Watson Health®

IBM® Digital Health Pass

Multi-Credential Verifier Library, Python

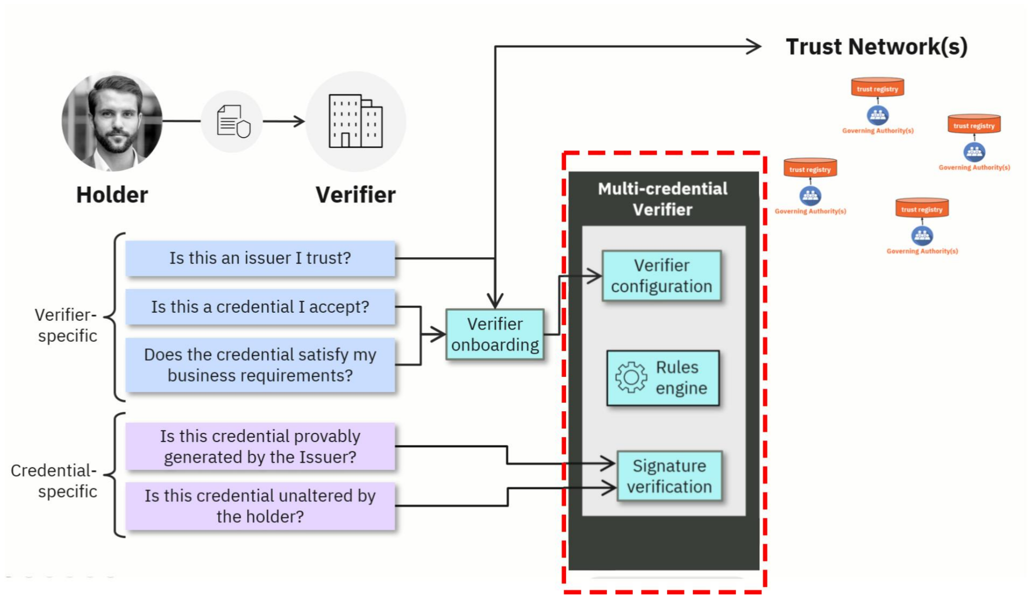
November 2021

Contents

Introduction

IBM® provides this extensible library for use by IBM Digital Health Pass customers that want to verify several types of digitally-verifiable healthcare credentials. The credential verifiers are referred to as **plug-ins**. If this library does not contain a verifier plug-in for a given type of credential, then you can create a custom plug-in and pass it to the library.

Figure: Multi-credential verifier in the IBM Digital Health Pass solution



Trust Network(s)
trust registry
Governing Authority(s)
Holder
Verifier
Verifier-specific
Is this an issuer I trust?
Is this a credential I accept?
Does the credential satisfy my business requirements?
Verifier onboarding
Multi-credential Verifier
Verifier configuration
Rules engine
Signature verification
Credential-specific
Is this credential provably generated by the Issuer?
Is this credential unaltered by the holder?

Available verifier plug-ins

In this SDK, IBM provides several plug-ins that can verify encrypted, digital immunization credentials.

Table 1: Currently-available verifier plug-ins

Healthcare credentials	Plug-in name
IBM Digital Health Pass (IDHP) and Good Health Pass (GHP)	idhp-verifier
Digital COVID Certificate (European Union)	eu-dcc-verifier
Vaccine Credential Initiative (VCI™) SMART Health Cards	vci-verifier

Using the library

To verify credentials, you must first include the library in your project. Then, you'll normally only need to import one class, **`CredentialVerifierBuilder`**.

```
`from
multi_cred_verifier_python.verifier.credential_verifier_build
er import CredentialVerifierBuilder`
```

To verify a credential:

```
    # Instantiate the builder.
    # builder is expensive to initialize so
    # reuse an instance to verify credentials
    builder = CredentialVerifierBuilder() \
        .set_healthpass_host_url(host_url) \
        .set_verifier_credential(verifier_credential) \
        .set_return_credential(True) \
        .set_return_metadata(True)

    #Initialize the builder instance
    init_response = builder.init()

    # If initialize fails, inspect response, and perhaps try
again
    if not init_response.success:
        return init_response

    # Start verifying credentials

    # Set the credential and build a verifier
    credential_verifier =
builder.set_credential(credential).build()

    # Verify the credential
    verify_result = credential_verifier.verify()

    # Verification was not successful.  Check
verify_result.message
    # and/or verify_result.error
    if not verify_result.success:
        return verify_result

    # Verification was successful
    print(verify_result)
```

builder.build() returns an instance of CredentialVerifier, which has only one public method, verify().

