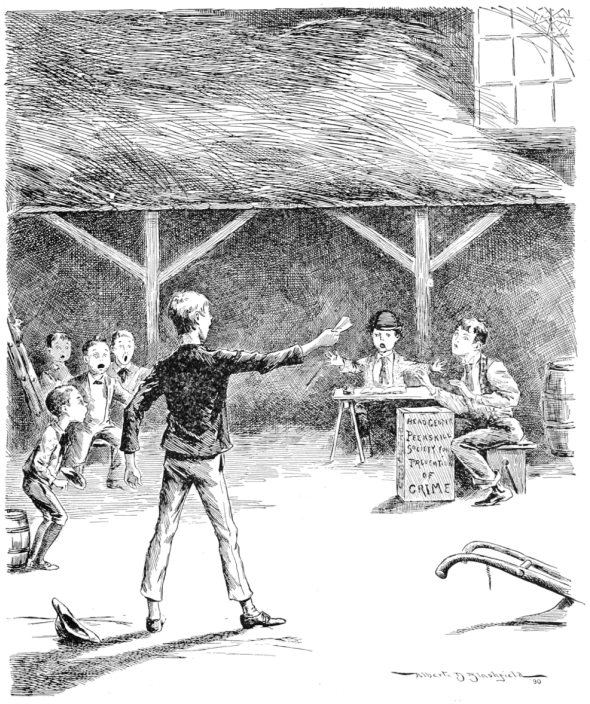
9 Where is ShimDBC.exe?

by Geoff Chappell

Microsoft's Shim Database Compiler might be a legend ... except that nobody seems ever to have made any story of it. It might be mythical ... except that it actually *does* exist. Indeed, it has been around for 15 years in more or less plain sight. Yet if you ask Google to search the Internet for occurrences of `shimdbc`, and especially of "`shimdbc.exe`" in quotes, you get either remarkably little or a tantalising hint, depending on your perspective.

Mostly, you get those scam sites that have prepared a page for seemingly every executable that has ever existed and can fix it for you if only you will please download their repair tool. But amongst this dross is a page from Microsoft's TechNet site. Google excerpts that "QFixApp uses the support utility ShimDBC.exe to test the group of selected fixes." Follow the link and you get to one of those relatively extensive pages that Microsoft sometimes writes to sketch a new feature for system administrators and advanced users (if not also to pat themselves on the back for the great new work). This page is from 2001 and is titled *Windows XP Application Compatibility Technologies*.[30]



[30] `https://technet.microsoft.com/library/bb457032.aspx`

9.1 Application Compatibility?

There can't be anything more boring in the whole of Windows, you may think. I certainly used to, and might still for applications if I cared enough, but Windows 8 brought *Application Compatibility* to kernel mode in a whole new way, and this I *do* care about.

The integrity of any kernel-mode driver that you or I write nowadays depends on what anyone else, well-meaning or not, can get into the `DRVMAIN.SDB` file in the `AppPatch` subdirectory of the Windows installation. This particular Shim Database file exists in earlier Windows versions too, but only to list drivers that the kernel is not to load. If you're the writer of a driver, there's nothing you can do at runtime about your driver being blocked from loading, and in some sense you're not even affected: you're not loaded and that's that. Starting with Windows 8, however, the `DRVMAIN.SDB` file defines the installed shim providers and either the registry or the file can associate your driver with one or more of these defined shim providers. When your driver gets loaded, the applicable shim providers get loaded too, if they are not already, and before long your driver's image in memory has been patched, both for how it calls out through its Import Address Table and how it gets called, *e.g.*, to handle I/O requests.

In this brave new world, is your driver really your driver? You might hope that Microsoft would at least give you the tools to find out, if only so that you can establish that a reported problem with your driver really is with your driver. After all, for the analogous shimming, patching, and whatever of applications, Microsoft has long provided an Application Compatibility Toolkit (ACT), recently re-branded as the Windows Assessment and Deployment Kit (ADK). The plausible thoroughness of this kit's Compatibility Administrator in presenting a tree view of the details is much of the reason that I, for one, regarded the topic as offering, at best, slim pickings for research. For the driver database, however, this kit does nothing—well, except to leave me thinking that the SDB file format and the API support through which SDB files get interpreted, created, and might be edited, are now questions I should want to answer for myself rather than imag-

ine they've already been answered well by whoever managed somehow to care about Application Compatibility all along.

