privacyIDEA Authentication System

Release 3.3

Cornelius Kölbel

Apr 13, 2020

CONTENTS

privacyIDEA is a modular authentication system. Using privacyIDEA you can enhance your existing applications like *local login*, *VPN*, *remote access*, *SSH connections*, access to web sites or *web portals* with a second factor during authentication. Thus boosting the security of your existing applications. Originally it was used for OTP authentication devices. But other "devices" like challenge response and SSH keys are also available. It runs on Linux and is completely Open Source, licensed under the AGPLv3.

privacyIDEA can read users from many different sources like flat files, different LDAP services, SQL databases and SCIM services. (see *Realms*)

Authentication devices to provide two factor authentication can be assigned to those users, either by administrators or by the users themselves. *Policies* define what a user is allowed to do in the web UI and what an administrator is allowed to do in the management interface.

The system is written in python, uses flask as web framework and an SQL database as datastore. Thus it can be enrolled quite easily providing a lean installation. (see *Installation*)

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Overview

privacyIDEA is a system that is used to manage devices for two factor authentication. Using privacyIDEA you can enhance your existing applications like local login, VPN, remote access, SSH connections, access to web sites or web portals with a second factor during authentication. Thus boosting the security of your existing applications.

In the beginning there were OTP tokens, but other means to authenticate like SSH keys are added. Other concepts like handling of machines or enrolling certificates are coming up, you may monitor this development on Github.

privacyIDEA is a web application written in Python based on the flask micro framework. You can use any webserver with a wsgi interface to run privacyIDEA. E.g. this can be Apache, Nginx or even werkzeug.

A device or item used to authenticate is still called a “token”. All token information is stored in an SQL database, while you may choose, which database you want to use. privacyIDEA uses SQLAlchemy to map the database to internal objects. Thus you may choose to run privacyIDEA with SQLite, MySQL, PostgreSQL, Oracle, DB2 or other database.

The code is divided into three layers, the API, the library and the database layer. Read about it at *Code Documentation*. privacyIDEA provides a clean *REST API*.

Administrators can use a Web UI or a command line client to manage authentication devices. Users can log in to the Web UI to manage their own tokens.

Authentication is performed via the API or certain plugins for FreeRADIUS, simpleSAMLphp, Wordpress, Contao, Dokuwiki… to either provide default protocols like RADIUS or SAML or to integrate into applications directly.

Due to this flexibility there are also many different ways to install and setup privacyIDEA. We will take a look at common ways to setup privacyIDEA in the section *Installation* but there are still many others.

1.2 Installation

The ways described here to install privacyIDEA are

- the installation via the *Python Package Index*, which can be used on any Linux distribution and
- ready made *Ubuntu Packages* for Ubuntu 16.04 LTS and 18.04 LTS.

If you want to upgrade please read *Upgrading*.

1.2.1 Python Package Index

You can install privacyidea on usually any Linux distribution in a python virtual environment. This way you keep all privacyIDEA code in one defined subdirectory.

privacyIDEA can run with Python 2.7 and 3.5, 3.6 or 3.7. Other versions either do not work or are not tested.

You first need to install some development packages. E.g. on Debian-based distributions like Ubuntu the packages are called

- libjpeg-dev
- libz-dev
- python-dev
- libffi-dev
- libssl-dev
- libxslt1-dev

Eventually you may have to install database-related packages such as

- libpq-dev

Now you can install privacyIDEA like this:

```
virtualenv /opt/privacyidea

cd /opt/privacyidea
source bin/activate
```

Now you are within the python virtual environment. Within the environment you can now run:

```
pip install privacyidea
```

To achieve a deterministic installation, you can now pin the installed versions to our tested versions like this:

```
pip install -r lib/privacyidea/requirements.txt
```

Please see the section *The Config File* for a quick setup of your configuration.

Then create the encryption key and the signing keys:

```
pi-manage create_enckey
pi-manage create_audit_keys
```

Create the database and the first administrator:

```
pi-manage createdb
pi-manage admin add admin -e admin@localhost
```

Now you can run the server for your first test:

```
pi-manage runserver
```

Stamp the database, so that privacyIDEA has the right database schema version. This is important for later update processes:

```
pi-manage db stamp head -d /opt/privacyidea/lib/privacyidea/migrations
```

Depending on the database you want to use, you may have to install additional packages (see above).

1.2.2 Ubuntu Packages

There are ready made packages for Ubuntu.

Packages of older releases of privacyIDEA up to version 2.23 are available for Ubuntu 14.04 LTS and Ubuntu 16.04 LTS from a public ppa repository[1].

For recent releases of privacyIDEA starting from version 3.0 a repository is available which provides packages for Ubuntu 16.04 LTS and 18.04 LTS[2].

Note: The packages `privacyidea-apache2` and `privacyidea-nginx` assume that you want to run a privacyIDEA system. These packages deactivate all other (default) websites. Instead, you may install the package `privacyidea-mysql` to install the privacyIDEA application and setup the database without any webserver configuration. After this, you can integrate privacyIDEA with your existing webserver configuration.

Read about the upgrading process in *Upgrading a packaged installation*.

Installing privacyIDEA 3.0 or higher

Before installing privacyIDEA 3.0 or upgrading to 3.0 you need to add the repository.

Add repository

The packages are digitally signed. First you need to download the signing key:

```
wget https://lancelot.netknights.it/NetKnights-Release.asc
```

On Ubuntu 16.04 check the fingerprint of the key:

```
gpg --with-fingerprint NetKnights-Release.asc
```

On 18.04 you need to run:

```
gpg --import --import-options show-only --with-fingerprint NetKnights-Release.asc
```

The fingerprint of the key is:

```
pub 4096R/AE250082 2017-05-16 NetKnights GmbH <release@netknights.it>
Key fingerprint = 0940 4ABB EDB3 586D EDE4 AD22 00F7 0D62 AE25 0082
```

Now add the signing key to your system:

```
apt-key add NetKnights-Release.asc
```

Now you need to add the repository for your release (either xenial/16.04LTS or bionic/18.04LTS)

You can do this by running the command:

```
add-apt-repository http://lancelot.netknights.it/community/xenial/stable
```

or:

[1] https://launchpad.net/~privacyidea

[2] Starting with privacyIDEA 2.15 Ubuntu 16.04 packages are provided. Starting with privacyIDEA 3.0 Ubuntu 16.04 and 18.04 packages are provided, Ubuntu 14.04 packages are dropped.

```
add-apt-repository http://lancelot.netknights.it/community/bionic/stable
```

As an alternative you can add the repo in a dedicated file. Create a new file `/etc/apt/sources.list.d/privacyidea-community.list` with the following contents:

```
deb http://lancelot.netknights.it/community/xenial/stable xenial main
```

or:

```
deb http://lancelot.netknights.it/community/bionic/stable bionic main
```

Installation of privacyIDEA 3.x

After having added the repositories, run:

```
apt update
apt install privacyidea-apache2
```

If you do not like the Apache2 webserver you could alternatively use the meta package `privacyidea-nginx`.

Now you may proceed to *First Steps*.

FreeRADIUS

privacyIDEA has a perl module to "translate" RADIUS requests to the API of the privacyIDEA server. This module plugs into FreeRADIUS. The FreeRADIUS does not have to run on the same machine as privacyIDEA. To install this module run:

```
apt-get install privacyidea-radius
```

For further details see rlm_perl.

1.2.3 CentOS Installation

Step-by-Step installation on CentOS

In this chapter we describe a way to install privacyIDEA on CentOS 7 based on the installation via *Python Package Index*. It follows the approach used in the enterprise packages (See *RPM Repository*).

Setting up the required services

In this guide we use Python 2.7 even though its end-of-life draws closer. CentOS 7 will support Python 2 until the end of its support frame. Basically the steps for using privacyIDEA with Python 3 are the same but several other packages need to be installed[1].

First the necessary packages need to be installed:

[1] https://stackoverflow.com/questions/42004986/how-to-install-mod-wgsi-for-apache-2-4-with-python3-5-on-centos-7

```
$ yum install mariadb-server httpd mod_wsgi mod_ssl python-virtualenv policycoreutils-
↪python
```

Now enable and configure the services:

```
$ systemctl enable --now httpd
$ systemctl enable --now mariadb
$ mysql_secure_installation
```

Setup the database for the privacyIDEA server:

```
$ echo 'create database pi;' | mysql -u root -p
$ echo 'create user "pi"@"localhost" identified by "<dbsecret>";' | mysql -u root -p
$ echo 'grant all privileges on pi.* to "pi"@"localhost";' | mysql -u root -p
```

If this should be a pinned installation (i.e. with all the package pinned to the versions with which we are developing/testing), some more packages need to be installed for building these packages:

```
$ yum install gcc postgresql-devel
```

Create the necessary directories:

```
$ mkdir /etc/privacyidea
$ mkdir /opt/privacyidea
$ mkdir /var/log/privacyidea
```

Add a dedicated user for the privacyIDEA server and change some ownerships:

```
$ useradd -r -M -d /opt/privacyidea privacyidea
$ chown privacyidea:privacyidea /opt/privacyidea /etc/privacyidea /var/log/privacyidea
```

Install the privacyIDEA server

Now switch to that user and install the virtual environment for the privacyIDEA server:

```
$ su - privacyidea
```

Create the virtual environment:

```
$ virtualenv /opt/privacyidea
```

and activate it:

```
$ . /opt/privacyidea/bin/activate
```

If this should be a pinned installation (that is the environment we use to build and test), we need to install some pinned dependencies first. They should match the version of the targeted privacyIDEA:

```
(privacyidea)$ pip install -r https://raw.githubusercontent.com/privacyidea/
↪privacyidea/v3.0.2/requirements.txt
```

Then just install the targeted privacyIDEA version with:

```
(privacyidea)$ pip install privacyidea==3.0.2
```

Setting up privacyIDEA

In order to setup privacyIDEA a configuration file must be added in `/etc/privacyidea/pi.cfg`. It should look something like this:

```
import logging
# The realm, where users are allowed to login as administrators
SUPERUSER_REALM = ['super']
# Your database
SQLALCHEMY_DATABASE_URI = 'mysql+pymysql://pi:<dbsecret>@localhost/pi'
# This is used to encrypt the auth_token
#SECRET_KEY = 't0p s3cr3t'
# This is used to encrypt the admin passwords
#PI_PEPPER = "Never know..."
# This is used to encrypt the token data and token passwords
PI_ENCFILE = '/etc/privacyidea/enckey'
# This is used to sign the audit log
PI_AUDIT_KEY_PRIVATE = '/etc/privacyidea/private.pem'
PI_AUDIT_KEY_PUBLIC = '/etc/privacyidea/public.pem'
PI_AUDIT_SQL_TRUNCATE = True
# The Class for managing the SQL connection pool
PI_ENGINE_REGISTRY_CLASS = "shared"
PI_AUDIT_POOL_SIZE = 20
PI_LOGFILE = '/var/log/privacyidea/privacyidea.log'
PI_LOGLEVEL = logging.INFO
```

Make sure the configuration file is not world readable:

```
(privacyidea)$ chmod 640 /etc/privacyidea/pi.cfg
```

More information on the configuration parameters can be found in *The Config File*.

In order to secure the installation a new `PI_PEPPER` and `SECRET_KEY` must be generated:

```
(privacyidea)$ PEPPER="$(tr -dc A-Za-z0-9_ </dev/urandom | head -c24)"
(privacyidea)$ echo "PI_PEPPER = '$PEPPER'" >> /etc/privacyidea/pi.cfg
(privacyidea)$ SECRET="$(tr -dc A-Za-z0-9_ </dev/urandom | head -c24)"
(privacyidea)$ echo "SECRET_KEY = '$SECRET'" >> /etc/privacyidea/pi.cfg
```

From now on the `pi-manage`-tool can be used to configure and manage the privacyIDEA server:

```
(privacyidea)$ pi-manage create_enckey  # encryption key for the database
(privacyidea)$ pi-manage create_audit_keys  # key for verification of audit log␣
↪entries
(privacyidea)$ pi-manage createdb  # create the database structure
(privacyidea)$ pi-manage db stamp head -d /opt/privacyidea/lib/privacyidea/migrations/
↪  # stamp the db
```

An administrative account is needed to configure and maintain privacyIDEA:

```
(privacyidea)$ pi-manage admin add <admin-user>
```

Setting up the Apache webserver

Now We need to set up apache to forward requests to privacyIDEA, so the next steps are executed as the `root`-user again.

First the SELinux settings must be adjusted in order to allow the `httpd`-process to access the database and write to the privacyIDEA logfile:

```
$ semanage fcontext -a -t httpd_sys_rw_content_t "/var/log/privacyidea(/.*)?"
$ restorecon -R /var/log/privacyidea
```

and:

```
$ setsebool -P httpd_can_network_connect_db 1
```

If the user store is an LDAP-resolver, the `httpd`-process also needs to access the ldap ports:

```
$ setsebool -P httpd_can_connect_ldap 1
```

If something does not seem right, check for "`denied`" entries in `/var/log/audit/audit.log`

For testing purposes we use a self-signed certificate which should already have been created. In production environments this should be replaced by a certificate from a trusted authority.

To correctly load the apache config file for privacyIDEA we need to disable some configuration first:

```
$ cd /etc/httpd/conf.d
$ mv ssl.conf ssl.conf.inactive
$ mv welcome.conf welcome.conf.inactive
$ curl -O https://raw.githubusercontent.com/NetKnights-GmbH/centos7/master/SOURCES/
↪privacyidea.conf
```

And we need a corresponding `wsgi`-script file in `/etc/privacyidea/`:

```
$ cd /etc/privacyidea
$ curl -O https://raw.githubusercontent.com/NetKnights-GmbH/centos7/master/SOURCES/
↪privacyideaapp.wsgi
```

After a restart of the apache webserver (`$ systemctl restart httpd`) everything should be up and running. You can log in with Your admin user at `https://<privacyidea server>` and start enrolling tokens.

RPM Repository

For customers with a valid service level agreement[2] with NetKnights there is an RPM repository, that can be used to easily install and update privacyIDEA on CentOS 7 / RHEL 7. For more information see[3].

[2] https://netknights.it/en/leistungen/service-level-agreements/

[3] https://netknights.it/en/additional-service-privacyidea-support-customers-centos-7-repository/

1.2.4 Upgrading

In any case before upgrading a major version read the document READ_BEFORE_UPDATE which is continuously updated in the Github repository. Note, that when you are upgrading over several major versions, read all the comments for all versions.

If you installed privacyIDEA via DEB or RPM repository you can use the normal system ways of *apt-get*, *aptitude* and *yum* to upgrade privacyIDEA to the current version.

If you want to upgrade an old Ubuntu installation from privacyIDEA 2.23 to privacyIDEA 3.0, please read upgrade_packaged_legacy.

Different upgrade processes

Depending on the way privacyIDEA was installed, there are different recommended update procedures. The following section describes the process for pip installations. Instructions for packaged versions on RHEL and Ubuntu are found in *Upgrading a packaged installation*.

Upgrading a pip installation

If you install privacyIDEA into a python virtualenv like */opt/privacyidea*, you can follow this basic upgrade process.

First you might want to backup your program directory:

```
tar -zcf privacyidea-old.tgz /opt/privacyidea
```

and your database:

```
source /opt/privacyidea/bin/activate
pi-manage backup create
```

Running upgrade

Starting with version 2.17 the script `privacyidea-pip-update` performs the update of the python virtualenv and the DB schema.

Just enter your python virtualenv (you already did so, when running the backup) and run the command:

> privacyidea-pip-update

The following parameters are allowed:

`-f` or `--force` skips the safety question, if you really want to update.

`-s` or `--skipstamp` skips the version stamping during schema update.

`-n` or `--noshema` completely skips the schema update and only updates the code.

Manual upgrade

Now you can upgrade the installation:

```
source /opt/privacyidea/bin/activate
pip install --upgrade privacyidea
```

Usually you will need to upgrade/migrate the database:

```
privacyidea-schema-upgrade /opt/privacyidea/lib/privacyidea/migrations
```

Now you need to restart your webserver for the new code to take effect.

Upgrading a packaged installation

In general, the upgrade of a packaged version of privacyIDEA should be done using the default tools (e.g. apt and yum). In any case, read the READ_BEFORE_UPDATE file. It is also a good idea to backup your system before upgrading.

Ubuntu upgrade

If you use the Ubuntu packages in a default setup, the upgrade can should be done using:

```
apt update
apt dist-upgrade
```

Note: In case you upgrade from the old privacyIDEA 2.23.x to the version 3.x you have to change from your ppa sources to the new repositories. If you are upgrading your Ubuntu release, e.g. from 14.04 to 16.04 the principal steps are

- Bring your Ubuntu 14.04 system up-to-date
- Run the release upgrade (do-release-upgrade)
- Eventually remove old repositories and add recent repositories as described in *Add repository*.
- Reinstall/Upgrade privacyIDEA 3.x

privacyIDEA 2.x installed the python packages to the system directly. The packages in the repository instead come with a virtual python environment. This may cause lots of obsolete packages after upgrading which may be removed with:

```
apt autoremove
```

CentOS upgrade

For a Red Hat Enterprise Linux (RHEL) installation run:

```
yum update
```

to upgrade.

1.2.5 The Config File

privacyIDEA reads its configuration from different locations:

1. default configuration from the module `privacyidea/config.py`
2. then from the config file `/etc/privacyidea/pi.cfg` if it exists and then
3. from the file specified in the environment variable `PRIVACYIDEA_CONFIGFILE`.

   export PRIVACYIDEA_CONFIGFILE=/your/config/file

The configuration is overwritten and extended in each step. I.e. values define in `privacyidea/config.py` that are not redefined in one of the other config files, stay the same.

You can create a new config file (either `/etc/privacyidea/pi.cfg`) or any other file at any location and set the environment variable. The file should contain the following contents:

```
# The realm, where users are allowed to login as administrators
SUPERUSER_REALM = ['super', 'administrators']
# Your database
SQLALCHEMY_DATABASE_URI = 'sqlite:////etc/privacyidea/data.sqlite'
# This is used to encrypt the auth_token
SECRET_KEY = 't0p s3cr3t'
# This is used to encrypt the admin passwords
PI_PEPPER = "Never know..."
# This is used to encrypt the token data and token passwords
PI_ENCFILE = '/etc/privacyidea/enckey'
# This is used to sign the audit log
PI_AUDIT_KEY_PRIVATE = '/home/cornelius/src/privacyidea/private.pem'
PI_AUDIT_KEY_PUBLIC = '/home/cornelius/src/privacyidea/public.pem'
# PI_AUDIT_MODULE = <python audit module>
# PI_AUDIT_SQL_URI = <special audit log DB uri>
# PI_LOGFILE = '....'
# PI_LOGLEVEL = 20
# PI_INIT_CHECK_HOOK = 'your.module.function'
# PI_CSS = '/location/of/theme.css'
# PI_UI_DEACTIVATED = True
```

Note: The config file is parsed as python code, so you can use variables to set the path and you need to take care for indentations.

`SQLALCHEMY_DATABASE_URI` defines the location of your database. You may want to use the MySQL database or Maria DB. There are two possible drivers, to connect to this database. Please read *MySQL database connect string*.

The `SUPERUSER_REALM` is a list of realms, in which the users get the role of an administrator.

`PI_INIT_CHECK_HOOK` is a function in an external module, that will be called as decorator to `token/init` and `token/assign`. This function takes the `request` and `action` (either "init" or "assing") as an arguments and can modify the request or raise an exception to avoid the request being handled.

There are three config entries, that can be used to define the logging. These are `PI_LOGLEVEL`, `PI_LOGFILE`, `PI_LOGCONFIG`. These are described in *Debugging and Logging*.

You can use `PI_CSS` to define the location of another cascading style sheet to customize the look and feel. Read more at *Themes*.

Note: If you ever need passwords being logged in the log file, you may set `PI_LOGLEVEL = 9`, which is a lower log level than `logging.DEBUG`. Use this setting with caution and always delete the logfiles!

privacyIDEA digitally signs the responses. You can disable this using the parameter `PI_NO_RESPONSE_SIGN`. Set this to *True* to suppress the response signature.

You can set `PI_UI_DEACTIVATED = True` to deactivate the privacyIDEA UI. This can be interesting if you are only using the command line client or your own UI and you do not want to present the UI to the user or the outside world.

Note: The API calls are all still accessible, i.e. privacyIDEA is technically fully functional.

Engine Registry Class

The `PI_ENGINE_REGISTRY_CLASS` option controls the pooling of database connections opened by SQL resolvers and the SQL audit module. If it is set to `"null"`, SQL connections are not pooled at all and new connections are opened for every request. If it is set to `"shared"`, connections are pooled on a per-process basis, i.e. every wsgi process manages one connection pool for each SQL resolver and the SQL audit module. Every request then checks out connections from this shared pool, which reduces the overall number of open SQL connections. If the option is left unspecified, its value defaults to `"null"`.

Audit parameters

`PI_AUDIT_MODULE` lets you specify an alternative auditing module. The default which is shipped with privacyIDEA is `privacyidea.lib.auditmodules.sqlaudit`. There is no need to change this, unless you know exactly what you are doing.

You can change the servername of the privacyIDEA node, which will be logged to the audit log using the variable `PI_AUDIT_SERVERNAME`.

You can run the database for the audit module on another database or even server. For this you can specify the database URI via `PI_AUDIT_SQL_URI`.

`PI_AUDIT_TRUNCATE = True` lets you truncate audit entries, that to the length of the database fields.

In certain cases when you experiencing problems you may use the parameters `PI_AUDIT_POOL_SIZE` and `PI_AUDIT_POOL_RECYCLE`. However, they are only effective if you also set `PI_ENGINE_REGISTRY_CLASS` to `"shared"`.

If you by any reason want to avoid signing audit entries you can set `PI_AUDIT_NO_SIGN = True`. If `PI_AUDIT_NO_SIGN` is set to `True` audit entries will not be signed and also the signature of audit entries will not be verified. Audit entries will appears with *signature fail*.

Monitoring parameters

PI_MONITORING_MODULE lets you specify an alternative statistics monitoring module. The monitoring module takes care of writing values with timestamps to a store. This is used e.g. by the *EventCounter* and *SimpleStats*.

The first available monitoring module is privacyidea.lib.monitoringmodules.sqlstats. It accepts the following additional parameters:

PI_MONITORING_SQL_URI can hold an alternative SQL connect string. If not specified the normal SQLALCHEMY_DATABASE_URI is used.

PI_MONITORING_POOL_SIZE (default 20) and PI_MONITORING_POOL_RECYCLE (default 600) let you configure pooling. It uses the settings from the above mentioned PI_ENGINE_REGISTRY_CLASS.

Note: A SQL database is probably not the best database to store time series. Other monitoring modules will follow.

privacyIDEA Nodes

privacyIDEA can run in a redundant setup. For statistics and monitoring purposes you can give these different nodes, dedicated names.

PI_NODE is a string with the name of this very node. PI_NODES is a list of all available nodes in the cluster.

If PI_NODE is not set, then PI_AUDIT_SERVERNAME is used as node name. If this is also not set, the node name is returned as “localnode”.

Trusted JWTs

Other applications can use the API without the need to call the /auth endpoint. This can be achieved by trusting private RSA keys to sign JWTs. You can define a list of corresponding public keys that are trusted for certain users and roles using the parameter PI_TRUSTED_JWT:

```
PI_TRUSTED_JWT = [{"public_key": "-----BEGIN PUBLIC KEY-----\
↪nMIIBIjANBgkqhkiG9w0BAQEF...",
                    "algorithm": "RS256",
                    "role": "user",
                    "realm": "realm1",
                    "username": "userA",
                    "resolver": "resolverX"}]
```

This entry means, that the private key, that corresponds to the given public key can sign a JWT, that can impersonate as the *userA* in resolver *resolverX* in *realmA*.

Note: The username can be a regular expression like “.*”. This way you could allow a private signing key to impersonate every user in a realm. (Starting with version 3.3)

A JWT can be created like this:

```
auth_token = jwt.encode(payload={"role": "user",
                                 "username": "userA",
                                 "realm": "realm1",
                                 "resolver": "resolverX"},
                                 key=private_key,
                                 algorithm="RS256")
```

Note: The user and the realm do not necessarily need to exist in any resolver! But there probably must be certain policies defined for this user. If you are using an administrative user, the realm for this administrative must be defined in `pi.cfg` in the list `SUPERUSER_REALM`.

1.2.6 Debugging and Logging

You can set `PI_LOGLEVEL` to a value 10 (Debug), 20 (Info), 30 (Warning), 40 (Error) or 50 (Critical). If you experience problems, set `PI_LOGLEVEL = 10` restart the web service and resume the operation. The log file `privacyidea.log` should contain some clues.

You can define the location of the logfile using the key `PI_LOGFILE`. Usually it is set to:

```
PI_LOGFILE = "/var/log/privacyidea/privacyidea.log"
```

Advanced Logging

You can also define a more detailed logging by specifying a log configuration file. By default the file is `/etc/privacyidea/logging.cfg`.

You can change the location of the logging configuration file in *The Config File* like this:

```
PI_LOGCONFIG = "/path/to/logging.yml"
```

Since Version 3.3 the logging configuration can be written in YAML[1]. Such a YAML based configuration could look like this:

```
version: 1
formatters:
  detail:
    class: privacyidea.lib.log.SecureFormatter
    format: '[%(asctime)s][%(process)d][%(thread)d][%(levelname)s][%(name)s:
↪%(lineno)d] %(message)s'

handlers:
  mail:
    class: logging.handlers.SMTPHandler
    mailhost: mail.example.com
    fromaddr: privacyidea@example.com
    toaddrs:
    - admin1@example.com
    - admin2@example.com
    subject: PI Error
    formatter: detail
    level: ERROR
  file:
    # Rollover the logfile at midnight
    class: logging.handlers.RotatingFileHandler
    backupCount: 5
    maxBytes: 1000000
    formatter: detail
    level: INFO
```

(continues on next page)

[1] https://yaml.org/

(continued from previous page)

```
    filename: /var/log/privacyidea/privacyidea.log
loggers:
  # The logger name is the qualname
  privacyidea:
    handlers:
    - file
    - mail
    level: INFO
root:
  level: WARNING
```

Different handlers can be used to send log messages to log-aggregators like splunk[2] or logstash[3].

The old python logging config file format is also still supported:

```
[formatters]
keys=detail

[handlers]
keys=file,mail

[formatter_detail]
class=privacyidea.lib.log.SecureFormatter
format=[%(asctime)s][%(process)d][%(thread)d][%(levelname)s][%(name)s:%(lineno)d]
↪%(message)s

[handler_mail]
class=logging.handlers.SMTPHandler
level=ERROR
formatter=detail
args=('mail.example.com', 'privacyidea@example.com', ['admin1@example.com',\
   'admin2@example.com'], 'PI Error')

[handler_file]
# Rollover the logfile at midnight
class=logging.handlers.RotatingFileHandler
backupCount=14
maxBytes=10000000
formatter=detail
level=DEBUG
args=('/var/log/privacyidea/privacyidea.log',)

[loggers]
keys=root,privacyidea

[logger_privacyidea]
handlers=file,mail
qualname=privacyidea
level=DEBUG

[logger_root]
level=ERROR
handlers=file
```

[2] https://www.splunk.com/

[3] https://www.elastic.co/logstash

Note: These examples define a mail handler, that will send emails to certain email addresses, if an ERROR occurs. All other DEBUG messages will be logged to a file.

Note: The filename extension is irrelevant in this case

1.2.7 The WSGI Script

Apache2 and Nginx are using a WSGI script to start the application.

This script is usually located at `/etc/privacyidea/privacyideaapp.py` or `/etc/privacyidea/privacyideaapp.wsgi` and has the following contents:

```
import sys
sys.stdout = sys.stderr
from privacyidea.app import create_app
# Now we can select the config file:
application = create_app(config_name="production",
                         config_file="/etc/privacyidea/pi.cfg")
```

In the `create_app`-call you can also select another config file.

Note: This way you can run several instances of privacyIDEA in one Apache2 server by defining several WSGIScriptAlias definitions pointing to different wsgi-scripts, that again reference different config files with different database definitions.

Running Apache instances

To run further Apache instances add additional lines in your Apache config:

```
WSGIScriptAlias /instance1 /etc/privacyidea1/privacyideaapp.wsgi
WSGIScriptAlias /instance2 /etc/privacyidea2/privacyideaapp.wsgi
WSGIScriptAlias /instance3 /etc/privacyidea3/privacyideaapp.wsgi
WSGIScriptAlias /instance4 /etc/privacyidea4/privacyideaapp.wsgi
```

It is a good idea to create a subdirectory in */etc* for each instance. Each wsgi script needs to point to the corresponding config file *pi.cfg*.

Each config file can define its own

- database
- encryption key
- signing key
- . . .

To create the new database you need the command *pi-manage*. The command *pi-manage* reads the configuration from */etc/privacyidea/pi.cfg*.

If you want to use another instance with another config file, you need to set an environment variable and create the database like this:

```
PRIVACYIDEA_CONFIGFILE=/etc/privacyidea3/pi.cfg pi-manage createdb
```

This way you can use *pi-manage* for each instance.

1.2.8 The pi-manage Script

pi-manage is the script that is used during the installation process to setup the database and do many other tasks.

Note: The interesting thing about pi-manage is, that it does not need the server to run as it acts directly on the database. Therefor you need read access to /etc/privacyidea/pi.cfg and the encryption key.

If you want to use a config file other than /etc/privacyidea/pi.cfg, you can set an environment variable:

```
PRIVACYIDEA_CONFIGFILE=/home/user/pi.cfg pi-manage
```

pi-manage always takes a command and sometimes a sub command:

```
pi-manage <command> [<subcommand>] [<parameters>]
```

For a complete list of commands and sub commands use the *-h* parameter.

You can do the following tasks.

Encryption Key

You can create an encryption key and encrypt the encryption key.

Create encryption key:

```
pi-manage create_enckey [--enckey_b64=BASE64_ENCODED_ENCKEY]
```

Note: The filename of the encryption key is read from the configuration. The key will not be created, if it already exists. Optionally, enckey can be passed via *–enckey_b64* argument, but it is not recommended. *–enckey_b64* must be a string with 96 bytes, encoded in base 64 in order to avoid ambiguous chars.

The encryption key is a plain file on your hard drive. You need to take care, to set the correct access rights.

You can also encrypt the encryption key with a passphrase. To do this do:

```
pi-manage encrypt_enckey /etc/privacyidea/enckey
```

and pipe the encrypted *enckey* to a new file.

Read more about the database encryption and the enckey in *Security Modules*.

Backup and Restore

You can create a backup which will be save to */var/lib/privacyidea/backup/*.

The backup will contain the database dump and the complete directory */etc/privacyidea*. You may choose if you want to add the encryption key to the backup or not.

Warning: If the backup includes the database dump and the encryption key all seeds of the OTP tokens can be read from the backup.

As the backup contains the etc directory and the database you only need this tar archive backup to perform a complete restore.

Rotate Audit Log

Audit logs are written to the database. You can use pi-manage to perform a log rotation.

> pi-manage rotate_audit

You can specify a highwatermark and a lowwatermark, age or a config file. Read more about it at *Cleaning up entries*.

API Keys

You can use `pi-manage` to create API keys. API keys can be used to

1. secure the access to the `/validate/check` API or
2. to access administrative tasks via the REST API.

You can create API keys for `/validate/check` using the command

> pi-manage api createtoken -r validate

If you want to secure the access to `/validate/check` you also need to define a policy in scope `authorizaion`. See *api_key_required*.

If you wan to use the API key to automate administrative REST API calls, you can use the command:

> pi-manage api createtoken -r admin

This command also generates an admin account name. But it does not create this admin account. You need to do so using `pi-manage admin`. You can now use this API key to enroll tokens as administrator.

Note: These API keys are not persistent. They are not stored in the privacyIDEA server. The API key is connected to the username, that is also generated. This means you have to create an administrative account with this very username to use this API key for this admin user. You also should set policies for this admin user, so that this API key has only restricted rights!

Note: The API key is valid for 365 days.

Policies

You can use `pi-manage policy` to enable, disable, create and delete policies. Using the sub commands `p_export` and `p_import` you can also export a backup of your policies and import this policy set later.

This could also be used to transfer the policies from one privacyIDEA instance to another.

1.2.9 Security Modules

Note: For a normal installation this section can be safely ignored.

privacyIDEA provides a security module that takes care of

- encrypting the token seeds,
- encrypting passwords from the configuration like the LDAP password,
- creating random numbers,
- and hashing values.

Note: The Security Module concept can also be used to add a Hardware Security Module to perform the above mentioned tasks.

Default Security Module

The `default` security module is implemented with the operating systems capabilities. The encryption key is located in a file *enckey* specified via `PI_ENCFILE` in the configuration file (*The Config File*).

This *enckey* contains three 32byte keys and is thus 96 bytes. This file has to be protected. So the access rights to this file are set accordingly.

In addition you can encrypt this encryption key with an additional password. In this case, you need to enter the password each time the privacyIDEA server is restarted and the password for decrypting the *enckey* is kept in memory.

The pi-manage Script contains the instruction how to encrypt the *enckey*

After starting the server, you can check, if the encryption key is accessible. To do so run:

```
privacyidea -U <yourserver> --admin=<youradmin> securitymodule
```

The output will contain `"is_ready": True` to signal that the encryption key is operational.

If it is not yet operational, you need to pass the password to the privacyIDEA server to decrypt the encryption key. To do so run:

```
privacyidea -U <yourserver> --admin=<youradmin> securitymodule  \
--module=default
```

Note: If the security module is not operational yet, you might get an error message “HSM not ready.”.

PKCS11 Security Module

The PKCS11 Security Module can be used to encrypt data with an hardware security module, that is connected via the PKCS11 interface. To encrypt and decrypt data you can use an RSA key pair that is stored on the HSM.

To activate this module add the following to the configuration file (*The Config File*)

> PI_HSM_MODULE = "privacyidea.lib.security.pkcs11.PKCS11SecurityModule"

Additional attributes are

`PI_HSM_MODULE_MODULE` which takes the pkcs11 library. This is the full specified path to the shared object file in the file system.

`PI_HSM_MODULE_KEY_ID` is the key id (integer) on the HSM.

AES HSM Security Module

The AES Hardware Security Module can be used to encrypt data with an hardware security module (HSM) connected via the PKCS11 interface. This module allows to use AES keys stored in the HSM to encrypt and decrypt data.

This module uses three keys, similarly to the content of `PI_ENCFILE`, identified as `token`, `config` and `value`.

To activate this module add the following to the configuration file (*The Config File*)

> PI_HSM_MODULE = "privacyidea.lib.security.aeshsm.AESHardwareSecurityModule"

Additional attributes are

`PI_HSM_MODULE_MODULE` which takes the pkcs11 library. This is the full specified path to the shared object file in the file system.

`PI_HSM_MODULE_SLOT` is the slot on the HSM where the keys are located (default: `1`).

`PI_HSM_MODULE_PASSWORD` is the password to access the slot.

`PI_HSM_MODULE_MAX_RETRIES` is the number privacyIDEA tries to perform a cryptographic operation like *decrypt*, *encrypt* or *random* if the first attempt with the HSM fails. The default value is 5.

Note: Some PKCS11 libraries for network attached HSMs also implement a retry. You should take this into account, since retries would multiply and it could take a while till a request would finally fail.

`PI_HSM_MODULE_KEY_LABEL` is the label prefix for the keys on the HSM (default: `privacyidea`). In order to locate the keys, the module will search for key with a label equal to the concatenation of this prefix, _ and the key identifier (respectively `token`, `config` and `value`).

`PI_HSM_MODULE_KEY_LABEL_TOKEN` is the label for `token` key (defaults to value based on `PI_HSM_MODULE_KEY_LABEL` setting).

`PI_HSM_MODULE_KEY_LABEL_CONFIG` is the label for `config` key (defaults to value based on `PI_HSM_MODULE_KEY_LABEL` setting).

`PI_HSM_MODULE_KEY_LABEL_VALUE` is the label for `value` key (defaults to value based on `PI_HSM_MODULE_KEY_LABEL` setting).

After installation you might want to take a look at *First Steps*.

1.3 First Steps

You installed privacyIDEA successfully according to *Installation*.

These first steps will guide you through the tasks of logging in to the management web UI, attaching your first users and enrolling the first token.

1.3.1 Add an administrator

PrivacyIDEA does not come with a pre-defined administrator user. If you just installed privacyIDEA, you need to create a new one by running:

```
pi-manage admin add admin -e admin@localhost
```

To configure privacyIDEA, continue with *Login to the Web UI*.

Note: Administrator accounts are used for various purposes in privacyIDEA. Once you need another administrator user, you should consider adding an admin policy to set up the permissions correctly. This is described in *Admin policies*.

You may also read *So what's the thing with all the admins?*.

1.3.2 Login to the Web UI

privacyIDEA has only one login form that is used by administrators and normal users to login. Administrators will be able to configure the system and to manage all tokens, while normal users will only be able to manage their own tokens.

You should enter your username with the right realm. You need to append the realm to the username like `username@realm`.

Login for administrators

Administrators can authenticate at this login form to access the management UI.

Administrators are stored in the database table `Admin` and can be managed with the tool:

```
pi-manage admin ...
```

The administrator just logs in with his username.

Note: You can configure privacyIDEA to authenticate administrators against privacyIDEA itself, so that administrators need to login with a second factor. See *So what's the thing with all the admins?* how to do this.

Login for normal users

Normal users authenticate at the login form to be able to manage their own tokens. By default users need to authenticate with the password from their user source.

E.g. if the users are located in an LDAP or Active Directory the user needs to authenticate with his LDAP/AD password.

But before a user can login, the administrator needs to configure realms, which is described in the next step *Creating your first realm*.

Note: The user my either login with his password from the userstore or with any of his tokens.

Note: The administrator may change this behaviour by creating an according policy, which then requires the user to authenticate against privacyIDEA itself. I.e. this way the user needs to authenticate with a second factor/token to access the self service portal. (see the policy section *login_mode*)

1.3.3 Creating your first realm

Note: When the administrator logs in and no useridresolver and no realm is defined, a popup appears, which asks you to create a default realm. During these first steps you may say "No", to get a better understanding.

Users in privacyIDEA are read from existing sources. See *Realms* for more information.

In these first steps we will simply read the users from your */etc/passwd* file.

Create a UserIdResolver

The UserIdResolver is the connector to the user source. For more information see *UserIdResolvers*.

- Go to *Config* -> *Users* to create a UserIdResolver.

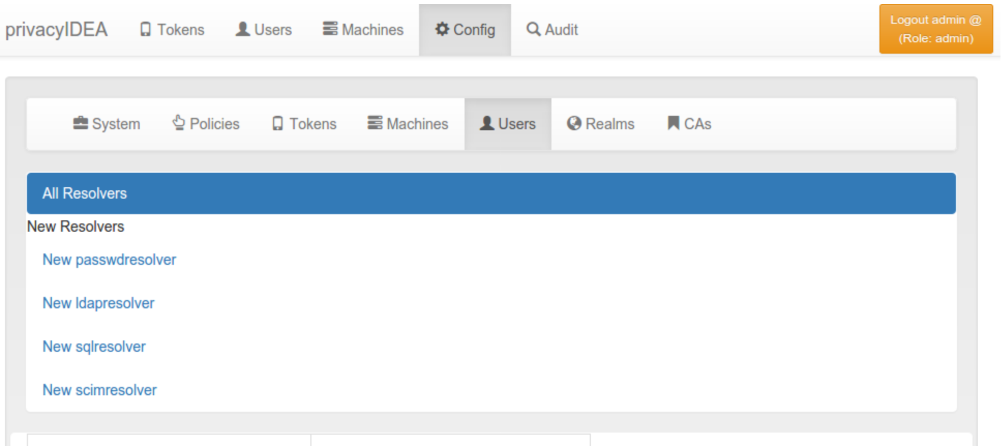
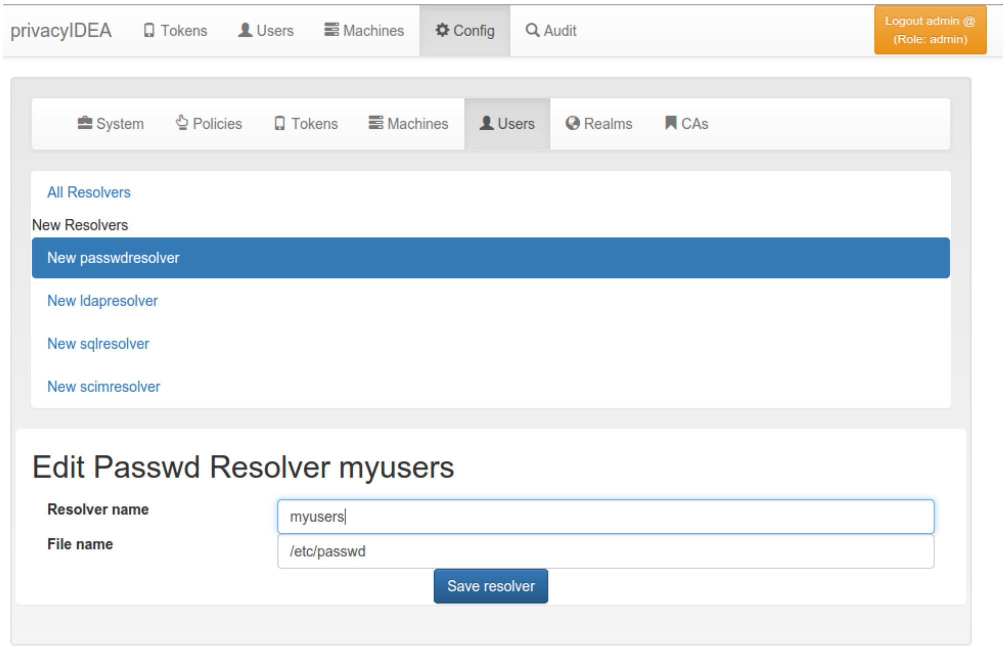


Fig. 1: *Create the first UserIdResolver*

- Choose *New passwdresolver* and

- Enter the name "myusers".
- Save it.



Edit Passwd Resolver myusers

Resolver name: myusers

File name: /etc/passwd

Fig. 2: *Create the first UserIdResolver*

You just created your first connection to a user source.

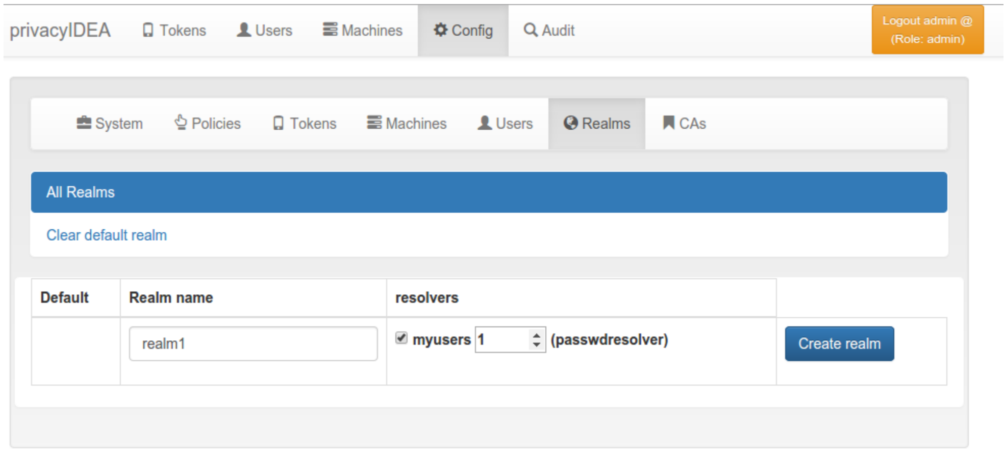
Create a Realm

User sources are grouped togeather to a so called "realm". For more information see *Realms*.

- Go to *Config -> Realms*
- Enter "realm1" as the new realm name and select the priority 1.
- Check the resolver "myusers" to be included into this realm.
- Save it.
- Go to *Users* and you will see the users from the */etc/passwd*.

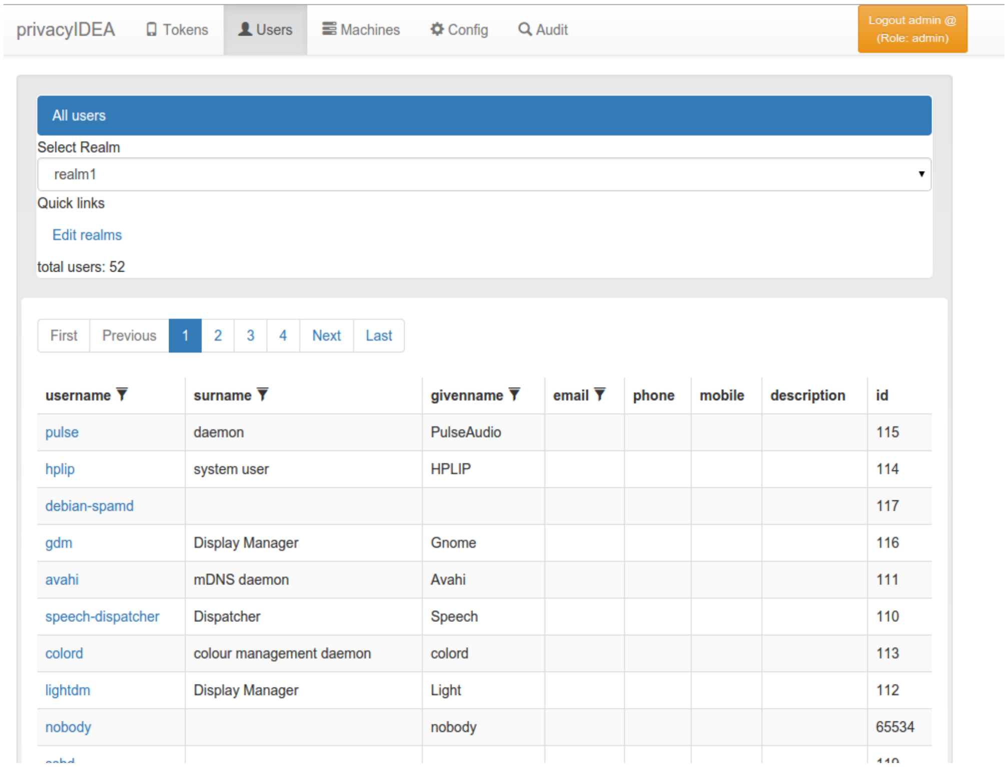
Congratulation! You created your first realm.

You are now ready to enroll a token to a user. Read *Enrolling your first token*.



Default	Realm name	resolvers
	realm1	☑ myusers 1 (passwdresolver)

Fig. 3: *Create the first Realm*



Select Realm: realm1

Quick links

total users: 52

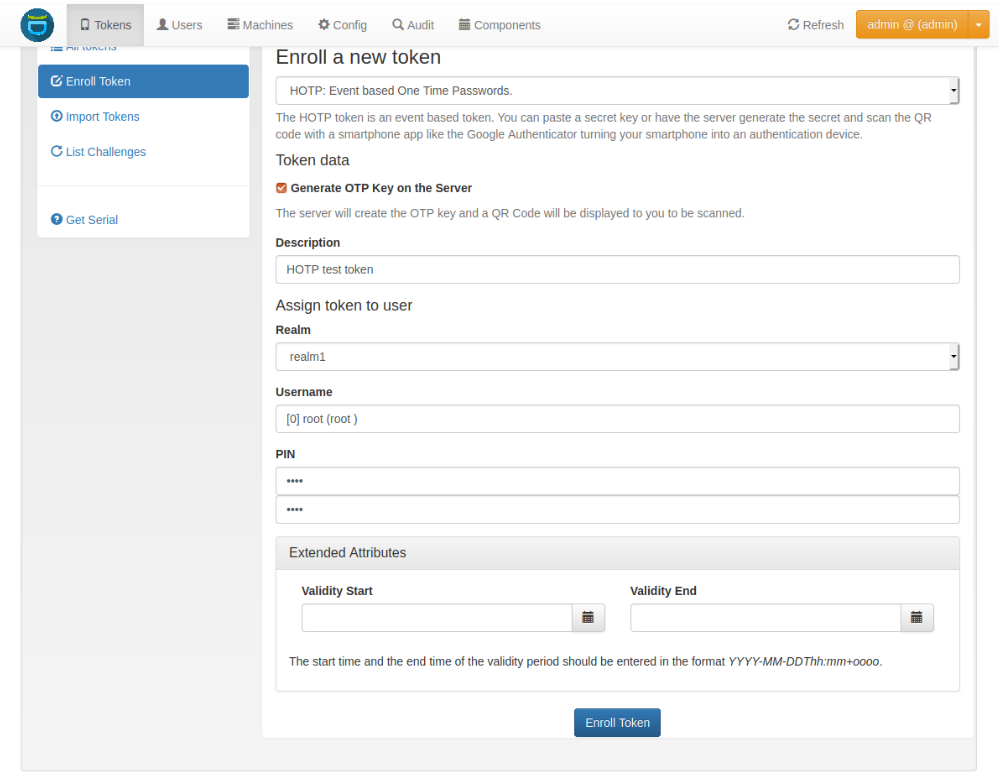
username	surname	givenname	email	phone	mobile	description	id
pulse	daemon	PulseAudio					115
hplip	system user	HPLIP					114
debian-spamd							117
gdm	Display Manager	Gnome					116
avahi	mDNS daemon	Avahi					111
speech-dispatcher	Dispatcher	Speech					110
colord	colour management daemon	colord					113
lightdm	Display Manager	Light					112
nobody		nobody					65534

Fig. 4: *The users from /etc/passwd*

1.3.4 Enrolling your first token

You may now enroll a new token. In this example we are using the Google Authenticator App, that you need to install on your smartphone.

- Go to *Tokens -> Enroll Token*



Enroll a new token

HOTP: Event based One Time Passwords.

The HOTP token is an event based token. You can paste a secret key or have the server generate the secret and scan the QR code with a smartphone app like the Google Authenticator turning your smartphone into an authentication device.

Token data

☑ **Generate OTP Key on the Server**

The server will create the OTP key and a QR Code will be displayed to you to be scanned.

Description

HOTP test token

Assign token to user

Realm

realm1

Username

[0] root (root)

PIN

••••

••••

Extended Attributes

Validity Start

Validity End

The start time and the end time of the validity period should be entered in the format *YYYY-MM-DDThh:mm+oooo*.

Fig. 5: *The Token Enrollment Dialog*

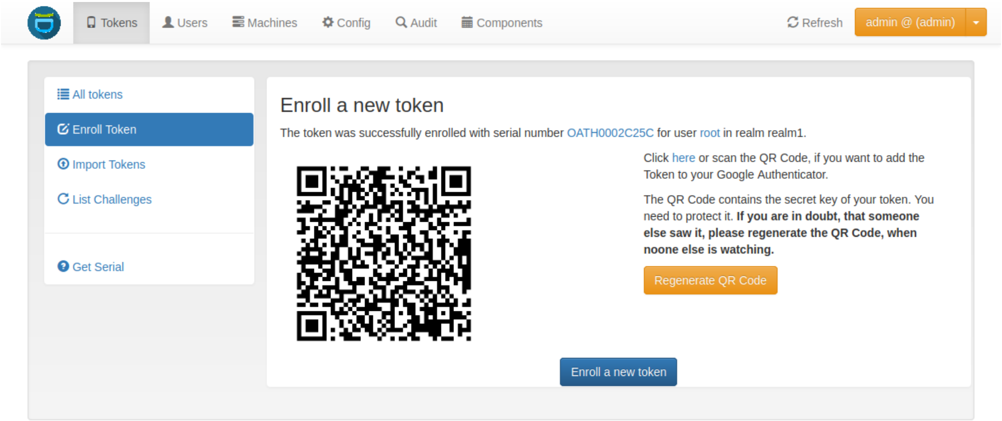
- Select the username *root*. When you start typing "r", "o"… the system will find the user root automatically.
- Enter a PIN. I entered "test" …
- … and click "Enroll Token".
- After enrolling the token you will see a QR code, that you need to scan with the Google Authenticator App.
- Click on the serial number link at the top of the dialog.
- Now you see the token details.
- Left to the button "Test Token" you can enter the PIN and the OTP value generated by the Google Authenticator.
- Click the button "Test Token". You should see a green "matching 1 tokens".

Congratulations! You just enrolled your first token to a user.

Now you are ready to attach applications to privacyIDEA in order to add two factor authentication to those applications. To attach applications read the chapter *Application Plugins*.

You may also go on reading the chapter *Configuration* to get a deeper insight in the configuration possibilities.

After these first steps you will be able to start attaching applications to privacyIDEA in order to add two factor authentication to those applications. You can



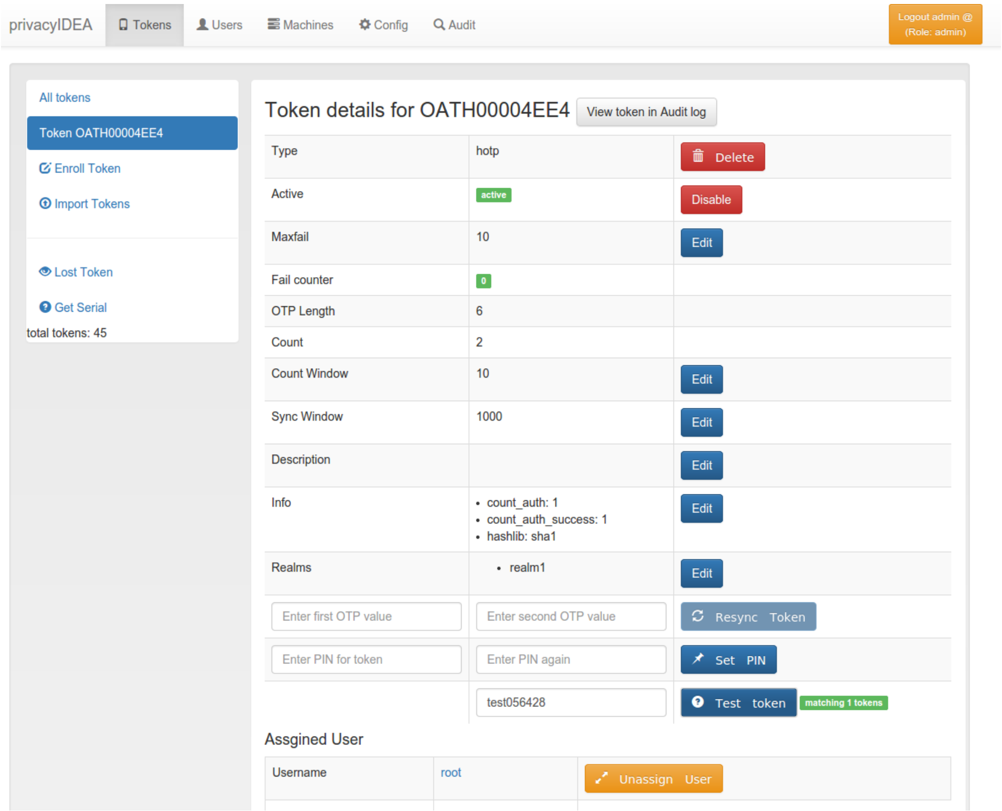
Enroll a new token

The token was successfully enrolled with serial number OATH0002C25C for user root in realm realm1.

Click here or scan the QR Code, if you want to add the Token to your Google Authenticator.

The QR Code contains the secret key of your token. You need to protect it. **If you are in doubt, that someone else saw it, please regenerate the QR Code, when noone else is watching.**

Fig. 6: *Enrollment Success*



total tokens: 45

Token details for OATH00004EE4

Type	hotp	Delete
Active	active	Disable
Maxfail	10	Edit
Fail counter	0	
OTP Length	6	
Count	2	
Count Window	10	Edit
Sync Window	1000	Edit
Description		Edit
Info	• count_auth: 1 • count_auth_success: 1 • hashlib: sha1	Edit
Realms	• realm1	Edit
Enter first OTP value	Enter second OTP value	Resync Token
Enter PIN for token	Enter PIN again	Set PIN
	test056428	Test token matching 1 tokens

Assgined User

Username	root	Unassign User

Fig. 7: *Test the Token*

- use a PAM module to authenticate with OTP at SSH or local login
- or the RADIUS plugin to configure your firewall or VPN to use OTP,
- or use an Apache2 plugin to do Basic Authentication with OTP.
- You can also setup different web applications to use OTP.

To attach applications read the chapter *Application Plugins*.

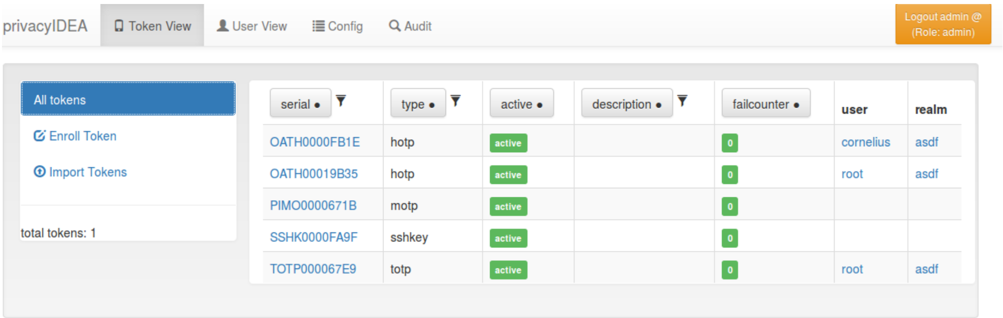
You may also go on reading the next chapter which gives an overview on the *WebUI* or you directly skip to *Configuration* to get a deeper insight in the configuration possibilities.

1.4 WebUI

privacyIDEA comes with a web-based user interface which is used to manage and configure the privacyIDEA server. It is also used a self-service portal for the average user, who manages his own tokens. This section gives an overview on the interface and links the respective sections in the documentation.

1.4.1 Tokens

The administrator can see all the tokens of all realms he is allowed to manage in the tokenview. Each token can be located in several realms and be assigned to one user. The administrator can see all the details of the token.



total tokens: 1

serial ●	type ●	active ●	description ●	failcounter ●	user	realm
OATH0000FB1E	hotp	active		0	cornelius	asdf
OATH00019B35	hotp	active		0	root	asdf
PIMO0000671B	motp	active		0		
SSHK0000FA9F	sshkey	active		0		
TOTP000067E9	totp	active		0	root	asdf

Fig. 8: *Tokens overview*

The administrator can click on one token, to show more details of this token and to perform actions on this token. Read on in token_details.

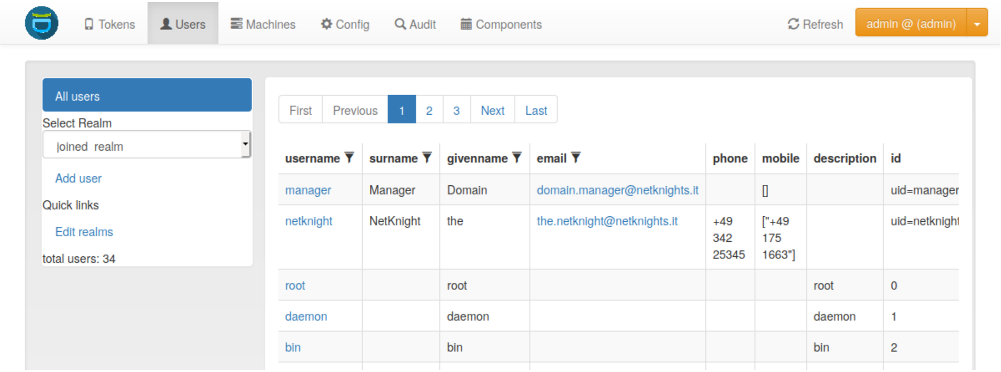
1.4.2 Users

The administrator can see all users fetched by *UserIdResolvers* located in *Realms* he is allowed to manage.

Note: Users are only visible, if the useridresolver is located within a realm. If you only define a useridresolver but no realm, you will not be able to see the users!

You can select one of the realms in the left drop down box. The administrator will only see the realms in the drop down box, that he is allowed to manage.

The list shows the users from the select realm. The username, surname, given name, email and phone are filled according to the definition of the useridresolver.



total users: 34

username	surname	givenname	email	phone	mobile	description	id
manager	Manager	Domain	domain.manager@netknights.it		[]		uid=manager
netknight	NetKnight	the	the.netknight@netknights.it	+49 342 25345	["+49 175 1663"]		uid=netknight
root		root				root	0
daemon		daemon				daemon	1
bin		bin				bin	2

Fig. 9: *The Users view list all users in a realm.*

Even if a realm contains several useridresolvers all users from all resolvers within this realm are displayed.

Read about the functionality of the users view in the following sections.

User Details

When clicking on a username, you can see the users details and perform several actions on the user.

You see a list of the users tokens and change to the token_details.

Enroll tokens

In the users details view you can enroll additional tokens to the user. In the enrollment dialog the user will be selected and you only need to choose what tokentype you wish to enroll for this user.

Assign tokens

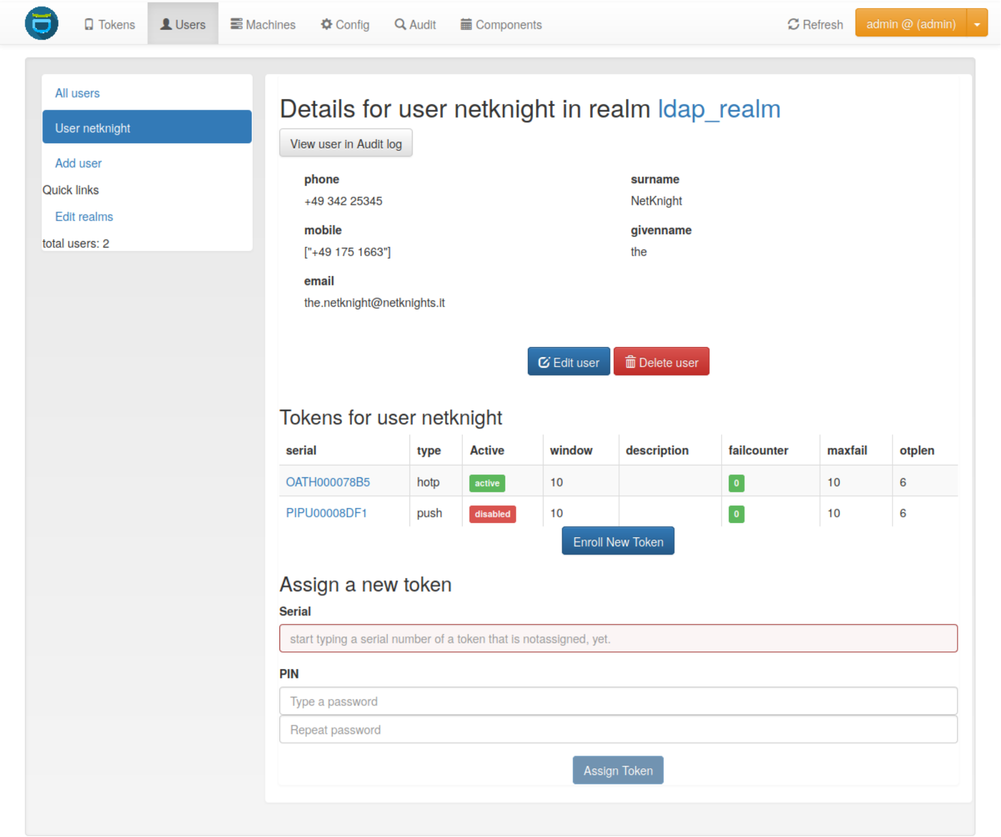
You can assign a new, already existing token to the user. Just start typing the token serial number. The system will search for tokens, that are not assigned yet and present you a list to choose from.

View Audit Log

You can also click *View user in Audit log* which will take you to the *Audit* log with a filter on this very user, so that you will only see audit entries regarding this user.

Edit user

If the user is located in a resolver, that is marked as editable, the administrator will also see a button “Edit User”. To read more about this, see *Manage Users*.



total users: 2

Details for user netknight in realm ldap_realm

phone +49 342 25345	**surname** NetKnight
mobile ["+49 175 1663"]	**givenname** the
email the.netknight@netknights.it	

Tokens for user netknight

serial	type	Active	window	description	failcounter	maxfail	otplen
OATH000078B5	hotp	active	10		0	10	6
PIPU00008DF1	push	disabled	10		0	10	6

Assign a new token

Serial

start typing a serial number of a token that is notassigned, yet.

PIN

Type a password

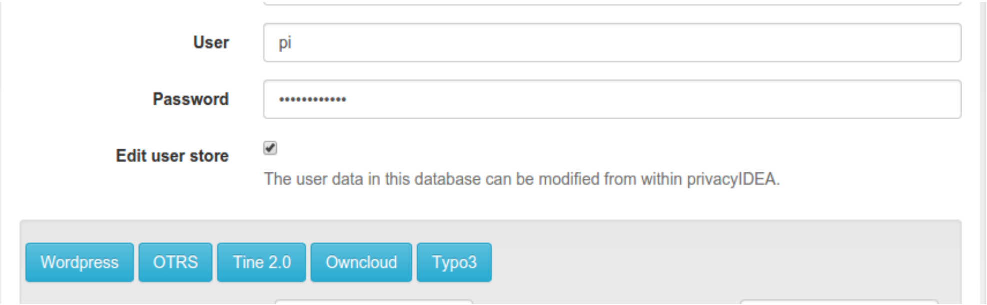
Repeat password

Fig. 10: *User Details.*

Manage Users

Since version 2.4 privacyIDEA allows you to edit users in the configured resolvers. At the moment this is possible for SQL resolvers.

In the resolver definition you need to check the new checkbox **Edit user store**.



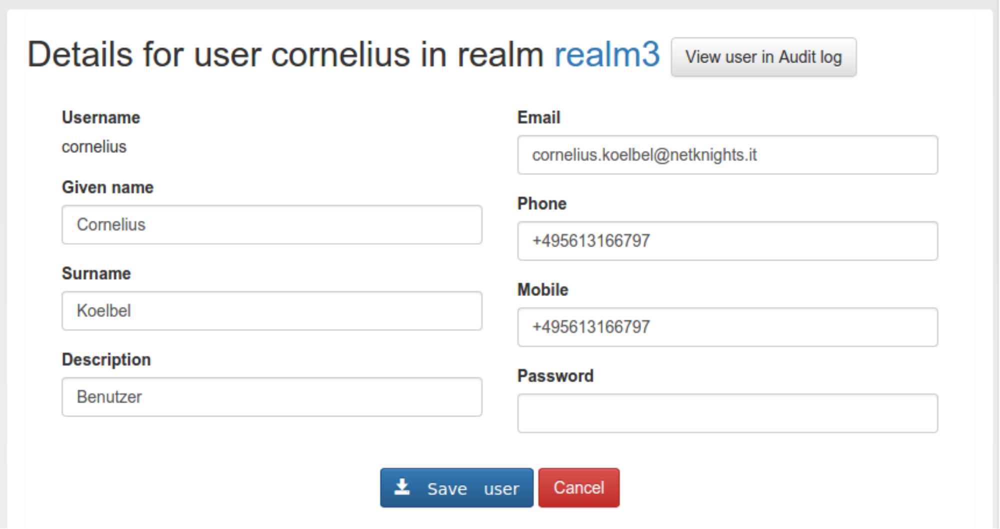
User pi

Password ••••••••••••

Edit user store ☑ The user data in this database can be modified from within privacyIDEA.

Fig. 11: *Users in SQL can be edited, when checking the checkbox.*

In the Users Detail view, the administrator then can click the button "Edit" and modify the user data and also set a new password.



Details for user cornelius in realm realm3

Username cornelius

Given name Cornelius

Surname Koelbel

Description Benutzer

Email cornelius.koelbel@netknights.it

Phone +495613166797

Mobile +495613166797

Password

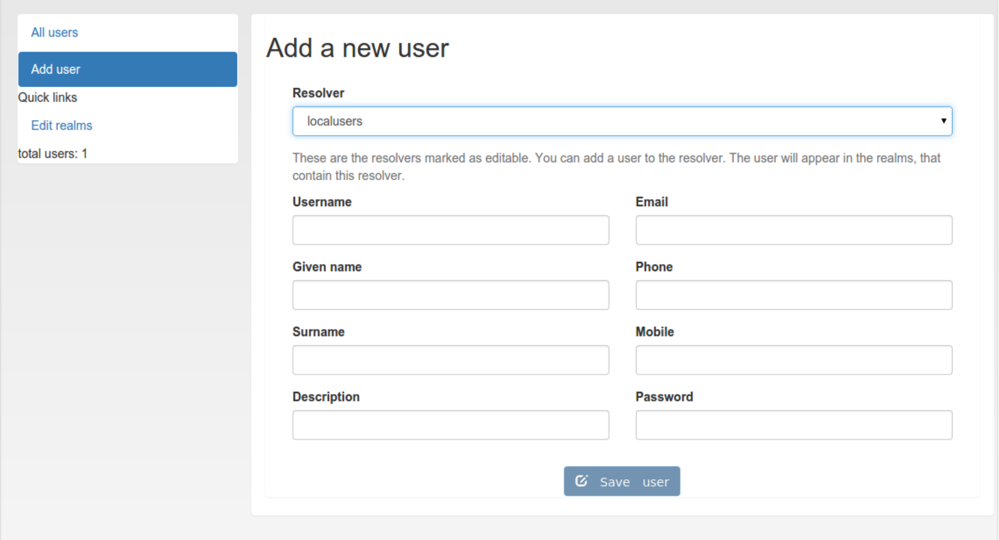
Fig. 12: *Edit the attributes of an existing user.*

Note: The data of the user will be modified in the user store (database). Thus the users data, which will be returned by a resolver, is changed. If the resolver is contained in several realms these changes will reflect in all realms.

If you want to add a user, you can click on *Add User* in the *User View*.

Users are contained in resolvers and added to resolvers. So you need to choose an existing resolver and not a realm. The user will be visible in all realms, the resolver is contained in.

Note: Of course you can set policies to allow or deny the administrator these rights.



Add a new user

Resolver: localusers

These are the resolvers marked as editable. You can add a user to the resolver. The user will appear in the realms, that contain this resolver.

Fig. 13: *Add a new user.*

Simple local users setup

You can setup a local users definition quite easily. Run:

```
pi-manage resolver create_internal test
```

This will create a database table "users_test" in your token database. And it will create a resolver "test" that refers to this database table.

Then you can add this resolver to realm:

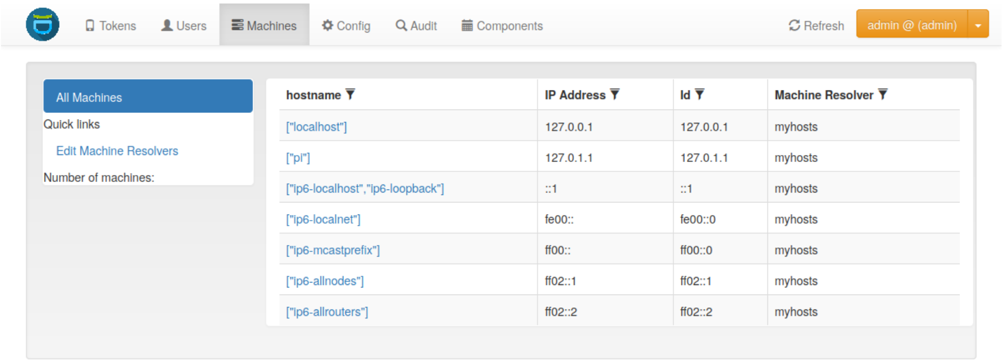
```
pi-manage realm create internal_realm test
```

Which will create a realm "internal_realm" containing the resolver "test". Now you can start adding users to this resolver as described above.

Note: This is an example of how to get started with users quite quickly. Of course you do not need to save the users table in the same database as the tokens. But in scenarios, where you do not have existing user stores or the user stores are managed by another department or are not accessible easily this may be sensible way.

1.4.3 Machines

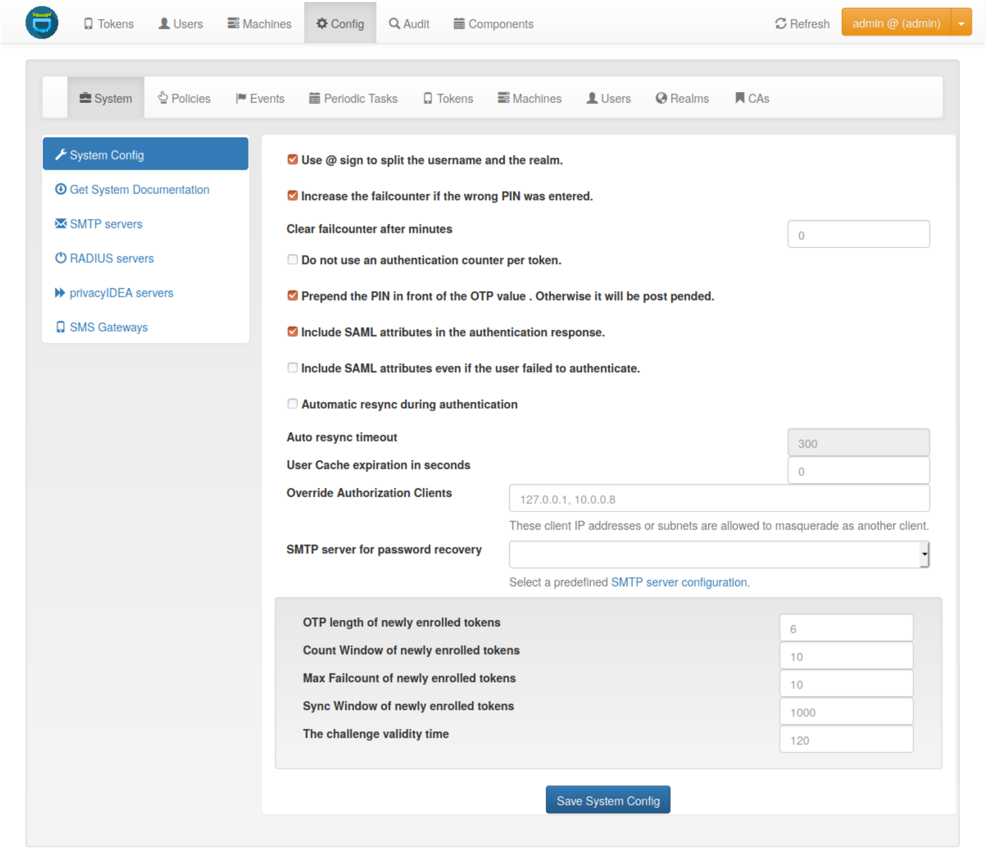
In this view *Machines* are listed which are fetched by the configured machine resolvers. Machines are only necessary if you plan special use cases like managing SSH keys or doing offline OTP. In most cases there is no need to manage machines and this view is empty.



hostname	IP Address	Id	Machine Resolver
["localhost"]	127.0.0.1	127.0.0.1	myhosts
["pi"]	127.0.1.1	127.0.1.1	myhosts
["ip6-localhost","ip6-loopback"]	::1	::1	myhosts
["ip6-localnet"]	fe00::	fe00::0	myhosts
["ip6-mcastprefix"]	ff00::	ff00::0	myhosts
["ip6-allnodes"]	ff02::1	ff02::1	myhosts
["ip6-allrouters"]	ff02::2	ff02::2	myhosts

1.4.4 Config

The configuration tab is the heart of the privacyIDEA server. It contains the general *System Config*, allows configuring *Policies* which are important to configure behavior of the system, manages the *Event Handler* and lets the user set up *Periodic Tasks*.



☑ Use @ sign to split the username and the realm.

☑ Increase the failcounter if the wrong PIN was entered.

Clear failcounter after minutes: 0

□ Do not use an authentication counter per token.

☑ Prepend the PIN in front of the OTP value . Otherwise it will be post pended.

☑ Include SAML attributes in the authentication response.

□ Include SAML attributes even if the user failed to authenticate.

□ Automatic resync during authentication

Auto resync timeout: 300

User Cache expiration in seconds: 0

Override Authorization Clients: 127.0.0.1, 10.0.0.8

These client IP addresses or subnets are allowed to masquerade as another client.

SMTP server for password recovery:

Select a predefined SMTP server configuration.

OTP length of newly enrolled tokens: 6

Count Window of newly enrolled tokens: 10

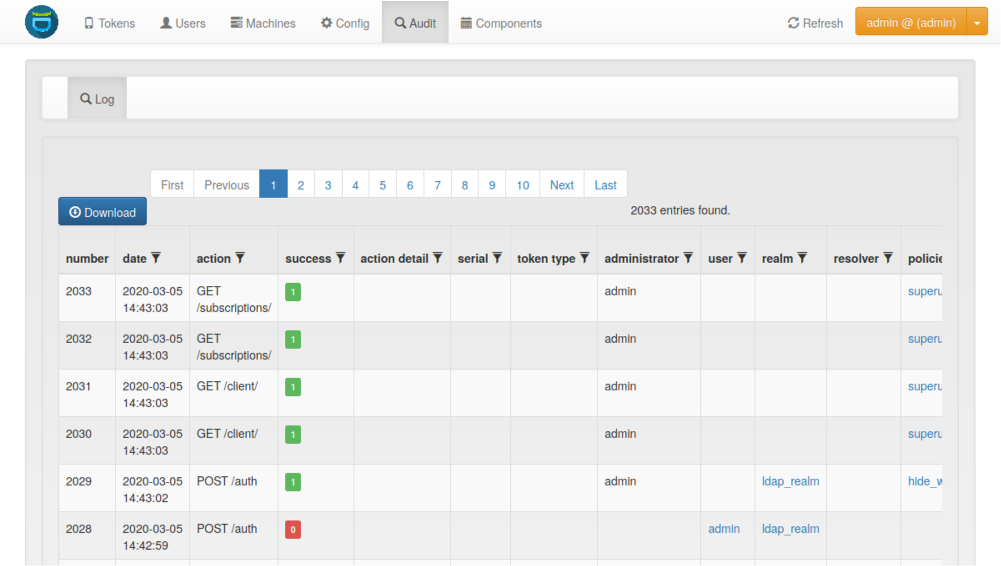
Max Failcount of newly enrolled tokens: 10

Sync Window of newly enrolled tokens: 1000

The challenge validity time: 120

1.4.5 Audit

In this tab, the *Audit* log is displayed which lists all events the server registers.

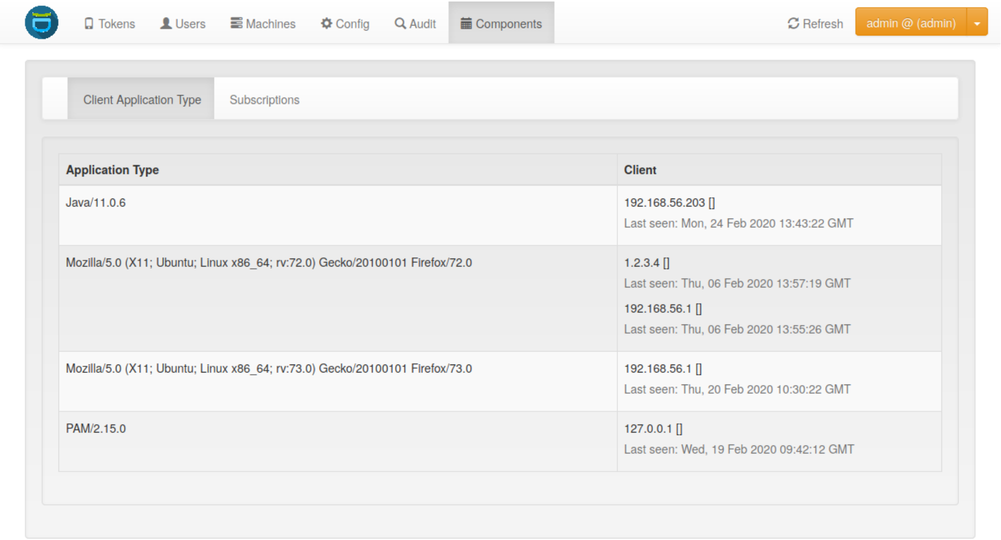


2033 entries found.

number	date	action	success	action detail	serial	token type	administrator	user	realm	resolver	policie
2033	2020-03-05 14:43:03	GET /subscriptions/	1				admin				superu
2032	2020-03-05 14:43:03	GET /subscriptions/	1				admin				superu
2031	2020-03-05 14:43:03	GET /client/	1				admin				superu
2030	2020-03-05 14:43:03	GET /client/	1				admin				superu
2029	2020-03-05 14:43:02	POST /auth	1				admin		ldap_realm		hide_w
2028	2020-03-05 14:42:59	POST /auth	0					admin	ldap_realm		

1.4.6 Components

Starting with privacyIDEA 2.15 you can see privacyIDEA components in the Web UI. privacyIDEA collects authenticating clients with their User Agent. Usually this is a type like *PAM*, *FreeRADIUS*, *Wordpress*, *OwnCloud*, … For more information, you may read on *Application Plugins*. This overview helps you to understand your network and keep track which clients are connected to your network.



Application Type	Client
Java/11.0.6	192.168.56.203 [] Last seen: Mon, 24 Feb 2020 13:43:22 GMT
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:72.0) Gecko/20100101 Firefox/72.0	1.2.3.4 [] Last seen: Thu, 06 Feb 2020 13:57:19 GMT 192.168.56.1 [] Last seen: Thu, 06 Feb 2020 13:55:26 GMT
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:73.0) Gecko/20100101 Firefox/73.0	192.168.56.1 [] Last seen: Thu, 20 Feb 2020 10:30:22 GMT
PAM/2.15.0	127.0.0.1 [] Last seen: Wed, 19 Feb 2020 09:42:12 GMT

Fig. 14: *The Components display client applications and subscriptions*

Subscriptions, e.g. with NetKnights, the company behind privacyIDEA, can also be viewed and managed in this tab.

1.5 Configuration

The configuration menu can be used to define useridresolvers and realms, set the system config and the token config.

It also contains a shortcut to the policy tab (see *Policies*).

1.5.1 UserIdResolvers

Each organisation or company usually has its users managed at a central location. This is why privacyIDEA does not provide its own user management but rather connects to existing user stores.

UserIdResolvers are connectors to those user stores, the locations, where the users are managed. Nowadays this can be LDAP directories or especially Active Directory, some times FreeIPA or the Redhat 389 service. But classically users are also located in files like /etc/passwd on standalone unix systems. Web services often use SQL databases as user store.

Today with many more online cloud services SCIM is also an uprising protocol to access userstores.

privacyIDEA already comes with UserIdResolvers to talk to all these user stores:

- Flatfile resolver,
- LDAP resolver,
- SQL resolver,
- SCIM resolver.

Note: New resolver types (python modules) can be added easily. See the module section for this (*UserIdResolvers*).

You can create as many UserIdResolvers as you wish and edit existing resolvers. When you have added all configuration data, most UIs of the UserIdResolvers have a button "Test resolver", so that you can test your configuration before saving it.

Starting with privacyIDEA 2.4 resolvers can be editable, i.e. you can edit the users in the user store. Read more about this at *Manage Users*.

Note: Using the policy `authentication:otppin=userstore` users can authenticate with the password from their user store, being the LDAP password, SQL password or password from flat file.

Flatfile resolver

Flatfile resolvers read files like `/etc/passwd`.

Note: The file `/etc/passwd` does not contain the unix password. Thus, if you create a flatfile resolver from this file the functionality with `otppin=userstore` is not available. You can create a flatfile with passwords using the tool `privacyidea-create-pwidresolver-user` which is usually found in `/opt/privacyidea/bin/`.

Create a flat file like this:

```
privacyidea-create-pwidresolver-user -u user2 -i 1002 >> /your/flat/file
```

LDAP resolver

The LDAP resolver can be used to access any kind of LDAP service like OpenLDAP, Active Directory, FreeIPA, Penrose, Novell eDirectory.

In case of Active Directory connections you might need to check the box `No anonymous referral chasing`. The underlying LDAP library is only able to do anonymous referral chasing. Active Directory will produce an error in this case[1].

The `Server URI` can contain a comma separated list of servers. The servers are used to create a server pool and are used with a round robin strategy[3].

Example:

```
ldap://server1, ldaps://server2:1636, server3, ldaps://server4
```

This will create LDAP requests to

- server1 on port 389
- server2 on port 1636 using SSL
- server3 on port 389
- server4 on port 636 using SSL.

The `Bind Type` with Active Directory can either be chosen as “Simple” or as “NTLM”.

Note: When using bind type “Simple” you need to specify the Bind DN like *cn=administrator,cn=users,dc=domain,dc=name*. When using bind type “NTLM” you need to specify Bind DN like *DOMAINNAME\username*.

The `LoginName attribute` is the attribute that holds the loginname. It can be changed to your needs.

Starting with version 2.20 you can provide a list of attributes in `LoginName Attribute` like:

```
sAMAccountName, userPrincipalName
```

This way a user can login with either his sAMAccountName or his principalName.

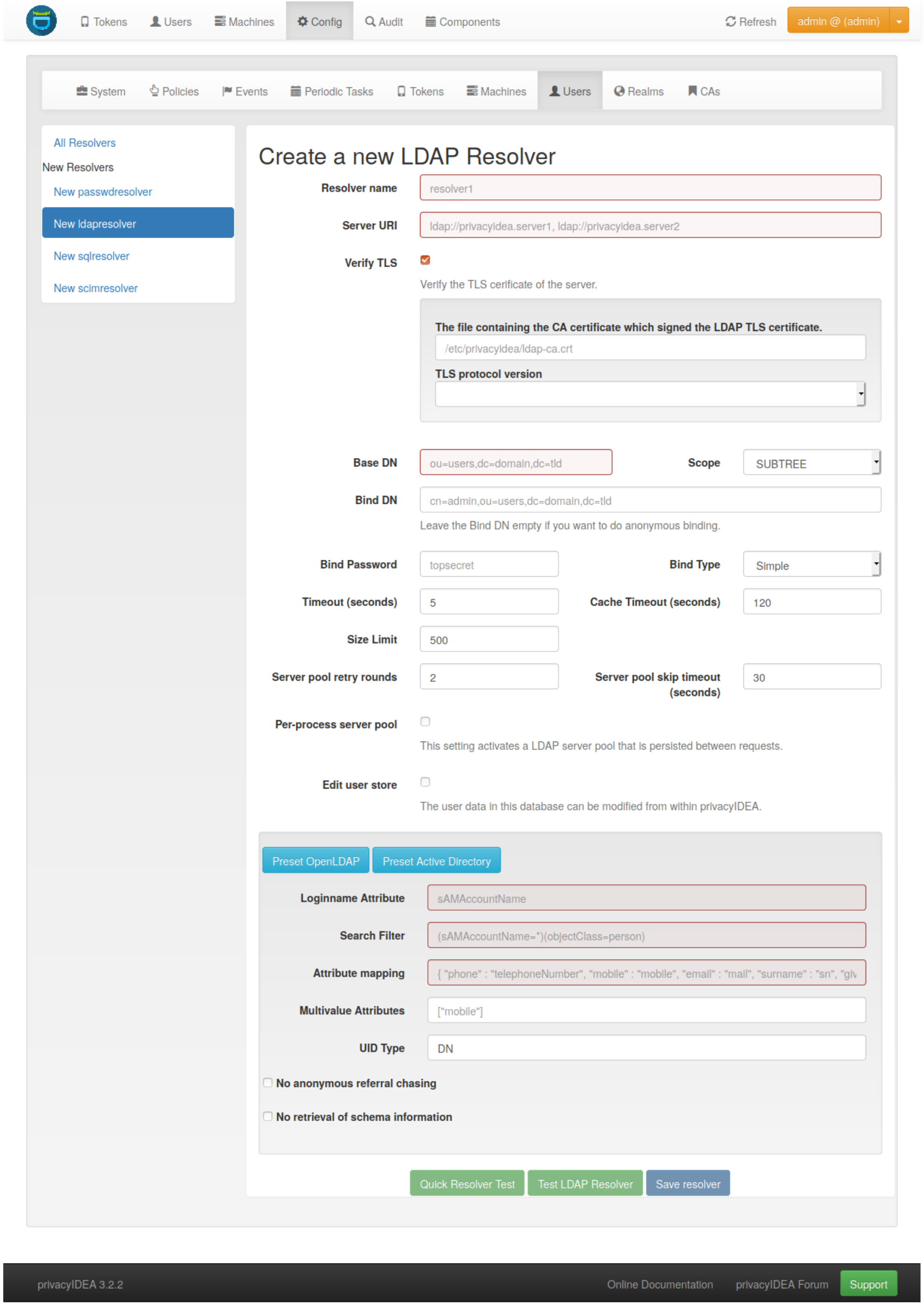
The `searchfilter` is used to list all possible users, that can be used in this resolver. The searchfilter is used for forward and backward search the object in LDAP.

The `attribute mapping` maps LDAP object attributes to user attributes in privacyIDEA. privacyIDEA knows the following attributes:

- phone,
- mobile,
- email,
- surname,
- givenname,
- password
- accountExpires.

