Federation Feeder Documentation

Release 1.1.2

Leif Johansson

Apr 29, 2020

Documentation

Author *Leif Johansson <leifj@sunet.se>*

Release 1.1.2

pyFF is a simple but reasonably complete SAML metadata processor. It is intended to be used by anyone who needs to aggregate, validate, combine, transform, sign or publish SAML metadata.

pyFF is used to run infrastructure for several identity federations of signifficant size including edugain.org.

pyFF supports producing and validating digital signatures on SAML metadata using the pyXMLSecurity package which in turn supports PKCS#11 and other mechanisms for talking to HSMs and other cryptographic hardware.

pyFF is also a complete implementation of the SAML metadata query protocol as described in draft-young-md-query and draft-young-md-query-saml and implements extensions to MDQ for searching which means pyFF can be used as the backend for a discovery service for large-scale identity federations.

Possible usecases include running an federation aggregator, filtering metadata for use by a discovery service, generating reports from metadata (eg certificate expiration reports), transforming metadata to add custom elements.

CHAPTER 1

Installation

1.1 Before you install

Make sure you have a reasonably modern python. pyFF is developed using 3.6 but 3.7 will probably become the norm soon. It is recommended that you install pyFF into a virtualenv

Start by installing some basic OS packages. For a debian/ubuntu install:

```
# apt-get install build-essential python-dev libxml2-dev libxslt1-dev libyaml-dev
```

and if you're on a centos system (or other yum-based systems):

```
# yum install python-devel  libxml2-devel libxslt-devel libyaml-devel
# easy_install pyyaml # bug in pip install pyyaml
# yum install make gcc kernel-devel kernel-headers glibc-headers
```

If you want to use OS packages instead of python packages from pypi then consider also installing the following packages before you begin:

1.1.1 With Sitepackages

This method re-uses existing OS-level python packages. This means you'll have fewer worries keeping your python environment in sync with OS-level libraries.

```
# apt-get install python-virtualenv
# virtualenv python-pyff
```

Choose this method if you want the OS to keep as many of your packages up to date for you.

1.1.2 Without Sitepackages

This method keeps everything inside your virtualenv. Use this method if you are developing pyFF or want to run multiple python-based applications in parallell without having to worry about conflicts between packages.

```
# cd $HOME
# apt-get install python-virtualenv
# virtualenv -p python3 python-pyff --no-site-packages
```

Choose this method for maximum control - ideal for development setups.

1.2 Verifying

To verify that python 3.6 is the default python in the pyFF environment run

```
# python --version
```

The result should be Python 3.6 or later.

To verify that the version of pip you have is the latest run.

```
# pip install --upgrade pip
```

1.3 Installing

Now that you have a virtualenv, its time to install pyFF into it. Start by activating your virtualenv:

```
# source python-pyff/bin/activate
```

Next install pyFF:

```
# cd $HOME
# cd pyFF
# LANG=en_US.UTF-8 pip install -e .
```

This will install a bunch of dependencies and compile bindings for both lxml, pyyaml as well as pyXMLSecurity. This may take some time to complete. If there are no errors and if you have the *pyff* binary in your **$PATH** you should be done.

```
# cd $HOME
# mkdir pyff-config
# cd pyff-config
```

1.4 Upgrading

Unless you've made modifications, upgrading should be as simple as running

```
# source python-pyff/bin/activate
# pip install -U pyff
```

This should bring your virtualenv up to the latest version of pyff and its dependencies. You probably need to restart pyffd manually though.

1.5 Next Steps

Now that you hopefully have a working installation of pyFF you are ready to start exploring all the ways pyFF can help you manage metadata. It may be good to go read the *Quick Start Instructions* now but in general pyFF should be run in the same directory that contains a pipeline in *yaml* format and depending on the nature of the pipeline additional files may be needed including things like...

- A list of metadata URLs.
- A set of files containing metadata URLs - eg *XRD* or *MDSL* files.
- A *key* and *crt* signing key pair which can be generated from *genkey.sh* in the scripts directory.

