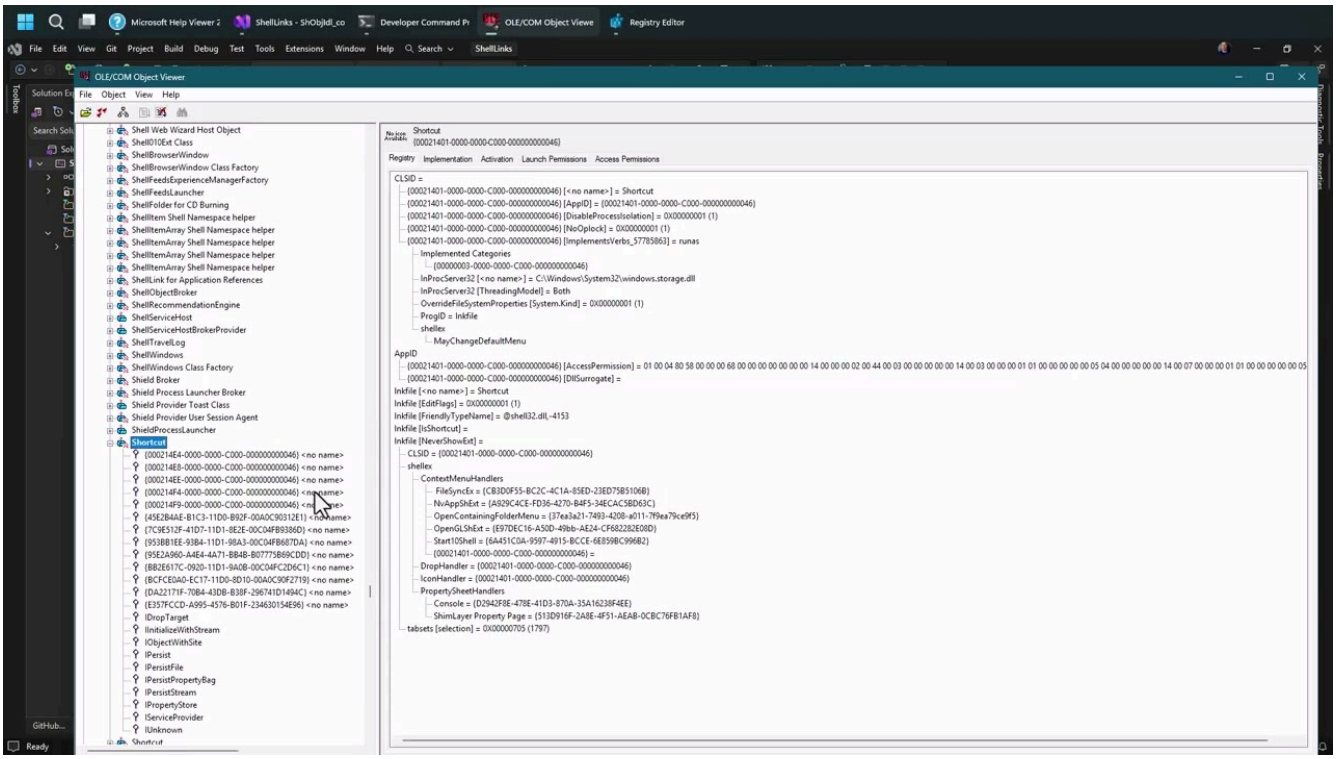
Windows Shell Links in C++: How to Read and Write .lnk Files

trainsec.net/library/windows-kernel/windows-shell-links-in-c-how-to-read-and-write-lnk-files