[1] http://blogs.technet.com/b/ad/archive/2009/07/06/referral-chasing.aspx

[3] https://github.com/cannatag/ldap3/blob/master/docs/manual/source/servers.rst#server-pool



Create a new LDAP Resolver

Resolver name: resolver1

Server URI: ldap://privacyidea.server1, ldap://privacyidea.server2

Verify TLS: ☑

Verify the TLS cerificate of the server.

The file containing the CA certificate which signed the LDAP TLS certificate.: /etc/privacyidea/ldap-ca.crt

TLS protocol version:

Base DN: ou=users,dc=domain,dc=tld

Scope: SUBTREE

Bind DN: cn=admin,ou=users,dc=domain,dc=tld

Leave the Bind DN empty if you want to do anonymous binding.

Bind Password: topsecret

Bind Type: Simple

Timeout (seconds): 5

Cache Timeout (seconds): 120

Size Limit: 500

Server pool retry rounds: 2

Server pool skip timeout (seconds): 30

Per-process server pool: □

This setting activates a LDAP server pool that is persisted between requests.

Edit user store: □

The user data in this database can be modified from within privacyIDEA.

Loginname Attribute: sAMAccountName

Search Filter: (sAMAccountName=*)(objectClass=person)

Attribute mapping: { "phone" : "telephoneNumber", "mobile" : "mobile", "email" : "mail", "surname" : "sn", "giv

Multivalue Attributes: ["mobile"]

UID Type: DN

□ No anonymous referral chasing

□ No retrieval of schema information

Fig. 15: *LDAP resolver configuration*

The above attributes are used for privacyIDEA's normal functionality and are listed in the userview. However, with a SAML authentication request user attributes can be returned. (see *SAML Attributes*). To return arbitrary attributes from the LDAP you can add additional keys to the attribute mapping with a key, you make up and the LDAP attribute like:

```
"homedir": "homeDirectory",
"studentID": "objectGUID"
```

"homeDirectory" and "objectGUID" being the attributes in the LDAP directory and "homedir" and "studentID" the keys returned in a SAML authentication request.

The `MULTIVALUEATTRIBUTES` config value can be used to specify a list of user attributes, that should return a list of values. Imagine you have a user mapping like `{ "phone" : "telephoneNumber", "email" : "mail", "surname" : "sn", "group": "memberOf"}`. Then you could specify `["email", "group"]` as the multi value attribute and the user object would return the emails and the group memberships of the user from the LDAP server as a list.

Note: If the `MULTIVALUEATTRIBUTES` is left blank the default setting is "mobile". I.e. the mobile number will be returned as a list.

The `MULTIVALUEATTRIBUTES` can be well used with the `samlcheck` endpoint (see *Validate endpoints*) or with the policy *add_user_in_response*.

The `UID Type` is the unique identifier for the LDAP object. If it is left blank, the distinguished name will be used. In case of OpenLDAP this can be *entryUUID* and in case of Active Directory *objectGUID*. For FreeIPA you can use *ipaUniqueID*.

Note: The attributes *entryUUID*, *objectGUID*, and *ipaUniqueID* are case sensitive!

The option `No retrieval of schema information` can be used to disable the retrieval of schema information[4] in order to improve performance. This checkbox is deactivated by default and should only be activated after having ensured that schema information are unnecessary.

The `CACHE_TIMEOUT` configures a short living per process cache for LDAP users. The cache is not shared between different Python processes, if you are running more processes in Apache or Nginx. You can set this to `0` to deactivate this cache.

The *Server pool retry rounds* and *Server pool skip timeout* settings configure the behavior of the LDAP server pool. When establishing a LDAP connection, the resolver uses a round-robin strategy to select a LDAP server from the pool. If the current server is not reachable, it is removed from the pool and will be re-inserted after the number of seconds specified in the *skip timeout*. If no server from the pool is reachable, the servers are queried again from the beginning. If a reachable server has not been found after the number of rounds specified in the *retry rounds*, the request fails.

By default, knowledge about unavailable pool servers is not persisted between requests. Consequently, a new request may retry to reach unavailable servers, even though the *skip timeout* has not passed yet. If the *Per-process server pool* is enabled, knowledge about unavailable servers is persisted within each process. This setting may improve performance in situations in which a LDAP server from the pool is down for extended periods of time.

[4] http://ldap3.readthedocs.io/schema.html

TLS certificates

Starting with privacyIDEA 2.18, in case of encrypted LDAPS connections privacyIDEA can verify the TLS certificate. (Python >= 2.7.9 required) To have privacyIDEA verify the TLS certificate, you need to check the according checkbox.

You can specify a file with the trusted CA certificate, that signed the TLS certificate. The default CA filename is */etc/privacyidea/ldap-ca.crt* and can contain a list of base64 encoded CA certificates. PrivacyIDEA will use the CA file if specifed. If you leave the field empty it will also try the system certificate store (*/etc/ssl/certs/ca-certificates.crt* or */etc/ssl/certs/ca-bundle.crt*).

Modifying users

Starting with privacyIDEA 2.12, you can define the LDAP resolver as editable. I.e. you can create and modify users from within privacyIDEA.

There are two additional configuration parameters for this case.

`DN Template` defines how the DN of the new LDAP object should be created. You can use *username*, *surname*, *givenname* and *basedn* to create the distiguished name.

Examples:

> CN=<givenname> <surname>,<basedn>
>
> CN=<username>,OU=external users,<basedn>
>
> uid=<username>,ou=users,o=example,c=com

`Object Classes` defines which object classes the user should be assigned to. This is a comma separated list. The usual object classes for Active Directory are

> top, person, organizationalPerson, user, inetOrgPerson

Expired Users

You may set

> "accountExpires": "accountExpires"

in the attribute mapping for Microsoft Active Directories. You can then call the user listing API with the parameter *accountExpires=1* and you will only see expired accounts.

This functionality is used with the script *privacyidea-expired-users*.

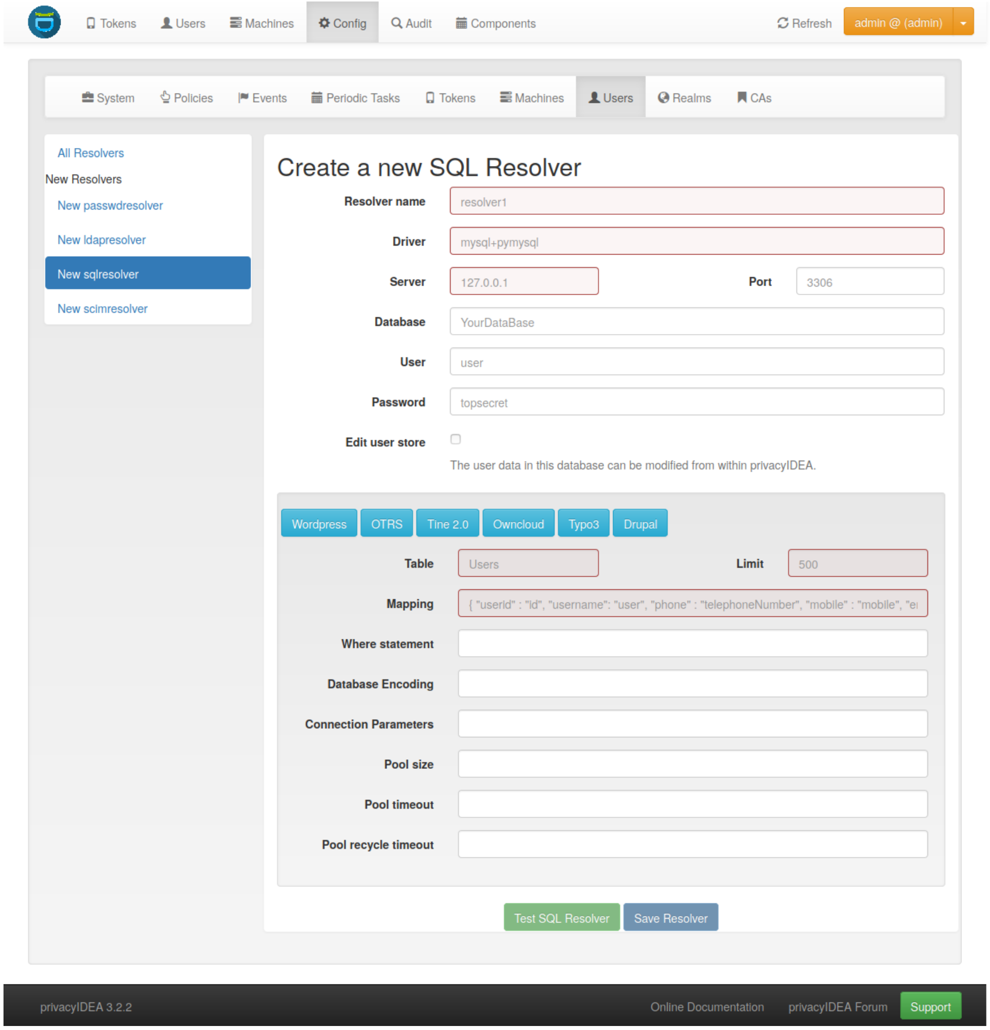
SQL resolver

The SQL resolver can be used to retrieve users from any kind of SQL database like MySQL, PostgreSQL, Oracle, DB2 or sqlite.

In the upper frame you need to configure the SQL connection. The SQL resolver uses SQLAlchemy internally. In the field `Driver` you need to set a driver name as defined by the SQLAlchemy dialects like "mysql" or "postgres".

In the `SQL attributes` frame you can specify how the users are identified.

The `Database table` contains the users.



Create a new SQL Resolver

Resolver name	resolver1		
Driver	mysql+pymysql		
Server	127.0.0.1	Port	3306
Database	YourDataBase		
User	user		
Password	topsecret		
Edit user store	□ The user data in this database can be modified from within privacyIDEA.		
Table	Users	Limit	500
Mapping	{ "userid" : "id", "username": "user", "phone" : "telephoneNumber", "mobile" : "mobile", "e		
Where statement			
Database Encoding			
Connection Parameters			
Pool size			
Pool timeout			
Pool recycle timeout			

Fig. 16: *SQL resolver configuration*

Note: At the moment, only one table is supported, i.e. if some of the user data like email address or telephone number is located in a second table, those data can not be retrieved.

The `Limit` is the SQL limit for a userlist request. This can be important if you have several thousand user entries in the table.

The `Attribute mapping` defines which table column should be mapped to which privayIDEA attribute. The known attributes are:

- userid *(mandatory)*,
- username *(mandatory)*,
- phone,
- mobile,
- email,
- givenname,
- surname,
- password.

The `password` attribute is the database column that contains the user password. This is used, if you are doing user authentication against the SQL database.

Note: There is no standard way to store passwords in an SQL database. privacyIDEA supports the most common ways like Wordpress hashes starting with *$P* or *$S*. Secure hashes starting with *{SHA}* or salted secure hashes starting with *{SSHA}*, *{SSHA256}* or *{SSHA512}*. Password hashes of length 64 are interpreted as OTRS sha256 hashes.

You can mark the users as `Editable`. The `Password_Hash_Type` can be used to determine which hash algorithm should be used, if a password of an editable user is written to the database.

You can add an additional `Where statement` if you do not want to use all users from the table.

The `poolSize` and `poolTimeout` determine the pooling behaviour. The `poolSize` (default 5) determine how many connections are kept open in the pool. The `poolTimeout` (default 10) specifies how long the application waits to get a connection from the pool.

Note: The pooling parameters only have an effect if the `PI_ENGINE_REGISTRY_CLASS` config option is set to `"shared"` (see *Engine Registry Class*). If you then have several SQL resolvers with the same connection and pooling settings, they will use the same shared connection pool. If you change the connection settings of an existing connection, the connection pool for the old connection settings will persist until the respective connections are closed by the SQL server or the web server is restarted.

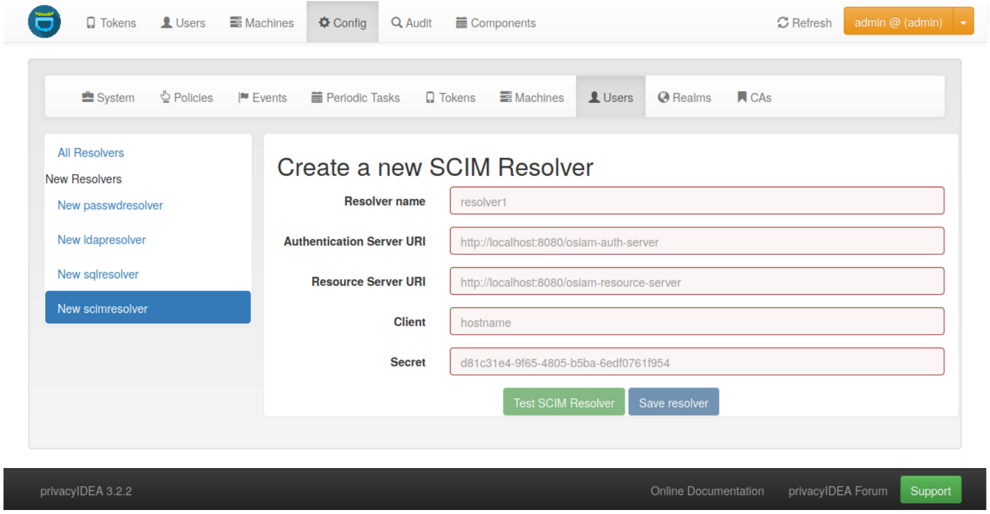
Note: The `Additional connection parameters` refer to the SQLAlchemy connection but are not used at the moment.

SCIM resolver

SCIM is a "System for Cross-domain Identity Management". SCIM is a REST-based protocol that can be used to ease identity management in the cloud.

The SCIM resolver is tested in basic functions with OSIAM[2], the "Open Source Idenity & Access Management".

To connect to a SCIM service you need to provide a URL to an authentication server and a URL to the resource server. The authentication server is used to authenticate the privacyIDEA server. The authentication is based on a `Client` name and the `Secret` for this client.



Create a new SCIM Resolver

Resolver name: resolver1

Authentication Server URI: http://localhost:8080/osiam-auth-server

Resource Server URI: http://localhost:8080/osiam-resource-server

Client: hostname

Secret: d81c31e4-9f65-4805-b5ba-6edf0761f954

User information is then retrieved from the resource server.

The available attributes for the `Attribute mapping` are:

- username *(mandatory)*,
- givenname,
- surname,
- phone,
- mobile,
- email.

User Cache

privacyIDEA does not implement local user management by design and relies on UserIdResolvers to connect to external user stores instead. Consequently, privacyIDEA queries user stores quite frequently, e.g. to resolve a login name to a user ID while processing an authentication request, which may introduce a significant slowdown. In order to optimize the response time of authentication requests, privacyIDEA 2.19 introduces the *user cache* which is located in the local database. It can be enabled in the system configuration (see *User Cache*).

A user cache entry stores the association of a login name in a specific UserIdResolver with a specific user ID for a predefined time called the *expiration timeout*, e.g. for one week. The processing of further authentication requests by the same user during this timespan does not require any queries to the user store, but only to the user cache.

The user cache should only be enabled if the association of users and user ID is not expected to change often: In case a user is deleted from the user store, but can still be found in the user cache and still has assigned tokens, the user will

[2] http://www.osiam.org

still be able to authenticate during the expiration timeout! Likewise, any changes to the user ID will not be noticed by privacyIDEA until the corresponding cache entry expires.

Expired cache entries are *not* deleted from the user cache table automatically. Instead, the tool `privacyidea-usercache-cleanup` should be used to delete expired cache entries from the database, e.g. in a cronjob.

However, cache entries are removed at some defined events:

- If a UserIdResolver is modified or deleted, all cache entries belonging to this resolver are deleted.
- If a user is modified or deleted in an editable UserIdResolver, all cache entries belonging to this user are deleted.

Note: Realms with multiple UserIdResolvers are a special case: If a user `userX` tries to authenticate in a realm with two UserIdResolvers `resolverA` (with highest priority) and `resolverB`, the user cache is queried to find the user ID of `userX` in the UserIdResolver `resolverA`. If the cache contains no matching entry, `resolverA` itself is queried for a matching user ID! Only if `resolverA` does not find a corresponding user, the user cache is queried to determine the user ID of `userX` in `resolverB`. If no matching entry can be found, `resolverB` is queried.

1.5.2 Realms

Users need to be in realms to have tokens assigned. A user, who is not member of a realm can not have a token assigned and can not authenticate.

You can combine several different UserIdResolvers (see *UserIdResolvers*) into a realm. The system knows one default realm. Users within this default realm can authenticate with their username.

Users in realms, that are not the default realm, need to be additionally identified. Therefor the users need to authenticate with their username and the realm like this:

```
user@realm
```

Relate User to a Realm

There are several options to relate a user to a specific realm during authentication. Usually, if only a login name is given, the user will be searched in the default realm, indicated with `defrealm` in the mapping table below.

If a *realm* parameter is given in a */auth* or */validate/check* request, it supersedes a possible *split* realm.

The following table shows different combinations of *user(name)*-parameter and *realm*-parameter. Depending on the *Split @ Sign*-setting, the following table shows in which realm the user will be searched.

Input parameter		*Split @ Sign*-setting	
user(name)	realm	true	false
user	–	user defrealm	user defrealm
user	realm1	user realm1	user realm1
user	unknown	–	–
user@realm1	–	user realm1	user@realm1 defrealm
user@realm1	realm1	user realm1	user@realm1 realm1
user@realm1	realm2	user realm2	user@realm1 realm2
user@realm2	realm1	user realm1	user@realm2 realm1
user@realm1	unknown	–	–
user@unknown	–	user@unknown defrealm	user@unknown defrealm
user@unknown	realm1	user@unknown realm1	user@unknown realm1
user@unknown	unknown	–	–

Note: Be aware that if the *Split @ Sign*-setting is *true*, a *realm* parameter is given **and** a user name with an @-sign is given where the part after the @ denotes a valid realm, the *realm* parameter will take precedence.

List of Realms

The realms dialog gives you a list of the already defined realms.

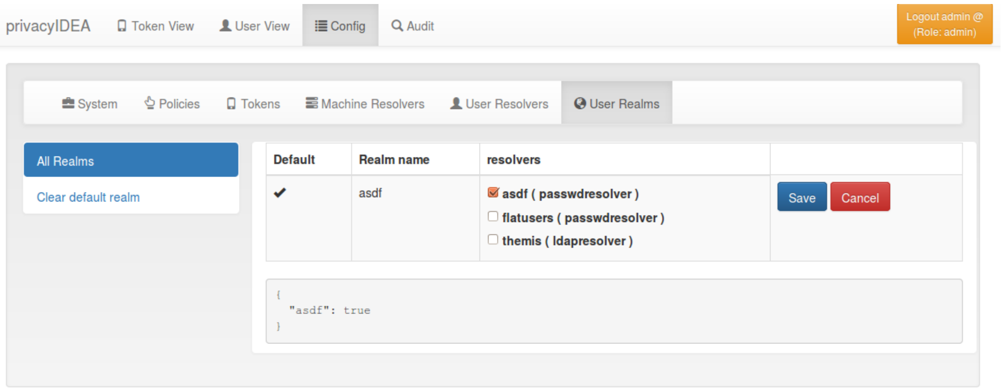
It shows the name of the realms, whether it is the default realm and the names of the resolvers, that are combined to this realm.

You can delete or edit an existing realm or create a new realm.

Edit Realm

Each realm has to have a unique name. The name of the realm is case insensitive. If you create a new realm with the same name like an existing realm, the existing realm gets overwritten.

If you click *Edit Realm* you can select which userresolver should be contained in this realm. A realm can contain several resolvers.



Default	Realm name	resolvers
✓	asdf	☑ asdf (passwdresolver) ☐ flatusers (passwdresolver) ☐ themis (ldapresolver)

```
{
  "asdf": true
}
```

Fig. 17: *Edit a realm*

Resolver Priority

Within a realm you can give each resolver a priority. The priority is used to find a user that is located in several resolvers. If a user is located in more than one resolver, the user will be taken from the resolver with the lowest number in the priority.

Priorities are numbers between 1 and 999. The lower the number the higher the priority.

Example:

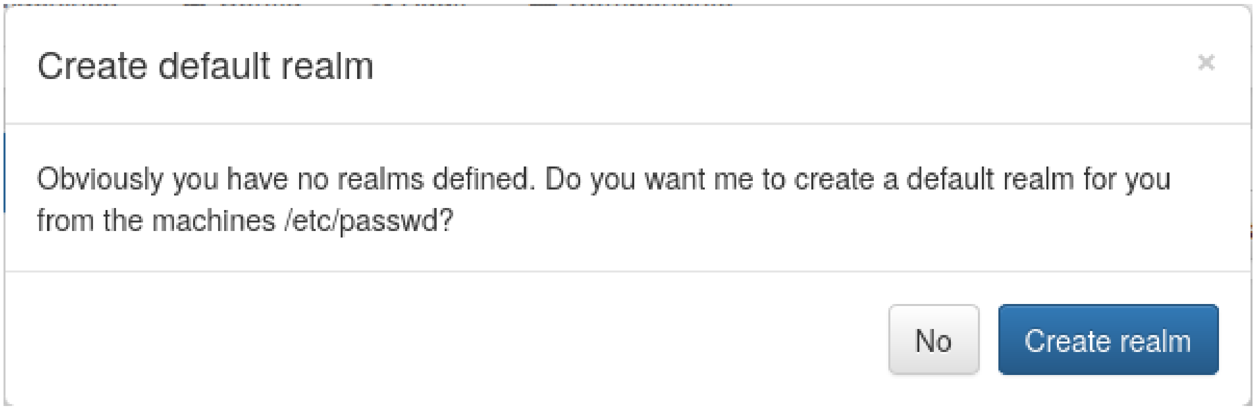
A user "administrator" is located in a resolver "users" which contains all Active Directory users. And the "administrator" is located in a resolver "admins", which contains all users in the Security Group "Domain Admins" from the very same domain. Both resolvers are in the realm "AD", "admins" with priority 1 and "users" with priority 2.

Thus the user "administrator@AD" will always resolve to the user located in resolver "admins".

This is useful to create policies for the security group "Domain Admins".

Note: A resolver has a priority per realm. I.e. a resolver can have a different priority in each realm.

Autocreate Realm



Create default realm

Obviously you have no realms defined. Do you want me to create a default realm for you from the machines /etc/passwd?

If you have a fresh installation, no resolver and no realm is defined. To get you up and running faster, the system will ask you, if it should create the first realm for you.

If you answer "yes", it will create a resolver named "deflocal" that contains all users from /etc/passwd and a realm named "defrealm" with this very resolver.

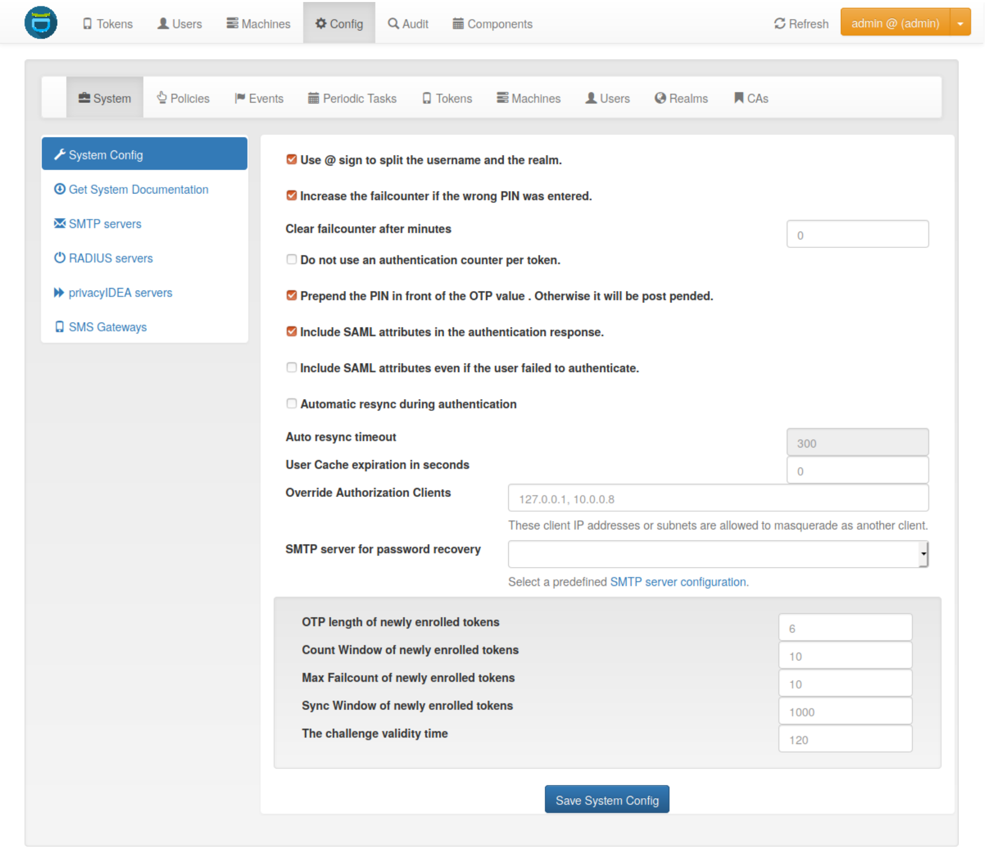
Thus you can immediately start assigning and enrolling tokens.

If you check "Do not ask again" this will be stored in a cookie in your browser.

Note: The realm "defrealm" will be the default realm. So if you create a new realm manually and want this new realm to be the default realm, you need to set this new realm to be default manually.

1.5.3 System Config

The system configuration has three logical topics: Settings, token default settings and GUI settings.



☑ Use @ sign to split the username and the realm.

☑ Increase the failcounter if the wrong PIN was entered.

Clear failcounter after minutes 0

□ Do not use an authentication counter per token.

☑ Prepend the PIN in front of the OTP value . Otherwise it will be post pended.

☑ Include SAML attributes in the authentication response.

□ Include SAML attributes even if the user failed to authenticate.

□ Automatic resync during authentication

Auto resync timeout 300

User Cache expiration in seconds 0

Override Authorization Clients 127.0.0.1, 10.0.0.8

These client IP addresses or subnets are allowed to masquerade as another client.

SMTP server for password recovery

Select a predefined SMTP server configuration.

OTP length of newly enrolled tokens	6
Count Window of newly enrolled tokens	10
Max Failcount of newly enrolled tokens	10
Sync Window of newly enrolled tokens	1000
The challenge validity time	120

Fig. 18: *The system config*

Settings

Split @ Sign

`splitAtSign` defines if the username like *user@company* given during authentication should be split into the loginname *user* and the realm name *company*. In most cases this is the wanted behaviour so this is enabled by default.

But given your users log in with email addresses like *user@gmail.com* and *otheruser@outlook.com* you probably do not want to split.

How a user is related to a realm is described here: *Relate User to a Realm*

This option also affects the login via the *Authentication endpoints*

SAML Attributes

`Return SAML attributes` defines if during an SAML authentication request additional SAML attributes should be returned. Usually an authentication response only returns *true* or *false*.

The SAML attributes are the known attributes that are defined in the attribute mapping e.g. of the LDAP resolver like *email*, *phone*, *givenname*, *surname* or any other attributes you fetch from the LDAP directory. For more information read *LDAP resolver*.

In addition you can set the parameter `ReturnSamlAttributesOnFail`. In this case the response contains the SAML attributes of the user, even if the user failed to authenticate.

FailCounterIncOnFalsePin

If during authentication the given PIN matches a token but the OTP value is wrong the failcounter of the tokens for which the PIN matches, is increased. If the given PIN does not match any token, by default no failcounter is increased. The later behaviour can be adapted by `FailCounterIncOnFalsePin`. If `FailCounterIncOnFalsePin` is set and the given OTP PIN does not match any token, the failcounter of *all* tokens is increased.

Automatically clearing Failcounter

If the failcounter reaches the maximum the token gets a timestamp, when the max fail count was reached. *After* the specified amount of minutes in `failcounter_clear_timeout` the following will clear the failcounter again:

- A successful authentication with correct PIN and correct OTP value
- A successfully triggered challenge (Usually this means a correct PIN)
- An authentication with a correct PIN, but a wrong OTP value (Only if *Resetting Failcounter on correct PIN* is set).

A “0” means automatically clearing the fail counter is not used.

Note: After the maximum failcounter is reached, new requests will not update the mentioned timestamp.

Also see *How to mitigate brute force and lock tokens*.

Resetting Failcounter on correct PIN

After the above mentioned timeout the failcounter is reset by a successful authentication (correct PIN and OTP value) or by the correct PIN of a challenge response token.

It can be also reset by the correct PIN of any token, when setting `ResetFailcounterOnPIN` to True. The default behaviour is, that the correct PIN of a normal token will *not* reset the failcounter after the clearing timeout.

Prepend PIN

`PrependPin` defines if the OTP PIN should be given in front ("pin123456") or in the back ("12345pin") of the OTP value.

AutoResync

`Auto resync` defines if the system should try to resync a token if a user provides a wrong OTP value. AutoResync works like this:

- If the counter of a wrong OTP value is within the resync window, the system remembers the counter of the OTP value for this token in the token info field `otp1c`.
- Now the user needs to authenticate a second time within `auto resync timeout` with the next successive OTP value.
- The system checks if the counter of the second OTP value is the successive value to `otp1c`.
- If it is, the token counter is set and the user is successfully authenticated.

Note: AutoResync works for all HOTP and TOTP based tokens including SMS and Email tokens.

User Cache

The setting `User Cache expiration in seconds` is used to enable the user cache and configure its expiration timeout. If its value is set to `0` (which is the default value), the user cache is disabled. Otherwise, the value determines the time in seconds after which entries of the user cache expire. For more information read *User Cache*.

Note: If the user cache is already enabled and you increase the expiration timeout, expired entries that still exist in the user cache could be considered active again!

Override Authorization Client

`Override Authorization client` is important with client specific policies (see *Policies*) and RADIUS servers or other proxies. In case of RADIUS the authenticating client for the privacyIDEA system will always be the RADIUS server, which issues the authentication request. But you can allow the RADIUS server IP to send another client information (in this case the RADIUS client) so that the policy is evaluated for the RADIUS client. A RADIUS server may add the API parameter *client* with a new IP address. A HTTP reverse proxy may append the respective client IP to the `X-Forwarded-For` HTTP header.

This field takes a comma separated list of sequences of IP Networks mapping to other IP networks.

Examples

```
10.1.2.0/24 > 192.168.0.0/16
```

Proxies in the sub net 10.1.2.0/24 may mask as client IPs 192.168.0.0/16. In this case the policies for the corresponding client in 192.168.x.x apply.

```
172.16.0.1
```

The proxy 172.16.0.1 may mask as any arbitrary client IP.

```
10.0.0.18 > 10.0.0.0/8
```

The proxy 10.0.0.18 may mask as any client in the subnet 10.x.x.x.

Note that the proxy definitions may be nested in order to support multiple proxy hops. As an example:

```
10.0.0.18 > 10.1.2.0/24 > 192.168.0.0/16
```

means that the proxy 10.0.0.18 may map to another proxy into the subnet 10.1.2.x, and a proxy in this subnet may mask as any client in the subnet 192.168.x.x.

With the same configuration, a proxy 10.0.0.18 may map to an application plugin in the subnet 10.1.2.x, which may in turn use a `client` parameter to mask as any client in the subnet 192.168.x.x.

Token default settings

Reset Fail Counter

`DefaultResetFailCount` will reset the failcounter of a token if this token was used for a successful authentication. If not checked, the failcounter will not be resetted and must be resetted manually.

Note: The following settings are token specific value which are set during enrollment. If you want to change this value of a token later on, you need to change this at the tokeninfo dialog.

Maximum Fail Counter

`DefaultMaxFailCount` is the maximum failcounter a token way get. If the failcounter exceeds this number the token can not be used unless the failcounter is resetted.

Note: In fact the failcounter will only increase till this maxfailcount. Even if more failed authentication request occur, the failcounter will not increase anymore.

Sync Window

`DefaultSyncWindow` is the window how many OTP values will be calculated during resync of the token.

OTP Length

`DefaultOtpLen` is the length of the OTP value. If no OTP length is specified during enrollment, this value will be used.

Count Window

`DefaultCountWindow` defines how many OTP values will be calculated during an authentication request.

Challenge Validity Time

`DefaultChallengeValidityTime` is the timeout for a challenge response authentication. If the response is set after the ChallengeValidityTime, the response is not accepted anymore.

SerialLength

The default length of generated serial numbers is an 8 digit hex string. If you need another length, it can be configured in the database table `Config` with the key word `SerialLength`.

No Authenitcation Counter

Usually privacyIDEA keeps track of how often a token is used for authentication and how often this authentication was successful. This is a per token counter. This information is written to the token database as a parameter of each token.

The setting "Do not use an authentication counter per token" (`no_auth_counter`) means that privacyIDEA does not track this information at all.

1.5.4 CA Connectors

You can use privacyIDEA to enroll certificates and assign certificates to users.

You can define connections to Certifacte Authorities, that are used when enrolling certificates.

When you enroll a Token of type *certificate* the Certificate Signing Request gets signed by one of the CAs attached to privacyIDEA by the CA connectors.

The first CA connector that ships with privacyIDEA is a connector to a local openSSL based Certificate Authority as shown in figure *A local CA definition*.

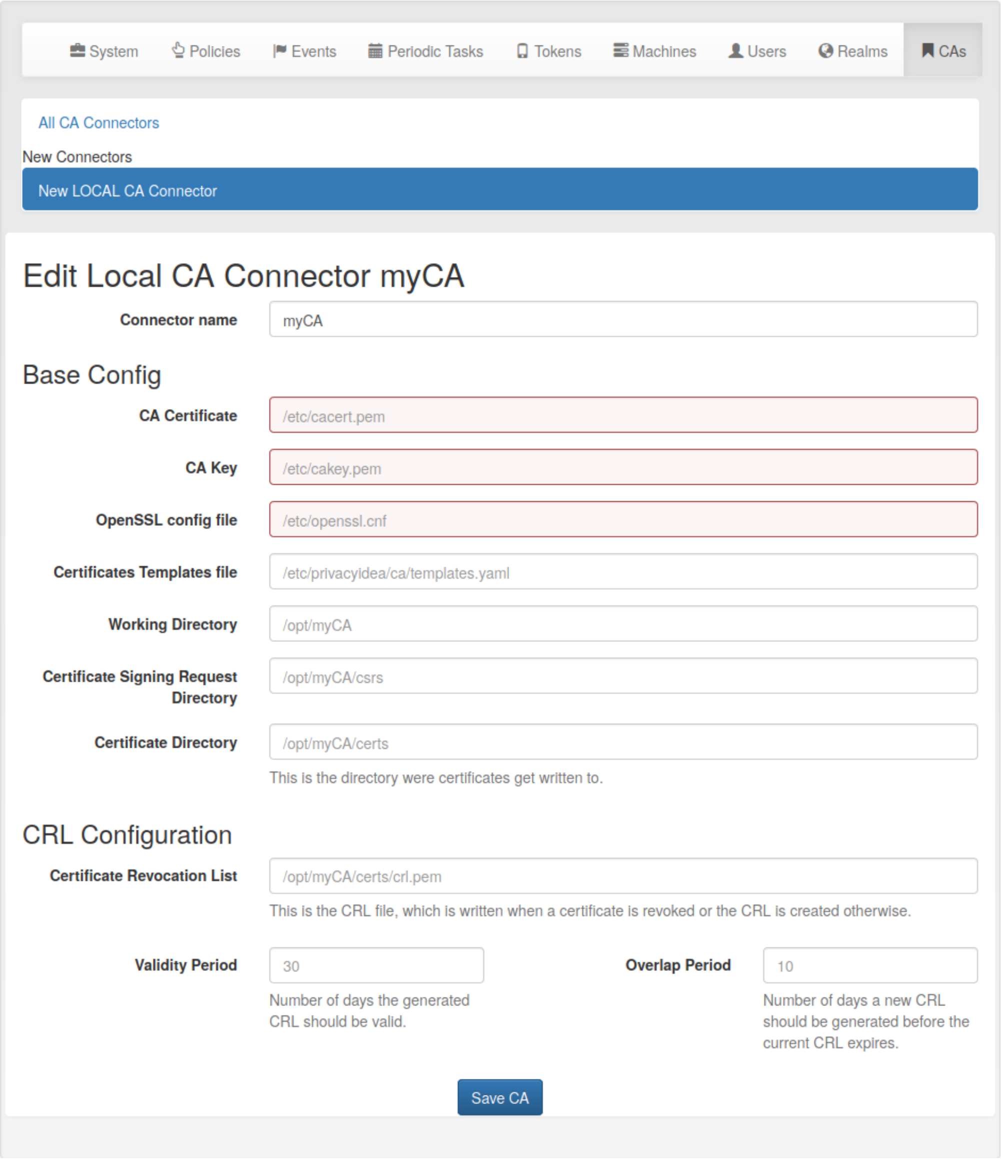
When enrolling a certificate token you can choose, which CA should sign the certificate request.

Local CA Connector

The local CA connector calls a local openssl configuration.

Starting with privacyIDEA version 2.12 an example *openssl.cnf* is provided in */etc/privacyidea/CA/openssl.cnf*.

Note: This configuration and also this description is ment to be as an example. When setting up a productive CA, you should ask a PKI consultant for assistance.



Edit Local CA Connector myCA

Connector name: myCA

Base Config

CA Certificate: /etc/cacert.pem

CA Key: /etc/cakey.pem

OpenSSL config file: /etc/openssl.cnf

Certificates Templates file: /etc/privacyidea/ca/templates.yaml

Working Directory: /opt/myCA

Certificate Signing Request Directory: /opt/myCA/csrs

Certificate Directory: /opt/myCA/certs

This is the directory were certificates get written to.

CRL Configuration

Certificate Revocation List: /opt/myCA/certs/crl.pem

This is the CRL file, which is written when a certificate is revoked or the CRL is created otherwise.

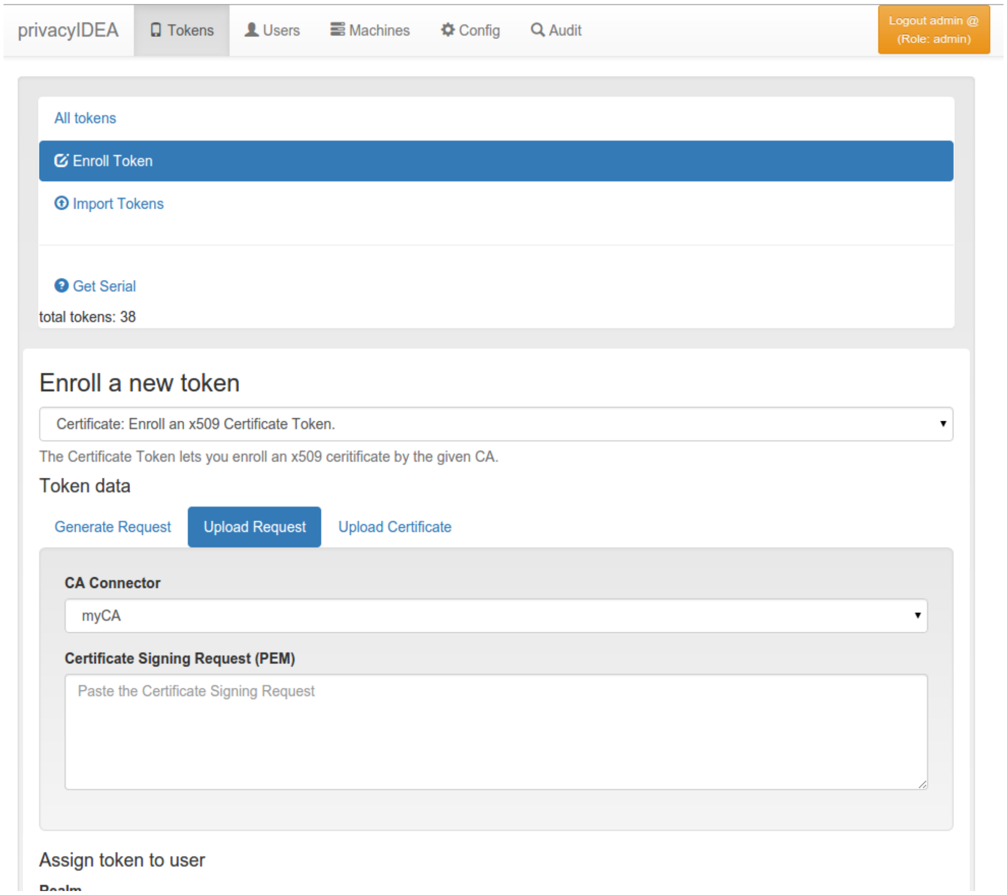
Validity Period: 30

Number of days the generated CRL should be valid.

Overlap Period: 10

Number of days a new CRL should be generated before the current CRL expires.

Fig. 19: *A local CA definition*



total tokens: 38

Enroll a new token

Certificate: Enroll an x509 Certificate Token.

The Certificate Token lets you enroll an x509 ceritificate by the given CA.

Token data

CA Connector

myCA

Certificate Signing Request (PEM)

Paste the Certificate Signing Request

Assign token to user

Realm

Fig. 20: *Enrolling a certificate token*

Manual Setup

1. Modify the parameters in the file */etc/privacyidea/CA/openssl.cnf* according to your needs.
2. Create your CA certificate:

```
openssl req -days 1500 -new -x509 -keyout /etc/privacyidea/CA/ca.key \
            -out /etc/privacyidea/CA/ca.crt \
            -config /etc/privacyidea/CA/openssl.cnf

chmod 0600 /etc/privacyidea/CA/ca.key
touch /etc/privacyidea/CA/index.txt
echo 01 > /etc/privacyidea/CA/serial
chown -R privacyidea /etc/privacyIDEA/CA
```

3. Now set up a local CA connector within privacyIDEA with the directory */etc/privacyidea/CA* and the files accordingly.

Easy Setup

Starting with privacyIDEA version 2.18 it gets easier to setup local CAs.

You can use the *The pi-manage Script* tool to setup a new CA like this:

> pi-manage ca create myCA

This will ask you for all necessary parameters for the CA and then automatically

1. Create the files for this new CA and
2. Create the CA connector in privacyIDEA.

Management

There are different ways to enroll a certificate token. See *Certificate Token*.

When an administrator *revokes* a certificate token, the certificate is revoked and a CRL is created.

Note: privacyIDEA does not create the CRL regularly. The CRL usually has a validity period of 30 days. I.e. you need to create the CRL on a regular basis. You can use openssl to do so or the pi-manage command.

Starting with version 2.18 the pi-manage command has an additional sub-command `ca`:

> pi-manage ca list

lists all configured *CA connectors*. You can use the *-v* switch to get more information.

You can create a new CRL with the command:

> pi-manage ca create_crl <CA name>

This command will check the *overlap period* and only create a new CRL if it is necessary. If you want to force the creation of the CRL, you can use the switch *-f*.

For more information on pi-manage see *The pi-manage Script*.

Templates

The *local CA* supports a kind of certificate templates. These “templates” are predefined combinations of *extensions* and *validity days*, as they are passed to openssl via the parameters `-extensions` and `-days`.

This way the administrator can define certificate templates with certain X.509 extensions like keyUsage, extendedKeyUsage, CDPs or AIAs and certificate validity periods.

The extensions are defined in YAML file and the location of this file is added to the CA connector definition.

The file can look like this, defining three templates “user”, “webserver” and “template3”:

> **user:** days: 365 extensions: “user”
>
> **webserver:** days: 750 extensions: “server”
>
> **template3:** days: 10 extensions: “user”

1.5.5 SMTP server configuration

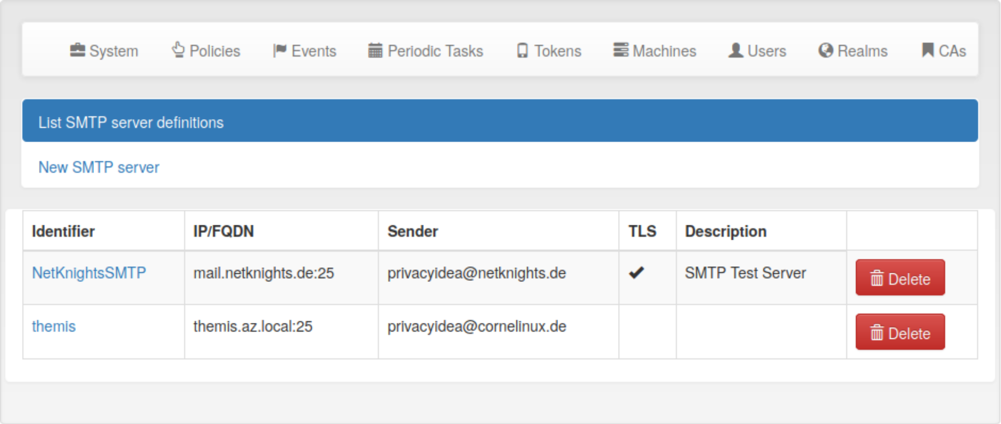
Starting with privacyIDEA 2.10 you can define SMTP server configurations. *SMTP server endpoints*.

An SMTP server configuration contains the

- server as FQDN or IP address,
- the port (defaults to 25),
- the sender email address,
- a username and password in case of authentication
- an optional description
- a TLS flag.

Each SMTP server configuration is addressed via a *unique identifier*. You can then use such a configuration for Email or SMS token, for PIN handling or in policies for *User registration*.

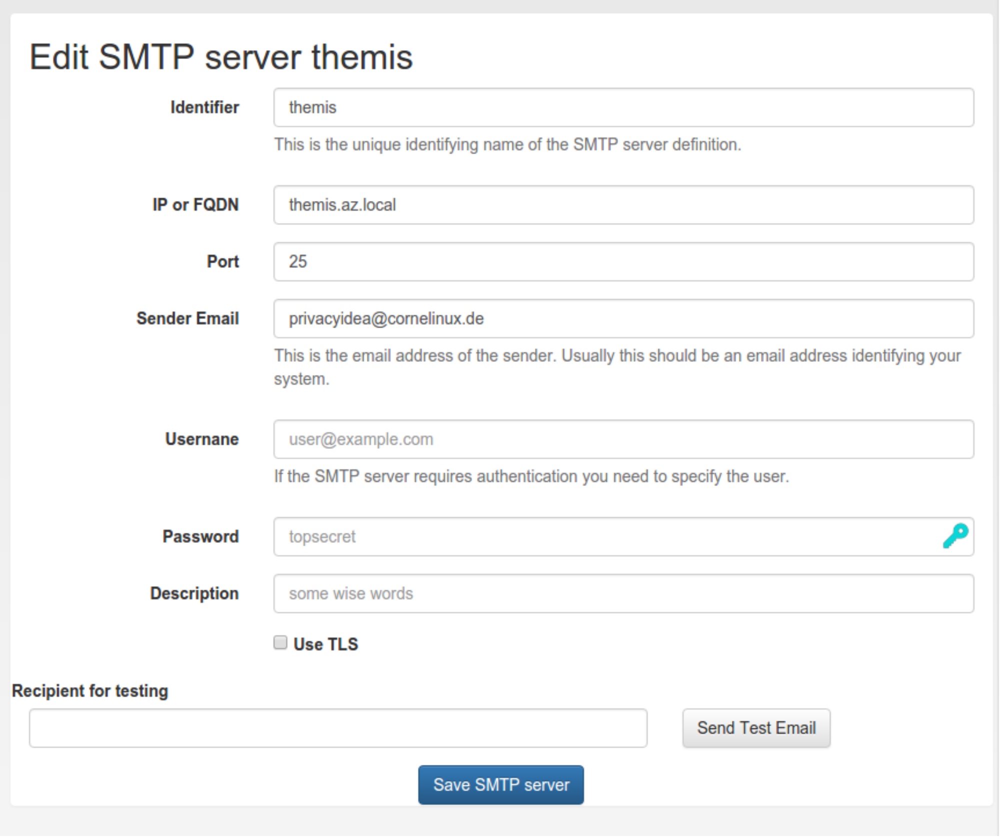
Under *Config->Sytem->SMTP servers* you can get a list of all configured SMTP servers, create new server definitions and delete them.



Identifier	IP/FQDN	Sender	TLS	Description	
NetKnightsSMTP	mail.netknights.de:25	privacyidea@netknights.de	✓	SMTP Test Server	Delete
themis	themis.az.local:25	privacyidea@cornelinux.de			Delete

Fig. 21: *The list of SMTP servers.*

In the edit dialog you can enter all necessary attributes to talk to the SMTP server. You can also send a test email, to verify if your settings are correct.



Edit SMTP server themis

Identifier: themis

This is the unique identifying name of the SMTP server definition.

IP or FQDN: themis.az.local

Port: 25

Sender Email: privacyidea@cornelinux.de

This is the email address of the sender. Usually this should be an email address identifying your system.

Usernane: user@example.com

If the SMTP server requires authentication you need to specify the user.

Password: topsecret

Description: some wise words

□ Use TLS

Recipient for testing

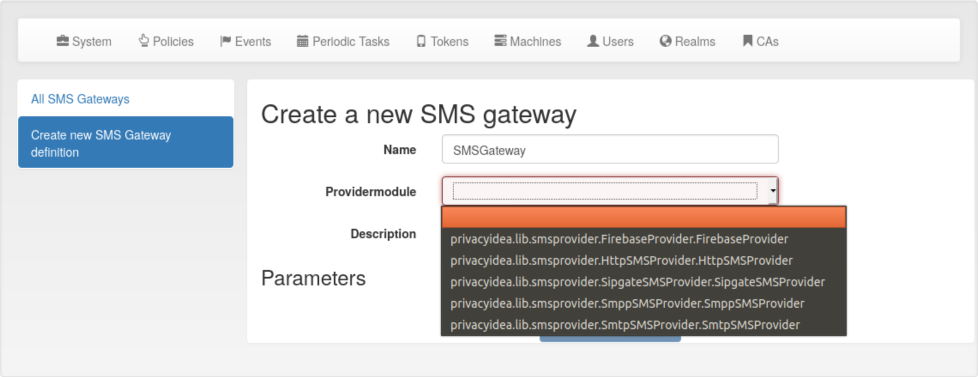
Fig. 22: *Edit an existing SMTP server definition.*

In case a *Job Queue* is configured, the SMTP server dialog shows a checkbox that enables sending all emails for the given SMTP server configuration via the job queue. Note that if the checkbox is checked, any test email will also be sent via the queue. This also means that privacyIDEA will display a success notice when the job has been sent to the queue successfully, which does not necessarily mean that the mail was actually sent. Thus, it is important to check that the test email is actually received.

1.5.6 SMS Gateway configuration

You can centrally define SMS gateways that can be used to send SMS with *SMS Token* or to use the SMS gateway for sending notifications.

There are different providers (gateways) to deliver SMS.



Create a new SMS gateway

Name: SMSGateway

Providermodule

privacyidea.lib.smsprovider.FirebaseProvider.FirebaseProvider
privacyidea.lib.smsprovider.HttpSMSProvider.HttpSMSProvider
privacyidea.lib.smsprovider.SipgateSMSProvider.SipgateSMSProvider
privacyidea.lib.smsprovider.SmppSMSProvider.SmppSMSProvider
privacyidea.lib.smsprovider.SmtpSMSProvider.SmtpSMSProvider

Description

Parameters

Firebase Provider

The Firebase provider was added in privacyIDEA 3.0. It sends notifications via the Google Firebase service and this is used for the *Push Token*. For an exemplary configuration, you may have a look on the articles on the privacyIDEA community website tagged with push token.

JSON config file

This is the location of the configuration file of the Firebase service. It has to be located on the privacyIDEA server.

apikey

The API key your Android app should use to connect to the Firebase service.

apiios

The API key your iOS app should use to connect to the Firebase service.

appid

The app ID your Android app should use to connect to the Firebase service.

appidios

The app ID your iOS app should use to connect to the Firebase service.

projectid

The project ID of the Firebase project, that is used to connect the app to.

projectnumber*

The project number of the Firebase project, that is used to connect the app to.

registration URL

This is the URL of your privacyIDEA server, which the push App should connect to for the second registration step. This URL usually ends with `/ttype/push`. Note, that the FQDN of your privacyIDEA server could be different for the smartphone App to connect to, than you usually use for connecting via the Web interface.

time to live

This is the time (in seconds) how long the privacyIDEA server accepts the response of the second registration step. The smartphone could have connection issues, so the second step could take some time to happen.

You can get all the necessary values *JSON config file*, *project ID*, *project number*, *app ID* and *API key* from your Firebase console.

HTTP provider

The HTTP provider can be used for any SMS gateway that provides a simple HTTP POST or GET request. This is the most commonly used provider. Each provider type defines its own set of parameters.

The following parameters can be used. These are parameters, that define the behaviour of the SMS Gateway definition.

CHECK_SSL

If the URL is secured via TLS (HTTPS), you can select, if the certificate should be verified or not.

PROXY, **HTTP_PROXY** and **HTTP_PROXY**

You can specify a proxy to connect to the HTTP gateway. Use the specific values to separate HTTP and HTTPS.

REGEXP

Regular expression to modify the phone number to make it compatible with provider.

HTTP_METHOD

Can be GET or POST.

RETURN_FAIL

If the text of `RETURN_FAIL` is found in the HTTP response of the gateway privacyIDEA assumes that the SMS could not be sent and an error occurred.

RETURN_SUCCESS

You can either use `RETURN_SUCCESS` or `RETURN_FAIL`. If the text of `RETURN_SUCCESS` is found in the HTTP response of the gateway privacyIDEA assumes that the SMS was sent successfully.

TIMEOUT

The timeout for contacting the API and receiving a response.

URL

This is the URL for the gateway.

USERNAME and **PASSWORD**

These are the username and the password if the HTTP request requires **basic authentication**.

PARAMETER

This can contain a dictionary of arbitrary fixed additional parameters. Usually this would also contain an ID or a password to identify you as a sender.

Options

You can define additional options. These are sent as parameters in the GET or POST request.

Note: The fixed parameters and the options can not have the same name! If you need an options, that has the same name as a parameter, you must not fill in the corresponding parameter.

Note: You can use the tags `{phone}` and `{otp}` to specify the mobile number and the otp value.

Examples

Clickatell

In case of the **Clickatell** provider the configuration will look like this:

- **URL**: http://api.clickatell.com/http/sendmsg
- **HTTP_METHOD**: GET
- **RETURN_SUCCESS**: ID

Set the additional **options** to be passed as HTTP GET parameters:

- user: *YOU*
- password: *your password*
- api_id: *you API ID*
- text: "Your OTP value is {otp}"
- to: {phone}

This will consturct an HTTP GET request like this:

```
http://api.clickatell.com/http/sendmsg?user=YOU&password=YOU&\
     api_id=YOUR API ID&text=....&to=....
```

where `text` and `to` will contain the OTP value and the mobile phone number. privacyIDEA will assume a successful sent SMS if the response contains the text "ID".

GTX-Messaging

GTX-Messaging is an SMS Gateway located in Germany.

The configuration looks like this (see[2]):

- **URL**: https://http.gtx-messaging.net/smsc.php
- **HTTP_METHOD**: GET
- **CHECK_SSL**: yes
- **RETURN_SUCCESS**: 200 OK

You need to set the additional **options**:

- user: <your account>
- pass: <the account password>
- to: {phone}
- text: Your OTP value is {otp}.

Note: The *user* and *pass* are not the credentials you use to login. You can find the required credentials for sending SMS in your GTX messaging account when viewing the details of your *routing account*.

Twilio

You can also use the **Twilio** service for sending SMS.[1].

- **URL**: https://api.twilio.com/2010-04-01/Accounts/B…8/Messages
- **HTTP_METHOD**: POST

For basic authentication you need:

- **USERNAME**: *your accountSid*
- **PASSWORD**: *your password*

Set the additional **options** as POST parameters:

- From: *your Twilio phone number*
- Body: {otp}
- To: {phone}

[2] https://www.gtx-messaging.com/de/api-docs/http/

[1] https://www.twilio.com/docs/api/rest/sending-messages

Sipgate provider

The sipgate provider connects to https://samurai.sipgate.net/RPC2 and takes only two arguments *USERNAME* and *PASSWORD*.

Parameters:

USERNAME

The sipgate username.

PASSWORD

The sipgate password.

PROXY

You can specify a proxy to connect to the HTTP gateway.

It takes not options.

If you activate debug log level you will see the submitted SMS and the response content from the Sipgate gateway.

SMPP Provider

The SMPP provider was added in privacyIDEA 2.22. It uses an SMS Center via the SMPP protocol to deliver SMS to the users.

You need to specify the **SMSC_HOST** and **SMSC_PORT** to talk to the SMS center. privacyIDEA need to authenticate against the SMS center. For this you can add the parameters **SYSTEM_ID** and **PASSWORD**. The parameter **S_ADDR** is the sender's number, shown to the users receiving an SMS. For the other parameters contact your SMS center operator.

SMTP provider

The SMTP provider sends an email to an email gateway. This is a specified, fixed mail address.

The mail should contain the phone number and the OTP value. The email gateway will send the OTP via SMS to the given phone number.

BODY

This is the body of the email. You can use this to explain the user, what he should do with this email. You can use the tags `{phone}` and `{otp}` to replace the phone number or the one time password.

MAILTO

This is the address where the email with the OTP value will be sent. Usually this is a fixed email address provided by your SMTP Gateway provider. But you can also use the tags `{phone}` and `{otp}` to replace the phone number or the one time password.

SMTPIDENTIFIED

Here you can select on of your centrally defined SMTP servers.

SUBJECT

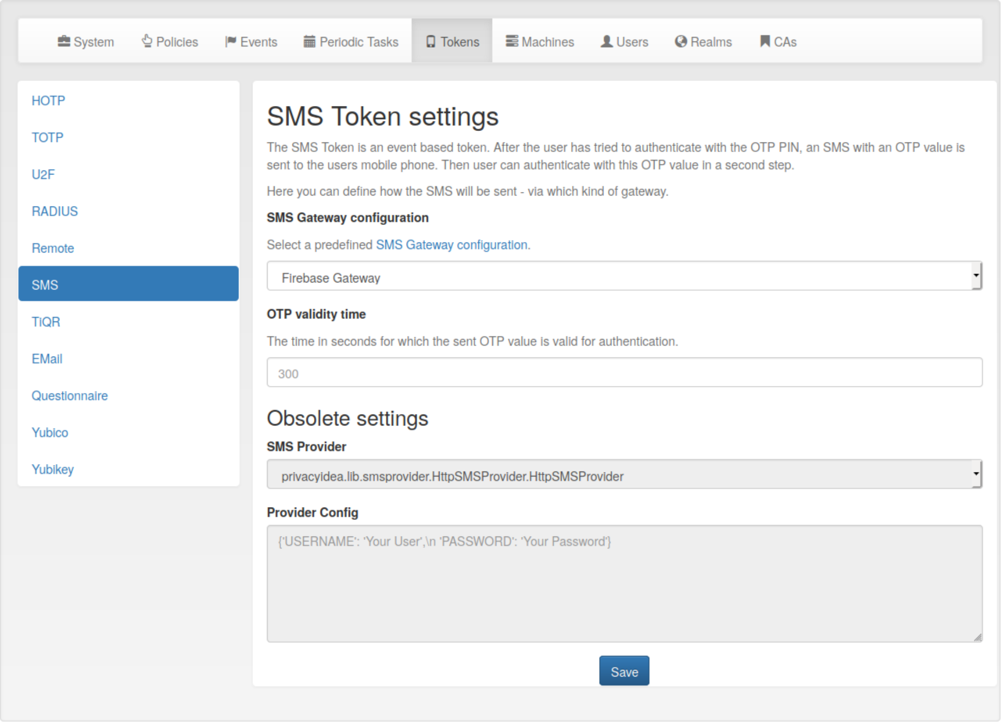
This is the subject of the email to be sent. You can use the tags `{phone}` and `{otp}` to replace the phone number or the one time password.

The default *SUBJECT* is set to *{phone}* and the default *BODY* to *{otp}*. You may change the *SUBJECT* and the *BODY* accordingly.

1.5.7 Token configuration

Each of the *Token types in privacyIDEA* can provide its own configuration dialog.

In this configuration dialog you can define default values for these token types. Some token additionally require *Configuration* such as the configuration of an SMTP server.



SMS Token settings

The SMS Token is an event based token. After the user has tried to authenticate with the OTP PIN, an SMS with an OTP value is sent to the users mobile phone. Then user can authenticate with this OTP value in a second step.

Here you can define how the SMS will be sent - via which kind of gateway.

SMS Gateway configuration

Select a predefined SMS Gateway configuration.

Firebase Gateway

OTP validity time

The time in seconds for which the sent OTP value is valid for authentication.

300

Obsolete settings

SMS Provider

privacyidea.lib.smsprovider.HttpSMSProvider.HttpSMSProvider

Provider Config

{'USERNAME': 'Your User',\n 'PASSWORD': 'Your Password'}

Fig. 23: *Examplary token configuration for an SMS Token*

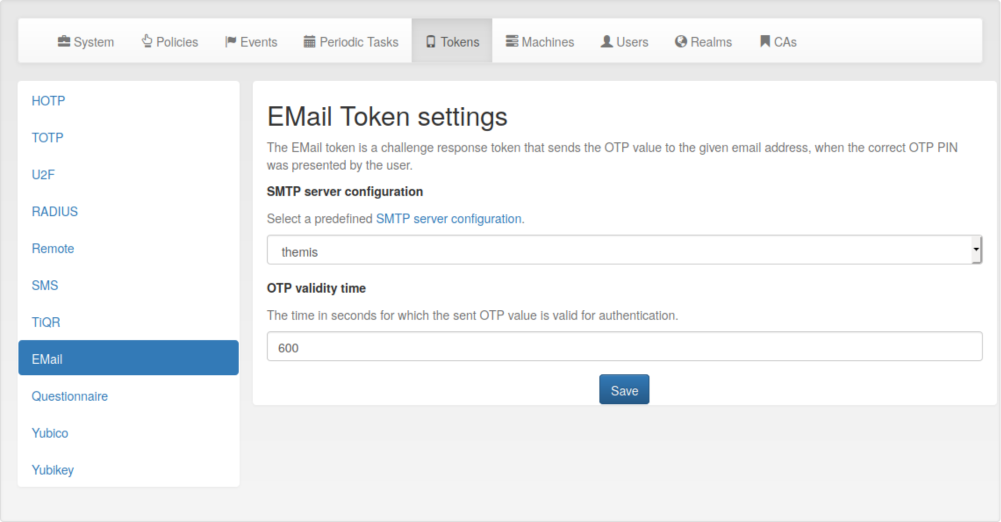
Email Token Configuration

For the email token to work, you have to first setup an *SMTP server configuration* and link it to the Email Token configuration at *Config -> Tokens -> Email*. The UI warns the user if one of these requirements is not fulfilled yet.

The Email OTP token creates a OTP value and sends this OTP value to the email address of the uses. The email can be triggered by authenticating with only the OTP PIN:

First step

In the first step the user will enter his OTP PIN and the sending of the email is triggered. The user is denied the access.



EMail Token settings

The EMail token is a challenge response token that sends the OTP value to the given email address, when the correct OTP PIN was presented by the user.

SMTP server configuration

Select a predefined SMTP server configuration.

themis

OTP validity time

The time in seconds for which the sent OTP value is valid for authentication.

600

Fig. 24: *Email Token configuration*

Seconds step

In the second step the user authenticates with the OTP PIN and the OTP value he received via email. The user is granted access.

Alternatively the user can authenticate with the *transaction_id* that was sent to him in the response during the first step and only the OTP value. The *transaction_id* assures that the user already presented the first factor (OTP PIN) successfully.

Configuration Parameters

Concurrent Challenges

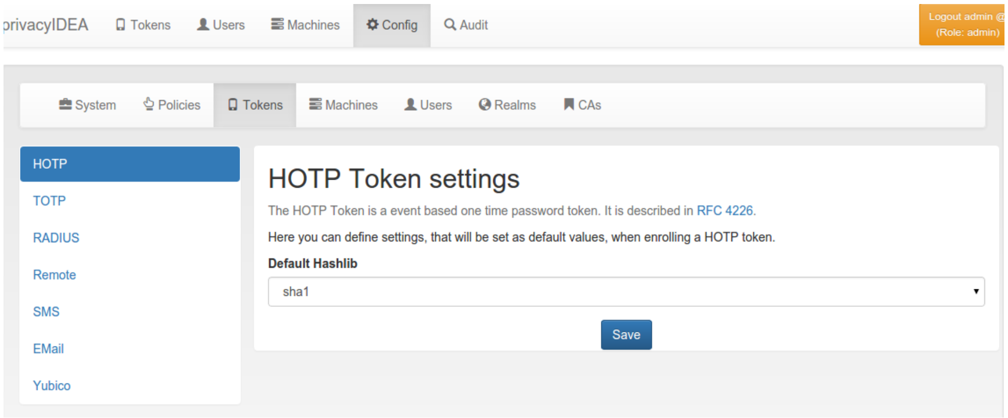
The config entry `email.concurrent_challenges` set in *The Config File* will save the sent OTP value in the challenge database. This way several challenges can be open at the same time. The user can answer the challenges in an arbitrary order. Set this to a true value. Defaults to off.

Deprecated Configuration Parameters

There are few more config entries handled, which are deprecated in recent versions of privacyIDEA.

- `email.mailserver` - The name or IP address of the mail server that is used to send emails.
- `email.port` - The port of the mail server.
- `email.username` - If the mail server requires authentication you need to enter a username. If no username is entered, no authentication is performed on the mail server.
- `email.password` - The password of the mail username to send emails.
- `email.mailfrom` - The mail address of the mail sender. This needs to correspond to the *Mail User*.
- `email.validtime` - This is the time in seconds, for how long the sent OTP value is valid. If a user tries to authenticate with the sent OTP value after this time, authentication will fail.
- `email.tls` - Whether the mail server should use TLS.

HOTP Token Config



HOTP Token settings

The HOTP Token is a event based one time password token. It is described in RFC 4226.

Here you can define settings, that will be set as default values, when enrolling a HOTP token.

Default Hashlib: sha1

Fig. 25: *HOTP Token configuration*

SMS Token Configuration

The SMS OTP token creates a OTP value and sends this OTP value to the mobile phone of the user. The SMS can be triggered by authenticating with only the OTP PIN:

First step

In the first step the user will enter his OTP PIN and the sending of the SMS is triggered. The user is denied the access.

Second step

In the second step the user authenticates with the OTP PIN and the OTP value he received via SMS. The user is granted access.

Alternatively the user can authenticate with the *transaction_id* that was sent to him in the response during the first step and only the OTP value. The *transaction_id* assures that the user already presented the first factor (OTP PIN) successfully.

A python SMS provider module defines how the SMS is sent. This can be done using an HTTP SMS Gateway. Most services like Clickatel or sendsms.de provide such a simple HTTP gateway. Another possibility is to send SMS via sipgate, which provides an XMLRPC API. The third possibility is to send the SMS via an SMTP gateway. The provider receives a specially designed email and sends the SMS accordingly. The last possibility to send SMS is to use an attached GSM modem.

Starting with version 2.13 the SMS configuration has been redesigned. You can now centrally define SMS gateways. These SMS gateways can be used for sending SMS OTP token but also for the event notifications. (See *User Notification Handler Module*)

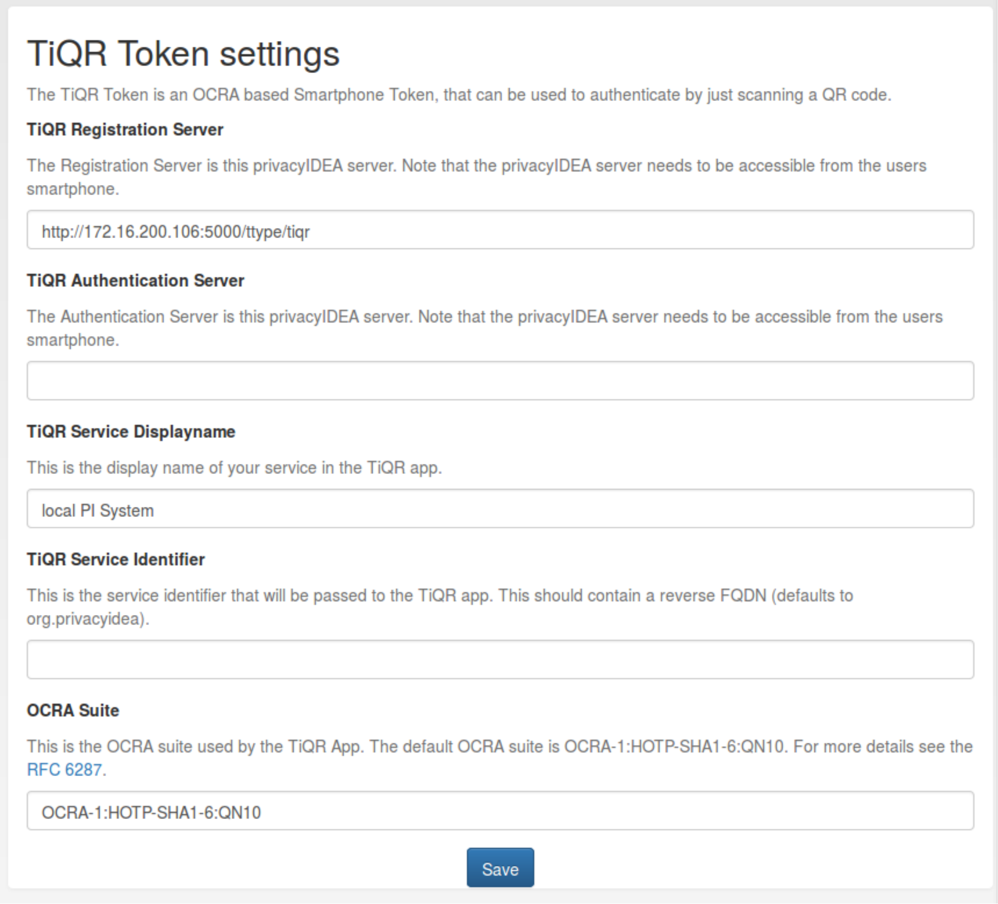
For configuring SMS Gateways read *SMS Gateway configuration*. I this token configuration you can select on defined gateway to send SMS for authentication.

Configuration Parameters

Concurrent Challenges

If set to True in *The Config File*, the config entry `sms.concurrent_challenges` will save the sent OTP value in the challenge database. This way several challenges can be open at the same time. The user can answer the challenges in an arbitrary order. Defaults to off.

TiQR Token Configuration



TiQR Token settings

The TiQR Token is an OCRA based Smartphone Token, that can be used to authenticate by just scanning a QR code.

TiQR Registration Server

The Registration Server is this privacyIDEA server. Note that the privacyIDEA server needs to be accessible from the users smartphone.

http://172.16.200.106:5000/ttype/tiqr

TiQR Authentication Server

The Authentication Server is this privacyIDEA server. Note that the privacyIDEA server needs to be accessible from the users smartphone.

TiQR Service Displayname

This is the display name of your service in the TiQR app.

local PI System

TiQR Service Identifier

This is the service identifier that will be passed to the TiQR app. This should contain a reverse FQDN (defaults to org.privacyidea).

OCRA Suite

This is the OCRA suite used by the TiQR App. The default OCRA suite is OCRA-1:HOTP-SHA1-6:QN10. For more details see the RFC 6287.

OCRA-1:HOTP-SHA1-6:QN10

Fig. 26: *TiQR Token configuration*

TiQR Registration Server

You need at least enter the *TiQR Registration Server*. This is the URL of your privacyIDEA installation, that can be reached from the smartphone during enrollment. So your smartphone needs to be on the same LAN (WLAN) like the privacyIDEA server or the enrollment URL needs to be accessible from the internet.

You also need to specify the path, which is usually */ttype/tiqr*.

During enrollment the parameter *action=metadata* and *action=enrollment* is added.

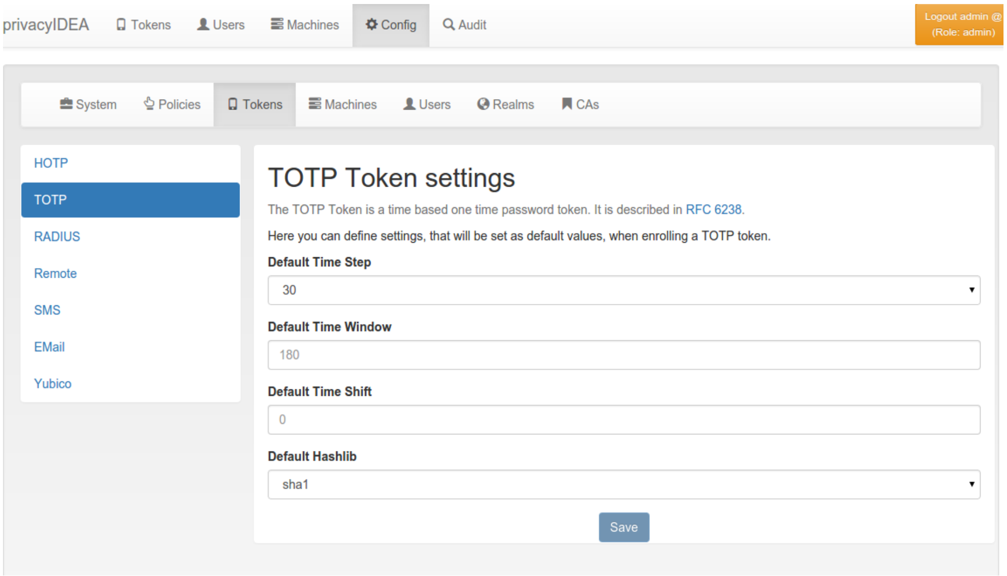
Note: We do not recommend putting the registration URL on the internet.

TiQR Authentication Server

This is the URL that is used during authentication. This can be another URL than the *Registration Server*. If it is left blank, the URL of the *Registration Server* is used.

During authentication the parameter *operation=login* is added.

TOTP Token Config



TOTP Token settings

The TOTP Token is a time based one time password token. It is described in RFC 6238.

Here you can define settings, that will be set as default values, when enrolling a TOTP token.

Default Time Step: 30

Default Time Window: 180

Default Time Shift: 0

Default Hashlib: sha1

Fig. 27: *TOTP Token configuration*

U2F Token Config

AppId

You need to configure the AppId of the privacyIDEA server. The AppId is define in the FIDO specification[1].

The AppId is the URL of your privacyIDEA and used to find or create the right key pair on the U2F device. The AppId must correspond the the URL that is used to call the privacyIDEA server.

Note: if you register a U2F device with an AppId https://privacyidea.example.com and try to authenticate at https://10.0.0.1, the U2F authentication will fail.

Note: The AppId must not contain any trailing slashes!

[1] https://fidoalliance.org/specs/fido-u2f-v1.0-nfc-bt-amendment-20150514/fido-appid-and-facets.html

Facets

If specifying the AppId as the FQDN you will only be able to authenticate at the privacyIDEA server itself or at any application in a sub directory on the privacyIDEA server. This is OK, if you are running a SAML IdP on the same server.

But if you also want to use the U2F token with other applications, you need to specify the AppId like this:

> https://privacyidea.example.com/pi-url/ttype/u2f

pi-url is the path, if you are running the privacyIDEA instance in a sub folder.

/ttype/u2f is the endpoint that returns a trusted facets list. Trusted facets are other hosts in the domain *example.com*. You need to define a policy that contains a list of the other hosts (*u2f_facets*).

For more information on AppId and trusted facets see[1].

For further details and for information how to add U2F to your application you can see the code documentation at *U2F Token*.

Workflow

You can use a U2F token on privacyIDEA and other hosts in the same Domain. To do so you need to do the following steps:

1. Configure the AppId to reflect your privacyIDEA server:

   > https://pi.your-network.com/ttype/u2f

   Add the path */ttype/u2f* is crucial. Otherwise privacyIDEA will not return the trusted facets.

2. Define a policy with the list of trusted facets. (see *u2f_facets*). Add the FQDNs of the hosts to the policy:

   > saml.your-network.com otherapp.your-network.com vpn.your-network.com

   Note: The privacyIDEA plugin for simpleSAMLphp supports U2F with privacyIDEA starting with version 2.8.

3. Now register a U2F token on https://pi.your-network.com. Due to the trusted facets you will also be able to use this U2F token on the other hosts.

4. Now got to https://saml.your-network.com and you will be able to authenticate with the very U2F token without any further registering.

WebAuthn Token Config

Trust Anchor Directory

You may define a directory containing trust roots for attestation certificates.

This should be a path to a local directory on the server which privacyIDEA has read access to. Any certificate in this directory will be trusted to correctly attest authenticators during enrollment.

This does not need to be set for WebAuthn to work, however without this, privacyIDEA can not check, whether an attestation certificate is actually trusted (it will still be checked for validity). Therefore it is mandatory to set this, if *webauthn_authenticator_attestation_level* is set to "trusted" through policy for any user.

WebAuthn Required Policies

For WebAuthn to work, a name and ID for the relying party need to be set. The relying party in WebAuthn represents the entity the user is registering with. In most cases this will be your company. In larger companies it is often helpful to segment according to department by setting up multiple ID and name policies for WebAuthn which apply to different users.

Relying Party ID

The ID of the relying party must be a fully-qualified domain name. Every web-service, where the WebAuthn token should be used needs to be reachable under a domain name which is a superset (i.e. a subdomain) of this ID. This means that a WebAuthn token enrolled with a relying party ID of *example.com* may be used to sign in to *privacyidea.example.com* and *owncloud.example.com*. However, this token will not be able to sign in to a service under *example.de*, or any other webservice that is not hosted on a subdomain of *example.com*.

See also: *webauthn_relying_party_id*.

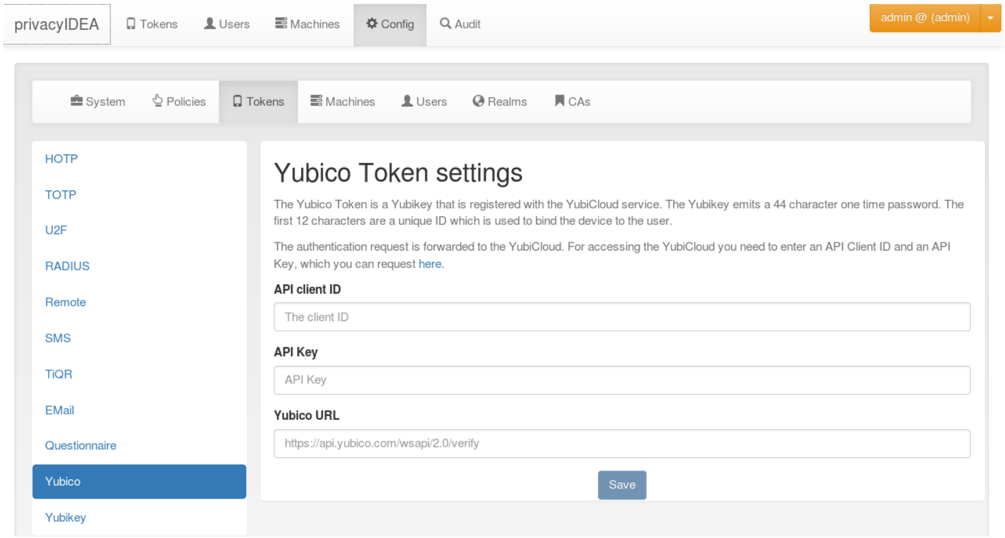
Relying Party Name

This is a human-readable name to go along with the relying party ID. It will usually be either the name of your company (if there is just one relying party for the entire company), or the name of the department or other organizational unit the relying party represents.

See also: *webauthn_relying_party_name*.

Yubico Cloud mode

The Yubico Cloud mode sends the One Time Password emitted by the yubikey to the Yubico Cloud service or another (possibly self hosted) validation server.



Yubico Token settings

The Yubico Token is a Yubikey that is registered with the YubiCloud service. The Yubikey emits a 44 character one time password. The first 12 characters are a unique ID which is used to bind the device to the user.

The authentication request is forwarded to the YubiCloud. For accessing the YubiCloud you need to enter an API Client ID and an API Key, which you can request here.

API client ID: The client ID

API Key: API Key

Yubico URL: https://api.yubico.com/wsapi/2.0/verify

Fig. 28: *Configure the Yubico Cloud mode*

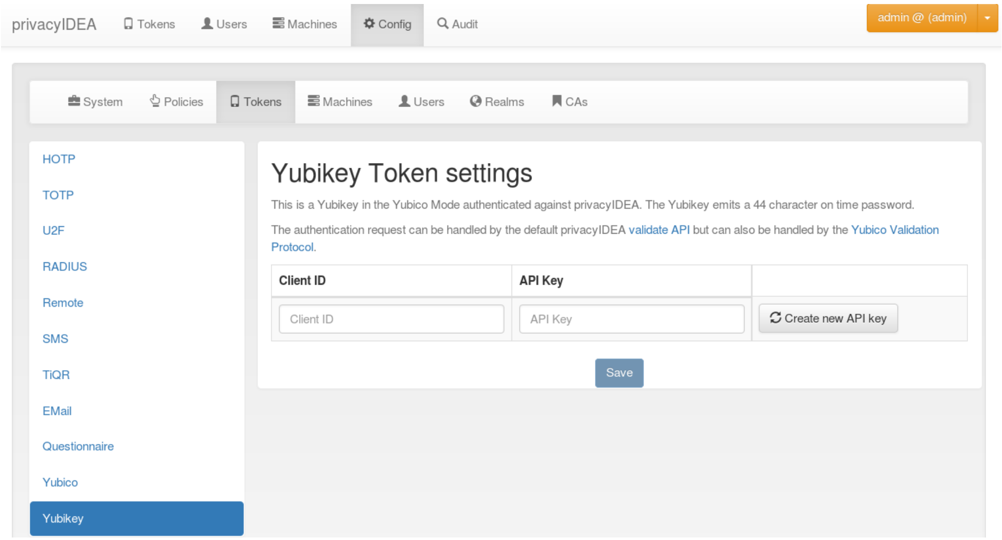
To contact the Yubico Cloud service you need to get an API key and a Client ID from Yubico and enter these here in the config dialog. In that case you can leave the Yubico URL blank and privacyidea will use the Yubico servers.

You can use another validation host, e.g. a self hosted validation server. If you use privacyidea token type yubikey, you can use the URL https://<privacyideaserver>/ttype/yubikey, other validation servers might use https://<validationserver>/wsapi/2.0/verify. You'll get the Client ID and API key from the configuration of your validation server.

You can get your own API key at[1].

Yubikey AES mode

The Yubico AES mode uses the same kind of token as the Yubico Cloud service, but validates the OTP in your local privacyidea server. So the secrets stay local to your system and are not stored in Yubico's Cloud service.



Yubikey Token settings

This is a Yubikey in the Yubico Mode authenticated against privacyIDEA. The Yubikey emits a 44 character on time password.

The authentication request can be handled by the default privacyIDEA validate API but can also be handled by the Yubico Validation Protocol.

Client ID	API Key	
Client ID	API Key	Create new API key

Save

Fig. 29: *Configure the Yubikey AES mode*

You can have more than one Client with a Client ID connect to your server. The Client ID starts with yubikey.apiid. and is followed by the API ID, which you'll need to configure your clients. With `create new API key` you generate a new API for that specific Client ID. The API key is used to sign the validation request sent to the server and the server signs the answer too. That way tampering or MITM attacks might be detected. It is possible to validate token without the API key, but then the request and answer can't be verify against the key. It is useful to use HTTPS for your validation requests, but this is another kind of protection.

OTP validation can either use the privacyidea API /validate/check or the Yubikey validation protocol /ttype/yubikey or - if enabled in your webserver configuration - /wsapi/2.0/verify.

1.5.8 privacyIDEA Appliance

privacyIDEA offers an appliance tool to manage your token administrators, RADIUS clients and also setup MySQL master-master replication. It can be found in a Github repository[1].

This tool is supposed to run on Ubuntu 16.04 LTS. You can find a ready install ISO at another Github reposity[2].

[1] https://upgrade.yubico.com/getapikey/.

[1] https://github.com/NetKnights-GmbH/privacyidea-appliance

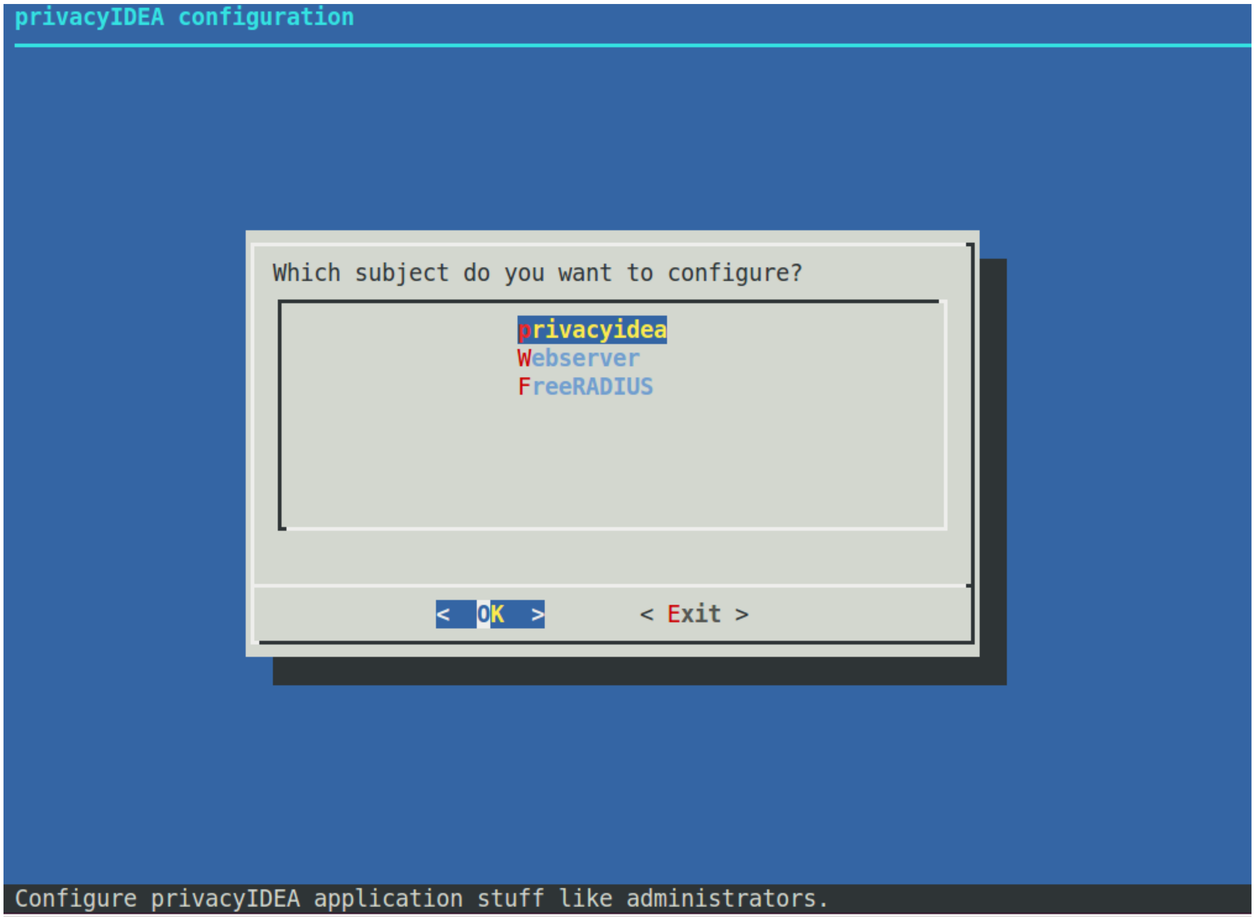
[2] https://github.com/NetKnights-GmbH/privacyidea-appliance-iso

Note: The ready made Ubuntu package for the appliance tool is only available with a Service Level Agreement from the company NetKnights[3].

To configure the system, login as the user root on your machine and run the command:

```
pi-appliance
```

This will bring you to this start screen.



privacyIDEA configuration

Which subject do you want to configure?

Configure privacyIDEA application stuff like administrators.

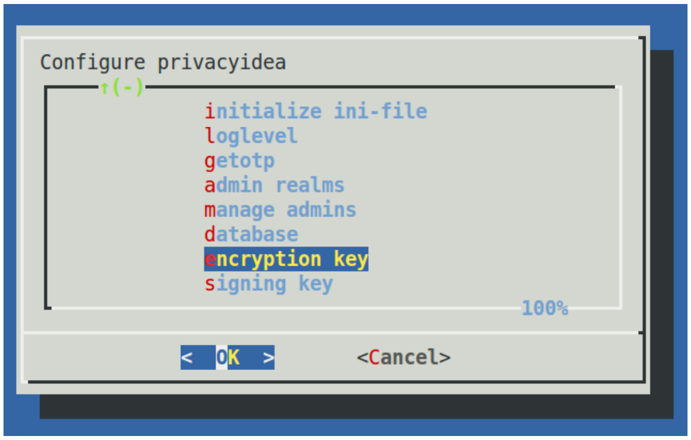
Fig. 30: Start screen of the appliance setup tool.

