Practical Windows Code and Driver Signing

Code and driver signing for Microsoft Windows 10, 8.1, 8, 7, Vista, and XP

Introduction

If you have ever installed some software or drivers in Windows, you have probably seen a dialog telling you the name of the company or person that published that software. This means that the publisher has cryptographically **signed** their work. Signing your software is important: by showing a nicer dialog to the end user, it gives end users more confidence that they are not installing malware. In the case of device drivers, signing is even *required* by certain versions of Windows in certain situations.

If you are a developer figuring out how to sign drivers or software, the aim of this guide is to tell you everything you need to know so that you can do it correctly.

My name is David Grayson and I work at Pololu Robotics & Electronics. In 2012 I went through the process of signing all of our company's USB drivers and most of our installers for Windows. I encountered so many problems along the way that could have been easily avoided if someone had told me about them ahead of time. If you are going through the same process, I sincerely hope that this document can clear up all of your confusion and **save you a lot of time**. I learned the hard way and now you can learn the easy way.

A lot of this information can be verified in official Microsoft documentation found on MSDN, and I will try to cite the official documentation when needed. The authoritative documents on kernel-mode code signing are kmsigning.doc and KMCS_walkthrough.doc. These are pretty good resources, but they are from 2007 and thus contain no information about Windows 7 and up, SHA-2, or the Windows Hardware Developer Center Dashboard portal. Also, their scope is more limited than the scope of this document because they don't talk about signing executables. Microsoft also announces changes to its code/driver signing requirements via MSDN blog posts (see the references section) but they do not have any updated documentation that gives you the full picture. Therefore, a lot of the things I say here are actually conclusions that I have drawn from my own experiments. When I am telling you something that I determined experimentally, I will use phrases like "it seems like" or "in my experience". When my experiments contradict the official documentation I will say so.

If you think any of the information I am providing here is wrong, please post a comment and let me know so we can figure it out.

This document only covers Windows XP 32-bit, Windows Vista, Windows 7, and Windows 8, Windows 8.1, and Windows 10.

The most useful part of this document is the signature requirements section.

This document was originally published in January 2013 and described many problems I had with certificates that use the SHA-2 hashing algorithm. Because of all these problems, I used to recommend sticking to SHA-1. Since then, Microsoft has announced that in the long-term, they intend to distrust SHA-1 throughout Windows in all contexts. Therefore, SHA-1 will not be a long term solution, and most people should probably use SHA-2 instead. In July 2015, I did a systematic set of experiments with different types of signatures. Using the data from those experiments, I have updated this document to better cover SHA-2 and the recent updates from Microsoft that allow it to be a viable option. Since then, I have been keeping an eye on new developments and updating this article.

Public key cryptography

Pretty much every secure thing you do with a computer, including code and driver signing, uses some form of public key cryptography. There are several different public key cryptography systems, including RSA (which is used for Windows code signing) and ECDSA. I am not going to really explain the mathematics behind any of these systems, but I will give you an idea of what they let us do. This will help you understand what a digital signature actually does.

The first thing that a public key cryptography system gives us is a way to generate a *key pair*, which consists of a *public key* and a *private key*. As the names suggest, the *private key* must be kept secret, but you can give the *public key* to anyone.

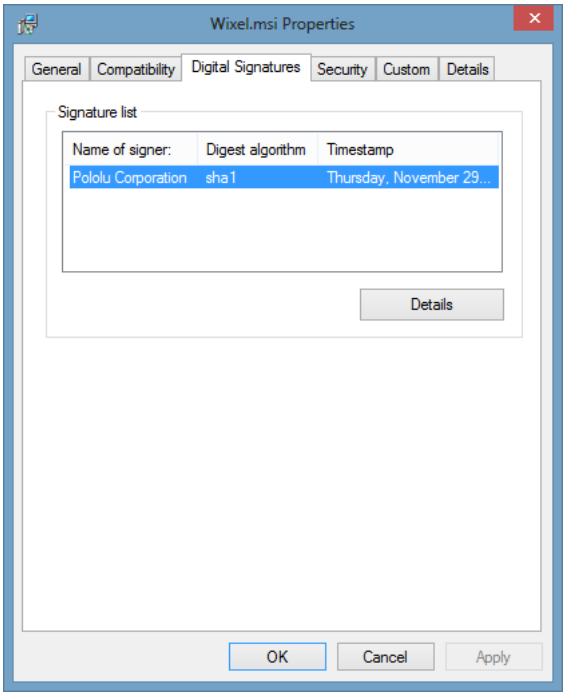
Many public key cryptography systems support the creation of digital signatures. Given a digital message that you want to sign and a private key, you can run a specific algorithm to generate a *digital signature*, a series of bytes that proves properties about the message. More specifically, anyone in the world can use your public key along with the message and its signature to verify that the signature was generated by someone who had access to your private key (hopefully you), and anyone in the world can verify that the signed message has not been modified since it was signed.

Another important concept to understand is the **hash function**, which is also called a digest algorithm or thumbprint algorithm. A hash function is a way to transform some sequence of bytes into a smaller sequence of bytes, usually with a fixed length, with the property that it is very hard to make two inputs to the hash function that give the same output. SHA-1 is a widely-used hash function but it is considered to be deprecated because of theoretical and practical attacks against it. SHA-2 is a newer family of hash functions, consisting of SHA-224, SHA-256, SHA-386, and SHA-512. Hash functions work well with signatures because it is more efficient to sign a hash of a file than to sign the entire contents of the file.

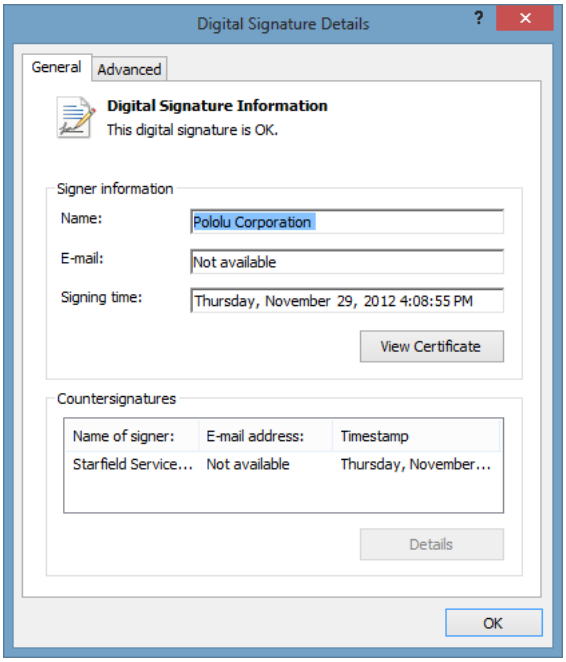
Anatomy of a signature

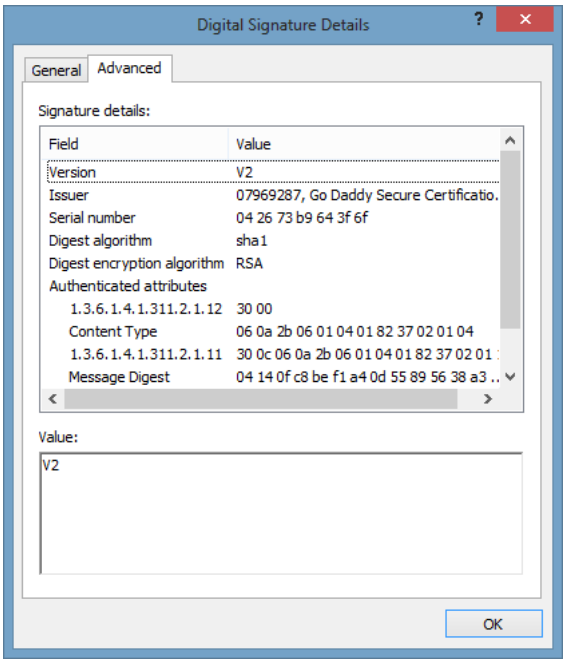
Windows has a series of dialog boxes that allow you to view the details about a signature embedded in a file. It is important that you know your way around these dialogs because they will help you understand the nature of the signature you are applying to your software.

If you right-click on a signed file and go to Properties, you will see a Digital Signatures tab.