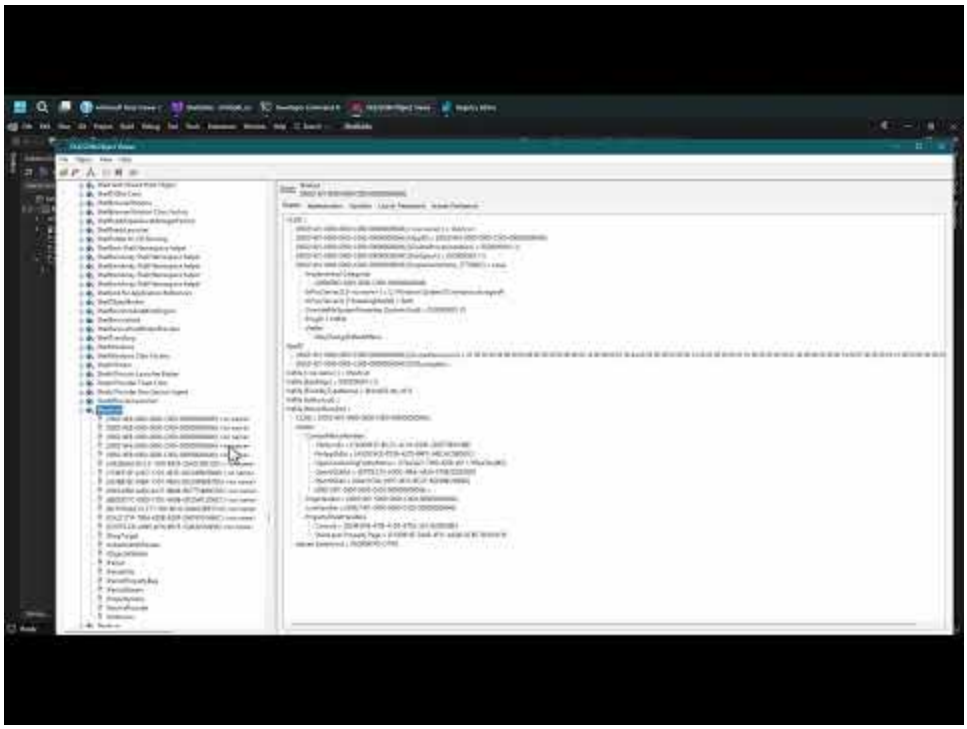
Pavel Yosifovich

May 10, 2026



Shell shortcuts are everywhere. Desktop icons, Start Menu entries, taskbar pins — they're all .lnk files under the hood. Most developers have interacted with them for years without thinking much about what's actually inside. And why would they? Explorer handles it all.

Until you need to create one programmatically. Or read one. Or modify one without launching Explorer. That's when you discover that shell links are implemented as COM objects.



Watch Video At: https://youtu.be/6Te8L-Yhe-M

Shell Links Are COM Objects, Not Text Files

The official name for a desktop shortcut is a **shell link**. The .lnk extension is how Explorer knows to treat the file specially, handing it off to the underlying COM implementation rather than opening it as a document.

These files are binary (not textual). If you try to open a .lnk in Notepad, what you get is the *target* — Explorer helpfully opens the file the shortcut points to, not the .lnk itself. Same thing with a hex editor like HxD: double-clicking hands you the PE, not the link file. You have to explicitly paste the .lnk path into the tool to get at the actual bytes. And when you do, it's clearly binary. You can spot the target path buried in there if you look — but there's no sane way to parse that manually.

So the right approach is programmatic, through COM.

The IShellLink Interface

The COM interface for working with shell links is `IShellLink` (or `IShellLinkW` for Unicode, which is what you want on any modern project). To get one, you use `CoCreateInstance` with **`CLSID_ShellLink`**:

```
#include <shlobj.h>    // IShellLink, CLSID_ShellLink
#include <atlbase.h>   // ATL smart pointers (CComPtr, CComQIPtr)

CComPtr<IShellLink> link;
HRESULT hr = link.CoCreateInstance(__uuidof(ShellLink)); // or CLSID_ShellLink
```

At this point you have a valid COM object — but an empty one. Calling `GetPath` on a freshly created shell link returns `S_FALSE`, which is one of those results that trips people up. It's not an error (the HRESULT value is non-negative), but here it means "I ran fine, there's just nothing here." The path buffer will have a null terminator as its first character. That's expected — the link is empty.

Loading an Existing .lnk File with IPersistFile

To read an existing shortcut — or to save one you've built or modified — you need a second interface: **`IPersistFile`**. This is a standard COM interface, and the shell link object implements it. You get to it through `QueryInterface`, which with ATL smart pointers looks like this:

```
CComQIPtr<IPersistFile> persist(link);
// persist is non-null if IPersistFile is supported (it is)
```

`IPersistFile` has two methods that matter here: `Load` and `Save`. Loading an existing shortcut:

```
hr = persist->Load(L"C:\\Users\\Pavel\\Desktop\\VLC.lnk", STGM_READ);
```

The second parameter is a storage mode constant from the **`STGM_`**`*` family — the same flags used by compound files, which is its own topic. For reading, `STGM_READ` is fine. For modifying and saving, you'll want `STGM_READWRITE`.

Once loaded, you can read any property the interface exposes:

```
WCHAR path[MAX_PATH];
hr = link->GetPath(path, MAX_PATH, nullptr, 0);
// path now contains the target, e.g. "C:\Program Files\VideoLAN\VLC\vlc.exe"
```

The `nullptr` for **`WIN32_FIND_DATA`** means you're not asking for file metadata — creation time, attributes, and so on. Pass a pointer if you want those details. The flags parameter at the end is mostly uninteresting; zero is good enough in most cases.