CHAPTER 2

Quick Start Instructions

There are a lot of options and knobs in pyFF - in many ways pyFF is a toolchain that can be configured to do a lot of tasks. In order to start exploring pyFF it is best to start with a simple example. Assuming you have read the installation instructions and have created and activated a virtualenv with pyFF installed do the following:

First create an empty directory and cd into it. In the directory create a file called edugain.fd with the following contents:

```
- load:
   - http://mds.edugain.org
- select:
- stats:
```

Now run pyFF like this:

```
# pyff edugain.fd
```

You should see output like this after a few seconds depending on the speed of your Internet connection you should see something like this:

```
---
total size:     5568
selected:       5567
          idps: 3079
           sps: 2487
---
```

Congratulations - you have successfully fetched, parsed, selected and printed stats for the edugain metadata feed. This is of course not a useful example (probably) but it illustrates a few points about how pyFF works:

- pyFF configuration is (mostly) in the form of yaml files
- The yaml file reprsents a list of instructions which are processed in order
- The *load* statement retrieves (and parses) SAML metadata from edugain.org
- The *select* statement is used to form an *active document* on which subsequent instructions operate
- Finally, the stats statement prints out some information about the current active document.

Next we'll learn how to do more than print statistics.

CHAPTER 3

Running pyFF

There are two ways to use pyFF:

a “batch” command-line tool called pyff # a wsgi application you can use with your favorite wsgi server - eg gunicorn

In either case you need to provide some configuration and a *pipeline* - instructions to tell pyFF what to do - in order for anything intersting to happen. In the *Quick Start Instructions* guide you saw how pyFF pipelines are constructed by creating yaml files. The full set of piplines is documented in `pyff.builtins`. When you run pyFF in batch-mode you typically want a fairly simple pipline that loads & transforms metadata and saves some form of output format.

3.1 Batch mode: pyff

The typical way to run pyFF in batch mode is something like this:

```
# pyff [--loglevel=<DEBUG|INFO|WARN|ERROR>] pipeline.yaml
```

For various historic reasons the yaml files in the examples directory all have the ‘.fd’ extension but pyFF doesn’t care how you name your pipeline files as long as they contain valid yaml.

This is in many ways the easiest way to run pyFF but it is also somewhat limited - eg it is not possible to produce an MDQ server using this method.

3.2 WSGI application: pyffd

Development of pyFF uses gunicorn to test but othe wsgi servers (eg apache mod-wsgi etc) should work equally well. Since all configuration of pyFF can be done using environment variables (cf pyff.constants:Config) it is pretty easy to integrate in most environments.

Running pyFFd using gunicorn goes something like this (incidentally this is also how the standard docker-image launches pyFFd):

```
# gunicorn --workers=1 --preload --bind 0.0.0.0:8080 -e PYFF_PIPELINE=pipeline.yaml --
↪threads 4 --worker-tmp-dir=/dev/shm  pyff.wsgi:app
```

The wsgi app is a lot more sophisticated than batch-mode and in particular interaction with workers/threads in gunicorn can be a bit unpredictable depending on which implementation of the various interfaces (metadata stores, schedulers, caches etc) you choose. It is usually easiest to use a single worker and multiple threads - at least until you know what you're doing.

The example above would launch the pyFF wsgi app on port 8080. However using pyFF in this way requires that you structure your pipeline a bit differently. In the name of flexibility, most of the request processing (with the exception of a few APIs such as webfinger and search which are always available) of the pyFF wsgi app is actually delegated to the pipeline. Lets look at a basic example:

```
- when update:
  - load:
      - http://mds.edugain.org
- when request:
  - select:
  - pipe:
      - when accept application/samlmetadata+xml application/xml:
          - first
          - finalize:
              cacheDuration: PT12H
              validUntil: P10D
          - sign:
              key: sign.key
              cert: sign.crt
          - emit application/samlmetadata+xml
          - break
      - when accept application/json:
          - discojson
          - emit application/json
          - break
```

Lets pick this pipeline apart. First notice the two *when* instructions. The pyff.builtins:when pipe is used to conditionally execute a set of instructions. There is essentially only one type of condition. When processing a pipeline pyFF keeps a state variable (a dict-like object) which changes as the instructions are processed. When the pipeline is launched the state is initialized with a set of key-value pairs used to control execution of the pipeline.

There are a few pre-defined states, in this case we're dealing with two: the execution mode *update* or *request* (we'll get to that one later) or the *accept* state used to implement content negotiation in the pyFF wsgi app. In fact there are two ways to express a condition for *when*: with one parameter in which case the condition evaluates to *True* iff the parameter is present as a key in the state object, or with two parameters in which case the condition evaluates to *True* iff the parameter is present and has the prescribed value.

