certitude Documentation

Release 1.0.1

Cory Benfield

June 02, 2016

Contents

Certitude provides Python bindings to the Windows and OS X system-native TLS certificate libraries. This allows Python programs to validate TLS certificates using the exact same logic used by native browsers on those platforms (e.g. Safari and Internet Explorer/Edge).

This means that by combining Certitude with the OpenSSL used by default on most Linuxes and BSDs, it is possible for a Python program to perform certificate validation in a platform-native manner on all major operating systems. This lets a Python program behave like a full native citizen of whatever platform it is installed on.

This documentation discusses how to use Certitude.

Contents:

CHAPTER 1

Installing Certitude

Certitude involves bindings to native-code libraries, and is a library that functions only on Windows and OS X. These two platforms traditionally have problems with native-code bindings from Python, due to their lack of compilers or fully-fledged dependency resolution.

For this reason, while Certitude is available as a source distribution, it also provides pre-compiled binary wheels for both Windows and OS X. For this reason it is *strongly preferred* that a recent pip is used for installing Certitude, as this will remove the need for a compiler toolchain that you may not have installed.

To install Certitude, the pip command is very simple:

```
$ pip install certitude
```

This should download and install a wheel that makes certitude available.

CHAPTER 2

Using Certitude

Certitude has one job: validating the TLS certificates a server has sent you. To do that, you need to pass Certitude the TLS certificate chain sent by the server, and the hostname you're expecting to connect to.

2.1 Getting A Certificate Chain

Certitude expects the TLS certificate chain as a list of TLS certificates stored in the DER representation. Unfortunately, the Python standard library's `ssl` module is not capable of providing the entire certificate chain, only the leaf certificate. This means that to use Certitude you will need to use pyopenssl or something like it: it's just the only way to guarantee that you get the complete certificate chain.

To get a certificate chain from PyOpenSSL, you'll want to make the connection as normal and then call `get_peer_cert_chain()`. This will get you your cert chain as a list of `X509` objects. These will need decoding.

Given an already existing connection `cnx`, you can get your list of certificates like this:

```
certs = cnx.get_peer_cert_chain()

encoded_certs = [
    OpenSSL.crypto.dump_certificate(OpenSSL.crypto.FILETYPE_ASN1, cert)
    for cert in certs
]
```

2.2 Validating The Chain

Once you have the chain in place, it's simple enough to validate it. Simply pass the chain into `certitude.validate_cert_chain` along with a unicode string containing the expected hostname. For example:

```
valid = validate_cert_chain(encoded_certs, u'http2bin.org')
```

The `validate_cert_chain` function returns `True` if the cert chain is valid, and `False` in any other case.

2.3 Notes

When validating certificates using certitude you'll likely want to *disable* OpenSSL's certificate validation. This is because OpenSSL and the platform-specific TLS validation code will build their certificate chains differently. In

particular, OpenSSL may be *unable* to validate a chain that the system library believes is valid. For that reason, put OpenSSL into the `VERIFY_NONE` mode and then handle the validation manually, *after* the connection is made but **before you send any data on it**.

We cannot stress this enough: **you must validate the certificates before sending or receiving data on the connection**.