You can configure privacyidea settings, the log level, administrators, encryption key and much more. You can configure the webserver settings and RADIUS clients.

All changes done in this setup tool are directly read from and written to the corresponding configuration files. The setup tool parses the original nginx and freeradius configuration files. So there is no additional place where this data is kept.

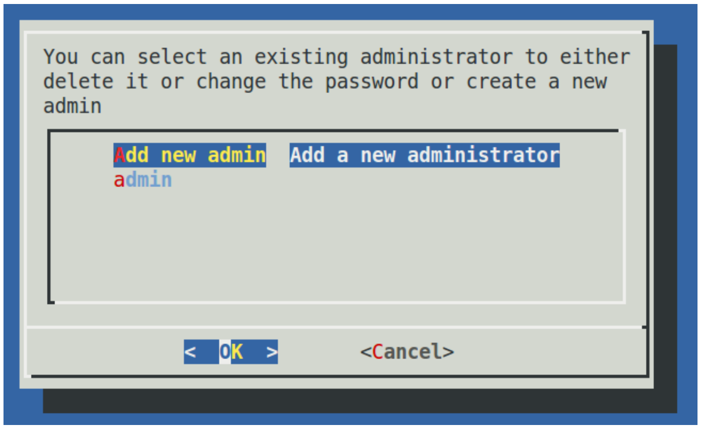
Note: You can also edit the clients.conf and other configuration files manually. The setup tool will also read those manual changes!

[3] https://netknights.it/en/produkte/privacyidea/



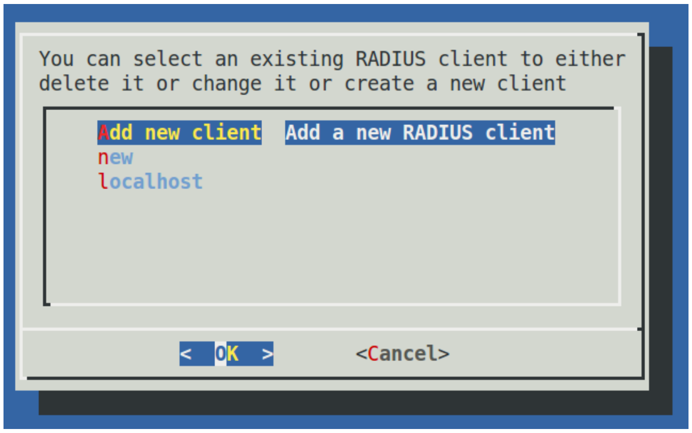
Configure privacyidea

Fig. 31: Configure privacyidea



You can select an existing administrator to either delete it or change the password or create a new admin

Fig. 32: You can create new token administrators, delete them and change their passwords.



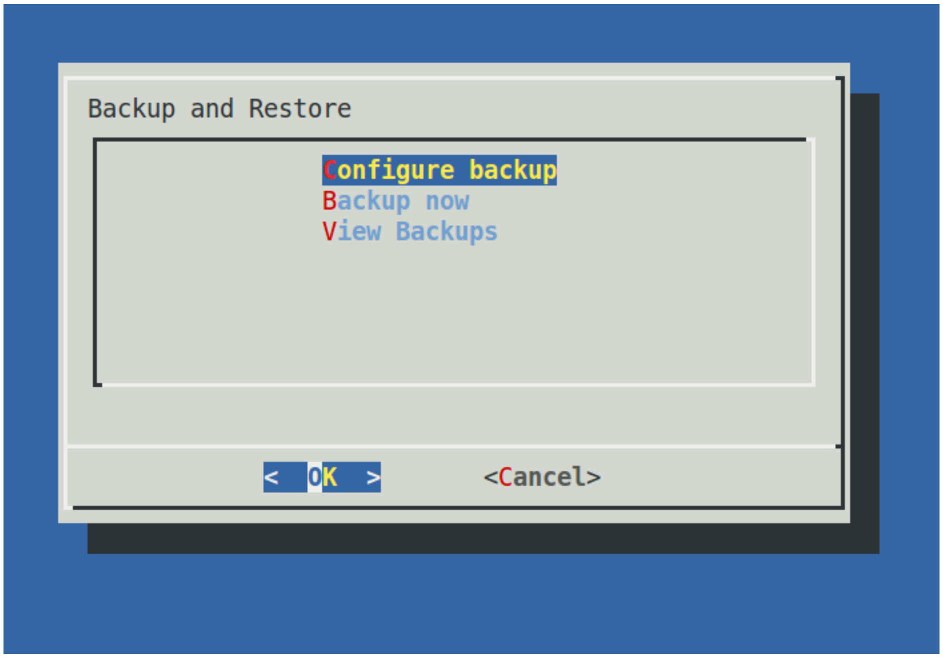
You can select an existing RADIUS client to either delete it or change it or create a new client

Fig. 33: In the FreeRADIUS settings you can create and delete RADIUS clients.

Backup and Restore

Starting with version 1.5 the setup tool also supports backup and restore. Backups are written to the directory */var/lib/privacyidea/backup*.

The backup contains all privacyIDEA configuration, the contents of the directory */etc/privacyidea*, the encryption key, the configured administrators, the complete token database (MySQL) and Audit log. Furthermore if you are running FreeRADIUS the backup also contains the */etc/freeradius/clients.conf* file.



Schedulded backup

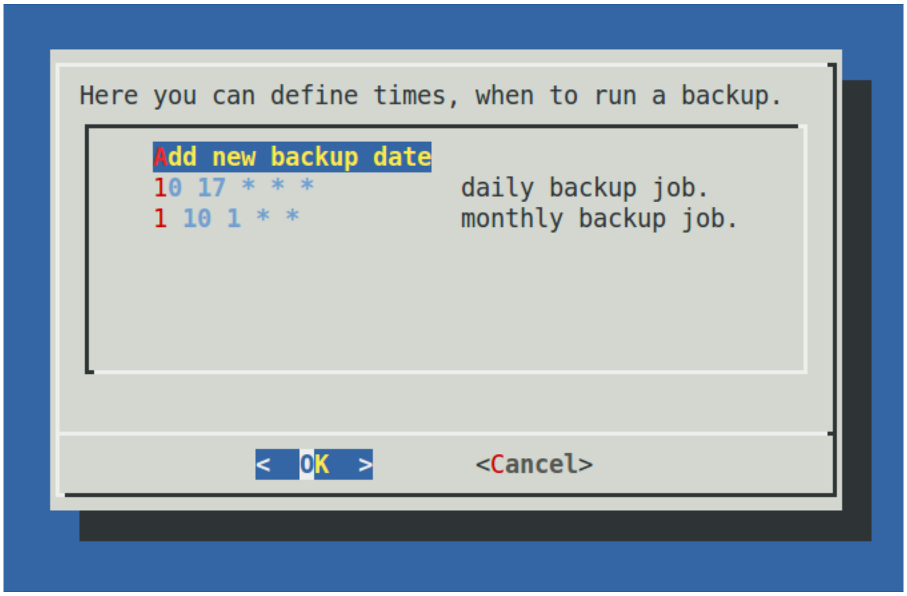
At the configuration point *Configure Backup* you can define times when a scheduled backup should be performed. This information is written to the file */etc/crontab*.

You can enter minutes, hours, day of month, month and day of week. If the entry should be valid for each e.g. month or hour, you need to enter a '*'.

In this example the *10 17 * * ** (minute=10, hour=17) means to perform a backup each day and each month at 17:10 (5:10pm).

The example *1 10 1 * ** (minute=1, hour=10, day of month=1) means to perform a backup on the first day of each month at 10:01 am.

Thus you could also perform backups only once a week at the weekend.



```
Here you can define times, when to run a backup.

    Add new backup date
    10 17 * * *           daily backup job.
    1 10 1 * *            monthly backup job.
```

Fig. 34: Scheduled backup

Immediate backup

If you want to run a backup right now you can choose the entry *Backup now*.

Restore

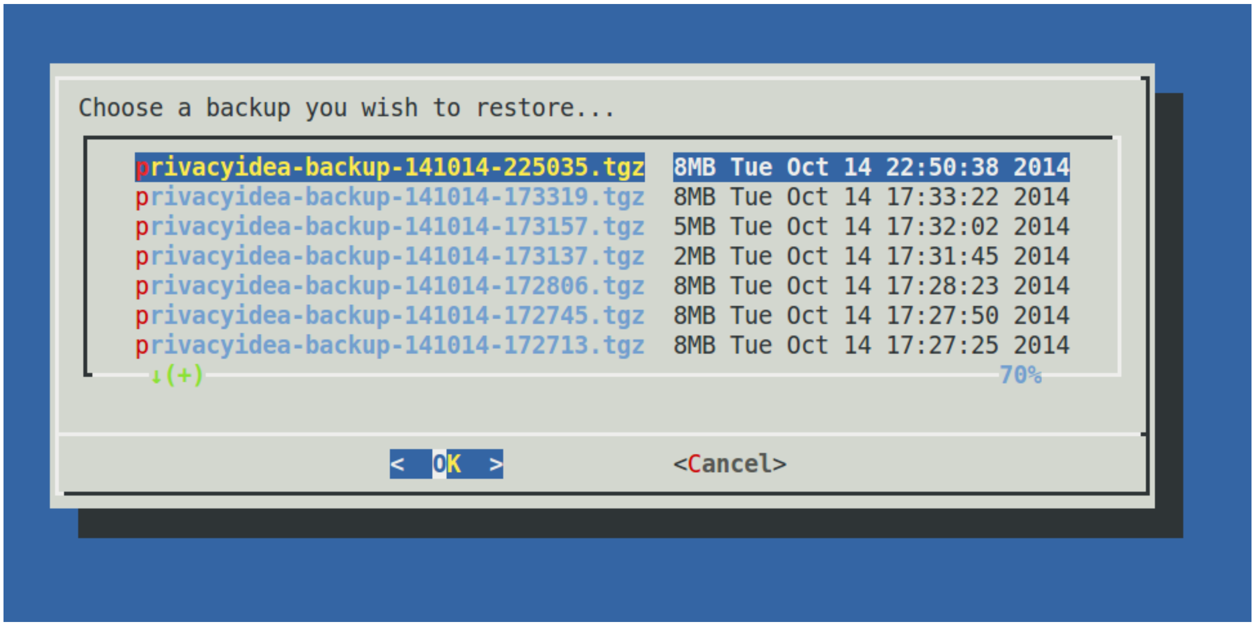
The entry *View Backups* will list all the backups available.

You can select a backup and you are asked if you want to restore the data.

Warning: Existing data is overwritten and will be lost.

1.6 Tokens

PrivacyIDEA is a token management system which supports a great variety of different token types. They each have different requirements concerning configuration and how the authentication works. This chapter explains the authentication modes, lists the supported hardware and software tokens and explains how the token types can be used with privacyIDEA.



```
Choose a backup you wish to restore...

  privacyidea-backup-141014-225035.tgz  8MB Tue Oct 14 22:50:38 2014
  privacyidea-backup-141014-173319.tgz  8MB Tue Oct 14 17:33:22 2014
  privacyidea-backup-141014-173157.tgz  5MB Tue Oct 14 17:32:02 2014
  privacyidea-backup-141014-173137.tgz  2MB Tue Oct 14 17:31:45 2014
  privacyidea-backup-141014-172806.tgz  8MB Tue Oct 14 17:28:23 2014
  privacyidea-backup-141014-172745.tgz  8MB Tue Oct 14 17:27:50 2014
  privacyidea-backup-141014-172713.tgz  8MB Tue Oct 14 17:27:25 2014
  ↓(+)                                                        70%
```

Fig. 35: All available backups

1.6.1 Authentication Modes

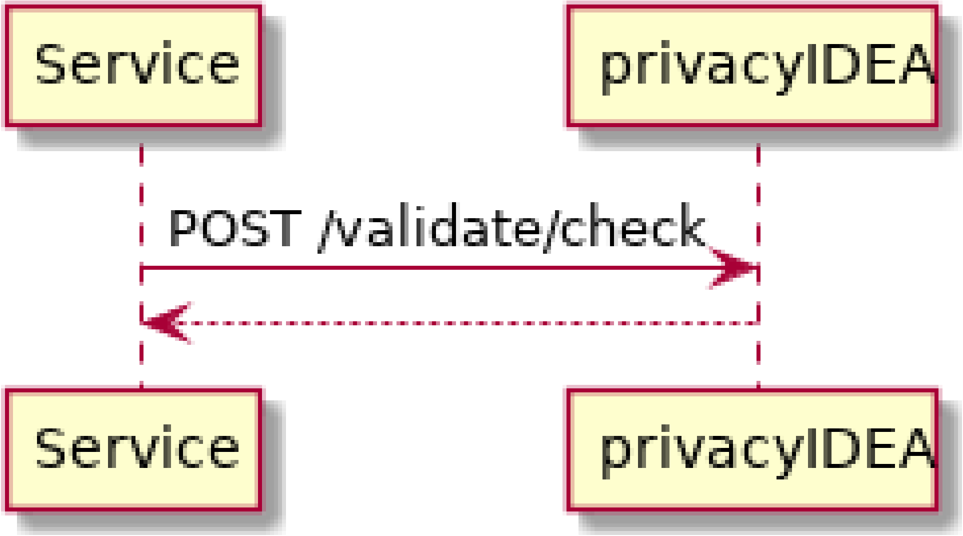
privacyIDEA supports a variety of tokens that implement different authentication flows. We call these flows *authentication modes*. Currently, tokens may implement three authentication modes, namely `authenticate`, `challenge` and `outofband`.

Application plugins need to implement the three authentication modes separately, as the modes differ in their user experience. For example:

- The HOTP token type implements the `authenticate` mode, which is a single-shot authentication flow. For each authentication request, the user uses their token to generate a new HOTP value and enters it along with their OTP PIN. The plugin sends both values to privacyIDEA, which decides whether the authentication is valid or not.
- The E-Mail and SMS token types implement the `challenge` mode. With such a token, the authentication flow consists of two steps: In a first step, the plugin triggers a challenge. privacyIDEA sends the challenge response — a fresh OTP value — to the user via E-Mail or SMS. In a second step, the user responds to the challenge by entering the respective OTP value in the plugin's login form. The plugin sends the challenge response to privacyIDEA, which decides whether the authentication is valid or not.
- The PUSH and TiQR token types implement the `outofband` mode. With a PUSH token, the authentication step also consists of two steps: In a first step, the user triggers a challenge. privacyIDEA pushes the challenge to the user's smartphone app. In a second step, the user approves the challenge on their phone, and the app responds to the challenge by communicating with the privacyIDEA server on behalf of the user. The plugin periodically queries privacyIDEA to check if the challenge has been answered correctly and the authentication is valid.

The following describes the authentication flows of the three authentication modes in more detail.

`authenticate` mode



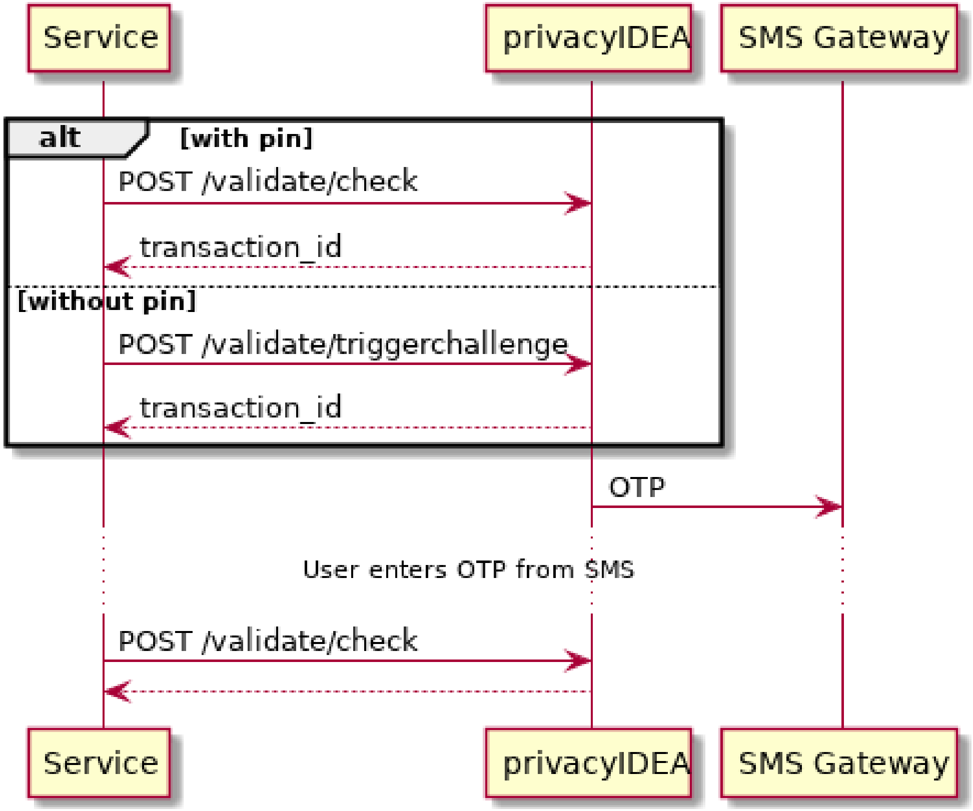
The *Service* is an application that is protected with a second factor by privacyIDEA.

- The user enters a OTP PIN along with an OTP value at the *Service*.
- The plugin sends a request to the `/validate/check` endpoint of privacyIDEA:

  ```
  POST /validate/check

  user=<user>&pass=<PIN+OTP>
  ```

  and privacyIDEA returns whether the authentication request has succeeded or not.

`challenge` mode



- The plugin triggers a challenge, for example via the `/validate/triggerchallenge` endpoint:

  ```
  POST /validate/triggerchallenge

  user=<user>
  ```

  Alternatively, a challenge can be triggered via the `/validate/check` endpoint with the PIN of a challenge-response token:

  ```
  POST /validate/check

  user=<user>&pass=<PIN>
  ```

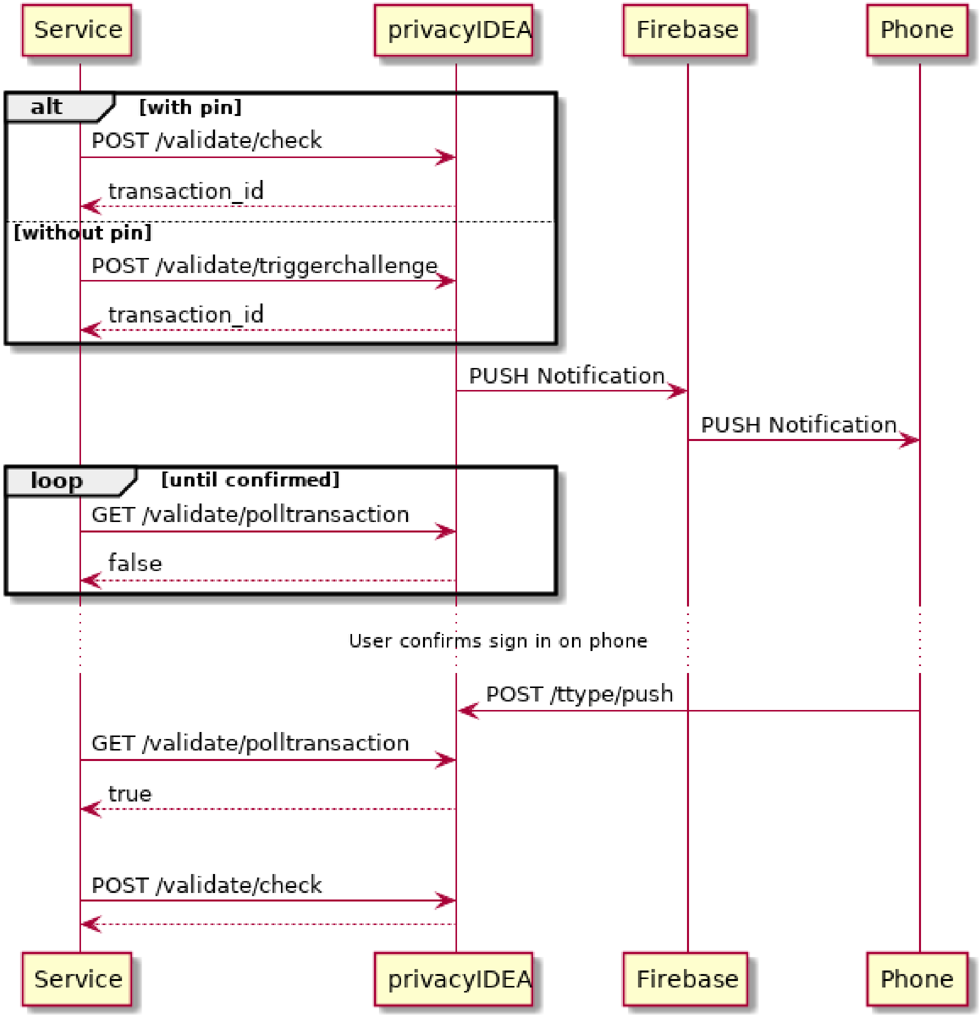
  In both variants, the plugin receives a transaction ID which we call `transaction_id` and asks the user for the challenge response.

- The user enters the challenge response, which we call `OTP`. The plugin forwards the response to privacyIDEA along with the transaction ID:

  ```
  POST /validate/check

  user=<user>&transaction_id=<transaction_id>&pass=<OTP>
  ```

  and privacyIDEA returns whether the authentication request succeeded or not.

`outofband` mode



- The plugin triggers a challenge, for example via the `/validate/triggerchallenge` endpoint:

```
POST /validate/triggerchallenge

user=<user>
```

or via the `/validate/check` endpoint with the PIN of a out-of-band token:

```
POST /validate/check

user=<user>&pass=<PIN>
```

In both variants, the plugin receives a transaction ID which we call `transaction_id`. The plugin may now periodically query the status of the challenge by polling the `/validate/polltransaction` endpoint:

```
GET /validate/polltransaction

transaction_id=<transaction_id>
```

If this endpoint returns `false`, the challenge has not been answered yet.

- The user approves the challenge on a separate device, e.g. their smartphone app. The app communicates with a tokentype-specific endpoint of privacyIDEA, which marks the challenge as answered. The exact communication depends on the token type.
- Once `/validate/polltransaction` returns `true`, the plugin *must* finalize the authentication via the `/validate/check` endpoint:

```
POST /validate/check

user=<user>&transaction_id=<transaction_id>&pass=
```

For the `pass` parameter, the plugin sends an empty string.

This step is crucial because the `/validate/check` endpoint takes defined authentication and authorization policies into account to decide whether the authentication was successful or not.

Note: The `/validate/polltransaction` endpoint does not require authentication and does not increase the failcounters of tokens. Hence, attackers may try to brute-force transaction IDs of correctly answered challenges. Due to the short expiration timeout and the length of the randomly-generated transaction IDs, it is unlikely that attackers correctly guess a transaction ID in time. Nonetheless, plugins must not allow users to inject transaction IDs, and plugins must not leak transaction IDs to users.

1.6.2 Hardware and Software Tokens

privacyIDEA supports a wide variety of tokens by different hardware vendors. It also supports token apps on the smartphone which handle software tokens.

Tokens not listed, will be probably supported, too, since most tokens use standard algorithms.

If in doubt drop your question on the mailing list.

Hardware Tokens

The following hardware tokens are known to work well.

Yubikey. The Yubikey is supported in all modes: AES (*Yubikey*), *HOTP Token* and *Yubico* Cloud. You can initialize the Yubikey yourself, so that the secret key is not known to the vendor.

eToken Pass. The eToken Pass is a push button token by SafeNet. It can be initialized with a special hardware device. Or you get a seed file, that you need to import to privacyIDEA. The eToken Pass can run as *HOTP Token* or *TOTP* token.

eToken NG OTP. The eToken NG OTP is a push button token by SafeNet. As it has a USB connector, you can initialize the token via the USB connector. Thus the hardware vendor does not know the secret key.

DaPlug. The DaPlug token is similar to the Yubikey and can be initialized via the USB connector. The secret key is not known to the hardware vendor.

Smartdisplayer OTP Card. This is a push button card. It features an eInk display, that can be read very good in all light condition at all angles. The Smartdisplayer OTP card is initialized at the factory and you get a seed file, that you need to import to privacyIDEA.

Feitian. The C100 and C200 tokens are classical, reasonably priced push button tokens. The C100 is an *HOTP Token* token and the C200 a *TOTP* token. These tokens are initialized at the factory and you get a seed file, that you need to import to privacyIDEA.

U2F. The Yubikey and the Daplug token are known U2F devices to work well with privacyIDEA. See *U2F*.

Smartphone Apps

privacyIDEA Authenticator. Our own privacyIDEA Authenticator is based on the concept of the Google Authenticator and works with the usual QR Code key URI enrollment. But on top it also allows for a more secure enrollment process (See *Two Step Enrollment*). It can be used for *HOTP Token*, *TOTP* and *Push Token*.

Google Authenticator. The Google Authenticator is working well in *HOTP Token* and *TOTP* mode. If you choose "Generate OTP Key on the Server" during enrollment, you can scan a QR Code with the Google Authenticator. See *Enrolling your first token* to learn how to do this.

FreeOTP. privacyIDEA is known to work well with the FreeOTP App. The FreeOTP App is a *TOTP* token. So if you scan the QR Code of an HOTP token, the OTP will not validate.

mOTP. Several mOTP Apps like "Potato", "Token2" or "DroidOTP" are supported.

1.6.3 Token types in privacyIDEA

The following list is an overview of the supported token types. For more details, consult the respective description listed in *Tokens*. Some token require prior configuration as described in *Token type details*.

- *Four Eyes* - Meta token that can be used to create a Two Man Rule.
- *Certificate Token* - A token that represents a client certificate.
- *Email* - A token that sends the OTP value to the EMail address of the user.
- *HOTP Token* - event based One Time Password tokens based on RFC4226.
- *Indexed Secret Token* - a challenge response token that asks the user for random positions from a secret string.
- Daplug - A hardware OTP token similar to the Yubikey.
- *mOTP Token* - time based One Time Password tokens for mobile phones based on an a public Algorithm.
- *OCRA* - A basic OATH Challenge Response token.
- *Paper Token (PPR)* - event based One Time Password tokens that get you list of one time passwords on a sheet of paper.
- *Push Token* - A challenge response token, that sends a challenge to the user's smartphone and the user simply accepts the request to login.
- *Password Token* - A password token used for *losttoken* scenario.
- *Questionnaire Token* - A token that contains a list of answered questions. During authentication a random question is presented as challenge from the list of answered questions is presented. The user must give the right answer.
- *Registration* - A special token type used for enrollment scenarios (see *Registration Code*).
- *RADIUS* - A virtual token that forwards the authentication request to a RADIUS server.
- registration
- *Remote* - A virtual token that forwards the authentication request to another privacyIDEA server.
- *SMS Token* - A token that sends the OTP value to the mobile phone of the user.
- *Spass - Simple Pass Token* - The simple pass token. A token that has no OTP component and just consists of the OTP pin or (if otppin=userstore is set) of the userstore password.
- *SSH Keys* - An SSH public key that can be managed and used in conjunction with the *Machines* concept.
- *TAN Token* -

- *TiQR* - A Smartphone token that can be used to login by only scanning a QR code.
- *TOTP* - time based One Time Password tokens based on RFC6238.
- *U2F* - A U2F device as specified by the FIDO Alliance. This is a USB device to be used for challenge response authentication.
- *VASCO* - The proprietary VASCO token.
- *WebAuthn* - The WebAuthn or FIDO2 token which can use several different mechanisms like USB tokens or TPMs to authenticate via public key cryptography.
- *Yubikey* - A Yubikey hardware initialized in the AES mode, that authenticates against privacyIDEA.
- *Yubico* - A Yubikey hardware that authenticates against the Yubico Cloud service.

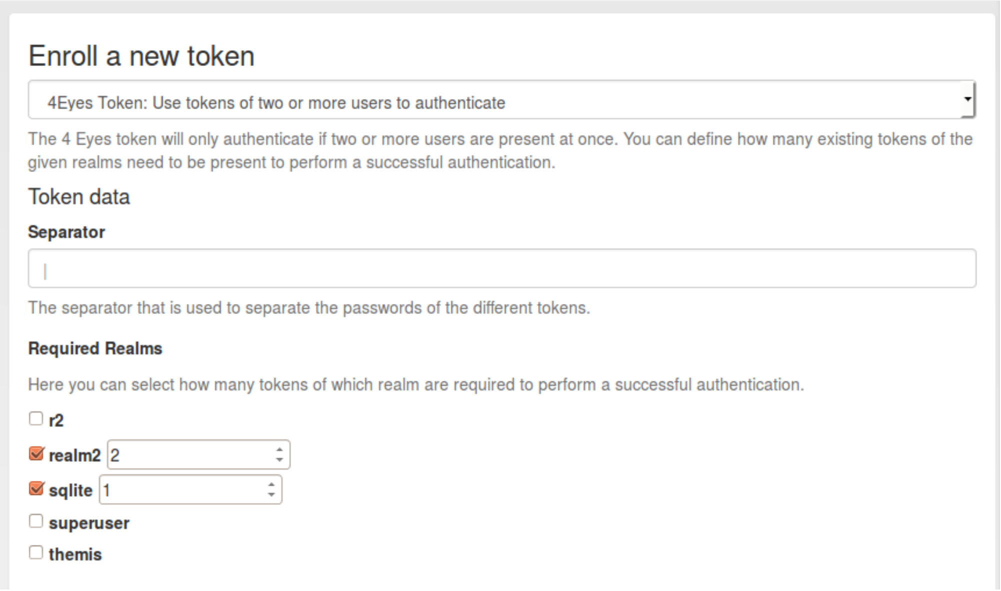
Token type details

Detailed information on the different token types used in privacyIDEA can be found in the following sections.

Four Eyes

Starting with version 2.6 privacyIDEA supports 4 Eyes Token. This is a meta token, that can be used to define, that two or more token must be used to authenticate. This way, you can set up a "two man rule".

You can define, from which realm how many unique tokens need to be present, when authenticating:



Enroll a new token

4Eyes Token: Use tokens of two or more users to authenticate

The 4 Eyes token will only authenticate if two or more users are present at once. You can define how many existing tokens of the given realms need to be present to perform a successful authentication.

Token data

Separator

|

The separator that is used to separate the passwords of the different tokens.

Required Realms

Here you can select how many tokens of which realm are required to perform a successful authentication.

☐ r2

☑ realm2 2

☑ sqlite 1

☐ superuser

☐ themis

Fig. 36: *Enroll a 4 eyes token*

In this example authentication will only be possbile if at least two tokens from *realm2* and one token from realm *sqlite* are present.

Authentication is done by concatenating the OTP PINs and the OTP values of all tokens. The concatenation is split by the *separator* character.

It does not matter, in which order the tokens from the realms are entered.

Example

Authentication as:

```
username: "root@r2"
password: "pin123456 secret789434 key098123"
```

The three blocks separated by the *blank* are checked, if they match tokens in the realms *realm2* and *sqlite*.

The response looks like this in case of success:

```
{
  "detail": {
    "message": "matching 1 tokens",
    "serial": "PI4E000219E1",
    "type": "4eyes"
  },
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": true
  },
  "version": "privacyIDEA 2.6dev0",
  "versionnumber": "2.6dev0"
}
```

In case of a failed authentication the response looks like this:

```
{
  "detail": {
    "foureyes": "Only found 0 tokens in realm themis",
    "message": "wrong otp value",
    "serial": "PI4E000219E1",
    "type": "4eyes"
  },
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": false
  },
  "version": "privacyIDEA 2.6dev0",
  "versionnumber": "2.6dev0"
}
```

Note: The 4Eyes Token verifies that unique tokens from each realm are used. I.e. if you require 2 tokens from a realm, you can not use the same token twice.

Warning: But it does not verify, if these two unique tokens belong to the same user. Thus you should create a poliy, that in such a realm a user may only have on token.

Certificate Token

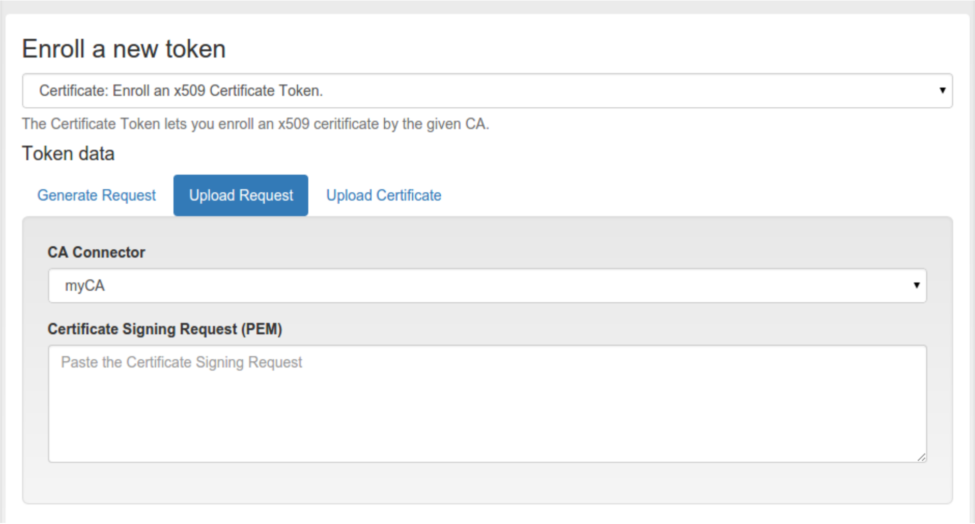
Starting with version 2.3 privacyIDEA supports certificates. A user can

- upload a certificate request,
- upload a certificate or
- he can generate a certificate request in the browser.

privacyIDEA does not sign certificate requests itself but connects to existing certificate authorities. To do so, you need to define *CA Connectors*.

Certificates are attached to the user just like normal tokens. One token of type *certificate* always contains only one certificate.

If you have defined a CA connector you can upload a certificate signing request (CSR) via the *Token Enroll Dialog* in the WebUI.



Enroll a new token

Certificate: Enroll an x509 Certificate Token.

The Certificate Token lets you enroll an x509 ceritificate by the given CA.

Token data

CA Connector: myCA

Certificate Signing Request (PEM): Paste the Certificate Signing Request

Fig. 37: *Upload a certificate signing request*

You need to choose the CA connector. The certificate will be signed by the CA accordingly. Just like all other tokens the certificate token can be attached to a user.

Generating Signing Requests

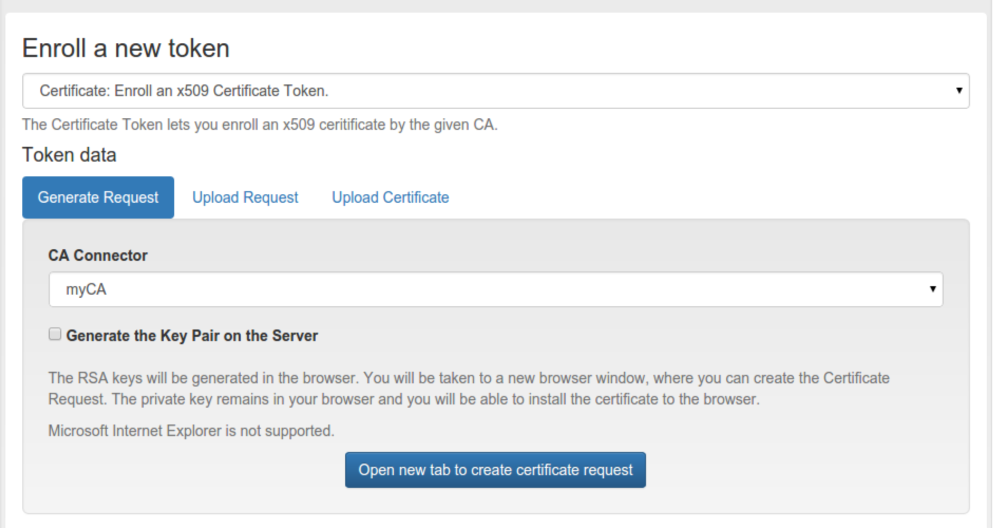
You can also generate the signing request directly in your browser.

Note: This uses the keygen HTML-tag that is not supported by the Internet Explorer!

When generating the certificate signing request this way the RSA keypair is generated on the client side in the browser.

The certificate is signed by the CA connected by the chosen CA connector.

Afterwards the user can install the certificate into the browser.



Enroll a new token

Certificate: Enroll an x509 Certificate Token.

The Certificate Token lets you enroll an x509 ceritificate by the given CA.

Token data

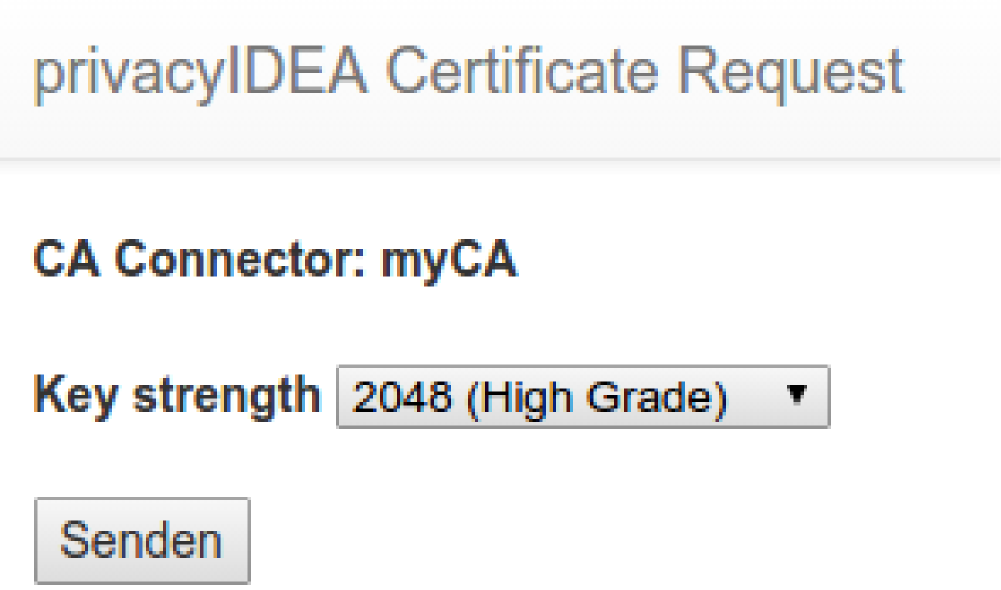
CA Connector: myCA

□ Generate the Key Pair on the Server

The RSA keys will be generated in the browser. You will be taken to a new browser window, where you can create the Certificate Request. The private key remains in your browser and you will be able to install the certificate to the browser.

Microsoft Internet Explorer is not supported.

Fig. 38: *Generate a certificate signing request*



privacyIDEA Certificate Request

CA Connector: myCA

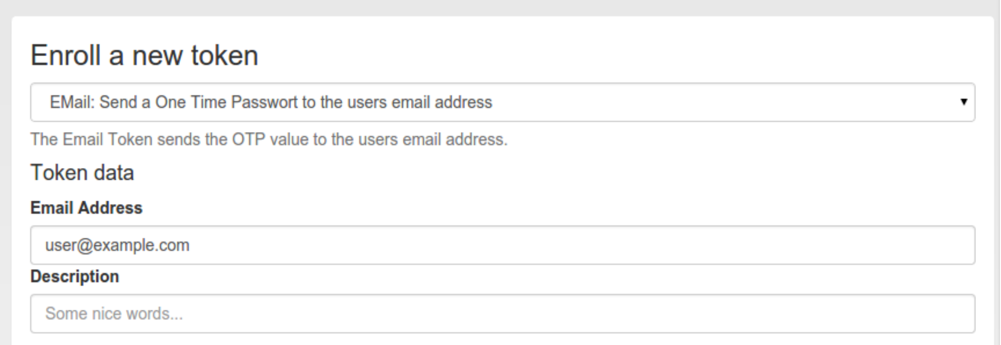
Key strength: 2048 (High Grade)

Fig. 39: *Download or install the client certificate*

Note: By requiring OTP authentication for the users to login to the WebUI (see *login_mode*) you can have two factor authentication required for the user to be allowed to enroll a certificate.

Email

The token type *email* sends the OTP value in an email to the users. You can configure the email server in *Email Token Configuration*.



Enroll a new token

EMail: Send a One Time Passwort to the users email address

The Email Token sends the OTP value to the users email address.

Token data

Email Address

user@example.com

Description

Some nice words...

Fig. 40: *Enroll an email token*

When enrolling an email token, you only need to specify the email address of the user.

The email token is a challenge response token. I.e. when using the OTP PIN in the first authentication request, the sending of the email will be triggered and in a second authentication request the OTP value from the email needs to be presented. It implements the *challenge authentication mode*.

For a more detailed insight see the code documentation *Email Token*.

HOTP Token

The HOTP token is - together with the *TOTP* - the most common token. The HOTP Algorithm is defined in RFC4225. The HOTP token is an event base token. The HOTP algorithm has some parameter, like if the generated OTP value will be 6 digits or 8 digits or if the SHA1 oder the SHA256 hashing algorithm is used.

The HOTP token implements the *authenticate mode*. With a suitable *challenge_response* policy, it may also be used in the *challenge mode*.

Hardware tokens

There are many token vendors out there who are using the official algorithm to build and sell hardware tokens. You can get HOTP based hardware tokens in different form factors, as a normal key fob for your key ring or as a display card for your purse.

Preseeded or Seedable

Usually the hardware tokens like keyfobs or display cards contain a secret key that was generated and implanted at the vendors factory. The vendor ships the tokens and a seed file.

> **Warning:** In this case privacyIDEA can not guarantee that the secret seed of the token is unique and if you are using a real strong factor.

privacyIDEA also supports the following seedable HOTP tokens:

- SafeNet eToken NG OTP
- SafeNet eToken Pass
- Yubikey in OATH mode
- Daplug

Those tokens can be initialized by privacyIDEA. Thus you can be sure, that only you are in possession of the secret seed.

Experiences

The above mentioned hardware tokens are known to play well with privacyIDEA. In theory all OATH/HOTP tokens should work well with privacyIDEA. However, there are good experiences with Smartdisplayer OTP cards[1] and Feitian C200[2] tokens.

Software tokens

Besides the hardware tokens there are also software tokens, implemented as Apps for your smartphone. These software tokens allow are seedable, so there is no vendor, knowing the secret seed of your OTP tokens.

But software tokens are software after all on device prone to security issues.

Experiences

The Google Authenticator can be enrolled easily in HOTP mode using the QR-Code enrollment Feature.

The Google Authenticator is available for iOS, Android and Blackberry devices.

Enrollment

Default settings for HOTP tokens can be configured at *HOTP Token Config*.

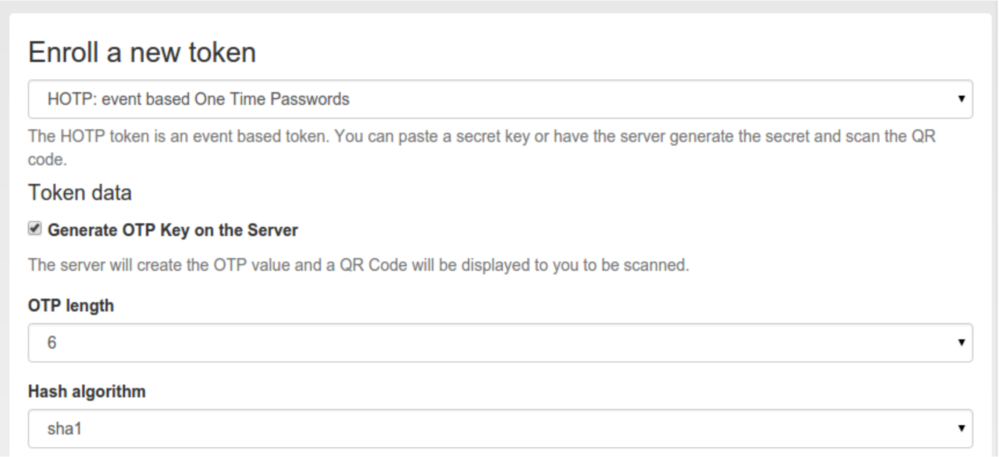
During enrollment you can choose, if the server should generate the key or if you have a key, that you can enter into the enrollment page.

As mentioned earlier, you can also choose the **OTP length** and the **hash algoriothm**.

After enrolling the token, the QR-Code, containing the secret seed, is displayed, so that you can scan this with your smartphone and import it to your app.

[1] https://netknights.it/en/produkte/smartdisplayer/

[2] https://netknights.it/en/produkte/oath-hotptotp/



Enroll a new token

HOTP: event based One Time Passwords ▾

The HOTP token is an event based token. You can paste a secret key or have the server generate the secret and scan the QR code.

Token data

☑ **Generate OTP Key on the Server**

The server will create the OTP value and a QR Code will be displayed to you to be scanned.

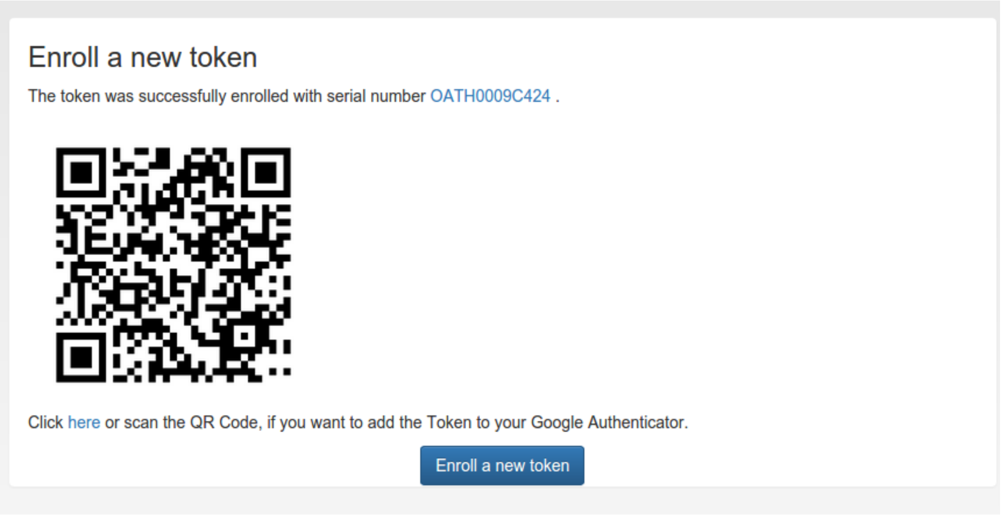
OTP length

6 ▾

Hash algorithm

sha1 ▾

Fig. 41: *Enroll an HOTP token*



Enroll a new token

The token was successfully enrolled with serial number OATH0009C424 .

Click here or scan the QR Code, if you want to add the Token to your Google Authenticator.

Fig. 42: *If the server generated the secret seed, you can scan the QR-Code*

Indexed Secret Token

The indexed secret token is a simple challenge response token.

A shared secret like "mySecret" is stored in the privacyIDEA server. When the token is used a challenge is sent to the user like "Give me the 2nd and the 4th position of your secret".

Then the user needs to respond with the concatenated characters from the given positions. In the example the response would be "ye".

Certain policies can be used to either preset or force the value of the indexed secret during enrollment to the value of a user attribute. The attribute specified in these policies is a privacyidea attribute from the attribute mapping of the corresponding user resolver.

mOTP Token

mOTP is a time based One Time Password token for mobile phones based on a public Algorithm.

OCRA

Starting with version 2.20 privacyIDEA supports common OCRA tokens. OCRA tokens can not be enrolled via the UI but need to be imported via a seed file. The OATH CSV seed file would look like this:

```
<serial>, <seed>, ocra, <ocrasuite>
```

The OCRA token is a challenge/response token. So the first authentication request issues a challenge. This challenge is the input for the response of the OCRA token.

For more information see *OCRA Token*.

DisplayTAN token

privacyIDEA supports the DisplayTAN[1], which can be used for securing banking transactions. The OCRA Algorithm is used to digitally sign transaction data. The transcation data can be verified by the user on an external banking card. All cryptographical processes are running on the external card, so that an attacker can not interfere with the user's component.

The DisplayTAN cards would be imported into privacyIDEA using the token import.

A banking website will use the *Validate endpoints* API.

The first call will trigger the challenge response mechanism. The first call needs to contain the transaction data: the recipient's account number and amount of money to transfer:

```
<account>~<amount>~
```

Please note the tilde:

```
POST https://privacyidea.example.com/validate/check

pass=pin
serial=ocra1234
challenge=1234567890~423,40~
```

(continues on next page)

[1] http://www.display-tan.com/

(continued from previous page)

```
addrandomchallenge=20
hashchallenge=sha1
```

This will result in a response like this:

```
{
  "jsonrpc": "2.0",
  "signature": "128057011582042...408",
  "detail": {
             "multi_challenge": [
                {
                 "attributes": {
                 "qrcode": "data:image/png;base64, iVBORw0KG..RK5CYII=",
                 "original_challenge": "83507112  ~320,
               00~cfbGSopfdDROOMjeu3IR",
                 "challenge": "f8a1818f35ae0cc64fe8a191961ec829487dfa82"
                 },
                 "serial": "ocra1234",
                 "transaction_id": "05221757445370623976"
                }
             ],
             "threadid": 139847557760768,
             "attributes": {
             "qrcode": "data:image/png;base64, iVBO...CYII=",
             "original_challenge": "83507112  ~320,00~cfbGSopfdDROOMjeu3IR",
             "challenge": "f8a1818f35ae0cc64fe8a191961ec829487dfa82"
             },
             "message": "Please answer the challenge",
             "serial": "ocra1234",
             "transaction_id": "05221757445370623976"
  },
  "versionnumber": "2.20.dev2",
  "version": "privacyIDEA 2.20.dev2",
  "result": {
             "status": true,
             "value": false
  },
  "time": 1504005837.417481,
  "id": 1
}
```

Note: The response also contains the QR code. The banking website should show the QR code, so that the user can scan it with the DisplayTAN App to transfer the data to the card.

The user can verify the data on the card and transaction data will be digitally signed on the card. The card will calculate an OTP value for this very transaction.

The banking website can now send the OTP value to privacyIDEA to check, if the user authorized the correct transaction data. The banking site will issue this request:

```
POST https://privacyidea.example.com/validate/check

serial=ocra1234
transaction_id=05221757445370623976
pass=54006635
```

privcyIDEA will respond with a usual authentication response:

```
{
 "jsonrpc": "2.0",
 "signature": "162....2454851",
 "detail": {
            "message": "Found matching challenge",
            "serial": "ocra1234",
            "threadid": 139847549368064
           },
 "versionnumber": "2.20.dev2",
 "version": "privacyIDEA 2.20.dev2",
 "result": {
            "status": true,
            "value": true
 },
 "time": 1504005901.823667,
 "id": 1
}
```

Paper Token (PPR)

The token type *paper* lets you print out a list of OTP values, which you can use to authenticate and cross of the list.

The paper token is based on the *HOTP Token*. I.e. you need to use one value after the other.

Customization

CSS

You can customize the look and feel of the printed paper token. You may change the style sheep `papertoken.css` which is only loaded for printing.

Header and Footer

Then you may add a header in front and a footer behind the table containing the OTP values.

Create the files

- static/customize/views/includes/token.enrolled.paper.top.html
- static/customize/views/includes/token.enrolled.paper.bottom.html

to display the contents before (top) and behind (bottom) the table.

Within these html templates you may use angular replacements. To get the serial number of the token use

> {{ tokenEnrolled.serial }}

to get the name and realm of the user use

> {{ newUser.user }} {{ newUser.realm }}

A good example for the `token.enrolled.paper.top.html` is:

> <h1>{{ enrolledToken.serial }}</h1> <p>

Please use the OTP values of your paper token in order one after the other. You may scratch of or otherwise mark used values.

</p>

A good example for the `token.enrolled.paper.bottom.html` is:

<p> The paper token is a weak second factor. Please assure, that no one gets hold of this paper and can make a copy of it.

</p> <p>

Store it at a safe location.

</p>

Note: You can change the directory *static/customize* to a URL that fits your needs the best by defining a variable *PI_CUSTOMIZATION* in the file *pi.cfg*. This way you can put all modifications in one place apart from the original code.

OTP Table

If you want to change the complete layout of the table you need to overwrite the file `static/components/token/views/token.enrolled.paper.html`. The scope variable {{ enrolledToken.otps }} contains an object with the complete OTP value list.

Push Token

The push token uses the *privacyIDEA Authenticator* app. You can get it from Google Play Store or Apple App Store.

The token type *push* sends a cryptographic challenge via the Google Firebase service to the smartphone of the user. This push notification is displayed on the smartphone of the user with a text that tells the user that he or somebody else requests to login to a service. The user can simply accept this request. The smartphone sends a cryptographically signed response to the privacyIDEA server and the login request gets marked as confirmed in the privacyIDEA server. The application checks for this mark and logs the user in automatically. For an example of how the components in a typical deployment of push tokens interact reference the following diagram.

To allow privacyIDEA to send push notifications, a Firebase service needs to be configured. To do so see *Firebase Provider*.

The PUSH token implements the *outofband mode*.

Configuration

The minimum necessary configuration is an `enrollment` policy *push_firebase_configuration*.

With the `authentication` policies *push_text_on_mobile* and *push_title_on_mobile* you can define the contents of the push notification.

If you want to use push tokens with legacy applications that are not yet set up to be compatible with out-of-band tokens, you can set the `authentication` policy *push_wait*. Please note, that setting this policy can interfere with other tokentypes and will impact performance, as detailed in the documentation for `push_wait`.

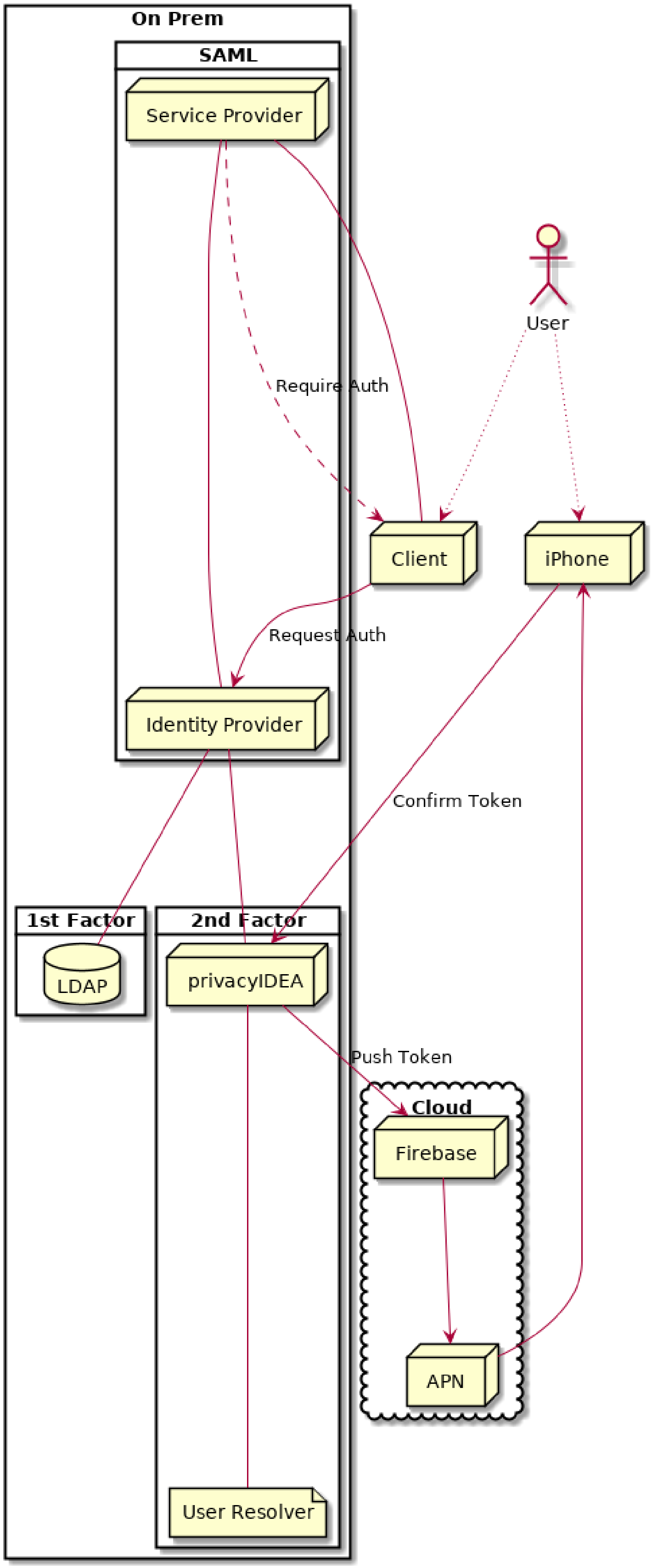


Fig. 43: A typical push token deployment

Enrollment

The enrollment of the push token happens in two steps.

Step 1

The user scans a QR code. This QR code contains the basic information for the push token and a enrollment URL, to which the smartphone should respond in the enrollment process.

The smartphone stores this data and creates a new key pair.

Step 2

The smartphone sends its Firebase ID, the public key of the keypair, the serial number and an enrollment credential back to the enrollment URL of the privacyIDEA server.

The server responds with it's public key for this token.

Authentication

Triggering the challenge

The authentication request is triggered by an application just the same like for any challenge response tokens either with the PIN to the endpoint `/validate/check` or via the endpoint `/validate/triggerchallenge`.

privacyIDEA sends a cryptographic challenge with a signature to the Firebase service. The firebase service sends the notification to the smartphone, which can verify the signature using the public key from enrollment step 2.

Accepting login

The user can now accept the login by tapping on the push notification. The smartphone sends the signed challenge back to the authentication URL of the privacyIDEA server. The privacyIDEA server verifies the response and marks this authentication request as successfully answered.

Login to application

The application can check with the orignial transaction ID with the privacyIDEA server, if the challenge has been successfully answered and automatically login the user.

More information

For a more detailed insight see the code documentation *Push Token*.

For an in depth view of the protocol see the github issue and the wiki page.

For recent information and a setup guide, visit the community blog

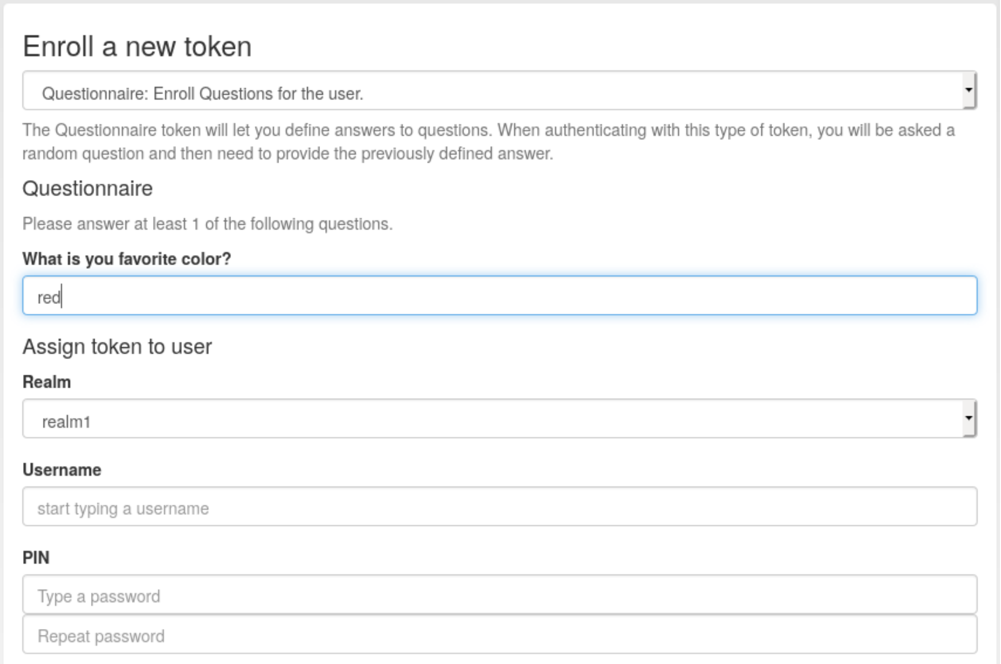
Password Token

This token is not enrolled by the user. It is automatically created whenever an authorized user initiates a *losttoken* scenario.

Questionnaire Token

The administrator can define a list of questions and also how many answers to the questions a user needs to define.

During enrollment of such a *questionnaire* type token, the user answers at least as many questions as specified by the administrator with answers only he knows.



Enroll a new token

Questionnaire: Enroll Questions for the user.

The Questionnaire token will let you define answers to questions. When authenticating with this type of token, you will be asked a random question and then need to provide the previously defined answer.

Questionnaire

Please answer at least 1 of the following questions.

What is you favorite color?

red

Assign token to user

Realm

realm1

Username

start typing a username

PIN

Type a password

Repeat password

This token is a challenge response token. During authentication the user gives the token PIN before he is presented with a random question to which he defined the answer during the token rollout.

Note: By default, no questions are defined, so the administrator has to setup those in “Config->Tokens->Questionnaire” before a questionnaire token can be rolled out successfully.

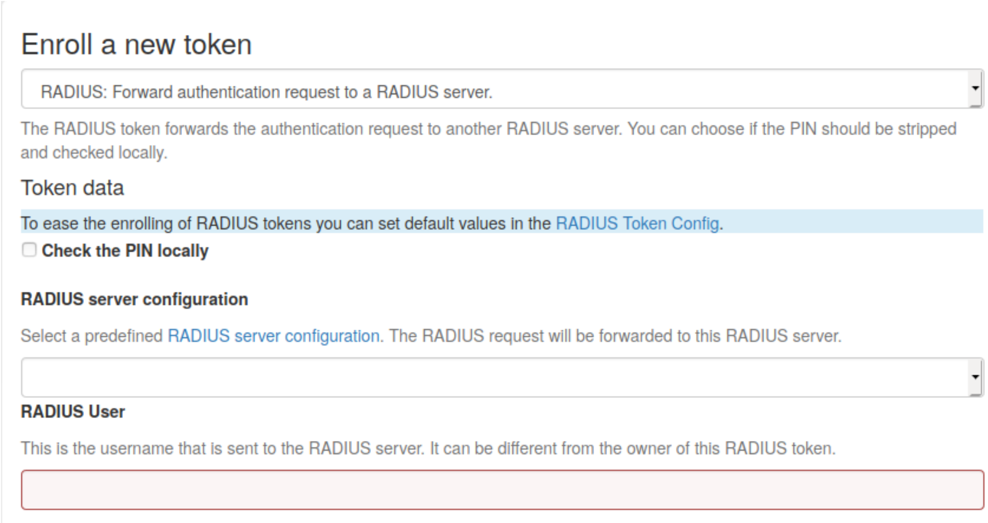
Note: If the administrator changes the questions *after* a token was enrolled, the enrolled token still works with the old questions and answers. I.e. an enrolled token is not affected by changing the questions by the administrator.

Note: As for all token, it is not changed after the rollout (see above note), so a change of the answers of an existing token is not possible.

RADIUS

The token type *RADIUS* forwards the authentication request to a RADIUS Server.

When forwarding the authentication request, you can change the username and mangle the password.



Enroll a new token

RADIUS: Forward authentication request to a RADIUS server.

The RADIUS token forwards the authentication request to another RADIUS server. You can choose if the PIN should be stripped and checked locally.

Token data

To ease the enrolling of RADIUS tokens you can set default values in the RADIUS Token Config.

□ Check the PIN locally

RADIUS server configuration

Select a predefined RADIUS server configuration. The RADIUS request will be forwarded to this RADIUS server.

RADIUS User

This is the username that is sent to the RADIUS server. It can be different from the owner of this RADIUS token.

Fig. 44: *Enroll a RADIUS token*

Check the PIN locally

If checked, the PIN of the token will be checked on the local server. If the PIN matches only the remaining part of the issued password will be sent to the RADIUS server.

RADIUS Server

The RADIUS server, to which the authentication request will be forwarded. You can specify the port like `my.radius.server:1812`.

RADIUS User

When forwarding the request to the RADIUS server, the authentication request will be issued for this user. If the user is left empty, the RADIUS request will be sent with the same user.

RADIUS Secret

The RADIUS secret for this RADIUS client.

Note: Using the RADIUS token you can design migration scenarios. When migrating from other (proprietary) OTP solutions, you can enroll a RADIUS token for the users. The RADIUS token points to the RADIUS server of the old solution. Thus the user can authenticate against privacyIDEA with the old, proprietary token, till he is enrolled a new token in privacyIDEA. The interesting thing is, that you also get the authentication request with the proprietary token in the audit log of privacyIDEA. This way you can have a scenario, where users are still using old tokens and other users are already using new (privacyIDEA) tokens. You will see all authentication requests in the pricacyIDEA system.

Registration

(See *Registration Code*)

The registration token can be used to create a registration code for a user. This registration code can be sent via postal mail to the user, so that the user can use this registration code as a second factor to login to a portal.

After a one single use, the registration code is deleted and can not be used a second time.

Note: The registration code can only be enrolled via the API to provide automated smooth workflow to your needs.

For a more detailed insight see the code documentation *Registration Code Token*.

Remote

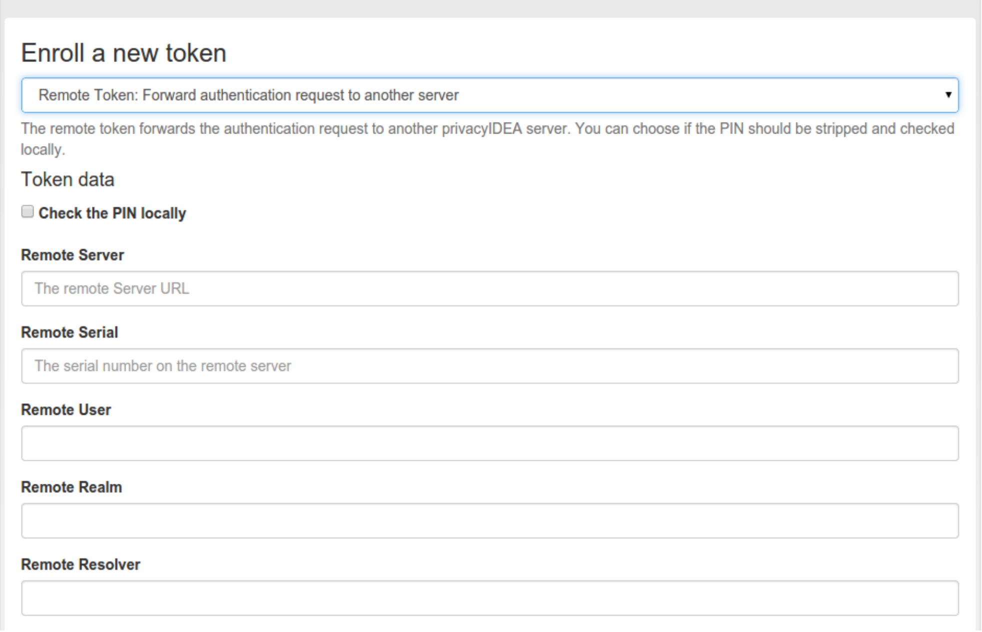
The token type *remote* forwards the authentication request to another privacyIDEA Server.

When forwarding the authentication request, you can

- change the username
- change the resolver
- change the realm
- change the serial number

and mangle the password.

The serial number of the token, that was used on the other privacyIDEA server, is stored in the tokeninfo of the remote token object in the key `last_matching_remote_serial`. This serial number can then be used in further workflows and e.g. be processed in event handlers.



Enroll a new token

Remote Token: Forward authentication request to another server ▾

The remote token forwards the authentication request to another privacyIDEA server. You can choose if the PIN should be stripped and checked locally.

Token data

□ Check the PIN locally

Remote Server: The remote Server URL

Remote Serial: The serial number on the remote server

Remote User:

Remote Realm:

Remote Resolver:

Fig. 45: *Enroll a Remote token*

Check the PIN locally

If checked, the PIN of the token will be checked on the local server. If the PIN matches only the remaining part of the issued password will be sent to the remote privacyIDEA server.

Remote Server

The privacyIDEA server, to which the authentication request will be forwarded. The path `/validate/check` will be added automatically. So a sensible input would be `https://my.other.server/`.

Remote Serial

If the *Remote Serial* is specified the given password will be checked against the serial number on the remote privacyIDEA server. Usernames will be ignored.

Remote User

When forwarding the request to the remote server, the authentication request will be issued for this user.

Remote Realm

When forwarding the request to the remote server, the authentication request will be issued for this realm.

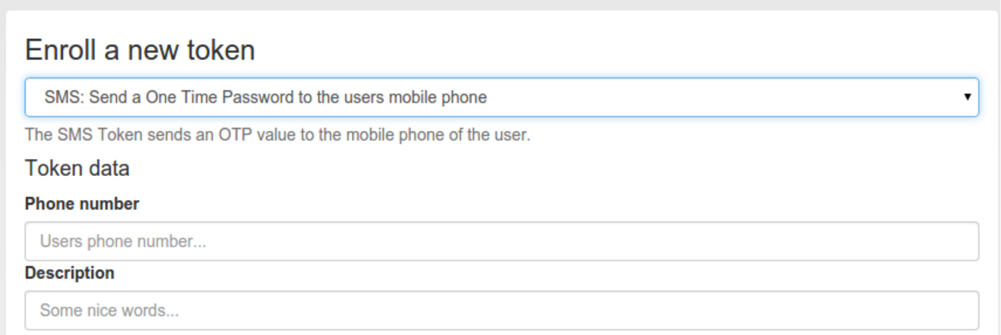
Remote Resolver

When forwarding the request to the remote server, the authentication request will be issued for this resolver.

Note: You can use *Remote Serial* to forward the request to a central privacyIDEA server, that only knows tokens but has no knowledge of users. Or you can use *Remote Serial* to forward the request to an existing to on *localhost* thus adding a second user to the same token.

SMS Token

The token type *sms* sends the OTP value via an SMS service. You can configure the SMS service in *SMS Token Configuration*.



Enroll a new token

SMS: Send a One Time Password to the users mobile phone ▾

The SMS Token sends an OTP value to the mobile phone of the user.

Token data

Phone number

Users phone number...

Description

Some nice words...

Fig. 46: *Enroll an SMS token*

When enrolling an SMS token, you only need to specify the mobile phone number.

SMS token is a challenge response token. I.e. when sending the OTP PIN in the first authentication request, the sending of the SMS will be triggered and in a second authentication request the OTP value from the SMS needs to be presented. It implements the *challenge authentication mode*.

For a more detailed insight see the code documentation *SMS Token*.

Spass - Simple Pass Token

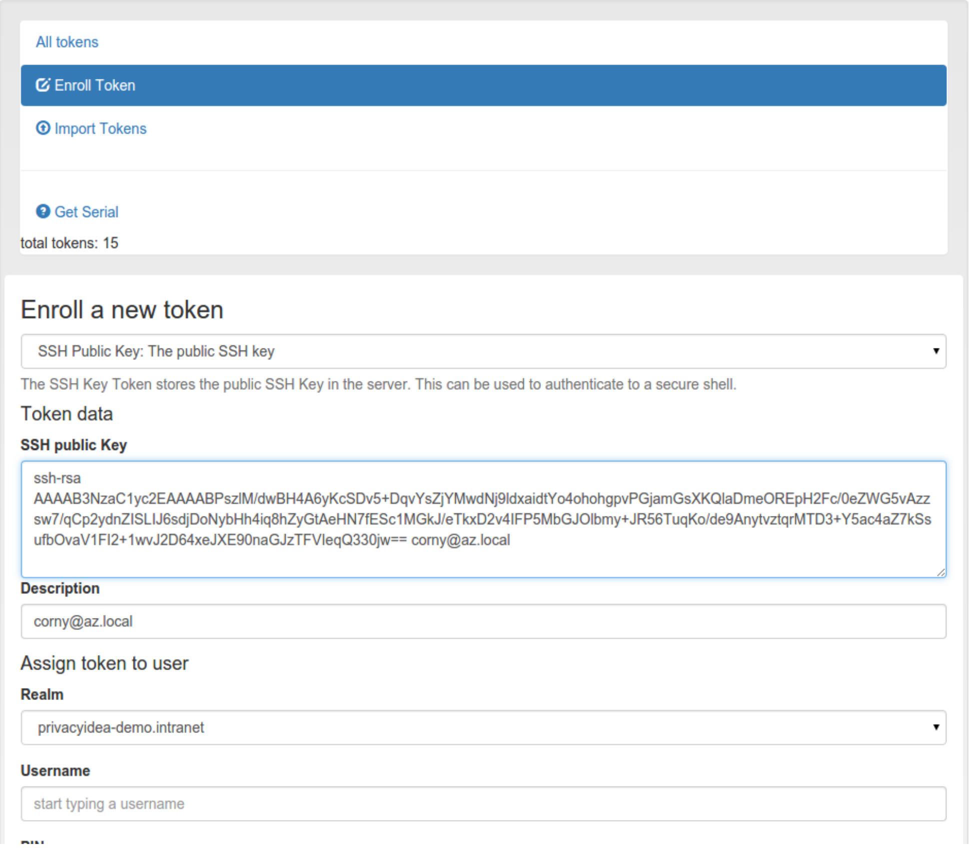
The OTP component of the *spass* token is always true. Thus the user only needs to provide the OTP pin or the userstore password - depending on the policy settings.

For a more detailed insight see the code documentation *SPass Token*.

SSH Keys

The token type *sshkey* is the public SSH key, that you can upload and assign to a user. The SSH key is only used for the application type **SSH** in conjunction with the *Machines* concept.

A user or the administrator can upload the public SSH key and assign to a user.



total tokens: 15

Enroll a new token

SSH Public Key: The public SSH key

The SSH Key Token stores the public SSH Key in the server. This can be used to authenticate to a secure shell.

Token data

SSH public Key

```
ssh-rsa
AAAAB3NzaC1yc2EAAAABPszlM/dwBH4A6yKcSDv5+DqvYsZjYMwdNj9ldxaidtYo4ohohgpvPGjamGsXKQlaDmeOREpH2Fc/0eZWG5vAzz
sw7/qCp2ydnZISLIJ6sdjDoNybHh4iq8hZyGtAeHN7fESc1MGkJ/eTkxD2v4IFP5MbGJOlbmy+JR56TuqKo/de9AnytvztqrMTD3+Y5ac4aZ7kSs
ufbOvaV1FI2+1wvJ2D64xeJXE90naGJzTFVleqQ330jw== corny@az.local
```

Description

corny@az.local

Assign token to user

Realm

privacyidea-demo.intranet

Username

start typing a username

Fig. 47: *Enroll an SSH key token*

Paste the SSH key into the text area. The comment in the SSH key will be used as token comment. You can assign the SSH key to a user and then use the SSH key in Application Definitions *SSH*.

Note: This way you can manage SSH keys centrally, as you do not need to distribute the SSH keys to all machines. You rather store the SSH keys centrally in privacyIDEA and use **privacyidea-authorizedkeys** to fetch the keys in real time during the login process.

TAN Token

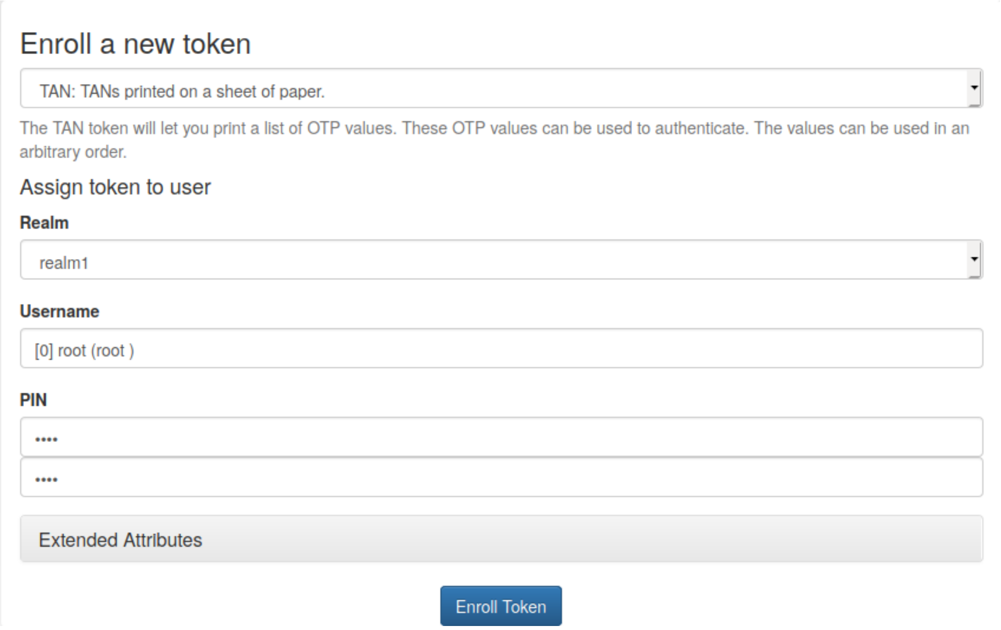
(added in version 2.23)

The token type *tan* is related to the *Paper Token (PPR)*.

In contrast to the *paper* token, a user can use the OTP values of a *tan* token in any arbitrary order.

A *tan* token can either be initialized with random OTP values. In this case the HOTP mechanism is used. Or it can be initialized or imported with a dedicated list of TANs.

After enrollment, you are prompted to print the generated TAN list.



Enroll a new token

TAN: TANs printed on a sheet of paper.

The TAN token will let you print a list of OTP values. These OTP values can be used to authenticate. The values can be used in an arbitrary order.

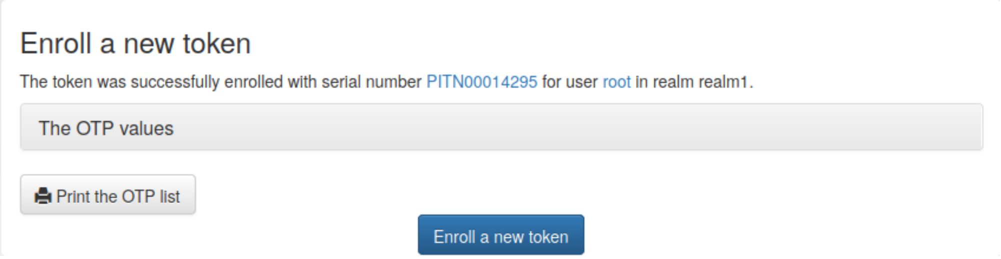
Assign token to user

Realm: realm1

Username: [0] root (root)

PIN: ••••

••••



Enroll a new token

The token was successfully enrolled with serial number PITN00014295 for user root in realm realm1.

Import of TAN token

The import schema for TAN tokens via the OATH CSV file look like this:

> <serial>, <seed>, tan, <white space separated list of tans>

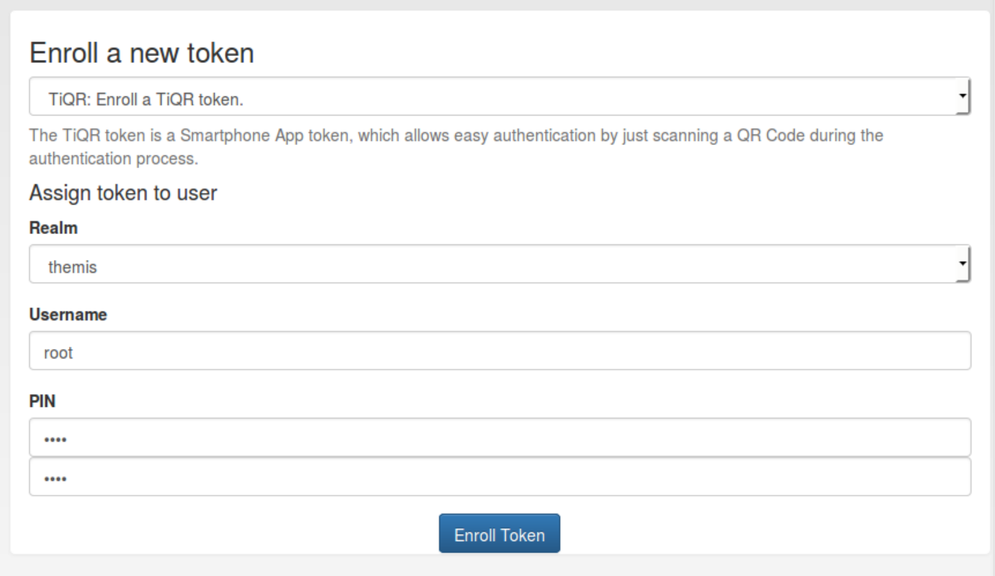
The TANs are located in the 4th column. TANs are separated by blanks or whitespaces. The *<seed>* is not used with a TAN token. You can leave this blank or set to any (not used) value.

TiQR

Starting with version 2.6 privacyIDEA supports the TiQR token. The TiQR token is a smartphone token, that can be used to login by only scanning a QR code.

The TiQR token implements the *outofband authentication mode*. The configuration is described in *TiQR Token Configuration*.

The token is also enrolled by scanning a QR code.



Enroll a new token

TiQR: Enroll a TiQR token.

The TiQR token is a Smartphone App token, which allows easy authentication by just scanning a QR Code during the authentication process.

Assign token to user

Realm: themis

Username: root

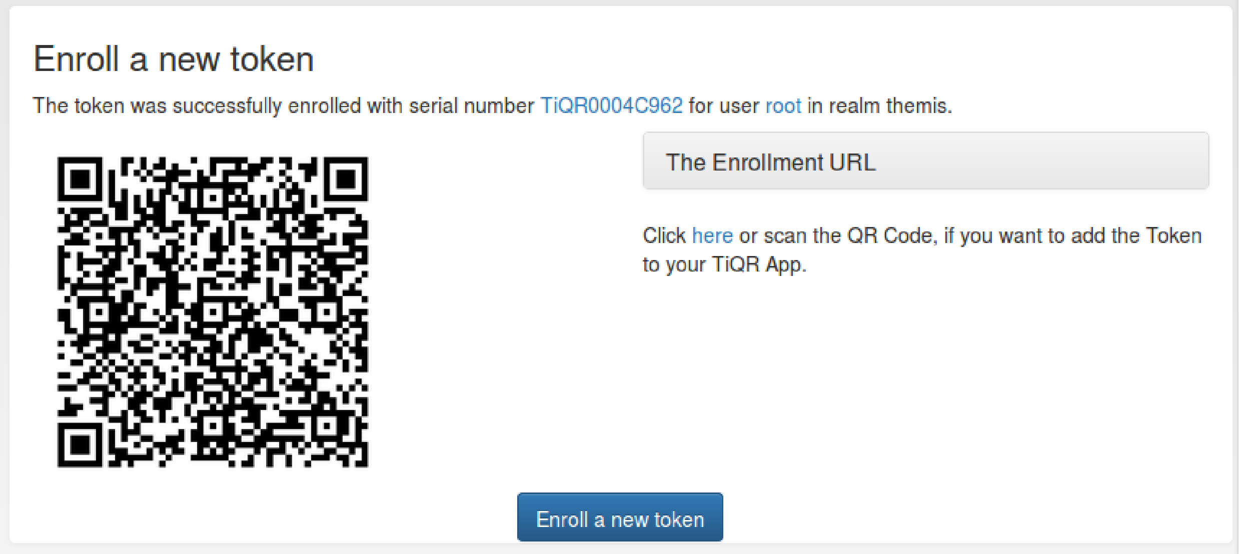
PIN: ••••

••••

Fig. 48: *Choose a user for the TiQR token*

You can only enroll a TiQR token, when a user is selected.

Note: You can not enroll a TiQR token without assign the token to a user.



Enroll a new token

The token was successfully enrolled with serial number TiQR0004C962 for user root in realm themis.

The Enrollment URL

Click here or scan the QR Code, if you want to add the Token to your TiQR App.

For more technical information about the TiQR token please see *TiQR Token*.

TOTP

The TOTP token is - together with the *HOTP Token* - the most common token. The TOTP Algorithm is defined in RFC6238. The TOTP token is a time based token. Roughly speaking the TOTP algorithm is the same algorithm like the HOTP, where the event based counter is replaced by the unix timestamp.

The TOTP algorithm has some parameter, like if the generated OTP value will be 6 digits or 8 digits or if the SHA1 oder the SHA256 hashing algorithm is used and the timestep being 30 or 60 seconds.

The TOTP token implements the *authenticate mode*. With a suitable *challenge_response* policy, it may also be used in the *challenge mode*.

Hardware tokens

The information about preseeded token and seedable tokens is the same as described in the section about *HOTP Token*.

The only available seedable pushbutton TOTP token is the *SafeNet eToken Pass*. The Yubikey can be used as a TOTP token, but only in conjunction with a smartphone app, since the yubikey has not its own clock.

Software tokens

Experiences

The Google Authenticator and the FreeOTP token can be enrolled easily in TOTP mode using the QR-Code enrollment Feature.

The Google Authenticator is available for iOS, Android and Blackberry devices.

Enrollment

Default settings for TOTP tokens can be configured at *TOTP Token Config*.

The enrollment is the same as described in *HOTP Token*. However, when enrolling TOTP token, you can specify some additional parameters.

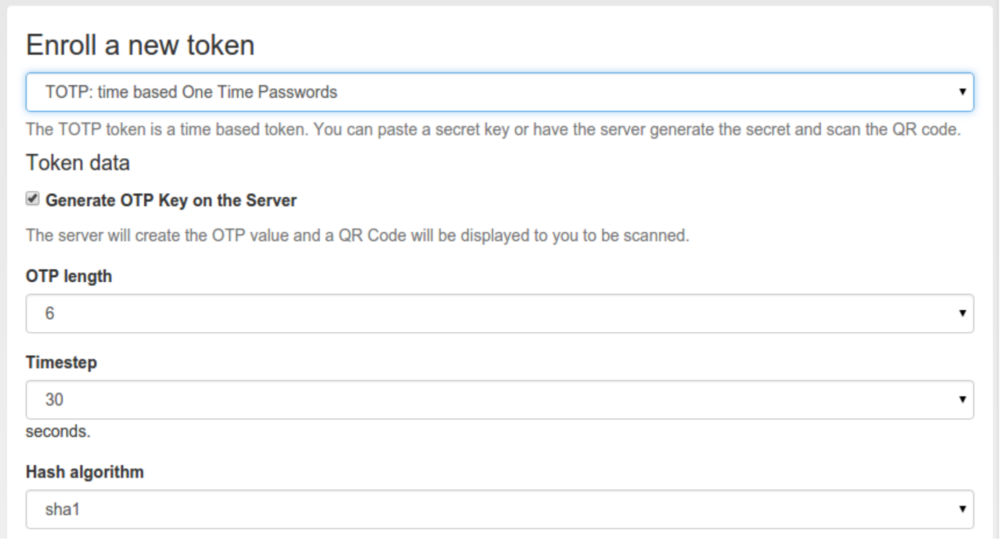
U2F

Starting with version 2.7 privacyIDEA supports U2F tokens. The administrator or the user himself can register a U2F device and use this U2F token to login to the privacyIDEA web UI or to authenticate at applications.

When enrolling the token a key pair is generated and the public key is sent to privacyIDEA. During this process the user needs to prove that he is present by either pressing the button (Yubikey) or by replugging the device (Plug-up token).

The device is identified and assigned to the user.

Note: This is a normal token object which can also be reassigned to another user.



Enroll a new token

TOTP: time based One Time Passwords

The TOTP token is a time based token. You can paste a secret key or have the server generate the secret and scan the QR code.

Token data

☑ **Generate OTP Key on the Server**

The server will create the OTP value and a QR Code will be displayed to you to be scanned.

OTP length

6

Timestep

30

seconds.

Hash algorithm

sha1

Fig. 49: *Enroll an TOTP token*

Note: As the key pair is only generated virtually, you can register one physical device for several users.

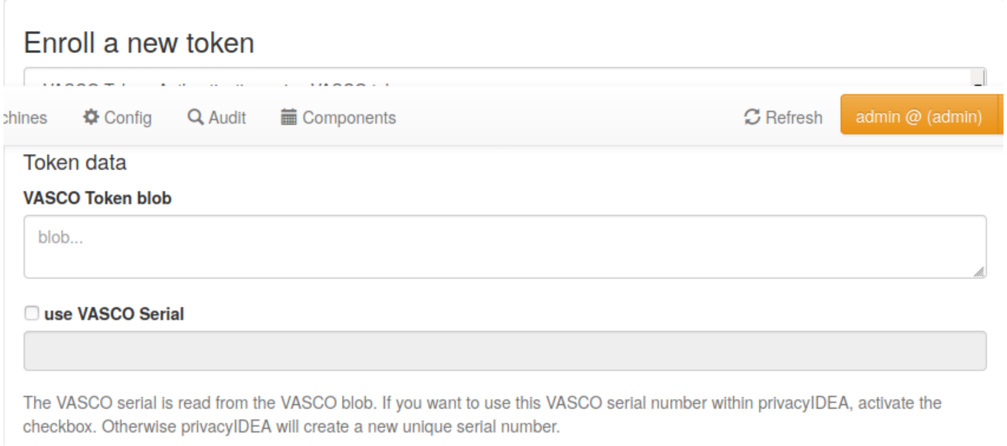
For configuring privacyIDEA for the use of U2F token, please see *U2F*.

