Azure Trusted Signing for plugin developers

Latest update: Nov 11, 2024.

This guide covers the steps necessary to set up a modern code signing flow using Azure Trusted Signing on Microsoft Windows, for example as part of a automated build or CI process. It is primarily meant for developers that are working on audio plugins and apps and covers a few specifics on things such as AAX code signing, PACE wrapped binaries, with the hope that it will be useful for some.

Azure Trusted Signing is a new end-to-end code signing service which is currently available to the general public as part of Microsoft Azure.

We recommend this service because it is provided by Microsoft itself and as such all tools come directly from Microsoft, removing the need for any third party tools to make code signing work, or for any hardware-backed certificate setups. All in all, our experience with it has been very good.

1. Current state of affairs

Since 2022, code signing certificates (EV) for Microsoft Windows are no longer allowed to be derived from "unprotected" private key files. Private keys must be generated securely in a Hardware Security Module (HSM) with FIPS 140-2 or EAL 4+ rating, and thus a HSM is now a necessity for plain old local code signing.

Normally one would use `signtool.exe` with a local private key file belonging to a public-private certificate keypair issued by one of the allowed authorities, but this no longer works without use of a local HSM such as a Yubikey dongle. This can complicate things because you now need either a local HSM, a dedicated shared HSM or remote HSM.

Fortunately, there are currently at least two viable alternatives out there that will be covered in this guide.

1. Azure Trusted Signing - a new Azure service aimed specifically at code signing, which we can recommend.
2. Azure Key Vault - an Azure service that provides a remote HSM.

As the Azure Key Vault is a replacement for HSM use cases, will cover the use of the much easier Azure Trusted Signing in this guide instead.

2. Azure Trusted Signing

The way Azure Trusted Signing works is by extending `signtool.exe`. This is a known tool for code signing which is included in the Windows SDK. The connection to Azure is used to create and use an ad hoc certificate which is then used to sign a binary.

These certificates are issued by Microsoft. In fact, Microsoft itself acts as a certificate authority with a root certificate chain already preinstalled in modern versions of Microsoft Windows.

Unlike certificates issued from authorities such as Sectigo, DigiCert and others, the certificates issued by ACS are only valid for a short time (e.g. several days) so that certificates cannot be stolen easily. To be fair, the new requirement for a HSM to store these certificates, instead of a file, also makes this much less likely but still.

In any case, `signtool.exe` automatically and transparently takes care of issuing a new certificates when necessary. In this sense, it is very common to modern non-profit certificate authorities such as Let's Encrypt.

2.1. Pricing

The pricing plan of Azure Trusted Signing is based on a reasonable monthly fee with a maximum quota of signatures per month (e.g. 5000 signatures per month) after which a per-signature cost gets activated. There is also a bigger tier if you require much more signatures per month, e.g. if you're running extensive CI systems or such. Typically the smallest tier will probably work for most independent developers.

2.2. Setting up ACS on Microsoft Azure

The following assumes that you have access to the Azure Trusted Signing service in your Microsoft Azure account. These steps will involve using the Microsoft Azure web portal to configure things. It will only be necessary to configure these things once, and you should be good to go afterwards.

2.2.1. Creating an ACS resource

First, create an ACS resource:

- Go to Trusted Signing Accounts
- Click "Create". Fill in the details, pick one of the following regions:
  - East US (https://eus.codesigning.azure.net)
  - West US (https://wus.codesigning.azure.net)
  - West Central US (https://wcus.codesigning.azure.net)
  - West US 2 (https://wus2.codesigning.azure.net)
  - North Europe (https://neu.codesigning.azure.net)
  - West Europe (https://weu.codesigning.azure.net)

2.2.2. Granting administrative access

Now that we have an ACS resource, we need to grant administrative access in order to access it:

- Open the Trusted Signing resource.
- Go to "Access control (IAM)" (left menu).
- Click "Add" and "Add role assignment" (top).
- Select "Trusted Signing Identity Verifier". Next.
- Type in and select your current user account.
- Keep clicking "Review + assign" until done.

2.2.3. Create a code signing app

We will now create a code signing app, which we will be able to use with any of our local code signing tools:

- In the Azure portal, search for Microsoft Entra ID (formerly: Azure Active Directory).
- Click "Add" and "App registration" (top).
- Name: codesigning-app (or anything else)
- Who can use this application or access this API?: Accounts in this organizational directory only (Default Directory only - Single tenant)
- Redirect URI: (leave as is)
- Click "Register".
- Click on the app resource you've just created.