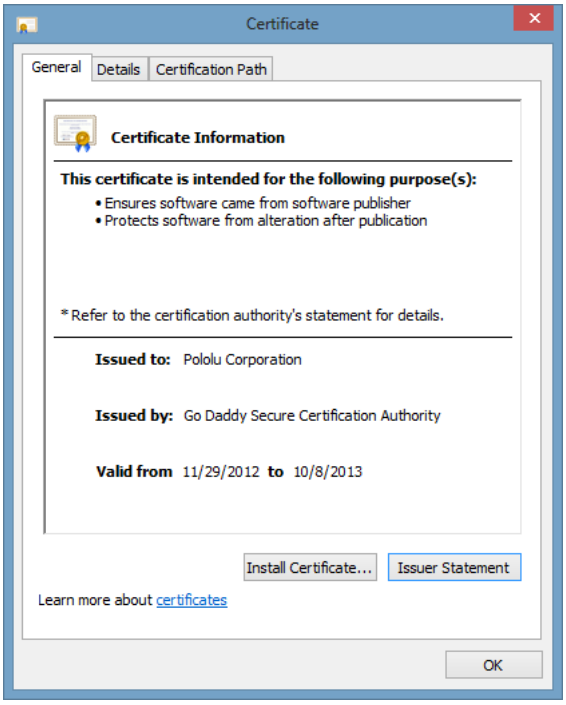


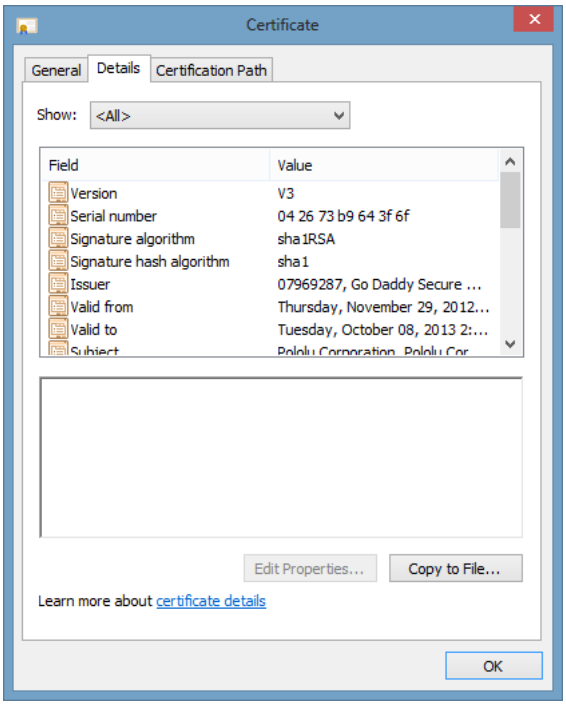
In the Digital Signatures tab, you can click on Details to open the Digital Signature Details dialog box. The digital signature is created by the publisher of the software.



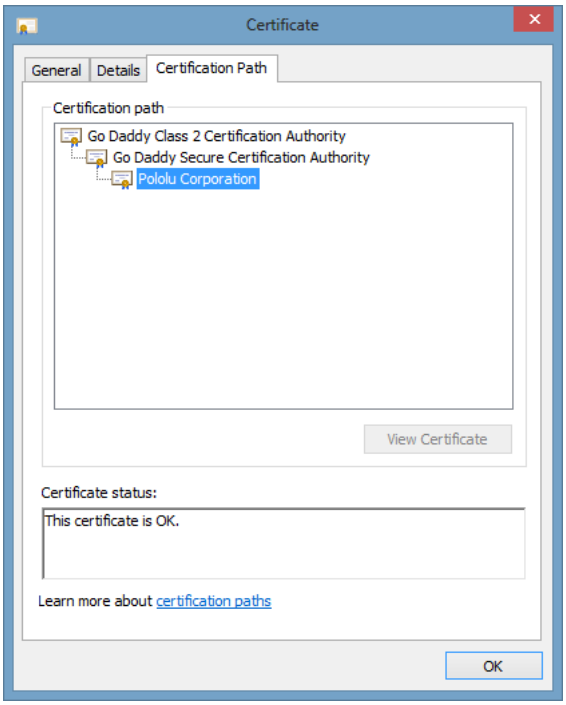


You can click on View Certificate to view the certificate that is embedded in the file's signature. The certificate is purchased from a certification authority such as Verisign.





You can click on Certification Path to view most of the certificates in the chain of trust. The point of these certificates is to prove that your certificate was issued by some trustable company. You can double-click on any certificate visible in the certification path to get information about it.



Some of the certificates shown in the certification path come from the file whose signature your are inspecting. Other certificates might come from your computer's certificate store, which you can see by running certmgr.msc. In particular, Windows seems to use certificates from the Intermediate Certification Authorities list and the Trusted Root Certification Authorities list to build the certification path. Unfortunately, I do not know of a good way to look at a signed file and tell exactly what certificates are embedded in it.

The names shown in the Certification Path are the "Friendly Names" of the certificates, which you can configure in certmgr.msc. Sometimes, multiple certificates might have the same friendly name, which makes it confusing to see what is going on. To clear up the confusion, I like to double-click on the certificates and look at the "Subject Key Identifier" and "Authority Key Identifier". I think that the Subject Key Identifier represents an entity you might trust, and every certificate simply represents a transfer of trust from some authority to the subject. If the authority and subject identifiers are the same, that is called a "self-signed" or "root" certificate.

The same subject can be found in multiple different certificates. For example, the "GlobalSign Root CA - R3" (subject key identifier 8f f0 4b 7f a8 2e 45 24 ae 4d 50 fa 63 9a 8b de e2 dd 1b bc) has its own root certificate as well as a cross-certificate issued by the GlobalSign Root CA, which seems to be an older and better supported authority.

Some day I might expand this section to include details about the different fields you can see in these dialogs, and why those pieces of information are necessary. For now, you should click around yourself and explore some signatures.

You can also use a hex editor such as WinHex to examine the embedded signatures; you can easily see the names of the signer and the organizations in the certification path. One great feature of WinHex is that it lets you compare two files and highlights the differences in them, so you can see exactly which bytes in the header are

modified and which bytes are appended to the end when the signature is added.

Signature requirements

To successfully release your software, you should make sure that your digital signature meets all the necessary requirements documented below. The requirements are summarized in the tables below, and then **the terms in the tables are defined and explained after the tables**. Each data cell of each table contains a boolean expression that combines different requirements using and (&) and or operations. I strongly suspect that this list is incomplete, so please post a comment if there is anything to add to it.

If you just care about your software working and don't mind if the user sees a scary warning message, these are the requirements your signature needs to meet in my experience:

Signature requirements for it to just work

	Running an executable	Installing a driver package	Loading a kernel module
on Windows XP	none	none	none
on Windows Vista 32-bit	Unsigned or SHA−1 or KB2763674	none	none
on Windows Vista 64-bit		none	MCVR & /t & SHA-1
on Windows 7	none	none	MCVR & (SHA−1 or KB3033929) & SHA-1 phase-out
on Windows 8, 8.1	none	TRCA & SHA-1 phase-out	MCVR & SHA-1 phase-out
on Windows 10	none	TRCA & SHA-1 phase-out	MCVR & SHA-1 phase-out & Portal

If you care about your software working AND you want to show up as the verified publisher in any warning dialog boxes and any Properties dialogs, these are the requirements your signature needs to meet in my experience:

Signature requirements for it to look good

	Running an executable	Installing a driver package	Loading a kernel module
on Windows XP	TRCA & /t & SHA-1 *sig*	WHQL	?
on Windows Vista 32-bit	TRCA & /t & SHA-1 *sig*	TRCA & /t	?
on Windows Vista 64-bit	& (SHA−1 or KB2763674)	TRCA & /t	?
on Windows 7	TRCA & SHA-1 phase-out	TRCA & SHA-1 phase-out	?
on Windows 8, 8.1	TRCA & SHA-1 phase-out	TRCA & SHA-1 phase-out	?
on Windows 10	TRCA & SHA-1 phase-out	TRCA & SHA-1 phase-out	?

SHA-1 phase-out

> The article Windows Enforcement of SHA1 Certificates from Microsoft describes how SHA-1 will eventually be distrusted in Windows in all contexts. To avoid future problems, it is best to start using SHA-2 (or higher) for everything, including the file digest, main certificate, timestamp digest, and timestamp certificate.

Portal

> Microsoft announced on 2015-04-01 that all new Windows 10 kernel mode drivers must be submitted to and digitally signed by the Windows Hardware Developer Center Dashboard portal. For backwards compatilibity, Windows 10 will still allow kernel mode drivers with signatures from older certificates under certain conditions, but you would need to have an older certificate so it is not very practical to take advantage of that for most people. The portal only accepts driver submissions from you if you sign them with an Extended Validation (EV) certificate, which is typically more expensive than a normal certificate.
>
> On 2016-07-26, Microsoft announced that this rule will only be enforced on Windows 10 systems that were freshly installed at build 1607 or later, with Secure Boot on.
>
> I used to think there was a really nice loophole in the original announcement that would allow most people to avoid the hassle of using the portal, but the wording in the FAQ of the 2016-07-26 annoucement makes it clear that there is no such loophole.
>
> For more information about the portal, see the Loading a kernel module section.

TRCA

> In the tables above, TRCA means the signature's chain of trust must go back to a certificate in the user's Trusted Root Certification Authorities (TRCA) list. As you can see in certmgr.msc, the TRCA list has certificates from several well-known companies such as a Verisign, Globalsign, Digicert, and Go Daddy. Many certificates are not present in the list initially, but Window will attempt to automatically install them from the various sources when they are needed to verify a signature.
>
> One way Windows can download root certificates is by connecting to Windows Update using the Internet. This is called the Microsoft Root Cerificate Program. However, I would not rely on the auto-update. Logically, it shouldn't work if the computer is disconnected from the internet. In my experience, even with an internet connection it does not always work reliably.
>
> Windows can also get the certificates it needs from crypt32.dll, and I think that method is much faster and more reliable.
>
> Therefore, your best bet is to make sure your chain of trust goes back to a certificate that is included in fresh installs of Windows, either in the TRCA or in crypt32.dll. Unfortunately, I don't have an authoritative list of those certificates. The TRCA requirement is documented in kmsigning.doc. I suspect that the "Trusted Publishers" or "Trusted People" lists would work just as well, if you convince your users to install a certificate there. You typically don't need a cross-certificate (specified with the `/ac` option to signtool) to meet this requirement. However, an intermediate certificate (which gets automatically used during signing if it is installed in your certificate store) could help by extending your chain of trust back to an older and better supported certificate.