For further details and for information how to add this to your application you can see the code documentation at *U2F Token*.

VASCO

Starting with version 2.22 privacyIDEA supports VASCO tokens.

VASCO OTP tokens are a proprietary OTP token. You can import the VASCO blobs from a CSV file or you the administrator can enroll a single VASCO token.



Enroll a new token

Token data

VASCO Token blob

blob...

□ **use VASCO Serial**

The VASCO serial is read from the VASCO blob. If you want to use this VASCO serial number within privacyIDEA, activate the checkbox. Otherwise privacyIDEA will create a new unique serial number.

Note: privacyIDEA uses a proprietary VASCO library vacman to verify the OTP values. Please note that you need to license this library from *VASCO Data Security N.V.* directly. The privacyIDEA project does not provide this library.

WebAuthn

Starting with version 3.4 privacyIDEA supports WebAuthn tokens. The administrator or the user himself can register a WebAuthn device and use this WebAuthn token to login to the privacyIDEA WebUI or to authenticate against applications.

When enrolling the token, a key pair is generated and the public key is sent to privacyIDEA. During this process, the user needs to prove that he is present, which typically happens by tapping a button on the token. The user may also be required by policy to provide some form of verification, which might be biometric or knowledge-based, depending on the token.

The devices is identified and assigned to the user.

Note: This is a normal token object which can also be reassigned to another user.

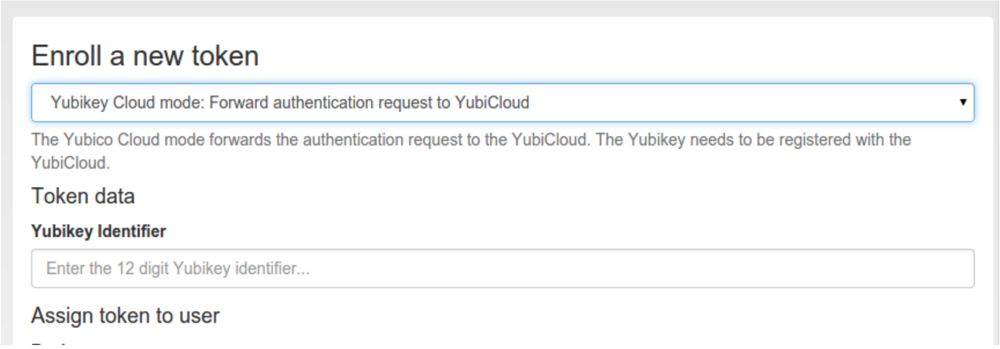
Note: As the key pair is only generated virtually, you can register one physical device for several users.

For configuring privacyIDEA for the use of WebAuthn tokens, please see *WebAuthn Token Config*.

For further details and information how to add this to your application, see the code documentation at *WebAuthn Token*.

Yubico

The token type *yubico* authenticates against the Yubico Cloud mode. You need to configure this at *Yubico Cloud mode*.



Enroll a new token

Yubikey Cloud mode: Forward authentication request to YubiCloud

The Yubico Cloud mode forwards the authentication request to the YubiCloud. The Yubikey needs to be registered with the YubiCloud.

Token data

Yubikey Identifier

Enter the 12 digit Yubikey identifier...

Assign token to user

Fig. 50: *Enroll a Yubico token*

The token is enrolled by simply saving the Yubikey token ID in the token object. You can either enter the 12 digit ID or you can simply press the Yubikey button in the input field, which will also assign the token.

Yubikey

The Yubikey is initialized with privacyIDEA and works in Yubicos own AES mode. It outputs a 44 character OTP value, consisting of a 12 character prefix and a 32 character OTP. But in contrast to the *Yubico* Cloud mode, in this mode the secret key is contained within the token and your own privacyIDEA installation.

If you have the time and care about privacy, you should prefer the Yubikey AES mode over the *Yubico* Cloud mode.

There are three possible ways to enroll a Yubikey token.

Note: We recommend that you use the `privacyidea` command line client, to initialize the Yubikeys. You can use the mass enrollment, which eases the process of initializing a whole bunch of tokens.

Run the command like this:

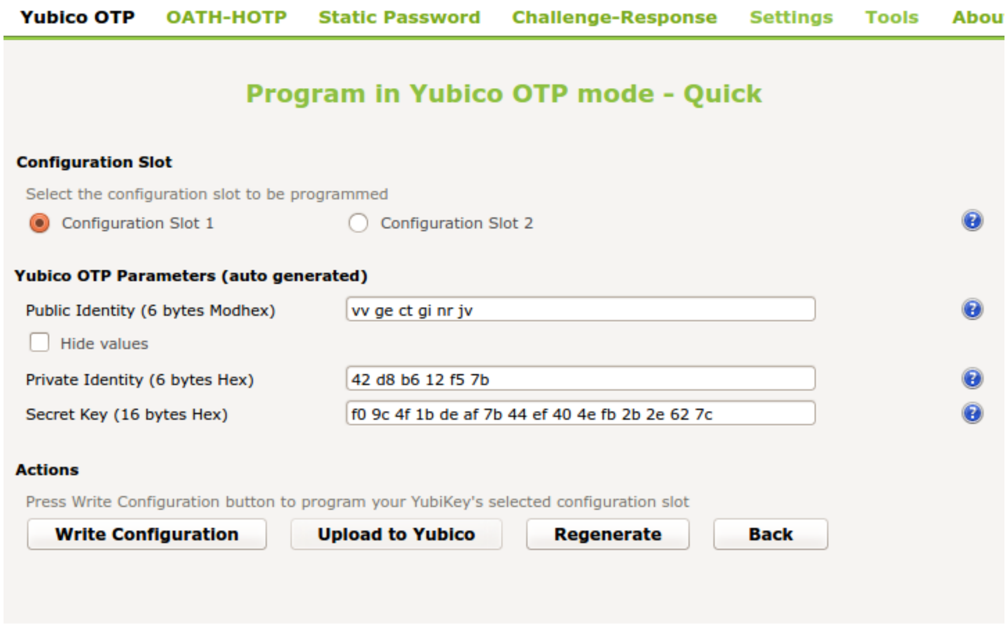
```
privacyidea -U https://your.privacyidea.server -a admin token \
yubikey_mass_enroll --yubimode YUBICO
```

This command initializes the token and stores the AES secret and prefix in privacyidea, so the token is immediately useful. You can choose the slot with `--yubislot`. For further help call `privcyidea yubikey_mass_enroll` with the `--help` option.

The second way to enroll a yubikey token is also using `yubikey_mass_enroll`, but with the option `--filename` to write to token configuration into the specified file. The resulting file can then be imported into privacyidea: Select Tokens -> Import Tokens, select "OATH CSV" and the file you just created.

Using the yubikey personalization GUI

Third and last you can use the privacyIDEA Web UI to enroll a Yubikey AES mode token, if you have initialized the yubikey with the external *ykpersonalize* tool.



Program in Yubico OTP mode - Quick

Configuration Slot

Select the configuration slot to be programmed

● Configuration Slot 1 ○ Configuration Slot 2

Yubico OTP Parameters (auto generated)

Public Identity (6 bytes Modhex): vv ge ct gi nr jv

□ Hide values

Private Identity (6 bytes Hex): 42 d8 b6 12 f5 7b

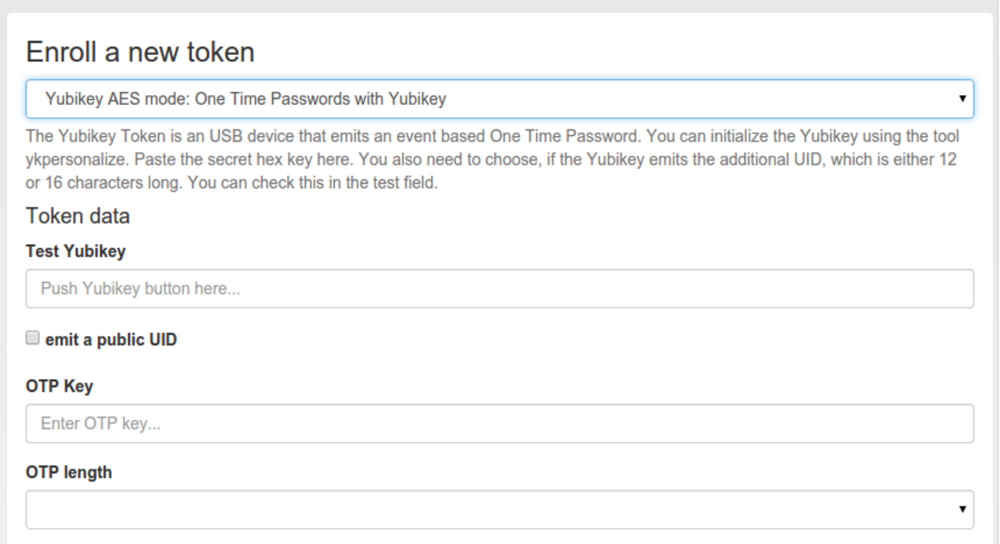
Secret Key (16 bytes Hex): f0 9c 4f 1b de af 7b 44 ef 40 4e fb 2b 2e 62 7c

Actions

Press Write Configuration button to program your YubiKey's selected configuration slot

Fig. 51: *Use the yubikey-personalization-gui to initialize the yubikey*

When using the yubikey personalization GUI you need to copy the value of "Secret Key (16 bytes Hex)". This is the secret OTP key, which you need to copy and paste in the field "OTP Key" in the privacyIDEA Web UI.



Enroll a new token

Yubikey AES mode: One Time Passwords with Yubikey ▾

The Yubikey Token is an USB device that emits an event based One Time Password. You can initialize the Yubikey using the tool ykpersonalize. Paste the secret hex key here. You also need to choose, if the Yubikey emits the additional UID, which is either 12 or 16 characters long. You can check this in the test field.

Token data

Test Yubikey

Push Yubikey button here...

□ **emit a public UID**

OTP Key

Enter OTP key...

OTP length

▾

Fig. 52: *Enroll a Yubikey AES mode token*

In the field "Test Yubikey" push the Yubikey button. This will grab the yubikey's public identifier and also determine the length of the *otp value*. The field *OTP value* is automatically filled.

Redirect api url to privacyideas /ttype/yubikey

Yubico servers use `/wsapi/2.0/verify` as the path in the validation URL. Some tools (e.g. Kolab 2fa) let the user/admin change the api host, but not the rest of the URL. Let's redirect the api URL to privacyideas /ttype/yubikey - you'll need to enable the following two lines in /etc/apache2/site-enabled/privacyidea.conf:

> RewriteEngine on RewriteRule "^/wsapi/2.0/verify" "/ttype/yubikey" [PT]

If you use nginx there is a similar line provided as a comment to the nginx configuration as well.

1.7 Policies

Policies can be used to define the reaction and behaviour of the system.

Each policy defines the behaviour in a certain area, called scope. privacyIDEA knows the scopes:

1.7.1 Admin policies

Admin policies are used to regulate the actions that administrators are allowed to do. Technically admin policies control the use of the REST API *Token endpoints*, *System endpoints*, *Realm endpoints* and *Resolver endpoints*.

Admin policies are implemented as decorators in *Policy Module* and *Policy Decorators*.

The `user` in the admin policies refers to the name of the administrator.

Starting with privacyIDEA 2.4 admin policies can also store a field "admin realm". This is used, if you define realms to be superuser realms. See *The Config File* for information how to do this. Read *So what's the thing with all the admins?* for more information on the admin realms.

This way it is easy to define administrative rights for big groups of administrative users like help desk users in the IT department.

All administrative actions also refer to the defined user realm. Meaning an administrator may have many rights in one user realm and only a few rights in another realm.

Creating a policy with `scope:admin`, `admin-realm:helpdesk`, `user:frank`, `action:enable` and `realm:sales` means that the administrator *frank* in the admin-realm *helpdesk* is allowed to enable tokens in the user-realm *sales*.

Note: As long as no admin policy is defined all administrators are allowed to do everything.

The following actions are available in the scope *admin*:

tokenlist

type: bool

This allows the administrator to list existing tokens in the specified user realm. Note, that the resolver in this policy is ignored.

If the policy with the action `tokenlist` is not bound to any user realm, this acts as a wild card and the admin is allowed to list all tokens.

If the action `tokenlist` is not active, but admin policies exist, then the admin is not allowed to list any tokens.

Note: As with all boolean policies, multiple *tokenlist* policies add up to create the resulting rights of the administrator. So if there are multiple matching policies for different realms, the admin will have list rights on all mentioned realms independent on the priority of the policies.

init

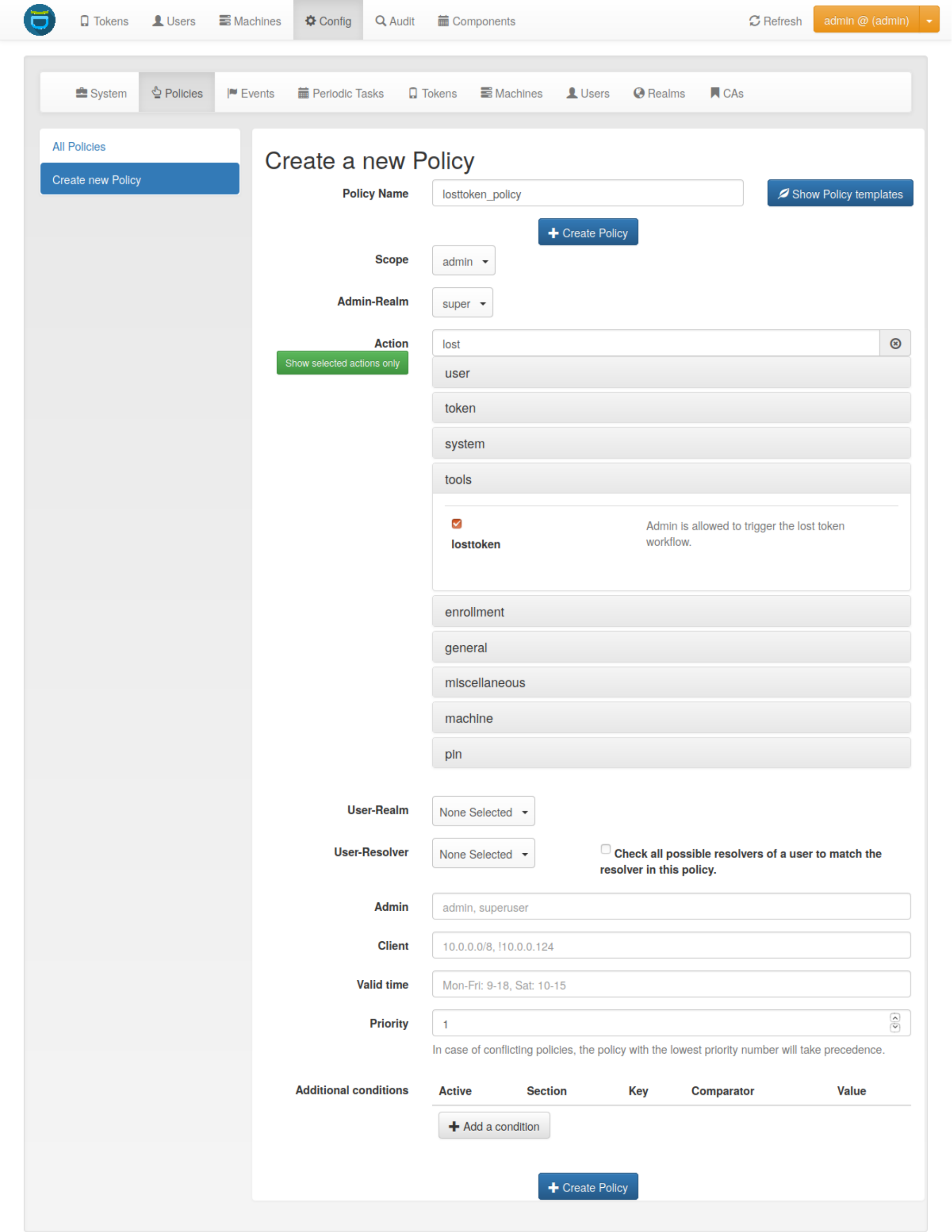
type: bool

There are `init` actions per token type. Thus you can create policy that allow an administrator to enroll SMS tokens but not to enroll HMAC tokens.

enable

type: bool

The `enable` action allows the administrator to activate disabled tokens.



Create a new Policy

Policy Name: losttoken_policy

Scope: admin

Admin-Realm: super

Action: lost

☑ **losttoken** Admin is allowed to trigger the lost token workflow.

User-Realm: None Selected

User-Resolver: None Selected

□ **Check all possible resolvers of a user to match the resolver in this policy.**

Admin: admin, superuser

Client: 10.0.0.0/8, !10.0.0.124

Valid time: Mon-Fri: 9-18, Sat: 10-15

Priority: 1

In case of conflicting policies, the policy with the lowest priority number will take precedence.

Additional conditions

Active	Section	Key	Comparator	Value

Fig. 53: *The Admin scope provides an additional field 'admin realm'.*

disable

type: bool

Tokens can be enabled and disabled. Disabled tokens can not be used to authenticate. The `disable` action allows the administrator to disable tokens.

revoke

type: bool

Tokens can be revoked. Usually this means the token is disabled and locked. A locked token can not be modified anymore. It can only be deleted.

Certain token types like *certificate* may define special actions when revoking a token.

set

type: bool

Tokens can have additional token information, which can be viewed in the token_details.

If the `set` action is defined, the administrator allowed to set those token information.

setpin

type: bool

If the `setpin` action is defined, the administrator is allowed to set the OTP PIN of a token.

setrandompin

type: bool

If the `setrandompin` action is defined, the administrator is allowed to call the endpoint, that sets a random token PIN.

enrollpin

type: bool

If the action `enrollpin` is defined, the administrator can set a token PIN during enrollment. If the action is not defined and the administrator tries to set a PIN during enrollment, this PIN is deleted from the request.

otp_pin_maxlength

type: integer

range: 0 - 31

This is the maximum allowed PIN length the admin is allowed to use when setting the OTP PIN.

Note: There can be token type specific policies like `spass_otp_pin_maxlength`, `spass_otp_pin_minlength` and `spass_otp_pin_contents`. If suche a token specific policy exists, it takes priority of the common PIN policy.

otp_pin_minlength

type: integer

range: 0 - 31

This is the minimum required PIN the admin must use when setting the OTP PIN.

otp_pin_contents

type: string

contents: cns

This defines what characters an OTP PIN should contain when the admin sets it.

c are letters matching [a-zA-Z].

n are digits matching [0-9].

s are special characters matching [[].:,;-_<>+*!/()=?$§%&#~^].

[allowedchars] is a specific list of allowed characters.

Example: The policy action `otp_pin_contents=cn, otp_pin_minlength=8` would require the admin to choose OTP PINs that consist of letters and digits which have a minimum length of 8.

`cn`

> *test1234* and *test12$$* would be valid OTP PINs. *testABCD* would not be a valid OTP PIN.

The logic of the `otp_pin_contents` can be enhanced and reversed using the characters `+` and `-`.

`-cn` (denial)

> The PIN must not contain a character and must not contain a number. *test1234* would not be a valid PIN, since it does contains numbers and characters. *test///* would not be a valid PIN, since it contains characters.

`-s` (denial)

> The PIN must not contain a special character. **test1234* would be a valid PIN. *test12$$* would not.

`+cn` (grouping)

> combines the two required groups. I.e. the OTP PIN should contain characters from the sum of the two groups. *test1234*, *test12$$*, *test* and *1234* would all be valid OTP PINs. Note, how this is different to `-s`, since it allows special characters to be included.

`[123456]`

> allows the digtits 1-6 to be used. *1122* would be a valid PIN. *1177* would not be a valid PIN.

otp_pin_set_random

type: integer

range: 1-31

The administrator can set a random pin for a token with the endpoint `token/setrandompin`. This policy is needed to define how long the PIN will be.

Note: The PIN will consist of digits and letters.

resync

type: bool

If the `resync` action is defined, the administrator is allowed to resynchronize a token.

assign

type: bool

If the `assign` action is defined, the administrator is allowed to assign a token to a user. This is used for assigning an existing token to a user but also to enroll a new token to a user.

Without this action, the administrator can not create a connection (assignment) between a user and a token.

unassign

type: bool

If the `unassign` action is defined, the administrator is allowed to unassign tokens from a user. I.e. the administrator can remove the link between the token and the user. The token still continues to exist in the system.

import

type: bool

If the `import` action is defined, the administrator is allowed to import token seeds from a token file, thus creating many new token objects in the systems database.

The right to upload tokens can be limited to certain realms. Thus the administrator could only upload tokens into realm he is allowed to manage.

remove

type: bool

If the `remove` action is defined, the administrator is allowed to delete a token from the system.

Note: If a token is removed, it can not be recovered.

Note: All audit entries of this token still exist in the audit log.

userlist

type: bool

If the `userlist` action is defined, the administrator is allowed to view the user list in a realm. An administrator might not be allowed to list the users, if he should only work with tokens, but not see all users at once.

Note: If an administrator has any right in a realm, the administrator is also allowed to view the token list.

checkstatus

type: bool

If the `checkstatus` action is defined, the administrator is allowed to check the status of open challenge requests.

manageToken

type: bool

If the `manageToken` action is defined, the administrator is allowed to manage the realms of a token.

A token may be located in multiple realms. This can be interesting if you have a pool of spare tokens and several realms but want to make the spare tokens available to several realm administrators. (Administrators, who have only rights in one realm)

Then all administrators can see these tokens and assign the tokens. But as soon as the token is assigned to a user in one realm, the administrator of another realm can not manage the token anymore.

getserial

type: bool

If the `getserial` action is defined, the administrator is allowed to calculate the token serial number for a given OTP value.

getrandom

type: bool

The `getrandom` action allows the administrator to retrieve random keys from the endpoint *getrandom*. This is an endpoint in *System endpoints*.

getrandom can be used by the client, if the client has no reliable random number generator. Creating API keys for the Yubico Validation Protocol uses this endpoint.

getchallenges

type: bool

This policy allows the administrator to retrieve a list of active challenges of a challenge response tokens. The administrator can view these challenges in the web UI.

losttoken

type: bool

If the `losttoken` action is defined, the administrator is allowed to perform the lost token process.

To only perform the lost token process the actions `copytokenuser` and `copytokenpin` are not necessary!

adduser

type: bool

If the `adduser` action is defined, the administrator is allowed to add users to a user store.

Note: The user store still must be defined as editable, otherwise no users can be added, edited or deleted.

updateuser

type: bool

If the `updateuser` action is defined, the administrator is allowed to edit users in the user store.

deleteuser

type: bool

If the `deleteuser` action is defined, the administrator is allowed to delete an existing user from the user store.

copytokenuser

type: bool

If the `copytokenuser` action is defined, the administrator is allowed to copy the user assignment of one token to another.

This functionality is also used during the lost token process. But you only need to define this action, if the administrator should be able to perform this task manually.

copytokenpin

type: bool

If the `copytokenpin` action is defined, the administrator is allowed to copy the OTP PIN from one token to another without knowing the PIN.

This functionality is also used during the lost token process. But you only need to define this action, if the administrator should be able to perform this task manually.

smtpserver_write

type: bool

To be able to define new *SMTP server configuration* or delete existing ones, the administrator needs this rights `smtpserver_write`.

smtpserver_read

type: bool

Allow the administrator to read the *SMTP server configuration*.

smsgateway_write

type: bool

To be able to define new *SMS Gateway configuration* or delete existing ones, the administrator needs the right `smsgateway_write`.

smsgateway_read

type: bool

Allow the administrator to read the *SMS Gateway configuration*.

periodictask_write

type: bool

Allow the administrator to write or delete *Periodic Tasks* definitions.

periodictask_read

type: bool

Allow the administrator to read the *Periodic Tasks* definitions.

eventhandling_write

type: bool

Allow the administrator to configure *Event Handler*.

eventhandling_read

type: bool

Allow the administrator to read *Event Handler*.

Note: Currently the policies do not take into account resolvers, or realms. Having the right to read event handlers, will allow the administrator to see all event handler definitions.

policywrite, policyread, policydelete

type: bool

Allow the administrator to write, read or delete policies.

Note: Currently the policies do not take into account resolvers, or realms. Having the right to read policies, will allow the administrator to see all policies.

resolverwrite, resolverread, resolverdelete

type: bool

Allow the administrator to write, read or delete user resolvers and realms.

Note: Currently the policies do not take into account resolvers, or realms. Having the right to read resolvers, will allow the administrator to see all resolvers and realms.

mresolverwrite, mresolverread, mresolverdelete

type: bool

Allow the administrator to write, read or delete machine resolvers.

configwrite, configread, configdelete

type: bool

Allow the administrator to write, read or delete system configuration.

auditlog

type: bool

The administrators are allowed to view the audit log. If the policy contains a user realm, than the administrator is only allowed to see entries which contain this very user realm. A list of user realms may be defined.

To learn more about the audit log, see *Audit*.

auditlog_download

type: bool

The administrator is allowed to download the audit log.

Note: The download is not restricted to filters and audit age. Thus, if you want to avoid, that an administrator can see older logs, you need to disallow downloading the data. Otherwise he may download the audit log and look at older entries manually.

auditlog_age

type: string

This limits the maximum age of displayed audit entries. Older entries are not remove from the audit table but the administrator is simply not allowed to view older entries.

Can be something like 10m (10 minutes), 10h (10 hours) or 10d (ten days).

trigger_challenge

type: bool

If set the administrator is allowed to call the API `/validate/triggerchallenge`. This API can be used to send an OTP SMS to user without having specified the PIN of the SMS token.

The usual setup that one administrative account has only this single policy and is only used for triggering challenges.

New in version 2.17.

hotp_2step and totp_2step

type: string

This allows or forces the administrator to enroll a smartphone based token in two steps. In the second step the smartphone generates a part of the OTP secret, which the administrator needs to enter. (see *Two Step Enrollment*). Possible values are *allow* and *force*. This works in conjunction with the enrollment parameters *{type}_2step_clientsize, {type}_2step_serversize, {type}_2step_difficulty*.

Such a policy can also be set for the user. See *hotp_2step and totp_2step*.

New in version 2.21

hotp_hashlib and totp_hashlib

type: string

Force the admin to enroll HOTP/TOTP Tokens with the specified hashlib. The corresponding input selector will be disabled in the web UI. Possible values are *sha1*, *sha256* and *sha512*, default is *sha1*.

New in 3.2

hotp_otplen and totp_otplen

type: int

Force the admin to enroll HOTP/TOTP Tokens with the specified otp length. The corresponding input selector will be disabled in the web UI. Possible values are *6* or *8*, default is *6*.

New in 3.2

totp_timestep

type: int

Enforce the timestep of the time-based OTP token. A corresponding input selection will be disabled/hidden in the web UI. Possible values are *30* or *60*, default is *30*.

New in 3.2

system_documentation

type: bool

The administrator is allowed to export a complete system documentation including resolvers and realm. The documentation is created as restructured text.

sms_gateways

type: string

Usually an SMS token sends the SMS via the SMS gateway that is system wide defined in the token settings. This policy takes a blank-separated list of configured SMS gateways. It allows the administrator to define an individual SMS gateway during token enrollment.

New in version 3.0.

indexedsecret_force_attribute

type: string

If an administrator enrolls an indexedsecret token then the value of the given user attribute is set as the secret. The admin does not know the secret and can not change the secret.

For more details of this token type see *Indexed Secret Token*.

New in version 3.3.

1.7.2 User Policies

In the Web UI users can manage their own tokens. User can login to the Web UI with the username of their useridresolver. I.e. if a user is found in an LDAP resolver pointing to Active Directory the user needs to login with his domain password.

User policies are used to define, which actions users are allowed to perform.

The user policies also respect the `client` input, where you can enter a list of IP addresses and subnets (like 10.2.0.0/16).

Using the `client` parameter you can allow different actions in if the user either logs in from the internal network or remotely from the internet via the firewall.

Technically user policies control the use of the REST API *Token endpoints* and are checked using *Policy Module* and *Policy Decorators*.

Note: If no user policy is defined, the user has all actions available to him, to manage his tokens.

The following actions are available in the scope *user*:

enroll

type: bool

There are `enroll` actions per token type. Thus you can create policies that allow the user to enroll SMS tokens but not to enroll HMAC tokens.

assgin

type: bool

The user is allowed to assgin an existing token, that is located in his realm and that does not belong to any other user, by entering the serial number.

disable

type: bool

The user is allowed to disable his own tokens. Disabled tokens can not be used to authenticate.

enable

type: bool

The user is allowed to enable his own tokens.

delete

type: bool

The user is allowed to delete his own tokens from the database. Those tokens can not be recovered. Anyway, the audit log concerning these tokens remains.

unassign

type: bool

The user is allowed to drop his ownership of the token. The token does not belong to any user anymore and can be reassigned.

resync

type: bool

The user is allowed to resynchronize the token if it has got out of synchronization.

reset

type: bool

The user is allowed to reset the failcounter of the token.

setpin

type: bool

The user is allowed to set the OTP PIN for his tokens.

setrandompin

type: bool

If the `setrandompin` action is defined, the user is allowed to call the endpoint, that sets a random PIN on his specified token.

setdescription

type: bool

The user is allowed to set the description of his tokens.

enrollpin

type: bool

If the action `enrollpin` is defined, the user can set a token PIN during enrollment. If the action is not defined and the user tries to set a PIN during enrollment, this PIN is deleted from the request.

otp_pin_maxlength

type: integer

range: 0 - 31

This is the maximum allowed PIN length the user is allowed to use when setting the OTP PIN.

Note: There can be token type specific policies like `spass_otp_pin_maxlength`, `spass_otp_pin_minlength` and `spass_otp_pin_contents`. If suche a token specific policy exists, it takes priority of the common PIN policy.

otp_pin_minlength

type: integer

range: 0 - 31

This is the minimum required PIN the user must use when setting the OTP PIN.

otp_pin_contents

type: string

contents: cns

This defines what characters an OTP PIN should contain when the user sets it.

This takes the same values like the admin policy *otp_pin_contents*.

auditlog

type: bool

This action allows the user to view and search the audit log for actions with his own tokens.

To learn more about the audit log, see *Audit*.

auditlog_age

type: string

This limits the maximum age of displayed audit entries. Older entries are not remove from the audit table but the user is simply not allowed to view older entries.

Can be something like 10m (10 minutes), 10h (10 hours) or 10d (ten days).

updateuser

type: bool

If the `updateuser` action is defined, the user is allowed to change his attributes in the user store.

Note: To be able to edit the attributes, the resolver must be defined as editable.

revoke

type: bool

Tokens can be revoked. Usually this means the token is disabled and locked. A locked token can not be modified anymore. It can only be deleted.

Certain token types like *certificate* may define special actions when revoking a token.

password_reset

type: bool

Introduced in version 2.10.

If the user is located in an editable user store, this policy can define, if the user is allowed to perform a password reset. During the password reset an email with a link to reset the password is sent to the user.

hotp_2step and totp_2step

type: string

This allows or forces the user to enroll a smartphone based token in two steps. In the second step the smartphone generates a part of the OTP secret, which the user needs to enter. (see *Two Step Enrollment*). Possible values are *allow* and *force*. This works in conjunction with the enrollment parameters *{type}_2step_clientsize, {type}_2step_serversize, {type}_2step_difficulty*.

Such a policy can also be set for the administrator. See *hotp_2step and totp_2step*.

New in version 2.21

sms_gateways

type: string

Usually an SMS tokens sends the SMS via the SMS gateway that is system wide defined in the token settings. This policy takes a blank separated list of configured SMS gateways. It allows the user to define an individual SMS gateway during token enrollment.

New in version 3.0.

hotp_hashlib and totp_hashlib

type: string

Force the user to enroll HOTP/TOTP Tokens with the specified hashlib. The corresponding input selector will be disabled/hidden in the web UI. Possible values are *sha1*, *sha256* and *sha512*, default is *sha1*.

hotp_otplen and totp_otplen

type: int

Force the user to enroll HOTP/TOTP Tokens with the specified otp length. The corresponding input selector will be disabled/hidden in the web UI. Possible values are *6* or *8*, default is *6*.

hotp_force_server_generate and totp_force_server_generate

type: bool

Enforce the key generation on the server. A corresponding input field for the key data will be disabled/hidden in the web UI. Default value is *false*.

totp_timestep

type: int

Enforce the timestep of the time-based OTP token. A corresponding input selection will be disabled/hidden in the web UI. Possible values are *30* or *60*, default is *30*.

indexedsecret_force_attribute

type: string

If a user enrolls an indexedsecret token then the value of the given user attribute is set as the secret. The user does not see the value and can not change the value.

For more details of this token type see *Indexed Secret Token*.

New in version 3.3.

1.7.3 Authentication policies

The scope *authentication* gives you more detailed possibilities to authenticate the user or to define what happens during authentication.

Technically the authentication policies apply to the REST API *Validate endpoints* and are checked using *Policy Module* and *Policy Decorators*.

The following actions are available in the scope *authentication*:

otppin

type: string

This action defines how the fixed password part during authentication should be validated. Each token has its own OTP PIN, but you can choose how the authentication should be processed:

`otppin=tokenpin`

> This is the default behaviour. The user needs to pass the OTP PIN concatenated with the OTP value.

`otppin=userstore`

> The user needs to pass the user store password concatenated with the OTP value. It does not matter if the OTP PIN is set or not. If the user is located in an Active Directory the user needs to pass his domain password together with the OTP value.

Note: The domain password is checked with an LDAP bind right at the moment of authentication. So if the user is locked or the password was changed authentication will fail.

`otppin=none`

> The user does not have to pass any fixed password. Authentication is only done via the OTP value.

passthru

type: str

If the user has no token assigned, he will be authenticated against the userstore or against the given RADIUS configuration. I.e. the user needs to provide the LDAP- or SQL-password or valid credentials for the RADIUS server.

Note: This is a good way to do a smooth enrollment. Users having a token enrolled will have to use the token, users not having a token, yet, will be able to authenticate with their domain password.

It is also a way to do smooth migrations from other OTP systems. The authentication request of users without a token is forwarded to the specified RADIUS server.

Note: The passthru policy overrides the authorization policy for *tokentype*. I.e. a user may authenticate due to the passthru policy (since he has no token) although a tokentype policy is active!

Warning: If the user has the right to delete his tokens in selfservice portal, the user could delete all his tokens and then authenticate with his static password again.

passthru_assign

type: str

This policy is only evaluated, if the policy `passthru` is set. If the user is authenticated against a RADIUS server, then privacyIDEA splits the sent password into PIN and OTP value and tries to find an unassigned token, that is in the user's realm by using the OTP value. If it can identify this token, it assigns this token to the user and sets the sent PIN.

The policy is configured with a string value, that contains * the position of the PIN * the OTP length and * the number of OTP values tested for each unassigned token (optional, default=100).

Examples are

- `8:pin` would be an eight digit OTP value followed by the PIN
- `pin:6:10000` would be the PIN followed by an 6 digit OTP value, 10.000 otp values would be checked for each token.

Note: This method can be used to automatically migrated tokens from an old system to privacyIDEA. The administrator needs to import all seeds of the old tokens and put the tokens in the user's realm.

Warning: This can be very time consuming if the OTP values to check is set to high!

passOnNoToken

type: bool

If the user has no token assigned an authentication request for this user will always be true.

Warning: Only use this if you know exactly what you are doing.

passOnNoUser

type: bool

If the user does not exist, the authentication request is successful.

Warning: Only use this if you know exactly what you are doing.

smstext

type: string

This is the text that is sent via SMS to the user trying to authenticate with an SMS token. You can use the tags *<otp>* and *<serial>*.

Starting with version 2.20 you can use the tag *{challenge}*. This will add the challenge data that was passed in the first authentication request in the challenge parameter. This could contain banking transaction data.

Default: *<otp>*

smsautosend

type: bool

A new OTP value will be sent via SMS if the user authenticated successfully with his SMS token. Thus the user does not have to trigger a new SMS when he wants to login again.

emailtext

type: string

This is the text that is sent via Email to be used with Email Token. This text should contain the OTP value.

The text can contain the following tags, that will be filled:

- {serial} the serial number of the token.
- {user} the given name of the token owner.
- {givenname} the given name of the token owner.
- {surname} the surname of the token owner.
- {username} the loginname of the token owner.
- {userrealm} the realm of the token owner.
- {tokentype} the type of the token.
- {recipient_givenname} the given name of the recipient.
- {recipient_surname} the surname of the recipient.
- {time} the current server time in the format HH:MM:SS.
- {date} the current server date in the format YYYY-MM-DD

Starting with version 2.20 you can use the tag *{challenge}*. This will add the challenge data that was passed in the first authentication request in the challenge parameter. This could contain banking transaction data.

Default: <*otp*>

You can also provide the filename to an email template. The filename must be prefixed with `file:` like `file:/etc/privacyidea/emailtemplate.html`. The template is an HTML file.

Note: If a message text is supplied directly, the email is sent as plain text. If the email template is read from a file, a HTML-only email is sent instead.

emailsubject

type: string

This is the subject of the Email sent by the Email Token. You can use the same tags as mentioned in `emailtext`.

Default: Your OTP

emailautosend

type: bool

If set, a new OTP Email will be sent, when successfully authenticated with an Email Token.

mangle

type: string

The `mangle` policy can mangle the authentication request data before they are processed. I.e. the parameters `user`, `pass` and `realm` can be modified prior to authentication.

This is useful if either information needs to be stripped or added to such a parameter. To accomplish that, the mangle policy can do a regular expression search and replace using the keyword *user*, *pass* (password) and *realm*.

A valid action could look like this:

```
action: mangle=user/.*(.{4})/user\\1/
```

This would modify a username like "userwithalongname" to "username", since it would use the last four characters of the given username ("name") and prepend the fixed string "user".

This way you can add, remove or modify the contents of the three parameters. For more information on the regular expressions see[1].

Note: You must escape the backslash as \ to refer to the found substrings.

Example: A policy to remove whitespace characters from the realm name would look like this:

```
action: mangle=realm/\\s//
```

Example: If you want to authenticate the user only by the OTP value, no matter what OTP PIN he enters, a policy might look like this:

[1] https://docs.python.org/2/library/re.html

```
action: mangle=pass/.*(.{6})/\\1/
```

Example: If you want to strip a string from the front of a username, for example to have "admin_username" resolve to just "username", it would look like this:

```
action: mangle=user/admin_(.*)/\\1/
```

challenge_response

type: string

This is a list of token types for which challenge response can be used during authentication. The list is separated by whitespaces like *"hotp totp"*.

u2f_facets

type: string

This is a white space separated list of domain names, that are trusted to also use a U2F device that was registered with privacyIDEA.

You need to specify a list of FQDNs without the https scheme like:

"host1.example.com host2.example.com firewall.example.com"

For more information on configuring U2F see *U2F*.

reset_all_user_tokens

type: bool

If a user authenticates successfully all failcounter of all of his tokens will be reset. This can be important, if using empty PINs or *otppin=None*.

auth_cache

type: string

The Authentication Cache caches the credentials of a successful authentication and allows to use the same credentials - also with an OTP value - for the specified amount of time.

The time to cache the credentials can be specified like "4h", "5m", "2d" (hours, minutes days) or "4h/5m". The notation 4h/5m means, that credentials are cached for 4 hours, but only may be used again, if every 5 minutes the authentication occurs. If the authentication with the same credentials would not occur within 5 minutes, the credentials can not be used anymore.

In future implementations the caching of the credentials could also be dependent on the clients IP address and the user agent.

Note: Cache entries are written to the database table `authcache`. Please note that expired entries are automatically deleted only when the user attempts to log in with the same expired credentials again. In all other cases, expired entries need to be deleted from this table manually by running:

```
pi-manage authcache cleanup --minutes MIN
```

which deletes all cache entries whose last authentication has occurred at least `MIN` minutes ago. As an example:

```
pi-manage authcache cleanup --minutes 300
```

will delete all authentication cache entries whose last authentication happened more than 5 hours ago.

It may make sense to create a cronjob that periodically cleans up old authentication cache entries.

Note: The AuthCache only works for user authentication, not for authentication with serials.

push_text_on_mobile

type: string

This is the text that should be displayed on the push notification during the login process with a *Push Token*. You can choose different texts for different users or IP addresses. This way you could customize push notifications for different applications.

push_title_on_mobile

type: string

This is the title of the push notification that is displayed on the user's smartphone during the login process with a *Push Token*.

push_wait

type: int

This can be set to a number of seconds. If this is set, the authentication with a push token is only performed via one request to `/validate/check`. The HTTP request to `/validate/check` will wait up to this number of seconds and check, if the push challenge was confirmed by the user.

This way push tokens can be used with any non-push-capable applications.

Sensible numbers might be 10 or 20 seconds.

Note: This behaviour can interfere with other tokentypes. Even if the user also has a normal HOTP token, the `/validate/check` request will only return after this number of seconds.

Warning: Using simple webserver setups like Apache WSGI this actually can block all available worker threads, which will cause privacyIDEA to become unresponsive if the number of open PUSH challenges exceeds the number of available worker threads!

challenge_text, challenge_text_header, challenge_test_footer

Using these policies the administrator can modify the challenge texts of e.g. Email-Token or SMS-Token. The action *challenge_text* changes the challenge text in general, no matter which challenge response token is used.

If the *challenge_text_header* is set and if there are more matching challenge response tokens, then the texts of all tokens are concatenated together. Double challenge texts are reduced to one text only.

The *challenge_text_header* and *challenge_text_footer* may contain HTML. If the *challenge_text_header* ends with an `<ul>` or `<ol>`, then all the challenge texts are formatted as an ordered or unordered list. In this case the *challenge_text_footer* also should contain the closing tag.

Note: The footer will only be used, if the header is also set.

indexedsecret_challenge_text

The Indexed Secret Token asks the user to provide the characters of the secret from certain positions. The default text is:

Please enter the position 3,1,6,7 from your secret.

with *3,1,6,7* being the positions of the characters, the user is supposed to enter. This text can be changed with this policy setting. The text needs to contain the python formatting tag *{0!s}* which will be replaced with the list of the requested positions.

For more details of this token type see *Indexed Secret Token*.

indexedsecret_count

The Indexed Secret Token asks the used for a number of characters from a shared secret. The default number to ask is 2.

The number of requested positions can be changed using this policy.

webauthn_allowed_transports

type: string

This action determines, which transports may be used to communicate with the authenticator, during authentication. For instance, if the authenticators used support both an USB connection and NFC wireless communication, they can be limited to USB only using this policy. The allowed transports are given as a space-separated list.

The default is to allow all transports (equivalent to a value of *usb ble nfc internal lightning*).

webauthn_timeout

type: integer

This action sets the time in seconds the user has to confirm an authentication request on his WebAuthn authenticator.

This is a client-side setting, that governs how long the client waits for the authenticator. It is independent of the time for which a challenge for a challenge response token is valid, which is governed by the server and controlled by a separate setting. This means, that if you want to increase this timeout beyond two minutes, you will have to also increase the challenge validity time, as documented in *Challenge Validity Time*.

This setting is a hint. It is interpreted by the client and may be adjusted by an arbitrary amount in either direction, or even ignored entirely.

The default timeout is 60 seconds.

Note: If you set this policy you may also want to set *webauthn_timeout*.

webauthn_user_verification_requirement

type: string

This action configures whether the user's identity should be checked when authenticating with a WebAuthn token. If this is set to required, any user signing in with their WebAuthn token will have to provide some form of verification. This might be biometric identification, or knowledge-based, depending on the authenticator used.

This defaults to *preferred*, meaning user verification will be performed if supported by the token.

Note: User verification is different from user presence checking. The presence of a user will always be confirmed (by asking the user to take action on the token, which is usually done by tapping a button on the authenticator). User verification goes beyond this by ascertaining, that the user is indeed the same user each time (for example through biometric means), only set this to *required*, if you know for a fact, that you have authenticators, that actually support some form of user verification (these are still quite rare in practice).

Note: If you configure this, you will likely also want to configure *webauthn_user_verification_requirement*.

1.7.4 Authorization policies

The scope *authorization* provides means to define what should happen if a user proved his identity and authenticated successfully.

Authorization policies take the realm, the user and the client into account.

Technically the authorization policies apply to the *Validate endpoints* and are checked using *Policy Module* and *Policy Decorators*.

The following actions are available in the scope *authorization*:

tokentype

type: string

Users will only be authorized with this very tokentype. The string can hold a space separated list of case sensitive tokentypes. It should look like:

> hotp totp spass

This is checked after the authentication request, so that a valid OTP value is wasted, so that it can not be used, even if the user was not authorized at this request

Note: Combining this with the client IP you can use this to allow remote access to sensitive areas only with one special token type while allowing access to less sensitive areas with other token types.

serial

type: string

Users will only be authorized with the serial number. The string can hold a regular expression as serial number.

This is checked after the authentication request, so that a valid OTP value is wasted, so that it can not be used, even if the user was not authorized at this request

Note: Combining this with the client IP you can use this to allow remote access to sensitive areas only with hardware tokens like the Yubikey, while allowing access to less secure areas also with a Google Authenticator.

tokeninfo

type: string

Users will only be authorized if the tokeninfo field of the token matches this regular expression.

This is checked after the authentication request, so that a valid OTP value can not be used anymore, even if authorization is forbidden.

A valid action could look like

> action = key/regexp/

Example:

> action = last_auth/^2018.*/

This would mean the tokeninfo field needs to start with “2018”.

setrealm

type: string

This policy is checked before the user authenticates. The realm of the user matching this policy will be set to the realm in this action.

Note: This can be used if the user can not pass his realm when authenticating at a certain client, but the realm needs to be available during authentication since the user is not located in the default realm.

no_detail_on_success

type: bool

Usually an authentication response returns additional information like the serial number of the token that was used to authenticate or the reason why the authentication request failed.

If this action is set and the user authenticated successfully this additional information will not be returned.

no_detail_on_fail

type: bool

Usually an authentication response returns additional information like the serial number of the token that was used to authenticate or the reason why the authentication request failed.

If this action is set and the user fails to authenticate this additional information will not be returned.

api_key_required

type: bool

This policy is checked *before* the user is validated.

You can create an API key, that needs to be passed to use the validate API. If an API key is required, but no key is passed, the authentication request will not be processed. This is used to avoid denial of service attacks by a rogue user sending arbitrary requests, which could result in the token of a user being locked.

You can also define a policy with certain IP addresses without issuing API keys. This would result in “blocking” those IP addresses from using the *validate* endpoint.

You can issue API keys like this:

```
pi-manage api createtoken -r validate
```

The API key (Authorization token) which is generated is valid for 365 days.

The authorization token has to be used as described in *Authentication endpoints*.

auth_max_success

type: string

Here you can specify how many successful authentication requests a user is allowed to perform during a given time. If this value is exceeded, the authentication attempt is canceled.

Specify the value like `2/5m` meaning 2 successful authentication requests per 5 minutes. If during the last 5 minutes 2 successful authentications were performed the authentication request is discarded. The used OTP value is invalidated.

Allowed time specifiers are *s* (second), *m* (minute) and *h* (hour).

Note: This policy depends on reading the audit log. If you use a non readable audit log like *Logger Audit* this policy will not work.

auth_max_fail

type: string

Here you can specify how many failed authentication requests a user is allowed to perform during a given time.

If this value is exceeded, authentication is not possible anymore. The user will have to wait.

If this policy is not defined, the normal behaviour of the failcounter applies. (see *Reset Fail Counter*)

Specify the value like `2/1m` meaning 2 successful authentication requests per minute. If during the last 5 minutes 2 successful authentications were performed the authentication request is discarded. The used OTP value is invalidated.

Allowed time specifiers are *s* (second), *m* (minute) and *h* (hour).

Note: This policy depends on reading the audit log. If you use a non readable audit log like *Logger Audit* this policy will not work.

last_auth

type: string

You can define if an authentication should fail, if the token was not successfully used for a certain time.

Specify a value like `12h`, `123d` or `2y` to disallow authentication, if the token was not successfully used for 12 hours, 123 days or 2 years.

The date of the last successful authentication is store in the *tokeninfo* field of a token and denoted in UTC.

u2f_req

type: string

Only the specified U2F devices are authorized to authenticate. The administrator can specify the action like this:

> u2f_req=subject/.*Yubico.*/

The the key word can be "subject", "issuer" or "serial". Followed by a regular expression. During registration of the U2F device the information from the attestation certificate is stored in the tokeninfo. Only if the regexp matches this value, the authentication with such U2F device is authorized.

add_user_in_response

type: bool

In case of a successful authentication additional user information is added to the response. A dictionary containing user information is added in `detail->user`.

add_resolver_in_response

type: bool

In case of a successful authentication the resolver and realm of the user are added to the response. The names are added in `detail->user-resolver` and `detail->user-realm`.

webauthn_authenticator_selection_list

type: string

This action configures a whitelist of authenticator models which may be authorized. It is a space-separated list of AAGUIDs. An AAGUID is a hexadecimal string (usually grouped using dashes, although these are optional) identifying one particular model of authenticator. To limit enrollment to a few known-good authenticator models, simply specify the AAGUIDs for each model of authenticator that is acceptable. If multiple policies with this action apply, the set of acceptable authenticators will be the union off all authenticators allowed by the various policies.

If this action is not configured, all authenticators will be deemed acceptable, unless limited through some other action.

Note: If you configure this, you will likely also want to configure *webauthn_authenticator_selection_list*

webauthn_req

type: string

This action allows filtering of WebAuthn tokens by the fields of the attestation certificate.

The action can be specified like this:

> webauthn_req=subject/.*Yubico.*/

The the key word can be "subject", "issuer" or "serial". Followed by a regular expression. During registration of the WebAuthn authenticator the information is fetched from the attestation certificate. Only if the attribute in the attestation certificate matches accordingly the token can be enrolled.

Note: If you configure this, you will likely also want to configure *webauthn_req*

1.7.5 Enrollment policies

The scope *enrollment* defines what happens during enrollment either by an administrator or during the user self enrollment.

Enrollment policies take the realms, the client (see *Policies*) and the user settings into account.

Technically enrollment policies control the use of the REST API *Token endpoints* and specially the *init* and *assign*-methods.

Technically the decorators in *API Policies* are used.

The following actions are available in the scope *enrollment*:

max_token_per_realm

type: int

This is the maximum allowed number of tokens in the specified realm.

Note: If you have several realms with realm admins and you imported a pool of hardware tokens you can thus limit the consumed hardware tokens per realm.

Note: If there are multiple matching policies, the *highest* maximum allowed number of tokens among the matching policies is enforced. Policy priorities are ignored.

max_token_per_user

type: int

Limit the maximum number of tokens per user in this realm.

There are also token type specific policies to limit the number of tokens of a specific token type, that a user is allowed to have assigned.

Note: If you do not set this action, a user may have unlimited tokens assigned.

Note: If there are multiple matching policies, the *highest* maximum allowed number of tokens among the matching policies is enforced. Policy priorities are ignored.

max_active_token_per_user

type: int

Limit the maximum number of active tokens per user.

There are also token type specific policies to limit the number of tokens of a specific token type, that a user is allowed to have assigned.

Note: Inactive tokens will not be taken into account. If the token already exists, it can be recreated if the token is already active.

tokenissuer

type: string

This sets the issuer label for a newly enrolled Google Authenticator. This policy takes a fixed string, to add additional information about the issuer of the soft token.

Starting with version 2.20 you can use the tags `{user}`, `{realm}`, `{serial}` and as new tags `{givenname}` and `{surname}` in the field issuer.

Note: A good idea is to set this to the instance name of your privacyIDEA installation or the name of your company.

tokenlabel

type: string

This sets the label for a newly enrolled Google Authenticator. Possible tags to be replaces are <u> for user, <r> for realm an <s> for the serial number.

The default behaviour is to use the serial number.

Note: This is useful to identify the token in the Authenticator App.

Note: Starting with version 2.19 the usage of `<u>`, `<s>` and `<r>` is deprecated. Instead you should use `{user}`, `{realm}`, `{serial}` and as new tags `{givenname}` and `{surname}`.

Warning: If you are only using `<u>` or `{user}` as tokenlabel and you enroll the token without a user, this will result in an invalid QR code, since it will have an empty label. You should rather use a label like "{user}@{realm}", which would result in "@".

autoassignment

type: string

allowed values: any_pin, userstore

Users can assign a token just by using this token. The user can take a token from a pool of unassigned tokens. When this policy is set, and the user has no token assigned, autoassignment will be done: The user authenticates with a new PIN or his userstore password and an OTP value from the token. If the OTP value is correct the token gets assigned to the user and the given PIN is set as the OTP PIN.

Note: Requirements are:

1. The user must have no other tokens assigned.
2. The token must be not assigned to any user.
3. The token must be located in the realm of the authenticating user.
4. (The user needs to enter the correct userstore password)

Warning: If you set the policy to *any_pin* the token will be assigned to the user no matter what pin he enters. In this case assigning the token is only a one-factor-authentication: the possession of the token.

otp_pin_random

type: int

Generates a random OTP PIN of the given length during enrollment. Thus the user is forced to set a certain OTP PIN.

Note: To use the random PIN, you also need to define a *pinhandling* policy.

pinhandling

type: string

If the `otp_pin_random` policy is defined, you can use this policy to define, what should happen with the random pin. The action value take the class of a PinHandler like `privacyidea.lib.pinhandling.base.PinHandler`. The base PinHandler just logs the PIN to the log file. You can add classes to send the PIN via EMail or print it in a letter.

For more information see the base class *PinHandler*.

change_pin_on_first_use

type: bool

If the administrator enrolls a token or resets a PIN of a token, then the PIN of this token is marked to be changed on the first (or next) use. When the user authenticates with the old PIN, the user is authenticated successfully. But the detail-response contains the keys "next_pin_change" and "pin_change". If "pin_change" is *True* the authenticating application must trigger the change of the PIN using the API */token/setpin*. See *Token endpoints*.

Note: If the application does not honour the "pin_change" attribute, then the user can still authenticate with his old PIN.

change_pin_every

type: string

This policy requires the user to change the PIN of his token on a regular basis. Enter a value followed by "d", e.g. change the PIN every 180 days will be "180d".

The date, when the PIN needs to be changed, is returned in the API response of */validate/check*. For more information see *change_pin_on_first_use*. To specifiy the contents of the PIN see *User Policies*.

otp_pin_encrypt

type: bool

If set the OTP PIN of a token will be encrypted. The default behaviour is to hash the OTP PIN, which is safer.

lostTokenPWLen

type: int

This is the length of the generated password for the lost token process.

lostTokenPWContents

type: string

This is the contents that a generated password for the lost token process should have. You can use

- c: for lowercase letters
- n: for digits
- s: for special characters (!#$%&()*+,-./:;<=>?@[]^_)
- C: for uppercase letters
- 8: Base58 character set

Example:

The action *lostTokenPWLen=10, lostTokenPWContents=Cns* could generate a password like *AC#!49MK))*.

Note: If you combine `8` with e.g. `C` there will be double characters like "A", "B"… Thus, those characters will have a higher probability of being part of the password. Also `C` would again add the character "I", which is not part of Base58.

lostTokenValid

type: int

This is how many days the replacement token for the lost token should be valid. After this many days the replacement can not be used anymore.

yubikey_access_code

type: string

This is a 12 character long access code in hex format to be used to initialize yubikeys. If no access code is set, yubikeys can be re-initialized by everybody. You can choose a company wide access code, so that Yubikeys can only be re-initialized by your own system.

You can add two access codes separated by a colon to change from one access code to the other.

> 313233343536:414243444546

papertoken_count

type: int

This is a specific action of the paper token. Here the administrator can define how many OTP values should be printed on the paper token.

tantoken_count

type: int

This is a specific action for the TAN token. The administrator can define how many TANs will be generated and printed.

u2f_req

type: string

Only the specified U2F devices are allowed to be registered. The action can be specified like this:

> u2f_req=subject/.*Yubico.*/

The the key word can be "subject", "issuer" or "serial". Followed by a regular expression. During registration of the U2F device the information is fetched from the attestation certificate. Only if the attribute in the attestation certificate matches accordingly the token can be registered.

u2f_no_verify_certificate

type: bool

By default the validity period of the attestation certificate of a U2F device gets verified during the registration process. If you do not want to verify the validity period, you can check this action.

{type}_2step_clientsize, {type}_2step_serversize, {type}_2step_difficulty

type: string

These are token type specific parameters. They control the key generation during the 2step token enrollment (see *Two Step Enrollment*).

The `serversize` is the optional size (in bytes) of the server's key part. The `clientsize` is the size (in bytes) of the smartphone's key part. The `difficulty` is a parameter for the key generation. In the implementation in version 2.21 PBKDF2 is used. In this case the `difficulty` specifies the number of rounds.

This is new in version 2.21.

---

type: bool

During enrollment of a privacyIDEA Authenticator smartphone app this policy is used to force the user to protect the token with a PIN.

Note: This only works with the privacyIDEA Authenticator. This policy has no effect, if the QR code is scanned with other smartphone apps.

This is new in version 3.1.

push_firebase_configuration

type: string

For enrolling a *Push Token*, the administrator can select which Firebase configuration should be used. The administrator can create several connections to the Firebase service (see *Firebase Provider*). This way even different Firebase configurations could be used depending on the user's realm or the IP address.

This is new in version 3.0.

webauthn_relying_party_id

type: string

This action sets the relying party id to use for the enrollment of new WebAuthn tokens, at defined by the WebAuthn specification[1]. Please note, that a token will be rolled out with one particular ID and that the relying party of an existing token can not be changed. In order to change the relying party id for existing tokens, they need to be deleted and new tokens need to be enrolled. This is a limitation of the WebAuthn standard and is unlikely to change in the future.

The relying party id is a valid domain string that identifies the WebAuthn Relying Party on whose behalf a given registration or authentication ceremony is being performed. A public key credential can only be used for authentication with the same entity (as identified by RP ID) it was registered with.

[1] https://w3.org/TR/webauthn-2/#rp-id

This id needs to be a registrable suffix of or equal to the effective domain for each webservice the tokens should be used with. This means if the token is being enrolled on – for example – *https://login.example.com*, them the relying party ID may be either *login.example.com*, or *example.com*, but not – for instance – *m.login.example.com*, or *com*. Similarly, a token enrolled with a relying party ID of *login.example.com* might be used by *https://login.example.com*, or even *https://m.login.example.com:1337*, but not by *https://example.com* (because the RP ID *login.example.com* is not a valid relying party ID for the domain *example.com*).

Note: This action needs to be set to be able to enroll WebAuthn tokens. For an overview of all the settings required for the use of WebAuthn, see *WebAuthn Token Config*.

webauthn_relying_party_name

type: string

This action sets the human-readable name for the relying party, as defined by the WebAuthn specification[2]. It should be the name of the entity whose web applications the WebAuthn tokens are used for.

Note: This action needs to be set to be able to enroll WebAuthn tokens. For an overview of all the settings required for the use of WebAuthn, see *WebAuthn Token Config*.

webauthn_timeout

type: integer

This action sets the time in seconds the user has to confirm enrollment on his WebAuthn authenticator.

This is a client-side setting, that governs how long the client waits for the authenticator. It is independent of the time for which a challenge for a challenge response token is valid, which is governed by the server and controlled by a separate setting. This means, that if you want to increase this timeout beyond two minutes, you will have to also increase the challenge validity time, as documented in *Challenge Validity Time*.

This setting is a hint. It is interpreted by the client and may be adjusted by an arbitrary amount in either direction, or even ignored entirely.

The default timeout is 60 seconds.

Note: If you set this policy you may also want to set *webauthn_timeout*.

webauthn_authenticator_attachment

type: string

This action configures whether to limit roll out of WebAuthn tokens to either only platform authenticators, or only platform authenticators. Cross-platform authenticators are authenticators, that are intended to be plugged into different devices, whereas platform authenticators are those, that are built directly into one particular device and can not (easily) be removed and plugged into a different device.

The default is to allow both *platform* and *cross-platform* attachment for authenticators.

[2] https://w3.org/TR/webauthn-2/#webauthn-relying-party

webauthn_authenticator_selection_list

type: string

This action configures a whitelist of authenticator models which may be enrolled. It is a space-separated list of AAGUIDs. An AAGUID is a hexadecimal string (usually grouped using dashes, although these are optional) identifying one particular model of authenticator. To limit enrollment to a few known-good authenticator models, simply specify the AAGUIDs for each model of authenticator that is acceptable. If multiple policies with this action apply, the set of acceptable authenticators will be the union off all authenticators allowed by the various policies.

If this action is not configured, all authenticators will be deemed acceptable, unless limited through some other action.

Note: If you configure this, you will likely also want to configure *webauthn_authenticator_selection_list*.

webauthn_user_verification_requirement

type: string

This action configures whether the user's identity should be checked when rolling out a new WebAuthn token. If this is set to required, any user rolling out a new WebAuthn token will have to provide some form of verification. This might be biometric identification, or knowledge-based, depending on the authenticator used.

This defaults to *preferred*, meaning user verification will be performed if supported by the token.

Note: User verification is different from user presence checking. The presence of a user will always be confirmed (by asking the user to take action on the token, which is usually done by tapping a button on the authenticator). User verification goes beyond this by ascertaining, that the user is indeed the same user each time (for example through biometric means), only set this to *required*, if you know for a fact, that you have authenticators, that actually support some form of user verification (these are still quite rare in practice).

Note: If you configure this, you will likely also want to configure *webauthn_user_verification_requirement*.

webauthn_public_key_credential_algorithm_preference

type: string

This action configures which algorithms should be preferred for the creation of WebAuthn asymmetric cryptography key pairs, and in which order. privacyIDEA currently supports ECDSA as well as RSASSA-PSS. Please check back with the manufacturer of your authenticators to get information on which algorithms are acceptable to your model of authenticator.

The default is to allow both ECDSA and RSASSA-PSS, but to prefer ECDSA over RSASSA-PSS.

Note: Not all authenticators will supports all algorithms. It should not usually be necessary to configure this action. Do *not* change this preference, unless you are sure you know what you are doing!

webauthn_authenticator_attestation_form

type: string

This action configures whether to request attestation data when enrolling a new WebAuthn token. Attestation is used to verify, that the authenticator being enrolled has been made by a trusted manufacturer. Since depending on the authenticator this may include personally identifying information, *indirect* attestation can be requested. If *indirect* attestation is requested the client may pseudonymize the attestation data. Attestation can also be turned off entirely.

The default is to request *direct* (full) attestation from the authenticator.

Note: In a normal business-context it will not be necessary to change this. If this is set to *none*, *webauthn_authenticator_attestation_level* must also be none.

Note: Authenticators enrolled with this option set to *none* can not be filtered using *webauthn_req* and *webauthn_authenticator_selection_list* or *webauthn_req* and *webauthn_authenticator_selection_list*, respectively. Applying these filters is not possible without attestation information, since the fields these actions rely upon will be missing. With *indirect* attestation, checking may be possible (depending on the client). If any of *webauthn_req*, *webauthn_authenticator_selection_list*, *webauthn_req*, or *webauthn_authenticator_selection_list* are set and apply to a request for a token without attestation information, access will be denied.

webauthn_authenticator_attestation_level

type: string

This action determines whether and how strictly to check authenticator attestation data. Set this to *none*, to allow any authenticator, even if the attestation information is missing completely. If this is set to *trusted*, strict checking is performed. No authenticator is allowed, unless it contains attestation information signed by a certificate trusted for attestation.

Note: Currently the certificate that signed the attestation needs to be trusted directly. Traversal of the trust path is not yet supported!

The default is *untrusted*. This will perform the attestation check like normal, but will not fail the attestation, if the attestation is self-signed, or signed by an unknown certificate.

Note: In order to be able to use *trusted* attestation, a directory needs to be provided, containing the certificates trusted for attestation. See *WebAuthn Token Config* for details.

Note: If this is set to *untrusted*, a manipulated token could send a self-signed attestation message with modified a modified AAGUID and faked certificate fields in order to bypass *webauthn_req* and *webauthn_authenticator_selection_list*, or *webauthn_req* and *webauthn_authenticator_selection_list*, respectively. If this is of concern for your attack scenarios, please make sure to properly configure your attestation roots!

webauthn_req

type: string

This action allows filtering of WebAuthn tokens by the fields of the attestation certificate.

The action can be specified like this:

> webauthn_req=subject/.*Yubico.*/

The the key word can be "subject", "issuer" or "serial". Followed by a regular expression. During registration of the WebAuthn authenticator the information is fetched from the attestation certificate. Only if the attribute in the attestation certificate matches accordingly the token can be enrolled.

Note: If you configure this, you will likely also want to configure *webauthn_req*.

1.7.6 WebUI Policies

login_mode

type: string

allowed values: "userstore", "privacyIDEA", "disable"

If set to *userstore* (default), users and administrators need to authenticate with the password of their userstore, being an LDAP service or an SQL database.

If this action is set to *login_mode=privacyIDEA*, the users and administrators need to authenticate against privacyIDEA when logging into the WebUI. I.e. they can not login with their domain password anymore but need to authenticate with one of their tokens.

If set to *login_mode=disable* the users and administrators of the specified realms can not login to the UI anymore.

Warning: If you set this action and the user deletes or disables all his tokens, he will not be able to login anymore.

Note: Administrators defined in the database using the pi-manage command can still login with their normal passwords.

Note: A sensible way to use this, is to combine this action in a policy with the `client` parameter: requiring the users to login to the Web UI remotely from the internet with OTP but still login from within the LAN with the domain password.

Note: Another sensible way to use this policy is to *disable* the login to the web UI either for certain IP addresses (`client`) or for users in certain realms.

remote_user

type: string

This policy defines, if the login to the privacyIDEA using the web servers integrated authentication (like basic authentication or digest authentication) should be allowed.

Possible values are "disable" and "allowed".

Note: The policy is evaluated before the user is logged in. At this point in time there is no realm known, so a policy to allow remote_user must not select any realm.

Note: The policy *login_mode* and *remote_user* work independent of each other. I.e. you can disable *login_mode* and allow *remote_user*.

You can use this policy to enable Single-Sign-On and integration into Kerberos or Active Directory. Add the following template into you apache configuration in /etc/apache2/sites-available/privacyidea.conf:

```
<Directory />
        # For Apache 2.4 you need to set this:
        # Require all granted
        Options FollowSymLinks
        AllowOverride None

        SSLRequireSSL
        AuthType Kerberos
        AuthName "Kerberos Login"
        KrbMethodNegotiate On
        KrbMethodK5Passwd On
        KrbAuthRealms YOUR-REALM
        Krb5KeyTab /etc/apache2/http.keytab
        KrbServiceName HTTP
        KrbSaveCredentials On
        <RequireAny>
            # Either we need a URL with no authentication or we need a valid user
            <RequireAny>
                # Any of these URL do NOT need a basic authentication
                Require expr %{REQUEST_URI} =~ m#^/validate#
                Require expr %{REQUEST_URI} =~ m#^/ttype#
            </RequireAny>
            Require valid-user
        </RequireAny>
</Directory>
```

logout_time

type: int

Set the timeout, after which a user in th WebUI will be logged out. The default timeout is 120 seconds.

Being a policy this time can be set based on clients, realms and users.

token_page_size

type: int

By default 15 tokens are displayed on one page in the token view. On big screens you might want to display more tokens. Thus you can define in this policy how many tokens should be displayed.

user_page_size

type: int

By default 15 users are displayed on one page in the user view. On big screens you might want to display more users. Thus you can define in this policy how many users should be displayed.

policy_template_url

type: str

Here you can define a URL from where the policies should be fetched. The default URL is a Github repository **[#defaulturl]_**.

Note: When setting a template_url policy the modified URL will only get active after the user has logged out and in again.

default_tokentype

type: str

You can define which is the default tokentype when enrolling a new token in the Web UI. This is the token, which will be selected, when entering the enrollment dialog.

tokenwizard

type: bool

If this policy is set and the user has no token, then the user will only see an easy token wizard to enroll his first token. If the user has enrolled his first token and he logs in to the web UI, he will see the normal view.

The user will enroll a token defined in *default_tokentype*.

Other sensible policies to combine are in *User Policies* the OTP length, the TOTP timestep and the HASH-lib.

You can add a prologue and epilog to the enrollment wizard in the greeting and after the token is enrolled and e.g. the QR code is displayed.

Create the files

- static/customize/views/includes/token.enroll.pre.top.html
- static/customize/views/includes/token.enroll.pre.bottom.html
- static/customize/views/includes/token.enroll.post.top.html
- static/customize/views/includes/token.enroll.post.bottom.html

to display the contents in the first step (pre) or in the second step (post).

Note: You can change the directory *static/customize* to a URL that fits your needs the best by defining a variable *PI_CUSTOMIZATION* in the file *pi.cfg*. This way you can put all modifications in one place apart from the original code.

If you want to adapt the privacyIDEA look and feel even more, read *Customization*.

realm_dropdown

type: str

If this policy is activated the web UI will display a realm dropdown box. Of course this policy can not filter for users or realms, since the user is not known at this moment.

The type of this action was changed to "string" in version 2.16. You can set a space separated list of realm names. Only these realmnames are displayed in the dropdown box.

Note: The realm names in the policy are not checked, if they really exist!

search_on_enter

type: bool

The searching in the user list is performed as live search. Each time a key is pressed, the new substring is searched in the user store.

Sometimes this can be too time consuming. You can use this policy to change the bahaviour that the administrator needs to press *enter* to trigger the search.

(Since privacyIDEA 2.17)

custom_baseline

type: str

The administrator can replace the file `templates/baseline.html` with another template. This way he can change the links to e.g. internal documentation or ticketing systems. The new file could be called `mytemplates/mybase.html`.

This will only work with a valid subscription of privacyIDEA Enterprise Edition.

Note: This policy is evaluated before login. So any realm or user setting will have no effect. But you can specify different baselines for different client IP addresses.

If you want to adapt the privacyIDEA look and feel even more, read *Customization*.

(Since privacyIDEA 2.21)

custom_menu

type: str

The administrator can replace the file `templates/menu.html` with another template. This way he can change the links to e.g. internal documentation or ticketing systems. The new file could be called `mytemplates/mymenu.html`.

This will only work with a valid subscription of privacyIDEA Enterprise Edition.

Note: This policy is evaluated before login. So any realm or user setting will have no effect. But you can specify different menus for different client IP addresses.

If you want to adapt the privacyIDEA look and feel even more, read *Customization*.

(Since privacyIDEA 2.21)

hide_buttons

type: bool

Buttons for actions that a user is not allowed to perform, are hidden instead of being disabled.

(Since privacyIDEA 3.0)

login_text

type: str

This way the text "Please sign in" on the login dialog can be changed. Since the policy can also depend on the IP address of the client, you can also choose different login texts depending on from where a user tries to log in.

(Since privacyIDEA 3.0)

show_android_privacyidea_authenticator

type: bool

If this policy is activated, the enrollment page for HOTP, TOTP and Push tokens will contain a QR code, that leads the user to the Google Play Store where he can directly install the privacyIDEA Authenticator App for Android devices.

(Since privacyIDEA 3.3)

show_ios_privacyidea_authenticator

If this policy is activated, the enrollment page for HOTP, TOTP and Push tokens will contain a QR code, that leads the user to the Apple App Store where he can directly install the privacyIDEA Authenticator App for iOS devices.

type: bool

(Since privacyIDEA 3.3)

show_custom_authenticator

type: str

If this policy is activated, the enrollment page for HOTP, TOTP and Push tokens will contain a QR code, that leads the user to the given URL.

The idea is, that an organization running privacyIDEA can create its own URL, where the user is taken to, e.g.

- Show information about the used Authenticator apps…
- Do a device identification and automatically redirect the user to Google Play Store or Apple App Store. Thus only need the user to show *one* QR code…
- If an organization has it's own customized app or chooses to use another app, lead the user to another App in the Google Play Store or Apple App Store.

Other scenarios are possible.

(Since privacyIDEA 3.3)

indexedsecret_preset_attribute

type: str

The secret in the enrollment dialog of the tokentype *indexedsecret* is preset with the value of the given user attribute.

For more details of this token type see *Indexed Secret Token*.

(Since privacyIDEA 3.3)

1.7.7 Register Policy

User registration

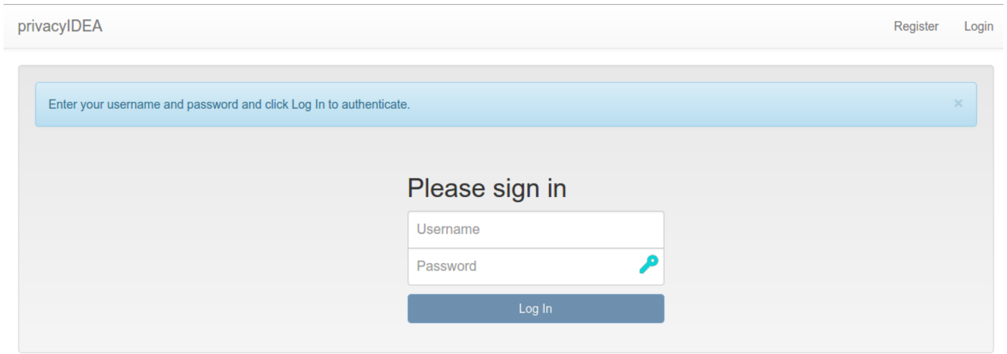
Starting with privacyIDEA 2.10 users are allowed to register with privacyIDEA. I.e. a user that does not exist in a given realm and resolver can create a new account.

Note: Registering new users is only possible, if there is a writeable resolver and if the necessary policy in the scope *register* is defined. For editable UserIdResolvers see *UserIdResolvers*.

If a register policy is defined, the login window of the Web UI gets a new link "Register".

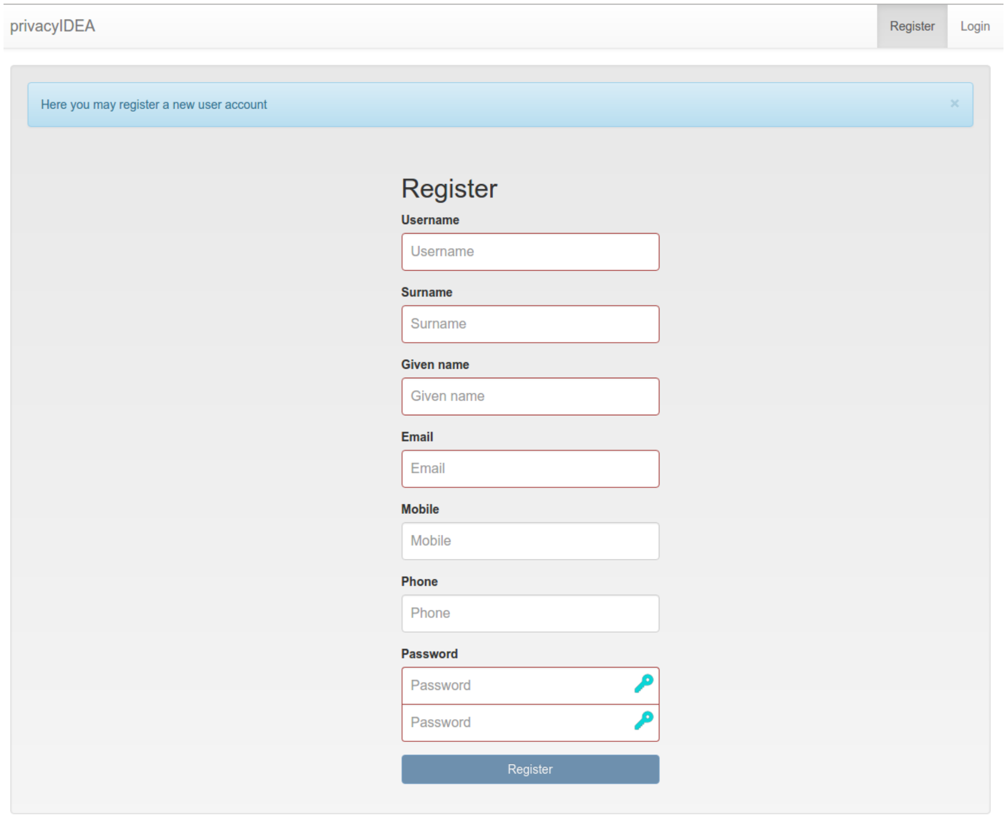
A user who clicks the link to register a new account gets this registration dialog:

During registration the user is also enrolled *Registration* token. This registration code is sent to the user via a notification email.



Enter your username and password and click Log In to authenticate.

Fig. 54: *Next to the login button is a new link 'register', so that new users are able to register.*



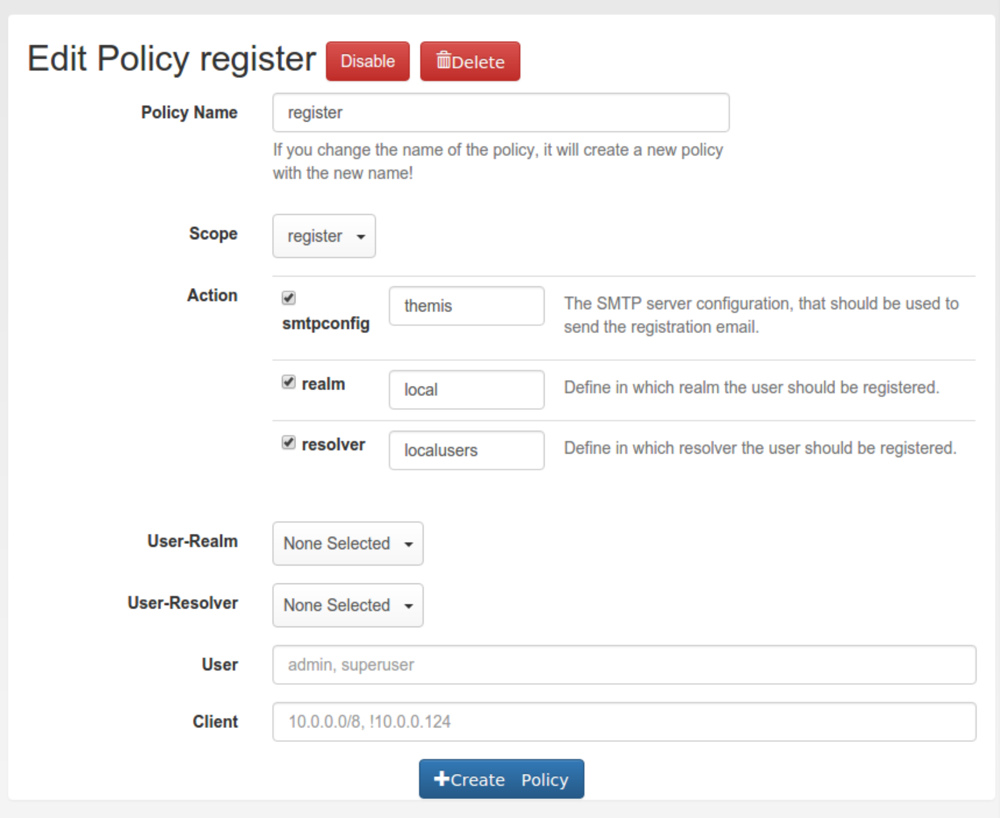
Here you may register a new user account

Fig. 55: *Registration form*

Note: Thus - using the right policies in scope *webui* and *authentication* - the user could login with the password he set during registration an the registration code he received via email.

Policy settings

In the scope *register* several settings define the behaviour of the registration process.



Edit Policy register

Policy Name: register

If you change the name of the policy, it will create a new policy with the new name!

Scope: register

Action:

☑ smtpconfig	themis	The SMTP server configuration, that should be used to send the registration email.
☑ realm	local	Define in which realm the user should be registered.
☑ resolver	localusers	Define in which resolver the user should be registered.

User-Realm: None Selected

User-Resolver: None Selected

User: admin, superuser

Client: 10.0.0.0/8, !10.0.0.124

Fig. 56: *Creating a new registration policy*

realm

type: string

This is the realm, in which a new user will be registered. If this realm is not specified, the user will be registered in the default realm.

resolver

type: string

This is the resolver, in which the new user will be registered. If this resolver is not specified, **registration is not possible!**

Note: This resolver must be an editable resolver, otherwise the user can not be created in this resolver.

smtpconfig

type: string

This is the unique identifier of the *SMTP server configuration*. This SMTP server is used to send the notification email with the registration code during the registration process.

Note: If there is no *smtpconfig* or set to a wrong identifier, the user will get no notification email.

requiredemail

type: string

This is a regular expression according to[1].

Only email addresses matching this regular expression are allowed to register.

Example: If you want to authenticate the user only by the OTP value, no matter what OTP PIN he enters, a policy might look like this:

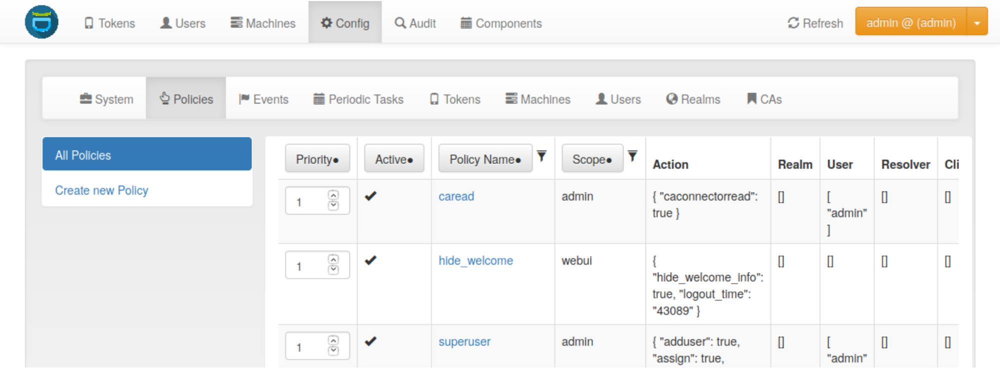
```
action: requiredemail=/.*@mydomain\..*/
```

This will allow all email addresses from the domains *mydomain.com*, *mydomain.net* etc. . .

You can define as many policies as you wish to. The logic of the policies in the scopes is additive.

Starting with privacyIDEA 2.5 you can use policy templates to ease the setup.

[1] https://docs.python.org/2/library/re.html

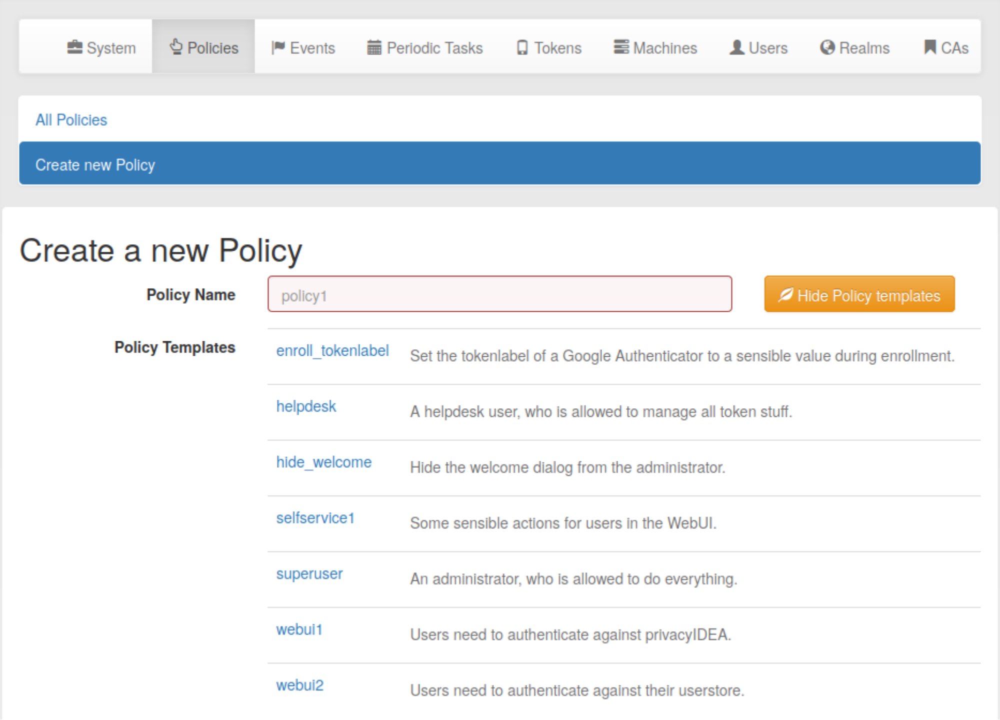


Priority	Active	Policy Name	Scope	Action	Realm	User	Resolver	Cli
1	✔	caread	admin	{ "caconnectorread": true }	[]	["admin"]	[]	[]
1	✔	hide_welcome	webui	{ "hide_welcome_info": true, "logout_time": "43089" }	[]	[]	[]	[]
1	✔	superuser	admin	{ "adduser": true, "assign": true,	[]	["admin"	[]	[]

Fig. 57: *Policy Definition*

1.7.8 Policy Templates

Policy templates are defined in a Github repository which can be changed using a WebUI policy *policy_template_url*. The templates are fetched from the given repository URL during runtime.



Create a new Policy

Policy Name: policy1

Policy Templates

enroll_tokenlabel	Set the tokenlabel of a Google Authenticator to a sensible value during enrollment.
helpdesk	A helpdesk user, who is allowed to manage all token stuff.
hide_welcome	Hide the welcome dialog from the administrator.
selfservice1	Some sensible actions for users in the WebUI.
superuser	An administrator, who is allowed to do everything.
webui1	Users need to authenticate against privacyIDEA.
webui2	Users need to authenticate against their userstore.

The policy templates are json files, which can contain common settings, that can be used to start your own polices. When creating a new policy, you can select an existing policy template as a starting point.

You may also fork the github repository and commit pull request to improve the policy templates. Or you may fork the github repository and use your own policy template URL for your policy templates.

A policy templates looks like this:

```
{
 "name": "template_name1",
 "scope": "enrollment",
```

(continues on next page)

(continued from previous page)

```
 "action": {
            "tokenlabel": "<u>@<r>/<s>",
            "autoassignment": true
           }
}
```

realms, *resolver* and *clients* are not used in the templates.

A template must be referenced in a special `index.json` file:

```
{
  "template_name1": "description1",
  "template_name2": "description2"
}
```

where the key is the name of the template file and the value is a description displayed in the WebUI.

Each policy can contain the following attributes:

policy name

A unique name of the policy. The name is the identifier of the policy. If you create a new policy with the same name, the policy is overwritten.

Note: In the web UI and the API policies can only be created with the characters 0-9, a-z, A-Z, “_”, “-“, ” ” and “.”. On a library level or during migration scripts policies with other characters could be created.

scope

The scope of the policy as described above.

action

This is the important part of the policy. Each scope provides its own set of actions. An action describes that something is *allowed* or that some behaviour is configured. A policy can contain several actions. Actions can be of type *boolean*, *string* or *integer*. Boolean actions are enabled by just adding this action - like `scope=user:action=disable`, which allows the user to disable his own tokens. *string* and *integer* actions require an additional value - like `scope=authentication:action='otppin=userstore'`.

user

This is the user, for whom this policy is valid. Depending on the scope the user is either an administrator or a normal authenticating user.

If this field is left blank, this policy is valid for all users.

resolver

This policy will be valid for all users in this resolver.

If this field is left blank, this policy is valid for all resolvers.

Note: Starting with version 2.17 you can use the parameter `check_all_resolvers`. This is *Check all possible resolvers of a user to match the resolver in this policy* in the Web UI.

Assume a user *user@realm1* is contained in *resolver1* and *resolver2* in the realm *realm1*, where *resolver1* is the resolver with the highest priority. If the user authenticates as *user@realm1*, only policies for *resolver1* will match, since the user is identified as *user.resolver1@realm1*.

If you also want to match a policy with `resolver=resolver2`, you need to select *Check all possible resolvers* in this policy. Thus this policy will match for all users, which are als contained in *resolver2* as a secondary resolver.

realm

This is the realm, for which this policy is valid.

If this field is left blank, this policy is valid for all realms.

client

This is the requesting client, for which this action is valid. I.e. you can define different policies if the user access is allowed to manage his tokens from different IP addresses like the internal network or remotely via the firewall.

You can enter several IP addresses or subnets divided by comma. Exclude item by prepending a minus sign (like `10.2.0.0/16, -10.2.0.1, 192.168.0.1`).

time

(added in privacyIDEA 2.12)

In the time field of a policy you can define a list of time ranges. A time range can consist of day of weeks (*dow*) and of times in 24h format. Possible values are:

```
<dow>: <hh>-<hh>
<dow>: <hh:mm>-<hh:mm>
<dow>-<dow>: <hh:mm>-<hh:mm>
```

You may use any combination of these. Like:

```
Mon-Fri: 8-18
```

to define certain policies to be active throughout working hours.

Note: If the time of a policy does not match, the policy is not found. Thus you can get effects you did not plan. So think at least *twice* before using time restricted policies.

priority

(added in privacyIDEA 2.23)

The priority field of policies contains a positive number and defaults to 1. In case of policy conflicts, policies with a lower priority number take precedence.