- Note down the "Application (client) ID" for later usage with signtool.
- Go to "Certificates & secrets" (left menu).
- Click "Client secrets" (top).
- Click "New client secret".
- Set "Expires" to the highest date possible.
- Click "Add".
- Note down the "Value", this is the secret for later usage with signtool.

2.2.4. Granting code signing app permissions

We will now grant our code signing app the right permissions:

- Open the Trusted Signing resource.
- Go to "Access control (IAM)" (left menu).
- Click "Add" and "Add role assignment" (top).
- Select "Trusted Signing Certificate Profile Signer". Next.
- Leave "User, Group or Service principal" selected. Click on "+ Select Members".
- Type in and select your app name (e.g. codesigning-app).
- Keep clicking "Review + assign" until done.

2.2.5. Verify identity

Now that the ACS basics have been set up, we need to set up ACS so we can issue certificates. One requirements for that is that your personal or company identity has been validated with Microsoft Azure. Let's start that process:

- Open the Trusted Signing resource.
- Go to "Identity validation" (left menu).
- Click "New" and "Public Trust" (top).
- Fill in the details. Note that Primary and Secondary E-mail(s) will not be published in any certificate. You may need a DUNS number as well.

Your mileage may vary, but expect the validation to take at least a day to be confirmed as it is a one-time verification that concerns your Microsoft Azure account.

2.2.6. Create certificate profile

We now need to create a certificate profile, which we can use for code signing:

- Open the Trusted Signing resource.
- Go to "Certificate profiles" (left menu).
- Click "Create" and "Public Trust" (top).

- Fill in a name and select the "Verified CN and O" appropriately (will appear when the identity validation process is done).
- Click "Create".

The certificate should now be ready to be used with signing.

Luckely, our setup on Microsoft Azure should now be complete.

2.3. Setting up Azure Trusted Signing for code signing

Now that ACS has been configured on Microsoft Azure, we can set up the code signing tools on our local development or automated build machine. The following steps assume that you have access to a machine running a recent version of Microsoft Windows, and you are able to execute commands in the command prompt.

2.3.1. Preparing the Azure CLI

To start off, we will install the Microsoft Azure CLI which will allow access from the machine to Azure.

- Install the Azure CLI.

We now need to create a "service principal" locally to enable our code signing tools to work. The following steps have been supplied to us and seem to work fine:

- Log in with the CLI using your admin/owner account:

```
az login
```

- Execute the following manual commands using the CLI

```
az ad sp create --id cf2ab426-f71a-4b61-bb8a-9e505b85bc2e
az ad app permission grant --id cf2ab426-f71a-4b61-bb8a-
9e505b85bc2e --api 00000003-0000-0000-c000-000000000000 --scope
User.Read
```

Note that these commands are only necessary once, in order to configure your system with the right credentials. There is no need to repeat these commands afterwards in case you just want to sign applications. The environment variables later in this guide will serve as a way to provide the necessary credentials instead.

2.3.2. Preparing `signtool.exe`

`signtool.exe` is the known code signing tools distributed by Microsoft with Windows SDKs. It can be used with ACS, which is excellent since it means that it is directly supported by Microsoft itself. In order to make it work however, it needs to be extended by means of a Dlib file.

This tool gets its credentials to your Microsoft Azure account either by command-line arguments, or by environment variables such as `AZURE_TENANT_ID`, `AZURE_CLIENT_ID` and `AZURE_CLIENT_SECRET` as we will use further ahead in this guide.

First, make sure that you have installed the minimum required dependencies:

- Windows 11 SDK 10.0.22621.755 or higher. This includes the minimum required version of `signtool.exe`.
- .NET 6.0 runtime. If this is not installed, signtool will fail silently without output.
- The Microsoft Trusted Signing Client containing the Dlib for signtool. You can either use use `nuget` to install this package, or download the package manually and open it as a zip archive. Install or extract this package to a known directory of your choice.

If you use NuGet, the following PowerShell command may be useful to you:

```
Invoke-WebRequest -Uri https://dist.nuget.org/win-x86-commandline/latest/nu
.\nuget.exe install Microsoft.Trusted.Signing.Client
```

After installing the above dependencies, we need to create a metadata configuration JSON file inside the directory of the Microsoft Trusted Signing Client in order to make the tool work with Microsoft Azure Trusted Signing.