MCVR

> In the tables above, MCVR means the signature's chain of trust must go back to the Microsoft Code Verification Root certificate, or some other certificate that is trusted by the kernel. Any signature that you get through the WHQL process should already satisfy this requirement. This is documented very clearly in kmsigning.doc, which explains that the kernel does not have access to the Trusted Root Certification Authorities list. A cross-certificate is typically needed to satisfy this requirement. Microsoft publishes a complete list of the Cross-Certificates for Kernel Mode Code Signing.

WHQL

> The signature must come from the WHQL program. My understanding is that you can submit your driver to Microsoft or some third party to be tested. If your driver is OK, they will sign your driver and give you legal permission to use the Windows Logo to sell your product. WHQL is **never actually required** for your software or drivers to work and probably harder than just using a standard code signing certificate. I have never gotten a driver WHQL-signed, so my experience with it is limited.

Unsigned

> This requirement is true if the file simply has no signature. Keep in mind that the table above uses boolean expressions, so when I write "X or Y or Z" it means that if any of those three are true, then your signature will work.

SHA-1

> A signature must be present and it must not use SHA-2 in any way, only SHA-1. This applies to the signature of the file itself and the signatures that secure the chain of trust to your certificate. This probably also applies to the timestamp and its chain of trust. Note that Microsoft is retiring SHA-1 and will eventually distrust it throughout Windows in all contexts, so sticking to SHA-1 will not be a long term solution.

SHA-1 *sig*

> The *signature* must be made using SHA-1 as the digest algorithm, but it is OK if parts of the certificate's chain of trust use SHA-2. This might also apply to digest algorithm used by the timestamp.

/t

> In the tables above, /t means that the signature should be timestamped using the `/t` option of signtool instead of `/tr`.

KB2763674

> In the tables above, KB2763674 means that KB2763674 must be installed, which in turn requires Windows Vista SP2.

KB3033929

> In the tables above, KB3033929 means that KB3033929 must be installed.

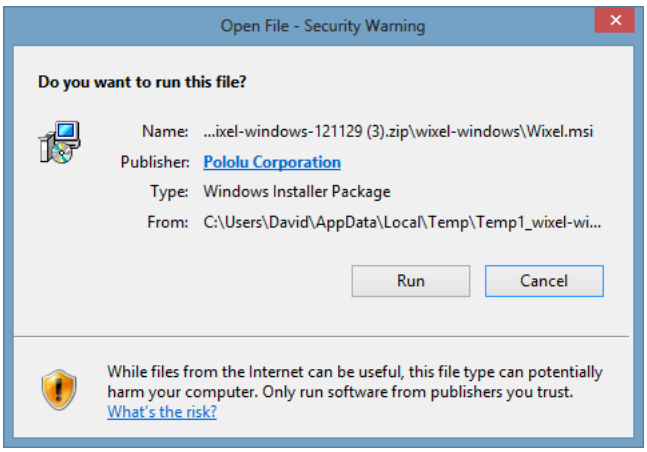
Don't use spaces in the INF file name. This is an additional requirement for driver package installation that was reported by Jimmy Kaz. I have not tested it myself, but he says that the driver package will appear to be unsigned in Windows 7 if the INF file has spaces in the name.

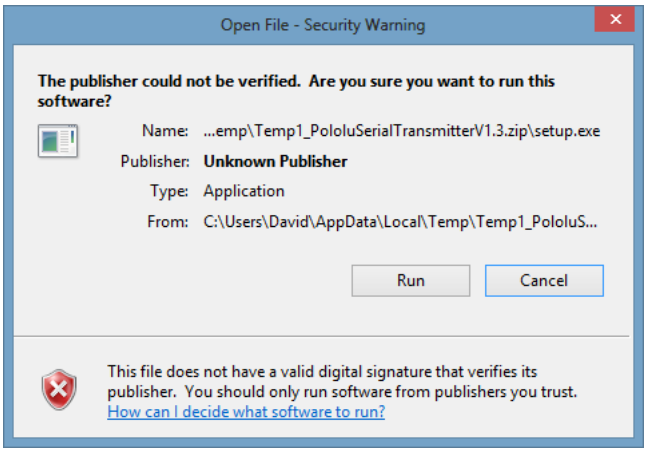
The content above is a concise summary of all the code and driver-signing requirements I know about. In the next three sections, I will explain each of the requirements and what you can expect if your software does not meet them.

Running an executable

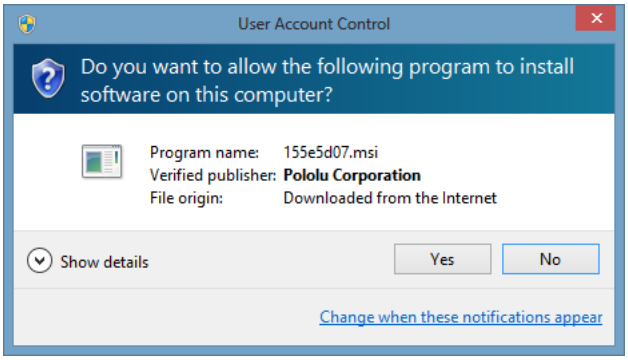
Digital signatures can be directly embedded inside executables (EXE files). They can also be embedded inside Windows Installer files (MSI files), and everything that I will say about executables also applies to Windows Installer files. Windows verifies the signature inside an executable file in two situations:

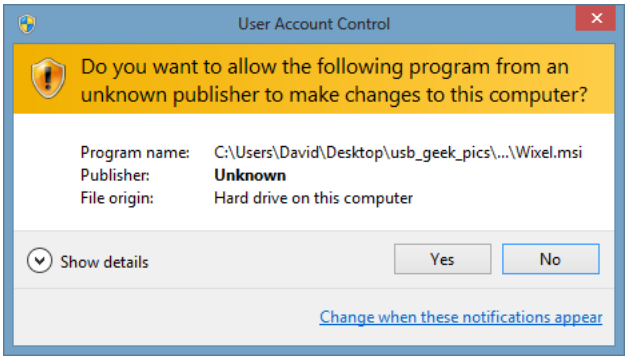
- If the file was downloaded from the internet (including network drives), Windows will show a "Open File - Security Warning" message when the user tries to run the file. The publisher information in the warning comes from the signature embedded in the file. Usually the warning is a simple dialog box, but in Windows 8 and later the warning is sometimes a SmartScreen dialog that takes over the whole screen. Two examples are shown below:





- If the executable requests administrator privileges, which is also known as elevating, Windows will display a UAC prompt. The publisher information in the prompt comes from the signature embedded in the file. Two examples are shown below:





Windows has never required signatures on executables. However, it is nice to sign your executable so that whenever the user sees a warning message about it, they will see your name as the publisher instead of being told that the publisher is unknown.

In my experience, in order for your signature to work properly on an executable, it should have a chain of trust that goes back to a certificate in the user's Trusted Root Certification Authorities list, which you can see by running certmgr.msc.

Use SHA-1 to sign an executable if Windows Vista matters

Even if your certificate is signed with SHA-2 and has SHA-2 certificates in its chain of trust, you have a choice about what digest to use when making a signature. To choose the algorithm, you should pass either `/fd sha1`, `/fd sha256`, or `/fd sha512` to signtool.

If you want your signature to look correct in Windows Vista, you will have to use SHA-1 as the digest algorithm when signing an executable. In my experience, SHA-2 signatures will be deemed invalid on Windows Vista, and Windows Vista will say "This digital signature is not valid." when you view the signature details. I suspect that Windows XP behaves the same way, but I have not tested it.

Be sure to check the "Digest algorithm" of the signature in its properties page to make sure your signature uses the desired algorithm.

Use /t to timestamp an executable if Windows Vista matters

When signing with signtool, you have a choice about whether to specify the timestamp server using the `/t` option or the `/tr` option. If you specify it with `/t`, signtool gets a timestamp from the server using a custom Microsoft protocol. If you specify it with `/tr`, signtool gets a timestamp from the server using RFC3161. But these aren't just different protocols, they also seem to affect something about the timestamp itself.

I have found through experimentation that timestamps made with `/tr` are not recognized on Windows Vista, for either executables or drivers. If you open the properties for your signature in Windows Vista, you will see that there is no timestamp listed. I suspect that Windows XP behaves the same way, but I have not tested it, but someone else has.

When I first wrote this document in 2013, I was convinced that you should use `/tr`. I had done an experiment where I made both types of timestamps, and I found that Internet Explorer 10.0.9200.16686 on Windows 7 64-bit SP1 and Internet Explorer 10.0.9200.16688 on Windows 8 64-bit flagged the `/t` executable as being corrupt or invalid. I was able to reproduce these results in 2015 if I used the exact same file and browser, but I was not able to reproduce them using IE11 or with a newly-signed file on IE10. The results I got earlier might be explained by a subtle bug in the Starfield timestamp server's implementation of `/t`, which for some reason was only detected by IE 10.