It can be used to resolve policy conflicts. An example is the *passthru* policy: Assume there are two passthru policies `pol1` and `pol2` that define different action values, e.g. `pol1` defines `passthru=userstore` and `pol2` defines `passthru=radius1`. If multiple policies match for an incoming authentication request, the priority value is used to determine the policy that should take precedence: Assuming `pol1` has a priority of 3 and `pol2` has a priority of 2, privacyIDEA will honor only the `pol2` policy and authenticate the user against the RADIUS server `radius1`.

Policy conflicts can still occur if multiple policies with the same priority specify different values for the same action.

additional conditions

(added in privacyIDEA 3.1)

Using conditions, you can specify more advanced rules that determine whether a policy is valid for a request.

Conditions are described in

1.7.9 Policy conditions

Since privacyIDEA 3.1, *policy conditions* allow to define more advanced rules for policy matching, i.e. for determining which policies are valid for a specific request.

Conditions can be added to a policy via the WebUI. In order for a policy to take effect during the processing of a request, the request has to match not only the ordinary policy attributes (see *Policies*), but also *all* additionally defined conditions that are currently active. If no active conditions are defined, only the ordinary policy attributes are taken into account.

Each policy condition performs a comparison of two values. The left value is taken from the current request. The comparison operator (called *Comparator*) and the right value are entered in the policy definition. Each condition consists of five parts:

- `Active` determines if the condition is currently active.
- `Section` refers to an aspect of the incoming request on which the condition is applied. The available sections are predefined, see *Sections*.
- The meaning of `Key` depends on the chosen `Section`. Typically, it determines the exact property of the incoming request on which the condition is applied.
- `Comparator` defines the comparison to be performed. The available comparators are predefined, see *Comparators*.
- `Value` determines the value the property should be compared against.

Sections

privacyIDEA 3.1 implements only one section, which is called `userinfo`.

`userinfo`

The section `userinfo` can be used to define conditions that are checked against attributes of the current user in the request (the so-called *handled user*). The validity of a policy condition with section `userinfo` is determined as follows:

- privacyIDEA retrieves the userinfo of the currently handled user. These are the user attributes as they are determined by the respective resolver. This is configured via the attribute mappings of resolvers (see *UserIdResolvers*).
- Then, it retrieves the userinfo attribute given by `Key`
- Finally, it uses the `Comparator` to compare the contents of the userinfo attribute with the given `Value`. The result of the comparison determines if the request matches the condition or not.

Note: There are situations in which the currently handled user cannot be determined. If privacyIDEA encounters a policy with `userinfo` conditions in such a situation, it throws an error and the current request is aborted.

Likewise, privacyIDEA raises an error if `Key` refers to an unknown userinfo attribute, or if the condition definition is invalid due to some other reasons. More detailed information are then written to the logfile.

As an example for a correct and useful `userinfo` condition, let us assume that you have configured a realm *ldaprealm* with a single LDAP resolver called *ldapres*. This resolver is configured to fetch users from a OpenLDAP server, with the following attribute mapping:

```
{ "phone": "telephoneNumber",
  "mobile": "mobile",
  "email": "mailPrimaryAddress",
  "groups": "memberOf",
  "surname": "sn",
  "givenname": "givenName" }
```

You can further define `groups` to be a multi-value attribute by setting the *Multivalue Attributes* option to `["groups"]`.

According to this mapping, users of *ldaprealm* will have userinfo entries `phone`, `mobile`, `email`, `groups`, `surname` and `givenname` which are filled with the respective values from the LDAP directory.

You can now configure a policy that disables the WebUI login for all users in the LDAP group `cn=Restricted Login,cn=groups,dc=test,dc=intranet` with an email address ending in `@example.com`:

- **Scope**: webui
- **Action**: `login_mode=disable`
- 1) **additional condition** (active):
    - **Section**: `userinfo`
    - **Key**: `email`
    - **Comparator**: `matches`
    - **Value**: `.*@example.com`

  2) **additional condition** (active):
    - **Section**: `userinfo`
    - **Key**: `groups`
    - **Comparator:** `contains`
    - **Value**: `cn=Restricted Login,cn=groups,dc=test,dc=intranet`

The policy only takes effect if the user that is trying to log in has a matching email address *and* is a member of the specified group. In other words, members of the group with an email address ending in `@privacyidea.org` will still be allowed to log in.

Note: Keep in mind that changes in the LDAP directory may not be immediately visible to privacyIDEA due to caching settings (see *LDAP resolver*).

If the userinfo of the user that is trying to log in does not contain attributes `email` or `groups` (due to a resolver misconfiguration, for example), privacyIDEA throws an error and the request is aborted.

`HTTP Request Header`

The section `HTTP Request header` can be used to define conditions that are checked against the request header key-value pairs.

The `Key` specifies the request header key. It is case-sensitive.

privacyIDEA uses the `Comparator` to check if the value of a header is equal or a substring of the required value.

Note: privacyIDEA raises an error if `Key` refers to an unknown request header. If the header in question is missing, the policy can not get completely evaluated. Be aware that requests, that do not contain the header `Key` will always fail! Thus, if you are using uncommon headers you should in addition restrict the policy e.g. to client IPs, to assure, that a request from this certain IP address will always contain the header, that is to be checked.

Comparators

The following comparators can be used in definitions of policy conditions:

- `equals` evaluates to true if the left value is equal to the right value, according to Python semantics. `!equals` evaluates to true if this is not the case.
- `contains` evaluates to true if the left value (a list) contains the right value as a member. `!contains` evaluates to true if this is not the case.
- `in` evaluates to true if the left value is contained in the list of values given by the right value. The right value is a comma-separated list of values. Individual values can be quoted using double-quotes. `!in` evaluates to true if the left value is not found in the list given by the right value.
- `matches` evaluates to true if the left value completely matches the regular expression given by the right value. `!matches` evaluates to true if this is not the case.

Error Handling

privacyIDEA's error handling when checking policy conditions is quite strict, in order to prevent policy misconfiguration from going unnoticed. If privacyIDEA encounters a policy condition that evaluates neither to true nor false, but simply *invalid* due to a misconfiguration, privacyIDEA throws an error and the current request is aborted.

1.8 Event Handler

Added in version 2.12.

What is the difference between *Policies* and event handlers?

Policies are used to define the behaviour of the system. With policies you can *change* the way the system reacts.

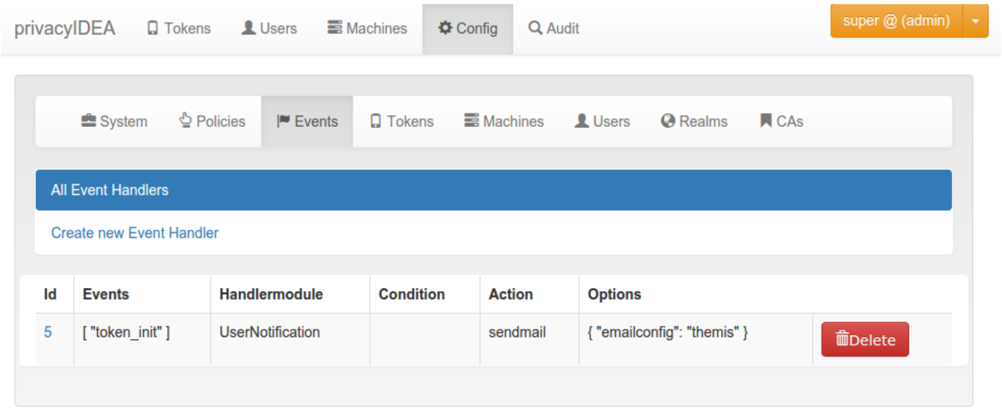
With event handlers you do not change the way the system reacts. But on certain events you can *trigger a new action* in addition to the behaviour defined in the policies.

These additional actions are also logged to the audit log. These actions are marked as *EVENT* in the audit log and you can see, which event triggered these actions. Thus a single API call can cause several audit log entries: One for the API call and more for the triggered actions.

1.8.1 Events

Each **API call** is an **event** and you can bind arbitrary actions to each event as you like.

Internally events are marked by a decorator “event” with an *event identifier*. At the moment not all events might be tagged. Please drop us a note to tag all further API calls.



Id	Events	Handlermodule	Condition	Action	Options	
5	["token_init"]	UserNotification		sendmail	{ "emailconfig": "themis" }	Delete

Fig. 58: *An action is bound to the event* token_init.

1.8.2 Pre and Post Handling

Added in Version 2.23.

With most event handlers you can decide if you want the action to be taken before the actual event or after the actual event. I.e. if all conditions would trigger certain actions the action is either triggered before (*pre*) the API request is processed or after (*post*) the request is processed.

Up to version 2.22 all actions where triggered after the request. In this case additional information from the response is available. E.g. if a user successfully authenticated the event will know the serial number of the token, which the user used to authenticate.

If the action is triggered before the API request is processed, the event can not know if the authentication request will be successful or which serial number a token would have. However, triggering the action *before* the API request is processed can have some interesting other advantages:

Example for Pre Handling

The administrator can define an event definition that would trigger on the event `validate/check` in case the the authenticating user does not have any token assigned.

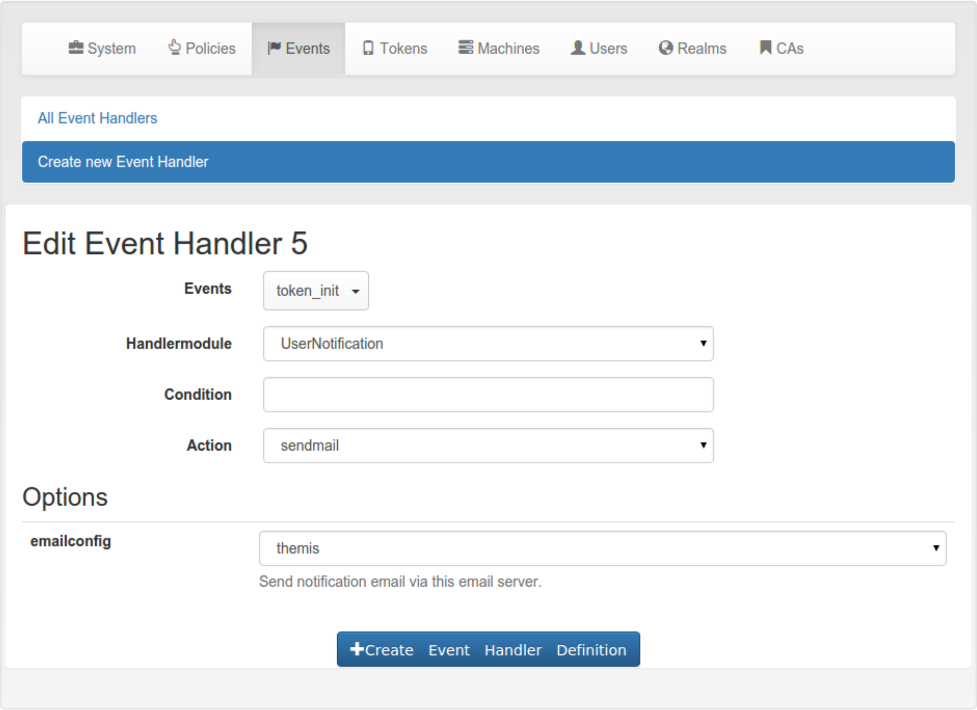
The *pre* event definition could call the Tokenhandler with the *enroll* action and enroll an email token with *dynamic_email* for this very user.

When the API request `validate/check` is now processed, the user actually now has an email token and can authenticate via challenge response with this very email token without an administrator ever enrolling or assigning a token for this user.

1.8.3 Handler Modules and Actions

The actions are defined in handler modules. So you bind a handler module and the action, defined in the handler module, to the events.

The handler module can define several actions and each action in the handler module can require additional options.



Edit Event Handler 5

Events: token_init

Handlermodule: UserNotification

Condition:

Action: sendmail

Options

emailconfig: themis

Send notification email via this email server.

Fig. 59: *The event* sendmail *requires the option* emailconfig.

1.8.4 Conditions

Added in version 2.14

An event handler module may also contain conditions. Only if all conditions are fulfilled, the action is triggered. Conditions are defined in the class property *conditions* and checked in the method *check_condition*. The base class for event handlers currently defines those conditions. So all event handlers come with the same conditions.

Note: In contrast to other conditions, the condition checking for `tokenrealms`, `tokenresolvers`, `serial` and `user_token_number` also evaluates to *true*, if this information can not be checked. I.e. if a request does not contain a *serial* or if the serial can not be determined, this condition will be evaluated as fulfilled.

Event Handlers are a mighty and complex tool to tweek the functioning of your privacyIDEA system. We recommend to test your definitions thoroughly to assure your expected outcome.

Basic conditions

The basic event handler module has the following conditions.

client_ip

The action is triggered if the client IP matches this value. The value can be a comma-separated list of single addresses or networks. To exclude entries, put a minus sign:

```
192.168.0.0/24,-192.168.0.12,10.0.0.2
```

count_auth

This can be '>100', '<99', or '=100', to trigger the action, if the tokeninfo field 'count_auth' is bigger than 100, less than 99 or exactly 100.

count_auth_fail

This can be '>100', '<99', or '=100', to trigger the action, if the difference between the tokeninfo field 'count_auth' and 'count_auth_success is bigger than 100, less than 99 or exactly 100.

count_auth_success

This can be '>100', '<99', or '=100', to trigger the action, if the tokeninfo field 'count_auth_success' is bigger than 100, less than 99 or exactly 100.

detail_error_message

This condition checks a regular expression against the `detail` section in the API response. The field `detail->error->message` is evaluated.

Error messages can be manyfold. In case of authentication you could get error messages like:

"The user can not be found in any resolver in this realm!"

With `token/init` you could get:

"missing Authorization header"

Note: The field `detail->error->message is only available in case of an internal error, i.e. if the response status is ``False`.

detail_message

This condition checks a regular expression against the `detail` section in the API response. The field `detail->message` is evaluated.

Those messages can be manyfold like:

"wrong otp pin"

"wrong otp value"

"Only 2 failed authentications per 1:00:00"

Note: The field `detail->message` is available in case of status `True`, like an authentication request that was handled successfully but failed.

detail_message

Here you can enter a regular expression. The condition only applies if the regular expression matches the detail->message in the response.

last_auth

This condition checks if the last authentication is older than the specified time delta. The timedelta is specified with "h" (hours), "d" (days) or "y" (years). Specifying *180d* would mean, that the action is triggered if the last successful authentication with the token was performed more than 180 days ago.

This can be used to send notifications to users or administrators to inform them, that there is a token, that might be orphaned.

logged_in_user

This condition checks if the logged in user is either an administrator or a normal user. This way the administrator can bind actions to events triggered by normal users or e.g. by help desk users. If a help desk user enrolls a token for a user, the user might get notified.

If a normal user enrolls some kind of token, the administrator might get notified.

otp_counter

The action is triggered, if the otp counter of a token has reached the given value. The value can either be an exact match or greater ('>100') or less ('<200') then a specified limit.

The administrator can use this condition to e.g. automatically enroll a new paper token for the user or notify the user that nearly all OTP values of a paper token have been spent.

realm

The condition *realm* matches the user realm. The action will only trigger, if the user in this event is located in the given realm.

This way the administrator can bind certain actions to specific realms. E.g. some actions will only be triggered, if the event happens for normal users, but not for users in admin- or helpdesk realms.

resolver

The resolver of the user, for which this event should apply.

result_status

The result.status within the response is True or False.

result_value

This condition checks the result of an event.

E.g. the result of the event *validate_check* can be a failed authentication. This can be the trigger to notify either the token owner or the administrator.

rollout_state

This is the rollout_state of a token. A token can be rolled out in several steps like the 2step HOTP/TOTP token. In this case the attribute "rollout_state" of the token contains certain values like 'clientwait' or 'enrolled'. This way actions can be triggered, depending on the step during an enrollment process.

serial

The action will only be triggered, if the serial number of the token in the event does match the regular expression.

This is a good idea to combine with other conditions. E.g. only tokens with a certain kind of serial number like Google Authenticator will be deleted automatically.

token_has_owner

The action is only triggered, if the token is or is not assigned to a user.

token_is_orphaned

The action is only triggered, if the user, to whom the token is assigned, does not exist anymore.

token_locked

The action is only triggered, if the token in the event is locked, i.e. the maximum failcounter is reached. In such a case the user can not use the token to authenticate anymore. So an action to notify the user or enroll a new token can be triggered.

token_validity_period

Checks if the token is in the current validity period or not. Can be set to *True* or *False*.

Note: `token_validity_period==False` will trigger an action if either the validitiy period is either *over* or has not *started*, yet.

tokeninfo

The tokeninfo condition can compare any arbitrary tokeninfo field against a fixed value. You can compare strings and integers. Integers are converted automatically. Valid compares are:

> myValue == 1000 myValue > 1000 myValue < 99 myTokenInfoField == EnrollmentState myTokenInfoField < ABC myTokenInfoField > abc

"myValue" and "myTokenInfoField" being any possible tokeninfo fields.

Starting with version 2.20 you can also compare dates in the isoformat like that:

> myValue > 2017-10-12T10:00+0200 myValue < 2020-01-01T00:00+0000

In addition you can also use the tag *{now}* to compare to the curren time *and* you can add offsets to *{now}* in seconds, minutes, hours or days:

> myValue < {now} myValue > {now}+10d myValue < {now}-5h

Which would match if the tokeninfo *myValue* is a date, which is later than 10 days from now or it the tokeninfo *myValue* is a date, which is 5 more than 5 hours in the past.

tokenrealm

In contrast to the *realm* this is the realm of the token - the *tokenrealm*. The action is only triggered, if the token within the event has the given tokenrealm. This can be used in workflows, when e.g. hardware tokens which are not assigned to a user are pushed into a kind of storage realm.

tokenresolver

The resolver of the token, for which this event should apply.

tokentype

The action is only triggered if the token in this event is of the given type. This way the administrator can design workflows for enrolling and reenrolling tokens. E.g. the tokentype can be a registration token and the registration code can be easily and automatically sent to the user.

user_token_number

The action is only triggered, if the user in the event has the given number of tokens assigned.

This can be used to e.g. automatically enroll a token for the user if the user has no tokens left (token_number == 0) of to notify the administrator if the user has to many tokens assigned.

1.8.5 Managing Events

Using the command `pi-manage events` you can list, delete, enable and disable events. You can also export the complete event definitions to a file or import the event definitions from a file again. During import you can specify if you want to remove all existing events or if you want to add the events from the file to the existing events in the database.

Note: Events are identified by an *id*! Due to database restrictions the id is ignored during import. So importing an event with the same name will create a second event with the same name but another id.

1.8.6 Available Handler Modules

User Notification Handler Module

The user notification handler module is used to send emails token owners or administrators in case of any event.

Possible Actions

sendmail

The *sendmail* action sends an email to the specified email address each time the event handler is triggered.

emailconfig

- *required* Option

The email is sent via this *SMTP server configuration*.

To

- *required* Option

This specifies to which type of user the notification should be sent. Possible recipient types are:

- token owner,
- logged in user,
- admin realm,
- internal admin,
- email address.

Depending on the recipient type you can enter additional information. The recipient type *email* takes a comma separated list of email addresses.

reply_to

Adds the specified `Reply-To` header to the email.

subject

The subject can take the same tags as the body, except for the `{googleurl_img}`.

mimetype

Possible mime types are:

- plain (default)
- html

You can choose if the email should be sent as plain text or HTML. If the email is sent as HTML, you can do the following:

```
<a href={googleurl_value}>Your new token</a>
```

Which will create a clickable link. Clicked on the smartphone, the token will be imported to the smartphone app.

You can also do this:

```
<img src={googleurl_img}>
```

This will add the QR Code as an inline data image into the HTML email.

> **Warning:** The KEY URI and the QR Code contain the secret OTP key in plain text. Everyone who receives this data has a detailed copy of this token. Thus we very much recommend to **never** send these data in an unencrypted email!

attach_qrcode

Instead of sending the QR-Code as an inline data image (which is not supported by some email clients (i.e. Outlook) or GMail[1]), enabling this option sends the email as a multipart message with the QR-Code image as an attachment. The attached image can be referenced in a HTML body via CID URL[2] with the *Content-ID* `token_image`:

```
<img src="cid:token_image" alt="Token Image" style="..."/>
```

sendsms

The *sendsms* action sends an SMS to the specified number each time the event handler is triggered.

To

- *required* Option

Possible recipients are:

- tokenowner

smsconfig

- *required* Option

The *SMS Gateway configuration* for sending the notification.

[1] https://stackoverflow.com/a/42014708/7036742

[2] https://tools.ietf.org/html/rfc2392

savefile

The *savefile* action saves a file to a spool directory. Each time the event handler is triggered a new file is saved.

In the `pi.cfg` file you can use the setting `PI_NOTIFICATION_HANDLER_SPOOLDIRECTORY` to configure a spool directory, where the notification files will be written. The default file location is `/var/lib/privacyidea/notifications/`. The directory needs to be writable for the user *privacyidea*.

filename

- *required* option
- The filename of the saved file. It can contain the tag `{random}` which will create a 16 characters long alpha numeric string. Thus you could have a filename like `notification-{random}.csv`.

In addition you can use all tags that can be used in the body also in the filename (some of them might not make a lot of sense!).

Note: Existing files are overwritten.

Body for all actions

All actions take the common option *body*:

body

- optional for *sendmail* and *sendsms*
- required for *savefile*

Here the administrator can specify the body of the notification, that is sent or saved. The body may contain the following tags

- {admin} name of the logged in user.
- {realm} realm of the logged in user.
- {action} the action that the logged in user performed.
- {serial} the serial number of the token.
- {url} the URL of the privacyIDEA system.
- {user} the given name of the token owner.
- {givenname} the given name of the token owner.
- {surname} the surname of the token owner.
- {username} the loginname of the token owner.
- {userrealm} the realm of the token owner.
- {tokentype} the type of the token.
- {registrationcode} the registration code in the detail response.
- {recipient_givenname} the given name of the recipient.
- {recipient_surname} the surname of the recipient.
- {googleurl_value} is the KEY URI for a google authenticator.
- {googleurl_img} is the data image source of the google authenticator QR code.

- {time} the current server time in the format HH:MM:SS.
- {date} the current server date in the format YYYY-MM-DD
- {client_ip} the client IP of the client, which issued the original request.
- {ua_browser} the user agent of the client, which issued the original request.
- {ua_string} the complete user agent string (including version number), which issued the original request.
- {pin} the PIN of the token when set with `/token/setrandompin`. You can remove the PIN from the response using the *response mangler*.

Code

This is the event handler module for user notifications. It can be bound to each event and can perform the action:

- sendmail: Send an email to the user/token owner
- sendsms: We can also notify the user with an SMS.
- savefile: Create a file which can be processed later

The module is tested in tests/test_lib_eventhandler_usernotification.py

class `privacyidea.lib.eventhandler.usernotification.`**`NOTIFY_TYPE`**

Allowed token owner

`ADMIN_REALM = 'admin realm'`

`EMAIL = 'email'`

`INTERNAL_ADMIN = 'internal admin'`

`LOGGED_IN_USER = 'logged_in_user'`

`TOKENOWNER = 'tokenowner'`

class `privacyidea.lib.eventhandler.usernotification.`**`UserNotificationEventHandler`**

An Eventhandler needs to return a list of actions, which it can handle.

It also returns a list of allowed action and conditions

It returns an identifier, which can be used in the eventhandlig definitions

`property actions`

This method returns a dictionary of allowed actions and possible options in this handler module.

Returns dict with actions

`property allowed_positions`

This returns the allowed positions of the event handler definition. :return: list of allowed positions

`description = 'This eventhandler notifies the user about actions on his tokens'`

`do`(*action*, *options=None*)

This method executes the defined action in the given event.

Parameters

- **`action`** –
- **`options`** (*`dict`*) – Contains the flask parameters g, request, response and the handler_def configuration

Returns

```
identifier = 'UserNotification'
```

Token Handler Module

The token event handler module is used to perform actions on tokens in certain events.

This way you can define workflows to automatically modify tokens, delete or even create new tokens.

Possible Actions

set tokenrealm

Here you can set the token realms of the token.

E.g. You could use this action to automatically put all newly enrolled tokens into a special realm by attaching this action to the event *token_init*.

delete

The token which was identified in the request will be deleted if all conditions are matched.

unassign

The token which was identified in the request will be unassign from the user if all conditions are matched.

disable

The token which was identified in the request will be disabled if all conditions are matched.

enable

The token which was identified in the request will be enabled if all conditions are matched.

enroll

If all conditions are matched a new token will be enrolled. This new token can be assigned to a user, which was identified in the request.

The administrator can specify the **tokentype** and the **realms** of the new token. By default the generation of the token will use the parameter `genkey`, to generate the otp key. (see *Token endpoints*).

The action `enroll` also can take the options **dynamic_phone** (in case of tokentype SMS) and **dynamic_email** (in case of tokentype email). Then these tokens are created with a dynamic loadable phone number or email address, that is read from the user store on each authentication request.

Finally the administrator can specify the option **additional_params**. This needs to be a dictionary with parameters, that get passed to the init request. You can specify all parameters, that would be used in a `/token/init` request:

> {“hashlib”: “sha256”, “type”: “totp”, “genkey”: 0, “otpkey”: “31323334”}

would create a TOTP token, that uses the SHA256 hashing algorithm instead of SHA1. `genkey: 0` overrides the default behaviour of generating an OTP secret. Instead the fixed OTP secret “31323334” (`otpkey`) is used.

set description

If all conditions are matched the description of the token identified in the request will be set.

You can use the tag `{current_time}` or `{now}` to set the current timestamp. In addition you can append an offset to *current_time* or *now* like `{now}-12d` or `{now}+10m`. This would write a timestamp which is 12 days in the passt or 10 minutes in the future. The plus or minus must follow without blank, allowed time identifiers are s (seconds), m (minutes), h (hours) and d (days).

Other tags are `{client_ip}` for the client IP address and `{ua_browser}` and `{ua_string}` for information on the user agent.

set validity

If all conditions are matched the validity period of the token will be set.

There are different possibilities to set the start and the end of the validity period. The event definition can either contain a fixed date and time or if can contain a time offset.

Fixed Time

A fixed time can be specified in the following formats.

Only date without time:

- 2016/12/23
- 23.12.2016

Date with time:

- 2016/12/23 9:30am
- 2016/12/23 11:20:pm
- 23.12.2016 9:30
- 23.12.2016 23:20

Starting with version 2.19 we recommend setting the fixed time in the ISO 8601 corresponding time format

- 2016-12-23T15:30+0600

Time Offset

You can also specify a time offset. In this case the validity period will be set such many days after the event occurred. This is indicated by using a “+” and a specifier for days (d), hours (h) and minutes (m).

E.g. `+30m` will set to start the validity period in 30 minutes after the event occurred.

`+30d` could set the validity period to end 30 days after an event occurred.

Note: This way you could easily define a event definition, which will set newly enrolled tokens to be only valid for a certain amount of days.

set countwindow

Here the count window of a token can be set. This requires an integer value.

set tokeninfo

Using the action `set tokeninfo` you can set any arbitrary tokeninfo attribute for the token. You need to specify the `key` of the tokeninfo and the `value`.

In the value field you can use the tag `{current_time}` to set the current timestamp. In addition you can append an offset to *current_time* or *now* like `{now}-12d` or `{now}+10m`. This would write a timestamp which is 12 days in the passt or 10 minutes in the future. The plus or minus must follow without blank, allowed time identifiers are s (seconds), m (minutes), h (hours) and d (days).

Other tags are `{client_ip}` for the client IP address and `{ua_browser}` and `{ua_string}` for information on the user agent and `{username}` and `{realm}` for information on the user in the parameters.

Note: Some tokens have token specific attributes that are stored in the tokeninfo. The TOTP token type has a `timeWindow`. The TOTP and the HOTP token store the `hashlib` in the tokeninfo, the SMS token stores the `phone` number.

Note: You can use this to set the `timeWindow` of a TOTP token for *Automatic initial synchronization*.

set failcounter

Using the action `set failcounter` you can reset the fail counter by setting it to 0 or also "block" the token by setting the fail counter to what ever value the "max_fail" is, e.g. 10. Only integer values are allowed.

set random pin

Sets a random PIN for the handled token. The PIN is then added to the response in `detail->pin`. This can be used in the *notification handler*. Please take care, that probably the PIN needs to be removed from the response using the *response mangler handler* after handling it with the notification handler.

Code

This is the event handler module for token actions. You can attach token actions like enable, disable, delete, unassign,… of the

- current token
- all the user's tokens
- all unassigned tokens
- all disabled tokens
- …

class `privacyidea.lib.eventhandler.tokenhandler.`**`ACTION_TYPE`**

Allowed actions

DELETE = 'delete'

DELETE_TOKENINFO = 'delete tokeninfo'

DISABLE = 'disable'

ENABLE = 'enable'

INIT = 'enroll'

SET_COUNTWINDOW = 'set countwindow'

SET_DESCRIPTION = 'set description'

SET_FAILCOUNTER = 'set failcounter'

SET_RANDOM_PIN = 'set random pin'

SET_TOKENINFO = 'set tokeninfo'

SET_TOKENREALM = 'set tokenrealm'

SET_VALIDITY = 'set validity'

UNASSIGN = 'unassign'

class privacyidea.lib.eventhandler.tokenhandler.**TokenEventHandler**

An Eventhandler needs to return a list of actions, which it can handle.

It also returns a list of allowed action and conditions

It returns an identifier, which can be used in the eventhandlig definitions

property actions

This method returns a dictionary of allowed actions and possible options in this handler module.

Returns dict with actions

property allowed_positions

This returns the allowed positions of the event handler definition. :return: list of allowed positions

description = 'This event handler can trigger new actions on tokens.'

do (*action*, *options=None*)

This method executes the defined action in the given event.

Parameters

- **action** –
- **options** (*dict*) – Contains the flask parameters g, request, response and the handler_def configuration

Returns

identifier = 'Token'

class privacyidea.lib.eventhandler.tokenhandler.**VALIDITY**

Allowed validity options

END = 'valid till'

START = 'valid from'

Script Handler Module

The script event handler module is used to trigger external scripts in case of certain events.

This way you can even add external actions to your workflows. You could trigger a database dump, an external printing device, a backup and much more.

Possible Actions

The actions of the script event handler are the scripts located in a certain script directory. The default script directory is `/etc/privacyidea/scripts`.

You can change the location of the script directory and give the new directory in the parameter `PI_SCRIPT_HANDLER_DIRECTORY` in your `pi.cfg`

> file.

Possible Options

Options can be passed to the script. Your script has to take care of the parsing of these parameters.

logged_in_role

Add the role of the logged in user. This can be either *admin* or *user*. If there is no logged in user, *none* will be passed.

The script will be called with the parameter

> –logged_in_role <role>

logged_in_user

Add the logged in user. If there is no logged in user, *none* will be passed.

The script will be called with the parameter

> –logged_in_user <username>@<realm>

realm

Add `--realm <realm>` as script parameter. If no realm is given, *none* will be passed.

serial

Add `--serial <serial number>` as script parameter. If no serial number is given, *none* will be passed.

user

Add `--serial <username>'` as script parameter. If no username is given, *none* will be passed.

Note: A possible script you could call is the *privacyidea-get-unused-tokens*.

Counter Handler Module

The counter event handler module is used to count certain events. You can define arbitrary counter names and each occurrence of an event will modify the counter in the counter table according to the selected action.

These counters can be used to graph time series of failed authentication, assigned tokens, user numbers or any other data with any condition over time.

Possible Actions

increase_counter

This action increases the counter in the database table `eventcounter`. If the counter does not exists, it will be created and increased.

decrease_counter

This action decreases the counter in the database table `eventcounter`. If the counter does not exists, it will be created and decreased.

> **Note:** This action will not decrease the counter beyond zero unless the option *allow_negative_values* is enabled.

reset_counter

This action resets the counter in the database table `eventcounter` to `zero`.

Possible Options

counter_name

This is the name of the counter in the database. You can have as many counters in as many event handlers as you like.

allow_negative_values

Only available for the `decrease_counter` action. Allows the counter to become negative. If set to `False` (default) decreasing stops at `zero`. .. note:: Since the option `allow_negative_values` is an attribute of the counter event handler action (and not the counter itself in the database) it is possible to define multiple event handler accessing the same counter. Thus if a negative counter is accessed by an event handler with the option `allow_negative_values` set to true, the counter will be reset to `zero`

Federation Handler Module

The federation event handler can be used to configure relations between several privacyIDEA instances. Requests can be forwarded to child privacyIDEA instances.

Note: The federation event handler can modify the original response. If the response was modified a new field `origin` will be added to the `detail` section in the response. The *origin* will contain the URL of the privacyIDEA server that finally handled the request.

Possible Actions

forward

A request (usually an authentication request *validate_check*) can be forwarded to another privacyIDEA instance. The administrator can define privacyIDEA instances centrally at *conifg* -> *privacyIDEA servers*.

In addition to the privacyIDEA instance the action `forward` takes the following parameters:

client_ip The originial client IP will be passed to the child privacyIDEA server. Otherwise the child privacyIDEA server will use the parent privacyIDEA server as client.

Note: You need to configure the allow override client in the child privacyIDEA server.

realm The forwarding request will change the realm to the specified realm . This might be necessary since the child privacyIDEA server could have

different realms than the parent privacyIDEA server.

resolver The forwarding request will change the resolver to the specified resolver. This might be necessary since the child privacyIDEA server could have different resolvers than the parent privacyIDEA server.

One simple possibility would be, that a user has a token in the parent privacyIDEA server and in the child privacyIDEA server. Configuring a forward event handler on the parent with the condition `result_value = False` would have the effect, that the user can either authenticate with the parent's token or with the child's token on the parent privacyIDEA server.

Federation can be used, if privacyIDEA was introduced in a subdivision of a larger company. When privacyIDEA should be enrolled to the complete company you can use federation. Instead of dropping the privacyIDEA instance in the subdivision and installing on single central privacyIDEA, the subdivision can still go on using the original privacyIDEA system (child) and the company will install a new top level privacyIDEA system (parent).

Using the federation handler you can setup many other, different scenarios we can not think of, yet.

Code

This is the event handler module for privacyIDEA federations. Requests can be forwarded to other privacyIDEA servers.

class privacyidea.lib.eventhandler.federationhandler.**ACTION_TYPE**

Allowed actions

FORWARD = 'forward'

class privacyidea.lib.eventhandler.federationhandler.**FederationEventHandler**

An Eventhandler needs to return a list of actions, which it can handle.

It also returns a list of allowed action and conditions

It returns an identifier, which can be used in the eventhandlig definitions

property actions

This method returns a dictionary of allowed actions and possible options in this handler module.

Returns dict with actions

description = 'This event handler can forward the request to other privacyIDEA servers

do (*action*, *options=None*)

This method executes the defined action in the given event.

Parameters

- **action** –
- **options** (*dict*) – Contains the flask parameters g, request, response and the handler_def configuration

Returns

identifier = 'Federation'

RequestMangler Handler Module

The RequestMangler is a special handler module, that can modify the request parameters of an HTTP request. This way privacyIDEA can change the data that is processed within the request.

Usually this handler is used in the **pre** location. However there might be occasions when you want to modify parameters only *before* passing them to the next **post** handler. In this case you can also use the RequestMangler handler in the **post** location.

Possible Actions

delete

This action simply deletes the given parameter from the request.

E.g. you could in certain cases delete the `transaction_id` from a `/validate/check` request. This way you would render challenge response inactive.

set

This action is used to add or modify additional request parameters.

You can set a parameter with the value or substrings of another parameter.

This is why this action takes the additional options *value*, *match_parameter* and *match_pattern*. *match_pattern* always needs to match the *complete* value of the *match_parameter*.

If you simply want to set a parameter to a fixed value you only need the options:

- *parameter*: as the name of the parameter you want to set and
- *value*: to set to a fixed value.

If you can to set a parameter based on the value of another parameter, you can use the regex notation **()** and the python string formatting tags **{0}**, **{1}**.

Example 1

To set the realm based on the user parameter:

```
parameter: realm
match_parameter: user
match_pattern: .*@(.*)
value: {0}
```

A request like:

```
user=surname.givenname@example.com
realm=
```

with an empty realm will be modified to:

```
user=surname.givenname@example.com
realm=example.com
```

since, the pattern `.*@(.*)` will match the email address and extract the domain after the "@" sign. The python tag "{0}" will be replaced with the matching domainname.

Example 2

To simply change the domain name in the very same parameter:

```
parameter: user
match_parameter: user
match_pattern: (.*)@example.com
value: {0}@newcompany.com
```

A request like:

```
user=surname.givenname@example.com
```

will be modified to:

```
user=surname.givenname@newcompany.com
```

Note: The *match_pattern* in the above example will not match "surname.givenname@example.company", since it always matches the complete value as mentioned above.

Code

This is the event handler module modifying request parameters.

class privacyidea.lib.eventhandler.requestmangler.**ACTION_TYPE**

Allowed actions

DELETE = 'delete'

SET = 'set'

class privacyidea.lib.eventhandler.requestmangler.**RequestManglerEventHandler**

An Eventhandler needs to return a list of actions, which it can handle.

It also returns a list of allowed action and conditions

It returns an identifier, which can be used in the eventhandlig definitions

property actions

This method returns a dictionary of allowed actions and possible options in this handler module.

Returns dict with actions

property allowed_positions

This returns the allowed positions of the event handler definition. :return: list of allowed positions

description = 'This event handler can modify the parameters in the request.'

do (*action*, *options=None*)

This method executes the defined action in the given event.

Parameters

- **action** –
- **options** (*dict*) – Contains the flask parameters g, request, response and the handler_def configuration

Returns

identifier = 'RequestMangler'

ResponseMangler Handler Module

The ResponseMangler is a special handler module, that can modify the response of an HTTP request. This way privacyIDEA can change the data sent back to the client, depending on certain conditions.

All actions take a JSON pointer, which looks like a path variable like `/result/value`.

Possible Actions

delete

This action simply deletes the given JSON pointer from the response.

Note: All keys underneath a node are deleted, to. So if the event handler deletes `/detail`, the entries `/detail/message` and ``/detail/error``will also be deleted.

Example

You can use this to delete `/detail/googleurl`, `/detail/oathurl` and `/detail/otpkey` in a `/token/init` event to hide the created QR code from the helpdesk admin. This way the QR code could be used internally, but could be hidden from the administrator.

set

This action is used to add additional pointers to the JSON response or to modify existing entries. Existing entries are overwritten.

This action takes the additional attributes `type` and `value`.

The value can be returned as a string, an integer or a boolean.

Code

This is the event handler module that can mangle the JSON response. We can add or delete key or even subtrees in the JSON response of a request.

The key is identified by a JSON Pointer (see https://tools.ietf.org/html/rfc6901)

class `privacyidea.lib.eventhandler.responsemangler.`**`ACTION_TYPE`**

Allowed actions

`DELETE = 'delete'`

`SET = 'set'`

class `privacyidea.lib.eventhandler.responsemangler.`**`ResponseManglerEventHandler`**

An Eventhandler needs to return a list of actions, which it can handle.

It also returns a list of allowed action and conditions

It returns an identifier, which can be used in the eventhandlig definitions

`property actions`

This method returns a dictionary of allowed actions and possible options in this handler module.

Returns dict with actions

`property allowed_positions`

This returns the allowed positions of the event handler definition. The ResponseMangler can only be located at the "post" position

Returns list of allowed positions

`description = 'This event handler can mangle the JSON response.'`

`do`(*action*, *options=None*)

This method executes the defined action in the given event.

Parameters

- **`action`** –
- **`options`** (*`dict`*) – Contains the flask parameters g, request, response and the handler_def configuration

Returns

`identifier = 'ResponseMangler'`

Logging Handler Module

The logging event handler can be used to log the occurrence of an event to the python logging facility. You can log arbitrary events with a configurable log message, loglevel and logger instance. Several tags are available to customize the log message.

The configuration to handle the log messages can be defined in detail with the *Advanced Logging*.

Possible Actions

logging

Emit a log message to the python logging facility when the specified event gets triggered (and the conditions match).

name

- *default:* `pi-eventlogger`

The name of the logger to use when emitting the log message. This can be used for a fine-grained control of the log messages via *Advanced Logging*.

Note: Logger names beginning with `privacyidea` will be handled by the default privacyIDEA logger and will end up in the privacyIDEA log.

level

- *default:* `INFO`

The log level for the emitted log message. The following levels are available:

- `ERROR`
- `WARNING`
- `INFO`
- `DEBUG`

message

- *default:* `"event={action} triggered"`

The message to send to the logging facility. This message can be customized with the following tags:

- **{admin}** The logged in user.
- **{realm}** The realm of the logged in user.
- **{action}** The action which triggered this event.
- **{serial}** The serial of a token used in this event.
- **{url}** The URL of the privacyIDEA system.
- **{user}** The given name of the token owner.
- **{surname}** The surname of the token owner.
- **{givenname}** The given name of the token owner.
- **{username}** The login of the token owner.
- **{userrealm}** The realm of the token owner.

- **{tokentype}** The type of the token.
- **{time}** The current server time (format: HH:MM:SS).
- **{date}** The current server date (format: YYYY-MM-DD).
- **{client_ip}** The IP of the client who triggered the event.
- **{ua_browser}** The user agent of the client, which issued the original request.
- **{ua_string}** The complete user agent string (including version number) which issued the original request.

Note: Not all tags are available in every event. It depends on the called API-Endpoint and passed parameter which tags exist. If a tag does not exist during the event handling, an empty string will be inserted.

1.9 Periodic Tasks

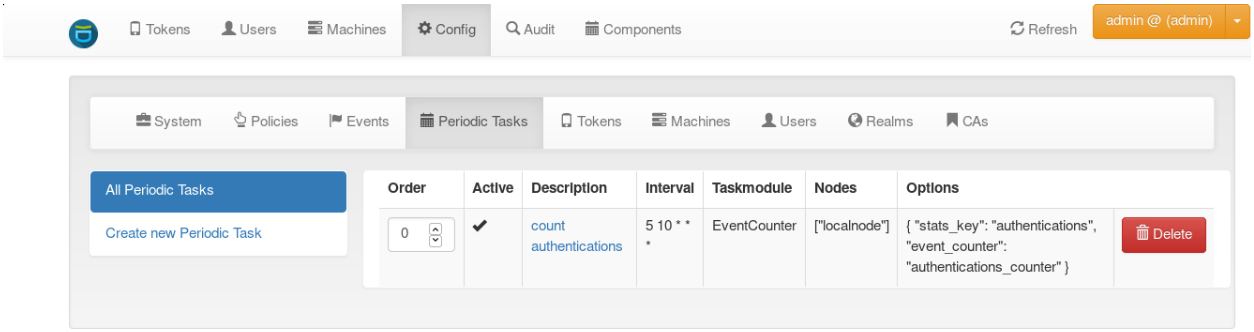
Starting with version 2.23, privacyIDEA comes with the ability to define periodically recurring tasks in the Web UI. The purpose of such tasks is to periodically execute certain processes automatically. The administrator defines which tasks should be executed using task modules. Currently there are task modules for simple statistics and for handling recorded events. Further task modules can be added easily.

As privacyIDEA is a web application, it can not actually execute the defined periodic tasks itself. For that, privacyIDEA comes with a script `privacyidea-cron` which must be invoked by the system cron daemon. This can, for example, be achieved by creating a file `/etc/cron.d/privacyidea` with the following contents (this is done automatically by the Ubuntu package):

```
*/5 * * * *     privacyidea     privacyidea-cron run_scheduled -c
```

This tells the system cron daemon to invoke the `privacyidea-cron` script every five minutes. At each invocation, the `privacyidea-cron` script determines which tasks should be executed and execute the scheduled tasks. The `-c` option tells the script to be quiet and only print to stderr in case of an error (see *The privacyidea-cron script*).

Periodic tasks can be managed in the WebUI by navigating to *Config->Periodic Tasks*:



Order	Active	Description	Interval	Taskmodule	Nodes	Options
0	✓	count authentications	5 10 * * *	EventCounter	["localnode"]	{ "stats_key": "authentications", "event_counter": "authentications_counter" }

Fig. 60: Periodic task definitions

Every periodic task has the following attributes:

description A human-readable, unique identifier

active A boolean flag determining whether the periodic task should be run or not.

order A number (at least zero) that can be used to rearrange the order of periodic tasks. This is used by `privacyidea-cron` to determine the running order of tasks if multiple periodic tasks are scheduled to be run. Tasks with a lower number are run first.

interval The periodicity of the task. This uses crontab notation, e.g. `*/30 * * * *` runs the task every 30 minutes.

Keep in mind that the entry in the system crontab determines the minimal resolution of periodic tasks: If you specify a periodic task that should be run every two minutes, but the `privacyidea-cron` script is invoked every five minutes only, the periodic task will actually be executed every five minutes!

nodes The names of the privacyIDEA nodes on which the periodic task should be executed. This is useful in a redundant master-master setup, because database-related tasks should then only be run on *one* of the nodes (because the replication will take care of propagating the database changes to the other node). The name of the local node as well as the names of remote nodes are configured in *The Config File*.

taskmodule The task module determines the actual activity of the task. privacyIDEA comes with several task modules, see *Task Modules*.

options The options are a set of key-value pairs that configure the behavior of the task module. Each task module can have it's own allowed options.

1.9.1 Task Modules

privacyIDEA comes with the following task modules:

SimpleStats

The `SimpleStats` task module is a *Periodic Tasks* to collect some basic statistics from the token database and write them to the time series database table `MonitoringStats`.

Options

The `SimpleStats` task module provides the following boolean options:

total_tokens

If activated, the total number of tokens in the token database will be monitored.

hardware_tokens

If activated, the total number of hardware tokens in the token database will be monitored.

software_tokens

If activated, the total number of software tokens in the token database will be monitored.

unassigned_hardware_tokens

If activated, the number of hardware tokens in the token database which are not assigned to a user will be monitored.

assigned_tokens

If activated, the number of tokens in the token database which are assigned to users will be monitored.

user_with_token

If activated, the number of users which have at least one token assigned will be monitored.

Note: The statistics key, with which the time series is identified in the `MonitoringStats` table, is the same as the option name.

Using a statistic with the same key in a different module, which writes to the `MonitoringStats` table, will corrupt the data.

Note: For each of these basic statistic values the token database will be queried. To avoid excessive load on the database, the `SimpleStats` task should not be executed too often.

EventCounter

The Event Counter task module can be used with the *Periodic Tasks* to create time series of certain events. An event could be a failed authentication request. Using the Event Counter, privacyIDEA can create graphs that display the development of failed authentication requests over time.

To do this, the Event Counter task module reads a counter value from the database table `EventCounter` and adds this current value in a time series in the database table `MonitoringStats`. As the administrator can use the event handler *Counter Handler Module* to record any arbitrary event under any condition, this task module can be used to graph any metrics in privacyIDEA, be it failed authentication requests per time unit, the number of token delete requests or the number of PIN resets per month.

Options

The Event Counter task module provides the following options:

event_counter

This is the name of the event counter key, that was defined in a *Counter Handler Module* definition and that is read from the database table `EventCounter`.

stats_key

This is the name of the statistics key that is written to the `MonitoringStats` database table. The event counter key stores the current number of counted events, the `stats_key` takes the current number and stores it with the timestamp as a time series.

reset_event_counter

This is a boolean value. If it is set to true (the checkbox is checked), then the event counter will be reset to zero, after the task module has read the key.

Resetting the the event counter results in a time series of "events per time interval". The time intervall is specified by the time intervall in which the Event Counter task module is called. If `reset_event_counter` is not checked, then the event handler will continue to increase the counter value. Use this, if you want to create a time series, that displays the absolute number of events.

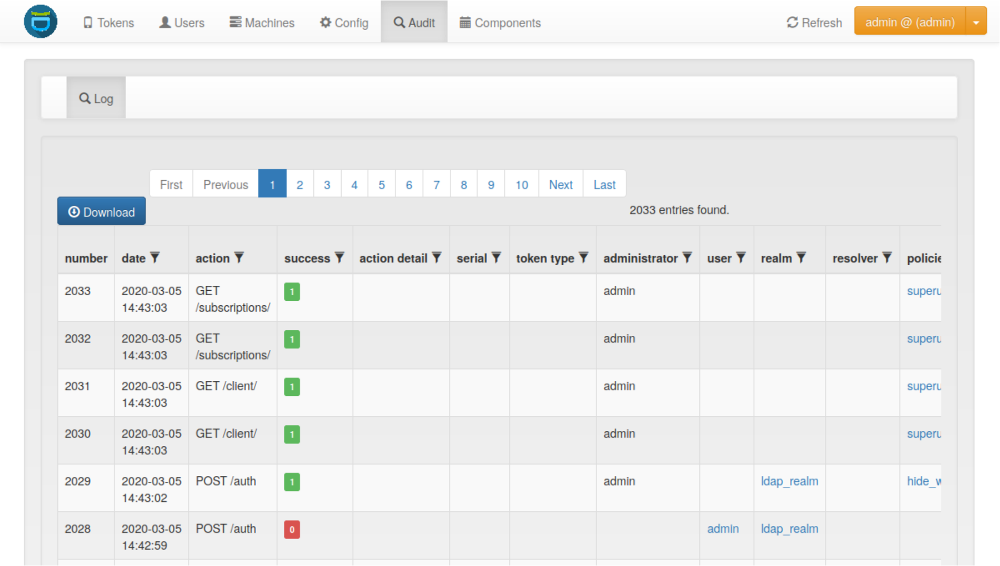
1.9.2 The `privacyidea-cron` script

The `privacyidea-cron` script is used to execute periodic tasks defined in the Web UI. The `run_scheduled` command collects all active jobs that are scheduled to run on the current node and executes them. The order is determined by their `ordering` values (tasks with low values are executed first). The `-c` option causes the script to is useful if the script is executed via the system crontab, as it causes the script to only print to stderr in case of errors.

The `list` command can be used to get an overview of defined jobs, and the `run_manually` command can be used to manually invoke tasks even though they are not scheduled to be run.

1.10 Audit

The systems provides a sophisticated audit log, that can be viewed in the WebUI.



2033 entries found.

number	date	action	success	action detail	serial	token type	administrator	user	realm	resolver	policie
2033	2020-03-05 14:43:03	GET /subscriptions/	1				admin				superu
2032	2020-03-05 14:43:03	GET /subscriptions/	1				admin				superu
2031	2020-03-05 14:43:03	GET /client/	1				admin				superu
2030	2020-03-05 14:43:03	GET /client/	1				admin				superu
2029	2020-03-05 14:43:02	POST /auth	1				admin		ldap_realm		hide_w
2028	2020-03-05 14:42:59	POST /auth	0					admin	ldap_realm		

Fig. 61: *Audit Log*

privacyIDEA comes with a default SQL audit module (see *Audit log*).

Starting with version 3.2 privacyIDEA also provides a *Logger Audit* and a *Container Audit* which can be used to send privacyIDEA audit log messages to services like splunk or logstash.

1.10.1 SQL Audit

Cleaning up entries

The `sqlaudit` module writes audit entries to an SQL database. For performance reasons the audit module does no log rotation during the logging process.

But you can set up a cron job to clean up old audit entries. Since version 2.19 audit entries can be either cleaned up based on the number of entries or based on on the age.

Cleaning based on the age takes precedence:

You can specify a *highwatermark* and a *lowwatermark*. To clean up the audit log table, you can call `pi-manage` at command line:

```
pi-manage rotate_audit --highwatermark 20000 --lowwatermark 18000
```

This will, if there are more than 20.000 log entries, clean all old log entries, so that only 18000 log entries remain.

Cleaning based on the age:

You can specify the number of days, how old an audit entry may be at a max.

> pi-manage rotate_audit –age 365

will delete all audit entries that are older than one year.

Cleaning based on the config file:

Using a config file you can define different retention times for the audit data. E.g. this way you can define, that audit entries about token listings can be deleted after one month, while the audit information about token creation will only deleted after ten years.

The config file is a YAML format and looks like this:

```
# DELETE auth requests of nils after 10 days
- rotate: 10
  user: nils
  action: .*/validate/check.*

# DELETE auth requests of friedrich after 7 days
- rotate: 7
  user: friedrich
  action: .*/validate/check.*

# Delete nagios user test auth directly
- rotate: 0
  user: nagiosuser
  action: POST /validate/check.*

# Delete token listing after one month
- rotate: 30
  action: ^GET /token

# Delete audit logs for token creating after 10 years
- rotate: 3650
  action: POST /token/init

# Delete everything else after 6 months
- rotate: 180
  action: .*
```

This is a list of rules. privacyIDEA iterates over *all* audit entries. The first matching rule for an entry wins. If the rule matches, the audit entry is deleted if the entry is older than the days specified in "rotate".

If is a good idea to have a *catch-all* rule at the end.

Note: The keys "user", "action"... correspond to the column names of the audit table. You can use any column name here like "date", "action", "action_detail", "success", "serial", "administrator", "user", "realm"... for a complete list see the model definition. You may use Python regular expressions for matching.

You can the add a call like

> pi-manage rotate_audit –config /etc/privacyidea/audit.yaml

in your crontab.

Access rights

You may also want to run the cron job with reduced rights. I.e. a user who has no read access to the original pi.cfg file, since this job does not need read access to the SECRET or PEPPER in the pi.cfg file.

So you can simply specify a config file with only the content:

```
PI_AUDIT_SQL_URI = <your database uri>
```

Then you can call `pi-manage` like this:

```
PRIVACYIDEA_CONFIGFILE=/home/cornelius/src/privacyidea/audit.cfg \
pi-manage rotate_audit
```

This will read the configuration (only the database uri) from the config file `audit.cfg`.

Table size

Sometimes the entires to be written to the database may be longer than the column in the database. You can either enlarge the columns in the database or you can set

> PI_AUDIT_SQL_TRUNCATE = True

in `pi.cfg`. This will truncate each entry to the defined column length.

1.10.2 Logger Audit

The *Logger Audit* module can be used to write audit log information to the Python logging facility and thus write log messages to a plain file, a syslog daemon, an email address or any destination that is supported by the Python logging mechanism. The log message passed to the python logging facility is a JSON-encoded string of the fields of the audit entry.

You can find more information about this in *Advanced Logging*.

To activate the *Logger Audit* module you need to configure the following settings in your `pi.cfg` file:

```
PI_AUDIT_MODULE = "privacyidea.lib.auditmodules.loggeraudit"
PI_AUDIT_SERVERNAME = "your choice"
PI_LOGCONFIG = "/etc/privacyidea/logging.cfg"
```

You can optionally set a custom logging name for the logger audit with:

```
PI_AUDIT_LOGGER_QUALNAME = "pi-audit"
```

It defaults to the module name `privacyidea.lib.auditmodules.loggeraudit`. In contrast to the *SQL Audit* you *need* a `PI_LOGCONFIG` otherwise the *Logger Audit* will not work correctly.

In the `logging.cfg` you then need to define the audit logger:

```
[logger_audit]
handlers=audit
qualname=privacyidea.lib.auditmodules.loggeraudit
level=INFO

```

(continues on next page)

(continued from previous page)

```
[handler_audit]
class=logging.handlers.RotatingFileHandler
backupCount=14
maxBytes=10000000
formatter=detail
level=INFO
args=('/var/log/privacyidea/audit.log',)
```

Note, that the `level` always needs to be *INFO*. In this example the audit log will be written to the file `/var/log/privacyidea/audit.log`.

Finally you need to extend the following settings with the defined audit logger and audit handler:

```
[handlers]
keys=file,audit

[loggers]
keys=root,privacyidea,audit
```

Note: The *Logger Audit* only allows to **write** audit information. It can not be used to **read** data. So if you are only using the *Audit Logger*, you will not be able to *view* audit information in the privacyIDEA Web UI! To still be able to *read* audit information, take a look at the *Container Audit*.

Note: The policies *auth_max_success* and *auth_max_fail* depend on reading the audit log. If you use a non readable audit log like the *Logger Audit* these policies will not work.

1.10.3 Container Audit

The *Container Audit* module is a meta audit module, that can be used to write audit information to more than one audit module.

It is configured in the `pi.cfg` like this:

```
PI_AUDIT_MODULE = 'privacyidea.lib.auditmodules.containeraudit'
PI_AUDIT_CONTAINER_WRITE = ['privacyidea.lib.auditmodules.sqlaudit','privacyidea.lib.
↪auditmodules.loggeraudit']
PI_AUDIT_CONTAINER_READ = 'privacyidea.lib.auditmodules.sqlaudit'
```

The key `PI_AUDIT_CONTAINER_WRITE` contains a list of audit modules, to which the audit information should be written. The listed audit modules need to be configured as mentioned in the corresponding audit module description.

The key `PI_AUDIT_CONTAINER_READ` contains one single audit module, that is capable of reading information. In this case the *SQL Audit* module can be used. The *Logger Audit* module can **not** be used for reading!

Using the *Container Audit* module you can on the one hand send audit information to external services using the *Logger Audit* but also keep the audit information visible within privacyIDEA using the *SQL Audit* module.

1.11 Machines

privacyIDEA lets you define Machine Resolvers to connect to existing machine stores. The idea is for users to be able to authenticate on those client machines. Not in all cases an online authentication request is possible, so that authentication items can be passed to those client machines.

In addition you need to define, which application on the client machine the user should authenticate to. Different application require different authentication items.

Therefore privacyIDEA can define application types. At the moment privacyIDEA knows the application `luks`, `offline` and `ssh`. You can write your own application class, which is defined in *Application Class*.

You need to assign an application and a token to a client machine. Each application type can work with certain token types and each application type can use additional parameters.

Note: Not all tokens work well with all applications!

1.11.1 SSH

Currently working token types: SSH

Parameters:

`user` (optional, default=root)

When the SSH token type is assigned to a client, the user specified in the user parameter can login with the private key of the SSH token.

In the `sshd_config` file you need to configure the `AuthorizedKeysCommand`. Set it to:

```
privacyidea-authorizedkeys
```

This will fetch the SSH public keys for the requesting machine.

The command expects a configuration file */etc/privacyidea/authorizedkeyscommand* which looks like this:

```
[Default]
url=https://localhost
admin=admin
password=test
nosslcheck=False
```

Note: To disable a SSH key for all servers, you simple can disable the SSH token in privacyIDEA.

Warning: In a productive environment you should not set **nosslcheck** to true, otherwise you are vulnerable to man in the middle attacks.

1.11.2 LUKS

Currently working token types: Yubikey Challenge Response

Parameters:

`slot` The slot to which the authentication information should be written

`partition` The encrypted partition (usually /dev/sda3 or /dev/sda5)

These authentication items need to be pulled on the client machine from the privacyIDEA server.

Thus, the following script need to be executed with root rights (able to write to LUKS) on the client machine:

```
privacyidea-luks-assign @secrets.txt --clearslot --name salt-minion
```

For more information please see the man page of this tool.

1.11.3 Offline

Currently working token types: HOTP.

Parameters:

`user` The local user, who should authenticate. (Only needed when calling machine/get_auth_items)

`count` The number of OTP values passed to the client.

The offline application also triggers when the client calls a /validate/check. If the user authenticates successfully with the correct token (serial number) and this very token is attached to the machine with an offline application the response to validate/check is enriched with a “auth_items” tree containing the salted SHA512 hashes of the next OTP values.

The client can cache these values to enable offline authentication. The caching is implemented in the privacyIDEA PAM module.

The server increases the counter to the last offline cached OTP value, so that it will not be possible to authenticate with those OTP values available offline on the client side.

1.12 Workflows and Tools

1.12.1 Import

Seed files that contain the secret keys of hardware tokens can be imported to the system via the menu *Import*.

The default import options are to import *SafeNet XML* file, *OATH CSV* files, *Yubikey CSV* files or *PSKC* files.

GPG Encryption

Starting with privacyIDEA 2.14 you can import GPG encrypted seed files. All files mentioned below can be encrypted this way.

privacyIDEA needs its own GPG key. You may create one like this:

```
mkdir /etc/privacyidea/gpg
GNUPGHOME=/etc/privacyidea/gpg gpg --gen-key
```

Then make sure, that the directory /etc/privacyidea/gpg is *chown 700* for the user *privacyidea*.

Now you can export the public key and hand it to your token vendor:

```
GNUPGHOME=/etc/privacyidea/gpg gpg -a --export <keyid>
```

Now the token vendor can send the seed file GPG encrypted. You do not need to decrypt the file and store the decrypted file on a network folder. Just import the GPG encrypted file to privacyIDEA!

Note: Using the key *PI_GNUPG_HOME* in pi.cfg you can change the default above mentioned *GNUPGHOME* directory.

Note: privacyIDEA imports an ASCII armored file. The file needs to be encrypted like this:

> gpg -e -a -r <keyid> import.csv

OATH CSV

This is a very simple CSV file to import HOTP, TOTP or OATH tokens. You can also convert your seed easily to this file format, to import the tokens.

The file format looks like this:

```
<serial>, <seed>, <type>, <otp length>, <time step>
```

For OCRA tokens it looks like this:

```
<serial>, <seed>, OCRA, <ocra suite>
```

serial is the serial number of the token that will also be used to identify the token in the database. Importing the same serial number twice will overwrite the token data.

seed is the secret key, that is used to calculate the OTP value. The seed is provided in a hexadecimal notation. Depending on the length either the SHA1 or SHA256 hash algorithm is identified.

type is either HOTP, TOTP or OCRA.

otp length is the length of the OTP value generated by the token. This is usually 6 or 8.

time step is the time step of TOTP tokens. This is usually 30 or 60.

ocra suite is the ocra suite of the OCRA token according to[1].

For TAN tokens it looks like this:

```
<serial>, <n/a>, TAN, <list of tans>
```

The list of tans is a whitespace separated list.

[1] http://tools.ietf.org/html/rfc6287#section-6

Import format version 2

A new import format allows to prepend a user, to whom the imported token should be assigned.

The file format needs to start with the first line

> # version: 2

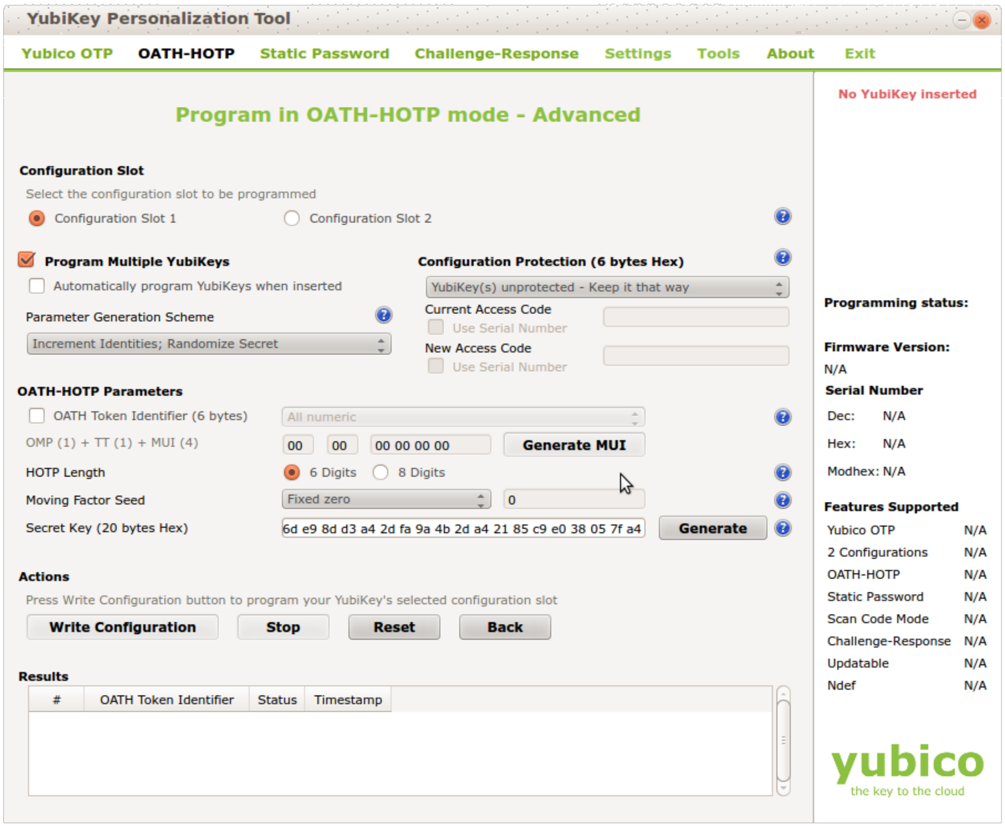
and the first three colums will be the user:

> <username>, <resolver>, <realm>, <serial>, <seed>, <type>, . . .

Note: The import will bail out, if a specified user does not exist.

Yubikey CSV

Here you can import the CSV file that is written by the Yubikey personalization tool[2]. privacyIDEA can import all Yubikey modes, either Yubico mode or HOTP mode.



Program in OATH-HOTP mode - Advanced

No YubiKey inserted

Configuration Slot

Select the configuration slot to be programmed

● Configuration Slot 1 ○ Configuration Slot 2

☑ Program Multiple YubiKeys

□ Automatically program YubiKeys when inserted

Parameter Generation Scheme: Increment Identities; Randomize Secret

Configuration Protection (6 bytes Hex): YubiKey(s) unprotected - Keep it that way

Current Access Code □ Use Serial Number

New Access Code □ Use Serial Number

OATH-HOTP Parameters

□ OATH Token Identifier (6 bytes): All numeric

OMP (1) + TT (1) + MUI (4): 00 00 00 00 00 00

HOTP Length: ● 6 Digits ○ 8 Digits

Moving Factor Seed: Fixed zero 0

Secret Key (20 bytes Hex): 6d e9 8d d3 a4 2d fa 9a 4b 2d a4 21 85 c9 e0 38 05 7f a4

Actions

Press Write Configuration button to program your YubiKey's selected configuration slot

Results

#	OATH Token Identifier	Status	Timestamp

Programming status:

Firmware Version: N/A

Serial Number

Dec:	N/A
Hex:	N/A
Modhex:	N/A

Features Supported

Yubico OTP	N/A
2 Configurations	N/A
OATH-HOTP	N/A
Static Password	N/A
Scan Code Mode	N/A
Challenge-Response	N/A
Updatable	N/A
Ndef	N/A

Note: There is an annoying drawback of the personalization tool: If you a initializing several HOTP yubikeys it will not write the serial number to the file.

[2] http://www.yubico.com/products/services-software/personalization-tools/use/

PSKC

The *Portable Symmetric Key Container* is specified in[3]. OATH compliant token vendors provide the token seeds in a PSKC file. privacyIDEA lets you import PSKC files. All necessary information (OTP length, Hash algorithm, token type) are read from the file.

PSKC files can be encrypted - either with a password or an AES key. You can provide this during the upload.

SafeNet XML

Safenet or former Aladdin provided seed files in their own XML format. This is the format to choose, if you have a file, that looks like this:

```
<Tokens>
    <Token serial="00040008CFA5">
    <CaseModel>5</CaseModel>
    <Model>101</Model>
    <ProductionDate>02/19/2009</ProductionDate>
    <ProductName>Safeword Alpine</ProductName>
    <Applications>
    <Application ConnectorID="{ab1397d2-ddb6-4705-b66e-9f83f322deb9}">
    <Seed>123412354</Seed>
    <MovingFactor>1</MovingFactor>
    </Application>
    </Applications>
    </Token>

    <Token ...>
    ...
    </Token>
 </Tokens>
```

Note: Safenet nowadays might provide you an XML file, which is probably a PKCS file. Please check the file contents!

1.12.2 Token Enrollment Wizard

The enrollment wizard helps the user to enroll his first token. When enrolling the first token, we assume, that the user is not very familiar with the privacyIDEA web UI. So the enrollment wizard only contains a very reduced API.

Necessary requirements for the enrollment wizard

- The enrollment wizard will only be displayed, if the user has no token assigned, yet. Thus the user must be able to login to the web UI with his userstore password. This is the default behaviour or set the corresponding policy.
- Set a policy in scope *webui* and activate the policy action *tokenwizard*.
- The user will not be able to choose a token type. But the default token type will be enrolled.

You can see the token enrollment wizard in action here: https://www.youtube.com/watch?v=diAGbsiG8_A

[3] https://tools.ietf.org/html/rfc6030

Customization

There are two dialog windows in the wizard. You can configure the text in the wizard in your html templates defined in these files:

Before the token is enrolled you can add your custom text in these two files static/customize/views/includes/token.enroll.pre.top.htm

static/customize/views/includes/token.enroll.pre.bottom.html

When it is enrolled and the user needs to do something (e.g. scanning the qr-code), you can modify the text here:

static/customize/views/includes/token.enroll.post.top.html static/customize/views/includes/token.enroll.post.bottom.html

Note: You can change the directory static/customize to a URL that fits your needs the best by defining a variable PI_CUSTOMIZATION in the file pi.cfg. This way you can put all modifications in one place apart from the original code.

Example

Your privacyIDEA system is running in the URL sub path `/pi`. The files could be addressed via a path component `mydesign` (in this case `pi/mydesign`). Thus the WebUI will look for the files in the URL path `/pi/mydesign/views/includes/`.

So you set in `pi.cfg`:

PI_CUSTOMIZATION = "/mydesign"

Your customized files are located in `/etc/privacyidea/customize/views/includes/`. In the Apache webserver you need to map `/pi/mydesign` to `/etc/privacyidea/customize`:

Alias /pi/mydesign /etc/privacyidea/customize

1.12.3 Tools

privacyIDEA comes with a list of command line tools, which also help to automate tasks.

privacyidea-token-janitor

Starting with version 2.19 privacyIDEA comes with a token janitor script. This script can find orphaned tokens, unused tokens or tokens of specific type, description or token info.

It can unassign, delete or disable those tokens and it can set additional tokeninfo or descriptions.

If you are unsure to directly delete orphaned tokens, because there might be a glimpse in the connection to your user store, you could as well in a first step *mark* the orphaned tokens. A day later you could run the script again and delete those tokens, which are (still) *orphaned* and *marked*.

privacyidea-get-unused-tokens

The script `privacyidea-get-unused-tokens` allows you to search for tokens, which were not used for authentication for a while. These tokens can be listed, disabled, marked or deleted.

You can specify how old the last authentication of such a token has to be. You can use the tags *h* (hours), *d* (day) and *y* (year). Sepcifying *180d* will find tokens, that were not used for authentication for the last 180 days.

The command:

```
privacyidea-get-unused-tokens disable 180d
```

will disable those tokens.

This script can be well used with the *Script Handler Module*.

1.12.4 Two Step Enrollment

Starting with version 2.21 privacyIDEA allows to enroll smartphone based tokens in a 2step enrollment.

With the rise of the smartphones and the fact that every user has a smartphone, carries it with him all the time and cares about it a lot, using the smartphone for authentication gets more and more attractive to IT departments.

Google came up with the Key URI[1] to use a QR code to easily enroll a smartphone token, i.e. transport the OTP secret from the server to the phone. However this bears some security issues as already pointed out[2].

This is why privacyIDEA allows to generate the OTP secret from a server component and from a client component (generated by the smartphone). This way the enrolled token is more tightly bound to this single smartphone and can not be copied that easily anymore.

Workflow

In a two step enrollment process the user clicks in the Web UI to enroll a token. The server generates a QR code and the user will scan this QR code with his smartphone app. The QR code contains the server component of the key and the information, that a second component is needed.

The smartphone generates the second component and displays this to the user.

The user enters this second component into the privacyIDEA Web UI.

Both the smartphone and the server calculate the OTP secret from both components.

Two Step policies

Two step enrollment is controlled by policies in the `admin`/`user` scope and in the `enrollment` scope.

Thus the administrator can *allow* or *force* a user (or other administrators) to do a two step enrollment. This way it is possible to avoid the enrollment of insecure Google Authenticator QR codes in the complete installation. (*hotp_2step and totp_2step*).

The default behaviour is to not allow a two step enrollment. Only if a corresponding `admin` or `user` policy is defined, two step enrollment is possible.

[1] https://github.com/google/google-authenticator/wiki/Key-Uri-Format

[2] https://netknights.it/en/the-problem-with-the-google-authenticator/

Key generation

In addition the administrator can define an enrollment policy to specify necessary parameters for the key generation.

Two step enrollment is possible for HOTP and TOTP tokens. Thus the administrator can define token type specific policies in the scope enrollment: hotp_2step_clientsize, totp_2step_clientsize, hotp_2step_difficulty... see *{type}_2step_clientsize, {type}_2step_serversize, {type}_2step_difficulty*.

privacyIDEA Authenticator

The privacyIDEA Authenticator[3] that is available from the Google Play Store supports the two step enrollment.

Specification

The two step enrollment simply adds some parameters to the original Key URI.

2step_output

This is the resulting key size, which the smartphone should generate (in bytes).

2step_salt

This is the length of the client component that the smartphone should generate (in bytes).

2step_difficulty

This is the number of rounds for the PBKDF2 that the smartphone should use to generate the OTP secret.

The secret parameter of the Key URI contains the server component.

The smartphone app then generates the client component, which is 2step_salt random bytes. It is then displayed in a human-readable format called base32check:

```
b32encode(sha1(client_component).digest()[0:4] + client_component).strip("=")
```

In other words, the first four bytes of the client component's SHA-1 hash are concatenated with the actual client component. The result is encoded using base32, whereas trailing padding characters are removed.

The second step of the enrollment process is realized as another request to the /token/init endpoint:

```
POST /token/init

serial=<token serial>
otpkey=<base32check(client_component)>
otpkeyformat=base32check
```

Server and smartphone app then use PBKDF2 to generate the final secret (see[4] for parameter names):

```
secret = PBKDF2(P=hexlify(<server component>),
                S=<client component>,
                c=<2step_difficulty>
                dkLen=<2step_output>)
```

[3] https://play.google.com/store/apps/details?id=it.netknights.piauthenticator

[4] https://www.ietf.org/rfc/rfc2898.txt

whereas `hexlify(<server component>)` denotes a hex-encoding (using lowercase letters) of the byte array which comprises the server component.

Note: Please note that the two-step enrollment process is currently *not* designed to protect against malicious attackers. Depending on the choice of iteration count and salt size, an attacker who knows the server component and an OTP value may be able to obtain the client component with a brute-force approach. However, two-step enrollment is still an improvement to the status quo, as a simple copy of the QR code does not immediately leak the OTP secret and obtaining the OTP secret using brute-force is not trivial.

1.13 Job Queue

privacyIDEA workflows often entail some time-consuming tasks, such as sending mails or SMS or saving usage statistics. Executing such tasks during the handling of API requests negatively affects performance. Starting with version 3.0, privacyIDEA allows to delegate certain tasks to external worker processes by using a job queue.

As an example, assume that privacyIDEA receives an authentication request by a user with an email token (see *Email*) via HTTP. privacyIDEA will send a one-time password via E-Mail. In order to do so, it communicates with a SMTP server. Normally, privacyIDEA handles all communication during the processing of the original authentication request, which increases the response time for the HTTP request, especially if the SMTP server is at a remote location.

A job queue can help to reduce the response time as follows. Instead of communicating with the SMTP server during request handling, privacyIDEA stores a so-called *job* in a job queue which says "Send an E-Mail to xyz@example.com with content '. . . '". privacyIDEA does not wait for the E-Mail to be actually sent, but already sends an HTTP response. An external *worker process* then retrieves the job from the queue and actually sends the corresponding E-Mail.

Using a job queue may improve the performance of your privacyIDEA server in case of a flaky connection to the SMTP server. Authentication requests that send E-Mails are then handled faster (because the privacyIDEA server does not actually communicate with the SMTP server), which means that the corresponding web server worker thread can handle the next request faster.

privacyIDEA 3.0 implements a job queue based on huey which uses a Redis server to store jobs. As of version 3.0, privacyIDEA allows to offload sending mails to the queue. Other jobs will be implemented in future versions.

1.13.1 Configuration

The job queue is disabled by default. In order to enable it, add the following configuration option to `pi.cfg`:

```
PI_JOB_QUEUE_CLASS = 'privacyidea.lib.queues.huey_queue.HueyQueue'
```

After a server restart, you will be able to instruct individual SMTP servers to send all mails via the job queue by checking a corresponding box in the SMTP server configuration (see *SMTP server configuration*). This means that you can have separate SMTP server configurations, some of which send mails via the job queue, some of which send mails during the request processing.

Note that you need to run a Redis server which is reachable for the privacyIDEA server. By default, huey assumes a locally running Redis server. You can use a configuration option to provide a different URL (see here for information on the URL format):

```
PI_JOB_QUEUE_URL = 'redis://somehost'
```

In addition to the privacyIDEA server, you will have to run a worker process which fetches jobs from the queue and executes them. You can start it as follows:

```
privacyidea-queue-huey
```

By default, the worker process logs to `privacyidea-queue.log` in the current working directory. You can pass a different logfile by using the `-l` option:

```
privacyidea-queue-huey -l /var/log/queue.log
```

As the script is heavily based on the huey consumer script, you can find information about additional options in the huey documentation.

Note that a side-effect of the queue is that the privacyIDEA server will not throw or log errors if a mail could not be sent. Hence, it is important to monitor the queue log file for errors.

1.14 Application Plugins

privacyIDEA comes with application plugins. These are plugins for applications like PAM, OTRS, Apache2, FreeRADIUS, ownCloud, simpleSAMLphp or Keycloak which enable these application to authenticate users against privacyIDEA.

You may also write your own application plugin or connect your own application to privacyIDEA. This is quite simple using a REST API *Validate endpoints*. In order to support more sophisticated token types like challenge-response or out-of-band tokens, you should take a look at the various *Authentication Modes*.

1.14.1 Pluggable Authentication Module

The PAM module of privacyIDEA directly communicates with the privacyIDEA server via the API. The PAM module also supports offline authentication. In this case you need to configure an offline machine application. (See *Offline*)

You can install the PAM module by using the source code file. It is a python module, that requires python-pam:

```
git clone https://github.com/privacyidea/pam_python.git
cd pam_python
pip install -r requirements.txt
python ./setup.py install
```

The configuration could look like this:

```
... pam_python.so /path/to/privacyidea_pam.py
url=https://localhost prompt=privacyIDEA_Authentication
```

The URL parameter defaults to `https://localhost`. You can also add the parameters `realm=` and `debug`.

If you want to disable certificate validation, which you should **not** do in a productive environment, you can use the parameter `nosslverify`.

A new parameter `cacerts=` lets you define a CA Cert-Bundle file, that contains the trusted certificate authorities in PEM format.

The default behaviour is to trigger an online authentication request. If the request was successful, the user is logged in. If the request was done with a token defined for offline authentication, then in addition all offline information is passed to the client and cached on the client so that the token can be used to authenticate without the privacyIDEA server available.

try_first_pass

Starting with version 2.8 privacyidea_pam supports *try_first_pass*. In this case the password that exists in the PAM stack will be sent to privacyIDEA. If this password is successfully validated, than the user is logged in without additional requests. If the password is not validated by privacyIDEA, the user is asked for an additional OTP value.

Note: This can be used in conjunction with the *passthru* policy. In this case users with no tokens will be able to login with only the password in the PAM stack.

Use cases SSH and VPN

PrivacyIDEA can be easily used to setup a secure SSH login combining SSH keys with a second factor. The configuration is given in SSH Keys and OTP: Really strong two factor authentication on the privacyIDEA website.

Read more about how to use PAM to do openvpn.

1.14.2 Using pam_yubico

If you are using yubikey tokens you might also use `pam_yubico`. You can use Yubikey tokens for two more or less distinct applications. The first is using privacyideas PAM module as described above. In this case privacyidea handles the policies for user access and password validation. This works fine, when you only use privacyidea for token validation.

The second mode is using the standard PAM module for yubikeys from Yubico `pam_yubico` to handle the token validation. The upside ist that you can use the PAM module included with you distribution, but there are downsides as well.

- You can't set a token PIN in privacyidea, because `pam_yubico` tries to use the token PIN entered by the user as a system password (which is likely to fail), i.e. the PIN will be stripped by `pam_yubico` and will not reach the privacyIDEA system.
- Setting the policy which tokens are valid for which users is done either in `~/.yubico/authorized_keys` or in the file given by the `authfile` option in the PAM configuration. The api server will only validate the token, but not check any kind of policy.

You can work around the restrictions by using a clever combination of tokentype yubikey and yubico as follows:

- enroll a yubikey token with `yubikey_mass_enroll --mode YUBICO`.
- do not set a token password.
- do not assign the token to a user.
- please make a note of yubikey.prefix (12 characters starting with vv).

Now the token can be used with `pam_yubico`, but will not allow any user access in privacyidea. If you want to use the token with `pam_yubico` see the manual page for details. You'll want something like the following in your PAM config:

```
auth required pam_yubico.so id=<apiid> key=<API key> \
     urllist=https://<privacyidea-server>/ttype/yubikey authfile=/etc/yubikeys/
↪authorized_yubikeys
```

The file `/etc/yubikeys/authorized_yubikeys` contains a line for each user with the username and the allowed tokens delimited by ":", for example:

```
<username>:<serial number1>:<prefix1>:<prefix2>
```

Now create a second token representing the Yubikey, but this time use the `Yubico Cloud mode`. Go to Tokens -> Enroll Token and select `Yubico Cloud mode`. Enter the 12 characters prefix you noted above and assign this token to a user and possibly set a token PIN. It would be nice to have the the serial number of the UBCM token correspond to the UBAM token, but this is right now not possible with the WebUI.

In the WebUI, test the UBAM token without a Token PIN, test the UBCM token with the stored Token PIN, and check the token info afterwards. Check the yubikey token via `/ttype/yubikey`, for example with:

```
ykclient --debug --url https://<privacyidea>/ttype/yubikey --apikey "<API key>" "apiid" <otp>
```

There should be successful authentications (count_auth_success), but no failures.

1.14.3 FreeRADIUS

Starting with privacyIDEA 2.19, there are two ways to integrate FreeRADIUS:

- Using a Perl-based privacyIDEA plugin, which is available for FreeRADIUS 2.0.x and above. It supports advanced use cases (such as challenge-response authentication or attribute mapping). Read more about it at rlm_perl.
- Using the rlm_rest plugin provided by FreeRADIUS 3.0.x and above. However, this setup does not support challenge-response or attribute mapping. Read more about it at rlm_rest.

With either setup, you can test the RADIUS setup using a command like this:

```
echo "User-Name=user, User-Password=password" | radclient -sx yourRadiusServer \
   auth topsecret
```

Note: Do not forget to configure the `clients.conf` accordingly.

1.14.4 Microsoft NPS server

You can also use the Microsoft Network Protection Server with privacyIDEA. A full featured integration guide can be found at the NetKnights webpage.

1.14.5 simpleSAMLphp Plugin

You can install the plugin for simpleSAMLphp using the source files from the GitHub Repository simplesamplphp-module-privacyidea.

Follow the simpleSAMLphp instructions to configure your authsources.php. A usual configuration will look like this:

```
'example-privacyidea' => array(
    'privacyidea:privacyidea',

    /*
     * The name of the privacyidea server and the protocol
     * A port can be added by a colon
     * Required.
```

(continues on next page)

(continued from previous page)

```
     */
    'privacyideaserver' => 'https://your.server.com',

    /*
     * Check if the hostname matches the name in the certificate
     * Optional.
     */
    'sslverifyhost' => False,

    /*
     * Check if the certificate is valid, signed by a trusted CA
     * Optional.
     */
    'sslverifypeer' => False,

    /*
     * The realm where the user is located in.
     * Optional.
     */
    'realm' => '',

    /*
     * This is the translation from privacyIDEA attribute names to
     * SAML attribute names.
     */
     'attributemap' => array('username' => 'samlLoginName',
                             'surname' => 'surName',
                             'givenname' => 'givenName',
                             'email' => 'emailAddress',
                             'phone' => 'telePhone',
                             'mobile' => 'mobilePhone',
                             ),
),
```

1.14.6 Keycloak

With the privacyIDEA keycloak-provider, there is a plugin available for the Keycloak identity manager. It is available from the GitHub repository keycloak-provider.

Like simpleSAMLphp, it can be used to realize single sign-on use cases with a strong second factor authentication.

1.14.7 TYPO3

You can install the privacyIDEA extension from the TYPO3 Extension Repository. The privacyIDEA extension is easily configured.

privacyIDEA Server URL

This is the URL of your privacyIDEA installation. You do not need to add the path *validate/check*. Thus the URL for a common installation would be *https://yourServer/*.

Check certificate

Whether the validity of the SSL certificate should be checked or not.

Warning: If the SSL certificate is not checked, the authentication request could be modified and the answer to the request can be modified, easily granting access to an attacker.

Enable privacyIDEA for backend users

If checked, a user trying to authenticate at the backend, will need to authenticate against privacyIDEA.

Enable privacyIDEA for frontend users

If checked, a user trying to authenticate at the frontend, will need to authenticate against privacyIDEA.

Pass to other authentication module

If the authentication at privacyIDEA fails, the credential the user entered will be verified against the next authentication module.

This can come in handy, if you are setting up the system and if you want to avoid locking yourself out.

Anyway, in a productive environment you probably want to uncheck this feature.

1.14.8 OTRS

The OTRS Plugin can be found in its own GitHub Repository.

This perl module needs to be installed to the directory `Kernel/System/Auth`.

To activate the OTP authentication you need to add the following to `Kernel/Config.pm`:

```
$Self->{'AuthModule'} = 'Kernel::System::Auth::privacyIDEA';
$Self->{'AuthModule::privacyIDEA::URL'} = \
        "https://localhost/validate/check";
$Self->{'AuthModule::privacyIDEA::disableSSLCheck'} = "yes";
```

Note: As mentioned earlier you should only disable the checking of the SSL certificate if you are in a test environment. For productive use you should never disable the SSL certificate checking.

Note: This plugin requires, that you also add the path *validate/check* to the URL.

1.14.9 Apache2

The Apache plugin uses `mod_wsgi` and `redis` to provide a basic authentication on Apache2 side and validating the credentials against privacyIDEA.

You need the authentication script `privacyidea_apache.py` and a valid configuration in `/etc/privacyidea/apache.conf`. Both can be found on GitHub.

To activate the OTP authentication on a "Location" or "Directory" you need to configure Apache2 like this:

```
<Directory /var/www/html/secretdir>
     AuthType Basic
     AuthName "Protected Area"
     AuthBasicProvider wsgi
     WSGIAuthUserScript /usr/share/pyshared/privacyidea_apache.py
```

(continues on next page)

(continued from previous page)

```
     Require valid-user
</Directory>
```

Note: Basic Authentication sends the base64 encoded password on each request. So the browser will send the same one time password with each request. Thus the authentication module needs to cache the password when the authentication is successful. Redis is used for caching the password.

Warning: As redis per default is accessible by every user on the machine, you need to use this plugin with caution! Every user on the machine can access the redis database to read the passwords of the users. The cached credentials are stored as pbkdf2+sha512 hash.

1.14.10 NGINX

The NGINX plugin uses the internal scripting language `lua` of the NGINX webserver and `redis` as caching backend to provide basic authentication against privacyIDEA.

You can retrieve the nginx plugin from GitHub.

To activate the OTP authentication on a "Location" you need to include the `lua` script that basically verifies the given credentials against the caching backend. New authentications will be sent to a different (internal) location via subrequest which points to the privacyIDEA authentication backend (via proxy_pass).

For the basic configuration you need to include the following lines to your `location` block:

```
location / {
    # additional plugin configuration goes here #
    access_by_lua_file 'privacyidea.lua';
}
location /privacyidea-validate-check {
    internal;
    proxy_pass https://privacyidea/validate/check;
}
```

You can customize the authentication plugin by setting some of the following variables in the secured `location` block:

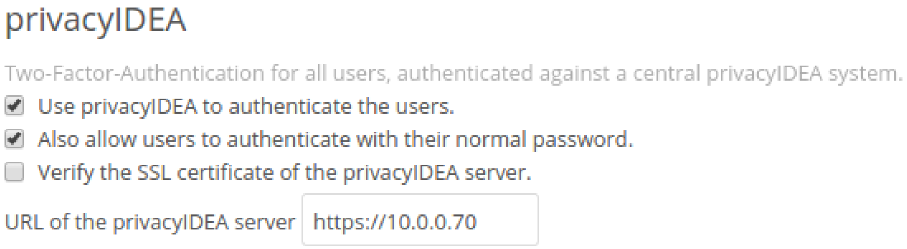
```
# redis host:port
# set $privacyidea_redis_host "127.0.0.1";
set $privacyidea_redis_post 6379;

# how long are accepted authentication allowed to be cached
# if expired, the user has to reauthenticate
set $privacyidea_ttl 900;

# privacyIDEA realm. leave empty == default
set $privacyidea_realm 'somerealm'; # (optional)

# pointer to the internal validation proxy pass
set $privacyidea_uri "/privacyidea-validate-check";

# the http realm presented to the user
set $privacyidea_http_realm "Secure zone (use PIN + OTP)";
```

Note: Basic Authentication sends the base64 encoded password on each request. So the browser will send the same one time password with each reqeust. Thus the authentication module needs to cache the password as the successful authentication. Redis is used for caching the password similar to the Apache2 plugin.