Looking at our example the first when clause evaluates to *True* when *update* is present in state. This happens when pyFF is in an update loop. The other when clause gets triggered when *request* is present in state which happens when pyFF is processing an incoming HTTP request.

There 'update' state name is only slightly "magical" - you could call it "foo" if you like. The way to trigger any branch like this is to POST to the */api/call/{state}* endpoint (eg using cURL) like so:

```
# curl -XPOST -s http://localhost:8080/api/call/update
```

This will trigger the update state (or foo if you like). You can have any number of entry-points like this in your pipeline and trigger them from external processes using the API. The result of the pipeline is returned to the caller (which means it is probably a good idea to use the *-t* option to gunicorn to increase the worker timeout a bit).

The *request* state is triggered when pyFF gets an incoming request on any of the URI contexts other than */api* and */.well-known/webfinger*, eg the main MDQ context */entities*. This is typically where you do most of the work in a pyFF MDQ server.

The example above uses the *select* pipe (`pyff.builtins.select()`) to setup an active document. When in request mode pyFF provides parameters for the request call by parsing the query parameters and URI path of the request according to the MDQ specification. Therefore the call to *select* in the pipeline above, while it may appear to have no parameters, is actually "fed" from the request processing of pyFF.

The subsequent calls to *when* implements content negotiation to provide a discojuice and XML version of the metadata depending on what the caller is asking for. This is key to using pyFF as a backend to the thiss.io discovery service. More than one content type may be specified to accommodate noncompliant MDQ clients.

The rest of the XML "branch" of the pipeline should be pretty easy to understand. First we use the `pyff.builtins.first()` pipe to ensure that we only return a single EntityDescriptor if our select match a single object. Next we set cacheDuration and validUntil parameters and sign the XML before returning it.

The rest of the JSON "branch" of the pipeline is even simpler: transform the XML in the active document to discojson format and return with the correct Content-Type.

3.3 The structure of a pipeline

Pipeline files are *yaml* documents representing a list of processing steps:

```
- step1
- step2
- step3
```

Each step represents a processing instruction. pyFF has a library of built-in instructions to choose from that include fetching local and remote metadata, xslt transforms, signing, validation and various forms of output and statistics.

Processing steps are called pipes. A pipe can have arguments and options:

```
- step [option]*:
    - argument1
    - argument2
    ...

- step [option]*:
    key1: value1
    key2: value2
    ...
```

Typically options are used to modify the behaviour of the pipe itself (think macros), while arguments provide runtime data to operate on. Documentation for each pipe is in the `pyff.builtins` Module. Also take a look at the *Examples*.

CHAPTER 4

Deploying pyFF

4.1 Running pyFF in docker

4.1.1 Building a docker image

There is a build environment for docker available at https://github.com/SUNET/docker-pyff. In order to build your own docker image, clone this repository and use *make* to build the latest version of pyFF:

```
# git clone https://github.com/SUNET/docker-pyff
...
# cd docker-pyff
# make
```

At the end of this you should be able to run pyff:<version> where <version> will depend on what is currently the latest supported version. Sometimes a version of docker is uploaded to dockerhub but there is no guarantee that those are current or even made by anyone affiliated with the pyFF project.

4.1.2 Running the docker image

The docker image is based on debian:stable and contains a full install of pyFF along with most of the optional components including PyKCS11. If you start pyFF with no arguments it launches a default pipeline that fetches edugain and exposes it as an MDQ server:

```
# docker run -ti -p 8080:8080
```

A pyFF MDQ service should now be exposed on port 8080. If you are running the old pyFF 1.x branch you may also have access to the default admin interface. If you are running pyFF 2.x you can now point an MDQ frontend to to port 8080 - eg *mdq-browser*.

4.2 Running pyFF in production

There are several aspects to consider when deploying pyFF in production. Sometimes you want to emphasize simplicity and then you can simply run a pyFF instance and combine with a management application (eg mdq-browser) and a discovery service to quickly setup a federation hub. This model is suitable if you are setting up a collaboration hub or an SP proxy that needs to keep track of a local metadata set along with a matching discovery service.

4.2.1 Scenario 1: all-in-one

If you are using docker you might deploy something like that using docker-compose (or something similar implemented using k8s etc). Assuming your.public.domain is the public address of the service you wish to deploy the follwoing compose file would give you a discovery service on port 80 and an admin UI on port 8080.