9.2 The SDB File Format

Relax! I'm not taking you to the depths of Application Compatibility, not even just for what's specific to driver shims. Our topic here *is* reverse engineering. Now that you know what these SDB files are and why we might care to know what's in them, I expect that if you have no interest at all in Application Compatibility, you can treat this part of this article as using SDB files just as an example for some general concerns about how we present reverse-engineered file formats. (And please don't skip ahead, but I promise that the final part is pretty much nothing but ugly hackery.)

Let's work even more specifically with just one example of an SDB file, shown in Figure 15. It's a little long, despite being nearly minimal. It defines one driver shim but no drivers to which this shim is to be applied.

Although Microsoft *has not* documented the SDB file format, Microsoft *has* documented a selection of API functions that work with SDB files, which is in some ways preferable. Perhaps by looking at these functions researchers and reverse engineers have come to know at least something of the file format, as evidenced by various tools they have published which interpret SDB files one way or another, typically as XML.

As a rough summary, an SDB file has a 3-dword header, for a major version, minor version, and signature, and the rest of the file is a list of variable-size tags which each have three parts:

1. a 16-bit `TAG`, whose numerical value tells of the tag's type and purpose;

2. a size in bytes, which can be given explicitly as a dword or may be implied by the high 4 bits of the `TAG`;

3. and then that many bytes of data, whose interpretation depends on the `TAG`.

Importantly for the power of the file format, the data for some tags (the ones whose high 4 bits are 7) is itself a list of tags. From this summary and a few details about the recognised `TAG` values, the implied sizes and the general interpretation of the data, *e.g.*, as word, dword, binary, or Unicode string—all of which can be gleaned from Microsoft's admittedly terse documentation of those API functions—you might think to reorganise the raw dump so that it retains every byte but more conveniently shows the hierarchy of tags, each with their `TAG`, size (if explicit) and data (if present). A decoding of Figure 15 is shown in Figure 16.

To manually verify that everything in the file is exactly as it should be, there is perhaps no better representation to work from than one that retains every byte. In practice, though, you'll want some interpretation. Indeed, the dump above does this already for the tags whose high 4 bits are 6. The data for any such tag is a string reference, specifically the offset of a `0x8801` tag within the `0x7801` tag (at offset `0x0142` in this example), and an automated dump can save you a little trouble by showing the offset's conversion to the string. Since those numbers for tags soon become tedious, you may prefer to name them. The names that Microsoft uses in its programming are documented for the roughly 100 tags that were defined ten years ago (for Windows Vista). All tags, documented or not (and now running to 260), have friendly names that can be obtained from the API function `SdbTagToString`. If you haven't suspected all along that Microsoft prepares SDB files from XML input, then you'll likely take "tag" as a hint to represent an SDB file's tags as XML tags. And this, give or take, is where some of the dumping tools you can find on the Internet leave things, such as in Figure 17.