- Create a `metadata.json` file with the following contents (replace details accordingly):

```
{
        "Endpoint": "<Code Signing Account Endpoint URL>",
        "CodeSigningAccountName": "<Code Signing Account Name>",
        "CertificateProfileName": "<Certificate Profile Name>"
}
```

    - Endpoint URL: choose according to the endpoint you've chosen for the Azure Trusted Signing:
        - East US (https://eus.codesigning.azure.net)
        - West US (https://wus.codesigning.azure.net)
        - West Central US (https://wcus.codesigning.azure.net)
        - West US 2 (https://wus2.codesigning.azure.net)

- North Europe (https://neu.codesigning.azure.net)
- West Europe (https://weu.codesigning.azure.net)

To help along with the configuration of the signtool, especially for automated build systems, we also require a number of environment variables to be set. These can be added either to the system globally, or can be set in your own scripts calling the sign tool on your own accord.

First, the sign tool uses a number of environment variables as credentials to authenticate with your Microsoft Azure account. Make sure the following environment variables are set accordingly to use with the Trusted Signing app you've created earlier (e.g. codesigning-app):

```
* `AZURE_TENANT_ID`: The Microsoft Entra tenant (directory) ID. Use the
value you noted down earlier. Can also be found in Microsoft Entra ID.
* `AZURE_CLIENT_ID`: The client (application) ID of an App Registration
in the tenant. Use the value you noted down earlier.
* `AZURE_CLIENT_SECRET`: A client secret ("value") that was generated
for the App Registration. Use the value you noted down earlier.
```

In addition, the commands and scripts specifically in this guide use a few environment variables of their own:

- `ACS_DLIB` should point to the exact filesystem path of the `Azure.CodeSigning.Dlib.dll` in the archive that was extracted above.
- `ACS_JSON` should point to the exact filesystem path of the `metadata.json` file that was created above.

Note that these last environment variables are specific to this guide only, and are thus different from the earlier environment variables expected by the signing tool.

2.3.3. Testing `signtool.exe`

We will now make sure that `signtool.exe` is working and capable of using ACS to sign executables.

Make sure that you have installed the minimum required dependencies:

- Windows 10 SDK 10.0.19041 or higher (or Windows 11 SDK). This includes the minimum required version of `signtool.exe`.
- .NET 6.0 runtime. If this is not installed, signtool will fail silently without output.

`signtool.exe` can typically be found at a location such as: `C:\Program Files (x86)\Windows Kits\10\bin\10.0.22000.0\x64`.

You should now be able to execute signtool accordingly:

```
signtool.exe sign /v /debug /fd SHA256 /tr
"http://timestamp.acs.microsoft.com" /td SHA256 /dlib %ACS_DLIB% /dmdf
%ACS_JSON% filetobesigned.exe
```

2.3.4. Using `signtool.exe` with AAX `wraptool.exe`

As of the latest update of this guide, PACE Eden SDK's `wraptool.exe` does not yet support Azure Trusted Signing as an option.

It is however possible to work around this issue by modifying the wrap tool in a way that allows for Azure Trusted Signing to work, this is because the wrap tool utility also uses Microsoft's `signtool.exe` internally, and ProTools on Windows expects binaries to be signed with a whitelisted Microsoft certificate authority.

Normally the wrap tool relies on either a certificate file and password (no longer possible due to HSM) or a sign id (HSM) to passed in by the developer, with no apparent support for any other tools or options. However, we can use utility scripts and let the wrap tool use these instead to injects all the necessary arguments for Azure Trusted Signing to work.

This section presents a stop-gap workaround while wraptools remains unsupported by PACE. It is by no means ideal but it works.

In order for this solution to work, make sure:

- Python 3 has been installed on the system. This is because the utility script below relies on it.
- Choose a directory on the system that is accessible and where you can place the utility scripts.

Now proceed by creating the utility scripts:


```
# args.tmp should contain untouched CLI arguments
args = None
with open('args.tmp', 'r') as f:
        args = f.read()

if args:
        with open('args.tmp', 'w') as f:
                # Filter and keep anything that starts with a C:\ path,
                match = re.search(r'\"?(c\:\\.*?)\"?$', args, re.IGNORE
                if match and match[1]:
```

```
                    f.write(match[1])
                    exit(0)

# Nothing to be done
exit(1)
```