Be aware of KB2763674 for Windows Vista SP2

If your certificate uses SHA-2 or has SHA-2 certificates in its chain of trust, then you should be aware of KB2763674, an update for Windows Vista SP2 distributed through Windows Update. On versions of Windows Vista without this update, when the end user double-clicks on a *downloaded* executable with a signature whose chain of trust uses SHA-2, nothing happens! There is no error message or activity of any kind.

I believe that there is some code in Windows Vista that checks the signature of the executable in order to show the publisher in the warning dialog for downloaded executables. Before the update, that code apparently could not handle SHA-2, and would silently exit.

There are several ways to get around this problem. Since the number of people using Windows Vista is pretty small these days, you can simply put a note in your documentation that tells Windows Vista users to make sure they have that update installed.

One workaround that the user can do is to run the executable from the Command Prompt, thereby bypassing the warning dialog and the signature checking that goes along with it.

Another workaround for the user to do would be to remove the special flag in the file system that marks the file as coming from the internet. I have not tested that but I expect it to work. Alternatively, you could distribute the executable unsigned.

It seems like this problem doesn't affect installers created with NSIS, and I think I know why. NSIS installers are always pre-emptively elevated by Windows, so when you run an NSIS installer you will always see the standard UAC warning and never see the special warning that results from running a downloaded executable. In that way, the buggy code in Windows Vista is bypassed.

Installing a driver package

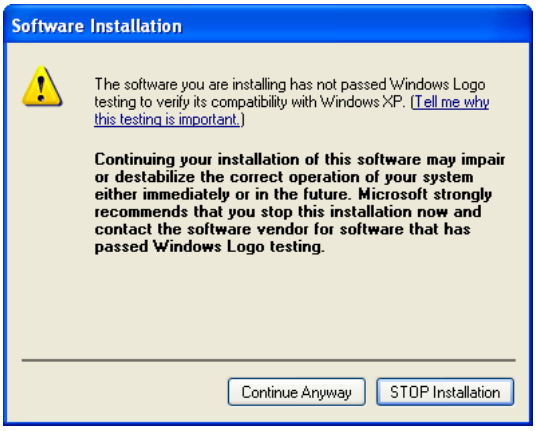
A driver package consists of a single INF file and the files that it references. You can have multiple INF files in the same directory, but in my experience Windows treats each INF file as a separate and independent driver package. A driver package can be signed by first generating a security catalog (CAT) file with cryptographic hashes of all the files, and then embedding a signature in the security catalog. The security catalog contains a list of file names and a hash of the contents of each file; you can simply double-click on it to inspect the information it contains and see its signature.

There are at least five ways to install a driver package. First, the user can right-click on the INF file and select "Install" if the INF file has a DefaultInstall section. (Actually, this method seems to work in Windows 8 and above even if the INF files does not have a DefaultInstall section.) Second, a program can call the SetupCopyOEMInf function. Third, a program or user can run PnPUtil. Fourth, a user can right-click on a matching device in the Device Manager, select "Update driver software...", and then tell Windows the directory where the driver package is stored. Fifth, the driver package can be shipped with DPInst executables that install it.

When the driver package installation is initiated, Windows will check for a signature and behave differently depending on what it finds; different versions of Windows behave differently.

Driver package installation in Windows XP

Back in Windows XP, it seems that the only kind of driver package signature Microsoft cared about was a WHQL signature. You can have a driver package that displays a nice install prompt in Windows Vista and up that indicates who the publisher is, but installing the same driver package in Windows XP results in the following warning:



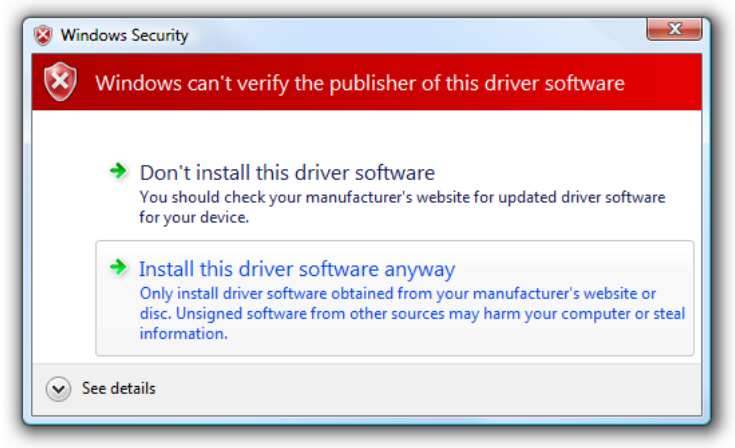
Driver package installation in Windows Vista and 7

Starting in Windows Vista, Microsoft changed their tune. Instead of warning users about whether or not the drivers have passed WHQL testing, Windows Vista and 7 warn the user about whether the publisher is verified or unverified. To show up as the verified publisher, you need to provide a CAT file with a proper signature. The requirements are documented in kmsigning.doc and in the signature requirements section above.

If you sign your driver package properly, users will see a friendly prompt when they install it in Windows Vista, 7, or 8:

The name in the prompt comes from from the INF file's DriverPackageDisplayName directive and the publisher comes from the verified signature on the CAT file.

However, if you don't sign your driver package, users will see a big red warning when they install it in Windows Vista or 7:



I think it was a good idea for Microsoft to make that change. WHQL testing is inflexible. If you change one byte of your driver, you would have to re-submit it to be tested again. Regular code signing is easier and cheaper: you can get a certificate for a couple hundred dollars per year that lets you sign as many driver packages as you want. This probably resulted in more companies making signed drivers, so the malware stood out more.

Driver package installation in Windows 8 and above

Starting in Windows 8, all driver packages have to be signed. Unfortunately, I have not seen any official document from Microsoft about this change, even though I asked about it on StackOverflow.

If you try to install an unsigned driver package which previously worked on older versions of Windows, you will get a simple error message:

This wasn't necessarily a bad decision on Microsoft's part, but it was bad news for a lot of small companies such as ours and individuals making USB devices on a small scale. For years, we had successfully distributed unsigned driver packages that worked fine in Windows XP, Vista, and 7 because they relied on kernel modules (SYS files) that are signed by Microsoft, in particular WinUSB and usbser.sys. Starting with Windows 8, we had to figure out the driver signing process or tell our Windows 8 customers to follow the complicated procedure for disabling driver signature enforcement.

If you are new to the industry and want to start making USB devices, the vendor ID from the USB-IF will cost you $5000 and the code signing certificate will probably cost a few hundred dollars per year. However, your signatures should keep working after the certificate expires if you make sure to use a timestamp when signing.

The friendly driver installation prompt for signed driver packages in Windows 8 looks pretty much the same as it did in Windows Vista and 7.

Loading a kernel module

Some driver packages contain kernel-mode code (SYS files) that need to get loaded into the kernel at some point, typically when a matching device is plugged into the computer.

Starting with Windows Vista 64-bit, kernel modules must come with a properly-signed security catalog (CAT file) or else they cannot be loaded into the kernel. In July 2007, six months after the release of Windows Vista, Microsoft published two documents about the new signing requirements: kmsigning.doc and KMCS_walkthrough.doc.

If your driver only uses WinUSB or usbser.sys, all you need to worry about is getting your driver package installed, as described in the Installing a driver package section. The kernel modules you are using have already been signed by Microsoft and you will have no trouble getting them loaded into the kernel after the driver package is installed.

The signatures for kernel-mode code are typically kept in the security catalog (CAT file) for the driver package, but in the case of a boot-start driver you are supposed to embed the signature inside the SYS file itself, according to kmsigning.doc.

Prior to Windows 10 1607 (Anniversary Update), you could use a cross-certificate to sign your CAT file and produce a signature that convinces Windows to load your SYS file into the kernel. The main requirement for that signature was just that the signature's chain of trust must go back to the Microsoft Code Verification Root certificate, or some other certificate that is trusted by the kernel. This is documented very clearly in kmsigning.doc, which explains that the kernel does not have access to the Trusted Root Certification Authorities list. Microsoft publishes a complete list of the Cross-Certificates for Kernel Mode Code Signing.

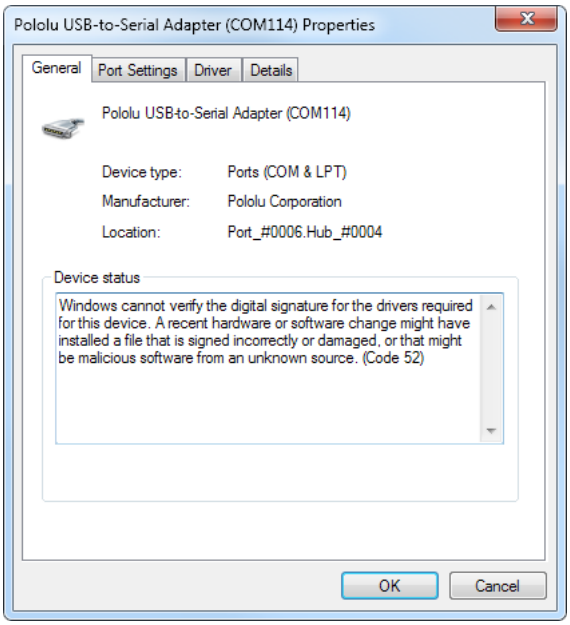
However, there are additional requirements for kernel module signatures in Windows 10 1607 and later. See the information about the portal below.