Warning: As redis per default is accessible by every user on the machine, you need to use this plugin with caution! Every user on the machine can access the redis database to read the passwords of the users. The cached credentials are stored as SHA1_HMAC hash. If you prefer a stronger hashing method feel free to extend the given `password_hash/verify` functions using additional lua libraries (for example by using `lua-resty-string`).

1.14.11 ownCloud

The ownCloud plugin is a ownCloud user backend. The directory `user_privacyidea` needs to be copied to your owncloud `apps` directory.



privacyIDEA

Two-Factor-Authentication for all users, authenticated against a central privacyIDEA system.

☑ Use privacyIDEA to authenticate the users.

☑ Also allow users to authenticate with their normal password.

□ Verify the SSL certificate of the privacyIDEA server.

URL of the privacyIDEA server: https://10.0.0.70

Fig. 62: *Activating the ownCloud plugin*

You can then activate the privacyIDEA ownCloud plugin by checking *Use privacyIDEA to authenticate the users.* All users now need to be known to privacyIDEA and need to authenticate using the second factor enrolled in privacyIDEA - be it an OTP token, Google Authenticator or SMS/Smartphone.

Checking *Also allow users to authenticate with their normal passwords.* lets the user choose if he wants to authenticate with the OTP token or with his original password from the original user backend.

Note: At the moment using a desktop client with a one time password is not supported.

ownCloud 9.1 and Nextcloud 10 come with a new two factor framework. The new privacyIDEA ownCloud App allows you to add a second factor, that is centrally managed by privacyIDEA to the ownCloud or Nextcloud installation.

The ownCloud privacyIDEA App is available from the ownCloud App Store.

The App requires a subscription file to work for more than ten users. You can get the subscription file from NetKnights.

1.14.12 Django

You can add two factor authentication with privacyIDEA to Django using this Django plugin.

You can simply add `PrivacyIDEA` class to the `AUTHENTICATION_BACKENDS` settings of Django.

1.14.13 OpenVPN

Read more about how to use OpenVPN with privacyidea at openvpn.

1.14.14 Windows

Credential Provider

The privacyIDEA Credential Provider adds two factor authentication to the Windows desktop or Terminal server. See http://privacyidea-credential-provider.readthedocs.io

Provider Class

There is a dot Net provider class, which you can use to integrate privacyIDEA authentication into other products and worflows. See https://github.com/sbidy/privacyIDEA_dotnetProvider

1.14.15 Further plugins

You can find further plugins for Dokuwiki, Wordpress, Contao and Django at cornelinux Github page.

1.15 Code Documentation

The code roughly has three levels: API, LIB and DB.

1.15.1 API level

The API level is used to access the system. For some calls you need to be authenticated as administrator, for some calls you can be authenticated as normal user. These are the `token` and the `audit` endpoint. For calls to the `validate` API you do not need to be authenticated at all.

At this level `Authentication` is performed. In the lower levels there is no authentication anymore.

The object `g.logged_in_user` is used to pass the authenticated user. The client gets a JSON Web Token to authenticate every request.

API functions are decorated with the decorators `admin_required` and `user_required` to define access rules.

REST API

This is the REST API for privacyidea. It lets you create the system configuration, which is denoted in the system endpoints.

Special system configuration is the configuration of

- the resolvers
- the realms
- the defaultrealm
- the policies.

Resolvers are dynamic links to existing user sources. You can find users in LDAP directories, SQL databases, flat files or SCIM services. A resolver translates a loginname to a user object in the user source and back again. It is also responsible for fetching all additional needed information from the user source.

Realms are collections of resolvers that can be managed by administrators and where policies can be applied.

Defaultrealm is a special endpoint to define the default realm. The default realm is used if no user realm is specified. If a user from realm1 tries to authenticate or is addressed, the notation user@realm1 is used. If the @realm1 is omitted, the user is searched in the default realm.

Policies are rules how privacyidea behaves and which user and administrator is allowed to do what.

Start to read about authentication to the API at *Authentication endpoints*.

Now you can take a look at the several REST endpoints. This REST API is used to authenticate the users. A user needs to authenticate when he wants to use the API for administrative tasks like enrolling a token.

This API must not be confused with the validate API, which is used to check, if a OTP value is valid. See *Validate endpoints*.

Authentication of users and admins is tested in tests/test_api_roles.py

You need to authenticate for all administrative tasks. If you are not authenticated, the API returns a 401 response.

To authenticate you need to send a POST request to /auth containing username and password.

Audit endpoint

GET /audit/

return a paginated list of audit entries.

Params can be passed as key-value-pairs.

Httpparam timelimit A timelimit, that limits the recent audit entries. This param gets overwritten by a policy auditlog_age. Can be 1d, 1m, 1h.

Example request:

```
GET /audit?realm=realm1 HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

```

(continues on next page)

(continued from previous page)

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": [
      {
         "serial": "....",
         "missing_line": "..."
      }
    ]
  },
  "version": "privacyIDEA unknown"
}
```

GET /audit/ (*csvfile*)

Download the audit entry as CSV file.

Params can be passed as key-value-pairs.

Example request:

```
GET /audit/audit.csv?realm=realm1 HTTP/1.1
Host: example.com
Accept: text/csv
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: text/csv

 {
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": [
       {
          "serial": "....",
          "missing_line": "..."
       }
     ]
   },
   "version": "privacyIDEA unknown"
 }
```

Authentication endpoints

This REST API is used to authenticate the users. A user needs to authenticate when he wants to use the API for administrative tasks like enrolling a token.

This API must not be confused with the validate API, which is used to check, if a OTP value is valid. See *Validate endpoints*.

Authentication of users and admins is tested in tests/test_api_roles.py

You need to authenticate for all administrative tasks. If you are not authenticated, the API returns a 401 response.

To authenticate you need to send a POST request to /auth containing username and password.

GET /auth/rights

This returns the rights of the logged in user.

Request Headers

- Authorization – The authorization token acquired by /auth request

POST /auth

This call verifies the credentials of the user and issues an authentication token, that is used for the later API calls. The authentication token has a validity, that is usually 1 hour.

JSON Parameters

- **username** – The username of the user who wants to authenticate to the API.
- **password** – The password/credentials of the user who wants to authenticate to the API.
- **realm** – The realm where the user will be searched.

Return A json response with an authentication token, that needs to be used in any further request.

Status Codes

- 200 OK – in case of success
- 401 Unauthorized – if authentication fails

Example Authentication Request:

```
POST /auth HTTP/1.1
Host: example.com
Accept: application/json

username=admin
password=topsecret
```

Example Authentication Response:

```
HTTP/1.0 200 OK
Content-Length: 354
Content-Type: application/json

{
     "id": 1,
     "jsonrpc": "2.0",
     "result": {
         "status": true,
         "value": {
             "token": "eyJhbGciOiJIUz....jdpn9kIjuGRnGejmbFbM"
         }
     },
     "version": "privacyIDEA unknown"
}
```

Response for failed authentication:

```
HTTP/1.1 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 203

```

(continues on next page)

(continued from previous page)

```
{
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "error": {
       "code": -401,
       "message": "missing Authorization header"
     },
     "status": false
   },
   "version": "privacyIDEA unknown",
   "config": {
     "logout_time": 30
   }
}
```

Example Request:

Requests to privacyidea then should use this security token in the Authorization field in the header.

```
GET /users/ HTTP/1.1
Host: example.com
Accept: application/json
Authorization: eyJhbGciOiJIUz....jdpn9kIjuGRnGejmbFbM
```

Validate endpoints

This module contains the REST API for doing authentication. The methods are tested in the file tests/test_api_validate.py

Authentication is either done by providing a username and a password or a serial number and a password.

Authentication workflow

Authentication workflow is like this:

In case of authenticating a user:

- *privacyidea.lib.token.check_user_pass()*
- *privacyidea.lib.token.check_token_list()*
- *privacyidea.lib.tokenclass.TokenClass.authenticate()*
- *privacyidea.lib.tokenclass.TokenClass.check_pin()*
- *privacyidea.lib.tokenclass.TokenClass.check_otp()*

In case if authenitcating a serial number:

- *privacyidea.lib.token.check_serial_pass()*
- *privacyidea.lib.token.check_token_list()*
- *privacyidea.lib.tokenclass.TokenClass.authenticate()*
- *privacyidea.lib.tokenclass.TokenClass.check_pin()*
- *privacyidea.lib.tokenclass.TokenClass.check_otp()*

POST /validate/triggerchallenge

An administrator can call this endpoint if he has the right of `triggerchallenge` (scope: admin). He can pass a `user` name and or a `serial` number. privacyIDEA will trigger challenges for all native challenges response tokens, possessed by this user or only for the given serial number.

The request needs to contain a valid PI-Authorization header.

Parameters

- **user** – The loginname/username of the user, who tries to authenticate.
- **realm** – The realm of the user, who tries to authenticate. If the realm is omitted, the user is looked up in the default realm.
- **serial** – The serial number of the token.

Return a json result with a "result" of the number of matching challenge response tokens

Example response for a successful triggering of challenge:

```
{"jsonrpc": "2.0",
 "signature": "1939...146964",
 "detail": {"transaction_ids": ["03921966357577766962"],
            "messages": ["Enter the OTP from the SMS:"],
            "threadid": 140422378276608},
 "versionnumber": "unknown",
 "version": "privacyIDEA unknown",
 "result": {"status": true,
            "value": 1},
 "time": 1482223663.517212,
 "id": 1}
```

Example response for response, if the user has no challenge token:

```
{"detail": {"messages": [],
            "threadid": 140031212377856,
            "transaction_ids": []},
 "id": 1,
 "jsonrpc": "2.0",
 "result": {"status": true,
            "value": 0},
 "signature": "205530282...54508",
 "time": 1484303812.346576,
 "version": "privacyIDEA 2.17",
 "versionnumber": "2.17"}
```

Example response for a failed triggering of a challenge. In this case

the `status` will be `false`.

```
{"detail": null,
 "id": 1,
 "jsonrpc": "2.0",
 "result": {"error": {"code": 905,
                      "message": "ERR905: The user can not be
                      found in any resolver in this realm!"},
            "status": false},
 "signature": "14468...081555",
 "time": 1484303933.72481,
 "version": "privacyIDEA 2.17"}
```

GET /validate/triggerchallenge

An administrator can call this endpoint if he has the right of `triggerchallenge` (scope: admin). He can pass a `user` name and or a `serial` number. privacyIDEA will trigger challenges for all native challenges response tokens, possessed by this user or only for the given serial number.

The request needs to contain a valid PI-Authorization header.

Parameters

- **`user`** – The loginname/username of the user, who tries to authenticate.
- **`realm`** – The realm of the user, who tries to authenticate. If the realm is omitted, the user is looked up in the default realm.
- **`serial`** – The serial number of the token.

Return a json result with a "result" of the number of matching challenge response tokens

Example response for a successful triggering of challenge:

```
{"jsonrpc": "2.0",
 "signature": "1939...146964",
 "detail": {"transaction_ids": ["03921966357577766962"],
            "messages": ["Enter the OTP from the SMS:"],
            "threadid": 140422378276608},
 "versionnumber": "unknown",
 "version": "privacyIDEA unknown",
 "result": {"status": true,
            "value": 1},
 "time": 1482223663.517212,
 "id": 1}
```

Example response for response, if the user has no challenge token:

```
{"detail": {"messages": [],
            "threadid": 140031212377856,
            "transaction_ids": []},
 "id": 1,
 "jsonrpc": "2.0",
 "result": {"status": true,
            "value": 0},
 "signature": "205530282...54508",
 "time": 1484303812.346576,
 "version": "privacyIDEA 2.17",
 "versionnumber": "2.17"}
```

Example response for a failed triggering of a challenge. In this case

the `status` will be `false`.

```
{"detail": null,
 "id": 1,
 "jsonrpc": "2.0",
 "result": {"error": {"code": 905,
                      "message": "ERR905: The user can not be
                      found in any resolver in this realm!"},
            "status": false},
 "signature": "14468...081555",
 "time": 1484303933.72481,
 "version": "privacyIDEA 2.17"}
```

GET /validate/polltransaction/(*transaction_id*)

GET /validate/polltransaction

Given a mandatory transaction ID, check if any non-expired challenge for this transaction ID has been answered. In this case, return true. If this is not the case, return false. This endpoint also returns false if no challenge with the given transaction ID exists.

This is mostly useful for out-of-band tokens that should poll this endpoint to determine when to send an authentication request to `/validate/check`.

JSON Parameters

- **`transaction_id`** – a transaction ID

POST /validate/offlinerefill

This endpoint allows to fetch new offline OTP values for a token, that is already offline. According to the definition it will send the missing OTP values, so that the client will have as much otp values as defined.

Parameters

- **`serial`** – The serial number of the token, that should be refilled.
- **`refilltoken`** – The authorization token, that allows refilling.
- **`pass`** – the last password (maybe password+OTP) entered by the user

Return

POST /validate/radiuscheck

POST /validate/check

check the authentication for a user or a serial number. Either a `serial` or a `user` is required to authenticate. The PIN and OTP value is sent in the parameter `pass`. In case of successful authentication it returns `result->value: true`.

In case of a challenge response authentication a parameter `exception=1` can be passed. This would result in a HTTP 500 Server Error response if an error occurred during sending of SMS or Email.

In case `/validate/radiuscheck` is requested, the responses are modified as follows: A successful authentication returns an empty HTTP 204 response. An unsuccessful authentication returns an empty HTTP 400 response. Error responses are the same responses as for the `/validate/check` endpoint.

Parameters

- **`serial`** – The serial number of the token, that tries to authenticate.
- **`user`** – The loginname/username of the user, who tries to authenticate.
- **`realm`** – The realm of the user, who tries to authenticate. If the realm is omitted, the user is looked up in the default realm.
- **`pass`** – The password, that consists of the OTP PIN and the OTP value.
- **`otponly`** – If set to 1, only the OTP value is verified. This is used in the management UI. Only used with the parameter serial.
- **`transaction_id`** – The transaction ID for a response to a challenge request
- **`state`** – The state ID for a response to a challenge request

Return a json result with a boolean "result": true

Example Validation Request:

```
POST /validate/check HTTP/1.1
Host: example.com
Accept: application/json

user=user
realm=realm1
pass=s3cret123456
```

Example response for a successful authentication:

```
HTTP/1.1 200 OK
Content-Type: application/json

 {
   "detail": {
     "message": "matching 1 tokens",
     "serial": "PISP0000AB00",
     "type": "spass"
   },
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": true
   },
   "version": "privacyIDEA unknown"
 }
```

Example response for this first part of a challenge response authentication:

```
HTTP/1.1 200 OK
Content-Type: application/json

 {
   "detail": {
     "serial": "PIEM0000AB00",
     "type": "email",
     "transaction_id": "12345678901234567890",
     "multi_challenge: [ {"serial": "PIEM0000AB00",
                          "transaction_id":  "12345678901234567890",
                          "message": "Please enter otp from your
                          email"},
                         {"serial": "PISM12345678",
                          "transaction_id": "12345678901234567890",
                          "message": "Please enter otp from your
                          SMS"}
     ]
   },
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": false
   },
   "version": "privacyIDEA unknown"
 }
```

In this example two challenges are triggered, one with an email and one with an SMS. The application and thus the user has to decide, which one to use. They can use either.

Note: All challenge response tokens have the same transaction_id in this case.

GET /validate/radiuscheck

GET /validate/check

check the authentication for a user or a serial number. Either a `serial` or a `user` is required to authenticate. The PIN and OTP value is sent in the parameter `pass`. In case of successful authentication it returns `result->value:  true`.

In case of a challenge response authentication a parameter `exception=1` can be passed. This would result in a HTTP 500 Server Error response if an error occurred during sending of SMS or Email.

In case `/validate/radiuscheck` is requested, the responses are modified as follows: A successful authentication returns an empty HTTP 204 response. An unsuccessful authentication returns an empty HTTP 400 response. Error responses are the same responses as for the `/validate/check` endpoint.

Parameters

- **serial** – The serial number of the token, that tries to authenticate.
- **user** – The loginname/username of the user, who tries to authenticate.
- **realm** – The realm of the user, who tries to authenticate. If the realm is omitted, the user is looked up in the default realm.
- **pass** – The password, that consists of the OTP PIN and the OTP value.
- **otponly** – If set to 1, only the OTP value is verified. This is used in the management UI. Only used with the parameter serial.
- **transaction_id** – The transaction ID for a response to a challenge request
- **state** – The state ID for a response to a challenge request

Return a json result with a boolean "result": true

Example Validation Request:

```
POST /validate/check HTTP/1.1
Host: example.com
Accept: application/json

user=user
realm=realm1
pass=s3cret123456
```

Example response for a successful authentication:

```
HTTP/1.1 200 OK
Content-Type: application/json

 {
   "detail": {
     "message": "matching 1 tokens",
     "serial": "PISP0000AB00",
     "type": "spass"
   },
   "id": 1,
```

(continues on next page)

(continued from previous page)

```
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": true
   },
   "version": "privacyIDEA unknown"
 }
```

Example response for this first part of a challenge response authentication:

```
HTTP/1.1 200 OK
Content-Type: application/json

 {
   "detail": {
     "serial": "PIEM0000AB00",
     "type": "email",
     "transaction_id": "12345678901234567890",
     "multi_challenge: [ {"serial": "PIEM0000AB00",
                          "transaction_id":  "12345678901234567890",
                          "message": "Please enter otp from your
                          email"},
                         {"serial": "PISM12345678",
                          "transaction_id": "12345678901234567890",
                          "message": "Please enter otp from your
                          SMS"}
     ]
   },
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": false
   },
   "version": "privacyIDEA unknown"
 }
```

In this example two challenges are triggered, one with an email and one with an SMS. The application and thus the user has to decide, which one to use. They can use either.

Note: All challenge response tokens have the same transaction_id in this case.

POST /validate/samlcheck

Authenticate the user and return the SAML user information.

Parameters

- **user** – The loginname/username of the user, who tries to authenticate.
- **realm** – The realm of the user, who tries to authenticate. If the realm is omitted, the user is looked up in the default realm.
- **pass** – The password, that consists of the OTP PIN and the OTP value.

Return a json result with a boolean “result”: true

Example response for a successful authentication:

```
HTTP/1.1 200 OK
Content-Type: application/json

 {
   "detail": {
     "message": "matching 1 tokens",
     "serial": "PISP0000AB00",
     "type": "spass"
   },
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": {"attributes": {
                 "username": "koelbel",
                 "realm": "themis",
                 "mobile": null,
                 "phone": null,
                 "myOwn": "/data/file/home/koelbel",
                 "resolver": "themis",
                 "surname": "Kölbel",
                 "givenname": "Cornelius",
                 "email": null},
               "auth": true}
   },
   "version": "privacyIDEA unknown"
 }
```

The response in value->attributes can contain additional attributes (like “myOwn”) which you can define in the LDAP resolver in the attribute mapping.

GET /validate/samlcheck

Authenticate the user and return the SAML user information.

Parameters

- **user** – The loginname/username of the user, who tries to authenticate.
- **realm** – The realm of the user, who tries to authenticate. If the realm is omitted, the user is looked up in the default realm.
- **pass** – The password, that consists of the OTP PIN and the OTP value.

Return a json result with a boolean “result”: true

Example response for a successful authentication:

```
HTTP/1.1 200 OK
Content-Type: application/json

 {
   "detail": {
     "message": "matching 1 tokens",
     "serial": "PISP0000AB00",
     "type": "spass"
   },
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
```

(continues on next page)

(continued from previous page)

```
    "value": {"attributes": {
                "username": "koelbel",
                "realm": "themis",
                "mobile": null,
                "phone": null,
                "myOwn": "/data/file/home/koelbel",
                "resolver": "themis",
                "surname": "Kölbel",
                "givenname": "Cornelius",
                "email": null},
              "auth": true}
  },
  "version": "privacyIDEA unknown"
}
```

The response in value->attributes can contain additional attributes (like "myOwn") which you can define in the LDAP resolver in the attribute mapping.

System endpoints

This is the REST API for system calls to create and read system configuration.

The code of this module is tested in tests/test_api_system.py

GET /system/names/caconnector

Return a list of defined CA connectors. Each item of the list is a dictionary with the CA connector information, including the name and defined templates, but excluding the CA connector data. This endpoint requires the enrollCERTIFICATE right.

GET /system/names/radius

Return the list of identifiers of all defined RADIUS servers. This endpoint requires the enrollRADIUS right.

GET /system/documentation

returns an restructured text document, that describes the complete configuration.

POST /system/setDefault

define default settings for tokens. These default settings are used when new tokens are generated. The default settings will not affect already enrolled tokens.

JSON Parameters

- **DefaultMaxFailCount** – Default value for the maximum allowed authentication failures
- **DefaultSyncWindow** – Default value for the synchronization window
- **DefaultCountWindow** – Default value for the counter window
- **DefaultOtpLen** – Default value for the OTP value length – usually 6 or 8
- **DefaultResetFailCount** – Default value, if the FailCounter should be reset on successful authentication [True|False]

Return a json result with a boolean "result": true

POST /system/setConfig

set a configuration key or a set of configuration entries

parameter are generic keyname=value pairs.

remark In case of key-value pairs the type information could be provided by an additional parameter with same keyname with the postfix ".type". Value could then be 'password' to trigger the storing of the value in an encrypted form

JSON Parameters

- **`key`** – configuration entry name
- **`value`** – configuration value
- **`type`** – type of the value: int or string/text or password. password will trigger to store the encrypted value
- **`description`** – additional information for this config entry

or

JSON Parameters

- **`pairs`** (*`key-value`*) – pair of &keyname=value pairs

Return a json result with a boolean "result": true

Example request 1:

```
POST /system/setConfig
key=splitAtSign
value=true

Host: example.com
Accept: application/json
```

Example request 2:

```
POST /system/setConfig
BINDDN=myName
BINDPW=mySecretPassword
BINDPW.type=password

Host: example.com
Accept: application/json
```

`GET /system/gpgkeys`

Returns the GPG keys in the config directory specified by PI_GNUPG_HOME.

Return A json list of the public GPG keys

`GET /system/random`

This endpoint can be used to retrieve random keys from privacyIDEA. In certain cases the client might need random data to initialize tokens on the client side. E.g. the command line client when initializing the yubikey or the WebUI when creating Client API keys for the yubikey.

In this case, privacyIDEA can create the random data/keys.

Query Parameters

- **`len`** – The length of a symmetric key (byte)
- **`encode`** – The type of encoding. Can be "hex" or "b64".

Return key material

POST /system/hsm
Set the password for the security module

GET /system/hsm
Get the status of the security module.

GET /system/(*key*)

GET /system/
This endpoint either returns all config entries or only the value of the one config key.

This endpoint can be called by the administrator but also by the normal user, so that the normal user gets necessary information about the system config

Parameters

- **key** – The key to return.

Return A json response or a single value, when queried with a key.

Rtype json or scalar

POST /system/test/(*tokentype*)
The call /system/test/email tests the configuration of the email token.

DELETE /system/(*key*)
delete a configuration key

JSON Parameters

- **key** – configuration key name

Returns a json result with the deleted value

Resolver endpoints

The code of this module is tested in tests/test_api_system.py

POST /resolver/test
Send the complete parameters of a resolver to the privacyIDEA server to test, if these settings will result in a successful connection. If you are testing existing resolvers, you can send the "__CENSORED__" password. privacyIDEA will use the already stored password from the database.

Return a json result with True, if the given values can create a working resolver and a description.

GET /resolver/(*resolver*)

GET /resolver/
returns a json list of the specified resolvers. The passwords of resolvers (e.g. Bind PW of the LDAP resolver or password of the SQL resolver) will be returned as "__CENSORED__". You can run a POST request to update the data and privacyIDEA will ignore the "__CENSORED__" password or you can even run a testresolver.

Parameters

- **resolver** (`str`) – the name of the resolver
- **type** (`str`) – Only return resolvers of type (like passwdresolver..)
- **editable** (`str`) – Set to "1" if only editable resolvers should be returned.

Return a json result with the configuration of resolvers

POST /resolver/(*resolver*)

This creates a new resolver or updates an existing one. A resolver is uniquely identified by its name.

If you update a resolver, you do not need to provide all parameters. Parameters you do not provide are left untouched. When updating a resolver you must not change the type! You do not need to specify the type, but if you specify a wrong type, it will produce an error.

Parameters

- **resolver** (*str*) – the name of the resolver.
- **type** – the type of the resolver. Valid types are passwdresolver,

ldapresolver, sqlresolver, scimresolver :type type: str :return: a json result with the value being the database id (>0)

Additional parameters depend on the resolver type.

LDAP:

- LDAPURI
- LDAPBASE
- BINDDN
- BINDPW
- TIMEOUT
- SIZELIMIT
- LOGINNAMEATTRIBUTE
- LDAPSEARCHFILTER
- LDAPFILTER
- USERINFO
- NOREFERRALS - True|False
- EDITABLE - True|False

SQL:

- Database
- Driver
- Server
- Port
- User
- Password
- Table
- Map

Passwd

- Filename

DELETE /resolver/(*resolver*)

This function deletes an existing resolver. A resolver can not be deleted, if it is contained in a realm

Parameters

- **resolver** – the name of the resolver to delete.

Return json with success or fail

Realm endpoints

The realm endpoints are used to define realms. A realm groups together many users. Administrators can manage the tokens of the users in such a realm. Policies and tokens can be assigned to realms.

A realm consists of several resolvers. Thus you can create a realm and gather users from LDAP and flat file source into one realm or you can pick resolvers that collect users from different points from your vast LDAP directory and group these users into a realm.

You will only be able to see and use user object, that are contained in a realm.

The code of this module is tested in tests/test_api_system.py

GET /realm/superuser

This call returns the list of all superuser realms as they are defined in *pi.cfg*. See *The Config File* for more information about this.

Return a json result with a list of realms

Example request:

```
GET /superuser HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": ["superuser",
              "realm2"]
    }
  },
  "version": "privacyIDEA unknown"
}
```

GET /realm/

This call returns the list of all defined realms. It takes no arguments.

Return a json result with a list of realms

Example request:

```
GET / HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": {
      "realm1_with_resolver": {
        "default": true,
        "resolver": [
          {
            "name": "reso1_with_realm",
            "type": "passwdresolver"
          }
        ]
      }
    }
  },
  "version": "privacyIDEA unknown"
}
```

POST /realm/(*realm*)

This call creates a new realm or reconfigures a realm. The realm contains a list of resolvers.

In the result it returns a list of added resolvers and a list of resolvers, that could not be added.

Parameters

- **realm** – The unique name of the realm
- **resolvers** (*string or list*) – A comma separated list of unique resolver names or a list object
- **priority** – Additional parameters priority.<resolvername> define the priority of the resolvers within this realm.

Return a json result with a list of Realms

Example request:

To create a new realm "newrealm", that consists of the resolvers "reso1_with_realm" and "reso2_with_realm" call:

```
POST /realm/newrealm HTTP/1.1
Host: example.com
Accept: application/json
Content-Length: 26
Content-Type: application/x-www-form-urlencoded

resolvers=reso1_with_realm, reso2_with_realm
priority.reso1_with_realm=1
priority.reso2_with_realm=2
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

```

(continues on next page)

(continued from previous page)

```
{
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
             "status": true,
             "value": {
                 "added": ["reso1_with_realm", "reso2_with_realm"],
                 "failed": []
             }
   }
   "version": "privacyIDEA unknown"
}
```

`DELETE /realm/`(*realm*)

This call deletes the given realm.

Parameters

- **`realm`** – The name of the realm to delete

Return a json result with value=1 if deleting the realm was successful

Example request:

```
DELETE /realm/realm_to_delete HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

 {
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": 1
   },
   "version": "privacyIDEA unknown"
 }
```

Default Realm endpoints

These endpoints are used to define the default realm, retrieve it and delete it.

`DELETE /defaultrealm`

This call deletes the default realm.

Return a json result with either 1 (success) or 0 (fail)

Example response:

```
 {
  "id": 1,
  "jsonrpc": "2.0",
```

(continues on next page)

(continued from previous page)

```
  "result": {
    "status": true,
    "value": 1
  },
  "version": "privacyIDEA unknown"
}
```

GET /defaultrealm

This call returns the default realm

Return a json description of the default realm with the resolvers

Example response:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": {
      "defrealm": {
        "default": true,
        "resolver": [
          {
            "name": "defresolver",
            "type": "passwdresolver"
          }
        ]
      }
    }
  },
  "version": "privacyIDEA unknown"
}
```

POST /defaultrealm/(*realm*)

This call sets the default realm.

Parameters

- **realm** – the name of the realm, that should be the default realm

Return a json result with either 1 (success) or 0 (fail)

Token endpoints

The token API can be accessed via /token.

You need to authenticate to gain access to these token functions. If you are authenticated as administrator, you can manage all tokens. If you are authenticated as normal user, you can only manage your own tokens. Some API calls are only allowed to be accessed by admintrators.

To see how to authenticate read *Authentication endpoints*.

POST /token/setrandompin/(*serial*)

POST /token/setrandompin

Set the OTP PIN for a specific token to a random value.

The token is identified by the unique serial number.

JSON Parameters

- **`serial`** (*`basestring`*) – the serial number of the single token to reset

Return In "value" returns the number of PINs set. The detail-section contains the key "pin" with the set PIN.

Rtype json object

`POST /token/description/`(*serial*)

`POST /token/description`

This endpoint can be used by the user or by the admin to set the description of a token.

JSON Parameters

- **`description`** (*`basestring`*) – The description for the token

Parameters

- **`serial`** –

Return

`GET /token/challenges/`(*serial*)

`GET /token/challenges/`

This endpoint returns the active challenges in the database or returns the challenges for a single token by its serial number

Query Parameters

- **`serial`** – The optional serial number of the token for which the challenges should be returned
- **`sortby`** – sort the output by column
- **`sortdir`** – asc/desc
- **`page`** – request a certain page
- **`pagesize`** – limit the number of returned tokens
- **`transaction_id`** – only returns challenges for this transaction_id. This is useful when working with push or tiqr tokens.

Return json

`POST /token/unassign`

Unssign a token from a user. You can either provide "serial" as an argument to unassign this very token or you can provide user and realm, to unassign all tokens of a user.

Return In case of success it returns the number of unassigned tokens in "value".

Rtype JSON object

`POST /token/copyuser`

Copy the token user from one token to the other.

JSON Parameters

- **`from`** (*`basestring`*) – the serial number of the token, from where you want to copy the pin.
- **`to`** (*`basestring`*) – the serial number of the token, from where you want to copy the pin.

Return returns value=True in case of success

Rtype bool

POST /token/disable/(*serial*)

POST /token/disable

Disable a single token or all the tokens of a user either by providing the serial number of the single token or a username and realm.

Disabled tokens can not be used to authenticate but can be enabled again.

JSON Parameters

- **serial** (*basestring*) – the serial number of the single token to disable
- **user** (*basestring*) – The login name of the user
- **realm** (*basestring*) – the realm name of the user

Return In case of success it returns the number of disabled tokens in "value".

Rtype json object

POST /token/copypin

Copy the token PIN from one token to the other.

JSON Parameters

- **from** (*basestring*) – the serial number of the token, from where you want to copy the pin.
- **to** (*basestring*) – the serial number of the token, from where you want to copy the pin.

Return returns value=True in case of success

Rtype bool

POST /token/assign

Assign a token to a user.

JSON Parameters

- **serial** – The token, which should be assigned to a user
- **user** – The username of the user
- **realm** – The realm of the user

Return In case of success it returns "value": True.

Rtype json object

POST /token/revoke/(*serial*)

POST /token/revoke

Revoke a single token or all the tokens of a user. A revoked token will usually be locked. A locked token can not be used anymore. For certain token types additional actions might occur when revoking a token.

JSON Parameters

- **serial** (*basestring*) – the serial number of the single token to revoke
- **user** (*basestring*) – The login name of the user
- **realm** (*basestring*) – the realm name of the user

Return In case of success it returns the number of revoked tokens in "value".

Rtype JSON object

POST /token/enable/(*serial*)

POST /token/enable

Enable a single token or all the tokens of a user.

JSON Parameters

- **serial** (*basestring*) – the serial number of the single token to enable
- **user** (*basestring*) – The login name of the user
- **realm** (*basestring*) – the realm name of the user

Return In case of success it returns the number of enabled tokens in "value".

Rtype json object

POST /token/resync/(*serial*)

POST /token/resync

Resync the OTP token by providing two consecutive OTP values.

JSON Parameters

- **serial** (*basestring*) – the serial number of the single token to reset
- **otp1** (*basestring*) – First OTP value
- **otp2** (*basestring*) – Second OTP value

Return In case of success it returns "value"=True

Rtype json object

POST /token/setpin/(*serial*)

POST /token/setpin

Set the the user pin or the SO PIN of the specific token. Usually these are smartcard or token specific PINs. E.g. the userpin is used with mOTP tokens to store the mOTP PIN.

The token is identified by the unique serial number.

JSON Parameters

- **serial** (*basestring*) – the serial number of the single token to reset
- **userpin** (*basestring*) – The user PIN of a smartcard
- **sopin** (*basestring*) – The SO PIN of a smartcard
- **otppin** (*basestring*) – The OTP PIN of a token

Return In "value" returns the number of PINs set.

Rtype json object

POST /token/reset/(*serial*)

POST /token/reset

Reset the failcounter of a single token or of all tokens of a user.

JSON Parameters

- **serial** (*basestring*) – the serial number of the single token to reset
- **user** (*basestring*) – The login name of the user
- **realm** (*basestring*) – the realm name of the user

Return In case of success it returns "value"=True

Rtype json object

POST /token/init

create a new token.

JSON Parameters

- **otpkey** – required: the secret key of the token
- **genkey** – set to =1, if key should be generated. We either need otpkey or genkey
- **keysize** – the size (byte) of the key. Either 20 or 32. Default is 20
- **serial** – the serial number/identifier of the token
- **description** – A description for the token
- **pin** – the pin of the token. “OTP PIN”
- **user** – the login user name. This user gets the token assigned
- **realm** – the realm of the user.
- **type** – the type of the token
- **tokenrealm** – additional realms, the token should be put into
- **otplen** – length of the OTP value
- **hashlib** – used hashlib sha1, sha256 or sha512
- **validity_period_start** – The beginning of the validity period
- **validity_period_end** – The end of the validity period
- **2stepinit** – set to =1 in conjunction with genkey=1 if you want a 2 step initialization process. Additional policies have to be set see *Two Step Enrollment*.
- **otpkeyformat** – used to supply the OTP key in alternate formats, currently hex or base32check (see *Two Step Enrollment*)

Return a json result with a boolean “result”: true

Depending on the token type there can be additional parameters. In the tokenclass you can see additional parameters in the method `update` when looking for `getParam` functions.

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
   "detail": {
     "googleurl": {
       "description": "URL for google Authenticator",
       "img": "<img    width=250   src="data:image/png;base64,
↪iVBORw0KGgoAAAANSUhEUgAAAcIAAAHCAQAAAABUY/
↪ToAAADsUlEQVR4nO2czY3bMBCF34QCfKSALcClyB2kpCAlpQOxlBQQgDwaoPBy4I+p9W4OSRaWF28OgizxgylgMJw/
↪0oi/k/
↪DlL0FApEiRIkWKFCnyeKRVmdrjNAFh3srTMuSS2qjLg2cr8pDkQpKMgF3SBITz1QA4YolVfQA4kiT35CNmK/
↪JQZLM8aQaWH+3pEkEgTZlhBojksgGAAS7/83+K/ORkOF/
↪NLtismiCfYXbOd+AxZivygCTXdCLCDJRLfTbhTo4wW5FHIJtyeAJIAJb4AobLBIP/
↪ZQRAwMcyakxIPtd3ivw4EqObXJzody9t1EKS63N9p8iPI4sO3QTwGSSbA1Q0x+cWunWRDolsUjSnxvau6VB0xMIMrp
↪i6WtedIhkXupS1MEsMRmaVafh7dVfXwGV0D+kMj3yXDOsIsngXQiV59R0tZIE7jC0b4VA3WE2Yo8CtkTPy7b8sPA8H
↪xOkR9B4maCbnF8c53vHGuuLVaTHRLZpBgYgweAVP0hLPElA+mFtVrvf3W/
↪aTM+brYij0j23o8JthAweNc1J5cCmSFNYDCAS5wfOVuRRyT7QpVL9F6XLN/
↪zjhG4ZSAHj1trmcgmLcfoWoq6/
↪B4LZLeqBxmVpxb5WobYfl8vaxfU7DSA4mdLh0S+TW5W2xXTiaWZ0WbALqiXmi5KU/
↪n5tN8p8r+TzaqUH936MKNW6/2uIkvZIZF/IEleDfAZZnYi1zSB/
↪DmVpa2YJZtVLxP5JmnfWCutty5qwNcFrWSsV2xGxs3+03+K/
↪Cxk74WtTWflDr652L0XtoZuylOLvJNb9H7XPzQ0DOX9RTokcpAhAzRYpN4LO5TsI1rQLx0SOci4z7VcSuvQZgxWX1g
↪dTe9U6RL6WtoIBqDs33NA7Xdey3SYzrWUi99L8IfJW4cC4pYNjg+Ow/
↪+O5vlPkx5OpnSsUzler2cbS29g8pmBmWH6elGMU+UqaFwS0NBBa9O45Rmhr26Mof0jkTt440MNlC9aOGQqzA8McaQs
```

(continues on next page)

(continued from previous page)

```
      "value": "otpauth://hotp/mylabel?
↪secret=GEZDGNBVGY3TQOJQGEZDGNBVGY3TQOJQ&counter=0"
    },
    "oathurl": {
      "description": "URL for OATH token",
      "img": "<img    width=250   src="data:image/png;base64,
↪iVBORw0KGgoAAAANSUhEUgAAAcIAAAHCAQAAAABUY/
↪ToAAADfElEQVR4nO2cTYrjMBCFX40EvZRvkKPIN5gz9c3so/
↪QBBqxlwObNQpIlp2cYaBI6zrxamDjyhywo6leyEV+T+ccXQUCkSJEiRYoUKfL5SCviy7+zmZWBAbARmwGpPjXeZU6F
↪MEBeAU/JoA52pOuk6Rd6f9H/
↪60xBWbwCMyG7Mg0j3mlPky5OOiB9v5AQACCQnONr4yDlFnpisdigQQAIM4WpE2oyAWy0umyfCku1QX5A81zpFPo5EH
↪J4rc+So+++S2zy1ofDVezMXmURtoZ1ynyEeRuh1xXSiwJPtCFRyUygupDIm+l5fa9Q+Na0rT8yCG3lw6JPEqtMZaCU
↪DK0cDWBXqapczY0ptxd5kFZjLEqzlJi6C4WyHYJjHZAOieyk2aGsSNyjoF2l0Jsg9TpE/
↪oVMHpgvK8wupRZkIwDMQy0S5QMfbVfsOdcp8v5kF1M3N9ZaGrX/sbf2g+yQyFtpPdW2/
↪75pTtGX5tWCcnuRt9L1OtguLcFve9DazmrpkMheOn3Ju4aA4tX6gVopiurbi7yV3Lc3IJ+vh0VuHoBbAWyeSH41hF+
↪J7z3UzG8PVSlqfPMrlm99W5FOSsUY8Noarmdkb+T7UTSF7Wv8kbyvyqcguL+u23k/
↪7cDvdmm9Vpxb5LzLbobErObbc/
↪lFzijw3eZtvcR4WAtjKx2Lmn1djztBAWN5ZPX3X24p8RrI719HcWNnsEVoz1vWPyJeJ7KXYoTln7A4Wcz6/
↪eQL7xxxyRr95IlwNskMiezF941ykSJEiRYoU+Z+TvwF49nApsKFZZAAAAABJRU5ErkJggg==
↪"/>",
      "value": "oathtoken:///addToken?name=mylabel&lockdown=true&
↪key=3132333435363738393031323334353637383930"
    },
    "otpkey": {
      "description": "OTP seed",
      "img": "<img    width=200   src="data:image/png;base64,
↪iVBORw0KGgoAAAANSUhEUgAAAUoAAAFKAQAAAABTUiuoAAAB70lEQVR4nO2aTY6jQAyFPw9IWYI0B+ijwNHhKH0DWI
↪CXkQkfIsnWRU/22ViZ4x/9pIQaKCBBhpooEeilqPGrAWzdjGYy8/
↪94QICfQftJEkTAIsBlYBKkqSf6DECAn0HnfMRkj4fnjfrATOrzxEQ6I6oX74bYGJuzxIQ6H9kqySqSjCfISDQX6CNp
↪19B9PgQsbgnPEBDonrCXyZMB/HMaFZOnu6DWz2aMZqaBZ79Vw9gu0W/
↪dBsU7qm4CL16aKq9geonhcq2BlqR4jirRSYImoaF8eO8c2boeXR38YnRavIwJkNFUsg1xudZAy5ywreSFyqcabgxr8
↪Ao2AJtXAYoIEYzsVi3i51kBz3Rq8O658RFhKVn4Rdesu6MYTemZoEm468kh+TejlWgNdjXoeMGVjOJXXnVJk6zboa1
↪"/>",
      "value": "seed://3132333435363738393031323334353637383930"
    },
    "serial": "OATH00096020"
  },
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": true
  },
  "version": "privacyIDEA unknown"
}
```

2 Step Enrollment

Some tokens might need a 2 step initialization process like a smartphone app. This way you can create a shared secret from a part generated by the privacyIDEA server and from a second part generated by the smartphone app/client.

The first API call would be

```
POST /token/init

2stepinit=1
```

The response would contain the otpkey generated by the server and the serial number of the token. At this point, the token is deactivated and marked as being in an enrollment state. The client would also generated a component of the key and send his component to the privacyIDEA server:

The second API call would be

```
POST /token/init

serial=<serial from the previous response>
otpkey=<key part generated by the client>
```

Each tokenclass can define its own way to generate the secret key by overwriting the method `generate_symmetric_key`. The Base Tokenclass contains an extremely simple way by concatenating the two parts. See *`generate_symmetric_key()`*

`POST /token/set/`(*serial*)

`POST /token/set`

This API is only to be used by the admin! This can be used to set token specific attributes like

- description
- count_window
- sync_window
- count_auth_max
- count_auth_success_max
- hashlib,
- max_failcount
- validity_period_start
- validity_period_end

The token is identified by the unique serial number or by the token owner. In the later case all tokens of the owner will be modified.

The validity period needs to be provided in the format YYYY-MM-DDThh:mm+oooo

JSON Parameters

- **`serial`** (*`basestring`*) – the serial number of the single token to reset
- **`user`** (*`basestring`*) – The username of the token owner
- **`realm`** (*`basestring`*) – The realm name of the token owner

Return returns the number of attributes set in "value"

Rtype json object

`GET /token/`

Display the list of tokens. Using different parameters you can choose, which tokens you want to get and also in which format you want to get the information (*outform*).

Query Parameters

- **`serial`** – Display the token data of this single token. You can do a not strict matching by specifying a serial like "*OATH*".
- **`type`** – Display only token of type. You ca do a non strict matching by specifying a tokentype like "*otp*", to file hotp and totp tokens.

- **user** – display tokens of this user
- **tokenrealm** – takes a realm, only the tokens in this realm will be displayed
- **description** (*basestring*) – Display token with this kind of description
- **sortby** – sort the output by column
- **sortdir** – asc/desc
- **page** – request a certain page
- **assigned** – Only return assigned (True) or not assigned (False) tokens
- **pagesize** – limit the number of returned tokens
- **user_fields** – additional user fields from the userid resolver of the owner (user)
- **outform** – if set to "csv", than the token list will be given in CSV

Return a json result with the data being a list of token dictionaries:

```
{ "data": [ { <token1> }, { <token2> } ]}
```

Rtype json

GET /token/getserial/(*otp*)

Get the serial number for a given OTP value. If the administrator has a token, he does not know to whom it belongs, he can type in the OTP value and gets the serial number of the token, that generates this very OTP value.

Query Parameters

- **otp** – The given OTP value
- **type** – Limit the search to this token type
- **unassigned** – If set=1, only search in unassigned tokens
- **assigned** – If set=1, only search in assigned tokens
- **count** – if set=1, only return the number of tokens, that will be searched
- **serial** – This can be a substring of serial numbers to search in.
- **window** – The number of OTP look ahead (default=10)

Return The serial number of the token found

POST /token/realm/(*serial*)

Set the realms of a token. The token is identified by the unique serial number

You can call the function like this: POST /token/realm?serial=<serial>&realms=<something> POST /token/realm/<serial>?realms=<hash>

JSON Parameters

- **serial** (*basestring*) – the serial number of the single token to reset
- **realms** (*basestring*) – The realms the token should be assigned to. Comma separated

Return returns value=True in case of success

Rtype bool

POST /token/info/(*serial*)**/**

key Add a specific tokeninfo entry to a token. Already existing entries with the same key are overwritten.

Parameters

- **serial** – the serial number/identifier of the token
- **key** – token info key that should be set

Query Parameters

- **value** – token info value that should be set

Return returns value=True in case the token info could be set

Rtype bool

DELETE /token/info/(*serial*)**/**

key Delete a specific tokeninfo entry of a token.

Parameters

- **serial** – the serial number/identifier of the token
- **key** – token info key that should be deleted

Return returns value=True in case a matching token was found, which does not necessarily mean

that the matching token had a tokeninfo value set in the first place. :rtype: bool

POST /token/load/(*filename*)

The call imports the given file containing token definitions. The file can be an OATH CSV file, an aladdin XML file or a Yubikey CSV file exported from the yubikey initialization tool.

The function is called as a POST request with the file upload.

JSON Parameters

- **filename** – The name of the token file, that is imported
- **type** – The file type. Can be "aladdin-xml", "oathcsv" or "yubikeycsv".
- **tokenrealms** – comma separated list of realms.
- **psk** – Pre Shared Key, when importing PSKC

Return The number of the imported tokens

Rtype int

POST /token/lost/(*serial*)

Mark the specified token as lost and create a new temporary token. This new token gets the new serial number "lost<old-serial>" and a certain validity period and the PIN of the lost token.

This method can be called by either the admin or the user on his own tokens.

You can call the function like this: POST /token/lost/serial

JSON Parameters

- **serial** (*basestring*) – the serial number of the lost token.

Return returns value=dictionary in case of success

Rtype bool

DELETE /token/(*serial*)

Delete a token by its serial number.

JSON Parameters

- **serial** – The serial number of a single token.

Return In case of success it return the number of deleted tokens in “value”

Rtype json object

User endpoints

The user endpoints is a subset of the system endpoint.

GET /user/

list the users in a realm

A normal user can call this endpoint and will get information about his own account.

Parameters

- **realm** – a realm that contains several resolvers. Only show users from this realm
- **resolver** – a distinct resolvername
- **<searchexpr>** – a search expression, that depends on the ResolverClass

Return json result with “result”: true and the userlist in “value”.

Example request:

```
GET /user?realm=realm1 HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

 {
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": [
       {
         "description": "Cornelius Kölbel,,+49 151 2960 1417,+49 561 3166797,
↪cornelius.koelbel@netknights.it",
         "email": "cornelius.koelbel@netknights.it",
         "givenname": "Cornelius",
         "mobile": "+49 151 2960 1417",
         "phone": "+49 561 3166797",
         "surname": "Kölbel",
         "userid": "1009",
         "username": "cornelius",
         "resolver": "name-of-resolver"
       }
     ]
   },
   "version": "privacyIDEA unknown"
 }
```

POST /user/

POST /user

Create a new user in the given resolver.

Example request:

```
POST /user
user=new_user
resolver=<resolvername>
surname=...
givenname=...
email=...
mobile=...
phone=...
password=...
description=...

Host: example.com
Accept: application/json
```

PUT /user/

PUT /user

Edit a user in the user store. The resolver must have the flag editable, so that the user can be deleted. Only administrators are allowed to edit users.

Example request:

```
PUT /user
user=existing_user
resolver=<resolvername>
surname=...
givenname=...
email=...
mobile=...
phone=...
password=...
description=...

Host: example.com
Accept: application/json
```

Note: Also a user can call this function to e.g. change his password. But in this case the parameter “user” and “resolver” get overwritten by the values of the authenticated user, even if he specifies another username.

DELETE /user/(*resolvername*)**/**

username Delete a User in the user store. The resolver must have the flag editable, so that the user can be deleted. Only administrators are allowed to delete users.

Delete a user object in a user store by calling

Example request:

```
DELETE /user/<resolvername>/<username>
Host: example.com
Accept: application/json
```

The code of this module is tested in tests/test_api_system.py

Policy endpoints

The policy endpoints are a subset of the system endpoint.

You can read more about policies at *Policies*.

GET /policy/check

This function checks, if the given parameters would match a defined policy or not.

Query Parameters

- **user** – the name of the user
- **realm** – the realm of the user or the realm the administrator want to do administrative tasks on.
- **resolver** – the resolver of a user
- **scope** – the scope of the policy
- **action** – the action that is done - if applicable
- **client** (*IP_Address*) – the client, from which this request would be issued

Return a json result with the keys allowed and policy in the value key

Rtype json

Status Codes

- 200 OK – Policy created or modified.
- 401 Unauthorized – Authentication failed

Example request:

```
GET /policy/check?user=admin&realm=r1&client=172.16.1.1 HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.0 200 OK
Content-Type: application/json

 {
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": {
       "pol_update_del": {
         "action": "enroll",
         "active": true,
         "client": "172.16.0.0/16",
         "name": "pol_update_del",
         "realm": "r1",
         "resolver": "test",
         "scope": "selfservice",
         "time": "",
         "user": "admin"
       }
     }
```

(continues on next page)

(continued from previous page)

```
  },
  "version": "privacyIDEA unknown"
}
```

GET /policy/defs/(*scope*)

GET /policy/defs

This is a helper function that returns the POSSIBLE policy definitions, that can be used to define your policies.

If the given scope is "conditions", this returns a dictionary with the following keys:

- `"sections"`, containing a dictionary mapping each condition section name to a dictionary with the following keys:
  - `"description"`, a human-readable description of the section
- **`"comparators"`, containing a dictionary mapping each comparator to a dictionary with the following keys:**
  - `"description"`, a human-readable description of the comparator

Query Parameters

- **scope** – if given, the function will only return policy definitions for the given scope.

Return The policy definitions of the allowed scope with the actions and action types. The top level key is the scope.

Rtype dict

GET /policy/export/(*export*)

GET /policy/(*name*)

GET /policy/

this function is used to retrieve the policies that you defined. It can also be used to export the policy to a file.

Query Parameters

- **name** – will only return the policy with the given name
- **export** – The filename needs to be specified as the third part of the URL like policy.cfg. It will then be exported to this file.
- **realm** – will return all policies in the given realm
- **scope** – will only return the policies within the given scope
- **active** – Set to true or false if you only want to display active or inactive policies.

Return a json result with the configuration of the specified policies

Rtype json

Status Codes

- 200 OK – Policy created or modified.
- 401 Unauthorized – Authentication failed

Example request:

In this example a policy "pol1" is created.

```
GET /policy/pol1 HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.0 200 OK
Content-Type: application/json

 {
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": {
       "pol_update_del": {
         "action": "enroll",
         "active": true,
         "client": "1.1.1.1",
         "name": "pol_update_del",
         "realm": "r1",
         "resolver": "test",
         "scope": "selfservice",
         "time": "",
         "user": "admin"
       }
     }
   },
   "version": "privacyIDEA unknown"
 }
```

POST /policy/disable/ (*name*)

Disable a given policy by its name.

JSON Parameters

- **name** – The name of the policy

Return ID in the database

POST /policy/enable/ (*name*)

Enable a given policy by its name.

JSON Parameters

- **name** – Name of the policy

Return ID in the database

POST /policy/import/ (*filename*)

This function is used to import policies from a file.

JSON Parameters

- **filename** – The name of the file in the request

Form Parameters

- **file** – The uploaded file contents

Return A json response with the number of imported policies.

Status Codes

- 200 OK – Policy created or modified.
- 401 Unauthorized – Authentication failed

Example request:

```
POST /policy/import/backup-policy.cfg HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.0 200 OK
Content-Type: application/json

 {
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": 2
   },
   "version": "privacyIDEA unknown"
 }
```

POST /policy/(*name*)

Creates a new policy that defines access or behaviour of different actions in privacyIDEA

JSON Parameters

- **name** (*basestring*) – name of the policy
- **scope** – the scope of the policy like "admin", "system", "authentication" or "selfservice"
- **adminrealm** – Realm of the administrator. (only for admin scope)
- **adminuser** – Username of the administrator. (only for admin scope)
- **action** – which action may be executed
- **realm** – For which realm this policy is valid
- **resolver** – This policy is valid for this resolver
- **user** – The policy is valid for these users. string with wild cards or list of strings
- **time** – on which time does this policy hold
- **client** (*IP address with subnet*) – for which requesting client this should be
- **active** – bool, whether this policy is active or not
- **check_all_resolvers** – bool, whether all all resolvers in which the user exists should be checked with this policy.
- **conditions** – a (possibly empty) list of conditions of the policy. Each condition is encoded as a list with 5 elements: `[section (string), key (string), comparator (string), value (string), active (boolean)]` Hence, the `conditions` parameter expects a list of lists. When privacyIDEA checks if a defined policy should take effect, *all* conditions of the policy must be fulfilled for the policy to match. Note that the order of conditions is not guaranteed to be preserved.

Return a json result with success or error

Status Codes

- 200 OK – Policy created or modified.
- 401 Unauthorized – Authentication failed

Example request:

In this example a policy "pol1" is created.

```
POST /policy/pol1 HTTP/1.1
Host: example.com
Accept: application/json

scope=admin
realm=realm1
action=enroll, disable
```

The policy POST request can also take the parameter of conditions. This is a list of conditions sets: [["userinfo", "memberOf", "equals", "groupA", "true"], [...]] With the entries being the `section`, the `key`, the `comparator`, the `value` and `active`. For more on conditions see *Policy conditions*.

Example response:

```
HTTP/1.0 200 OK
Content-Length: 354
Content-Type: application/json

 {
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": {
       "setPolicy pol1": 1
     }
   },
   "version": "privacyIDEA unknown"
 }
```

DELETE /policy/(*name*)

This deletes the policy of the given name.

JSON Parameters

- **name** – the policy with the given name

Return a json result about the delete success. In case of success value > 0

Status Codes

- 200 OK – Policy created or modified.
- 401 Unauthorized – Authentication failed

Example request:

In this example a policy "pol1" is created.

```
DELETE /policy/pol1 HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.0 200 OK
Content-Type: application/json

{
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": 1
   },
   "version": "privacyIDEA unknown"
}
```

Event endpoints

This endpoint is used to create, modify, list and delete Event Handling Configuration. Event handling configuration is stored in the database table "eventhandling"

The code of this module is tested in tests/test_api_events.py

GET /event/ (*eventid*)

GET /event/

returns a json list of the event handling configuration

Or

returns a list of available events when calling as /event/available

Or

the available handler modules when calling as /event/handlermodules

POST /event

This creates a new event handling definition

Parameters

- **name** – A describing name of the event.bool
- **id** – (optional) when updating an existing event you need to specify the id
- **event** – A comma separated list of events
- **handlermodule** – A handlermodule
- **action** – The action to perform
- **ordering** – An integer number
- **position** – "pre" or "post"
- **conditions** – Conditions, when the event will trigger
- **options.** – A list of possible options.

GET /event/conditions/ (*handlermodule*)

Return the list of conditions a handlermodule provides.

Parameters

- **handlermodule** – Identifier of the handler module like "UserNotification"

Return list oft actions

GET /event/positions/ (*handlermodule*)

Return the list of positions a handlermodule provides.

Parameters

- **handlermodule** – Identifier of the handler module like "UserNotification"

Return list oft actions

GET /event/actions/ (*handlermodule*)

Return the list of actions a handlermodule provides.

Parameters

- **handlermodule** – Identifier of the handler module like "UserNotification"

Return list oft actions

POST /event/disable/ (*eventid*)

Disable a given policy by its name.

JSON Parameters

- **name** – The name of the policy

Return ID in the database

POST /event/enable/ (*eventid*)

Enable a given event by its id.

JSON Parameters

- **eventid** – ID of the event

Return ID in the database

DELETE /event/ (*eid*)

this function deletes an existing event handling configuration

Parameters

- **eid** – The id of the event handling configuration

Return json with success or fail

This endpoint is used to create, modify, list and delete Machine Resolvers. Machine Resolvers fetch machine information from remote machine stores like a hosts file or an Active Directory.

The code of this module is tested in tests/test_api_machineresolver.py

Machine Resolver endpoints

POST /machineresolver/test

This function tests, if the given parameter will create a working machine resolver. The Machine Resolver Class itself verifies the functionality. This can also be network connectivity to a Machine Store.

Return a json result with bool

GET /machineresolver/

returns a json list of all machine resolver.

Parameters

- **type** – Only return resolvers of type (like "hosts"…)

POST /machineresolver/(*resolver*)

This creates a new machine resolver or updates an existing one. A resolver is uniquely identified by its name.

If you update a resolver, you do not need to provide all parameters. Parameters you do not provide are left untouched. When updating a resolver you must not change the type! You do not need to specify the type, but if you specify a wrong type, it will produce an error.

Parameters

- **resolver** (*basestring*) – the name of the resolver.
- **type** (*string*) – the type of the resolver. Valid types are… "hosts"

Return a json result with the value being the database id (>0)

Additional parameters depend on the resolver type.

hosts:

- filename

DELETE /machineresolver/(*resolver*)

this function deletes an existing machine resolver

Parameters

- **resolver** – the name of the resolver to delete.

Return json with success or fail

GET /machineresolver/(*resolver*)

This function retrieves the definition of a single machine resolver.

Parameters

- **resolver** – the name of the resolver

Return a json result with the configuration of a specified resolver

This REST API is used to list machines from Machine Resolvers.

The code is tested in tests/test_api_machines

Machine endpoints

POST /machine/tokenoption

This sets a Machine Token option or deletes it, if the value is empty.

Parameters

- **hostname** – identify the machine by the hostname
- **machineid** – identify the machine by the machine ID and the resolver name
- **resolver** – identify the machine by the machine ID and the resolver name
- **serial** – identify the token by the serial number
- **application** – the name of the application like "luks" or "ssh".

Parameters not listed will be treated as additional options.

Return

GET /machine/authitem/(*application*)

GET /machine/authitem

This fetches the authentication items for a given application and the given client machine.

Parameters

- **challenge** (*basestring*) – A challenge for which the authentication item is calculated. In case of the Yubikey this can be a challenge that produces a response. The authentication item is the combination of the challenge and the response.
- **hostname** (*basestring*) – The hostname of the machine

Return dictionary with lists of authentication items

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

 {
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": { "ssh": [ { "username": "....",
                           "sshkey": "...."
                         }
                       ],
                "luks": [ { "slot": ".....",
                            "challenge": "...",
                            "response": "...",
                            "partition": "..."
                        ]
              }
   },
   "version": "privacyIDEA unknown"
 }
```

POST /machine/token

Attach an existing token to a machine with a certain application.

Parameters

- **hostname** – identify the machine by the hostname
- **machineid** – identify the machine by the machine ID and the resolver name
- **resolver** – identify the machine by the machine ID and the resolver name
- **serial** – identify the token by the serial number
- **application** – the name of the application like “luks” or “ssh”.

Parameters not listed will be treated as additional options.

Return json result with “result”: true and the machine list in “value”.

Example request:

```
POST /token HTTP/1.1
Host: example.com
Accept: application/json

{ "hostname": "puckel.example.com",
```

(continues on next page)

(continued from previous page)

```
  "machienid": "12313098",
  "resolver": "machineresolver1",
  "serial": "tok123",
  "application": "luks" }
```

GET /machine/token

Return a list of MachineTokens either for a given machine or for a given token.

Parameters

- **serial** – Return the MachineTokens for a the given Token
- **hostname** – Identify the machine by the hostname
- **machineid** – Identify the machine by the machine ID and the resolver name
- **resolver** – Identify the machine by the machine ID and the resolver name

Return

GET /machine/

List all machines that can be found in the machine resolvers.

Parameters

- **hostname** – only show machines, that match this hostname as substring
- **ip** – only show machines, that exactly match this IP address
- **id** – filter for substring matching ids
- **resolver** – filter for substring matching resolvers
- **any** – filter for a substring either matching in “hostname”, “ip” or “id”

Return json result with “result”: true and the machine list in “value”.

Example request:

```
GET /hostname?hostname=on HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

 {
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": [
       {
         "id": "908asljdas90ad0",
         "hostname": [ "flavon.example.com", "test.example.com" ],
         "ip": "1.2.3.4",
         "resolver_name": "machineresolver1"
       },
       {
         "id": "1908209x48x2183",
```

(continues on next page)

(continued from previous page)

```
        "hostname": [ "london.example.com" ],
        "ip": "2.4.5.6",
        "resolver_name": "machineresolver1"
      }
    ]
  },
  "version": "privacyIDEA unknown"
}
```

DELETE /machine/token/(*serial*)**/***machineid***/***resolver***/***application*
Detach a token from a machine with a certain application.

Parameters

- **machineid** – identify the machine by the machine ID and the resolver name
- **resolver** – identify the machine by the machine ID and the resolver name
- **serial** – identify the token by the serial number
- **application** – the name of the application like "luks" or "ssh".

Return json result with "result": true and the machine list in "value".

Example request:

```
DELETE /token HTTP/1.1
Host: example.com
Accept: application/json

{ "hostname": "puckel.example.com",
  "resolver": "machineresolver1",
  "application": "luks" }
```

privacyIDEA Server endpoints

This endpoint is used to create, update, list and delete privacyIDEA server definitions. privacyIDEA server definitions can be used for Remote-Tokens and for Federation-Events.

The code of this module is tested in tests/test_api_privacyideaserver.py

POST /privacyideaserver/test_request
Test the privacyIDEA definition :return:

GET /privacyideaserver/
This call gets the list of privacyIDEA server definitions

POST /privacyideaserver/(*identifier*)
This call creates or updates a privacyIDEA Server definition

Parameters

- **identifier** – The unique name of the privacyIDEA server definition
- **url** – The URL of the privacyIDEA server
- **tls** – Set this to 0, if tls should not be checked
- **description** – A description for the definition

DELETE /privacyideaserver/(*identifier*)

This call deletes the specified privacyIDEA server configuration

Parameters

- **identifier** – The unique name of the privacyIDEA server definition

CA Connector endpoints

This is the REST API for managing CA connector definitions. The CA connectors are written to the database table "caconnector".

The code is tested in tests/test_api_caconnector.py.

GET /caconnector/(*name*)

GET /caconnector/

returns a json list of the available CA connectors

POST /caconnector/(*name*)

Create a new CA connector

DELETE /caconnector/(*name*)

Delete a specific CA connector

Recover endpoints

This module provides the REST API for th password recovery for a user managed in privacyIDEA.

The methods are also tested in the file tests/test_api_register.py

POST /recover/reset

reset the password with a given recovery code. The recovery code was sent by get_recover_code and is bound to a certain user.

JSON Parameters

- **recoverycode** – The recoverycode sent the the user
- **password** – The new password of the user

Return a json result with a boolean "result": true

POST /recover

This method requests a recover code for a user. The recover code it sent via email to the user.

Query Parameters

- **user** – username of the user
- **realm** – realm of the user
- **email** – email of the user

Return JSON with value=True or value=False

Register endpoints

This module contains the REST API for registering as a new user. This endpoint can be used without any authentication, since a new user can register.

The methods are tested in the file tests/test_api_register.py

`GET /register`

This endpoint returns the information if registration is allowed or not. This is used by the UI to either display the registration button or not.

Return JSON with value=True or value=False

`POST /register`

Register a new user in the realm/userresolver. To do so, the user resolver must be writeable like an SQLResolver.

Registering a user in fact creates a new user and also creates the first token for the user. The following values are needed to register the user:

- username (mandatory)
- givenname (mandatory)
- surname (mandatory)
- email address (mandatory)
- password (mandatory)
- mobile phone (optional)
- telephone (optional)

The user receives a registration token via email to be able to login with his self chosen password and the registration token.

JSON Parameters

- **`username`** – The login name of the new user. Check if it already exists
- **`givenname`** – The givenname of the new user
- **`surname`** – The surname of the new user
- **`email`** – The email address of the new user
- **`password`** – The password of the new user. This is the resolver password of the new user.
- **`mobile`** – The mobile phone number
- **`phone`** – The phone number (land line) of the new user

Return a json result with a boolean “result”: true

Monitoring endpoints

This endpoint is used fetch monitoring/statistics data

The code of this module is tested in tests/test_api_monitoring.py

GET /monitoring/(*stats_key*)

GET /monitoring/
return a list of all available statistics keys in the database if no *stats_key* is specified.

If a stats_key is specified it returns the data of this key. The parameters "start" and "end" can be used to specify a time window, from which the statistics data should be fetched.

GET /monitoring/(*stats_key*)**/last**
Get the last value of the stats key

DELETE /monitoring/(*stats_key*)
Delete the statistics data of a certain stats_key.

You can specify the start date and the end date when to delete the monitoring data. You should specify the dates including the timezone. Otherwise your client could send its local time and the server would interpret it as its own local time which would result in deleting unexpected entries.

You can specify the dates like 2010-12-31 22:00+0200

Periodic Task endpoints

These endpoints are used to create, modify and delete periodic tasks.

This module is tested in tests/test_api_periodictask.py

GET /periodictask/taskmodules/
Return a list of task module identifiers.

GET /periodictask/nodes/
Return a list of available nodes

GET /periodictask/
Return a list of objects of defined periodic tasks.

POST /periodictask/
Create or replace an existing periodic task definition.

Parameters

- **id** – ID of an existing periodic task definition that should be updated
- **name** – Name of the periodic task
- **active** – true if the periodic task should be active
- **interval** – Interval at which the periodic task should run (in cron syntax)
- **nodes** – Comma-separated list of nodes on which the periodic task should run
- **taskmodule** – Task module name of the task
- **ordering** – Ordering of the task, must be a number >= 0.
- **options** – A dictionary (possibly JSON) of periodic task options, mapping unicodes to unicodes

Return ID of the periodic task

GET /periodictask/options/(*taskmodule*)
Return the available options for the given taskmodule.

Parameters

- **taskmodule** – Identifier of the task module

Return a dictionary mapping option keys to description dictionaries

POST /periodictask/disable/(*ptaskid*)
Disable a certain periodic task.

Parameters

- **ptaskid** – ID of the periodic task

Return ID of the periodic task

POST /periodictask/enable/(*ptaskid*)
Enable a certain periodic task.

Parameters

- **ptaskid** – ID of the periodic task

Return ID of the periodic task

GET /periodictask/(*ptaskid*)
Return the dictionary describing a periodic task.

Parameters

- **ptaskid** – ID of the periodic task

DELETE /periodictask/(*ptaskid*)
Delete a certain periodic task.

Parameters

- **ptaskid** – ID of the periodic task

Return ID of the periodic task

This endpoint is used to get the information from the server, which application types are known and which options these applications provide.

Applications are used to attach tokens to machines.

The code of this module is tested in tests/test_api_applications.py

Application endpoints

GET /application/
returns a json list of the available applications

Tokentype endpoints

This API endpoint is a generic endpoint that can be used by any token type.

The tokentype needs to implement a classmethod *api_endpoint* and can then be called by /ttype/<tokentype>. This way, each tokentype can create its own API without the need to change the core API.

The TiQR Token uses this API to implement its special functionalities. See *TiQR Token*.

`POST /ttype/`(*ttype*)

This is a special token function. Each token type can define an additional API call, that does not need authentication on the REST API level.

Return Token Type dependent

`GET /ttype/`(*ttype*)

This is a special token function. Each token type can define an additional API call, that does not need authentication on the REST API level.

Return Token Type dependent

SMTP server endpoints

This endpoint is used to create, update, list and delete SMTP server definitions. SMTP server definitions can be used for several purposes like EMail-Token, SMS Token with SMTP gateway, notification like PIN handler and registration.

The code of this module is tested in tests/test_api_smtpserver.py

`POST /smtpserver/send_test_email`

Test the email configuration :return:

`GET /smtpserver/`

This call gets the list of SMTP server definitions

`POST /smtpserver/`(*identifier*)

This call creates or updates an SMTP server definition.

Parameters

- **`identifier`** – The unique name of the SMTP server definition
- **`server`** – The FQDN or IP of the mail server
- **`port`** – The port of the mail server
- **`username`** – The mail username for authentication at the SMTP server
- **`password`** – The password for authentication at the SMTP server
- **`tls`** – If the server should do TLS
- **`description`** – A description for the definition

`DELETE /smtpserver/`(*identifier*)

This call deletes the specified SMTP server configuration

Parameters

- **`identifier`** – The unique name of the SMTP server definition

SMS Gateway endpoints

This endpoint is used to create, modify, list and delete SMS gateway definitions. These gateway definitions are written to the database table "smsgateway" and "smsgatewayoption".

The code of this module is tested in tests/test_api_smsgateway.py

`GET /smsgateway/`(*gwid*)

`GET /smsgateway/`

returns a json list of the gateway definitions

Or

returns a list of available sms providers with their configuration /smsgateway/providers

`POST /smsgateway`

This creates a new SMS gateway definition or updates an existing one.

JSON Parameters

- **`name`** – The unique identifier of the SMS gateway definition
- **`module`** – The providermodule name
- **`description`** – An optional description of the definition
- **`option.*`** – Additional options for the provider module (module specific)

`DELETE /smsgateway/option/`(*gwid*)**`/`**

option this function deletes an option of a gateway definition

Parameters

- **`gwid`** – The id of the sms gateway definition

Return json with success or fail

`DELETE /smsgateway/`(*identifier*)

this function deletes an existing smsgateway definition

Parameters

- **`identifier`** – The name of the sms gateway definition

Return json with success or fail

RADIUS server endpoints

This endpoint is used to create, update, list and delete RADIUS server definitions. RADIUS server definitions can be used for several purposes like RADIUS-Token or RADIUS-passthru policies.

The code of this module is tested in tests/test_api_radiusserver.py

`POST /radiusserver/test_request`

Test the RADIUS definition :return:

`GET /radiusserver/`

This call gets the list of RADIUS server definitions

`POST /radiusserver/`(*identifier*)

This call creates or updates a RADIUS server definition.

Parameters

- **`identifier`** – The unique name of the RADIUS server definition
- **`server`** – The FQDN or IP of the RADIUS server
- **`port`** – The port of the RADIUS server
- **`secret`** – The RADIUS secret of the RADIUS server
- **`description`** – A description for the definition

`DELETE /radiusserver/`(*identifier*)
This call deletes the specified RADIUS server configuration

Parameters

- **`identifier`** – The unique name of the RADIUS server definition

Subscriptions endpoints

This is the controller API for client component subscriptions like ownCloud plugin or RADIUS Credential Provider.

`GET /subscriptions/`(*application*)

`GET /subscriptions/`
Return the subscription object as JSON.

`POST /subscriptions/`
Upload a new subscription file

`DELETE /subscriptions/`(*application*)
Delete an existing subscription

1.15.2 LIB level

At the LIB level all library functions are defined. There is no authentication on this level. Also there is no flask/Web/request code on this level.

Request information and the `logged_in_user` need to be passed to the functions as parameters, if they are needed.

If possible, policies are checked with policy decorators.

library functions

Based on the database models, which are tested in tests/test_db_model.py, there are different modules.

resolver.py contains functions to simply deal with resolver definitions. On this level users and realms are not know, yet.

realm.py contains functions to deal with realm. Realms are a list of several resolvers. So prior to bother the realm.py, the resolver.py should be understood and working. On this level, users are not known, yet.

user.py contains functions to deal with users. A user object is an entity in a realm. And of course the user object itself can be found in a resolver. But you need to have working resolver.py and realm.py to be able to work with user.py

For further details see the following modules:

Users

There are the library functions for user functions. It depends on the lib.resolver and lib.realm.

There are and must be no dependencies to the token functions (lib.token) or to webservices!

This code is tested in tests/test_lib_user.py

class privacyidea.lib.user.**User**(*login=''*, *realm=''*, *resolver=''*)

The user has the attributes login, realm and resolver.

Usually a user can be found via "login@realm".

A user object with an empty login and realm should not exist, whereas a user object could have an empty resolver.

check_password(*password*)

The password of the user is checked against the user source

Parameters **password** – The clear text password

Returns the username of the authenticated user. If unsuccessful, returns None

Return type string/None

delete()

This deletes the user in the user store. I.e. the user in the SQL database or the LDAP gets deleted.

Returns True in case of success

exist()

Check if the user object exists in the user store :return: True or False

get_ordererd_resolvers()

returns a list of resolvernames ordered by priority. The resolver with the lowest priority is the first. If resolvers have the same priority, they are ordered alphabetically.

Returns list or resolvernames

get_search_fields()

Return the valid search fields of a user. The search fields are defined in the UserIdResolver class.

Returns searchFields with name (key) and type (value)

Return type dict

get_user_identifiers()

This returns the UserId information from the resolver object and the resolvertype and the resolvername (former: getUserId) (former: getUserResolverId) :return: The userid, the resolver type and the resolver name

like (1000, "passwdresolver", "resolver1")

Return type tuple

get_user_phone(*phone_type='phone'*, *index=None*)

Returns the phone number or a list of phone numbers of a user.

Parameters

- **phone_type** (*string*) – The type of the phone, i.e. either mobile or phone (land line)

- **index** – The index of the selected phone number of list of the phones of the user. If the index is given, this phone number as string is returned. If the index is omitted, all phone numbers are returned.

Returns list with phone numbers of this user object

get_user_realms()

Returns a list of the realms, a user belongs to. Usually this will only be one realm. But if the user object has no realm but only a resolver, than all realms, containing this resolver are returned. This function is used for the policy module

Returns realms of the user

Return type list

property info

return the detailed information for the user

Returns a dict with all the userinformation

Return type dict

is_empty()

login = ''

realm = ''

resolver = ''

update_user_info(*attributes*, *password=None*)

This updates the given attributes of a user. The attributes can be “username”, “surname”, “givenname”, “email”, “mobile”, “phone”, “password”

Parameters

- **attributes** (*dict*) – A dictionary of the attributes to be updated
- **password** – The password of the user

Returns True in case of success

privacyidea.lib.user.**create_user**(*resolvername*, *attributes*, *password=None*)

This creates a new user in the given resolver. The resolver must be editable to do so.

The attributes is a dictionary containing the keys “username”, “email”, “phone”, “mobile”, “surname”, “givenname”, “password”.

We return the UID and not the user object, since the user could be located in several realms!

Parameters

- **resolvername** (*basestring*) – The name of the resolver, in which the user should be created
- **attributes** (*dict*) – Attributes of the user
- **password** – The password of the user

Returns The uid of the user object

privacyidea.lib.user.**get_user_from_param**(*param*, *optionalOrRequired=True*)

Find the parameters user, realm and resolver and create a user object from these parameters.

An exception is raised, if a user in a realm is found in more than one resolvers.

Parameters **param** (*dict*) – The dictionary of request parameters

Returns User as found in the parameters

Return type User object

`privacyidea.lib.user.`**`get_user_list`**(*param=None*, *user=None*)

`privacyidea.lib.user.`**`get_username`**(*userid*, *resolvername*)

Determine the username for a given id and a resolvername.

Parameters

- **`userid`** (*`string`*) – The id of the user in a resolver
- **`resolvername`** – The name of the resolver

Returns the username or “” if it does not exist

Return type string

`privacyidea.lib.user.`**`log_used_user`**(*user*, *other_text=''*)

This creates a log message combined of a user and another text. The user information is only added, if user.login != user.used_login

Parameters

- **`user`** (*`User object`*) – A user to log
- **`other_text`** – Some additional text

Returns str

`privacyidea.lib.user.`**`split_user`**(*username*)

Split the username of the form user@realm into the username and the realm splitting myemail@emailprovider.com@realm is also possible and will return (myemail@emailprovider.com, realm).

If for a user@domain the “domain” does not exist as realm, the name is not split, since it might be the user@domain in the default realm

We can also split realmuser to (user, realm)

Parameters **`username`** (*`string`*) – the username to split

Returns username and realm

Return type tuple

Token Class

The following token types are known to privacyIDEA. All are inherited from the base tokenclass describe below.

4 Eyes Token

`class` `privacyidea.lib.tokens.foureyestoken.`**`FourEyesTokenClass`**(*db_token*)

The FourEyes token can be used to implement the Two Man Rule. The FourEyes token defines how many tokens of which realms are required like: * 2 tokens of RealmA * 1 token of RealmB

Then users (the owners of those tokens) need to login by everyone entering their OTP PIN and OTP value. It does not matter, in which order they enter the values. All their PINs and OTPs are concatenated into one password field but need to be separated by the splitting sign.

The FourEyes token again splits the password value and tries to authenticate each of the these passwords in the realms using the function `check_realm_pass`.

The FourEyes token itself does not provide an OTP PIN.

The token is initialized using additional parameters at token/init:

Example Authentication Request:

```
POST /auth HTTP/1.1
Host: example.com
Accept: application/json

type=4eyes
user=cornelius
realm=realm1
4eyes=realm1:2,realm2:1
separator=%20
```

`authenticate`(*passw*, *user=None*, *options=None*)

do the authentication on base of password / otp and user and options, the request parameters.

Here we contact the other privacyIDEA server to validate the OtpVal.

Parameters

- **`passw`** – the password / otp
- **`user`** – the requesting user
- **`options`** – the additional request parameters

Returns tuple of (success, otp_count - 0 or -1, reply)

`static convert_realms`(*realms*)

This function converts the realms as given by the API parameter to a dictionary.

realm1:2,realm2:1 -> {"realm1":2, "realm2":1}

Parameters **`realms`** (`basestring`) – a serialized list of realms

Returns dict of realms

`static get_class_info`(*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **`key`** (`string`) – subsection identifier
- **`ret`** (`user defined`) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

`static get_class_prefix`()

return the token type prefix

`static get_class_type`()

return the class type identifier

`static realms_dict_to_string`(*realms*)

This function converts the realms - if it is a dictionary - to a string.

{"realm1": {"selected": True,

"count": 1 },

"realm2": {"selected": True, "count": 2} -> realm1:1,realm2:2

Parameters **realms** (*dict*) – the realms as they are passed from the WebUI

Returns realms

Return type basestring

update (*param*)

This method is called during the initialization process. :param param: parameters from the token init :type param: dict :return: None

Certificate Token

class privacyidea.lib.tokens.certificatetoken.**CertificateTokenClass** (*aToken*)

Token to implement an X509 certificate. The certificate can be enrolled by sending a CSR to the server or the keypair is created by the server. If the server creates the keypair, the user can download a PKCS12 file. The OTP PIN is used as passphrase for the PKCS12 file.

privacyIDEA is capable of working with different CA connectors.

Valid parameters are *request* or *certificate*, both PEM encoded. If you pass a *request* you also need to pass the *ca* that should be used to sign the request. Passing a *certificate* just uploads the certificate to a new token object.

A certificate token can be created by an administrative task with the token/init api like this:

Example Initialization Request:

```
POST /auth HTTP/1.1
Host: example.com
Accept: application/json

type=certificate
user=cornelius
realm=realm1
request=<PEM encoded request>
ca=<name of the ca connector>
```

Example Initialization Request, key generation on servers side

In this case the certificate is created on behalf of another user.

```
POST /auth HTTP/1.1
Host: example.com
Accept: application/json

type=certificate
user=cornelius
realm=realm1
generate=1
ca=<name of the ca connector>
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
```

(continues on next page)

(continued from previous page)

```
    "detail": {
      "certificate": "...PEM..."
    },
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
      "status": true,
      "value": true
    },
    "version": "privacyIDEA unknown"
  }
```

get_as_dict()

This returns the token data as a dictionary. It is used to display the token list at /token/list.

The certificate token can add the PKCS12 file if it exists

Returns The token data as dict

Return type dict

static get_class_info(*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

static get_class_prefix()

static get_class_type()

get_init_detail(*params=None*, *user=None*)

At the end of the initialization we return the certificate and the PKCS12 file, if the private key exists.

hKeyRequired = False

revoke()

This revokes the token. We need to determine the CA, which issues the certificate, contact the connector and revoke the certificate

Some token types may revoke a token without locking it.

set_pin(*pin*, *encrypt=False*)

set the PIN of a token. The PIN of the certificate token is stored encrypted. It is used as passphrase for the PKCS12 file.

Parameters

- **pin** (*basestring*) – the pin to be set for the token
- **encrypt** (*bool*) – If set to True, the pin is stored encrypted and can be retrieved from the database again

update(*param*)

This method is called during the initialization process. :param param: parameters from the token init :type param: dict :return: None

using_pin = False

Daplug Token

class privacyidea.lib.tokens.daplugtoken.**DaplugTokenClass**(*a_token*)

daplug token class implementation

check_otp(*anOtpVal*, *counter=None*, *window=None*, *options=None*)

checkOtp - validate the token otp against a given otpvalue

Parameters

- **anOtpVal** (*string, format: efekeiebekeh*) – the otpvalue to be verified
- **counter** (*int*) – the counter state, that should be verified
- **window** (*int*) – the counter +window, which should be checked
- **options** (*dict*) – the dict, which could contain token specific info

Returns the counter state or -1

Return type int

check_otp_exist(*otp*, *window=10*)

checks if the given OTP value is/are values of this very token. This is used to autoassign and to determine the serial number of a token.

Parameters

- **otp** (*string*) – the to be verified otp value
- **window** (*int*) – the lookahead window for the counter

Returns counter or -1 if otp does not exist

Return type int

static get_class_info(*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or string

static get_class_prefix()

Return the prefix, that is used as a prefix for the serial numbers. :return: oath

static get_class_type()

return the token type shortname

Returns 'hotp'

Return type string

get_multi_otp(*count=0*, *epoch_start=0*, *epoch_end=0*, *curTime=None*, *timestamp=None*)

return a dictionary of multiple future OTP values of the HOTP/HMAC token

WARNING: the dict that is returned contains a sequence number as key. This it NOT the otp counter!

Parameters

- **count** (*int*) – how many otp values should be returned
- **epoch_start** – Not used in HOTP
- **epoch_end** – Not used in HOTP
- **curTime** – Not used in HOTP
- **timestamp** – not used in HOTP
- **counter_index** – whether the counter should be used as index

Returns tuple of status: boolean, error: text and the OTP dictionary

get_otp (*current_time=None*)

return the next otp value

Parameters **curTime** – Not Used in HOTP

Returns next otp value and PIN if possible

Return type tuple

resync (*otp1*, *otp2*, *options=None*)

resync the token based on two otp values - external method to do the resync of the token

Parameters

- **otp1** (*string*) – the first otp value
- **otp2** (*string*) – the second otp value
- **options** (*dict or None*) – optional token specific parameters

Returns counter or -1 if otp does not exist

Return type int

split_pin_pass (*passw*, *user=None*, *options=None*)

Split the password into the token PIN and the OTP value

take the given password and split it into the PIN and the OTP value. The splitting can be dependent of certain policies. The policies may depend on the user.

Each token type may define its own way to slit the PIN and the OTP value.

Parameters

- **passw** – the password to split
- **user** (*User object*) – The user/owner of the token
- **options** (*dict*) – can be used be the token types.

Returns tuple of pin and otp value

Returns tuple of (split status, pin, otp value)

Return type tuple

Email Token

class privacyidea.lib.tokens.emailtoken.**EmailTokenClass**(*aToken*)

Implementation of the EMail Token Class, that sends OTP values via SMTP. (Similar to SMSTokenClass)

EMAIL_ADDRESS_KEY = 'email'

check_otp(*anOtpVal*, *counter=None*, *window=None*, *options=None*)

check the otpval of a token against a given counter and the window

Parameters **passw** (*string*) – the to be verified passw/pin

Returns counter if found, -1 if not found

Return type int

create_challenge(*transactionid=None*, *options=None*)

create a challenge, which is submitted to the user

Parameters

- **transactionid** – the id of this challenge
- **options** – the request context parameters / data You can pass exception=1 to raise an exception, if the SMS could not be sent. Otherwise the message is contained in the response.

Returns

tuple of (bool, message and data) bool, if submit was successful message is submitted to the user data is preserved in the challenge attributes - additional attributes, which are displayed in the

output

static get_class_info(*key=None*, *ret='all'*)

returns all or a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

:rtype : s.o.

static get_class_prefix()

Return the prefix, that is used as a prefix for the serial numbers. :return: oath

static get_class_type()

return the generic token class identifier

is_challenge_request(*passw*, *user=None*, *options=None*)

check, if the request would start a challenge

We need to define the function again, to get rid of the is_challenge_request-decorator of the HOTP-Token

Parameters

- **passw** – password, which might be pin or pin+otp
- **options** – dictionary of additional request parameters

Returns returns true or false

classmethod test_config(*params=None*)

This method is used to test the token config. Some tokens require some special token configuration like the SMS-Token or the Email-Token. To test this configuration, this classmethod is used.

It takes token specific parameters and returns a tuple of a boolean and a result description.

Parameters **params** (*dict*) – token specific parameters

Returns success, description

Return type tuple

update(*param*, *reset_failcount=True*)

update - process initialization parameters

Parameters **param** (*dict*) – dict of initialization parameters

Returns nothing

HOTP Token

class privacyidea.lib.tokens.hotptoken.**HotpTokenClass**(*db_token*)

hotp token class implementation

check_otp(*anOtpVal*, *counter=None*, *window=None*, *options=None*)

check if the given OTP value is valid for this token.

Parameters

- **anOtpVal** (*string*) – the to be verified otpvalue
- **counter** (*int*) – the counter state, that should be verified
- **window** (*int*) – the counter +window, which should be checked
- **options** (*dict*) – the dict, which could contain token specific info

Returns the counter state or -1

Return type int

check_otp_exist(*otp*, *window=10*, *symetric=False*, *inc_counter=True*)

checks if the given OTP value is/are values of this very token. This is used to autoassign and to determine the serial number of a token.

Parameters

- **otp** (*string*) – the to be verified otp value
- **window** (*int*) – the lookahead window for the counter

Returns counter or -1 if otp does not exist

Return type int

generate_symmetric_key(*server_component*, *client_component*, *options=None*)

Generate a composite key from a server and client component using a PBKDF2-based scheme.

Parameters

- **server_component** (*hex string*) – The component usually generated by privacyIDEA
- **client_component** (*hex string*) – The component usually generated by the client (e.g. smartphone)

- **options** –

Returns the new generated key as hex string

Return type str

static get_class_info(*key=None*, *ret='all'*)

returns a subtree of the token definition Is used by lib.token.get_token_info

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict

static get_class_prefix()

Return the prefix, that is used as a prefix for the serial numbers. :return: oath

static get_class_type()

return the token type shortname

Returns 'hotp'

Return type string

classmethod get_default_settings(*g*, *params*)

This method returns a dictionary with default settings for token enrollment. These default settings are defined in SCOPE.USER or SCOPE.ADMIN and are hotp_hashlib, hotp_otplen. If these are set, the user or admin will only be able to enroll tokens with these values.

The returned dictionary is added to the parameters of the API call. :param g: context object, see documentation of `Match` :param params: The call parameters :type params: dict :return: default parameters

get_init_detail(*params=None*, *user=None*)

to complete the token initialization some additional details should be returned, which are displayed at the end of the token initialization. This is the e.g. the enrollment URL for a Google Authenticator.

get_multi_otp(*count=0*, *epoch_start=0*, *epoch_end=0*, *curTime=None*, *timestamp=None*, *counter_index=False*)

return a dictionary of multiple future OTP values of the HOTP/HMAC token

WARNING: the dict that is returned contains a sequence number as key. This it NOT the otp counter!

Parameters

- **count** (*int*) – how many otp values should be returned
- **epoch_start** – Not used in HOTP
- **epoch_end** – Not used in HOTP
- **curTime** – Not used in HOTP
- **timestamp** – not used in HOTP
- **counter_index** – whether the counter should be used as index

Returns tuple of status: boolean, error: text and the OTP dictionary

get_otp(*current_time=None*)

return the next otp value

Parameters **`curTime`** – Not Used in HOTP

Returns next otp value and PIN if possible

Return type tuple

`static get_setting_type`(*key*)

This function returns the type of the token specific config/setting. This way a tokenclass can define settings, that can be "public" or a "password". If this setting is written to the database, the type of the setting is set automatically in set_privacyidea_config

The key name needs to start with the token type.

Parameters **`key`** – The token specific setting key

Returns A string like "public"

`static get_sync_timeout`()

get the token sync timeout value

Returns timeout value in seconds

Return type int

`property hashlib`

`is_previous_otp`(*otp*, *window=10*)

Check if the OTP values was previously used.