Notice already that choices are made about what to show and how. If you don't show the offset in bytes that each XML tag has as an SDB tag in the original SDB file, then you risk complicating your presentation of data, as with the string references, whose interpretation depends on those file offsets. But show the offsets and your XML quickly looks messy. Once your editorial choices go so far that you don't reproduce every byte but instead build more and more interpretation into the XML, why show every tag? Notably, the string table that's the data for tag `0x7801` (`TAG_STRINGTABLE`) and the indexes that are the data for tag `0x7802` (`TAG_INDEXES`) must be generated automatically from the data for tag `0x7001` (`TAG_DATABASE`) such that the last may be all you want to bother with. Observe that for any tag that has children, the subtags that don't have children come first, and perhaps you'll plumb for a different style of XML in which each tag that has no

```
00000000:  02 00 00 00 01 00 00 00-73 64 62 66 02 78 CA 00  ........sdbf.x..
00000010:  00 00 03 78 14 00 00 00-02 38 07 70 03 38 01 60  ...x.....8.p.8.‘
00000020:  16 40 01 00 00 00 01 98-00 00 00 00 03 78 0E 00  .@..........x..
00000030:  00 00 02 38 17 70 03 38-01 60 01 98 00 00 00 00  ...8.p.8.‘......
00000040:  03 78 0E 00 00 00 02 38-07 70 03 38 04 90 01 98  .x.....8.p.8....
00000050:  00 00 00 00 03 78 14 00-00 00 02 38 1C 70 03 38  .....x.....8.p.8
00000060:  01 60 16 40 02 00 00 00-01 98 00 00 00 00 03 78  .‘.@...........x
00000070:  14 00 00 00 02 38 1C 70-03 38 0B 60 16 40 02 00  .....8.p.8.‘.@..
00000080:  00 00 01 98 00 00 00 00-03 78 14 00 00 00 02 38  .........x.....8
00000090:  1A 70 03 38 01 60 16 40-02 00 00 00 01 98 00 00  .p.8.‘.@........
000000A0:  00 00 03 78 14 00 00 00-02 38 1A 70 03 38 0B 60  ...x.....8.p.8.‘
000000B0:  16 40 02 00 00 00 01 98-00 00 00 00 03 78 1A 00  .@...........x..
000000C0:  00 00 02 38 25 70 03 38-01 60 01 98 0C 00 00 00  ...8%p.8.‘......
000000D0:  00 00 52 45 4B 43 41 48-14 01 00 00 01 70 60 00  ..REKCAH.....p‘.
000000E0:  00 00 01 50 D8 C1 31 3C-70 10 D2 01 22 60 06 00  ...P..1<p..."‘..
000000F0:  00 00 01 60 1C 00 00 00-23 40 01 00 00 00 07 90  ...‘....#@......
00000100:  10 00 00 00 28 22 AB F9-12 33 73 4A B6 F9 93 6D  ....("...3sJ...m
00000110:  70 E1 12 EF 25 70 28 00-00 00 01 60 50 00 00 00  p...%p(....‘P...
00000120:  10 90 10 00 00 00 C8 E4-9C 91 69 D0 21 45 A5 45  ..........i.!E.E
00000130:  01 32 B0 63 94 ED 17 40-03 00 00 00 03 60 64 00  .2.c...@.....‘d.
00000140:  00 00 01 78 7A 00 00 00-01 88 10 00 00 00 32 00  ...xz.........2.
00000150:  2E 00 31 00 2E 00 30 00-2E 00 33 00 00 00 01 88  ..1...0...3.....
00000160:  2E 00 00 00 48 00 61 00-63 00 6B 00 65 00 64 00  ....H.a.c.k.e.d.
00000170:  20 00 44 00 72 00 69 00-76 00 65 00 72 00 20 00  .D.r.i.v.e.r. .
00000180:  44 00 61 00 74 00 61 00-62 00 61 00 73 00 65 00  D.a.t.a.b.a.s.e.
00000190:  00 00 01 88 0E 00 00 00-48 00 61 00 63 00 6B 00  ........H.a.c.k.
000001A0:  65 00 72 00 00 00 01 88-16 00 00 00 68 00 61 00  e.r.........h.a.
000001B0:  63 00 6B 00 65 00 72 00-2E 00 73 00 79 00 73 00  c.k.e.r...s.y.s.
000001C0:  00 00 ..
```

Figure 15. ShimDB File

child tags is represented as an attribute and value, *e.g.*,

```
<DATABASE
    TIME="0x01D210703C31C1D8"
    COMPILER_VERSION="2.1.0.3"
    NAME="Hacked Driver Database"
    OS_PLATFORM="0x00000001"
    DATABASE_ID="0x28 0x22 0xAB 0xF9 0x12 0x33
    0x73 0x4A 0xB6 0xF9 0x93 0x6D 0x70 0xE1 0
    x12 0xEF">
    <KSHIM
        NAME="Hacker"
        FIX_ID="0xC8 0xE4 0x9C 0x91 0x69 0xD0 0
    x21 0x45 0xA5 0x45 0x01 0x32 0xB0 0x63 0
    x94 0xED"
        FLAGS="0x00000003"
        MODULE="hacker.sys" />
</DATABASE>
```

Whether you choose XML in this style or to have every tag's data between opening and closing tags, there are any number of ways to represent the data for each tag. For instance, once you know that the binary data for tag `0x9007` (`TAG_DATABASE_ID`) or tag `0x9010` (`TAG_FIX_ID`) is always a `GUID`, you might more conveniently represent it in the usual string form. Instead of showing the data for tag `0x5001` (`TAG_TIME`) as a raw qword, why not show that you know it's a Windows `FILETIME` and present it as 16/09/2016 23:15:37.944? Or, on the grounds that it too must be generated automatically, you might decide not to show it at all!