It is important to note that a given signature might be good enough to get a driver package installed, but not good enough to get the kernel modules (SYS files) loaded into the kernel. The part of Windows that checks signatures for driver package installation is apparently different from the part that checks signatures for loading kernel modules, and they each impose different requirements on the signature.

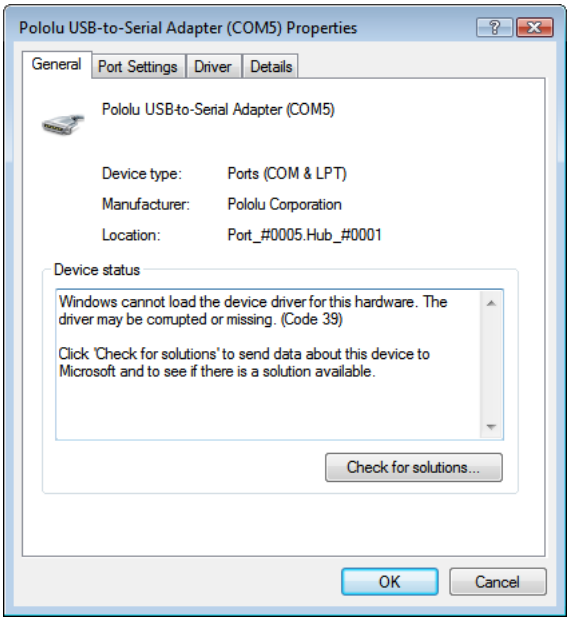
One important detail is that the signature can come from any security catalog installed on the system; the signature does not actually have to be in the security catalog for the INF file that is matching the device. In other words, if Corporation X makes a kernel module and properly signs it for Windows Vista/7, Corporation Y can make a driver package for their own product that uses the kernel module and Corporation Y does not have to provide any kind of signature for it to work in Windows Vista/7. This is not particularly surprising if you think about it: the dangers of loading code into the kernel depend only the code itself, not the device or INF file it is being used with. In fact, this detail is what allows our CP2102 USB-to-Serial Bridge Driver to work on Windows XP/Vista/7/8. The signature that I put on our catalog file (plluvcp.cat) does not meet all the requirements to get silabser.sys loaded into the kernel, but the signature that Silicon Laboratories put on their catalog file (slabvcp.cat) does, and that's all that matters.

When it is time load a SYS file into the kernel, it seems that Windows will scan all the files in the security catalog store (*C:\Windows\System32\catroot*) to see if one of them contains a hash for the SYS file and an adequate signature. If it finds what it is looking for, the loading succeeds.

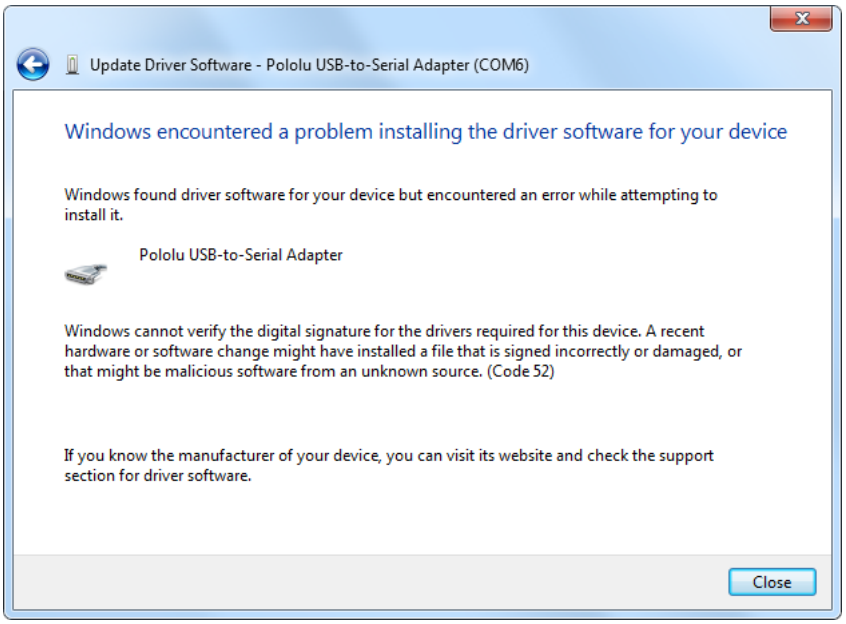
If Windows cannot find a properly-signed security catalog for your kernel module, the Properties dialog for your device will show a Code 52 error:

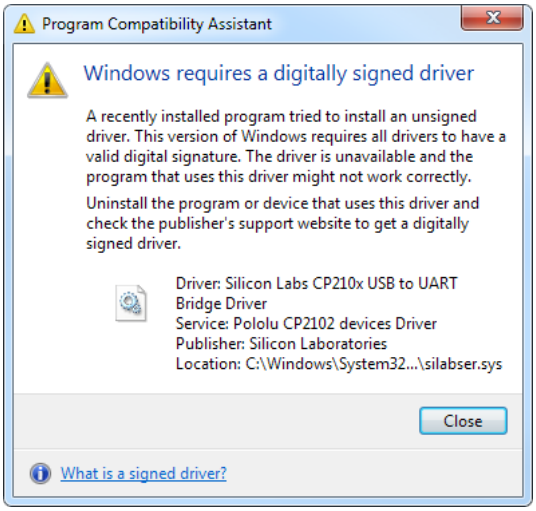


I have also sometimes seen a Code 39 error happen in this situation:

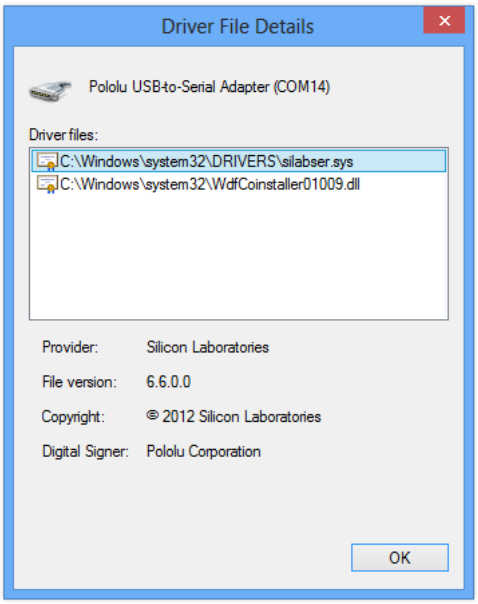


Sometimes Windows can detect that the kernel module is not signed as soon as you try to install the driver package instead of later when you plug in a matching device. Here are some error messages you might see if that happens:





The digital signature for a kernel module also affects what users see in the Device Manager. Just double-click on a device using the module, select the **Details tab**, and click **Driver details**. Ideally you would want your company's name to show up in this dialog box, but I have not done enough research to really know what the requirements are. That is why I put question marks in the "Loading a kernel module" column in the table above entitled "Signature requirements for it to look good".



Windows 10 kernel modules must be signed by the portal

Microsoft announced on 2015-04-01 that all new Windows 10 kernel mode drivers must be submitted to and digitally signed by the Windows Hardware Developer Center Dashboard portal, which is a web service provided by Microsoft.

On 2016-07-26, Microsoft announced that this rule will only be enforced on Windows 10 systems that were freshly installed at build 1607 or later, with Secure Boot on.

I do not have any personal experience using the portal so I can only repeat what Microsoft has claimed about how it works. According to an interview with Microsoft Program Manager James Murray on 2015-07-24, there are two different routes for getting your driving signed in the portal: "Hardware Compatibility" and "attestation".

The **Hardware Compatibility** route is the old route where your driver is subjected to a series of tests designed by Microsoft to ensure its quality. Then Microsoft will supposedly sign your driver properly for all the versions of Windows you are interested in, and you can have a single driver that works on all those versions.

The **attestation** route is the new route where your driver does not have to pass any tests, but the resulting driver only works on Windows 10.

The portal will only accept driver submissions from you if you sign them with an Extended Validation (EV) certificate, which is typically more expensive than a normal certificate.

This post about driver signing by Christoph Lüders documents his experience purchasing an EV certificate, getting an account on the portal, and going through the attestation route.

Microsoft's documentation for the portal might be useful. Also, the first presentation in this video from Microsoft about driver certification is mostly a repetition of the information from their written announcements but it is does allude to a registry key that you can use to turn off the portal requirement for the purpose of testing (without actually specifying the path to the key unfortunately).

SHA-2 certificates require KB3033929 on Windows 7

If your certificate uses SHA-2 or has SHA-2 certificates in its chain of trust and you are using it to sign kernel modules, then you should be aware of KB3033929, an update for Windows 7 distributed through Windows Update. On versions of Windows 7 without this update, the kernel will reject signatures made with certificates that use SHA-2, so they cannot be used to get a kernel module to load.