verifier.verify() returns a VerificationResult instance, which has this structure:

```
{
    "success": "true or false boolean",
    "message": "status message",
    "credType": "IDHP, GHP, SHC, DCC, UNKNOWN",
    "credential": "the extracted credential",
    "metadata": "the metadata extracted from the credential
with custom keys"
    "error": "axios error that occurred while communicating
with healthpass api"
    "warnings": "pending cache and/or verifier credential
expiration warnings"
    "credential": "the decoded credential"
}
```

Using CredentialVerifierBuilder

set_healthpass_host_url(healthpass_host_url) (required)

This sets the Healthpass Host URL that is used to obtain tokens, verifier configurations, and public keys:

```
CredentialVerifierBuilder().set_healthpass_host_url(healthpas
s_host_url)
```

init() (required)

This must be called after instantiating a CredentialVerifierBuilder, and before verifying credentials to initialize the builder.

Initialization logs in with the verifier credential, downloads the verifier configuration, pre-cache public keys (if configured to do so), and configures the builder to create verifiers. In addition, after setting a new verifier credential, init() must be called, using set_verifier_credential().

set_credential(credential) (required)

This sets the credential to be verified. The credential parameter can either be an object or a string. A credential must always be provided:

```
verifyResult = CredentialVerifierBuilder()
    .set_credential(credential)
    .build()
    .verify();
```

set_metadata_language(lang) (optional)

This sets the two-letter country code for the language to use for keys for the credential metadata returned in the VerificationResult. The default is en, for English:

```
new CredentialVerifierBuilder().setMetadataLanguage(lang);
```

set_additional_plugins(additionalPlugins) (optional)

This sets custom credential verifier plug-ins that the library does not provide. The additionalPlugins parameter can be either a single VerifierPlugin, or an array of VerifierPlugin. For more information about creating custom verifier plug-ins, see Creating a custom credential verifier plug-in.

set_disabled_plugins(disabledPlugins) (optional)

This sets the names of disabled credential verifier plug-ins. These plug-ins do not execute during credential verification:

```
    verifyResult = CredentialVerifierBuilder()
        .set_credential(credential)
        .set_disabled_plugins([
            'divoc-verifier',
            'eu-dgc-verifier',
            'vci-verifier'
        ])
        .build()
        .verify();
```

set_return_credential(returnCredential) (optional)

This sets whether a successful validation returns the credential in the payload (VerificationResult). The default is false:

```
    verifyResult = CredentialVerifierBuilder()
        .set_credential(credential)
        .set_return_credential(True)
        .build()
        .verify();
```

set_extras(extras) (optional)

This can be anything that is needed by a custom credential verifier plug-in:

```
    verifyResult = CredentialVerifierBuilder()
        .set_credential(credential)
        .set_extras({anything: 'anything needed for custom
plugin'})
        .build()
        .verify();
```

For more information, see Creating a custom credential verifier plug-in.

Verifier credential and cache expiration

A verifier credential has an expiration date within the JSON. The cache has a time to live (TTL) in seconds, which is specified in the verifier configuration. Both have a grace period of 90 percent of either the expiration date or the TTL, when the **VerificationResult** payload contains warnings of the pending expiration.

Example: If the cache expires in 24 hours, then the grace period starts in 21.6 hours, and the payload returns with a warning.

When the grace period begins, the cache automatically begins refreshing itself, if there is network connectivity. A new verifier credential can be set on the builder instance by calling **setVerifierCredential**, and then calling **init**.

If either the verifier credential or the cache expiration is reached, then verification is not possible until the issue is resolved. Here are the warning and errors that are returned:

> The cache will expire on <*DATE*>. Connect to network to refresh cache before then to continue verifying credentials.
>
> Verifier credential will expire on <*DATE*>. Set a new verifier credential while connected to network before then to continue verifying credentials.
>
> Verifier credential expired on <*DATE*>. Set a new verifier credential while connected to network to continue verifying credentials.
>
> Cache expired on <*DATE*>. Connect to network to refresh cache to continue verifying credentials.