Take care to check which version of the software components is the latest and greatest (and/or apropriate for your situation) and modify accordingly.

```
version: "3"
services:
   mdq-browser:
      image: docker.sunet.se/mdq-browser:1.0.1
      container_name: mdq_browser
      ports:
         - "8080:80"
      environment:
         - MDQ_URL=http://pyff
         - PYFF_APIS=true
   thiss:
      image: docker.sunet.se/thiss-js:1.1.2
      container_name: thiss
      ports:
         - "80:80"
      environment:
         - MDQ_URL=http://pyff/entities/
         - BASE_URL=https://your.public.domain
         - STORAGE_DOMAIN=your.public.domain
         - SEARCH_URL=http://pyff/api/search
   pyff:
      image: docker.sunet.se/pyff:stable
      container_name: pyff-api
```

4.2.2 Scenario 2: offline signing

Sometimes security is paramount and it may be prudent to firewall the signing keys for your identity federation but you still want to provide a scalable MDQ service. The MDQ specification doesn't actually require online access to the signing key. It is possible to create an MDQ service that only consists of static files served from a simple webserver or even from a CDN.

The pyFF wsgi server implements the webfinger protocol as described in **RFC 7033** and this endpoint can be use to list all objects in the MDQ server. A simple script provided in the scripts directory of the pyFF distribution uses webfinger and wget to make an isomorphic copy of the pyFF instance.

Run an instance of pyff on a firewalled system with access to the signing keys - eg via an HSM # Use the script to mirror the pyFF instance to a local directory and copy that directory over to the public webserver or CDN

```
# docker run -d -p 8080:8080 pyff:1.1.0
# docker run -ti pyff:1.1.0 mirror-mdq.sh -A http://localhost:8080/ /some/dir
```

This will create an offline copy of http://localhost:8080/ in /some/dir. You can use rsync+ssh syntax instead (eg user@host:/some/dir) to make a copy to a remote host using rsync+ssh. This way it is possible to have a lot of control over how metadata is generated and published while at the same time providing a scalable public interface to your metadata feed.

Currently the script traverses all objects in the pyFF instance everytime it is called so allow for enough time to sign every object when you setup your mirror cycle.

CHAPTER 5

Examples

Here are some more example pipelines. Most of these are designed for batch-mode pyff but the concepts can be easily included in wsgi-style pipelines with multiple entry-points.

5.1 Example 1 - A simple pull

Fetch SWAMID metadata, split it up into EntityDescriptor elements and store each as a separate file in /tmp/swamid-2.0.xml.

```
- load:
    - http://mds.swamid.se/md/swamid-2.0.xml
- select
- publish: "/tmp/swamid-2.0.xml"
- stats
```

This is a simple example in 3 steps: load, select, store and stats. Each of these commands operate on a metadata repository that starts out as empty. The first command (load) causes a URL to be downloaded and the SAML metadata found there is stored in the metadata repository. The next command (select) creates an active document (which in this case consists of all EntityDescriptors in the metadata repository). Next, (publish) is called which causes the active document to be stored in an XML file. Finally the stats command prints out some information about the metadata repository.

This is essentially a 1-1 operation: the metadata loaded is stored in a local file. Next we'll look at a more complex example that involves filtering and transformation.

5.2 Example 2 - Grab the IdPs from edugain

Grab edugain metadata, select the IdPs (using an XPath expression), run it through the built-in 'tidy' XSL stylesheet (cf below) which cleans up some known problems, sign the result and write the lot to a file.

```
- load:
   - http://mds.edugain.org
   - edugain-signer.crt
- select:
   - "http://mds.edugain.org!//md:EntityDescriptor[md:IDPSSODescriptor]"
- xslt:
    stylesheet: tidy.xsl
- finalize:
    cacheDuration: PT5H
    validUntil: P10D
- sign:
    key: sign.key
    cert: sign.crt
- publish: /tmp/edugain-idp.xml
- stats
```

In this case the select (which uses an xpath in this case) picks the EntityDescriptors that contain at least one IDPSSODescriptor - in other words all IdPs. The xslt command transforms the result of this select using an xslt transformation. The finalize command sets cacheDuration and validUntil (to 10 days from the current date and time) on the EntitiesDescriptor element which is the result of calling select. The sign command performs an XML-dsig on the EntitiesDescriptor.