2. Create a batch file called `aax-signtool.bat` with the following contents in a directory of your choice:

```
@echo off

:: KDSP Signtool wrapper for Eden SDK / AAX wraptool
::
:: wraptool invokes signtool but makes a lot of assumptions about
how we're going to sign
:: which are incompatible with Azure Trusted Signing.
::
:: This tool removes all its arguments and replaces it with the
correct or necessary ones.
:: Please adjust accordingly if necessary.
::
:: Run Eden SDK's wraptool as follows:
::
:: wraptool.exe sign --signtool signtool.bat --signid 1 --verbose --
installedbinaries --account ... --password ... --wcguid ... --in ...
::
:: signid 1 is bogus, but wraptool needs this nonsense in order to
start up..
::
:: The following environment variables are necessary:
::
:: SIGNTOOL_PATH
:: ACS_DLIB (points to Dlib.dll file)
:: ACS_JSON (points to the metadata.json file)
:: AZURE_TENANT_ID (Microsoft Azure tenant ID)
:: AZURE_CLIENT_ID (Microsoft Azure codesigning app client ID)
:: AZURE_SECRET_ID (Microsoft Azure codesigning app secret value)
::

:: Get script root dir, so we can find aax-signtool.py
set root=%~dp0
set root=%root:~0,-1%

:: wraptool seems to mangle signtool's args and doesn't properly
quote-escape the final binary path,
:: and batch is not easy with string handling, so we use python to
fix things up..
set args=%*
echo %args%>args.tmp
```

```
echo Patched signtool: Input arguments: %args%
python "%root%\aax-signtool.py"
set /p args=<args.tmp
echo Patched signtool: Filtered arguments: %args%
set file="%args%"

echo Patched signtool: File to sign: %file%

if not defined SIGNTOOL_PATH (
        echo Patched signtool: ERROR: SIGNTOOL_PATH not defined
        exit /b 1000
)
if not exist "%SIGNTOOL_PATH%" (
        echo Patched signtool: ERROR: Could not find signtool.exe at
"%SIGNTOOL_PATH%"
        exit /b 1000
)

echo Patched signtool: Executing: "%SIGNTOOL_PATH%" sign /v /debug
/fd SHA256 /tr "http://timestamp.acs.microsoft.com" /td SHA256 /dlib
%ACS_DLIB% /dmdf %ACS_JSON% %file%
"%SIGNTOOL_PATH%" sign /v /debug /fd SHA256 /tr
"http://timestamp.acs.microsoft.com" /td SHA256 /dlib %ACS_DLIB%
/dmdf %ACS_JSON% %file%
@if %errorlevel% neq 0 exit /b %errorlevel%

echo Patched signtool: Success
```

If you're curious as to what these utility scripts do, the purpose of these two scripts is:

```
1. To hook the PACE wrap tool's call to Microsoft's sign tool, by acting
as a proxy to the actual sign tool.
2. To inject the proper command-line arguments to Microsoft's sign tool,
along with the original arguments from the PACE wrap tool, so that Azure
Trusted Signing will be used instead of the regular HSM code signing.
3. To sanitize the arguments that the PACE wrap tool is passing to the
Microsoft sign tool, so things don't break. This is specifically the
purpose of the Python script, and is not ideal but necessary as this
step is very complicated to do with batch scripting. Another option
would've been a single Python script but it is unclear if the PACE wrap
tool would've accepted this.
```

In any case, you should now be able to run `wraptool.exe` such that it invokes `signtool.exe` making use of Azure Trusted Signing. In order to this, you will need to pass the following arguments, e.g.:

```
wraptool.exe sign --signtool aax-signtool.bat --signid 1 --verbose --
installedbinaries --account ... --password ... --wcguid ... --in ...
```

Where `aax-signtool.bat` should point its absolute path inside your chosen directory, such as `C:\Tools\aax-signtools.bat`.

You can ignore the `--signid 1` argument, but it is necessary to make wraptool work as it expects this argument to be present (remember, it is still thinking we are using HSM code signing, which we are not). This argument is completely ignored and defused by the actual utility scripts.

You should now be able to code sign and wrap binaries using Azure Trusted Signing. Remember to use `wraptool.exe sign` for code signing AAX plugins, and `wraptool.exe wrap` for wrapping and code signing binaries such as standalone applications.

3. Final words

With any luck, you should now have Azure Trusted Signing working with `signtool.exe` to sign application binaries, as well as with `wraptool.exe` to wrap and/or sign binaries.

Your mileage may vary depending on your situation, but the above instructions have been shown to been helpful for a lot of people so far!

Happy hacking!

🐨🚀