SHA-2 certificates do not work for Vista kernel modules

If your certificate uses SHA-2 or has SHA-2 certificates in its chain of trust, then you will not be able to use it to get kernel modules loaded into the Windows Vista 64-bit kernel. I tried to make this work on multiple occasions but I was never able to. Unlike Windows 7, there is no update to fix this. If you really need to make new kernel-mode drivers for Windows Vista 64-bit, you might try instructing your users on how to disable driver signature enforcement.

Use /t for timestamps if Windows Vista matters

I have not tested it, but I suspect Windows Vista 64-bit will not accept timestamps made with the `/tr` option when it is loading kernel modules, because that is how it behaves when checking a signature on an executable or driver package. Therefore, you should use `/t` instead.

Choosing a certificate

You will need to choose a certificate issuer and purchase a code signing certificate from them.

Code Signing Certificate from GlobalSign

A good option is the code signing certificate offered by Globalsign. You will have to choose whether to get an Extended Validation (EV) certificate or a normal certificate. The EV certificate is more expensive and probably more of a hassle, but it is required by Microsoft if you are going to sign kernel-mode drivers and you want those drivers

to generally work on Windows 10. Also, an EV certificate will give you "immediate reputation with Microsoft SmartScreen", making it less likely for users to see random errors when they download signed executables from you.

We purchased a normal code signing certificate from GlobalSign in 2015 and renewed it in 2016, and it has worked fine for signing our executables and driver packages. I also tested the 2015 certificate to see if it could sign kernel-mode drivers, and that worked, but that was before the new rules about the Windows Hardware Developer Center Dashboard portal were fully enforced. We got our certificate for only about $219 per year. The staff at GlobalSign handled our order very quickly: I was able to download and use the certificate withing 24 hours of placing the order.

However, your experience buying a certificate from them will be harder than ours, because of new code-signing security rules implemented on 2017-02-01. In particular, you won't be able to download the private key and certificate online; the private key will be provided to you on a USB token (SafeNet eToken 5100) that must be connected to your computer while making signatures. Globalsign is going above and beyond what the new rules actually require. The new rules only require the entity buying the certificate to attest that they are storing the private key using certain security procedures, and the rules allow the private key to be stored in a standard USB drive. To learn about the new rules, see the document Minimum Requirements for the Issuance and Management of Publicly-Trusted Code Signing Certificates by the Code Signing Working Group from 2016-09-22.

Be sure to install GlobalSign's R1-R3 cross-certificate on the computer that will be making signatures. You can find that certificate on this page of their website, and there is also a copy of it here. This certificate gets installed to the "Intermediate Certification Authorities" list, which is shown in certmgr.msc. My experiments have shown that by installing the certificate there, signtool will find it and include it in signatures that you make. The result is that any computer checking the signature will look for the GlobalSign root R1 certificate instead of looking for the GlobalSign root R3 certificate. The R1 certificate is much older, so it is likely to be available on more computers. For example, I found that on an internet-disconnected Windows 7 machine, the R1 certificate is available while the R3 certificate is not. On an internet-disconnected Windows Vista computer, unfortunately neither of the certificates were available. Regardless of what type of signature you are making, the R1-R3 cross-certificate will help your signature be recognized on more computers.

To learn more about how to do these experiments yourself and verify these results, see the testing section.

Code Signing Certificate from K Software

Another certificate authority to consider is K Software, a partner of Comodo. At the time of this writing, their certificates are considerably cheaper than GlobalSign's. Unlike GlobalSign, if you get a non-EV certificate, you can generate your own private key and store it on your computer; no USB hardware is required. **K Software provides a special deal for readers of this article: just enter "CPNDEG" in the "Partner Coupon" field to get 10% off the price of your certificate!**

I have not tried using a certificate from them yet, but I intend to.

Code Signing Certificate from Go Daddy

We used a SHA-1 code signing certificate from GoDaddy from 2012 to 2015, and then switched to GlobalSign. The GoDaddy certificate worked for signing executables and driver packages, but did not work for kernel-mode drivers (SYS files) because there was no cross certificate available to extend the chain of trust back to the Microsoft Code Verification Root. If you get a SHA-2 code signing certificate from GoDaddy, it might be just as good as the GlobalSign certificate I mentioned above, but I have not tested it.

Finding a good certificate product

Here are some tips for finding a good certificate:

- If you need to sign kernel-mode drivers and want them to generally work on Windows 10, make sure to get an Extended Validation (EV) certificate so you can submit your drivers to the Windows Hardware Developer Center Dashboard portal.
- If you need to sign kernel-mode drivers, it used to be important to make sure your chain of trust can reach back to the Microsoft Code Verification Root via one of the cross-certificates available on Microsoft's Cross-Certificates for Kernel Mode Code Signing web page. However, that is not important any more because the Windows Hardware Developer Center Dashboard portal supposedly will sign your drivers with a signature that works on all the versions of Windows that you specify.
- It is a good idea to look at a few different Windows computers to see which certificates are already installed in the Trusted Root Certification Authorities list, which is visible from certmgr.msc. You might also want to look at the certificates embedded as resources inside C:\Windows\System32\crypt32.dll, because those certificates can be automatically installed on demand. Unfortunately, I have not found or developed a good way to look at the certificates in crypt32.dll. Your goal is to buy a certificate whose chain of trust is rooted in a certificate that will already be a Trusted Root Certification Authority (or be inside crypt32.dll) on all of your users' computers so you don't have to rely on Windows Update. For example, on my Windows 8 computer I see "GlobalSign Root CA" in my Trusted Root Certification Authorities, which is one indication that GlobalSign is a good company to buy a certificate from.
- Make sure the certificate authority you are considering has a decent return policy.

How to sign

This section will explain what to do after you have purchased the code signing certificate in order to actually use it. This information can mostly be found from official sources, but some of those sources (e.g. kmsigning.doc) are out of date.

First of all, you should follow the installation instructions from your certificate provider. These will include some sort of procedure to get a private key and certificate that can be used on your computer. This process will probably involve installing one or more intermediate certificates on your computer so that you have a complete chain of trust from your certificate to a root certificate of your provider. After you have followed those instructions, you should open up certmgr.msc and look at your certificate to make sure everything looks good.

Cross-certificate

You might need to download an appropriate cross-certificate in order to extend your chain of trust and meet all the desired signature requirements. All of the standard cross-certificates that go back to the Microsoft Code Verification Root are available for download from Microsoft. Your certificate provider might have some other useful cross-certificates available for download on their website. To use a cross-certificate, you will have to include an argument of the form `/ac "path-to-your-cross-cert.ct"` when you invoke signtool. However, cross-certificates do not matter much anymore now that the Windows Hardware Developer Center Dashboard portal is available, which will sign drivers for you.

Intermediate certificate

It might be beneficial to download an intermediate certificate and install it on the computer making signatures. For example, as I explained above, the GlobalSign R1-R3 intermediate certificate extends the chain of trust from their new R3 certificate (which uses SHA-2) back to their older, better supported R1 certificate. My experiments

have shown that this kind of intermediate certificate will get automatically included in signatures that you make if it is installed on the signing computer. If the authority of the intermediate certificate corresponds to a root certificate that is older and better supported than your normal root certificate, then using the intermediate certificate could allow your signature to be recognized by more computers. However, if that authority's certificate is poorly supported, then using the intermediate certificate could probably hurt you, so watch out for that.

Digest algorithm

The digest algorithm (or file digest) is the hash function used on your file before it is signed. You have to choose whether to use SHA-1, SHA-256, or SHA-512. (If you do not want to choose, it is possible to apply multiple signatures to most types of files, but this does not work for MSI files.)

I recommend that you use SHA-256 or SHA-512 because SHA-1 will eventually be distrusted throughout Windows in all contexts. However, Windows Vista users will have a degraded experience if you don't use SHA-1.

I have not tested SHA-512 myself, but John Dallman reports that it works fine in Windows 7 and later, at least for signing executables. He says he is using SHA-512 in the hopes that his signatures will last longer; like SHA-1, SHA-256 might someday be deemed vulnerable and be distrusted.

To use SHA-256 as the digest algorithm (recommended), include the arguments `/fd sha256` when you invoke signtool.

To use SHA-512 as the digest algorithm, include the arguments `/fd sha512` when you invoke signtool.

To use SHA-1 as the digest algorithm, include the arguments `/fd sha1` when you invoke signtool.

Note that there is no way to specify the digest algorithm when running inf2cat; it seems like CAT files always use SHA-1.

Timestamp server, protocol, and digest algorithm

Make sure to timestamp your signatures so they will continue to work after your certificate expires. Your certificate provider should provide the URL of a timestamp server in their documentation, but you can probably use the timestamp server from any provider for free. Here is a list of timestamp servers I have heard about:

- http://rfc3161timestamp.globalsign.com/standard
- http://rfc3161timestamp.globalsign.com/advanced
- http://timestamp.globalsign.com/scripts/timstamp.dll
- http://timestamp.globalsign.com/scripts/timestamp.dll
- http://timestamp.globalsign.com/?signature=sha2
- http://sha256timestamp.ws.symantec.com/sha256/timestamp
- http://tsa.starfieldtech.com
- http://timestamp.entrust.net/TSS/RFC3161sha2TS
- http://timestamp.geotrust.com/tsa
- http://timestamp.comodoca.com

If you are using a GlobalSign certificate, I recommend using the GlobalSign timestamping server. That way, both your main signature and your timestamp signature can chain back to the same root certificate. Every root certificate that your signature relies on is a liability because it might be missing or unavailable on the user's system. If possible, it is better to rely on just one root certificate instead of two.