$2,111

$1,478 or $150 X 10 payments

Windows Master Developer

Takes you from a "generic" C programmer to a master Windows programmer in user mode and kernel mode.

Become Windows Master Developer

Setting Properties and Saving

`IShellLink` has a corresponding `Set*` methods for every property you can read:

- `SetPath` — the target file or folder
- `SetArguments` — command-line arguments passed to the target
- `SetDescription` — the tooltip description
- `SetHotkey` — keyboard shortcut to launch the link
- `SetIconLocation` — icon file path and index (doesn't have to come from the executable)
- `SetShowCmd` — how the window opens: normal, minimized, maximized

There are also advanced properties accessible through related interfaces — like setting the "run as administrator" flag, which is more interesting than it might sound (more on that below).

After making your changes, you save with:

```
hr = persist->Save(L"C:\\Users\\Pavel\\Desktop\\VLC.lnk", TRUE);
```

The second parameter controls whether this path becomes the current file path for the object. `TRUE` is the typical value.

Discovering Interfaces: OleView and the Registry

You might wonder how you'd know that `IPersistFile` is implemented by the same object as `IShellLink`. In general, for any COM class you're not familiar with, there's a tool for this: **OleView** (OLE/COM Object Viewer), included with a Visual Studio installation (and the Windows SDK).

Launch it from a Visual Studio developer command prompt, go to "All Objects" — which despite the name means all *registered COM classes* — and search for "Shortcut." Once you find the right entry, verify the CLSID against the one in your code (**`CLSID_ShellLink`** has lots of zeros — it's been hardcoded by Microsoft). Double-click to instantiate it. OleView calls `CoCreateInstance` and then queries the object for every interface registered in **`HKEY_CLASSES_ROOT\Interface`**. You'll see `IPersistFile` in the list. You'll also see `IShellLink` listed by its interface ID rather than a friendly name — because its registry entry doesn't include one. A minor nuisance, but you can verify it matches by pressing F12 on `IShellLinkW` in your code.

This brings up an interesting limitation of classic COM. The **`IUnknown`** interface — the base of everything — only gives you **`QueryInterface`**, **`AddRef`**, and **`Release`**. There is no way to ask a

COM object “what interfaces do you support?” You can only ask “do you support *this specific* interface?” OleView works around this by querying every interface registered in `HKCR\Interface` — a big list, but not necessarily complete. A class might implement interfaces that aren’t registered, and OleView will never find those.

The Windows Runtime solved this with **`IInspectable`**, which extends `IUnknown` and adds a `GetIids` method — exactly what you’d want. That was largely driven by the need to support JavaScript in UWP. In classic COM, we work with what we have.

Security Angle: LNK Persistence

Programmatic creation of .lnk files is a well-known persistence mechanism. An attacker with write access to a startup folder — user-level or system-level — can drop a shell link pointing to any target, with any arguments, and have it execute on the next login. The target doesn’t even have to exist at creation time; you can create a shell link pointing to a path that isn’t there yet, which is useful for staged payloads.

From a defensive perspective: any monitoring solution that tracks .lnk file creation or modification in startup locations should inspect not just the file name but the target, arguments, and working directory. A VLC shortcut that suddenly has a different target or unexpected arguments is worth flagging.

Key Takeaways

- Shell links (.lnk files) are implemented as COM objects. Work with them through `IShellLink`.
- `GetPath` returning `S_FALSE` on a fresh link is expected behavior — it means empty, not failure.
- `IPersistFile` is what you need to touch the disk. `IShellLink` alone doesn’t load or save anything.
- OleView is a good tool for exploring a COM class’s supported interfaces — but it can only find interfaces registered in `HKCR\Interface`.
- Classic COM has no `GetIids`. WinRT does, via `IInspectable`.

Keep Learning

If this made you want to go deeper on COM — how it actually works, how to implement your own COM objects, and why so much of Windows is built on it — the **COM Programming 1** course at TrainSec covers it from first principles: interfaces, ATL, DLL servers, and advanced COM techniques.

For working with the Windows API more broadly — processes, handles, kernel objects, and memory — **Windows System Programming 1** is the right starting point.

Both are part of the Windows Master Developer learning path on trainsec.net.

$2,111

$1,478 or $150 X 10 payments

Windows Master Developer

Takes you from a “generic” C programmer to a master Windows programmer in user mode and kernel mode.

Become Windows Master Developer