If I labour the presentation, it's to make the point that what's produced by any number of dumping tools inevitably varies according to purpose and taste. Let's say a hundred researchers want a tool for the easy reading of SDB files. Yes, that's doubtful, but 100 is a good round number. Then ninety will try to crib code from someone else—because, you know, who wants to reinvent the wheel—and what you get from the others will each be different, possibly very different, not just for its output but especially for what the source code shows of the file format. Worse, because nine out of ten programmers don't bother much with commenting, even for a tool they may intend as showing off their coding skills, you may have to pick through the source code to extract the file format. That may be easier than reverse-engineering Microsoft's binaries that work with the file, but not necessarily by much—and not necessarily leaving you with the same confidence that what you've learnt about the file format is cor-

```
00000000:  Header: MajorVersion=0x00000002 MinorVersion=0x00000001 Magic=0x66626473
0000000C:  Tag=0x7802 Size=0x000000CA Data=
00000012:      Tag=0x7803 Size=0x00000014 Data=
00000018:          Tag=0x3802 Data=0x7007
0000001C:          Tag=0x3803 Data=0x6001
00000020:          Tag=0x4016 Data=0x00000001
00000026:          Tag=0x9801 Size=0x00000000
0000002C:      Tag=0x7803 Size=0x0000000E Data=
00000032:          Tag=0x3802 Data=0x7017
00000036:          Tag=0x3803 Data=0x6001
0000003A:          Tag=0x9801 Size=0x00000000
00000040:      Tag=0x7803 Size=0x0000000E Data=
     ⋮

000000BC:      Tag=0x7803 Size=0x0000001A Data=
000000C2:          Tag=0x3802 Data=0x7025
000000C6:          Tag=0x3803 Data=0x6001
000000CA:          Tag=0x9801 Size=0x0000000C Data=0x00 0x00 0x52 0x45 0x4B 0x43 0x41 0x48 0x14 0x01 0x00 0x00
000000DC:  Tag=0x7001 Size=0x00000060
000000E2:      Tag=0x5001 Data=0x01D210703C31C1D8
000000EC:      Tag=0x6022 Data=0x00000006 => L"2.1.0.3"
000000F2:      Tag=0x6001 Data=0x0000001C => L"Hacked Driver Database"
000000F8:      Tag=0x4023 Data=0x00000001
000000FE:      Tag=0x9007 Size=0x00000010 Data=0x28 0x22 0xAB 0xF9 0x12 0x33 0x73 0x4A 0xB6 0xF9 0x93 0x6D
                                                0x70 0xE1 0x12 0xEF
00000114:      Tag=0x7025 Size=0x00000028
0000011A:          Tag=0x6001 Data=0x00000050 => L"Hacker"
00000120:          Tag=0x9010 Size=0x00000010 Data=0xC8 0xE4 0x9C 0x91 0x69 0xD0 0x21 0x45 0xA5 0x45 0x01 0x32
                                                     0xB0 0x63 0x94 0xED
00000136:          Tag=0x4017 Data=0x00000003
0000013A:          Tag=0x6003 Data=0x00000064 => L"hacker.sys"
00000142:  Tag=0x7801 Size=0x0000007A Data=
00000148:      Tag=0x8801 Size=0x00000010 Data=L"2.1.0.3"
0000015E:      Tag=0x8801 Size=0x0000002E Data=L"Hacked Driver Database"
00000192:      Tag=0x8801 Size=0x0000000E Data=L"Hacker"
000001A6:      Tag=0x8801 Size=0x00000016 Data=L"hacker.sys"
```

Figure 16. ShimDB File (Decoded from Figure 15)

```
<INDEXES>
  <INDEX>
    <INDEX_TAG>0x7007</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_FLAGS>0x00000001</INDEX_FLAGS>
    <INDEX_BITS></INDEX_BITS>
  </INDEX>
  <INDEX>
    <INDEX_TAG>0x7017</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_BITS></INDEX_BITS>
  </INDEX>
  ...
  <INDEX>
    <INDEX_TAG>0x7025</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_BITS>0x00 0x00 0x52 0x45 0x4B 0x43 0x41 0x48 0x14 0x01 0x00 0x00</INDEX_BITS>
  </INDEX>
</INDEXES>
<DATABASE>
  <TIME>0x01D210703C31C1D8</TIME>
  <COMPILER_VERSION>0x00000006</COMPILER_VERSION>
  <NAME>0x0000001C</NAME>
  <OS_PLATFORM>0x00000001</OS_PLATFORM>
  <DATABASE_ID>0x28 0x22 0xAB 0xF9 0x12 0x33 0x73 0x4A 0xB6 0xF9 0x93 0x6D 0x70 0xE1 0x12 0xEF</
    DATABASE_ID>
  <KSHIM>
    <NAME>0x00000050</NAME>
    <FIX_ID>0xC8 0xE4 0x9C 0x91 0x69 0xD0 0x21 0x45 0xA5 0x45 0x01 0x32 0xB0 0x63 0x94 0xED</
    FIX_ID>
    <FLAGS>0x00000003</FLAGS>
    <MODULE>0x00000064</MODULE>
  </KSHIM>
</DATABASE>
<STRINGTABLE>
  <STRINGTABLE_ITEM>2.1.0.3</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>Hacked Driver Database</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>Hacker</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>hacker.sys</STRINGTABLE_ITEM>
</STRINGTABLE>
```

