OIDCtest Documentation

Release 0.5.0

Roland Hedberg

Dec 20, 2018

Contents

The task of these tools is to verify that a specific entity or library, follows the standard as specified in:

- OpenID Connect Core 1.0
- OpenID Connect Discovery 1.0
- OpenID Connect Client Registration 1.0

Contents:

CHAPTER 1

How to use OIDCtest

Release 0.5.0

Date Dec 20, 2018

Before you can use OIDCtest, you'll need to get it installed. If you have not done it yet, read the *Quick install guide*

1.1 Quick install guide

For all this to work you need to have Python installed. The development has been done using 3.5 . It's made to also work with 2.7 .

1.1.1 Prerequisites

For installing OIDCtest you will need

- otest
- oic
- jwkest

otest

This you have to get from github:

```
$ git clone https://github.com/openid-certification/otest
$ cd otest
$ python setup.py install
```

oic and jwkest

Both of these should be pulled in from pypi when you install *otest* . If not you can get them from PyPi:

```
$ pip install jwkest
$ pip install oic
```

oidctest

Must be gotten from github:

```
$ git clone https://github.com/openid-certification/oidctest
$ cd oidctest
$ python setup.py install
```

Setup

Once you have oidctest installed you can construct a new directory from which to run your test tools. To do this you can run the script make_test_dir.py like this:

```
$ oidc_setup.py <root of the oidctest source tree> <target directory>
```

for me this turned out to be:

```
$ oidc_setup.py /Users/roland/code/oidctest oidf
```

Note: that the root of the source tree specification must be absolute.

This will build a file tree in 'oidf' that will loook like this:

```
oidf --+-- oidc_op
       |
       +-- oidc_rpinst
       |
       +-- oidc_rplib
```

Dependent on whether you want to run OP, RP instance or RP library tests one or the other of these library will be of more interest to you.

How you use the different tools are described here:

oidc_op *How to run OP testing using OIDCtest*

oidc_rpinst *How to run RP instance testing using OIDCtest*

oidc_rplib *How to run RP library testing using OIDCtest*

1.2 How to run OP testing using OIDCtest

This tool is for testing an OpenID Connect Provider instances compliance with the standard.

Release 0.5.0

Date Dec 20, 2018

Before you can do any OP testing, you'll need to install oidctest. If you have not done it yet, read the *Quick install guide* documentation.

1.2.1 An overview of the OP test tool

A basic assumption for the tool is that when you want to test an OpenID Connect Provider (OP) you want to test one specific aspect at a time. You may therefor want to have several configurations per OP. It is for instance common to have one configuration per response_type. Following on that you will run one test instance per configuration.

How to configure a test instance

There are 2 basic ways of configuring a test instance, either you use the web interface provided or you can instead be content with using the RESTish interface.

- How to configure a test instance using the *Running the OP test tool using the form based interface for configuration*
- How to configure a test instance using the *Configuring the OP test tool using the REST interface*

Once you have configured a test instance, using the test instance doesn't differ depending on how you have configured it.

Running a test instance

To run a test instance you have to use a Python script 'optest.py' which is part of the oidctest distribution.

There are a number of arguments the script takes and I will go through them below one by one.

This is the overall pattern:

```
$ optest.py -h
usage: optest.py [-h] [-k] [-i ISSUER] [-f FLOWDIR] [-p PORT] [-M MAKODIR]
                 [-S STATICDIR] [-s] [-t TAG] [-m PATH2PORT]
                 config

positional arguments:
  config

optional arguments:
  -h, --help    show this help message and exit
  -k            insecure mode for when you're expecting to talk HTTPS to
                servers that use self-signed certificates
  -i ISSUER     The issuer ID of the OP
  -f FLOWDIR    A directory that contains the flow definitions for all the tests
  -p PORT       Which port the server should listen on
  -M MAKODIR    Root directory for the MAKO template files
  -S STATICDIR  Directory where static files are kept
  -s            Whether the server should support incoming HTTPS
  -t TAG        An identifier used to distinguish between different
                configuration for the same OP instance
  -m PATH2PORT  CSV file containing the path-to-port mapping that the reverse
                proxy (if used) is using
```

-h/–help

Will print the usage description as shown above

-k

If nothing else is said the tool will try to verify the certificates used in the HTTPS connection. This will not work if the OP uses self-signed certificates. Hence, the *-f* flag will turn of certification verification.