Parameters

- **`otp`** –
- **`window`** –

Returns

`resync`(*otp1*, *otp2*, *options=None*)

resync the token based on two otp values

Parameters

- **`otp1`** (*`string`*) – the first otp value
- **`otp2`** (*`string`*) – the second otp value
- **`options`** (*`dict or None`*) – optional token specific parameters

Returns counter or -1 if otp does not exist

Return type int

`update`(*param*, *reset_failcount=True*)

process the initialization parameters

Do we really always need an otpkey? the otpKey is handled in the parent class :param param: dict of initialization parameters :type param: dict

Returns nothing

mOTP Token

class privacyidea.lib.tokens.motptoken.**MotpTokenClass**(*db_token*)

check_otp(*anOtpVal*, *counter=None*, *window=None*, *options=None*)
validate the token otp against a given otpvalue

Parameters

- **anOtpVal** (*str*) – the to be verified otpvalue
- **counter** (*int*) – the counter state, that should be verified
- **window** (*int*) – the counter +window, which should be checked
- **options** (*dict*) – the dict, which could contain token specific info

Returns the counter state or -1

Return type int

static get_class_info(*key=None*, *ret='all'*)
returns a subtree of the token definition Is used by lib.token.get_token_info

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

:rtype : dict or string

static get_class_prefix()

static get_class_type()

get_init_detail(*params=None*, *user=None*)
to complete the token normalisation, the response of the initialization should be build by the token specific method, the getInitDetails

update(*param*, *reset_failcount=True*)
update - process initialization parameters

Parameters **param** (*dict*) – dict of initialization parameters

Returns nothing

OCRA Token

The OCRA token is the base OCRA functionality. Usually it is created by importing a CSV or PSKC file.

This code is tested in tests/test_lib_tokens_tiqr.

Implementation

class `privacyidea.lib.tokens.ocratoken.`**`OcraTokenClass`**(*db_token*)

The OCRA Token Implementation

`check_otp`(*otpval*, *counter=None*, *window=None*, *options=None*)

This function is invoked by `TokenClass.check_challenge_response` and checks if the given password matches the expected response for the given challenge.

Parameters

- **`otpval`** – the password (pin + otp)
- **`counter`** – ignored
- **`window`** – ignored
- **`options`** – dictionary that *must* contain "challenge"

Returns >=0 if the challenge matches, -1 otherwise

`create_challenge`(*transactionid=None*, *options=None*)

This method creates a challenge, which is submitted to the user. The submitted challenge will be preserved in the challenge database.

If no transaction id is given, the system will create a transaction id and return it, so that the response can refer to this transaction.

Parameters

- **`transactionid`** – the id of this challenge
- **`options`** (*`dict`*) – the request context parameters / data

Returns tuple of (bool, message, transactionid, attributes)

Return type tuple

The return tuple builds up like this: `bool` if submit was successful; `message` which is displayed in the JSON response; additional `attributes`, which are displayed in the JSON response.

static `get_class_info`(*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **`key`** (*`string`*) – subsection identifier
- **`ret`** (*`user defined`*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

static `get_class_prefix`()

Return the prefix, that is used as a prefix for the serial numbers. :return: OCRA :rtype: basestring

static `get_class_type`()

Returns the internal token type identifier :return: ocra :rtype: basestring

static `get_import_csv`(*l*)

Read the list from a csv file and return a dictionary, that can be used to do a token_init.

Parameters **`l`** (*`list`*) – The list of the line of a csv file

Returns A dictionary of init params

is_challenge_request (*passw*, *user=None*, *options=None*)

check, if the request would start a challenge In fact every Request that is not a response needs to start a challenge request.

At the moment we do not think of other ways to trigger a challenge.

This function is not decorated with @challenge_response_allowed

as the OCRA token is always a challenge response token!

Parameters

- **passw** – The PIN of the token.
- **options** – dictionary of additional request parameters

Returns returns true or false

update (*param*)

This method is called during the initialization process.

Parameters **param** (*dict*) – parameters from the token init

Returns None

verify_response (*passw=None*, *challenge=None*)

This method verifies if the *passw* is the valid OCRA response to the *challenge*. In case of success we return a value > 0

Parameters **passw** (*string*) – the password (pin+otp)

Returns return otp_counter. If -1, challenge does not match

Return type int

Paper Token

class privacyidea.lib.tokens.papertoken.**PaperTokenClass** (*db_token*)

The Paper Token allows to print out the next e.g. 100 OTP values. This sheet of paper can be used to authenticate and strike out the used OTP values.

static get_class_info (*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

static get_class_prefix ()

Return the prefix, that is used as a prefix for the serial numbers. :return: PPR

static get_class_type ()

return the token type shortname

Returns ‘paper’

Return type string

update(*param*, *reset_failcount=True*)
process the initialization parameters

Do we really always need an otpkey? the otpKey is handled in the parent class :param param: dict of initialization parameters :type param: dict

Returns nothing

PasswordToken

class privacyidea.lib.tokens.passwordtoken.**PasswordTokenClass**(*aToken*)
This Token does use a fixed Password as the OTP value. In addition, the OTP PIN can be used with this token. This Token can be used for a scenario like losttoken

class SecretPassword(*secObj*)

check_password(*password*)
Parameters **password** (*str*) –
Returns result of password check: 0 if success, -1 if failed
Return type int

get_password()

check_otp(*anOtpVal*, *counter=None*, *window=None*, *options=None*)
This checks the static password

Parameters **anOtpVal** – This contains the “OTP” value, which is the static

password :return: result of password check, 0 in case of success, -1 if fail :rtype: int

static get_class_info(*key=None*, *ret='all'*)
returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

static get_class_prefix()

static get_class_type()

set_otplen(*otplen=0*)
sets the OTP length to the length of the password

Parameters **otplen** (*int*) – This is ignored in this class

Result None

update(*param*)
This method is called during the initialization process. :param param: parameters from the token init :type param: dict :return: None

Push Token

class privacyidea.lib.tokens.pushtoken.**PushTokenClass**(*db_token*)

The PUSH token uses the firebase service to send challenges to the users smartphone. The user confirms on the smartphone, signes the challenge and sends it back to privacyIDEA.

The enrollment occurs in two enrollment steps:

# Step 1

The device is enrolled using a QR code, that looks like this:

otpauth://pipush/PIPU0006EF85?url=https://yourprivacyideaserver/enroll/this/token&ttl=120

# Step 2

In the QR code is a URL, where the smartphone sends the remaining data for the enrollment.

POST https://yourprivacyideaserver/ttype/push enrollment_credential=<some credential> serial=<token serial> fbtoken=<firebase token> pubkey=<public key>

For more information see: https://github.com/privacyidea/privacyidea/issues/1342 https://github.com/privacyidea/privacyidea/wiki/concept%3A-PushToken

classmethod api_endpoint(*request*, *g*)

This provides a function which is called by the API endpoint /ttype/push which is defined in api/ttype.py

The method returns return "json", {}

This endpoint is used for the 2nd enrollment step of the smartphone. Parameters sent:

- serial
- fbtoken
- pubkey

This endpoint is also used, if the smartphone sends the signed response to the challenge during authentication Parameters sent:

- serial
- nonce (which is the challenge)
- signature (which is the signed nonce)

Parameters

- **request** – The Flask request
- **g** – The Flask global object g

Returns dictionary

authenticate(*passw*, *user=None*, *options=None*)

High level interface which covers the check_pin and check_otp This is the method that verifies single shot authentication. The challenge is send to the smartphone app and privacyIDEA waits for the response to arrive.

Parameters

- **passw** (*string*) – the password which could be pin+otp value
- **user** (*User object*) – The authenticating user
- **options** (*dict*) – dictionary of additional request parameters

Returns

returns tuple of 1. true or false for the pin match, 2. the otpcounter (int) and the 3. reply (dict) that will be added as

additional information in the JSON response of `/validate/check`.

Return type tuple

check_challenge_response (*user=None*, *passw=None*, *options=None*)

This function checks, if the challenge for the given transaction_id was marked as answered correctly. For this we check the otp_status of the challenge with the transaction_id in the database.

We do not care about the password

Parameters

- **user** (*User object*) – the requesting user
- **passw** (*string*) – the password (pin+otp)
- **options** (*dict*) – additional arguments from the request, which could be token specific. Usually “transaction_id”

Returns return otp_counter. If -1, challenge does not match

Return type int

create_challenge (*transactionid=None*, *options=None*)

This method creates a challenge, which is submitted to the user. The submitted challenge will be preserved in the challenge database.

If no transaction id is given, the system will create a transaction id and return it, so that the response can refer to this transaction.

Parameters

- **transactionid** – the id of this challenge
- **options** (*dict*) – the request context parameters / data

Returns tuple of (bool, message, transactionid, attributes)

Return type tuple

The return tuple builds up like this: `bool` if submit was successful; `message` which is displayed in the JSON response; additional `attributes`, which are displayed in the JSON response.

static get_class_info (*key=None*, *ret='all'*)

returns all or a subtree of the token definition

Parameters

- **key** (*str*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

:rtype : s.o.

static get_class_prefix ()

static get_class_type ()

return the generic token class identifier

get_init_detail(*params=None*, *user=None*)

This returns the init details during enrollment.

In the 1st step the QR Code is returned.

is_challenge_request(*passw*, *user=None*, *options=None*)

check, if the request would start a challenge

We need to define the function again, to get rid of the is_challenge_request-decorator of the base class

Parameters

- **passw** – password, which might be pin or pin+otp
- **options** – dictionary of additional request parameters

Returns returns true or false

mode = ['authenticate', 'challenge', 'outofband']

update(*param*, *reset_failcount=True*)

process the initialization parameters

We need to distinguish the first authentication step and the second authentication step.

1. **step:** parameter type contained. parameter genkey contained.
2. **step:** parameter serial contained parameter fbtoken contained parameter pubkey contained

Parameters **param** (*dict*) – dict of initialization parameters

Returns nothing

Questionnaire Token

class privacyidea.lib.tokens.questionnairetoken.**QuestionnaireTokenClass**(*db_token*)

This is a Questionnaire Token. The token stores a list of questions and answers in the tokeninfo database table. The answers are encrypted. During authentication a random answer is selected and presented as challenge. The user has to remember and pass the right answer.

check_answer(*given_answer*, *challenge_object*)

Check if the given answer is the answer to the sent question. The question for this challenge response was stored in the challenge_object.

Then we get the answer from the tokeninfo.

Parameters

- **given_answer** – The answer given by the user
- **challenge_object** – The challenge object as stored in the database

Returns in case of success: 1

check_challenge_response(*user=None*, *passw=None*, *options=None*)

This method verifies if there is a matching question for the given passw and also verifies if the answer is correct.

It then returns the the otp_counter = 1

Parameters

- **user** (*User object*) – the requesting user
- **passw** (*string*) – the password - in fact it is the answer to the question

- **options** (*dict*) – additional arguments from the request, which could be token specific. Usually "transaction_id"

Returns return 1 if the answer to the question is correct, -1 otherwise.

Return type int

create_challenge(*transactionid=None*, *options=None*)

This method creates a challenge, which is submitted to the user. The submitted challenge will be preserved in the challenge database.

The challenge is a randomly selected question of the available questions for this token.

If no transaction id is given, the system will create a transaction id and return it, so that the response can refer to this transaction.

Parameters

- **transactionid** – the id of this challenge
- **options** (*dict*) – the request context parameters / data

Returns tuple of (bool, message, transactionid, attributes)

Return type tuple

The return tuple builds up like this: `bool` if submit was successful; `message` which is displayed in the JSON response; additional `attributes`, which are displayed in the JSON response.

classmethod get_class_info(*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

static get_class_prefix()

Return the prefix, that is used as a prefix for the serial numbers. :return: QUST :rtype: basestring

static get_class_type()

Returns the internal token type identifier :return: qust :rtype: basestring

static get_setting_type(*key*)

The setting type of questions is public, so that the user can also read the questions.

Parameters key – The key of the setting

Returns "public" string

is_challenge_request(*passw*, *user=None*, *options=None*)

The questionnaire token is always a challenge response token. The challenge is triggered by providing the PIN as the password.

Parameters

- **passw** (*string*) – password, which might be pin or pin+otp
- **user** (*User object*) – The user from the authentication request
- **options** (*dict*) – dictionary of additional request parameters

Returns true or false

Return type bool

update(*param*)
This method is called during the initialization process.

Parameters **param** (*dict*) – parameters from the token init

Returns None

RADIUS Token

class privacyidea.lib.tokens.radiustoken.**RadiusTokenClass**(*db_token*)

authenticate(*passw*, *user=None*, *options=None*)
do the authentication on base of password / otp and user and options, the request parameters.

This is only called after it is verified, that the upper level is no challenge-request or challenge-response

The "options" are read-only in this method. They are not modified here. authenticate is the last method in the loop `check_token_list`.

communication with RADIUS server: yes, if is no previous "radius_result" If there is a "radius" result in the options, we do not query the radius server

modification of options: options can be modified if we query the radius server. However, this is not important since authenticate is the last call.

Parameters

- **passw** – the password / otp
- **user** – the requesting user
- **options** – the additional request parameters

Returns tuple of (success, otp_count - 0 or -1, reply)

check_challenge_response(*user=None*, *passw=None*, *options=None*)
This method verifies if there is a matching question for the given passw and also verifies if the answer is correct.

It then returns the the otp_counter = 1

Parameters

- **user** (*User object*) – the requesting user
- **passw** (*string*) – the password - in fact it is the answer to the question
- **options** (*dict*) – additional arguments from the request, which could be token specific. Usually "transaction_id"

Returns return otp_counter. If -1, challenge does not match

Return type int

check_otp(*otpval*, *counter=None*, *window=None*, *options=None*)
Originally check_otp returns an OTP counter. I.e. in a failed attempt we return -1. In case of success we return 1 :param otpval: :param counter: :param window: :param options: :return:

property check_pin_local
lookup if pin should be checked locally or on radius host

Returns bool

create_challenge(*transactionid=None*, *options=None*)

create a challenge, which is submitted to the user

This method is called after `is_challenge_request` has verified, that a challenge needs to be created.

communication with RADIUS server: no modification of options: no

Parameters

- **transactionid** – the id of this challenge
- **options** – the request context parameters / data

Returns

tuple of (bool, message and data) bool, if submit was successful message is submitted to the user data is preserved in the challenge attributes - additional attributes, which are displayed in the

output

static get_class_info(*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or string

static get_class_prefix()

return the token type prefix

static get_class_type()

return the class type identifier

is_challenge_request(*passw*, *user=None*, *options=None*)

This method checks, if this is a request, that triggers a challenge. It depends on the way, the pin is checked - either locally or remotely. In addition, the RADIUS token has to be configured to allow challenge response.

communication with RADIUS server: yes modification of options: The communication with the RADIUS server can

change the options, radius_state, radius_result, radius_message

Parameters

- **passw** (*string*) – password, which might be pin or pin+otp
- **user** (*User object*) – The user from the authentication request
- **options** (*dict*) – dictionary of additional request parameters

Returns true or false

is_challenge_response(*passw*, *user=None*, *options=None*)

This method checks, if this is a request, that is the response to a previously sent challenge. But we do not query the RADIUS server.

This is the first method in the loop `check_token_list`.

communication with RADIUS server: no modification of options: The “radius_result” key is set to None

Parameters

- **passw** (*string*) – password, which might be pin or pin+otp
- **user** (*User object*) – the requesting user
- **options** (*dict*) – dictionary of additional request parameters

Returns true or false

Return type bool

split_pin_pass (*passw*, *user=None*, *options=None*)

Split the PIN and the OTP value. Only if it is locally checked and not remotely.

update (*param*)

second phase of the init process - updates parameters

Parameters **param** – the request parameters

Returns

- nothing -

Registration Code Token

class privacyidea.lib.tokens.registrationtoken.**RegistrationTokenClass** (*aToken*)

Token to implement a registration code. It can be used to create a registration code or a “TAN” which can be used once by a user to authenticate somewhere. After this registration code is used, the token is automatically deleted.

The idea is to provide a workflow, where the user can get a registration code by e.g. postal mail and then use this code as the initial first factor to authenticate to the UI to enroll real tokens.

A registration code can be created by an administrative task with the token/init api like this:

Example Authentication Request:

```
POST /token/init HTTP/1.1
Host: example.com
Accept: application/json

type=registration
user=cornelius
realm=realm1
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
   "detail": {
     "registrationcode": "12345808124095097608"
   },
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
```

(continues on next page)

(continued from previous page)

```
      "value": true
    },
    "version": "privacyIDEA unknown"
  }
```

static get_class_info(*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

static get_class_prefix()

static get_class_type()

get_init_detail(*params=None*, *user=None*)

At the end of the initialization we return the registration code.

inc_count_auth_success()

Increase the counter, that counts successful authentications In case of successful authentication the token does needs to be deleted.

update(*param*)

This method is called during the initialization process. :param param: parameters from the token init :type param: dict :return: None

Remote Token

class privacyidea.lib.tokens.remotetoken.**RemoteTokenClass**(*db_token*)

The Remote token forwards an authentication request to another privacyIDEA server. The request can be forwarded to a user on the other server or to a serial number on the other server. The PIN can be checked on the local privacyIDEA server or on the remote server.

Using the Remote token you can assign one physical token to many different users.

authenticate(*passw*, *user=None*, *options=None*)

do the authentication on base of password / otp and user and options, the request parameters.

Here we contact the other privacyIDEA server to validate the OtpVal.

Parameters

- **passw** – the password / otp
- **user** – the requesting user
- **options** – the additional request parameters

Returns tuple of (success, otp_count - 0 or -1, reply)

check_otp(*otpval*, *counter=None*, *window=None*, *options=None*)

run the http request against the remote host

Parameters

- **otpval** – the OTP value
- **counter** (*int*) – The counter for counter based otp values
- **window** – a counter window
- **options** (*dict*) – additional token specific options

Returns counter of the matching OTP value.

Return type int

property check_pin_local

lookup if pin should be checked locally or on remote host

Returns bool

static get_class_info(*key=None*, *ret='all'*)

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or string

static get_class_prefix()

return the token type prefix

static get_class_type()

return the class type identifier

is_challenge_request(*passw*, *user=None*, *options=None*)

This method checks, if this is a request, that triggers a challenge. It depends on the way, the pin is checked - either locally or remote

Parameters

- **passw** (*string*) – password, which might be pin or pin+otp
- **user** (*User object*) – The user from the authentication request
- **options** (*dict*) – dictionary of additional request parameters

Returns true or false

update(*param*)

second phase of the init process - updates parameters

Parameters **param** – the request parameters

Returns

- nothing -

SMS Token

class privacyidea.lib.tokens.smstoken.**SmsTokenClass**(*db_token*)

The SMS token sends an SMS containing an OTP via some kind of gateway. The gateways can be an SMTP or HTTP gateway or the special sipgate protocol. The Gateways are defined in the SMSProvider Modules.

The SMS token is a challenge response token. I.e. the first request needs to contain the correct OTP PIN. If the OTP PIN is correct, the sending of the SMS is triggered. The second authentication must either contain the OTP PIN and the OTP value or the transaction_id and the OTP value.

Example 1st Authentication Request:

```
POST /validate/check HTTP/1.1
Host: example.com
Accept: application/json

user=cornelius
pass=otppin
```

Example 1st response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
   "detail": {
     "transaction_id": "xyz"
   },
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
     "status": true,
     "value": false
   },
   "version": "privacyIDEA unknown"
 }
```

After this, the SMS is triggered. When the SMS is received the second part of authentication looks like this:

Example 2nd Authentication Request:

```
POST /validate/check HTTP/1.1
Host: example.com
Accept: application/json

user=cornelius
transaction_id=xyz
pass=otppin
```

Example 1st response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
   "detail": {
   },
   "id": 1,
```

(continues on next page)

(continued from previous page)

```
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": true
  },
  "version": "privacyIDEA unknown"
}
```

check_otp (*anOtpVal*, *counter=None*, *window=None*, *options=None*)

check the otpval of a token against a given counter and the window

Parameters **passw** (*string*) – the to be verified passw/pin

Returns counter if found, -1 if not found

Return type int

create_challenge (*transactionid=None*, *options=None*)

create a challenge, which is submitted to the user

Parameters

- **transactionid** – the id of this challenge
- **options** – the request context parameters / data You can pass exception=1 to raise an exception, if the SMS could not be sent. Otherwise the message is contained in the response.

Returns

tuple of (bool, message and data) bool, if submit was successful message is submitted to the user data is preserved in the challenge attributes - additional attributes, which are displayed in the

output

static get_class_info (*key=None*, *ret='all'*)

returns all or a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

:rtype : s.o.

static get_class_prefix ()

Return the prefix, that is used as a prefix for the serial numbers. :return: oath

static get_class_type ()

return the generic token class identifier

is_challenge_request (*passw*, *user=None*, *options=None*)

check, if the request would start a challenge

We need to define the function again, to get rid of the is_challenge_request-decorator of the HOTP-Token

Parameters

- **passw** – password, which might be pin or pin+otp
- **options** – dictionary of additional request parameters

Returns returns true or false

update(*param*, *reset_failcount=True*)
process initialization parameters

Parameters **param** (*dict*) – dict of initialization parameters

Returns nothing

SPass Token

class privacyidea.lib.tokens.spasstoken.**SpassTokenClass**(*db_token*)
This is a simple pass token. It does have no OTP component. The OTP checking will always succeed. Of course, an OTP PIN can be used.

authenticate(*passw*, *user=None*, *options=None*)
in case of a wrong passw, we return a bad matching pin, so the result will be an invalid token

check_otp(*otpval*, *counter=None*, *window=None*, *options=None*)
As we have no otp value we always return true. (counter == 0)

static get_class_info(*key=None*, *ret='all'*)
returns a subtree of the token definition Is used by lib.token.get_token_info

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict

static get_class_prefix()

static get_class_type()

static is_challenge_request(*passw*, *user*, *options=None*)
The spass token does not support challenge response :param passw: :param user: :param options: :return:

static is_challenge_response(*passw*, *user*, *options=None*, *challenges=None*)
This method checks, if this is a request, that is the response to a previously sent challenge.

The default behaviour to check if this is the response to a previous challenge is simply by checking if the request contains a parameter `state` or `transactionid` i.e. checking if the `options` parameter contains a key `state` or `transactionid`.

This method does not try to verify the response itself! It only determines, if this is a response for a challenge or not. The response is verified in check_challenge_response.

Parameters

- **passw** (*string*) – password, which might be pin or pin+otp
- **user** (*User object*) – the requesting user
- **options** (*dict*) – dictionary of additional request parameters

Returns true or false

Return type bool

update(*param*)
Update the token object

Parameters **param** – a dictionary with different params like keysize, description, genkey, otpkey, pin

Type param: dict

SSHKey Token

class privacyidea.lib.tokens.sshkeytoken.**SSHkeyTokenClass**(*db_token*)

The SSHKeyTokenClass provides a TokenClass that stores the public SSH key and can give the public SSH key via the getotp function. This can be used to manage SSH keys and retrieve the public ssh key to import it to authorized keys files.

static get_class_info(*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dictionary

static get_class_prefix()

static get_class_type()

get_sshkey()

returns the public SSH key

Returns SSH pub key

Return type string

mode = ['authenticate']

update(*param*)

The key holds the public ssh key and this is required

The key probably is of the form “ssh-rsa BASE64 comment”

using_pin = False

TiQR Token

The TiQR token is a special App based token, which allows easy login and which is based on OCRA.

It generates an enrollment QR code, which contains a link with the more detailed enrollment information.

For a description of the TiQR protocol see

- https://www.usenix.org/legacy/events/lisa11/tech/full_papers/Rijswijk.pdf
- https://github.com/SURFnet/tiqr/wiki/Protocol-documentation.
- https://tiqr.org

The TiQR token is based on the OCRA algorithm. It lets you authenticate with your smartphone by scanning a QR code.

The TiQR token is enrolled via /token/init, but it requires no otpkey, since the otpkey is generated on the smartphone and pushed to the privacyIDEA server in a seconds step.

Enrollment

1. Start enrollment with /token/init
2. Scan the QR code in the details of the JSON result. The QR code contains a link to /ttype/tiqr?action=metadata
3. The TiQR Smartphone App will fetch this link and get more information
4. The TiQR Smartphone App will push the otpkey to a link /ttype/tiqr?action=enrollment and the token will be ready for use.

Authentication

An application that wants to use the TiQR token with privacyIDEA has to use the token in challenge response.

1. Call `/validate/check?user=<user>&pass=<pin>` with the PIN of the TiQR token
2. The details of the JSON response contain a QR code, that needs to be shown to the user. In addition the application needs to save the `transaction_id` in the response.
3. The user scans the QR code.
4. The TiQR App communicates with privacyIDEA via the API /ttype/tiqr. In this step the response of the App to the challenge is verified. The successful authentication is stored in the Challenge DB table. (No need for the application to take any action)
5. Now, the application needs to poll `/validate/polltransaction?transaction_id=<transaction_id>` to check the transaction status. If the endpoint returns `false`, the challenge has not been answered yet.
6. Once `/validate/polltransaction` returns true, the application needs to finalize the authentication with a request `/validate/check?user=<user>&transaction_id=<transaction_id>&pass=`. The `pass` can be empty. If `value=true` is returned, the user authenticated successfully with the TiQR token.

This code is tested in tests/test_lib_tokens_tiqr.

Implementation

class `privacyidea.lib.tokens.tiqrtoken.`**`TiqrTokenClass`**(*db_token*)

The TiQR Token implementation.

classmethod **`api_endpoint`**(*request*, *g*)

This provides a function to be plugged into the API endpoint /ttype/<tokentype> which is defined in api/ttype.py See *Tokentype endpoints*.

Parameters

- **`request`** – The Flask request
- **`g`** – The Flask global object g

Returns Flask Response or text

`check_challenge_response`(*user=None*, *passw=None*, *options=None*)

This function checks, if the challenge for the given transaction_id was marked as answered correctly. For this we check the otp_status of the challenge with the transaction_id in the database.

We do not care about the password

Parameters

- **user** (*User object*) – the requesting user
- **passw** (*string*) – the password (pin+otp)
- **options** (*dict*) – additional arguments from the request, which could be token specific. Usually "transaction_id"

Returns return otp_counter. If -1, challenge does not match

Return type int

create_challenge (*transactionid=None*, *options=None*)

This method creates a challenge, which is submitted to the user. The submitted challenge will be preserved in the challenge database.

If no transaction id is given, the system will create a transaction id and return it, so that the response can refer to this transaction.

Parameters

- **transactionid** – the id of this challenge
- **options** (*dict*) – the request context parameters / data

Returns tuple of (bool, message, transactionid, attributes)

Return type tuple

The return tuple builds up like this: `bool` if submit was successful; `message` which is displayed in the JSON response; additional `attributes`, which are displayed in the JSON response.

static get_class_info (*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

static get_class_prefix ()

Return the prefix, that is used as a prefix for the serial numbers. :return: TiQR :rtype: basestring

static get_class_type ()

Returns the internal token type identifier :return: tiqr :rtype: basestring

get_init_detail (*params=None*, *user=None*)

At the end of the initialization we return the URL for the TiQR App.

mode = ['authenticate', 'challenge', 'outofband']

update (*param*)

This method is called during the initialization process.

Parameters param (*dict*) – parameters from the token init

Returns None

TOTP Token

class privacyidea.lib.tokens.totptoken.**TotpTokenClass**(*db_token*)

check_otp(*anOtpVal*, *counter=None*, *window=None*, *options=None*)
validate the token otp against a given otpvalue

Parameters

- **anOtpVal** (*string*) – the to be verified otpvalue
- **counter** – the counter state, that should be verified. For TOTP

this is the unix system time (seconds) divided by 30/60 :type counter: int :param window: the counter +window (sec), which should be checked :type window: int :param options: the dict, which could contain token specific info :type options: dict :return: the counter or -1 :rtype: int

check_otp_exist(*otp*, *window=None*, *options=None*, *symetric=True*, *inc_counter=True*)
checks if the given OTP value is/are values of this very token at all. This is used to autoassign and to determine the serial number of a token. In fact it is a check_otp with an enhanced window.

Parameters

- **otp** (*string*) – the to be verified otp value
- **window** (*int*) – the lookahead window for the counter in seconds!!!

Returns counter or -1 if otp does not exist

Return type int

static get_class_info(*key=None*, *ret='all'*)
returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

static get_class_prefix()
Return the prefix, that is used as a prefix for the serial numbers. :return: TOTP

static get_class_type()
return the token type shortname

Returns 'totp'

Return type string

classmethod get_default_settings(*g*, *params*)
This method returns a dictionary with default settings for token enrollment. These default settings are defined in SCOPE.USER or SCOPE.ADMIN and are totp_hashlib, totp_timestep and totp_otplen. If these are set, the user or admin will only be able to enroll tokens with these values.

The returned dictionary is added to the parameters of the API call. :param g: context object, see documentation of `Match` :param params: The call parameters :type params: dict :return: default parameters

static get_import_csv(*l*)
Read the list from a csv file and return a dictionary, that can be used to do a token_init.

Parameters `l` (*list*) – The list of the line of a csv file

Returns A dictionary of init params

`get_multi_otp`(*count=0*, *epoch_start=0*, *epoch_end=0*, *curTime=None*, *timestamp=None*)
return a dictionary of multiple future OTP values of the HOTP/HMAC token

Parameters

- **count** (*int*) – how many otp values should be returned
- **epoch_start** – not implemented
- **epoch_end** – not implemented
- **curTime** (*datetime*) – Simulate the servertime
- **timestamp** (*epoch time*) – Simulate the servertime

Returns tuple of status: boolean, error: text and the OTP dictionary

`get_otp`(*current_time=None*, *do_truncation=True*, *time_seconds=None*, *challenge=None*)
get the next OTP value

Parameters **current_time** – the current time, for which the OTP value

should be calculated for. :type current_time: datetime object :param time_seconds: the current time, for which the OTP value should be calculated for (date +%s) :type: time_seconds: int, unix system time seconds :return: next otp value, and PIN, if possible :rtype: tuple

`static get_setting_type`(*key*)
This function returns the type of the token specific config/setting. This way a tokenclass can define settings, that can be "public" or a "password". If this setting is written to the database, the type of the setting is set automatically in set_privacyidea_config

The key name needs to start with the token type.

Parameters **key** – The token specific setting key

Returns A string like "public"

`property hashlib`

`resync`(*otp1*, *otp2*, *options=None*)
resync the token based on two otp values external method to do the resync of the token

Parameters

- **otp1** (*string*) – the first otp value
- **otp2** (*string*) – the second otp value
- **options** (*dict or None*) – optional token specific parameters

Returns counter or -1 if otp does not exist

Return type int

`resyncDiffLimit = 1`

`property timeshift`

`property timestep`

`property timewindow`

`update`(*param*, *reset_failcount=True*)
This is called during initialization of the token to add additional attributes to the token object.

Parameters **param** (*dict*) – dict of initialization parameters

Returns nothing

U2F Token

U2F is the "Universal 2nd Factor" specified by the FIDO Alliance. The register and authentication process is described here:

https://fidoalliance.org/specs/fido-u2f-v1.0-nfc-bt-amendment-20150514/fido-u2f-raw-message-formats.html

But you do not need to be aware of this. privacyIDEA wraps all FIDO specific communication, which should make it easier for you, to integrate the U2F tokens managed by privacyIDEA into your application.

U2F Tokens can be either

- registered by administrators for users or
- registered by the users themselves.

Enrollment

The enrollment/registering can be completely performed within privacyIDEA.

But if you want to enroll the U2F token via the REST API you need to do it in two steps:

1. Step

```
POST /token/init HTTP/1.1
Host: example.com
Accept: application/json

type=u2f
```

This step returns a serial number.

2. Step

```
POST /token/init HTTP/1.1
Host: example.com
Accept: application/json

type=u2f
serial=U2F1234578
clientdata=<clientdata>
regdata=<regdata>
```

clientdata and *regdata* are the values returned by the U2F device.

You need to call the javascript function

> u2f.register([registerRequest], [], function(u2fData) { });

and the responseHandler needs to send the *clientdata* and *regdata* back to privacyIDEA (2. step).

Authentication

The U2F token is a challenge response token. I.e. you need to trigger a challenge e.g. by sending the OTP PIN/Password for this token.

Get the challenge

```
POST /validate/check HTTP/1.1
Host: example.com
Accept: application/json

user=cornelius
pass=tokenpin
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
   "detail": {
     "attributes": {
                     "hideResponseInput": true,
                     "img": ...imageUrl...
                     "u2fSignRequest": {
                         "challenge": "...",
                         "appId": "...",
                         "keyHandle": "...",
                         "version": "U2F_V2"
                     }
                   },
     "message": "Please confirm with your U2F token (Yubico U2F EE ...)"
     "transaction_id": "02235076952647019161"
   },
   "id": 1,
   "jsonrpc": "2.0",
   "result": {
       "status": true,
       "value": false,
   },
   "version": "privacyIDEA unknown"
 }
```

Send the Response

The application now needs to call the javascript function *u2f.sign* with the *u2fSignRequest* from the response.

> var signRequests = [error.detail.attributes.u2fSignRequest]; u2f.sign(signRequests, function(u2fResult) {});

The response handler function needs to call the */validate/check* API again with the signatureData and clientData returned by the U2F device in the *u2fResult*:

```
POST /validate/check HTTP/1.1
Host: example.com
Accept: application/json

user=cornelius
pass=
transaction_id=<transaction_id>
signaturedata=signatureData
clientdata=clientData
```

Implementation

class privacyidea.lib.tokens.u2ftoken.**U2fTokenClass**(*db_token*)

The U2F Token implementation.

classmethod api_endpoint(*request*, *g*)

This provides a function to be plugged into the API endpoint /ttype/u2f

The u2f token can return the facet list at this URL.

Parameters

- **request** – The Flask request
- **g** – The Flask global object g

Returns Flask Response or text

check_otp(*otpval*, *counter=None*, *window=None*, *options=None*)

This checks the response of a previous challenge. :param otpval: N/A :param counter: The authentication counter :param window: N/A :param options: contains "clientdata", "signaturedata" and

"transaction_id"

Returns A value > 0 in case of success

create_challenge(*transactionid=None*, *options=None*)

This method creates a challenge, which is submitted to the user. The submitted challenge will be preserved in the challenge database.

If no transaction id is given, the system will create a transaction id and return it, so that the response can refer to this transaction.

Parameters

- **transactionid** – the id of this challenge
- **options** (*dict*) – the request context parameters / data

Returns tuple of (bool, message, transactionid, attributes)

Return type tuple

The return tuple builds up like this: `bool` if submit was successful; `message` which is displayed in the JSON response; additional `attributes`, which are displayed in the JSON response.

static get_class_info(*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier

- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

static get_class_prefix()

Return the prefix, that is used as a prefix for the serial numbers. :return: U2F :rtype: basestring

static get_class_type()

Returns the internal token type identifier :return: u2f :rtype: basestring

get_init_detail(*params=None*, *user=None*)

At the end of the initialization we ask the user to press the button

is_challenge_request(*passw*, *user=None*, *options=None*)

check, if the request would start a challenge In fact every Request that is not a response needs to start a challenge request.

At the moment we do not think of other ways to trigger a challenge.

This function is not decorated with @challenge_response_allowed

as the U2F token is always a challenge response token!

Parameters

- **passw** – The PIN of the token.
- **options** – dictionary of additional request parameters

Returns returns true or false

update(*param*, *reset_failcount=True*)

This method is called during the initialization process.

Parameters **param** (*dict*) – parameters from the token init

Returns None

Vasco Token

WebAuthn Token

WebAuthn is the Web Authentication API specified by the FIDO Alliance. The register and authentication process is described here:

https://w3c.github.io/webauthn/#sctn-rp-operations

But you do not need to be aware of this. privacyIDEA wraps all FIDO specific communication, which should make it easier for you, to integrate the U2F tokens managed by privacyIDEA into your application.

WebAuthn tokens can be either

- registered by administrators for users or
- registered by the users themselves.

Beware the WebAuthn tokens can only be used if the privacyIDEA server and the applications and services the user needs to access all reside under the same domain or subdomains thereof.

This means a WebAuthn token registered by privacyidea.mycompany.com can be used to sign in to sites like mycompany.com and vpn.mycompany.com, but not (for example) mycompany.someservice.com.

Enrollment

The enrollment/registering can be completely performed within privacyIDEA.

But if you want to enroll the WebAuthn token via the REST API you need to do it in two steps:

1. Step

```
POST /token/init HTTP/1.1
Host: example.com
Accept: application/json

type=webauthn
```

This step returns a nonce, a relying party (containing a name and an ID generated from your domain), and a serial number, along with a transaction ID, and a message to display to the user. It will also pass some additional options regarding timeout, which authenticators are acceptable, and what key types are acceptable to the server.

2. Step

```
POST /token/init HTTP/1.1
Host: example.com
Accept: application/json

type=webauthn
transaction_id=<transaction_id>
description=<description>
clientdata=<clientDataJSON>
regdata=<attestationObject>
registrationclientextensions=<registrationClientExtensions>
```

clientDataJSON and *attestationObject* are the values returned by the WebAuthn authenticator. *description* is an optional description string for the new token.

You need to call the javascript function

navigator .credentials .create({

challenge: <nonce>, rp: <relyingParty>, user: {

id: Uint8Array.from(<serialNumber>, c => c.charCodeAt(0)), name: <name>, displayName: <displayName>

}, pubKeyCredParams: [

{ alg: <preferredAlgorithm>, type: “public-key”

}, {

alg: <alternativeAlgorithm>, type: “public-key”

}

], authenticatorSelection: <authenticatorSelection>, timeout: <timeout>, attestation: <attestation>, extensions: {

authnSel: <authenticatorSelectionList>

}

}) .then(function(credential) { <responseHandler> }) .catch(function(error) { <errorHandler> });

Here *nonce*, *relyingParty*, *serialNumber*, *preferredAlgorithm*, *alternativeAlgorithm*, *authenticatorSelection*, *timeout*, *attestation*, *authenticatorSelectionList*, *name*, and *displayName* are the values provided by the server in the *webAuthnRegisterRequest* field in the response from the first step. *alternativeAlgorithm*, *authenticatorSelection*, *timeout*, *attestation*, and *authenticatorSelectionList* are optional. If *attestation* is not provided, the client should default to *direct* attestation. If *timeout* is not provided, it may be omitted, or a sensible default chosen. Any other optional values must be omiffed, if the server has not sent them. Please note that the nonce will be a binary, encoded using the web-safe base64 algorithm specified by WebAuthn, and needs to be decoded and passed as Uint8Array.

If an *authenticationSelectionList* was given, the *responseHandler* needs to verify, that the field *authnSel* of *credential.getExtensionResults()* contains true. If this is not the case, the *responseHandler* should abort and call the *errorHandler*, displaying an error telling the user to use his company-provided token.

The *responseHandler* needs to then send the *clientDataJSON*, *attestationObject*, and *registrationClientExtensions* contained in the *response* field of the *credential* (2. step) back to the server. If enrollment succeeds, the server will send a response with a *webAuthnRegisterResponse* field, containing a *subject* field with the description of the newly created token.

The server expects the *clientDataJSON* and *attestationObject* encoded as web-safe base64 as defined by the WebAuthn standard. This encoding is similar to standard base64, but ‘-‘ and ‘_’ should be used in the alphabet instead of ‘+’ and ‘/’, respectively, and any padding should be omitted.

The *registrationClientExtensions* are optional and should simply be omitted, if the client does not provide them. It the *registrationClientExtensions* are available, they must be encoded as a utf-8 JSON string, then sent to the server as web-safe base64.

Please beware that the btoa() function provided by ECMA-Script expects a 16-bit encoded string where all characters are in the range 0x0000 to 0x00FF. The *attestationObject* contains CBOR-encoded binary data, returned as an ArrayBuffer.

The problem and ways to solve it are described in detail in this MDN-Article:

https://developer.mozilla.org/en-US/docs/Web/API/WindowBase64/Base64_encoding_and_decoding#The_Unicode_Problem

Authentication

The WebAuthn token is a challenge response token. I.e. you need to trigger a challenge, either by sending the OTP PIN/Password for this token to the /validate/check endpoint, or by calling the /validate/triggerchallenge endpoint using a service account with sufficient permissions.

Get the challenge (using /validate/check)

The /validate/check endpoint can be used to trigger a challenge using the PIN for the token (without requiring any special permissions).

Request

```
POST /validate/check HTTP/1.1
Host: example.com
Accept: application/json

user=<username>
pass=<password>
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "detail": {
        "attributes": {
            "hideResponseInput": true,
            "img": <imageUrl>,
            "webAuthnSignRequest": {
                "challenge": <nonce>,
                "allowCredentials": [{
                    "id": <credentialId>,
                    "type": <credentialType>,
                    "transports": <allowedTransports>,
                }],
                "rpId": <relyingPartyId>,
                "userVerification": <userVerificationRequirement>,
                "timeout": <timeout>
            }
        },
        "message": "Please confirm with your WebAuthn token",
        "transaction_id": <transactionId>
    },
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "status": true,
        "value": false
    },
    "versionnumber": <privacyIDEAversion>
}
```

Get the challenge (using /validate/triggerchallenge)

The /validate/triggerchallenge endpoint can be used to trigger a challenge using a service account (without requiring the PIN for the token).

Request

```
POST /validate/triggerchallenge HTTP/1.1
Host: example.com
Accept: application/json
PI-Authorization: <authToken>

user=<username>
serial=<tokenSerial>
```

Providing the *tokenSerial* is optional. If just a user is provided, a challenge will be triggered for every challenge response token the user has.

Response

```
HTTP/1.1 200 OK
Content-Type: application/json
```

(continues on next page)

(continued from previous page)

```
{
    "detail": {
        "attributes": {
            "hideResponseInput": true,
            "img": <imageUrl>,
            "webAuthnSignRequest": {
                "challenge": <nonce>,
                "allowCredentials": [{
                    "id": <credentialId>,
                    "type": <credentialType>,
                    "transports": <allowedTransports>,
                }],
                "rpId": <relyingPartyId>,
                "userVerification": <userVerificationRequirement>,
                "timeout": <timeout>
            }
        },
        "message": "Please confirm with your WebAuthn token",
        "messages": ["Please confirm with your WebAuthn token"],
        "multi_challenge": [{
            "attributes": {
                "hideResponseInput": true,
                "img": <imageUrl>,
                "webAuthnSignRequest": {
                    "challenge": <nonce>,
                    "allowCredentials": [{
                        "id": <credentialId>,
                        "type": <credentialType>,
                        "transports": <allowedTransports>,
                    }],
                    "rpId": <relyingPartyId>,
                    "userVerification": <userVerificationRequirement>,
                    "timeout": <timeout>
                }
            },
            "message": "Please confirm with your WebAuthn token",
            "serial": <tokenSerial>,
            "transaction_id": <transactionId>,
            "type": "webauthn"
        }],
        "serial": <tokenSerial>,
        "threadid": <threadId>,
        "transaction_id": <transactionId>,
        "transaction_ids": [<transactionId>],
        "type": "webauthn"
    },
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "status": true,
        "value": 1
    },
    "versionnumber": <privacyIDEAversion>
}
```

Send the Response

The application now needs to call the javascript function *navigator.credentials.get* with *publicKeyCredentialRequestOptions* built using the *nonce*, *credentialId*, *allowedTransports*, *userVerificationRequirement* and *timeout* from the server. The timeout is optional and may be omitted, if not provided, the client may also pick a sensible default. Please note that the nonce will be a binary, encoded using the web-safe base64 algorithm specified by WebAuthn, and needs to be decoded and passed as Uint8Array.

> **const publicKeyCredentialRequestOptions = {** challenge: <nonce>, allowCredentials: [{
>
> > id: Uint8Array.from(<credentialId>, c=> c.charCodeAt(0)), type: <credentialType>, transports: <allowedTransports>
>
> }], userVerification: <userVerificationRequirement>, rpId: <relyingPartyId>, timeout: <timeout>
>
> } navigator
>
> > .credentials .get({publicKey: publicKeyCredentialRequestOptions}) .then(function(assertion) { <responseHandler> }) .catch(function(error) { <errorHandler> });

The *responseHandler* needs to call the */validate/check* API providing the *serial* of the token the user is signing in with, and the *transaction_id*, for the current challenge, along with the *id*, returned by the WebAuthn device in the *assertion* and the *authenticatorData*, *clientDataJSON* and *signature*, *userHandle*, and *assertionClientExtensions* contained in the *response* field of the *assertion*.

clientDataJSON, *authenticatorData* and *signature* should be encoded as web-safe base64 without padding. For more detailed instructions, refer to "2. Step" under "Enrollment" above.

The *userHandle* and *assertionClientExtensions* are optional and should be omitted, if not provided by the authenticator. The *assertionClientExtensions* – if available – must be encoded as a utf-8 JSON string, and transmitted to the server as web-safe base64. The *userHandle* is simply passed as a string, note – however – that it may be necessary to re-encode this to utf-16, since the authenticator will return utf-8, while the library making the http request will likely require all parameters in the native encoding of the language (usually utf-16).

```
POST /validate/check HTTP/1.1
Host: example.com
Accept: application/json

user=<user>
pass=
transaction_id=<transaction_id>
credentialid=<id>
clientdata=<clientDataJSON>
signaturedata=<signature>
authenticatordata=<authenticatorData>
userhandle=<userHandle>
assertionclientextensions=<assertionClientExtensions>
```

Implementation

class privacyidea.lib.tokens.webauthntoken.**WebAuthnTokenClass**(*db_token*)

The WebAuthn Token implementation.

check_otp(*otpval*, *counter=None*, *window=None*, *options=None*)

This checks the response of a previous challenge.

Since this is not a traditional token, otpval and window are unused. The information from the client is instead passed in the fields *serial*, *id*, *assertion*, *authenticatorData*, *clientDataJSON*, and *signature* of the options dictionary.

Parameters

- **otpval** (*None*) – Unused for this token type
- **counter** (*int*) – The authentication counter
- **window** (*None*) – Unused for this token type
- **options** (*dict*) – Contains the data from the client, along with policy configurations.

Returns A numerical value where values larger than zero indicate success.

Return type int

create_challenge(*transactionid=None*, *options=None*)

Create a challenge for challenge-response authentication.

This method creates a challenge, which is submitted to the user. The submitted challenge will be preserved in the challenge database.

If no transaction id is given, the system will create a transaction id and return it, so that the response can refer to this transaction.

This method will return a tuple containing a bool value, indicating whether a challenge was successfully created, along with a message to display to the user, the transaction id, and a dictionary containing all parameters and data needed to respond to the challenge, as per the api.

Parameters

- **transactionid** (*basestring*) – The id of this challenge
- **options** (*dict*) – The request context parameters and data

Returns Success status, message, transaction id and response details

Return type (bool, basestring, basestring, dict)

decrypt_otpkey()

This method fetches a decrypted version of the otp_key.

This method becomes necessary, since the way WebAuthn is implemented in PrivacyIdea, the otpkey of a WebAuthn token is the credential_id, which may encode important information and needs to be sent to the client to allow the client to create an assertion for the authentication process.

Returns The otpkey decrypted and encoded as WebAuthn base64.

Return type basestring

static get_class_info(*key=None*, *ret='all'*)

returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier

- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or scalar

static get_class_prefix()

Return the prefix, that is used as a prefix for the serial numbers.

Returns WAN

Return type basestring

static get_class_type()

Returns the internal token type identifier

Returns webauthn

Return type basestring

get_init_detail(*params=None*, *user=None*)

At the end of the initialization we ask the user to confirm the enrollment with his token.

This will prepare all the information the client needs to build the publicKeyCredentialCreationOptions to call navigator.credentials.create() with. It will then be called again, once the token is created and provide confirmation of the successful enrollment to the client.

Parameters

- **params** (*dict*) – A dictionary with parameters from the request.
- **user** (User) – The user enrolling the token.

Returns The response detail returned to the client.

Return type dict

static get_setting_type(*key*)

Fetch the type of a setting specific to WebAuthn tokens.

The WebAuthn token defines several public settings. When these are written to the database, the type of the setting is automatically stored along with the setting by set_privacyidea_config().

The key name needs to be in WEBAUTHN_TOKEN_SPECIFIC_SETTINGS.keys() and match /^webauthn./. If the specified setting does not exist, a ValueError will be thrown.

Parameters key (*basestring*) – The token specific setting key

Returns The setting type

Return type "public"

is_challenge_request(*passw*, *user=None*, *options=None*)

Check if the request would start a challenge.

Every request that is not a response needs to spawn a challenge.

Note: This function does not need to be decorated with @challenge_response_allowed, as the WebAuthn token is always a challenge response token!

Parameters

- **passw** (*basestring*) – The PIN of the token
- **user** (User) – The User making the request
- **options** (*dict*) – Dictionary of additional request parameters

Returns Whether to trigger a challenge

Return type bool

update(*param*, *reset_failcount=True*)

This method is called during the initialization process.

Parameters

- **param** (*dict*) – Parameters from the token init.
- **reset_failcount** (*bool*) – Whether to reset the fail count.

Returns Nothing

Return type None

Yubico Token

class privacyidea.lib.tokens.yubicotoken.**YubicoTokenClass**(*db_token*)

check_otp(*anOtpVal*, *counter=None*, *window=None*, *options=None*)

Here we contact the Yubico Cloud server to validate the OtpVal.

static get_class_info(*key=None*, *ret='all'*)

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type dict or string

static get_class_prefix()

static get_class_type()

update(*param*)

Update the token object

Parameters param – a dictionary with different params like keysize, description, genkey, otpkey, pin

Type param: dict

Yubikey Token

class privacyidea.lib.tokens.yubikeytoken.**YubikeyTokenClass**(*db_token*)

The Yubikey Token in the Yubico AES mode

classmethod api_endpoint(*request*, *g*)

This provides a function to be plugged into the API endpoint /ttype/yubikey which is defined in api/ttype.py

The endpoint /ttype/yubikey is used for the Yubico validate request according to https://developers.yubico.com/yubikey-val/Validation_Protocol_V2.0.html

Parameters

- **request** – The Flask request

- **g** – The Flask global object g

Returns Flask Response or text

Required query parameters

Query id The id of the client to identify the correct shared secret

Query otp The OTP from the yubikey in the yubikey mode

Query nonce 16-40 bytes of random data

Optional parameters h, timestamp, sl, timeout are not supported at the moment.

check_otp (*anOtpVal*, *counter=None*, *window=None*, *options=None*)
validate the token otp against a given otpvalue

Parameters

- **anOtpVal** (*string*) – the to be verified otpvalue
- **counter** (*int*) – the counter state. It is not used by the Yubikey because the current counter value is sent encrypted inside the OTP value
- **window** (*int*) – the counter +window, which is not used in the Yubikey because the current counter value is sent encrypted inside the OTP, allowing a simple comparison between the encrypted counter value and the stored counter value
- **options** (*dict*) – the dict, which could contain token specific info

Returns the counter state or an error code (< 0):

-1 if the OTP is old (counter < stored counter) -2 if the private_uid sent in the OTP is wrong (different from the one stored with the token) -3 if the CRC verification fails :rtype: int

check_otp_exist (*otp*, *window=None*)
checks if the given OTP value is/are values of this very token. This is used to autoassign and to determine the serial number of a token.

static check_yubikey_pass (*passw*)
if the Token has set a PIN the user must also enter the PIN for authentication!

This checks the output of a yubikey in AES mode without providing the serial number. The first 12 (of 44) or 16 of 48) characters are the tokenid, which is stored in the tokeninfo yubikey.tokenid or the prefix yubikey.prefix.

Parameters passw (*string*) – The password that consist of the static yubikey prefix and the otp

Returns True/False and the User-Object of the token owner

Return type dict

static get_class_info (*key=None*, *ret='all'*)
returns a subtree of the token definition

Parameters

- **key** (*string*) – subsection identifier
- **ret** (*user defined*) – default return value, if nothing is found

Returns subsection if key exists or user defined

Return type s.o.

static get_class_prefix ()

static get_class_type()

is_challenge_request(*passw*, *user=None*, *options=None*)
This method checks, if this is a request, that triggers a challenge.

Parameters

- **passw** (*string*) – password, which might be pin or pin+otp
- **user** (*User object*) – The user from the authentication request
- **options** (*dict*) – dictionary of additional request parameters

Returns true or false

update(*param*, *reset_failcount=True*)
Update the token object

Parameters param – a dictionary with different params like keysize, description, genkey, otpkey, pin

Type param: dict

class privacyidea.lib.tokenclass.**TokenClass**(*db_token*)

add_init_details(*key*, *value*)
(was addInfo) Adds information to a volatile internal dict

add_tokeninfo(*key*, *value*, *value_type=None*)
Add a key and a value to the DB tokeninfo :param key: :param value: :return:

add_user(*user*, *report=None*)
Set the user attributes (uid, resolvername, resolvertype) of a token.

Parameters

- **user** – a User() object, consisting of loginname and realm
- **report** – tbdf.

Returns None

classmethod api_endpoint(*request*, *g*)
This provides a function to be plugged into the API endpoint /ttype/<tokentype> which is defined in api/ttype.py

The method should return return "json", {}

or return "text", "OK"

Parameters

- **request** – The Flask request
- **g** – The Flask global object g

Returns Flask Response or text

authenticate(*passw*, *user=None*, *options=None*)
High level interface which covers the check_pin and check_otp This is the method that verifies single shot authentication like they are done with push button tokens.

It is a high level interface to support other tokens as well, which do not have a pin and otp separation - they could overwrite this method

If the authentication succeeds an OTP counter needs to be increased, i.e. the OTP value that was used for this authentication is invalidated!

Parameters

- **passw** (*string*) – the password which could be pin+otp value
- **user** (*User object*) – The authenticating user
- **options** (*dict*) – dictionary of additional request parameters

Returns

returns tuple of 1. true or false for the pin match, 2. the otpcounter (int) and the 3. reply (dict) that will be added as

additional information in the JSON response of `/validate/check`.

Return type tuple

static challenge_janitor()

Just clean up all challenges, for which the expiration has expired.

Returns None

check_all(*message_list*)

Perform all checks on the token. Returns False if the token is either: * auth counter exceeded * not active * fail counter exceeded * validity period exceeded

This is used in the function token.check_token_list

Parameters **message_list** – A list of messages

Returns False, if any of the checks fail

check_auth_counter()

This function checks the count_auth and the count_auth_success. If the counters are less or equal than the maximum allowed counters it returns True. Otherwise False.

Returns success if the counter is less than max

Return type bool

check_challenge_response(*user=None*, *passw=None*, *options=None*)

This method verifies if there is a matching challenge for the given passw and also verifies if the response is correct.

It then returns the new otp_counter of the token.

In case of success the otp_counter will be >= 0.

Parameters

- **user** (*User object*) – the requesting user
- **passw** (*string*) – the password (pin+otp)
- **options** (*dict*) – additional arguments from the request, which could be token specific. Usually “transactionid”

Returns return otp_counter. If -1, challenge does not match

Return type int

check_failcount()

Checks if the failcounter is exceeded. It returns True, if the failcounter is less than maxfail :return: True or False

check_last_auth_newer(*last_auth*)

Check if the last successful authentication with the token is newer than the specified time delta which is passed as 10h, 7d or 1y.

It returns True, if the last authentication with this token is **newer*** than the specified delta.

Parameters **last_auth** (*basestring*) – 10h, 7d or 1y

Returns bool

check_otp(*otpval*, *counter=None*, *window=None*, *options=None*)

This checks the OTP value, AFTER the upper level did the checkPIN

In the base class we do not know, how to calculate the OTP value. So we return -1. In case of success, we should return >=0, the counter

Parameters

- **otpval** – the OTP value
- **counter** (*int*) – The counter for counter based otp values
- **window** – a counter window
- **options** (*dict*) – additional token specific options

Returns counter of the matching OTP value.

Return type int

check_otp_exist(*otp*, *window=None*)

checks if the given OTP value is/are values of this very token. This is used to autoassign and to determine the serial number of a token.

Parameters

- **otp** – the OTP value
- **window** (*int*) – The look ahead window

Returns True or a value > 0 in case of success

check_pin(*pin*, *user=None*, *options=None*)

Check the PIN of the given Password. Usually this is only dependent on the token itself, but the user object can cause certain policies.

Each token could implement its own PIN checking behaviour.

Parameters

- **pin** (*string*) – the PIN (static password component), that is to be checked.
- **user** (*User object*) – for certain PIN policies (e.g. checking against the user store) this is the user, whose password would be checked. But at the moment we are checking against the userstore in the decorator "auth_otppin".
- **options** – the optional request parameters

Returns If the PIN is correct, return True

Return type bool

check_reset_failcount()

Checks if we should reset the failcounter due to the FAILCOUNTER_CLEAR_TIMEOUT

Returns True, if the failcounter was resetted

check_validity_period()

This checks if the datetime.datetime.now() is within the validity period of the token.

Returns success

Return type bool

create_challenge(*transactionid=None*, *options=None*)

This method creates a challenge, which is submitted to the user. The submitted challenge will be preserved in the challenge database.

If no transaction id is given, the system will create a transaction id and return it, so that the response can refer to this transaction.

Parameters

- **transactionid** – the id of this challenge
- **options** (`dict`) – the request context parameters / data

Returns tuple of (bool, message, transactionid, attributes)

Return type tuple

The return tuple builds up like this: `bool` if submit was successful; `message` which is displayed in the JSON response; additional `attributes`, which are displayed in the JSON response.

static decode_otpkey(*otpkey*, *otpkeyformat*)

Decode the otp key which is given in a specific format.

Supported formats:

- `hex`, in which the otpkey is returned verbatim
- `base32check`, which is specified in `decode_base32check`

In case the OTP key is malformed or if the format is unknown, a ParameterError is raised.

Parameters

- **otpkey** – OTP key passed by the user
- **otpkeyformat** – “hex” or “base32check”

Returns hex-encoded otpkey

del_tokeninfo(*key=None*)

delete_token()

delete the database token

enable(*enable=True*)

generate_symmetric_key(*server_component*, *client_component*, *options=None*)

This method generates a symmetric key, from a server component and a client component. This key generation could be based on HMAC, KDF or even Diffie-Hellman.

The basic key-generation is simply replacing the last n byte of the server component with bytes of the client component.

Parameters

- **server_component** (`str`) – The component usually generated by privacyIDEA. This is a hex string
- **client_component** (`str`) – The component usually generated by the client (e.g. smartphone). This is a hex string.

- **options** –

Returns the new generated key as hex string

Return type str

get_as_dict()

This returns the token data as a dictionary. It is used to display the token list at /token/list.

Returns The token data as dict

Return type dict

static get_class_info(*key=None*, *ret='all'*)

static get_class_prefix()

static get_class_type()

get_count_auth()

Return the number of all authentication tries

get_count_auth_max()

Return the number of maximum allowed authentications

get_count_auth_success()

Return the number of successful authentications

get_count_auth_success_max()

Return the maximum allowed successful authentications

get_count_window()

classmethod get_default_settings(*g*, *params*)

This method returns a dictionary with default settings for token enrollment. These default settings depend on the token type and the defined policies.

The returned dictionary is added to the parameters of the API call. :param g: context object, see documentation of `Match` :param params: The call parameters :type params: dict :return: default parameters

get_failcount()

static get_hashlib(*hLibStr*)

Returns a hashlib function for a given string :param hLibStr: the hashlib :type hLibStr: string :return: the hashlib :rtype: function

static get_import_csv(*l*)

Read the list from a csv file and return a dictionary, that can be used to do a token_init.

Parameters **l** (*list*) – The list of the line of a csv file

Returns A dictionary of init params

get_init_detail(*params=None*, *user=None*)

to complete the token initialization, the response of the initialisation should be build by this token specific method. This method is called from api/token after the token is enrolled

get_init_detail returns additional information after an admin/init like the QR code of an HOTP/TOTP token. Can be anything else.

Parameters

- **params** (*dict*) – The request params during token creation token/init
- **user** (*User object*) – the user, token owner

Returns additional descriptions

Return type dict

get_init_details()

return the status of the token rollout

Returns return the status dict.

Return type dict

get_max_failcount()

get_multi_otp(*count=0*, *epoch_start=0*, *epoch_end=0*, *curTime=None*, *timestamp=None*)

This returns a dictionary of multiple future OTP values of a token.

Parameters

- **count** – how many otp values should be returned
- **epoch_start** – time based tokens: start when
- **epoch_end** – time based tokens: stop when
- **curTime** (*datetime object*) – current time for TOTP token (for selftest)
- **timestamp** (*int*) – unix time, current time for TOTP token (for selftest)

Returns True/False, error text, OTP dictionary

Return type Tuple

get_otp(*current_time=''*)

The default token does not support getting the otp value will return a tuple of four values a negative value is a failure.

Returns something like: (1, pin, otpval, combined)

get_otp_count()

get_otp_count_window()

get_otplen()

get_pin_hash_seed()

get_realms()

Return a list of realms the token is assigned to :return: realms :rtype:l list

get_serial()

static get_setting_type(*key*)

This function returns the type of the token specific config/setting. This way a tokenclass can define settings, that can be "public" or a "password". If this setting is written to the database, the type of the setting is set automatically in set_privacyidea_config

The key name needs to start with the token type.

Parameters **key** – The token specific setting key

Returns A string like "public"

get_sync_window()

get_tokeninfo(*key=None*, *default=None*)

return the complete token info or a single key of the tokeninfo. When returning the complete token info dictionary encrypted entries are not decrypted. If you want to receive a decrypted value, you need to call it directly with the key.

Parameters

- **key** (*string*) – the key to return
- **default** (*string*) – the default value, if the key does not exist

Returns the value for the key

Return type int or string

get_tokentype()

get_type()

get_user_displayname()
Returns a tuple of a user identifier like user@realm and the displayname of "givenname surname".

Returns tuple

get_user_id()

get_validity_period_end()
returns the end of validity period (if set) if not set, "" is returned. :return: the end of the validity period :rtype: string

get_validity_period_start()
returns the start of validity period (if set) if not set, "" is returned. :return: the start of the validity period :rtype: string

hKeyRequired = False

inc_count_auth()
Increase the counter, that counts authentications - successful and unsuccessful

inc_count_auth_success()
Increase the counter, that counts successful authentications Also increase the auth counter

inc_failcount()

inc_otp_counter(*counter=None*, *increment=1*, *reset=True*)
Increase the otp counter and store the token in the database

Before increasing the token.count the token.count can be set using the parameter counter.

Parameters

- **counter** (*int*) – if given, the token counter is first set to counter and then increased by increment
- **increment** (*int*) – increase the counter by this amount
- **reset** (*bool*) – reset the failcounter if set to True

Returns the new counter value

is_active()

is_challenge_request(*passw*, *user=None*, *options=None*)
This method checks, if this is a request, that triggers a challenge.

The default behaviour to trigger a challenge is, if the `passw` parameter only contains the correct token pin *and* the request contains a `data` or a `challenge` key i.e. if the `options` parameter contains a key `data` or `challenge`.

Each token type can decide on its own under which condition a challenge is triggered by overwriting this method.

please note: in case of pin policy == 2 (no pin is required) the `check_pin` would always return true! Thus each request containing a `data` or `challenge` would trigger a challenge!

The Challenge workflow is like this.

When an authentication request is issued, first it is checked if this is a request which will create a new challenge (is_challenge_request) or if this is a response to an existing challenge (is_challenge_response). In these two cases during request processing the following functions are called.

is_challenge_request or is_challenge_response

|

V V

create_challenge check_challenge

|

V V

challenge_janitor challenge_janitor

Parameters

- **passw** (*string*) – password, which might be pin or pin+otp
- **user** (*User object*) – The user from the authentication request
- **options** (*dict*) – dictionary of additional request parameters

Returns true or false

Return type bool

is_challenge_response (*passw*, *user=None*, *options=None*)

This method checks, if this is a request, that is the response to a previously sent challenge.

The default behaviour to check if this is the response to a previous challenge is simply by checking if the request contains a parameter `state` or `transactionid` i.e. checking if the `options` parameter contains a key `state` or `transactionid`.

This method does not try to verify the response itself! It only determines, if this is a response for a challenge or not. The response is verified in check_challenge_response.

Parameters

- **passw** (*string*) – password, which might be pin or pin+otp
- **user** (*User object*) – the requesting user
- **options** (*dict*) – dictionary of additional request parameters

Returns true or false

Return type bool

is_fit_for_challenge (*messages*, *options=None*)

This method is called if a cryptographically matching response to a challenge was found. This method may implement final checks, if there is anything that should deny the success of the authentication with the response to the challenge.

The options dictionary can also contain the transaction_id, so even the challenge table for this token can be used for checking.

Parameters

- **options** (*dict*) –
- **messages** (*list*) – This is a list of messages. This method can append new information to this message list.

Returns True or False

is_locked()
Check if the token is in a locked state A locked token can not be modified

Returns True, if the token is locked.

is_orphaned()
Return True if the token is orphaned.

An orphaned token means, that it has a user assigned, but the user does not exist in the user store (anymore) :return: True / False

classmethod is_outofband()

is_pin_change(*password=False*)
Returns true if the pin of the token needs to be changed. :param password: Whether the password needs to be changed. :type password: bool

Returns True or False

is_previous_otp(*otp*, *window=10*)
checks if a given OTP value is a previous OTP value, that lies in the past or has a lower counter.

This is used in case of a failed authentication to return the information, that this OTP values was used previously and is invalid.

Parameters

- **otp** (*basestring*) – The OTP value.
- **window** (*int*) – A counter window, how far we should look into the past.

Returns bool

is_revoked()
Check if the token is in the revoked state

Returns True, if the token is revoked

mode = ['authenticate', 'challenge']

reset()
Reset the failcounter

resync(*otp1*, *otp2*, *options=None*)

revoke()
This revokes the token. By default it 1. sets the revoked-field 2. set the locked field 3. disables the token.

Some token types may revoke a token without locking it.

save()
Save the database token

set_count_auth(*count*)
Sets the counter for the occurred login attepms as key "count_auth" in token info :param count: a number :type count: int

set_count_auth_max(*count*)
Sets the counter for the maximum allowed login attempts as key "count_auth_max" in token info :param count: a number :type count: int

set_count_auth_success(*count*)
Sets the counter for the occurred successful logins as key "count_auth_success" in token info :param count: a number :type count: int

set_count_auth_success_max(*count*)
Sets the counter for the maximum allowed successful logins as key "count_auth_success_max" in token info :param count: a number :type count: int

set_count_window(*countWindow*)

set_defaults()
Set the default values on the database level

set_description(*description*)
Set the description on the database level

Parameters **description** (*string*) – description of the token

set_failcount(*failcount*)
Set the failcounter in the database

set_hashlib(*hashlib*)

set_init_details(*details*)

set_maxfail(*maxFail*)

set_next_pin_change(*diff=None*, *password=False*)
Sets the timestamp for the next_pin_change. Provide a difference like 90d (90 days).

Either provider the :param diff: The time delta. :type diff: basestring :param password: Do no set next_pin_change but next_password_change :return: None

set_otp_count(*otpCount*)

set_otpkey(*otpKey*)

set_otplen(*otplen*)

set_pin(*pin*, *encrypt=False*)
set the PIN of a token. Usually the pin is stored in a hashed way. :param pin: the pin to be set for the token :type pin: basestring :param encrypt: If set to True, the pin is stored encrypted and

can be retrieved from the database again

set_pin_hash_seed(*pinhash*, *seed*)

set_realms(*realms*, *add=False*)
Set the list of the realms of a token. :param realms: realms the token should be assigned to :type realms: list :param add: if the realms should be added and not replaced :type add: boolean

set_so_pin(*soPin*)

set_sync_window(*syncWindow*)

set_tokeninfo(*info*)
Set the tokeninfo field in the DB. Old values will be deleted. :param info: dictionary with key and value :type info: dict :return:

set_type(*tokentype*)
Set the tokentype in this object and also in the underlying database-Token-object.

Parameters **tokentype** (*string*) – The type of the token like HOTP or TOTP

set_user_pin(*userPin*)

set_validity_period_end(*end_date*)

sets the end date of the validity period for a token :param end_date: the end date in the format YYYY-MM-DDTHH:MM+OOOO

if the format is wrong, the method will throw an exception

set_validity_period_start(*start_date*)

sets the start date of the validity period for a token :param start_date: the start date in the format YYYY-MM-DDTHH:MM+OOOO

if the format is wrong, the method will throw an exception

split_pin_pass(*passw*, *user=None*, *options=None*)

Split the password into the token PIN and the OTP value

take the given password and split it into the PIN and the OTP value. The splitting can be dependent of certain policies. The policies may depend on the user.

Each token type may define its own way to slit the PIN and the OTP value.

Parameters

- **passw** – the password to split
- **user** (*User object*) – The user/owner of the token
- **options** (*dict*) – can be used be the token types.

Returns tuple of pin and otp value

Returns tuple of (split status, pin, otp value)

Return type tuple

status_validation_fail()

callback to enable a status change, if auth failed

status_validation_success()

callback to enable a status change, if auth succeeds

static test_config(*params=None*)

This method is used to test the token config. Some tokens require some special token configuration like the SMS-Token or the Email-Token. To test this configuration, this classmethod is used.

It takes token specific parameters and returns a tuple of a boolean and a result description.

Parameters **params** (*dict*) – token specific parameters

Returns success, description

Return type tuple

update(*param*, *reset_failcount=True*)

Update the token object

Parameters **param** – a dictionary with different params like keysize, description, genkey, otpkey, pin

Type param: dict

property user

return the user (owner) of a token If the token has no owner assigned, we return None

Returns The owner of the token

Return type User object or None

using_pin = True

Token Functions

This module contains all top level token functions. It depends on the models, lib.user and lib.tokenclass (which depends on the tokenclass implementations like lib.tokens.hotptoken)

This is the middleware/glue between the HTTP API and the database

`privacyidea.lib.token.`**`add_tokeninfo`**(*serial*, *info*, *value=None*, *value_type=None*, *user=None*)

Sets a token info field in the database. The info is a dict for each token of key/value pairs.

Parameters

- **serial** (*basestring*) – The serial number of the token
- **info** – The key of the info in the dict
- **value** – The value of the info
- **value_type** – The type of the value. If set to "password" the value

is stored encrypted :type value_type: basestring :param user: The owner of the tokens, that should be modified :type user: User object :return: the number of modified tokens :rtype: int

`privacyidea.lib.token.`**`assign_token`**(*serial*, *user*, *pin=None*, *encrypt_pin=False*, *err_message=None*)

Assign token to a user. If the PIN is given, the PIN is reset.

Parameters

- **serial** (*basestring*) – The serial number of the token
- **user** (*User object*) – The user, to whom the token should be assigned.
- **pin** (*basestring*) – The PIN for the newly assigned token.
- **encrypt_pin** (*bool*) – Whether the PIN should be stored in an encrypted way
- **err_message** (*basestring*) – The error message, that is displayed in case the token is already assigned

`privacyidea.lib.token.`**`check_otp`**(*serial*, *otpval*)

This function checks the OTP for a given serial number :param serial: :param otpval: :return:

`privacyidea.lib.token.`**`check_realm_pass`**(*realm*, *passw*, *options=None*)

This function checks, if the given passw matches any token in the given realm. This can be used for the 4-eyes token. Only tokens that are assigned are tested.

It returns the res True/False and a reply_dict, which contains the serial number of the matching token.

Parameters

- **realm** – The realm of the user
- **passw** – The password containing PIN+OTP
- **options** (*dict*) – Additional options that are passed to the tokens

Returns tuple of bool and dict

privacyidea.lib.token.**check_serial**(*serial*)

This checks, if the given serial number can be used for a new token. it returns a tuple (result, new_serial) result being True if the serial does not exist, yet. new_serial is a suggestion for a new serial number, that does not exist, yet.

Parameters **serial** – Seral number that is to be checked, if it can be used for

a new token. :type serial: string :result: bool and serial number :rtype: tuple

privacyidea.lib.token.**check_serial_pass**(*serial*, *passw*, *options=None*)

This function checks the otp for a given serial

If the OTP matches, True is returned and the otp counter is increased.

The function tries to determine the user (token owner), to derive possible additional policies from the user.

Parameters

- **serial** (*basestring*) – The serial number of the token
- **passw** (*basestring*) – The password usually consisting of pin + otp
- **options** (*dict*) – Additional options. Token specific.

Returns tuple of result (True, False) and additional dict

Return type tuple

privacyidea.lib.token.**check_token_list**(*tokenobject_list*, *passw*, *user=None*, *options=None*, *allow_reset_all_tokens=False*)

this takes a list of token objects and tries to find the matching token for the given passw. It also tests, * if the token is active or * the max fail count is reached, * if the validity period is ok…

This function is called by check_serial_pass, check_user_pass and check_yubikey_pass.

Parameters

- **tokenobject_list** – list of identified tokens
- **passw** – the provided passw (mostly pin+otp)
- **user** – the identified use - as class object
- **options** – additional parameters, which are passed to the token
- **allow_reset_all_tokens** – If set to True, the policy reset_all_user_tokens is evaluated to reset all user tokens accordingly. Note: This parameter is used in the decorator.

Returns tuple of success and optional response

Return type (bool, dict)

privacyidea.lib.token.**check_user_pass**(*user*, *passw*, *options=None*)

This function checks the otp for a given user. It is called by the API /validate/check

If the OTP matches, True is returned and the otp counter is increased.

Parameters

- **user** (*User object*) – The user who is trying to authenticate
- **passw** (*basestring*) – The password usually consisting of pin + otp
- **options** (*dict*) – Additional options. Token specific.

Returns tuple of result (True, False) and additional dict

Return type tuple

`privacyidea.lib.token.`**`copy_token_pin`**(*serial_from*, *serial_to*)

This function copies the token PIN from one token to the other token. This can be used for workflows like lost token.

In fact the PinHash and the PinSeed are transferred

Parameters

- **`serial_from`** (*`basestring`*) – The token to copy from
- **`serial_to`** (*`basestring`*) – The token to copy to

Returns True. In case of an error raise an exception

Return type bool

`privacyidea.lib.token.`**`copy_token_realms`**(*serial_from*, *serial_to*)

Copy the realms of one token to the other token

Parameters

- **`serial_from`** – The token to copy from
- **`serial_to`** – The token to copy to

Returns None

`privacyidea.lib.token.`**`copy_token_user`**(*serial_from*, *serial_to*)

This function copies the user from one token to the other token. In fact the user_id, resolver and resolver type are transferred.

Parameters

- **`serial_from`** (*`basestring`*) – The token to copy from
- **`serial_to`** (*`basestring`*) – The token to copy to

Returns True. In case of an error raise an exception

Return type bool

`privacyidea.lib.token.`**`create_challenges_from_tokens`**(*token_list*, *reply_dict*, *options=None*)

Get a list of active tokens and create challenges for these tokens. The reply_dict is modified accordingly. The transaction_id and the messages are added to the reply_dict.

Parameters

- **`token_list`** – The list of the token objects, that can do challenge response
- **`reply_dict`** – The dictionary that is passed to the API response
- **`options`** – Additional options. Passed from the upper layer

Returns None

`privacyidea.lib.token.`**`create_tokenclass_object`**(*db_token*)

(was createTokenClassObject) create a token class object from a given type If a tokenclass for this type does not exist, the function returns None.

Parameters **`db_token`** (*`database token object`*) – the database referenced token

Returns instance of the token class object

Return type tokenclass object

privacyidea.lib.token.**delete_tokeninfo**(*serial*, *key*, *user=None*)

Delete a specific token info field in the database.

Parameters

- **serial** (*basestring*) – The serial number of the token
- **key** – The key of the info in the dict
- **value** – The value of the info
- **user** (*User object*) – The owner of the tokens, that should be modified

Returns the number of tokens matching the serial and user. This number also includes tokens that did not have

the token info *key* set in the first place! :rtype: int

privacyidea.lib.token.**enable_token**(*serial*, *enable=True*, *user=None*)

Enable or disable a token, or all tokens of a single user. This can be checked with is_token_active.

Enabling an already active token will return 0.

Parameters

- **serial** (*basestring*) – The serial number of the token
- **enable** (*bool*) – False is the token should be disabled
- **user** (*User object*) – all tokens of the user will be enabled or disabled

Returns Number of tokens that were enabled/disabled

Return type

privacyidea.lib.token.**gen_serial**(*tokentype=None*, *prefix=None*)

generate a serial for a given tokentype

Parameters

- **tokentype** (*str*) – the token type prefix is done by a lookup on the tokens
- **prefix** (*str*) – A prefix to the serial number

Returns serial number

Return type str

privacyidea.lib.token.**get_dynamic_policy_definitions**(*scope=None*)

This returns the dynamic policy definitions that come with the new loaded token classes.

Parameters **scope** – an optional scope parameter. Only return the policies of

this scope. :return: The policy definition for the token or only for the scope.

privacyidea.lib.token.**get_multi_otp**(*serial*, *count=0*, *epoch_start=0*, *epoch_end=0*, *curTime=None*, *timestamp=None*)

This function returns a list of OTP values for the given Token. Please note, that the tokentype needs to support this function.

Parameters

- **serial** (*basestring*) – the serial number of the token
- **count** – number of the next otp values (to be used with event or time based tokens)
- **epoch_start** – unix time start date (used with time based tokens)
- **epoch_end** – unix time end date (used with time based tokens)

- **curTime** (*datetime*) – Simulate the servertime
- **timestamp** (*int*) – Simulate the servertime (unix time in seconds)

Returns dictionary of otp values

Return type dictionary

privacyidea.lib.token.**get_num_tokens_in_realm**(*realm*, *active=True*)

This returns the number of tokens in one realm. :param realm: The name of the realm :type realm: basestring :param active: If only active tokens should be taken into account :type active: bool :return: The number of tokens in the realm :rtype: int

privacyidea.lib.token.**get_one_token**(**args*, ***kwargs*)

Fetch exactly one token according to the given filter arguments, which are passed to `get_tokens`. Raise `ResourceNotFoundError` if no token was found. Raise `ParameterError` if more than one token was found.

privacyidea.lib.token.**get_otp**(*serial*, *current_time=None*)

This function returns the current OTP value for a given Token. The tokentype needs to support this function. if the token does not support getting the OTP value, a -2 is returned. If the token could not be found, ResourceNotFoundError is raised.

Parameters

- **serial** – serial number of the token
- **current_time** (*datetime*) – a fake servertime for testing of TOTP token

Returns tuple with (result, pin, otpval, passw)

Return type tuple

privacyidea.lib.token.**get_realms_of_token**(*serial*, *only_first_realm=False*)

This function returns a list of the realms of a token

Parameters

- **serial** (*basestring*) – the exact serial number of the token
- **only_first_realm** (*bool*) – Wheather we should only return the first realm

Returns list of the realm names

Return type list

privacyidea.lib.token.**get_serial_by_otp**(*token_list*, *otp=''*, *window=10*)

Returns the serial for a given OTP value The tokenobject_list would be created by get_tokens()

Parameters

- **token_list** (*list of token objects*) – the list of token objects to be investigated
- **otp** – the otp value, that needs to be found
- **window** (*int*) – the window of search

Returns the serial for a given OTP value and the user

Return type basestring

privacyidea.lib.token.**get_token_by_otp**(*token_list*, *otp=''*, *window=10*)

search the token in the token_list, that creates the given OTP value. The tokenobject_list would be created by get_tokens()

Parameters

- **token_list** (*list of token objects*) – the list of token objects to be investigated
- **otp** (*basestring*) – the otp value, that needs to be found
- **window** (*int*) – the window of search

Returns The token, that creates this OTP value

Return type Tokenobject

privacyidea.lib.token.**get_token_owner**(*serial*)

returns the user object, to which the token is assigned. the token is identified and retrieved by it's serial number

If the token has no owner, None is returned

Wildcards in the serial number are ignored. This raises ResourceNotFoundError if the token could not be found.

Parameters **serial** (*basestring*) – serial number of the token

Returns The owner of the token

Return type User object or None

privacyidea.lib.token.**get_token_type**(*serial*)

Returns the tokentype of a given serial number. If the token does not exist or can not be deterimined, an empty string is returned.

Parameters **serial** (*string*) – the serial number of the to be searched token

Returns tokentype

Return type string

privacyidea.lib.token.**get_tokenclass_info**(*tokentype*, *section=None*)

return the config definition of a dynamic token

Parameters

- **tokentype** (*basestring*) – the tokentype of the token like "totp" or "hotp"
- **section** (*basestring*) – subsection of the token definition - optional

Returns dict - if nothing found an empty dict

Return type dict

privacyidea.lib.token.**get_tokens**(*tokentype=None*, *realm=None*, *assigned=None*, *user=None*, *serial=None*, *serial_wildcard=None*, *active=None*, *resolver=None*, *rollout_state=None*, *count=False*, *revoked=None*, *locked=None*, *tokeninfo=None*, *maxfail=None*)

(was getTokensOfType) This function returns a list of token objects of a * given type, * of a realm * or tokens with assignment or not * for a certain serial number or * for a User

E.g. thus you can get all assigned tokens of type totp.

Parameters

- **tokentype** (*basestring*) – The type of the token. If None, all tokens are returned.
- **realm** (*basestring*) – get tokens of a realm. If None, all tokens are returned.
- **assigned** (*bool*) – Get either assigned (True) or unassigned (False) tokens. If None get all tokens.
- **user** (*User Object*) – Filter for the Owner of the token

- **serial** (*basestring*) – The exact serial number of a token
- **serial_wildcard** (*basestring*) – A wildcard to match token serials
- **active** (*bool*) – Whether only active (True) or inactive (False) tokens should be returned
- **resolver** (*basestring*) – filter for the given resolver name
- **rollout_state** – returns a list of the tokens in the certain rollout state. Some tokens are not enrolled in a single step but in multiple steps. These tokens are then identified by the DB-column rollout_state.
- **count** (*bool*) – If set to True, only the number of the result and not the list is returned.
- **revoked** (*bool*) – Only search for revoked tokens or only for not revoked tokens
- **locked** (*bool*) – Only search for locked tokens or only for not locked tokens
- **tokeninfo** (*dict*) – Return tokens with the given tokeninfo. The tokeninfo is a key/value dictionary
- **maxfail** – If only tokens should be returned, which failcounter reached maxfail

Returns A list of tokenclasses (lib.tokenclass).

Return type list

privacyidea.lib.token.**get_tokens_from_serial_or_user**(*serial*, *user*, ***kwargs*)

Fetch tokens, either by (exact) serial, or all tokens of a single user. In case a serial number is given, check that exactly one token is returned and raise a ResourceNotFoundError if that is not the case. In case a user is given, the result can also be empty. :param serial: exact serial number or None :param user: a user object or None :param kwargs: additional argumens to `get_tokens` :return: a (possibly empty) list of tokens

privacyidea.lib.token.**get_tokens_in_resolver**(*resolver*)

Return a list of the token ojects, that contain this very resolver

Parameters **resolver** (*basestring*) – The resolver, the tokens should be in

Returns list of tokens with this resolver

Return type list of token objects

privacyidea.lib.token.**get_tokens_paginate**(*tokentype=None*, *realm=None*, *assigned=None*, *user=None*, *serial=None*, *active=None*, *resolver=None*, *rollout_state=None*, *sortby=<sqlalchemy.orm.attributes.InstrumentedAttribute object>*, *sortdir='asc'*, *psize=15*, *page=1*, *description=None*, *userid=None*, *allowed_realms=None*)

This function is used to retrieve a token list, that can be displayed in the Web UI. It supports pagination. Each retrieved page will also contain a "next" and a "prev", indicating the next or previous page. If either does not exist, it is None.

Parameters

- **tokentype** –
- **realm** –
- **assigned** (*bool*) – Returns assigned (True) or not assigned (False) tokens
- **user** (*User object*) – The user, whose token should be displayed
- **serial** – a pattern for matching the serial
- **active** –

- **resolver** (*basestring*) – A resolver name, which may contain "*" for filtering.
- **userid** (*basestring*) – A userid, which may contain "*" for filtering.
- **rollout_state** –
- **sortby** (*A Token column or a string.*) – Sort by a certain Token DB field. The default is Token.serial. If a string like "serial" is provided, we try to convert it to the DB column.
- **sortdir** (*basestring*) – Can be "asc" (default) or "desc"
- **psize** (*int*) – The size of the page
- **page** (*int*) – The number of the page to view. Starts with 1 ;-)
- **allowed_realms** (*list*) – A list of realms, that the admin is allowed to see

Returns dict with tokens, prev, next and count

Return type dict

privacyidea.lib.token.**get_tokens_paginated_generator**(*tokentype=None*, *realm=None*, *assigned=None*, *user=None*, *serial_wildcard=None*, *active=None*, *resolver=None*, *rollout_state=None*, *revoked=None*, *locked=None*, *tokeninfo=None*, *maxfail=None*, *psize=1000*)

Fetch chunks of psize tokens that match the filter criteria from the database and generate lists of token objects. See get_tokens for information on the arguments.

Note that individual lists may contain less than psize elements if a token entry has an invalid type.

Parameters **psize** – Maximum size of chunks that are fetched from the database

Returns This is a generator that generates non-empty lists of token objects.

privacyidea.lib.token.**import_token**(*serial*, *token_dict*, *default_hashlib=None*, *tokenrealms=None*)

This function is used during the import of a PSKC file.

Parameters

- **serial** – The serial number of the token
- **token_dict** – A dictionary describing the token like: {"type": ...,

  "description": ..., "otpkey": ..., "counter: ..., "timeShift": ... }
- **default_hashlib** –

Returns the token object

privacyidea.lib.token.**init_token**(*param*, *user=None*, *tokenrealms=None*, *tokenkind=None*)

create a new token or update an existing token

Parameters

- **param** (*dict*) – initialization parameters like: serial (optional) type (optionl, default=hotp) otpkey
- **user** (*User Object*) – the token owner
- **tokenrealms** (*list*) – the realms, to which the token should belong

- **tokenkind** – The kind of the token, can be "software", "hardware" or "virtual"

Returns token object or None

Return type TokenClass object

privacyidea.lib.token.**is_token_active**(*serial*)

Return True if the token is active, otherwise false Raise ResourceError if the token could not be found.

Parameters **serial** (*basestring*) – The serial number of the token

Returns True or False

Return type bool

privacyidea.lib.token.**is_token_owner**(*serial*, *user*)

Check if the given user is the owner of the token with the given serial number :param serial: The serial number of the token :type serial: str :param user: The user that needs to be checked :type user: User object :return: Return True or False :rtype: bool

privacyidea.lib.token.**lost_token**(*serial*, *new_serial=None*, *password=None*, *validity=10*, *contents='8'*, *pw_len=16*, *options=None*)

This is the workflow to handle a lost token. The token <serial> is lost and will be disabled. A new token of type password token will be created and assigned to the user. The PIN of the lost token will be copied to the new token. The new token will have a certain validity period.

Parameters

- **serial** – Token serial number
- **new_serial** – new serial number
- **password** – new password
- **validity** (*int*) – Number of days, the new token should be valid
- **contents** – The contents of the generated password. "C": upper case

characters, "c": lower case characters, "n": digits and "s": special characters, "8": base58 :type contents: A string like "Ccn" :param pw_len: The length of the generated password :type pw_len: int :param options: optional values for the decorator passed from the upper API level :type options: dict

Returns result dictionary

privacyidea.lib.token.**remove_token**(*serial=None*, *user=None*)

remove the token that matches the serial number or all tokens of the given user and also remove the realm associations and all its challenges

Parameters

- **user** (*User object*) – The user, who's tokens should be deleted.
- **serial** (*basestring*) – The serial number of the token to delete (exact)

Returns The number of deleted token

Return type int

privacyidea.lib.token.**reset_token**(*serial*, *user=None*)

Reset the failcounter of a single token, or of all tokens of one user. :param serial: serial number (exact) :param user: :return: The number of tokens, that were resetted :rtype: int

privacyidea.lib.token.**resync_token**(*serial*, *otp1*, *otp2*, *options=None*, *user=None*)

Resyncronize the token of the given serial number and user by searching the otp1 and otp2 in the future otp values.

Parameters

- **serial** (*basestring*) – token serial number (exact)
- **otp1** (*basestring*) – first OTP value
- **otp2** (*basestring*) – second OTP value, directly after the first
- **options** (*dict*) – additional options like the servertime for TOTP token

Returns

`privacyidea.lib.token.`**`revoke_token`**(*serial*, *user=None*)

Revoke a token, or all tokens of a single user.

Parameters

- **serial** (*basestring*) – The serial number of the token (exact)
- **enable** (*bool*) – False is the token should be disabled
- **user** (*User object*) – all tokens of the user will be enabled or disabled

Returns Number of tokens that were enabled/disabled

Return type

`privacyidea.lib.token.`**`set_count_auth`**(*serial*, *count*, *user=None*, *max=False*, *success=False*)

The auth counters are stored in the token info database field. There are different counters, that can be set

count_auth -> max=False, success=False count_auth_max -> max=True, success=False count_auth_success -> max=False, success=True count_auth_success_max -> max=True, success=True

Parameters

- **count** (*int*) – The counter value
- **user** (*User object*) – The user owner of the tokens tokens to modify
- **serial** (*basestring*) – The serial number of the one token to modifiy (exact)
- **max** – True, if either count_auth_max or count_auth_success_max are

to be modified :type max: bool :param success: True, if either count_auth_success or count_auth_success_max are to be modified :type success: bool :return: number of modified tokens :rtype: int

`privacyidea.lib.token.`**`set_count_window`**(*serial*, *countwindow=10*, *user=None*)

The count window is used during authentication to find the matching OTP value. This sets the count window per token.