Figure 17. Illegible XML From a ShimDB Dumping Tool

rect and comprehensive. Writing a tool that dumps an undocumented file format may be more rewarding for you as a programmer but it is not nearly the same as documenting the file format.

9.3 Reversing XML to SDB

But is there really no definitive XML for representing SDB files? Of all the purposes that motivate anyone to work with SDB files closely enough to need to know the file format, one has special standing: Microsoft's creation of SDB files from XML input. If we had Microsoft's tool for that, then wouldn't most researchers plumb for reversing its work to recover the XML source? After all, most reverse engineers and certainly the popular reverse-engineering tools don't take binary code and unassemble it just to what you see in the debugger. No, they disassemble it into assembly language that can be edited and re-assembled. Many go further and try to decompile it into C or C++ that can be edited and re-compiled (even if it doesn't look remotely like anything you'd be pleased to have from a human programmer). In this context, the SDB to XML conversion to want is something you could feed to Microsoft's Shim Database Compiler for compilation back to SDB. Anything else is pseudo-code. It may be fine in its way for understanding the content, and some may prefer it to a raw dump interpreted with reference to documentation of the file format, but however widely it gets accepted it is nonetheless pseudo-code.

The existence of something that someone at Microsoft refers to as a Shim Database Compiler has been known for at least a decade because Microsoft's documentation of tag `0x6022` (`TAG_COMPILER_VERSION`), apparently contemporaneous with Windows Vista, describes this tag's data as the "Shim Database Compiler version." And what, then, is the `ShimDBC.exe` from the even older TechNet article if it's not this Shim Database Compiler?

But has anyone outside Microsoft ever seen this compiler? Dig out an installation disc for Windows XP from 2001, look in the Support Tools directory, install the ACT version 2.0 from its self-extracting executable, and perhaps install the Support Tools too in case that's what the TechNet article means by "support utility." For your troubles, which may include having to install Windows XP, you'll get the article's `QFixApp.exe`, and the Compatibility Administrator, as `CompatAdmin.exe`, and some other possibly useful or at least instructive tools such as `GrabMI.exe`, but you don't get any file named `ShimDBC.exe`. I suspect that `ShimDBC.exe` never has existed in public as any sort of self-standing utility or even as its own file. Even if it did once upon a time, we should want a modern version that knows the modern tags such as `0x7025` (`TAG_KSHIM`) for defining driver shims.

For some good news, look into either `QFixApp.exe` or `CompatAdmin.exe` using whatever is your tool of choice for inspecting executables. Inside each, not as resources but intermingled with the code and data, are several instances of ShimDBC as text. We've had Microsoft's Shim Database Compiler for 15 years since the release of Windows XP. All along, the code and data for the console program `ShimDBC.exe`, from its `wmain` function inwards, has been linked into the GUI programs `QFixApp.exe` and `CompatAdmin.exe` (of which only the latter survives to modern versions of the ACT). Each of the GUI programs has a `WinMain` function that's first to execute after the C Run-Time (CRT) initialisation. Whenever either of the GUI programs wants to create an SDB file, it composes the Unicode text of a command line for the fake `ShimDBC.exe` and calls a routine that first parses this into the `argc` and `argv` that are expected for a `wmain` function and which then simply calls the `wmain` function. Where the TechNet article says QFixApp *uses* `ShimDBC.exe`, it is correct, but it doesn't mean that QFixApp executes `ShimDBC.exe` as a separate program, more that QFixApp simulates such execution from the ShimDBC code and data that's built in.