You should decide which algorithm to use for the timestamp: SHA-1 or SHA-2. I recommend using SHA-2 because SHA-1 will eventually be distrusted in Windows in all contexts. However, SHA-2 timestamps do not work in Windows Vista.

If you choose SHA-1 for the timestamp digest, you have a choice to either use the Authenticode protocol or RFC3161.

If you choose to use SHA-2 for the timestamp digest, you must use RFC3161.

To timestamp your signature using the RFC3161 protocol and SHA-2 (recommended), include the arguments `/tr http://timestampserver.com /td sha256` when you invoke signtool.

To timestamp your signature using the RFC3161 protocol and SHA-1, include the arguments `/tr http://timestampserver.com /td sha1` when you invoke signtool.

To timestamp your signature using the Authenticode protocol and SHA-1, include the arguments `/t http://timestampserver.com` when you invoke signtool.

Some time-stamping servers will disobey your `/td` argument, so be sure to inspect your signature to make sure it uses the right digest algorithm for the timestamp.

Note that the `/td` option doesn't just control the digest algorithm used for the timestamp, but usually timestamp servers also use it to select a proper certificate whose chain of trust uses that same algorithm.

Signtool and inf2cat

To sign anything, you will need the Signtool.exe (Sign Tool) utility from Microsoft. To obtain signtool.exe, I installed the latest version of the **Windows SDK**.

To sign driver packages, you first need to use another tool called Inf2Cat (Inf2Cat.exe) to create the security catalog (CAT) file, which you can then sign with signtool. To obtain inf2cat.exe, I installed the latest version of the **Windows Driver Kit (WDK)**.

You should probably get the latest versions of both signtool and inf2cat to ensure that your drivers will support the latest versions of Windows.

Links to the Microsoft website for downloading the Windows SDK and Windows WDK tend to break, so I do not have any here. I recommend using a search engine to search for "Windows SDK download" and "Windows WDK download" in order to find the latest versions. I think the SDK should be installed first. The WDK will install itself into the same folder as the SDK, which will be something like "C:\Program Files (x86)\Windows Kits\10" by default.

You can probably figure out how to use inf2cat and signtool from the documentation, but here are some examples of how to use them.

This is an example batch (.bat) script. You can simply drag an executable or MSI file onto it, and it will sign the file for you:

```
"C:\Program Files (x86)\Windows Kits\10\bin\x86\signtool" sign /v /ac "your-cross-cert.crt" /n "Your company name" /fd sha1 /tr http://timesta
pause
```

This is another example batch script. You can put it in the same directory as your driver package and then double-click on it to create the security catalog and sign it.

```
"C:\Program Files (x86)\Windows Kits\10\bin\x86\inf2cat" /v /driver:%~dp0 /os:XP_X86,Vista_X86,Vista_X64,7_X86,7_X64,8_X86,8_X64,6_3_X86,6_3_X
"C:\Program Files (x86)\Windows Kits\10\bin\x86\signtool" sign /v /ac "your-cross-cert.crt" /n "Your company name" /tr http://timestamp.global
pause
```

For both of these batch files, if you are using a cross-certificate, I recommend just putting the cross-certificate in the same directory as the batch file to make the `/ac` parameter simpler.

Verifying

You should use the verify option of signtool.exe to check your signatures while you are still learning the process. The documentation of the options for signtool verify is pretty confusing, so I will tell you what you need to know:

- To test a signature for the purpose of *running an executable* or *installing a driver package*, the correct option is `/pa`. I infer this from KMCS_Walkthrough.doc.
- To test a signature for the purpose of loading kernel-mode code, the correct option is `/kp`.

Here is an example batch script that verifies the signature of a file you drop onto it, using `/pa`:

```
"C:\Program Files (x86)\Windows Kits\10\bin\x86\signtool" verify /v /pa %1
pause
```

Tip: Run signtool verify *without* the `/v` option to see whether a signature's timestamp is an Authenticode timestamp (signed with `/t`) or an RFC3161 timestamp (signed with `/tr`). The distinction between these two types of timestamps is sometimes important and this is the only way I know to verify that the correct type was used.

Unfortunately, signtool verify has limited usefulness. It will make sure that your chain of trust extends back to the right place, but it will not tell you about most of the other signature requirements that I have documented above. The chain of trust reported by signtool verify is probably affected by the set of trusted root certificates and intermediate certificates that are installed on your computer. To actually be confident in your signatures, you need to properly **test** them, so keep reading.

Testing

It's pretty obvious that it would be ideal to test your signed drivers/executables on every different version of Windows you are targeting.

What isn't obvious is that when you are testing executables or MSI files, **you should run them right after downloading them from the internet**. As I explained in the Installing a driver package section, there is a bug in unpatched versions of Windows Vista that only manifests itself if the file was downloaded from the internet, and there could easily be more bugs like that. Just throw your executables into a zip file at a secret URL and download them onto the test computer. Generally, you will know that you are testing executables correctly if Windows displays an extra warning when you try to run the executable.

You should test your downloadable file (e.g. your ZIP file or installer) by downloading it in Internet Explorer to make sure there are no problems when Internet Explorer checks your signature. In my experience, Internet Explorer checks the signatures on EXE downloads (and probably MSI too), but in future versions it might reach inside ZIP files and check the signatures on the executable files inside.

You should conduct these tests on a machine that does not have any intermediate certificates from your certificate provider or timestamp provider installed. You can find them and delete them using the "Intermediate Certification Authorities" list in certmgr.msc. Windows will use those intermediate certificates to help build a chain of trust back to a trusted root certificate, so having them installed on your testing computer could affect the results of your tests. Deleting those certificates forces Windows to find them in the signed file itself.

You should also try deleting the root certificates that your main signature and your timestamp rely on. You can find them and delete them using the "Trusted Root Certification Authorities" list in certmgr.msc. Ideally, you would be able to delete those certificates, disable the computer's internet connection, and then verify that your signature still works. Windows will attempt to automatically install the root certificates it needs to verify your signature. In my experience, the automatic installation happens whenever Windows shows a dialog box that contains the publisher information of your signed file, and it also happens whenever you open the "Digital Signature Details" window in the properties of the file. If the signature gets successfully verified by Windows, you should see the certificates that you deleted reappear in certmgr.msc after you refresh it.

If your driver package includes a kernel-mode driver, the implication of Microsoft's driver signing changes in Windows 10, version 1607 is that you should test your driver on a Windows 10 system that was freshly installed at version 1607 or later and has Secure Boot enabled. Such a system has stricter kernel-mode driver signature requirements than other Windows 10 systems.

Signing myths

It's annoying when you ask for help and the good people trying to help you end up telling you things that are untrue or half-truths. Here are some of the myths I have encountered:

Myth: Kernel-mode drivers require WHQL testing

> Let's say that the booby girls at GoDaddy don't give the user much confidence in the a certificate issued by them, promising that the driver always works. Drivers for the 64-bit version of Windows have to be qualified by Microsoft, not them girls. Google "whql labs certifications". Hans Passant, who has 300,000+ reputation on StackOverflow, in response to my *question*

> A customized installation [generated by our software] does not contain certified drivers for Windows XP/2003/Vista/7. Certification must be performed by Microsoft for the new driver installation. An uncertified installation will not cause any other problems other than the warning message displayed by Windows XP/2003/Vista when installing uncertified drivers. Uncertified drivers cannot be installed in Windows 7 unless they are installed with a testing certificate or the Ignore Serial Signing option is enabled by pressing F8 on start up and selecting the corresponding option. Silicon Labs, makers of the CP2102 serial bridge, in AN220

This is a myth. Microsoft isn't trying to assert total control over what gets loaded into the kernel. They just want to make it easier to figure out what went wrong when the user experiences a blue screen of death or some other problem.

So starting with Windows Vista 64-bit, Windows requires signatures for loading kernel-mode code. Starting with Windows 8, they also require driver packages to be signed. Starting with Windows 10, kernel-mode drivers have to be signed by Microsoft using their portal, but that does not entail any testing of the quality of the driver. The code could still contain infinite loops and viruses, but at least it can be tracked to its source when problems arise!

See the signing requirements section for a complete explanation of what you need to do.

Myth: DefaultInstall doesn't work with signed drivers

> The INF file of a driver package must not contain an INF DefaultInstall section if the driver package is to be digitally signed. Microsoft, in the INF Default Install Section documentation

The documentation is incorrect. I have distributed signed drivers with DefaultInstall sections to our customers since November 2012 and the DefaultInstall section has caused no problems. I don't see any reason why there should be a problem. While I was figuring out the signing process in 2012, I used the DefaultInstall section almost exclusively as my method for testing driver package installation.