For reference the 'tidy' xsl is included with pyFF and looks like this:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
                xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata">

  <xsl:template match="@ID"/>
  <xsl:template match="@Id"/>
  <xsl:template match="@xml:id"/>
  <xsl:template match="@validUntil"/>
  <xsl:template match="@cacheDuration"/>
  <xsl:template match="@xml:base"/>
  <xsl:template match="ds:Signature"/>
  <xsl:template match=
↪"md:OrganizationName|md:OrganizationURL|md:OrganizationDisplayName">
    <xsl:if test="normalize-space(text()) != ''">
            <xsl:copy><xsl:apply-templates select="node()|@*"/></xsl:copy>
    </xsl:if>
  </xsl:template>

  <xsl:template match="text()|comment()|@*">
    <xsl:copy/>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

5.3 Example 3 - Use an XRD file

Sometimes it is useful to keep metadata URLs and signing certificates used for validation in a separate file and pyFF supports XRD-files for this purpose. Modify the previous example to look like this:

```
- load:
   - links.xrd
- select: "!//md:EntityDescriptor[md:IDPSSODescriptor]"
- xslt:
    stylesheet: tidy.xsl
- sign:
    key: sign.key
    cert: sign.crt
- publish: /tmp/idp.xml
- stats
```

Note that in this case the select doesn't include the http://mds.edugain.org prefix before the '!'-sign. This causes the xpath to operate on all source URLs, rather than just the single source http://mds.edugain.org . It would have been possible to call select with multiple arguments, each using a different URL from the file links.xrd which contains the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<XRDS xmlns="http://docs.oasis-open.org/ns/xri/xrd-1.0">
    <XRD>
        <Subject>http://mds.swamid.se/md/swamid-2.0.xml</Subject>
        <Link rel="urn:oasis:names:tc:SAML:2.0:metadata" href="http://mds.swamid.se/
↪md/swamid-2.0.xml">
            <Title>SWAMID</Title>
            <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
                <ds:X509Data>
                    <ds:X509Certificate>
                    MIIFyzCCA7OgAwIBAgIJAI9LJsUJXDMVMA0GCSqGSIb3DQEBCwUAMHwxCzAJBgNV
                    BAYTAlNFMRIwEAYDVQQIDAlTdG9ja2hvbG0xEjAQBgNVBAcMCVN0b2NraG9sbTEO
                    MAwGA1UECgwFU1VORVQxDzANBgNVBAsMBlNXQU1JRDEkMCIGA1UEAwwbU1dBTUlE
                    IG1ldGFkYXRhIHNpZ25lciB2Mi4wMB4XDTE2MTIwNjA5MjgyMFoXDTM2MTIwNjA5
                    MjgyMFowfDELMAkGA1UEBhMCU0UxEjAQBgNVBAgMCVN0b2NraG9sbTESMBAGA1UE
                    BwwJU3RvY2tob2xtMQ4wDAYDVQQKDAVTVU5FVDEPMA0GA1UECwwGU1dBTUlEMSQw
                    IgYDVQQDDBtTV0FNSUQgbWV0YWRhdGEgc2lnbmVyIHYyLjAwggIiMA0GCSqGSIb3
                    DQEBAQUAA4ICDwAwggIKAoICAQDQVw72PnIo9QIeV439kQnPcxZh/LddKw86eIU+
                    nMfl4TpjSIyqTu4KJSnXbJyqXg+jQj3RzE9BUblpGrR7okmQwOh2nh+5A6SmyTOR
                    p7VEVT/Zw0GNnQi9gAW7J8Cy+Gnok4LeILI5u43hPylNKAnvs1+bo0ZlbHM6U5jm
                    6MlO+lrYA9dZzoPQqoCQbr3OweAaq5g8H54HuZacpYa3Q2GnUa4v+xywjntPdSQU
                    RTAbWWyJl3cHctX5+8UnX8nGCaxoBZqNp9PcEopyYJX8O1nrLumBMqu9Uh6GW1nx
                    OHfKDLvUoykG3Dm704ENVs88KaJXB1qQNsjdlm14UI9XCZbHfnFVnQ53ehsGFMha
                    Bf/Abd6v2wnhBLH/RxEUlw347qSeokw+SdDTSdW8jOEBiSqP/8BUzpCcbGlgAsVO
                    NKUS0K7IB2Bb79YYhyMvmJl24BGtkX+VM/mv47dxOtfzNFCMtUcJ2Dluv0xJG8xI
                    ot7umx/kbMBLuq7WdWELZJrgpt2bb9sXtYBpuxtGCW5g7+U7MNN1aKCiCSfq09YH
                    qu2DsU7HHAxEcGFXBiepBliCwZ24WLQh53bA3rihaln7SjdapT9VuSTpCvytb9RX
                    rq39mVuHMXvWYOG20XTV0+8U2vnsjAwsy28xPAcrLWRWoZbRJ+RoGp6L3GACq+t+
                    HPIukwIDAQABo1AwTjAdBgNVHQ4EFgQUQ2iqKQV/mMZDeJDtLXvy0Bsn/BQwHwYD
                    VR0jBBgwFoAUQ2iqKQV/mMZDeJDtLXvy0Bsn/BQwDAYDVR0TBAUwAwEB/zANBgkq
                    hkiG9w0BAQsFAAOCAgEAHviIAfS8viUN8Qk//U1p6Z1VK5718NeS7uqabug/SwhL
                    Vxtg/0x9FPJYf05HXj4moAf2W1ZLnhr0pnEPGDbdHAgDC672fpaAV7DO95d7xubc
                    rofR7Of2fehYSUZbXBWFiQ+xB5QfRsUFgB/qgHUolgn+4RXniiBYlWe6QJVncHx+
                    FtxD+vh1l5rLNkJgJLw2Lt3pbemSxUvv0CJtnK4jt2y95GsWGu1uSsVLrs0PR1Lj
                    kuxL6zZH4Pp9yjRDOUhbVYAnQ017mdcjvHYtp7c4GIWgyaBkDoMtU6fAt70QpeGj
                    XhecXk7Llx+oYNdZn14ZdFPRGMyAESLrT4Zf9M7QS3ypnWn/Ux0SwKWbnPUeRVbO
```

(continues on next page)

(continued from previous page)