Unfortunately, CompatAdmin does not provide, even in secret, for passing a command line of our choice through `WinMain` to `wmain`. But, c'mon, we're hackers. You'll already be ahead of me: we can patch the file. Make a copy of `CompatAdmin.exe` as `ShimDBC.exe`, and use your favourite debugger or disassembler to find three things:

- the program's `WinMain` function;
- the routine the program passes the fake command line to for parsing and for calling `wmain`;
- the address of the Import Address Table entry for calling the `GetCommandLineW` function.

Ideally, you might then assemble something like

```
call    dword ptr [__imp__GetCommandLineW@0]
mov     ecx,eax
call    SimulateShimDBCExecution
ret     10h
```

over the very start of `WinMain`. In practice, you have to allow for relocations. Our indirect call to `GetCommandLineW` will need a fixup if the program doesn't get loaded at its preferred address. Worse, if we overwrite any fixup sites in `WinMain`, then our code will get corrupted if fixups get applied. But these are small chores that are bread and butter for practised reverse engineers. For concreteness, I give the patch details for the 32-bit `CompatAdmin.exe` from the ACT version 6.1 for Windows 8.1 in Table 2.

For hardly any trouble, we get an executable that still contains all its GUI material (except for the 17 bytes we've changed) but never executes it and instead runs the console-application code with the command line that we give when running the patched program. Microsoft surely has `ShimDBC.exe` as a self-standing console application, but what we get from patching `CompatAdmin.exe` must be close to the next best thing, certainly for so little effort. It's still a GUI program, however, so to see what it writes to standard output we must explicitly give it a standard output. At a Command Prompt with administrative privilege, enter

```
shimdbc -?  >help.txt
```

to get the built-in ShimDBC program's mostly accurate description of its command-line syntax, including most of the recognised command-line options.

To produce the SDB file that is this article's example, write the following as a Unicode text file named `test.xml`:

```
<?xml version="1.0" encoding="UTF-16" ?>
<DATABASE NAME="Hacked Driver Database"
 ID="{F9AB2228-3312-4A73-B6F9-936D70E112EF}">
  <LIBRARY>
    <KSHIM NAME="Hacker" FILE="hacker.sys"
     ID="{919CE4C8-D069-4521-A545-0132B06394ED}
     "
     LOGO="YES" ONDEMAND="YES" />
  </LIBRARY>
</DATABASE>
```

and feed it to the compiler via the command line

```
shimdbc Driver test.xml test.sdb >test.txt
```

I may be alone in this, but if you're going to tell me that I should know that you know the SDB file format when all you have to show is a tool that converts SDB to XML, then this would better be the XML that your tool produces from this article's example of an SDB file. Otherwise, as far as I'm concerned for studying any SDB file, I'm better off with a raw dump in combination with actual documentation of the file format.

Do not let it go unnoticed, though, that the XML that works for Microsoft's ShimDBC needs attributes that differ from the programmatic names that Microsoft has documented for the tags or the friendly names that can be obtained from the `SdbTagToString` function. For instance, the `0x6003` tag (`TAG_MODULE`) is compiled from an attribute named not `MODULE` but `FILE`. The `0x4017` tag (`TAG_FLAGS`) is synthesised from two attributes. Even harder to have guessed is that a `LIBRARY` tag is needed in the XML but does not show at all in the SDB file, *i.e.*, as a tag `0x7002` (`TAG_LIBRARY`). So, to know what XML is acceptable to Microsoft's compiler for creating an SDB file, you'll have to reverse-engineer the compiler or do a lot of inspired guesswork.

Happy hunting!

File Offset	Original	Patched	Remarks
0x0002FB54	8B FF	EB 08	jump to instruction that will use existing fixup site
0x0002FB56	55		
0x0002FB57	8B EC		
0x0002FB59	81 EC 88 05 00 00		
0x0002FB5E		FF 15 D0 30 49 00	incorporate existing fixup site at file offset 0x0002FB60
0x0002FB5F	A1 00 60 48 00		
0x0002FB64	33 C5	8B C8	
0x0002FB66	89 45 FC	E8 55 87 01 00	no fixup required for this direct call within `.text` section
0x0002FB69	8B 45 08		
0x0002FB6B		C2 10 00	
0x0002FB6C	53		
0x0002FB6D	56		

Table 2. Patch details for the 32-bit `CompatAdmin.exe` from the ACT version 6.1 for Windows 8.1.