The DefaultInstall section allows a user to install your INF file simply by right-clicking on it and selecting "Install". Really you should use DPInst, SetupCopyOEMInf, or PnPUtil to install drivers, but the DefaultInstall section is easy to add and it could be useful to some customers, so you should have it.

For example, if your driver is named `foo_driver.inf`, you should add the following lines:

```
[DefaultInstall]
CopyINF=foo_driver.inf
```

You can even reference multiple INF files in the CopyINF directive if you want.

Myth: The INF version number indicates OS support

> Create an INF file in your driver package directory and edit it for Windows Vista. Specifically, change the build date to 4/1/2006 or greater and the version to 6. For example: `DriverVer=04/01/2006, 6.0.1.0` Microsoft, in kmsigning.doc

Generally, kmsigning.doc is pretty good, but that line is wrong.

I don't know if their claim about the date is correct, because I have never tried dating one of my drivers before 2006, but they are definitely wrong about the version number. I have successfully distributed drivers to thousands of customers who run Windows XP/Vista/7, and our driver versions are typically in the 1.0.0.0 to 3.0.0.0 range.

The INF DriverVer Directive is documented here on MSDN. If the DriverVer version number were important in some way, that should be documented on that page, not buried on page 11 of kmsigning.doc. In fact, the DriverVer version is *optional* according to that page.

I think the best practice for the version number is to start it at 1.0.0, and whenever you edit the file for any reason you should increase the version number and update the driver date. I recommend using semantic versioning.

Half-truth: Windows 7 doesn't support SHA-2

> In some cases, you might want to sign a driver package with two different signatures. For example, suppose you want your driver to run on Windows 7 and Windows 8. Windows 8 supports signatures created with the SHA256 hashing algorithm, but Windows 7 does not. For Windows 7, you need a signature created with the SHA1 hashing algorithm. Microsoft, from Signing a Driver for Public Release on MSDN

This is a half truth. In my experience, SHA-2 signatures on driver packages (i.e. CAT files) work just fine in Windows Vista and Windows 7 for the purpose of driver package installation. However, they don't work for the purpose of loading kernel modules (SYS files) into the kernel on systems that do not have the KB3033929 update installed. If you use SHA-2 to sign a driver-package that has kernel-mode code, you will get a Code 52 error when you plug in your device and actually try to use the driver on systems without KB3033929. If your driver package doesn't contain any new kernel modules (e.g. you use WinUSB or usbser.sys), a SHA-2 signature will work fine. For more details about this, see the signature requirements section above.

Sometimes telling your customer a half-truth can be worse than just telling a myth. The first time I read the paragraph from the MSDN documentation quoted above, I just assumed it was totally wrong because in my experience the SHA-2 signature was working fine for my WinUSB and usbser.sys-based driver packages. There is a kernel of truth to that paragraph, but unfortunately I could not receive that truth because it was veiled in inaccuracy. Keep that in mind the next time you try to write documentation or explain something to someone: if you say something that goes against their experience of the world, they will discount what you are saying.

On the other hand, someone once told me:

> Signing is perhaps the least suitable area to show off creativity and independent thinking. Just the opposite. Try to follow the instructions precisely. ... No matter what they scribble at Stack Overflow – the WDK documentations says the ultimate truth (when updated, of course). Pavel A., in response to my question on MSDN

Well, Pavel was right in this case. If I had turned off all of my creativity and independent thinking, I would have accepted that paragraph as the truth (even though it contradicts all available evidence) and it would have saved me some pain later.

I can't just turn off my brain, but I think it is important that we compromise with Pavel. We should take the documentation seriously, and when it says something that contradicts our experience, we should consider the possibility that the documentation could be correct in some other domain that we haven't tested yet. Therefore, some of the myths I listed above might *actually* be half-truths.

References

1. Getting a kernel mode driver signed for Windows 10. Christoph Lüders. 2016-09-09.
2. Driver Certification on Windows Client and Server. Microsoft Channel 9. 2016-05-27. Hour-long video.
3. Minimum Requirements for the Issuance and Management of Publicly-Trusted Code Signing Certificates. Code Signing Working Group. 2016-09-22. New requirements for protecting private keys using hardware are in section 16.3, and also mentioned in a blog post.
4. CAs To Apply Microsoft's New Digital Cert Code-Signing Requirements. Redmond Magazine. 2016-12-09.
5. Windows Hardware Dev Center dashboard. Microsoft. Documents the portal you must use for signing kernel-mode drivers.
6. Windows Enforcement of SHA1 Certificates. Microsoft. States that SHA-1 will eventually be distrusted throughout Windows in all contexts.
7. Driver Signing changes in Windows 10, version 1607. Microsoft. 2016-07-26.
8. SHA1 Certificate Signature Check (SHA1SigCertCheck.ps1). Microsoft. 2016-02-16. Script for checking if your signature uses SHA-1 and whether the SHA-1 deprecation applies.
9. How to Sign Windows Drivers & Executables. Adafruit. 2016-03-14. Explicit tutorial with plenty of screenshots.
10. Renew your Windows Code Signing Certificates by December 31, 2015. DCSoft blog. 2015-12-14.

11. Questions and Answers: Windows 10 Driver Signing. OSR poses questions to James Murray of Microsoft. 2015-07-24.
12. Windows Code Signing Hash Algorithm Support. GlobalSign. 2015-10-22. Nice chart about SHA-1 and SHA-256.
13. Authenticode in 2015. Eric Law, ex-Microsoft employee. 2015-01-28.
14. Driver signing changes in Windows 10. Windows Hardware Certification Blog. 2015-04-01.
15. KB2763674. Microsoft. SHA-2 bug fix for Vista.
16. KB3033929. Microsoft. SHA-2 for Windows 7.
17. KB2921916. Microsoft. Specific SHA-2 bug fix for Windows 7.
18. Digital Signatures for Kernel Modules on Windows (kmsigning.doc). Microsoft. 2007-07-25.
19. Kernel-Mode Code Signing Walkthrough (KMCS_walkthrough.doc). Microsoft. 2007-07-25.
20. Windows root certificate program members. Microsoft.
21. Cross-Certificates for Kernel Mode Code Signing. Microsoft.
22. Signtool.exe (Sign Tool). Microsoft.
23. Inf2Cat. Microsoft.
24. Automatic root certificate update problems when verifying my signed INF driver package. David Grayson. 2012-10-03.
25. How does Windows automatically install certificates in the Trusted Root Certification Authorities list? David Grayson. 2015-07-08.
26. Certificate Chaining Engine (CCE). Microsoft. Last updated 2013-11-20.
27. Signing Windows 8 Drivers. PiXCL Automation Technologies. 2013-03-09.
28. Microsoft Security Advisory (2880823). Microsoft. 2013-11-13.
29. Korean translation of this article. Heejune Kim. 2015-07-14.
30. Windows Authenticode Portable Executable Signature Format. Microsoft. 2008-03-21.

Comments

I would like to hear from you! I do not host comments here, but if you have anything to say, please post it to the MSDN thread I have started. If you have found this document to be useful, please up-vote my original post in that thread and write a reply saying thanks.

Revision History

- 2018-05-22: Add more information about Windows 10 in the "Loading a Kernel Module" section.
- 2017-08-30: Add K Software.
- 2017-04-12: I was wrong about the loophole; revised the article accordingly. Also added what I know about the new hardware security modules that are required as of 2017-02-01.
- 2017-02-23: Made it clear the SHA-1 will eventually be distrusted by Windows in all contexts, removed some advice about how to use SHA-1.
- 2017-01-26: Added a reference about the new hardware security module (HSM) requirement and other changes starting on 2017-02-01. (Thanks, John Dallman!)
- 2016-09-07: Changed the Digest section to recommend SHA-2, since that's what we do.
- 2016-08-07: Added info about Windows 10 build 1607.
- 2016-04-14: Removed links for downloading the WDK and SDK. Added references to new resources from Microsoft and Adafruit. Apparently WHQL testing is free now, so adjusted some mentions of "money" and "expensive".
- 2015-11-10: Added the really nice portal loophole. Added discussion in "How to sign" about how to pick digest algorithms.
- 2015-11-09: Added "SHA-1 phase-out" to to the signature requirements section.
- 2015-08-07: Added inf2cat OS options 6_3_X86 and 6_3_X64.
- 2015-07-23: Added information about the new driver signing portal for Windows 10.
- 2015-07-15: Major changes. Updated the document for SHA-2 and Windows 10. Also changed it recommend /t over /tr. Also added information about how intermediate certificates work and how they can be useful.
- 2015-03-20: Added information about KB3033929 in the note at the top.
- 2015-02-08: Added tip from Jimmy Kaz about avoiding spaces in INF file names.
- 2015-01-14: Added links to PnPUtil.
- 2014-12-16: Added a note at the top about how SHA-1 is going away so parts of this document will need to be updated.
- 2014-09-12: Added a mention of KB2763674, which should make SHA-2 executables usable on Windows Vista.
- 2014-03-07: Added a mentions of Microsoft Security Advisory (2880823).
- 2013-10-02: Added section Use /tr to specify the timestamp server and changed all examples to use /tr.
- 2013-03-11: Added new info from PiXCL about how to double-sign with SHA1 and SHA2 to the Use SHA-1 for kernel modules prior to Windows 8 section.
- 2013-01-07: Initial release.