```
                    VZZ+M0jmdYK6o+UU5xH3peRWSJIjjRaKjbVlW5GgHwGFmQc/LN+va2jjThRsQWWt
                    zEwObijedInQ6wfL/VzFAwlWWoDAzKK9qnK4Rf3ORKkvhKrUa//2OYnZD0kHtHiC
                    OL+iFRLtJ/DQP5iZAF+M1Hta7acLmQ8v7Mn1ZR9lyDWzFx57VOKKtJ6RAmBvxOdP
                    8cIgBNvLAEdXh2knOLqYU/CeaGkxTD7Y0SEKx6OxEEdafba//MBkVLt4bRoLXts6
                    6JY25FqFh3eJZjR6h4W1NW8KnBWuy+ITGfXxoJSsX78/pwAY+v32jRxMZGUi1J4=
                    </ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </Link>
    </XRD>
    <XRD>
        <Subject>https://incommon.org</Subject>
        <Link rel="urn:oasis:names:tc:SAML:2.0:metadata" href="http://md.incommon.org/
↪InCommon/InCommon-metadata.xml">
            <Title>InCommon Metadata (main aggregate)</Title>
            <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
                <ds:X509Data>
                    <ds:X509Certificate>
    MIIDgTCCAmmgAwIBAgIJAJRJzvdpkmNaMA0GCSqGSIb3DQEBCwUAMFcxCzAJBgNV
    BAYTAlVTMRUwEwYDVQQKDAxJbkNvbW1vbiBMTEMxMTAvBgNVBAMMKEluQ29tbW9u
    IEZlZGVyYXRpb24gTWV0YWRhdGEgU2lnbmluZyBLZXkwHhcNMTMxMjE2MTkzNDU1
    WhcNMzcxMjE4MTkzNDU1WjBXMQswCQYDVQQGEwJVUzEVMBMGA1UECgwMSW5Db21t
    b24gTExDMTEwLwYDVQQDDChJbkNvbW1vbiBGZWRlcmF0aW9uIE1ldGFkYXRhIFNp
    Z25pbmcgS2V5MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA0Chdkrn+
    dG5Zj5L3UIw+xeWgNzm8ajw7/FyqRQ1SjD4Lfg2WCdlfjOrYGNnVZMCTfItoXTSp
    g4rXxHQsykeNiYRu2+02uMS+1pnBqWjzdPJE0od+q8EbdvE6ShimjyNn0yQfGyQK
    CNdYuc+75MIHsaIOAEtDZUST9Sd4oeU1zRjV2sGvUd+JFHveUAhRc0b+JEZfIEuq
    /LIU9qxm/+gFaawlmojZPyOWZ1JlswbrrJYYyn10qgnJvjh9gZWXKjmPxqvHKJcA
    TPhAh2gWGabWTXBJCckMe1hrHCl/vbDLCmz0/oYuoaSDzP6zE9YSA/xCplaHA0mo
    C1Vs2H5MOQGlewIDAQABo1AwTjAdBgNVHQ4EFgQU5ij9YLU5zQ6K75kPgVpyQ2N/
    lPswHwYDVR0jBBgwFoAU5ij9YLU5zQ6K75kPgVpyQ2N/lPswDAYDVR0TBAUwAwEB
    /zANBgkqhkiG9w0BAQsFAAOCAQEAaQkEx9xvaLUt0PNLvHMtxXQPedCPw5xQBd2V
    WOsWPYspRAOSNbU1VloY+xUkUKorYTogKUY1q+uh2gDIEazW0uZZaQvWPp8xdxWq
    Dh96n5US06lszEc+Lj3dqdxWkXRRqEbjhBFh/utXaeyeSOtaX65GwD5svDHnJBcl
    AGkzeRIXqxmYG+I2zMm/JYGzEnbwToyC7yF6Q8cQxOr37hEpqz+WN/x3qM2qyBLE
    CQFjmlJrvRLkSL15PCZiu+xFNFd/zx6btDun5DBlfDS9DG+SHCNH6Nq+NfP+ZQ8C
    GzP/3TaZPzMlKPDCjp0XOQfyQqFIXdwjPFTWjEusDBlm4qJAlQ==
                    </ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </Link>
    </XRD>
</XRDS>
```

The structure of the file should be fairly self-evident. Only links with @rel="urn:oasis:names:tc:SAML:2.0:metadata" will be parsed. If a KeyInfo with a X509Certificate element (usual base64-encoded certificate format) then this certificate is used to validate the signature on the downloaded SAML metadata. Note that while 'load' supports validation based on certificate fingerprint the XRD format does not and you will have to include Base64-encoded certificates if you want validation to work.

5.4 Example 4 - Sign using a PKCS#11 module

Fetch SWAMID metadata (and validate the signature using a certificate matching the given SHA1 fingerprint), select the Identity Providers, tidy it up a bit and sign with the key with the label 'signer' in the PKCS#11 module

/usr/lib/libsofthsm.so. If a certificate is found in the same PKCS#11 object, that certificate is included in the Signature object.