-i

The Issuer identifier of the OP that is to be tested.

-f

A directory that contains descriptions of all the tests in a domain specific manner. It contains one file per test and the content of the file is a JSON document adhering to a specific syntax. If you want to understand more about the test descriptions you can read more about them in *This page describes the format of the test descriptions*.

-p

Which port the test instance should listen on. Each test instance **MUST** have its own port.

-M

Mako is used as the bases for the WEB UI. If nothing is specified the directories that contains the MAKO templates ('htdocs', 'modules', ...) are expected to be in the directory from which optest.py is run. If that is not the case you have to give the path to the root here.

-S

There are a bunch of static files that the tool must be able to access. These are all the javascirpt files, the png, gif and css files. If nothing is specified they are expected to be in a directory named 'static' in the directory from which optest.py is run.

-s

If the test instance should use HTTPS then set this flag. If so the configuration file must contain specifications of there the certificate and key files are.

-t

If you have several configurations for one and the same OP then you can set a name each one of them, this is the *tag*.

-m

If you are running the test instance behind a reverse proxy you will want to translate between a path specification on the external side of the proxy and a port on the inside.

If for instance the test tool runs on optest.example.com then the publicly available path may be https://optest.example.com/test-00 which then by the reverse proxy would be translated into for instance http://localhost:8090. Given that the test instance was listening on port 8090.

The file that the *-m* flag points to is a csv file with two columns, the first contains the externally visible path and the second contains the internal port:

```
Path,Port
test-00,8090
test-01,8091
test-02,8092
```

and so on.

config

The configuration file looks like this:

```
import os

BASEDIR = os.path.abspath(os.path.dirname(__file__))

SERVER_CERT = "certs/cert.pem"
SERVER_KEY = "certs/key.pem"
CERT_CHAIN = None

# VERIFY_SSL = False

BASE = 'http://localhost'
ENT_PATH = 'entities'
ENT_INFO = 'entity_info'

KEYS = [
    {"key": "keys/enc.key", "type": "RSA", "use": ["enc"]},
    {"key": "keys/sig.key", "type": "RSA", "use": ["sig"]},
    {"crv": "P-256", "type": "EC", "use": ["sig"]},
    {"crv": "P-256", "type": "EC", "use": ["enc"]}
]
```

This configuration file is most probably the same for every test instance. The configuration in the entity_info is part of it the same for every instance the other part is default values for some parameters. What is in the entities directory is not the same for two test instances.

SERVER_CERT, SERVER_KEY and CERT_CHAIN

Are only necessary if the test instance is supposed to do HTTPS.

BASE

The base from which the urls, that the test instance (as an RP) publishes, are constructed. This includes claims like *redirect_uris*, *jwks_uri*, *tos_url*, *logo_uri*, *client_uri*, *policy_uri*, *sector_identifier_uri* and possibly more.

ENT_PATH

A path to where the test configurations are stored. The configurations are stored in a tree of the form <issuer identifier>/<tag> like this:

```
https%3A%2F%2Fexample.com --+-- code
                           |
                           +-- idtoken
```

As you can see the *issuer identifier* is quoted to be URL safe. The same goes for the tag though that isn't obvious from the example above.

ENT_INFO

This is information about the test instance which is static and should not differ between different test instances. Some of the information here represents default values and may be changed.

KEYS

The test instance needs a set of key for signing and encryption. This is where the set of keys are defined. The configuration sample above specifies 4 keys, two RSA keys and 2 elliptic curve keys. For each type one for signing and one for encryption purposes.

Usage examples

The test tool can run in two ways. It can be stand alone, listening on a, probably non-standard port. Or it can be run behind a reverse proxy which then converts a external path to an internal port.