Verification messages

This section lists messages returned in the **VerificationResult** payload.

Table 2: VerificationResult messages

Message text	Message value
Certificate's signature is not valid	The credential's signature is not valid.
Credential is not valid. Failing rule id(s):<*Rule_IDs*>	The credential failed at least one rule. This includes a comma-delimited list of the failed rule IDs.
Credential is valid	The credential passes signature and rules validation.
Display mapping not found for <*credType*>	The display mapping for a credential type is not found in the verifier configuration.
Revoke status validation failed :: Credential is revoked	The credential is revoked.
Rules not found	The rules for a credential type are not found in the verifier configuration.
Trust lists not found	The trust list for a credential type is not found in the verifier configuration.
Unknown Credential Type	Verifying an unknown credential type
Unknown Issuer	The issuer is not found in the Healthpass API.
Unknown public key format	The public key that is used to verify a credential is in an unknown format.

Network errors

If there is an error performing a request, the error object is returned in the **VerificationResult** error field, for debugging purposes.

Creating a custom credential verifier plug-in

You can create custom credential verifier plug-ins to verify credentials that are not supported by the library's provided plug-ins.

This creates a plug-in and imports the VerifierPluginBase, VerificationResult, and CredentialVerifierParams classes:

```
    from multi_cred_verifier_python.verifier.verifier_plugin
import VerifierPluginBase
    from
multi_cred_verifier_python.verifier.verification_result
import VerificationResult
    from
multi_cred_verifier_python.verifier.credential_verifier_param
s import CredentialVerifierParams
```

The plug-in must extend the abstract VerifierPluginBase class. There are two methods that must be implemented: verify(params) and get_name().

A string for the name of the plug-in should be returned from get_name(). The credential verification logic is in verify(params). During verification, an instance of CredentialVerifierParams is passed as the single parameter. The parameter contains all properties that are set on the builder instance. This method should return a VerificationResult instance, indicating whether the verification was successful.

To check whether the passed credential is of the type intended to be verified by the plug-in, a check should be placed at the beginning of the verification logic. If it isn't the intended type, then a VerificationResult instance should be returned; this instance contains both a False first parameter and a None second parameter:

```
class VerifierPluginBase(VerifierPluginBase):
    def verify(self, params: CredentialVerifierParams):
        credential = params.get_credential()
        public_key = params.get_extras()["public_key"]

        result =
self.check_is_example_credential_type(credential)

        if not result.success:
            return VerificationResult(False, None)

        result = self.verify_signature(credential,
public_key)

        if not result.success:
            return VerificationResult(
                False, 'Signature verification failed for
example credential type'
                )
        return VerificationResult(
            True, 'Signature verification passed for example
credential type', credential
            )

    def get_name(self):
        return 'example-credential-verifier'


    def check_is_example_credential_type(credential):
        return credential.type == 'ExampleCredentialType'

    def verify_signature(credential, publicKey):
        # return True if able to verify the credential's
        # signature with the public key, else return False.
        return True
```

To use the custom plug-in, simply pass the class to the builder using the set_additional_plugins() method:

```
    result = CredentialVerifierBuilder()
        .set_additional_plugins(ExampleCredentialVerifier)
        .set_credential(credential)
        .build()
        .verify();
```

Library licenses

This section lists open source libraries used in this SDK.

Table 3: Libraries and sources for this SDK

Library	Source
base45	BSD-2-Clause License (https://github.com/kirei/python-base45)
cbor2	MIT License (https://github.com/agronholm/cbor2
cwt	MIT License (https://github.com/dajiaji/python-cwt
json_logic_qubit	MIT License (https://github.com/qubitproducts/json-logic-py
jsonpath-python	MIT License (https://github.com/zhangxianbing/jsonpath-python
jwcrypto	LGPL-3.0 License (https://github.com/latchset/jwcrypto
pycryptodome	Apache® Software License, BSD License, Public Domain (BSD, Public Domain) (https://pypi.org/project/pycryptodome)
PyJWT	MIT License (https://github.com/jpadilla/pyjwt
python-dateutil	Either Apache 2.0 License or BSD 3-Clause (https://github.com/dateutil/dateutil
pytz	MIT License (https://pypi.org/project/pytz/)
requests-cache	BSD-2-Clause License (https://github.com/reclosedev/requests-cache)