```
- load:
   - http://mds.swamid.se/md/swamid-2.0.xml␣
↪A6:78:5A:37:C9:C9:0C:25:AD:5F:1F:69:22:EF:76:7B:C9:78:67:67:3A:AF:4F:8B:EA:A1:A7:6D:A3:A8:E5:85
- select: "!//md:EntityDescriptor[md:IDPSSODescriptor]"
- xslt:
    stylesheet: tidy.xsl
- sign:
    key: pkcs11:///usr/lib/libsofthsm.so/signer
- publish: /tmp/idp.xml
- stats
```

Running this example requires some preparation. Run the 'p11setup.sh' script in the examples directory. This results in a SoftHSM token being setup with the PIN 'secret1' and SO_PIN 'secret2'. Now run pyFF (assuming you are using a unix-like environment).

```
# env PYKCS11PIN=secret1 SOFTHSM_CONF=softhsm.conf pyff --loglevel=DEBUG p11.fd
```

CHAPTER 6

Extending pyFF

Not much here yet - come back later or UTSL

CHAPTER 7

Frequently Asked Questions

7.1 I get 'select is empty' but I know my xpath should match. What is wrong?

You may have forgotten to include namespaces in your xpath expression. For instance *//EntityDescriptor* won't match anything - *//md:EntityDescriptor* is what you want etc. PyFF is not a full XML processor and supports a set of well-known XML namespaces commonly used in SAML metadata by prefix only. The full list of prefixes can be found in `pyff.constants`

CHAPTER 8

pyff package

8.1 Submodules

8.1.1 pyff.api module

8.1.2 pyff.builtins module

8.1.3 pyff.constants module

8.1.4 pyff.decorators module

8.1.5 pyff.exceptions module

8.1.6 pyff.fetch module

8.1.7 pyff.i18n module

8.1.8 pyff.locks module

8.1.9 pyff.logs module

8.1.10 pyff.mdq module

8.1.11 pyff.mdx module

8.1.12 pyff.merge_strategies module

8.1.13 pyff.parse module

8.1.14 pyff.pipes module

8.1.15 pyff.repo module

8.1.16 pyff.resource module

Index

R