Stand alone

Here the test tool is configured to listen to a specific port. It can be any port but common is that it's not one of the system ports. Which is necessary since the test tool normally is not run by root.

If the tool is stand-alone it has to deal with TLS/SSL itself. To do this the necessary keys and certificates has to be constructed and placed in the *certs* directory. It is also necessary to use the -s flag to get the software to do HTTPS. If for some reason there are problems with verifying the certificates used by the OP, the -k flag kan be use to turn off certificate verification.

Very simple command example where there is a flows.yaml file and a configuration file named 'config'

```
optest.py -p 8091 -i https://example.com/op -t default -s -f flows.yaml config
```

Reverse proxy setup

If a reverse proxy is used then the there will be an external URL that the RP is known as to the outside but also and internal URL which is only used between the proxy and the test tool.

An example could be that the external URL is: https://example.com/optest/op1

while the internal URL is: http://localhost:8666/

To accomplish this a couple of things has to happen. If you are running an Apache server as your reverse proxy you can find a description of the necessary steps on the apache reverse proxy page. You probably want to pre-configure a list of path-to-port mappings. Besides doing this in the reverese proxy you should also construct a csv file that contains the *path2port* mapping.

If you do that, the test tool will construct the correct external URL based on the *port* specification and the mapping defined in the csv file.

Since the reverse proxy will probably be used to terminate the HTTPS tunnel the tool will not have to deal with certificates which leaves us with the following simple command:

```
optest.py -p 8092 -i https://example.com/op -t default -f flows.yaml -m reverse.csv␣
↪config
```

1.2.2 Running the OP test tool using the form based interface for configuration

The configuration server

The configuration server is again a Python script:

```
$ config_server.py -h
usage: config_server.py [-h] [-b BASE_URL] [-c TEST_TOOL_CONF] [-f FLOWDIR]
                        [-m PATH2PORT] [-p PORT] [-t] [-M MAKO_DIR]
                        config

positional arguments:
  config

optional arguments:
  -h, --help         show this help message and exit
  -b BASE_URL
  -c TEST_TOOL_CONF
  -f FLOWDIR
  -m PATH2PORT
  -p PORT
  -t
  -M MAKO_DIR
```

-b

You should really set this in the configuration file rather then using this option. Anyway this is the base from which the tool will construct the necessary URLs.

-c

More about the test tool configuration *here*

-f

The *flows* information is passed on to the test tool instance

-m

The *path2port* information is passed on to the test tool instance

-p

The port the configuration server should listen to

-t

Turns on HTTPS support. If set the configuration server will not listen to HTTP calls only to HTTPS.

-M

The *Mako dir* information is passed on to the test tool instance

config

The configuration file looks like this:

```
import os

BASEDIR = os.path.abspath(os.path.dirname(__file__))

SERVER_CERT = "./certs/cert.pem"
SERVER_KEY = "./certs/key.pem"
CERT_CHAIN = None

#VERIFY_SSL = False

BASE_URL = 'http://localhost'
ENT_PATH = './entities'
ENT_INFO = './entity_info'
MAKO_DIR = './heart_mako'

FLOWDIR = './flows'

PATH2PORT = './path2port.csv'
PORT_MIN = 9100
PORT_MAX = 9149
```

SERVER_CERT, SERVER_KEY and CERT_CHAIN

Are only necessary if the test instance is supposed to do HTTPS.

BASE_URL

passed on to the configuration of a test tool instance The base from which the urls, that the test instance (as an RP) publishes, are constructed. This includes claims like *redirect_uris*, *jwks_uri*, *tos_url*, *logo_uri*, *client_uri*, *policy_uri*, *sector_identifier_uri* and possibly more.

ENT_PATH

passed on to the configuration of a test tool instance A path to where the test configurations are stored. The configurations are stored in a tree of the form <issuer identifier>/<tag> like this:

```
https%3A%2F%2Fexample.com --+-- code
                           |
                           +-- idtoken
```

As you can see the *issuer identifier* is quoted to be URL safe. The same goes for the tag though that isn't obvious from the example above.

ENT_INFO

passed on to the configuration of a test tool instance This is information about the test instance which is static and should not differ between different test instances. Some of the information here represents default values and may be changed.

MAKO_DIR

passed on to the configuration of a test tool instance Where the MAKO template files can be found. This is the root directory so within this directory there must be a ht_docs directory with the actual templates.

FLOWDIR

passed on to the configuration of a test tool instance Directory that contains descriptions of all the tests in a domain specific manner. One test per file and the information is stored in a JSON format. If you want to understand more about the test descriptions you can read more about them in *This page describes the format of the test descriptions*.

PATH2PORT

passed on to the configuration of a test tool instance More about this *here*.

PORT_MAX, PORT_MIN

Defines the number of test instances that the configuration server can spin off and which ports it can use for these. When all ports are taken no more test instance can be started unless a running test instance is removed.

The web interface

When you have started a configuration server you can connect to the port it listens on and see this:


OIDCtesting/OPtest/confserv0.png

This page gives you two options, you can either browse the existing test instance configurations until you find the one you want or you can go to the page where you can create a new configuration.

Browsing test configurations

What you will see if you chose browsing test instances is something like this:


OIDCtesting/OPtest/issuerlist.png

Picking one of the issuers, the next page shown will look like this:


OIDCtesting/OPtest/testinstance.png

Here you can remove a test instance configuration, restart the instance if it's running or just start it if it's not or change the configuration. If you change the configuration then the test instance is restarted.

Creating a new configuration

Now, if you go down the configuration way you will get a page like this:


OIDCtesting/OPtest/testconf0.png

which filled in will look like this:

If you click **submit** you will get the next page where there are some things you may want to change.

Tool Configuration

Once you have configured everything to you liking and have clicked *Save&Start* a test instance will be started with the configuration you just constructed.

acr_values

Here you can enter a list of the Authentication Context Class References that your OP supports.

claims_locales

Languages and scripts supported for values in Claims being returned, represented as a list of BCP47 [RFC5646] language tag values.

enc

Whether the OP support encryption/decryption.

extra

The test tool contains three sets of tests. Those that are testing something that is mandatory to support, those that test things that are optional to support and those that just tests nice to suuport things. By setting this to **True** you will get the extra set of nice-to-have support tests.

insecure

Even though your server supports HTTPS which it **must** according to the standard you may use self-signed certificates which can not be verifies. If that is the case you **MUST** set *insecure* to **True** which will tell the test instance to not attempt any verification of the certificate.

login_hint

login_hint is something a RP may use as a hint to the Authorization Server about the login identifier the End-User might use to log in. Here you can set such a hint using a format that you know your OP will understand.

profile

This is a short-form of the test profile you have set. You can not change this value since it is constructed from the base values.

sig

Whether the OP supports signing/verifying signatures.

tag

A value you set at the beginning of the configuration to distinguish this configuration from others you may have. *Can not be changed*

ui_locales

Languages and scripts supported for the user interface, represented as a list of BCP47 [RFC5646] language tag values.

webfinger_email

If you want to test your OsP support for Webfinger queries you have to supply a resource specification to use. It can either be a email/acct like string. Which you would then enter here.

webfinger_url

The webfinger resource you want to use is might also be an URL. Which you would then enter here. If you specify both *webfinger_email* and *webfinger_url* both will be tested.

1.2.3 Configuring the OP test tool using the REST interface

Intentionally left blank Content to come.

1.2.4 This page describes the format of the test descriptions

1.3 How to run RP library testing using OIDCtest

Release 0.5.0

Date Dec 20, 2018

Before you can do any RP library testing, you'll need to install oidctest. If you have not done it yet, read the *Quick install guide*

1.3.1 Running the RP library test tool

Intentionally left blank :-)

1.4 How to run RP instance testing using OIDCtest

Release 0.5.0

Date Dec 20, 2018

Before you can do any RP instance testing, you'll need to install oidctest. If you have not done it yet, read the *Quick install guide*

1.4.1 Running the RP instance test tool

Intentionally left blank :-)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search