Parameters

- **serial** (*basestring*) – The serial number of the token (exact)
- **countwindow** (*int*) – the size of the window
- **user** (*User object*) – The owner of the tokens, which should be modified

Returns number of modified tokens

Return type int

`privacyidea.lib.token.`**`set_defaults`**(*serial*)

Set the default values for the token with the given serial number (exact) :param serial: token serial :type serial: basestring :return: None

`privacyidea.lib.token.`**`set_description`**(*serial*, *description*, *user=None*)

Set the description of a token

Parameters

- **serial** (*basestring*) – The serial number of the token (exact)
- **description** (*str*) – The description for the token
- **user** (*User object*) – The owner of the tokens, which should be modified

Returns number of modified tokens

Return type int

`privacyidea.lib.token.`**`set_failcounter`**(*serial*, *counter*, *user=None*)

Set the fail counter of a token.

Parameters

- **serial** – The serial number of the token (exact)
- **counter** – THe counter to which the fail counter should be set
- **user** – An optional user

Returns Number of tokens, where the fail counter was set.

`privacyidea.lib.token.`**`set_hashlib`**(*serial*, *hashlib='sha1'*, *user=None*)

Set the hashlib in the tokeninfo. Can be something like sha1, sha256…

Parameters

- **serial** (*basestring*) – The serial number of the token (exact)
- **hashlib** (*basestring*) – The hashlib of the token
- **user** (*User object*) – The User, for who's token the hashlib should be set

Returns the number of token infos set

Return type int

`privacyidea.lib.token.`**`set_max_failcount`**(*serial*, *maxfail*, *user=None*)

Set the maximum fail counts of tokens. This is the maximum number a failed authentication is allowed.

Parameters

- **serial** (*basestring*) – The serial number of the token (exact)
- **maxfail** (*int*) – The maximum allowed failed authentications
- **user** (*User object*) – The owner of the tokens, which should be modified

Returns number of modified tokens

Return type int

`privacyidea.lib.token.`**`set_otplen`**(*serial*, *otplen=6*, *user=None*)

Set the otp length of the token defined by serial or for all tokens of the user. The OTP length is usually 6 or 8.

Parameters

- **serial** (*basestring*) – The serial number of the token (exact)
- **otplen** (*int*) – The length of the OTP value
- **user** (*User object*) – The owner of the tokens

Returns number of modified tokens

Return type int

`privacyidea.lib.token.`**`set_pin`**(*serial*, *pin*, *user=None*, *encrypt_pin=False*)

Set the token PIN of the token. This is the static part that can be used to authenticate.

Parameters

- **`pin`** (*`basestring`*) – The pin of the token
- **`user`** – If the user is specified, the pins for all tokens of this

user will be set :type user: User object :param serial: If the serial is specified, the PIN for this very token will be set. (exact) :return: The number of PINs set (usually 1) :rtype: int

`privacyidea.lib.token.`**`set_pin_so`**(*serial*, *so_pin*, *user=None*)

Set the SO PIN of a smartcard. The SO Pin can be used to reset the PIN of a smartcard. The SO PIN is stored in the database, so that it could be used for automatic processes for User PIN resetting.

Parameters

- **`serial`** (*`basestring`*) – The serial number of the token (exact)
- **`so_pin`** (*`basestring`*) – The Security Officer PIN

Returns The number of SO PINs set. (usually 1)

Return type int

`privacyidea.lib.token.`**`set_pin_user`**(*serial*, *user_pin*, *user=None*)

This sets the user pin of a token. This just stores the information of the user pin for (e.g. an eTokenNG, Smartcard) in the database

Parameters

- **`serial`** (*`basestring`*) – The serial number of the token (exact)
- **`user_pin`** (*`basestring`*) – The user PIN

Returns The number of PINs set (usually 1)

Return type int

`privacyidea.lib.token.`**`set_realms`**(*serial*, *realms=None*, *add=False*)

Set all realms of a token. This sets the realms new. I.e. it does not add realms. So realms that are not contained in the list will not be assigned to the token anymore.

If the token could not be found, a ResourceNotFoundError is raised.

Thus, setting realms=[] clears all realms assignments.

Parameters

- **`serial`** (*`basestring`*) – the serial number of the token (exact)
- **`realms`** (*`list`*) – A list of realm names
- **`add`** (*`bool`*) – if the realms should be added and not replaced

`privacyidea.lib.token.`**`set_sync_window`**(*serial*, *syncwindow=1000*, *user=None*)

The sync window is the window that is used during resync of a token. Such many OTP values are calculated ahead, to find the matching otp value and counter.

Parameters

- **`serial`** (*`basestring`*) – The serial number of the token (exact)
- **`syncwindow`** (*`int`*) – The size of the sync window
- **`user`** (*`User object`*) – The owner of the tokens, which should be modified

Returns number of modified tokens

Return type int

`privacyidea.lib.token.`**`set_validity_period_end`**(*serial*, *user*, *end*)

Set the validity period for the given token.

Parameters

- **`serial`** – serial number (exact)
- **`user`** –
- **`end`** (*`basestring`*) – Timestamp in the format DD/MM/YY HH:MM

`privacyidea.lib.token.`**`set_validity_period_start`**(*serial*, *user*, *start*)

Set the validity period for the given token.

Parameters

- **`serial`** – serial number (exact)
- **`user`** –
- **`start`** (*`basestring`*) – Timestamp in the format DD/MM/YY HH:MM

`privacyidea.lib.token.`**`token_exist`**(*serial*)

returns true if the token with the exact given serial number exists

Parameters **`serial`** – the serial number of the token

`privacyidea.lib.token.`**`unassign_token`**(*serial*, *user=None*)

unassign the user from the token, or all tokens of a user

Parameters

- **`serial`** – The serial number of the token to unassign (exact). Can be None
- **`user`** – A user whose tokens should be unassigned

Returns number of unassigned tokens

Application Class

`privacyidea.lib.applications.`**`MachineApplicationBase`**

alias of `privacyidea.lib.applications.base.MachineApplication`

Policy Module

Base function to handle the policy entries in the database. This module only depends on the db/models.py

The functions of this module are tested in tests/test_lib_policy.py

A policy has the attributes

- name
- scope
- action
- realm
- resolver

- user
- client
- active

`name` is the unique identifier of a policy. `scope` is the area, where this policy is meant for. This can be values like admin, selfservice, authentication… `scope` takes only one value.

`active` is bool and indicates, whether a policy is active or not.

`action`, `realm`, `resolver`, `user` and `client` can take a comma separated list of values.

realm and resolver

If these are empty '*', this policy matches each requested realm.

user

If the user is empty or '*', this policy matches each user. You can exclude users from matching this policy, by prepending a '-' or a '!'. `*, -admin` will match for all users except the admin.

You can also use regular expressions to match the user like `customer_.*` to match any user, starting with *customer_*.

Note: Regular expression will only work for exact machtes. *user1234* will not match *user1* but only *user1…*

client

The client is identified by its IP address. A policy can contain a list of IP addresses or subnets. You can exclude clients from subnets by prepending the client with a '-' or a '!'. `172.16.0.0/24, -172.16.0.17` will match each client in the subnet except the 172.16.0.17.

time

You can specify a time in which the policy should be active. Time formats are

<dow>-<dow>:<hh>:<mm>-<hh>:<mm>, … <dow>:<hh>:<mm>-<hh>:<mm> <dow>:<hh>-<hh>

and any combination of it. "dow" being day of week Mon, Tue, Wed, Thu, Fri, Sat, Sun.

class `privacyidea.lib.policy.`**`ACTION`**

This is the list of usual actions.

`ADDRESOLVERINRESPONSE = 'add_resolver_in_response'`

`ADDUSER = 'adduser'`

`ADDUSERINRESPONSE = 'add_user_in_response'`

`APIKEY = 'api_key_required'`

`APPIMAGEURL = 'appimageurl'`

`ASSIGN = 'assign'`

`AUDIT = 'auditlog'`

AUDIT_AGE = 'auditlog_age'

AUDIT_DOWNLOAD = 'auditlog_download'

AUTHITEMS = 'fetch_authentication_items'

AUTHMAXFAIL = 'auth_max_fail'

AUTHMAXSUCCESS = 'auth_max_success'

AUTH_CACHE = 'auth_cache'

AUTOASSIGN = 'autoassignment'

CACONNECTORDELETE = 'caconnectordelete'

CACONNECTORREAD = 'caconnectorread'

CACONNECTORWRITE = 'caconnectorwrite'

CHALLENGERESPONSE = 'challenge_response'

CHALLENGETEXT = 'challenge_text'

CHALLENGETEXT_FOOTER = 'challenge_text_footer'

CHALLENGETEXT_HEADER = 'challenge_text_header'

CHANGE_PIN_EVERY = 'change_pin_every'

CHANGE_PIN_FIRST_USE = 'change_pin_on_first_use'

CLIENTTYPE = 'clienttype'

CONFIGDOCUMENTATION = 'system_documentation'

COPYTOKENPIN = 'copytokenpin'

COPYTOKENUSER = 'copytokenuser'

CUSTOM_BASELINE = 'custom_baseline'

CUSTOM_MENU = 'custom_menu'

DEFAULT_TOKENTYPE = 'default_tokentype'

DELETE = 'delete'

DELETEUSER = 'deleteuser'

DIALOG_NO_TOKEN = 'dialog_no_token'

DISABLE = 'disable'

EMAILCONFIG = 'smtpconfig'

ENABLE = 'enable'

ENCRYPTPIN = 'encrypt_pin'

ENROLLPIN = 'enrollpin'

EVENTHANDLINGREAD = 'eventhandling_read'

EVENTHANDLINGWRITE = 'eventhandling_write'

FORCE_APP_PIN = 'force_app_pin'

GETCHALLENGES = 'getchallenges'

GETRANDOM = 'getrandom'

```
GETSERIAL = 'getserial'

HIDE_BUTTONS = 'hide_buttons'

HIDE_WELCOME = 'hide_welcome_info'

IMPORT = 'importtokens'

LASTAUTH = 'last_auth'

LOGINMODE = 'login_mode'

LOGIN_TEXT = 'login_text'

LOGOUTTIME = 'logout_time'

LOSTTOKEN = 'losttoken'

LOSTTOKENPWCONTENTS = 'losttoken_PW_contents'

LOSTTOKENPWLEN = 'losttoken_PW_length'

LOSTTOKENVALID = 'losttoken_valid'

MACHINELIST = 'machinelist'

MACHINERESOLVERDELETE = 'mresolverdelete'

MACHINERESOLVERREAD = 'mresolverread'

MACHINERESOLVERWRITE = 'mresolverwrite'

MACHINETOKENS = 'manage_machine_tokens'

MANAGESUBSCRIPTION = 'managesubscription'

MANGLE = 'mangle'

MAXACTIVETOKENUSER = 'max_active_token_per_user'

MAXTOKENREALM = 'max_token_per_realm'

MAXTOKENUSER = 'max_token_per_user'

NODETAILFAIL = 'no_detail_on_fail'

NODETAILSUCCESS = 'no_detail_on_success'

OTPPIN = 'otppin'

OTPPINCONTENTS = 'otp_pin_contents'

OTPPINMAXLEN = 'otp_pin_maxlength'

OTPPINMINLEN = 'otp_pin_minlength'

OTPPINRANDOM = 'otp_pin_random'

OTPPINSETRANDOM = 'otp_pin_set_random'

PASSNOTOKEN = 'passOnNoToken'

PASSNOUSER = 'passOnNoUser'

PASSTHRU = 'passthru'

PASSTHRU_ASSIGN = 'passthru_assign'

PASSWORDRESET = 'password_reset'

PERIODICTASKREAD = 'periodictask_read'
```

PERIODICTASKWRITE = 'periodictask_write'

PINHANDLING = 'pinhandling'

POLICYDELETE = 'policydelete'

POLICYREAD = 'policyread'

POLICYTEMPLATEURL = 'policy_template_url'

POLICYWRITE = 'policywrite'

PRIVACYIDEASERVERREAD = 'privacyideaserver_read'

PRIVACYIDEASERVERWRITE = 'privacyideaserver_write'

RADIUSSERVERREAD = 'radiusserver_read'

RADIUSSERVERWRITE = 'radiusserver_write'

REALM = 'realm'

REALMDROPDOWN = 'realm_dropdown'

REGISTERBODY = 'registration_body'

REMOTE_USER = 'remote_user'

REQUIREDEMAIL = 'requiredemail'

RESET = 'reset'

RESETALLTOKENS = 'reset_all_user_tokens'

RESOLVER = 'resolver'

RESOLVERDELETE = 'resolverdelete'

RESOLVERREAD = 'resolverread'

RESOLVERWRITE = 'resolverwrite'

RESYNC = 'resync'

REVOKE = 'revoke'

SEARCH_ON_ENTER = 'search_on_enter'

SERIAL = 'serial'

SET = 'set'

SETDESCRIPTION = 'setdescription'

SETHSM = 'set_hsm_password'

SETPIN = 'setpin'

SETRANDOMPIN = 'setrandompin'

SETREALM = 'setrealm'

SETTOKENINFO = 'settokeninfo'

SHOW_ANDROID_AUTHENTICATOR = 'show_android_privacyidea_authenticator'

SHOW_CUSTOM_AUTHENTICATOR = 'show_custom_authenticator'

SHOW_IOS_AUTHENTICATOR = 'show_ios_privacyidea_authenticator'

SHOW_SEED = 'show_seed'

SMSGATEWAYREAD = 'smsgateway_read'

SMSGATEWAYWRITE = 'smsgateway_write'

SMTPSERVERREAD = 'smtpserver_read'

SMTPSERVERWRITE = 'smtpserver_write'

STATISTICSDELETE = 'statistics_delete'

STATISTICSREAD = 'statistics_read'

SYSTEMDELETE = 'configdelete'

SYSTEMREAD = 'configread'

SYSTEMWRITE = 'configwrite'

TIMEOUT_ACTION = 'timeout_action'

TOKENINFO = 'tokeninfo'

TOKENISSUER = 'tokenissuer'

TOKENLABEL = 'tokenlabel'

TOKENLIST = 'tokenlist'

TOKENPAGESIZE = 'token_page_size'

TOKENREALMS = 'tokenrealms'

TOKENTYPE = 'tokentype'

TOKENWIZARD = 'tokenwizard'

TOKENWIZARD2ND = 'tokenwizard_2nd_token'

TRIGGERCHALLENGE = 'triggerchallenge'

UNASSIGN = 'unassign'

UPDATEUSER = 'updateuser'

USERDETAILS = 'user_details'

USERLIST = 'userlist'

USERPAGESIZE = 'user_page_size'

class privacyidea.lib.policy.**ACTIONVALUE**

This is a list of usual action values for e.g. policy action-values like otppin.

DISABLE = 'disable'

NONE = 'none'

TOKENPIN = 'tokenpin'

USERSTORE = 'userstore'

class privacyidea.lib.policy.**AUTOASSIGNVALUE**

This is the possible values for autoassign

NONE = 'any_pin'

USERSTORE = 'userstore'

class privacyidea.lib.policy.**CONDITION_SECTION**

This is a list of available sections for conditions of policies

HTTP_REQUEST_HEADER = 'HTTP Request header'

USERINFO = 'userinfo'

class privacyidea.lib.policy.**GROUP**

These are the allowed policy action groups. The policies will be grouped in the UI.

ENROLLMENT = 'enrollment'

GENERAL = 'general'

MACHINE = 'machine'

PIN = 'pin'

SYSTEM = 'system'

TOKEN = 'token'

TOOLS = 'tools'

USER = 'user'

class privacyidea.lib.policy.**LOGINMODE**

This is the list of possible values for the login mode.

DISABLE = 'disable'

PRIVACYIDEA = 'privacyIDEA'

USERSTORE = 'userstore'

class privacyidea.lib.policy.**MAIN_MENU**

These are the allowed top level menu items. These are used to toggle the visibility of the menu items depending on the rights of the user

AUDIT = 'audit'

COMPONENTS = 'components'

CONFIG = 'config'

MACHINES = 'machines'

TOKENS = 'tokens'

USERS = 'users'

class privacyidea.lib.policy.**Match**(*g*, ***kwargs*)

This class provides a high-level API for policy matching. It should not be instantiated directly. Instead, code should use one of the provided classmethods to construct a `Match` object. See the respective classmethods for details.

A `Match` object encapsulates a policy matching operation, i.e. a call to `PolicyClass.match_policies`. In order to retrieve the matching policies, one should use one of `policies()`, `action_values()` and `any()`. By default, these functions write the matched policies to the audit log. This behavior can be explicitly disabled.

Every classmethod expects a so-called "context object" as its first argument. The context object implements the following attributes:

- **`audit_object`: an `Audit` object which is used to write the used policies to the audit log.** In case False is passed for `write_to_audit_log`, the audit object may be None.
- `policy_object`: a `PolicyClass` object that is used to retrieve the matching policies.
- `client_ip`: the IP of the current client, as a string

- **`logged_in_user`: a dictionary with keys "username", "realm", "role" that describes the** currently logged-in (managing) user

In our case, this context object is usually the `flask.g` object.

classmethod action_only(*g*, *scope*, *action*)
Match active policies solely based on a scope and an action, which may also be None. The client IP is matched implicitly. :param g: context object :param scope: the policy scope. SCOPE.ADMIN cannot be passed, `admin` must be used instead. :param action: the policy action, or None :rtype: `Match`

action_values(*unique*, *allow_white_space_in_action=False*, *write_to_audit_log=True*)
Return a dictionary of action values extracted from the matching policies. The dictionary maps each action value to a list of policies which define this action value. :param unique: If True, return only the prioritized action value.

See `PolicyClass.get_action_values` for details.

Parameters

- **`allow_white_space_in_action`** – If True, allow whitespace in action values. See `PolicyClass.get_action_values` for details.
- **`write_to_audit_log`** – If True, augment the audit log with the names of all policies whose action values are returned

Return type dict

classmethod admin(*g*, *action*, *user_obj=None*)
Match admin policies with an action and, optionally, a realm. Assumes that the currently logged-in user is an admin, and throws an error otherwise. Policies will be matched against the admin's username and adminrealm, and optionally also the provided user_obj on which the admin is acting The client IP is matched implicitly. :param g: context object :param action: the policy action :param user_obj: the user against which policies should be matched. Can be None. :type user_obj: User or None :rtype: `Match`

classmethod admin_or_user(*g*, *action*, *user_obj*)
Depending on the role of the currently logged-in user, match either scope=ADMIN or scope=USER policies. If the currently logged-in user is an admin, match policies against the username, adminrealm and the given user_obj on which the admin is acting. If the currently logged-in user is a user, match policies against the username and the given realm. The client IP is matched implicitly. :param g: context object :param action: the policy action :param user_obj: the user_obj on which the administrator is acting :rtype: `Match`

allowed(*write_to_audit_log=True*)
Determine if the matched action is allowed in the scope 'admin' or 'user'. This is the case

- *either* if there are no active policies defined in the matched scope
- *or* the action is explicitly allowed by a policy in the matched scope

Example usage:: is_allowed = Match.user(g, scope=SCOPE.USER, action=ACTION.ENROLLPIN, user=user_object).allowed() # is_allowed is now true # * *either* if there is no active policy defined with scope=SCOPE.USER at all # * *or* if there is a policy matching the given scope, action, user and client IP.

Parameters **`write_to_audit_log`** – If True, write the list of matching policies to the audit log

Returns True or False

any (*write_to_audit_log=True*)

Return True if at least one policy matches. :param write_to_audit_log: If True, write the list of matching policies to the audit log :return: True or False

classmethod generic (*g*, *scope=None*, *realm=None*, *resolver=None*, *user=None*, *user_object=None*, *client=None*, *action=None*, *adminrealm=None*, *adminuser=None*, *time=None*, *active=True*, *sort_by_priority=True*)

Low-level legacy policy matching interface: Search for active policies and return them sorted by priority. All parameters that should be used for matching have to be passed explicitly. The client IP has to be passed explicitly. See `PolicyClass.match_policies` for details. :rtype: `Match`

policies (*write_to_audit_log=True*)

Return a list of policies. The list is sorted by priority, which means that prioritized policies appear first. :param write_to_audit_log: If True, write the list of matching policies to the audit log :return: a list of policy dictionaries :rtype: list

classmethod realm (*g*, *scope*, *action*, *realm*)

Match active policies with a scope, an action and a user realm. The client IP is matched implicitly. :param g: context object :param scope: the policy scope. SCOPE.ADMIN cannot be passed, `admin` must be used instead. :param action: the policy action :param realm: the realm to match :rtype: `Match`

classmethod user (*g*, *scope*, *action*, *user_object*)

Match active policies with a scope, an action and a user object (which may be None). The client IP is matched implicitly. :param g: context object :param scope: the policy scope. SCOPE.ADMIN cannot be passed, `admin` must be used instead. :param action: the policy action :param user_object: the user object to match. Might also be None, which means that the policy

> attributes `user`, `realm` and `resolver` are ignored.
>
> **Return type** `Match`

exception privacyidea.lib.policy.**MatchingError** (*description='server error!'*, *id=903*)

class privacyidea.lib.policy.**PolicyClass**

A policy object can be used to query the current set of policies. The policy object itself does not store any policies. Instead, every query uses `get_config_object` to retrieve the request-local config object which contains the current set of policies.

Hence, reloading the request-local config object also reloads the set of policies.

static check_for_conflicts (*policies*, *action*)

Given a (not necessarily sorted) list of policy dictionaries and an action name, check that there are no action value conflicts.

This raises a PolicyError if there are multiple policies with the highest priority which define different values for **action**.

Otherwise, the function just returns nothing.

Parameters

- **policies** – list of dictionaries
- **action** – string

static extract_action_values (*policies*, *action*, *unique=False*, *allow_white_space_in_action=False*)

Given an action, extract all values the given policies specify for that action. :param policies: a list of policy dictionaries :type policies: list :param action: a policy action :type action: action :param unique: if True, only consider the policy with the highest priority

> and check for policy conflicts (in this case, raise a PolicyError).

Parameters **allow_white_space_in_action** – Some policies like emailtext would allow entering text with whitespaces. These whitespaces must not be used to separate action values!

Returns a dictionary mapping action values to lists of matching policies.

filter_policies_by_conditions (*policies*, *user_object=None*, *request_headers=None*)
Given a list of policy dictionaries and a current user object (if any), return a list of all policies whose conditions match the given user object. Raises a PolicyError if a condition references an unknown section. :param policies: a list of policy dictionaries :param user_object: a User object, or None if there is no current user :param request_headers: The HTTP headers :type request_headers: Request object :return: generates a list of policy dictionaries

get_action_values (*action*, *scope='authorization'*, *realm=None*, *resolver=None*, *user=None*, *client=None*, *unique=False*, *allow_white_space_in_action=False*, *adminrealm=None*, *adminuser=None*, *user_object=None*, *audit_data=None*)

Get the defined action values for a certain action like scope: authorization action: tokentype

would return a dictionary of {tokentype: policyname}

scope: authorization action: serial

would return a dictionary of {serial: policyname}

All parameters not described below are covered in the documentation of `match_policies`.

Parameters

- **unique** – if set, the function will only consider the policy with the highest priority and check for policy conflicts.
- **allow_white_space_in_action** (*bool*) – Some policies like emailtext would allow entering text with whitespaces. These whitespaces must not be used to separate action values!
- **audit_data** – This is a dictionary, that can take audit_data in the g object. If set, this dictionary will be filled with the list of triggered policynames in the key “policies”. This can be useful for policies like ACTION.OTPPIN - where it is clear, that the found policy will be used. It could make less sense with an action like ACTION.LASTAUTH - where the value of the action needs to be evaluated in a more special case.

Return type dict

list_policies (*name=None*, *scope=None*, *realm=None*, *active=None*, *resolver=None*, *user=None*, *client=None*, *action=None*, *adminrealm=None*, *adminuser=None*, *sort_by_priority=True*)
Return the policies, filtered by the given values.

The following rule holds for all filter arguments:

If `None` is passed as a value, policies are not filtered according to the argument at all. As an example, if `realm=None` is passed, policies are matched regardless of their `realm` attribute. If any value is passed (even the empty string), policies are filtered according to the given value. As an example, if `realm=''` is passed, only policies that have a matching (or empty) realm attribute are returned.

The only exception is the `client` parameter, which does not accept the empty string, and throws a ParameterError if the empty string is passed.

Parameters

- **name** – The name of the policy
- **scope** – The scope of the policy

- **realm** – The realm in the policy
- **active** – One of None, True, False: All policies, only active or only inactive policies
- **resolver** – Only policies with this resolver
- **user** (*basestring*) – Only policies with this user
- **client** –
- **action** – Only policies, that contain this very action.
- **adminrealm** – This is the realm of the admin. This is only evaluated in the scope admin.
- **adminuser** – This is the username of the admin. This in only evaluated in the scope admin.
- **sort_by_priority** – If true, sort the resulting list by priority, ascending

by their policy numbers. :type sort_by_priority: bool :return: list of policies :rtype: list of dicts

match_policies(*name=None*, *scope=None*, *realm=None*, *active=None*, *resolver=None*, *user=None*, *user_object=None*, *client=None*, *action=None*, *adminrealm=None*, *adminuser=None*, *time=None*, *sort_by_priority=True*, *audit_data=None*, *request_headers=None*)

Return all policies matching the given context. Optionally, write the matching policies to the audit log.

In order to retrieve policies matching the current user, callers can *either* pass a user(name), resolver and realm, *or* pass a user object from which login name, resolver and realm will be read. In case of conflicting parameters, a ParameterError will be raised.

This function takes all parameters taken by `list_policies`, plus some additional parameters.

Parameters

- **name** – see `list_policies`
- **scope** – see `list_policies`
- **realm** – see `list_policies`
- **active** – see `list_policies`
- **resolver** – see `list_policies`
- **user** – see `list_policies`
- **client** – see `list_policies`
- **action** – see `list_policies`
- **adminrealm** – see `list_policies`
- **adminuser** – see `list_policies`
- **sort_by_priority** –
- **user_object** (`User or None`) – the currently active user, or None
- **time** (`datetime or None`) – return only policies that are valid at the specified time. Defaults to the current time.
- **audit_data** – A dictionary with audit data collected during a request. This

method will add found policies to the dictionary. :type audit_data: dict or None :param request_headers: A dict with HTTP headers :return: a list of policy dictionaries

property policies

Shorthand to retrieve the set of policies of the request-local config object

`ui_get_enroll_tokentypes`(*client*, *logged_in_user*)

Return a dictionary of the allowed tokentypes for the logged in user. This used for the token enrollment UI.

It looks like this:

{"hotp": "HOTP: event based One Time Passwords", "totp": "TOTP: time based One Time Passwords", "spass": "SPass: Simple Pass token. Static passwords", "motp": "mOTP: classical mobile One Time Passwords", "sshkey": "SSH Public Key: The public SSH key", "yubikey": "Yubikey AES mode: One Time Passwords with Yubikey", "remote": "Remote Token: Forward authentication request to another server", "yubico": "Yubikey Cloud mode: Forward authentication request to YubiCloud", "radius": "RADIUS: Forward authentication request to a RADIUS server", "email": "EMail: Send a One Time Passwort to the users email address", "sms": "SMS: Send a One Time Password to the users mobile phone", "certificate": "Certificate: Enroll an x509 Certificate Token."}

Parameters

- **`client`** (*`basestring`*) – Client IP address
- **`logged_in_user`** (*`dict`*) – The Dict of the logged in user

Returns list of token types, the user may enroll

`ui_get_main_menus`(*logged_in_user*, *client=None*)

Get the list of allowed main menus derived from the policies for the given user - admin or normal user. It fetches all policies for this user and compiles a list of allowed menus to display or hide in the UI.

Parameters

- **`logged_in_user`** – The logged in user, a dictionary with keys "username", "realm" and "role".
- **`client`** – The IP address of the client

Returns A list of MENUs to be displayed

`ui_get_rights`(*scope*, *realm*, *username*, *client=None*)

Get the rights derived from the policies for the given realm and user. Works for admins and normal users. It fetches all policies for this user and compiles a maximum list of allowed rights, that can be used to hide certain UI elements.

Parameters

- **`scope`** – Can be SCOPE.ADMIN or SCOPE.USER
- **`realm`** – Is either user users realm or the adminrealm
- **`username`** – The loginname of the user
- **`client`** – The HTTP client IP

Returns A list of actions

class `privacyidea.lib.policy.`**`REMOTE_USER`**

The list of possible values for the remote_user policy.

`ACTIVE = 'allowed'`

`DISABLE = 'disable'`

class `privacyidea.lib.policy.`**`SCOPE`**

This is the list of the allowed scopes that can be used in policy definitions.

ADMIN = 'admin'

AUDIT = 'audit'

AUTH = 'authentication'

AUTHZ = 'authorization'

ENROLL = 'enrollment'

REGISTER = 'register'

USER = 'user'

WEBUI = 'webui'

class privacyidea.lib.policy.**TIMEOUT_ACTION**

This is a list of actions values for idle users

LOCKSCREEN = 'lockscreen'

LOGOUT = 'logout'

privacyidea.lib.policy.**delete_all_policies**()

privacyidea.lib.policy.**delete_policy**(*name*)

Function to delete one named policy. Raise ResourceNotFoundError if there is no such policy.

Parameters **name** – the name of the policy to be deleted

Returns the ID of the deleted policy

Return type int

privacyidea.lib.policy.**enable_policy**(*name*, *enable=True*)

Enable or disable the policy with the given name :param name: :return: ID of the policy

privacyidea.lib.policy.**export_policies**(*policies*)

This function takes a policy list and creates an export file from it

Parameters **policies** (*list of policy dictionaries*) – a policy definition

Returns the contents of the file

Return type string

privacyidea.lib.policy.**get_action_values_from_options**(*scope*, *action*, *options*)

This function is used in the library level to fetch policy action values from a given option dictionary.

The matched policies are *not* written to the audit log.

Returns A scalar, string or None

privacyidea.lib.policy.**get_policy_condition_comparators**()

Returns a dictionary mapping comparators to dictionaries with the following keys: * "description", a human-readable description of the comparator

privacyidea.lib.policy.**get_policy_condition_sections**()

Returns a dictionary mapping condition sections to dictionaries with the following keys: * "description", a human-readable description of the section

privacyidea.lib.policy.**get_static_policy_definitions**(*scope=None*)

These are the static hard coded policy definitions. They can be enhanced by token based policy definitions, that can be found in lib.token.get_dynamic_policy_definitions.

Parameters **scope** (*basestring*) – Optional the scope of the policies

Returns allowed scopes with allowed actions, the type of action and a

description. :rtype: dict

`privacyidea.lib.policy.`**`import_policies`**(*file_contents*)

This function imports policies from a file. The file has a config_object format, i.e. the text file has a header

[<policy_name>] key = value

and key value pairs.

Parameters **`file_contents`** (*`basestring`*) – The contents of the file

Returns number of imported policies

Return type int

`privacyidea.lib.policy.`**`set_policy`**(*name=None*, *scope=None*, *action=None*, *realm=None*, *resolver=None*, *user=None*, *time=None*, *client=None*, *active=True*, *adminrealm=None*, *adminuser=None*, *priority=None*, *check_all_resolvers=False*, *conditions=None*)

Function to set a policy. If the policy with this name already exists, it updates the policy. It expects a dict of with the following keys: :param name: The name of the policy :param scope: The scope of the policy. Something like "admin" or "authentication" :param action: A scope specific action or a comma separated list of actions :type active: basestring :param realm: A realm, for which this policy is valid :param resolver: A resolver, for which this policy is valid :param user: A username or a list of usernames :param time: N/A if type() :param client: A client IP with optionally a subnet like 172.16.0.0/16 :param active: If the policy is active or not :type active: bool :param adminrealm: The name of the realm of administrators :type adminrealm: str :param adminuser: A comma separated list of administrators :type adminuser: str :param priority: the priority of the policy (smaller values having higher priority) :type priority: int :param check_all_resolvers: If all the resolvers of a user should be

checked with this policy

Parameters **`conditions`** – A list of 5-tuples (section, key, comparator, value, active) of policy conditions

Returns The database ID od the the policy

Return type int

Job Queue

The following queue classes are known to privacyIDEA

Huey Queue Class

`class` `privacyidea.lib.queues.huey_queue.`**`HueyQueue`**(*options*)

`enqueue`(*name*, *args*, *kwargs*)

Schedule an invocation of a job on the external job queue.

Parameters

- **`name`** – Unique job name
- **`args`** – Tuple of positional arguments
- **`kwargs`** – Dictionary of keyword arguments

Returns None

property huey

property jobs

register_job(*name*, *func*)

Add a job to the internal registry.

Parameters

- **name** – Unique job name
- **func** – Function that should be executed by an external job queue

privacyidea.lib.queue.**JOB_COLLECTOR = <privacyidea.lib.queue.JobCollector object>**

A singleton is fine here, because it is only used at import time and once when a new app is created. Afterwards, the object is unused.

class privacyidea.lib.queue.**JobCollector**

For most third-party job queue modules, the jobs are discovered by tracking all functions decorated with a `@job` decorator. However, in order to invoke the decorator, one usually needs to provide the queue configuration (e.g. the redis server) already. In privacyIDEA, we cannot do that, because the app config is not known yet – it will be known when `create_app` is called! Thus, we cannot directly use the @job decorator, but need a job collector that collects jobs in privacyIDEA code and registers them with the job queue module when `create_app` has been called.

property jobs

register_app(*app*)

Create an instance of a `BaseQueue` subclass according to the app config's `PI_JOB_QUEUE_CLASS` option and store it in the `job_queue` config. Register all collected jobs with this application. This instance is shared between threads! This function should only be called once per process.

Parameters **app** – privacyIDEA app

register_job(*name*, *func*, *args*, *kwargs*)

Register a job with the collector.

Parameters

- **name** – unique name of the job
- **func** – function of the job
- **args** – arguments passed to the job queue's `register_job` method
- **kwargs** – keyword arguments passed to the job queue's `register_job` method

privacyidea.lib.queue.**get_job_queue**()

Get the job queue registered with the current app. If no job queue is configured, raise a ServerError.

privacyidea.lib.queue.**has_job_queue**()

Return a boolean describing whether the current app has an app queue configured.

privacyidea.lib.queue.**job**(*name*, **args*, ***kwargs*)

Decorator to mark a job to be collected by the job collector. All arguments are passed to `register_job`.

privacyidea.lib.queue.**register_app**(*app*)

Register the app `app` with the global job collector, if a PI_JOB_QUEUE_CLASS is non-empty. Do nothing otherwise.

privacyidea.lib.queue.**wrap_job**(*name*, *result*)

Wrap a job and return a function that can be used like the original function. The returned function will always return `result`. This assumes that a queue is configured! Otherwise, calling the resulting function will fail with a ServerError.

Returns a function

Base class

class `privacyidea.lib.queues.base.`**`BaseQueue`**(*options*)

A queue object represents an external job queue and is configured with a dictionary of options. It allows to register jobs, which are Python functions that may be executed outside of the request lifecycle. Every job is identified by a unique job name. It then allows to delegate (or "enqueue") an invocation of a job (which is identified by its job name) to the external job queue. Currently, the queue only supports fire-and-forget jobs, i.e. jobs without any return value.

`enqueue`(*name*, *args*, *kwargs*)

Schedule an invocation of a job on the external job queue.

Parameters

- **`name`** – Unique job name
- **`args`** – Tuple of positional arguments
- **`kwargs`** – Dictionary of keyword arguments

Returns None

`register_job`(*name*, *func*)

Add a job to the internal registry.

Parameters

- **`name`** – Unique job name
- **`func`** – Function that should be executed by an external job queue

API Policies

Pre Policies

These are the policy decorators as PRE conditions for the API calls. I.e. these conditions are executed before the wrapped API call. This module uses the policy base functions from privacyidea.lib.policy but also components from flask like g.

Wrapping the functions in a decorator class enables easy modular testing.

The functions of this module are tested in tests/test_api_lib_policy.py

`privacyidea.api.lib.prepolicy.`**`allowed_audit_realm`**(*request=None*, *action=None*)

This decorator function takes the request and adds additional parameters to the request according to the policy for the SCOPE.ADMIN or ACTION.AUDIT :param request: :param action: :return: True

`privacyidea.api.lib.prepolicy.`**`api_key_required`**(*request=None*, *action=None*)

This is a decorator for check_user_pass and check_serial_pass. It checks, if a policy scope=auth, action=apikeyrequired is set. If so, the validate request will only performed, if a JWT token is passed with role=validate.

`privacyidea.api.lib.prepolicy.`**`auditlog_age`**(*request=None*, *action=None*)

This pre condition checks for the policy auditlog_age and set the "timelimit" parameter of the audit search API.

Check ACTION.AUDIT_AGE

The decorator can wrap GET /audit/

Parameters

- **request** (*Request Object*) – The request that is intercepted during the API call
- **action** (*basestring*) – An optional Action

Returns Always true. Modified the parameter request

privacyidea.api.lib.prepolicy.**check_admin_tokenlist**(*request=None*, *action=None*)

Depending on the policy scope=admin, action=tokenlist, the allowed_realms parameter is set to define, the token of which realms and administrator is allowed to see.

Sets the allowed_realms None: means the admin has no restrictions []: the admin can not see any realms ["realm1", "realm2"...]: the admin can see these realms

Parameters **request** –

Returns

privacyidea.api.lib.prepolicy.**check_anonymous_user**(*request=None*, *action=None*)

This decorator function takes the request and verifies the given action for the SCOPE USER without an authenticated user but the user from the parameters.

This is used with password_reset

Parameters

- **request** –
- **action** –

Returns True otherwise raises an Exception

privacyidea.api.lib.prepolicy.**check_base_action**(*request=None*, *action=None*, *anonymous=False*)

This decorator function takes the request and verifies the given action for the SCOPE ADMIN or USER. :param request: :param action: :param anonymous: If set to True, the user data is taken from the request

parameters.

Returns True otherwise raises an Exception

privacyidea.api.lib.prepolicy.**check_external**(*request=None*, *action='init'*)

This decorator is a hook to an external check function, that is called before the token/init or token/assign API.

Parameters

- **request** (*flask Request object*) – The REST request
- **action** (*basestring*) – This is either "init" or "assign"

Returns either True or an Exception is raised

privacyidea.api.lib.prepolicy.**check_max_token_realm**(*request=None*, *action=None*)

Pre Policy This checks the maximum token per realm. Check ACTION.MAXTOKENREALM

This decorator can wrap: /token/init (with a realm and user) /token/assign /token/tokenrealms

Parameters

- **req** (*Request Object*) – The request that is intercepted during the API call
- **action** (*basestring*) – An optional Action

Returns True otherwise raises an Exception

`privacyidea.api.lib.prepolicy.`**`check_max_token_user`**`(`*request=None*, *action=None*`)`

Pre Policy This checks the maximum token per user policy. Check ACTION.MAXTOKENUSER Check ACTION.MAXACTIVETOKENUSER

This decorator can wrap: /token/init (with a realm and user) /token/assign

Parameters

- **`req`** –
- **`action`** –

Returns True otherwise raises an Exception

`privacyidea.api.lib.prepolicy.`**`check_otp_pin`**`(`*request=None*, *action=None*`)`

This policy function checks if the OTP PIN that is about to be set follows the OTP PIN policies ACTION.OTPPINMAXLEN, ACTION.OTPPINMINLEN and ACTION.OTPPINCONTENTS and token-type-specific PIN policy actions in the SCOPE.USER or SCOPE.ADMIN. It is used to decorate the API functions.

The pin is investigated in the params as "otppin" or "pin"

In case the given OTP PIN does not match the requirements an exception is raised.

`privacyidea.api.lib.prepolicy.`**`check_token_init`**`(`*request=None*, *action=None*`)`

This decorator function takes the request and verifies if the requested tokentype is allowed to be enrolled in the SCOPE ADMIN or the SCOPE USER. :param request: :param action: :return: True or an Exception is raised

`privacyidea.api.lib.prepolicy.`**`check_token_upload`**`(`*request=None*, *action=None*`)`

This decorator function takes the request and verifies the given action for scope ADMIN :param req: :param filename: :return:

`privacyidea.api.lib.prepolicy.`**`encrypt_pin`**`(`*request=None*, *action=None*`)`

This policy function is to be used as a decorator for several API functions. E.g. token/assign, token/setpin, token/init If the policy is set to define the PIN to be encrypted, the request.all_data is modified like this: encryptpin = True

It uses the policy SCOPE.ENROLL, ACTION.ENCRYPTPIN

`privacyidea.api.lib.prepolicy.`**`enroll_pin`**`(`*request=None*, *action=None*`)`

This policy function is used as decorator for init token. It checks, if the user or the admin is allowed to set a token PIN during enrollment. If not, it deleted the PIN from the request.

`privacyidea.api.lib.prepolicy.`**`indexedsecret_force_attribute`**`(`*request*, *action*`)`

This is a token specific wrapper for indexedsecret token for the endpoint /token/init The otpkey is overwritten with the value from the user attribute specified in policy scope=SCOPE.USER and SCOPE.ADMIN, action=PIIXACTION.FORCE_ATTRIBUTE. :param request: :param action: :return:

`privacyidea.api.lib.prepolicy.`**`init_random_pin`**`(`*request=None*, *action=None*`)`

This policy function is to be used as a decorator in the API init function If the policy is set accordingly it adds a random PIN to the request.all_data like.

It uses the policy SCOPE.ENROLL, ACTION.OTPPINRANDOM to set a random OTP PIN during Token enrollment

`privacyidea.api.lib.prepolicy.`**`init_token_defaults`**`(`*request=None*, *action=None*`)`

This policy function is used as a decorator for the API init function. Depending on policy settings it can add token specific default values like totp_hashlib, hotp_hashlib, totp_otplen…

privacyidea.api.lib.prepolicy.**init_tokenlabel**(*request=None*, *action=None*)

This policy function is to be used as a decorator in the API init function. It adds the tokenlabel definition to the params like this: params : { “tokenlabel”: “<u>@<r>” }

In addition it adds the tokenissuer to the params like this: params : { “tokenissuer”: “privacyIDEA instance” }

It also checks if the force_app_pin policy is set and adds the corresponding value to params.

It uses the policy SCOPE.ENROLL, ACTION.TOKENLABEL and ACTION.TOKENISSUER to set the tokenlabel and tokenissuer of Smartphone tokens during enrollment and this fill the details of the response.

privacyidea.api.lib.prepolicy.**is_remote_user_allowed**(*req*)

Checks if the REMOTE_USER server variable is allowed to be used.

Note: This is not used as a decorator!

Parameters **req** – The flask request, containing the remote user and the client IP

Returns a bool value

privacyidea.api.lib.prepolicy.**mangle**(*request=None*, *action=None*)

This pre condition checks if either of the parameters pass, user or realm in a validate/check request should be rewritten based on an authentication policy with action “mangle”. See *mangle* for an example.

Check ACTION.MANGLE

This decorator should wrap /validate/check

Parameters

- **request** (*Request Object*) – The request that is intercepted during the API call
- **action** (*basestring*) – An optional Action

Returns Always true. Modified the parameter request

privacyidea.api.lib.prepolicy.**mock_fail**(*req*, *action*)

This is a mock function as an example for check_external. This function creates a problem situation and the token/init or token/assign will show this exception accordingly.

privacyidea.api.lib.prepolicy.**mock_success**(*req*, *action*)

This is a mock function as an example for check_external. This function returns success and the API call will go on unmodified.

privacyidea.api.lib.prepolicy.**papertoken_count**(*request=None*, *action=None*)

This is a token specific wrapper for paper token for the endpoint /token/init. According to the policy scope=SCOPE.ENROLL, action=PAPERACTION.PAPER_COUNT it sets the parameter papertoken_count to enroll a paper token with such many OTP values.

Parameters

- **request** –
- **action** –

Returns

class privacyidea.api.lib.prepolicy.**prepolicy**(*function*, *request*, *action=None*)

This is the decorator wrapper to call a specific function before an API call. The prepolicy decorator is to be used in the API calls. A prepolicy decorator then will modify the request data or raise an exception privacyidea.api.lib.prepolicy.**pushtoken_add_config**(*request*, *action*)

This is a token specific wrapper for push token for the endpoint /token/init According to the policy scope=SCOPE.ENROLL, action=SSL_VERIFY or action=FIREBASE_CONFIG the parameters are added to the enrollment step. :param request: :param action: :return:

privacyidea.api.lib.prepolicy.**pushtoken_disable_wait**(*request*, *action*)

This is used for the /auth endpoint and sets the PUSH_ACTION.WAIT parameter to False.

Parameters

- **request** –
- **action** –

Returns

privacyidea.api.lib.prepolicy.**pushtoken_wait**(*request*, *action*)

This is a auth specific wrapper to decorate /validate/check According to the policy scope=SCOPE.AUTH, action=push_wait

Parameters

- **request** –
- **action** –

Returns

privacyidea.api.lib.prepolicy.**realmadmin**(*request=None*, *action=None*)

This decorator adds the first REALM to the parameters if the administrator, calling this API is a realm admin. This way, if the admin calls e.g. GET /user without realm parameter, he will not see all users, but only users in one of his realms.

TODO: If a realm admin is allowed to see more than one realm, this is not handled at the moment. We need to change the underlying library functions!

Parameters

- **request** – The HTTP reqeust
- **action** – The action like ACTION.USERLIST

privacyidea.api.lib.prepolicy.**required_email**(*request=None*, *action=None*)

This precondition checks if the "email" parameter matches the regular expression in the policy scope=register, action=requiredemail. See *requiredemail*.

Check ACTION.REQUIREDEMAIL

This decorator should wrap POST /register

Parameters

- **request** – The Request Object
- **action** – An optional Action

Returns Modifies the request parameters or raises an Exception

privacyidea.api.lib.prepolicy.**save_client_application_type**(*request*, *action*)

This decorator is used to write the client IP and the HTTP user agent (clienttype) to the database.

In fact this is not a **policy** decorator, as it checks no policy. In fact, we could however one day define this as a policy, too. :param req: :return:

`privacyidea.api.lib.prepolicy.`**`set_random_pin`**(*request=None*, *action=None*)

This policy function is to be used as a decorator in the API setrandompin function If the policy is set accordingly it adds a random PIN to the request.all_data like.

It uses the policy ACTION.OTPPINSETRANDOM in SCOPE.ADMIN or SCOPE.USER to set a random OTP PIN

`privacyidea.api.lib.prepolicy.`**`set_realm`**(*request=None*, *action=None*)

Pre Policy This pre condition gets the current realm and verifies if the realm should be rewritten due to the policy definition. I takes the realm from the request and - if a policy matches - replaces this realm with the realm defined in the policy

Check ACTION.SETREALM

This decorator should wrap /validate/check

Parameters

- **`request`** (*`Request Object`*) – The request that is intercepted during the API call
- **`action`** (*`basestring`*) – An optional Action

Returns Always true. Modified the parameter request

`privacyidea.api.lib.prepolicy.`**`sms_identifiers`**(*request=None*, *action=None*)

This is a token specific wrapper for sms tokens to be used with the endpoint /token/init. According to the policy scope=SCOPE.ADMIN or scope=SCOPE.USER action=SMSACTION.GATEWAYS the sms.identifier is only allowed to be set to the listed gateways.

Parameters

- **`request`** –
- **`action`** –

Returns

`privacyidea.api.lib.prepolicy.`**`tantoken_count`**(*request=None*, *action=None*)

This is a token specific wrapper for tan token for the endpoint /token/init. According to the policy scope=SCOPE.ENROLL, action=TANACTION.TANTOKEN_COUNT it sets the parameter tantoken_count to enroll a tan token with such many OTP values.

Parameters

- **`request`** –
- **`action`** –

Returns

`privacyidea.api.lib.prepolicy.`**`twostep_enrollment_activation`**(*request=None*, *action=None*)

This policy function enables the two-step enrollment process according to the configured policies. It is used to decorate the `/token/init` endpoint.

If a `<type>_2step` policy matches, the `2stepinit` parameter is handled according to the policy. If no policy matches, the `2stepinit` parameter is removed from the request data.

`privacyidea.api.lib.prepolicy.`**`twostep_enrollment_parameters`**(*request=None*, *action=None*)

If the `2stepinit` parameter is set to true, this policy function reads additional configuration from policies and adds it to `request.all_data`, that is:

- `{type}_2step_serversize` is written to `2step_serversize`

- `{type}_2step_clientsize` is written to ``2step_clientsize`
- `{type}_2step_difficulty` is written to `2step_difficulty`

If no policy matches, the value passed by the user is kept.

This policy function is used to decorate the `/token/init` endpoint.

`privacyidea.api.lib.prepolicy.`**`u2ftoken_allowed`**`(`*request*, *action*`)`

This is a token specific wrapper for u2f token for the endpoint /token/init. According to the policy scope=SCOPE.ENROLL, action=U2FACTION.REQ it checks, if the assertion certificate is an allowed U2F token type.

If the token, which is enrolled contains a non allowed attestation certificate, we bail out.

Parameters

- **request** –
- **action** –

Returns

`privacyidea.api.lib.prepolicy.`**`u2ftoken_verify_cert`**`(`*request*, *action*`)`

This is a token specific wrapper for u2f token for the endpoint /token/init According to the policy scope=SCOPE.ENROLL, action=U2FACTION.NO_VERIFY_CERT it can add a parameter to the enrollment parameters to not verify the attestation certificate. The default is to verify the cert. :param request: :param action: :return:

`privacyidea.api.lib.prepolicy.`**`webauthntoken_allowed`**`(`*request*, *action*`)`

This is a token specific wrapper for WebAuthn token for the endpoint /token/init.

According to the policy scope=SCOPE.ENROLL, action=WEBAUTHNACTION.REQ it checks, if the assertion certificate is for an allowed WebAuthn token type. According to the policy scope=SCOPE.ENROLL, action=WEBAUTHNACTION.AUTHENTICATOR_SELECTION_LIST it checks, whether the AAGUID is whitelisted. Note: If self-attestation is allowed, it is – of course – possible to bypass the check for WEBAUTHNACTION.REQ

If the token, which is being enrolled does not contain an allowed attestation certificate, or does not have an allowed AAGUID, we bail out.

A very similar check (same policy actions, different policy scope) is performed during authorization, however due to architectural limitations, this lives within the token implementation itself.

Parameters

- **request** –
- **action** –

Returns

Return type

`privacyidea.api.lib.prepolicy.`**`webauthntoken_auth`**`(`*request*, *action*`)`

This is a WebAuthn token specific wrapper for the endpoints /validate/triggerchallenge, /validate/check, and /auth.

This will enrich the challenge creation request for WebAuthn tokens with the necessary configuration information from policy actions with scope=SCOPE.AUTH. The request will be augmented with the allowed transports and the text to display to the user when asking to confirm the challenge on the token, as specified by the actions WEBAUTHNACTION.ALLOWED_TRANSPORTS, and ACTION.CHALLENGETEXT, respectively.

Both of these policies are optional, and have sensible defaults.

Parameters

- **request** –
- **action** –

Returns

Return type

privacyidea.api.lib.prepolicy.**webauthntoken_authz**(*request*, *action*)

This is a WebAuthn token specific wrapper for the /auth, and /validate/check endpoints.

This will enrich the authorization request for WebAuthn tokens with the necessary configuration information from policy actions with scope=SCOPE.AUTHZ. This is currently the authorization pendant to webauthntoken_allowed(), but maybe expanded to cover other authorization policies in the future, should any be added. The request will as of now simply be augmented with the policies the attestation certificate is to be matched against.

Parameters

- **request** –
- **action** –

Returns

:rtype

privacyidea.api.lib.prepolicy.**webauthntoken_enroll**(*request*, *action*)

This is a token specific wrapper for the WebAuthn token for the endpoint /token/init.

This will enrich the initialization request for WebAuthn tokens with the necessary configuration information from policy actions with scope=SCOPE.ENROLL. The request will be augmented with a name and id for the relying party, as specified by the with actions WEBAUTHNACTION.RELYING_PARTY_NAME and WEBAUTHNACTION.RELYING_PARTY_ID, respectively, authenticator attachment preference, public key credential algorithm preferences, authenticator attestation requirement level, authenticator attestation requirement form, allowed AAGUIDs, and the text to display to the user when asking to confirm the challenge on the token, as specified by the actions WEBAUTHNACTION.AUTHENTICATOR_ATTACHMENT, WEBAUTHNACTION.PUBLIC_KEY_CREDENTIAL_ALGORITHM_PREFERENCE, WEBAUTHNACTION.AUTHENTICATOR_ATTESTATION_LEVEL, WEBAUTHNACTION.AUTHENTICATOR_ATTESTATION_FORM, WEBAUTHNACTION.AUTHENTICATOR_SELECTION_LIST, and ACTION.CHALLENGETEXT, respectively.

Setting WEBAUTHNACTION.RELYING_PARTY_NAME and WEBAUTHNACTION.RELYING_PARTY_ID is mandatory, and if either of these is not set, we bail out.

Parameters

- **request** –
- **action** –

Returns

Return type

privacyidea.api.lib.prepolicy.**webauthntoken_request**(*request*, *action*)

This is a WebAuthn token specific wrapper for all endpoints using WebAuthn tokens.

This wraps the endpoints /token/init, /validate/triggerchallenge, /auth, and /validate/check. It will add WebAuthn configuration information to the requests, wherever a piece of information is needed for several different requests and thus cannot be provided by one of the more specific wrappers without adding unnecessary redundancy.

Depending on the type of request, the request will be augmented with some (or all) of the authenticator timeout, user verification requirement and list of allowed AAGUIDs for the current scope, as specified by the policies with the determined scope and the actions WEBAUTHNACTION.TIMEOUT, WEBAUTHNACTION.USER_VERIFICATION_REQUIREMENT, and WEBAUTHNACTION.AUTHENTICATOR_SELECTION_LIST, respectively.

The value of the ORIGIN http header will also be added to the request for the ENROLL and AUTHZ scopes. This is to make the unit tests not require mocking.

Parameters

- **request** –
- **action** –

Returns

Return type

Post Policies

These are the policy decorators as POST conditions for the API calls. I.e. these conditions are executed after the wrapped API call. This module uses the policy base functions from privacyidea.lib.policy but also components from flask like g.

Wrapping the functions in a decorator class enables easy modular testing.

The functions of this module are tested in tests/test_api_lib_policy.py

privacyidea.api.lib.postpolicy.**add_user_detail_to_response**(*request*, *response*)

This policy decorated is used in the AUTHZ scope. If the boolean value add_user_in_response is set, the details will contain a dictionary "user" with all user details.

Parameters

- **request** –
- **response** –

Returns

privacyidea.api.lib.postpolicy.**autoassign**(*request*, *response*)

This decorator decorates the function /validate/check. Depending on ACTION.AUTOASSIGN it checks if the user has no token and if the given OTP-value matches a token in the users realm, that is not yet assigned to any user.

If a token can be found, it assigns the token to the user also taking into account ACTION.MAXTOKENUSER and ACTION.MAXTOKENREALM. :return:

privacyidea.api.lib.postpolicy.**check_serial**(*request*, *response*)

This policy function is to be used in a decorator of an API function. It checks, if the token, that was used in the API call has a serial number that is allowed to be used.

If not, a PolicyException is raised.

Parameters **response** (*Response object*) – The response of the decorated function

Returns A new (maybe modified) response

privacyidea.api.lib.postpolicy.**check_tokeninfo**(*request*, *response*)

This policy function is used as a decorator for the validate API. It checks after a successful authentication if the token has a matching tokeninfo field. If it does not match, authorization is denied. Then a PolicyException is raised.

Parameters **response** (*Response object*) – The response of the decorated function

Returns A new modified response

privacyidea.api.lib.postpolicy.**check_tokentype**(*request*, *response*)

This policy function is to be used in a decorator of an API function. It checks, if the token, that was used in the API call is of a type that is allowed to be used.

If not, a PolicyException is raised.

Parameters **response** (*Response object*) – The response of the decorated function

Returns A new (maybe modified) response

privacyidea.api.lib.postpolicy.**construct_radius_response**(*request*, *response*)

This decorator implements the /validate/radiuscheck endpoint. In case this URL was requested, a successful authentication results in an empty response with a HTTP 204 status code. An unsuccessful authentication results in an empty response with a HTTP 400 status code. :return:

privacyidea.api.lib.postpolicy.**get_webui_settings**(*request*, *response*)

This decorator is used in the /auth API to add configuration information like the logout_time or the policy_template_url to the response. :param request: flask request object :param response: flask response object :return: the response

privacyidea.api.lib.postpolicy.**mangle_challenge_response**(*request*, *response*)

This policy decorator is used in the AUTH scope to decorate the /validate/check endpoint. It can modify the contents of the response “detail”->”message” to allow a better readability for a challenge response text.

Parameters

- **request** –
- **response** –

Returns

privacyidea.api.lib.postpolicy.**no_detail_on_fail**(*request*, *response*)

This policy function is used with the AUTHZ scope. If the boolean value no_detail_on_fail is set, the details will be stripped if the authentication request failed.

Parameters

- **request** –
- **response** –

Returns

privacyidea.api.lib.postpolicy.**no_detail_on_success**(*request*, *response*)

This policy function is used with the AUTHZ scope. If the boolean value no_detail_on_success is set, the details will be stripped if the authentication request was successful.

Parameters

- **request** –
- **response** –

Returns

privacyidea.api.lib.postpolicy.**offline_info**(*request*, *response*)

This decorator is used with the function /validate/check. It is not triggered by an ordinary policy but by a MachineToken definition. If for the given Client and Token an offline application is defined, the response is enhanced with the offline information - the hashes of the OTP.

class privacyidea.api.lib.postpolicy.**postpolicy**(*function*, *request=None*)

Decorator that allows one to call a specific function after the decorated function. The postpolicy decorator is to be used in the API calls.

class privacyidea.api.lib.postpolicy.**postrequest**(*function*, *request=None*)

Decorator that is supposed to be used with after_request.

privacyidea.api.lib.postpolicy.**save_pin_change**(*request*, *response*, *serial=None*)

This policy function checks if the next_pin_change date should be stored in the tokeninfo table.

1. Check scope:enrollment and ACTION.CHANGE_PIN_FIRST_USE. This action is used, when the administrator enrolls a token or sets a PIN
2. Check scope:enrollment and ACTION.CHANGE_PIN_EVERY is used, if the user changes the PIN.

This function decorates /token/init and /token/setpin. The parameter "pin" and "otppin" is investigated.

Parameters

- **request** –
- **action** –

Returns

privacyidea.api.lib.postpolicy.**sign_response**(*request*, *response*)

This decorator is used to sign the response. It adds the nonce from the request, if it exist and adds the nonce and the signature to the response.

Note: This only works for JSON responses. So if we fail to decode the JSON, we just pass on.

The usual way to use it is, to wrap the after_request, so that we can also sign errors.

@postrequest(sign_response, request=request) def after_request(response):

Parameters

- **request** – The Request object
- **response** – The Response object

Policy Decorators

These are the policy decorator functions for internal (lib) policy decorators. policy decorators for the API (pre/post) are defined in api/lib/policy

The functions of this module are tested in tests/test_lib_policy_decorator.py

privacyidea.lib.policydecorators.**auth_cache**(*wrapped_function*, *user_object*, *passw*, *options=None*)

Decorate lib.token:check_user_pass. Verify, if the authentication can be found in the auth_cache.

Parameters

- **wrapped_function** – usually "check_user_pass"
- **user_object** – User who tries to authenticate
- **passw** – The PIN and OTP
- **options** – Dict containing values for "g" and "clientip".

Returns Tuple of True/False and reply-dictionary `privacyidea.lib.policydecorators.`**`auth_lastauth`**(*wrapped_function*, *user_or_serial*, *passw*, *options=None*)

This decorator checks the policy settings of ACTION.LASTAUTH If the last authentication stored in tokeninfo last_auth_success of a token is exceeded, the authentication is denied.

The wrapped function is usually token.check_user_pass, which takes the arguments (user, passw, options={}) OR token.check_serial_pass with the arguments (user, passw, options={})

Parameters

- **`wrapped_function`** – either check_user_pass or check_serial_pass
- **`user_or_serial`** – either the User user_or_serial or a serial
- **`passw`** –
- **`options`** – Dict containing values for "g" and "clientip"

Returns Tuple of True/False and reply-dictionary

`privacyidea.lib.policydecorators.`**`auth_otppin`**(*wrapped_function*, **args*, ***kwds*)

Decorator to decorate the tokenclass.check_pin function. Depending on the ACTION.OTPPIN it * either simply accepts an empty pin * checks the pin against the userstore * or passes the request to the wrapped_function

Parameters **`wrapped_function`** – In this case the wrapped function should be

tokenclass.check_ping :param *args: args[1] is the pin :param **kwds: kwds["options"] contains the flask g :return: True or False

`privacyidea.lib.policydecorators.`**`auth_user_does_not_exist`**(*wrapped_function*, *user_object*, *passw*, *options=None*)

This decorator checks, if the user does exist at all. If the user does exist, the wrapped function is called.

The wrapped function is usually token.check_user_pass, which takes the arguments (user, passw, options={})

Parameters

- **`wrapped_function`** –
- **`user_object`** –
- **`passw`** –
- **`options`** – Dict containing values for "g" and "clientip"

Returns Tuple of True/False and reply-dictionary

`privacyidea.lib.policydecorators.`**`auth_user_has_no_token`**(*wrapped_function*, *user_object*, *passw*, *options=None*)

This decorator checks if the user has a token at all. If the user has a token, the wrapped function is called.

The wrapped function is usually token.check_user_pass, which takes the arguments (user, passw, options={})

Parameters

- **`wrapped_function`** –
- **`user_object`** –
- **`passw`** –
- **`options`** – Dict containing values for "g" and "clientip"

Returns Tuple of True/False and reply-dictionary privacyidea.lib.policydecorators.**auth_user_passthru**(*wrapped_function*, *user_object*, *passw*, *options=None*)

This decorator checks the policy settings of ACTION.PASSTHRU. If the authentication against the userstore is not successful, the wrapped function is called.

The wrapped function is usually token.check_user_pass, which takes the arguments (user, passw, options={})

Parameters

- **wrapped_function** –
- **user_object** –
- **passw** –
- **options** – Dict containing values for "g" and "clientip"

Returns Tuple of True/False and reply-dictionary

privacyidea.lib.policydecorators.**auth_user_timelimit**(*wrapped_function*, *user_object*, *passw*, *options=None*)

This decorator checks the policy settings of ACTION.AUTHMAXSUCCESS, ACTION.AUTHMAXFAIL If the authentication was successful, it checks, if the number of allowed successful authentications is exceeded (AUTHMAXSUCCESS).

If the AUTHMAXFAIL is exceed it denies even a successful authentication.

The wrapped function is usually token.check_user_pass, which takes the arguments (user, passw, options={})

Parameters

- **wrapped_function** –
- **user_object** –
- **passw** –
- **options** – Dict containing values for "g" and "clientip"

Returns Tuple of True/False and reply-dictionary

privacyidea.lib.policydecorators.**challenge_response_allowed**(*func*)

This decorator is used to wrap tokenclass.is_challenge_request. It checks, if a challenge response authentication is allowed for this token type. To allow this, the policy

scope:authentication, action:challenge_response must be set.

If the tokentype is not allowed for challenge_response, this decorator returns false.

See *challenge_response*.

Parameters **func** – wrapped function

privacyidea.lib.policydecorators.**config_lost_token**(*wrapped_function*, **args*, ***kwds*)

Decorator to decorate the lib.token.lost_token function. Depending on ACTION.LOSTTOKENVALID, ACTION.LOSTTOKENPWCONTENTS, ACTION.LOSTTOKENPWLEN it sets the check_otp parameter, to signal how the lostToken should be generated.

Parameters

- **wrapped_function** – Usually the function lost_token()
- **args** – argument "serial" as the old serial number
- **kwds** – keyword arguments like "validity", "contents", "pw_len"

kwds["options"] contains the flask g

Returns calls the original function with the modified "validity",

"contents" and "pw_len" argument

class privacyidea.lib.policydecorators.**libpolicy**(*decorator_function*)

This is the decorator wrapper to call a specific function before a library call in contrast to prepolicy and postpolicy, which are to be called in API Calls.

The decorator expects a named parameter "options". In this options dict it will look for the flask global "g".

privacyidea.lib.policydecorators.**login_mode**(*wrapped_function*, **args*, ***kwds*)

Decorator to decorate the lib.auth.check_webui_user function. Depending on ACTION.LOGINMODE it sets the check_otp parameter, to signal that the authentication should be performed against privacyIDEA.

Parameters

- **wrapped_function** – Usually the function check_webui_user
- **args** – arguments user_obj and password
- **kwds** – keyword arguments like options and !check_otp!

kwds["options"] contains the flask g :return: calls the original function with the modified "check_otp" argument

privacyidea.lib.policydecorators.**reset_all_user_tokens**(*wrapped_function*, **args*, ***kwds*)

Resets all tokens if the corresponding policy is set.

Parameters

- **token** – The successful token, the tokenowner is used to find policies.
- **tokenobject_list** – The list of all the tokens of the user
- **options** – options dictionary containing g.

Returns None

Event Handler

The following event handlers are known to privacyIDEA

Event Handler Base Class

class privacyidea.lib.eventhandler.base.**BaseEventHandler**

An Eventhandler needs to return a list of actions, which it can handle.

It also returns a list of allowed action and conditions

It returns an identifier, which can be used in the eventhandlig definitions

property actions

This method returns a list of available actions, that are provided by this event handler. :return: dictionary of actions.

property allowed_positions

This returns the allowed positions of the event handler definition. This can be "post" or "pre" or both. :return: list of allowed positions

check_condition(*options*)

Check if all conditions are met and if the action should be executed. The the conditions are met, we return "True" :return: True

property conditions

The UserNotification can filter for conditions like * type of logged in user and * successful or failed value.success

allowed types are str, multi, text, regexp

Returns dict

description = 'This is the base class of an EventHandler with no functionality'

do (*action*, *options=None*)

This method executes the defined action in the given event.

Parameters

- **action** –
- **options** (*dict*) – Contains the flask parameters g and request and the handler_def configuration

Returns

property events

This method returns a list allowed events, that this event handler can be bound to and which it can handle with the corresponding actions.

An eventhandler may return an asterisk [“*”] indicating, that it can be used in all events. :return: list of events

identifier = 'BaseEventHandler'

User Notification Event Handler

class privacyidea.lib.eventhandler.usernotification.**UserNotificationEventHandler**

An Eventhandler needs to return a list of actions, which it can handle.

It also returns a list of allowed action and conditions

It returns an identifier, which can be used in the eventhandlig definitions

property actions

This method returns a dictionary of allowed actions and possible options in this handler module.

Returns dict with actions

property allowed_positions

This returns the allowed positions of the event handler definition. :return: list of allowed positions

description = 'This eventhandler notifies the user about actions on his tokens'

do (*action*, *options=None*)

This method executes the defined action in the given event.

Parameters

- **action** –
- **options** (*dict*) – Contains the flask parameters g, request, response and the handler_def configuration

Returns

identifier = 'UserNotification'

class privacyidea.lib.event.**EventConfiguration**

This class is supposed to contain the event handling configuration during the Request. The currently defined events are fetched from the request-local config object.

property events

Shortcut for retrieving the currently defined event handlers from the request-local config object.

get_event (*eventid*)

Return the reduced list with the given eventid. This list should only have one element.

Parameters **eventid** (*int or None*) – id of the event

Returns list with one element

get_handled_events (*eventname*, *position='post'*)

Return a list of the event handling definitions for the given eventname and the given position.

Parameters

- **eventname** – The name of the event
- **position** – the position of the event definition

Returns

privacyidea.lib.event.**delete_event** (*event_id*)

Delete the event configuration with this given ID. :param event_id: The database ID of the event. :type event_id: int :return:

privacyidea.lib.event.**enable_event** (*event_id*, *enable=True*)

Enable or disable the and event :param event_id: ID of the event :return:

class privacyidea.lib.event.**event** (*eventname*, *request*, *g*)

This is the event decorator that calls the event handler in the handler module. This event decorator can be used at any API call

privacyidea.lib.event.**get_handler_object** (*handlername*)

Return an event handler object based on the Name of the event handler class

Parameters **handlername** – The identifier of the Handler Class

Returns

privacyidea.lib.event.**set_event** (*name*, *event*, *handlermodule*, *action*, *conditions=None*, *ordering=0*, *options=None*, *id=None*, *active=True*, *position='post'*)

Set an event handling configuration. This writes an entry to the database eventhandler.

Parameters

- **name** – The name of the event definition
- **event** (*basestring*) – The name of the event to react on. Can be a single event or a comma separated list.
- **handlermodule** (*basestring*) – The identifier of the event handler module. This is an identifier string like "UserNotification"
- **action** (*basestring*) – The action to perform. This is an action defined by the handler module
- **conditions** (*dict*) – A condition. Only if this condition is met, the action is performed.
- **ordering** (*integer*) – An optional ordering of the event definitions.
- **options** (*dict*) – Additional options, that are needed as parameters for the action

- **id** (*int*) – The DB id of the event. If the id is given, the event is updated. Otherwise a new entry is generated.
- **position** (*basestring*) – The position of the event handler being "post" or "pre"

Returns The id of the event.

SMS Provider

The following SMS providers are know to privacyIDEA

HTTP SMS Provider

class privacyidea.lib.smsprovider.HttpSMSProvider.**HttpSMSProvider**(*db_smsprovider_object=None*, *smsgateway=None*)

classmethod parameters()

Return a dictionary, that describes the parameters and options for the SMS provider. Parameters are required keys to values.

Returns dict

submit_message(*phone*, *message*)

send a message to a phone via an http sms gateway

Parameters

- **phone** – the phone number
- **message** – the message to submit to the phone

Returns

Sipgate SMS Provider

class privacyidea.lib.smsprovider.SipgateSMSProvider.**SipgateSMSProvider**(*db_smsprovider_object=No* *smsgateway=None*)

classmethod parameters()

Return a dictionary, that describes the parameters and options for the SMS provider. Parameters are required keys to values.

Returns dict

submit_message(*phone*, *message*)

Sends the SMS. It should return a bool indicating if the SMS was sent successfully.

In case of SMS send fail, an Exception should be raised. :return: Success :rtype: bool

SMTP SMS Provider

class privacyidea.lib.smsprovider.SmtpSMSProvider.**SmtpSMSProvider**(*db_smsprovider_object=None*, *smsgateway=None*)

classmethod parameters()

Return a dictionary, that describes the parameters and options for the SMS provider. Parameters are required keys to values.

Returns dict

submit_message(*phone*, *message*)

Submits the message for phone to the email gateway.

Returns true in case of success

In case of a failure an exception is raised

SMSProvider is the base class for submitting SMS. It provides 3 different imlementations:

- HTTP: submitting SMS via an HTTP gateway of an SMS provider
- SMTP: submitting SMS via an SMTP gateway of an SMS provider
- Sipgate: submitting SMS via Sipgate service

Base Class

class privacyidea.lib.smsprovider.SMSProvider.**ISMSProvider**(*db_smsprovider_object=None*, *smsgateway=None*)

the SMS Provider Interface - BaseClass

check_configuration()

This method checks the sanity of the configuration of this provider. If there is a configuration error, than an exception is raised. :return:

load_config(*config_dict*)

Load the configuration dictionary

Parameters **config_dict** (`dict`) – The conifugration of the SMS provider

Returns None

classmethod parameters()

Return a dictionary, that describes the parameters and options for the SMS provider. Parameters are required keys to values with defined keys, while options can be any combination.

Each option is the key to another dict, that describes this option, if it is required, a description and which values it can take. The values are optional.

Additional options can not be named in advance. E.g. some provider specific HTTP parameters of HTTP gateways are options. The HTTP parameter for the SMS text could be “text” at one provider and “sms” at another one.

The options can be fixed values or also take the tags {otp}, {user}, {phone}.

Returns dict

submit_message(*phone*, *message*)

Sends the SMS. It should return a bool indicating if the SMS was sent successfully.

In case of SMS send fail, an Exception should be raised. :return: Success :rtype: bool

UserIdResolvers

The useridresolver is responsible for getting userids for loginnames and vice versa.

This base module contains the base class UserIdResolver.UserIdResolver and also the community class PasswdIdResolver.IdResolver, that is inherited from the base class.

Base class

class `privacyidea.lib.resolvers.UserIdResolver.`**`UserIdResolver`**

`add_user`(*attributes=None*)

Add a new user in the useridresolver. This is only possible, if the UserIdResolver supports this and if we have write access to the user store.

Parameters

- **`username`** (*`basestring`*) – The login name of the user
- **`attributes`** – Attributes according to the attribute mapping

Returns The new UID of the user. The UserIdResolver needs to

determine the way how to create the UID.

`checkPass`(*uid*, *password*)

This function checks the password for a given uid. returns true in case of success false if password does not match

Parameters

- **`uid`** (*`string or int`*) – The uid in the resolver
- **`password`** (*`string`*) – the password to check. Usually in cleartext

Returns True or False

Return type bool

`close`()

Hook to close down the resolver after one request

`delete_user`(*uid*)

Delete a user from the useridresolver. The user is referenced by the user id. :param uid: The uid of the user object, that should be deleted. :type uid: basestring :return: Returns True in case of success :rtype: bool

property editable

Return true, if the Instance! of this resolver is configured editable. :return:

classmethod `getResolverClassDescriptor`()

return the descriptor of the resolver, which is - the class name and - the config description

Returns resolver description dict

Return type dict

static `getResolverClassType`()

provide the resolver type for registration

static `getResolverDescriptor`()

return the descriptor of the resolver, which is - the class name and - the config description

Returns resolver description dict

Return type dict

getResolverId()
get resolver specific information :return: the resolver identifier string - empty string if not exist

static getResolverType()
getResolverType - return the type of the resolver

Returns returns the string 'ldapresolver'

Return type string

getUserId(*loginName*)
The loginname is resolved to a user_id. Depending on the resolver type the user_id can be an ID (like in /etc/passwd) or a string (like the DN in LDAP)

It needs to return an emptry string, if the user does not exist.

Parameters **loginName** (*sting*) – The login name of the user

Returns The ID of the user

Return type str

getUserInfo(*userid*)
This function returns all user information for a given user object identified by UserID. :param userid: ID of the user in the resolver :type userid: int or string :return: dictionary, if no object is found, the dictionary is empty :rtype: dict

getUserList(*searchDict=None*)
This function finds the user objects, that have the term 'value' in the user object field 'key'

Parameters **searchDict** (*dict*) – dict with key values of user attributes - the key may be something like 'loginname' or 'email' the value is a regular expression.

Returns list of dictionaries (each dictionary contains a user object) or an empty string if no object is found.

Return type list of dicts

getUsername(*userid*)
Returns the username/loginname for a given userid :param userid: The userid in this resolver :type userid: string :return: username :rtype: string

property has_multiple_loginnames
Return if this resolver has multiple loginname attributes :return: bool

loadConfig(*config*)
Load the configuration from the dict into the Resolver object. If attributes are missing, need to set default values. If required attributes are missing, this should raise an Exception.

Parameters **config** (*dict*) – The configuration values of the resolver

classmethod testconnection(*param*)
This function lets you test if the parameters can be used to create a working resolver. The implementation should try to connect to the user store and verify if users can be retrieved. In case of success it should return a text like "Resolver config seems OK. 123 Users found."

Parameters **param** (*dict*) – The parameters that should be saved as the resolver

Returns returns True in case of success and a descriptive text

Return type tuple

update_user (*uid*, *attributes=None*)

Update an existing user. This function is also used to update the password. Since the attribute mapping know, which field contains the password, this function can also take care for password changing.

Attributes that are not contained in the dict attributes are not modified.

Parameters

- **uid** (*basestring*) – The uid of the user object in the resolver.
- **attributes** (*dict*) – Attributes to be updated.

Returns True in case of success

PasswdResolver

class privacyidea.lib.resolvers.PasswdIdResolver.**IdResolver**

checkPass (*uid*, *password*)

This function checks the password for a given uid. returns true in case of success false if password does not match

We do not support shadow passwords. so the seconds column of the passwd file needs to contain the crypted password

If the password is a unicode object, it is encoded according to ENCODING first.

Parameters

- **uid** (*int*) – The uid of the user
- **password** (*sting*) – The password in cleartext

Returns True or False

Return type bool

checkUserId (*line*, *pattern*)

Check if a userid matches a pattern. A pattern can be "=1000", ">=1000", "<2000" or "between 1000,2000".

Parameters

- **line** (*dict*) – the dictionary of a user
- **pattern** (*string*) – match pattern with <, <=…

Returns True or False

Return type bool

checkUserName (*line*, *pattern*)

check for user name

classmethod getResolverClassDescriptor ()

return the descriptor of the resolver, which is - the class name and - the config description

Returns resolver description dict

Return type dict

static getResolverClassType ()

provide the resolver type for registration

static getResolverDescriptor()
return the descriptor of the resolver, which is - the class name and - the config description

Returns resolver description dict

Return type dict

getResolverId()
return the resolver identifier string, which in fact is filename, where it points to.

static getResolverType()
getResolverType - return the type of the resolver

Returns returns the string 'ldapresolver'

Return type string

getSearchFields(*searchDict=None*)
show, which search fields this userIdResolver supports

TODO: implementation is not completed

Parameters **searchDict** (*dict*) – fields, which can be queried

Returns dict of all searchFields

Return type dict

getUserId(*LoginName*)
search the user id from the login name

Parameters **LoginName** – the login of the user (as unicode)

Returns the userId

Return type str

getUserInfo(*userId*, *no_passwd=False*)
get some info about the user as we only have the loginId, we have to traverse the dict for the value

Parameters

- **userId** – the to be searched user
- **no_passwd** – retrun no password

Returns dict of user info

getUserList(*searchDict=None*)
get a list of all users matching the search criteria of the searchdict

Parameters **searchDict** – dict of search expressions

getUsername(*userId*)
Returns the username/loginname for a given userid :param userid: The userid in this resolver :type userid: string :return: username :rtype: str

loadConfig(*configDict*)
The UserIdResolver could be configured from the pylons app config - here this could be the passwd file , whether it is /etc/passwd or /etc/shadow

loadFile()
Loads the data of the file initially. if the self.fileName is empty, it loads /etc/passwd. Empty lines are ignored.

static setup(*config=None*, *cache_dir=None*)
this setup hook is triggered, when the server starts to serve the first request

Parameters **config** (*the privacyidea config dict*) – the privacyidea config

LDAPResolver

class privacyidea.lib.resolvers.LDAPIdResolver.**IdResolver**

add_user (*attributes=None*)

Add a new user to the LDAP directory. The user can only be created in the LDAP using a DN. So we have to construct the DN out of the given attributes.

attributes are these "username", "surname", "givenname", "email", "mobile", "phone", "password"

Parameters **attributes** (*dict*) – Attributes according to the attribute mapping

Returns The new UID of the user. The UserIdResolver needs to

determine the way how to create the UID.

checkPass (*uid*, *password*)

This function checks the password for a given uid. - returns true in case of success - false if password does not match

static create_connection (*authtype=None*, *server=None*, *user=None*, *password=None*, *auto_bind=False*, *client_strategy='SYNC'*, *check_names=True*, *auto_referrals=False*, *receive_timeout=5*, *start_tls=False*)

Create a connection to the LDAP server.

Parameters

- **authtype** –
- **server** –
- **user** –
- **password** –
- **auto_bind** –
- **client_strategy** –
- **check_names** –
- **auto_referrals** –
- **receive_timeout** – At the moment we do not use this, since receive_timeout is not supported by ldap3 < 2.

Returns

classmethod create_serverpool (*urilist*, *timeout*, *get_info=None*, *tls_context=None*, *rounds=2*, *exhaust=30*, *pool_cls=<class 'ldap3.core.pooling.ServerPool'>*)

This create the serverpool for the ldap3 connection. The URI from the LDAP resolver can contain a comma separated list of LDAP servers. These are split and then added to the pool.

See https://github.com/cannatag/ldap3/blob/master/docs/manual/source/servers.rst#server-pool

Parameters

- **urilist** (*basestring*) – The list of LDAP URIs, comma separated
- **timeout** (*float*) – The connection timeout

- **get_info** – The get_info type passed to the ldap3.Sever constructor. default: ldap3.SCHEMA, should be ldap3.NONE in case of a bind.
- **tls_context** – A ldap3.tls object, which defines if certificate verification should be performed
- **rounds** – The number of rounds we should cycle through the server pool before giving up
- **exhaust** – The seconds, for how long a non-reachable server should be removed from the serverpool
- **pool_cls** – `ldap3.ServerPool` subclass that should be instantiated

Returns Server Pool

Return type serverpool_cls

delete_user(*uid*)

Delete a user from the LDAP Directory.

The user is referenced by the user id. :param uid: The uid of the user object, that should be deleted. :type uid: basestring :return: Returns True in case of success :rtype: bool

property editable

Return true, if the instance of the resolver is configured editable :return:

classmethod getResolverClassDescriptor()

return the descriptor of the resolver, which is - the class name and - the config description

Returns resolver description dict

Return type dict

static getResolverClassType()

provide the resolver type for registration

static getResolverDescriptor()

return the descriptor of the resolver, which is - the class name and - the config description

Returns resolver description dict

Return type dict

getResolverId()

Returns the resolver Id This should be an Identifier of the resolver, preferable the type and the name of the resolver.

Returns the id of the resolver

Return type str

static getResolverType()

getResolverType - return the type of the resolver

Returns returns the string 'ldapresolver'

Return type string

getUserId(*LoginName*)

resolve the loginname to the userid.

Parameters **LoginName** (*str*) – The login name from the credentials

Returns UserId as found for the LoginName

Return type str

getUserInfo(*userId*)
This function returns all user info for a given userid/object.

Parameters **userId** (*string*) – The userid of the object

Returns A dictionary with the keys defined in self.userinfo

Return type dict

getUserList(*searchDict=None*)

Parameters **searchDict** (*dict*) – A dictionary with search parameters

Returns list of users, where each user is a dictionary

getUsername(*user_id*)
Returns the username/loginname for a given user_id :param user_id: The user_id in this resolver :type user_id: string :return: username :rtype: string

get_persistent_serverpool(*get_info=None*)
Return a process-level instance of `LockingServerPool` for the current LDAP resolver configuration. Retrieve it from the app-local store. If such an instance does not exist yet, create one. :param get_info: one of ldap3.SCHEMA, ldap3.NONE, ldap3.ALL :return: a `LockingServerPool` instance

get_serverpool_instance(*get_info=None*)
Return a `ServerPool` instance that should be used. If `SERVERPOOL_PERSISTENT` is enabled, invoke `get_persistent_serverpool` to retrieve a per-process server pool instance. If it is not enabled, invoke `create_serverpool` to retrieve a per-request server pool instance. :param get_info: one of ldap3.SCHEMA, ldap3.NONE, ldap3.ALL :return: a `ServerPool`/`LockingServerPool` instance

property has_multiple_loginnames
Return if this resolver has multiple loginname attributes :return: bool

loadConfig(*config*)
Load the config from conf.

Parameters **config** (*dict*) – The configuration from the Config Table

'#ldap_uri': 'LDAPURI', '#ldap_basedn': 'LDAPBASE', '#ldap_binddn': 'BINDDN', '#ldap_password': 'BINDPW', '#ldap_timeout': 'TIMEOUT', '#ldap_sizelimit': 'SIZELIMIT', '#ldap_loginattr': 'LOGINNAMEATTRIBUTE', '#ldap_searchfilter': 'LDAPSEARCHFILTER', '#ldap_mapping': 'USERINFO', '#ldap_uidtype': 'UIDTYPE', '#ldap_noreferrals' : 'NOREFERRALS', '#ldap_editable' : 'EDITABLE', '#ldap_certificate': 'CACERTIFICATE',

static split_uri(*uri*)
Splits LDAP URIs like: * ldap://server * ldaps://server * ldap[s]://server:1234 * server :param uri: The LDAP URI :return: Returns a tuple of Servername, Port and SSL(bool)

classmethod testconnection(*param*)
This function lets you test the to be saved LDAP connection.

Parameters **param** (*dict*) – A dictionary with all necessary parameter to test the connection.

Returns Tuple of success and a description

Return type (bool, string)

Parameters are: BINDDN, BINDPW, LDAPURI, TIMEOUT, LDAPBASE, LOGINNAMEATTRIBUTE, LDAPSEARCHFILTER, USERINFO, SIZELIMIT, NOREFERRALS, CACERTIFICATE, AUTHTYPE, TLS_VERIFY, TLS_VERSION, TLS_CA_FILE, SERVERPOOL_ROUNDS, SERVERPOOL_SKIP

update_user(*uid*, *attributes=None*)

Update an existing user. This function is also used to update the password. Since the attribute mapping know, which field contains the password, this function can also take care for password changing.

Attributes that are not contained in the dict attributes are not modified.

Parameters

- **uid** (*basestring*) – The uid of the user object in the resolver.
- **attributes** (*dict*) – Attributes to be updated.

Returns True in case of success

Audit log

Base class

class privacyidea.lib.auditmodules.base.**Audit**(*config=None*)

add_policy(*policyname*)

This method adds a triggered policyname to the list of triggered policies. :param policyname: A string or a list of strings as policynames :return:

add_to_log(*param*, *add_with_comma=False*)

Add to existing log entry :param param: :param add_with_comma: If set to true, new values will be appended comma separated :return:

audit_entry_to_dict(*audit_entry*)

If the search_query returns an iterator with elements that are not a dictionary, the audit module needs to provide this function, to convert the audit entry to a dictionary.

csv_generator(*param=None*, *user=None*, *timelimit=None*)

A generator that can be used to stream the audit log

Parameters **param** –

Returns

finalize_log()

This method is called to finalize the audit_data. I.e. sign the data and write it to the database. It should hash the data and do a hash chain and sign the data

get_audit_id()

get_count(*search_dict*, *timedelta=None*, *success=None*)

Returns the number of found log entries. E.g. used for checking the timelimit.

Parameters **param** – List of filter parameters

Returns number of found entries

get_total(*param*, *AND=True*, *display_error=True*, *timelimit=None*)

This method returns the total number of audit entries in the audit store

property has_data

initialize()

initialize_log(*param*)

This method initialized the log state. The fact, that the log state was initialized, also needs to be logged. Therefor the same params are passed as i the log method.

is_readable = False

log (*param*)

This method is used to log the data. During a request this method can be called several times to fill the internal audit_data dictionary.

Add new log details in param to the internal log data self.audit_data.

Parameters **param** (`dict`) – Log data that is to be added

Returns None

log_token_num (*count*)

Log the number of the tokens. Can be passed like log_token_num(get_tokens(count=True))

Parameters **count** (`int`) – Number of tokens

Returns

read_keys (*pub*, *priv*)

Set the private and public key for the audit class. This is achieved by passing the entries.

#priv = config.get("privacyideaAudit.key.private") #pub = config.get("privacyideaAudit.key.public")

Parameters

- **pub** (`string with filename`) – Public key, used for verifying the signature
- **priv** (`string with filename`) – Private key, used to sign the audit entry

Returns None

search (*search_dict*, *page_size=15*, *page=1*, *sortorder='asc'*, *timelimit=None*)

This function is used to search audit events.

param: Search parameters can be passed.

return: A pagination object

This function is deprecated.

search_query (*search_dict*, *page_size=15*, *page=1*, *sortorder='asc'*, *sortname='number'*, *timelimit=None*)

This function returns the audit log as an iterator on the result

SQL Audit module

class privacyidea.lib.auditmodules.sqlaudit.**Audit** (*config=None*)

This is the SQLAudit module, which writes the audit entries to an SQL database table. It requires the configuration parameters in pi.cfg: * PI_AUDIT_KEY_PUBLIC * PI_AUDIT_KEY_PRIVATE

If you want to host the SQL Audit database in another DB than the token DB, you can use: * PI_AUDIT_SQL_URI

It also takes the optional parameters: * PI_AUDIT_POOL_SIZE * PI_AUDIT_POOL_RECYCLE * PI_AUDIT_SQL_TRUNCATE * PI_AUDIT_NO_SIGN

You can use PI_AUDIT_NO_SIGN = True to avoid signing of the audit log.

If PI_CHECK_OLD_SIGNATURES = True old style signatures (text-book RSA) will be checked as well, otherwise they will be marked as 'FAIL'.

audit_entry_to_dict (*audit_entry*)

If the search_query returns an iterator with elements that are not a dictionary, the audit module needs to provide this function, to convert the audit entry to a dictionary.

clear()
Deletes all entries in the database table. This is only used for test cases! :return:

csv_generator(*param=None*, *user=None*, *timelimit=None*)
Returns the audit log as csv file. :param timelimit: Limit the number of dumped entries by time :type timelimit: datetime.timedelta :param param: The request parameters :type param: dict :param user: The user, who issued the request :return: None. It yields results as a generator

finalize_log()
This method is used to log the data. It should hash the data and do a hash chain and sign the data

get_count(*search_dict*, *timedelta=None*, *success=None*)
Returns the number of found log entries. E.g. used for checking the timelimit.

Parameters **param** – List of filter parameters

Returns number of found entries

get_total(*param*, *AND=True*, *display_error=True*, *timelimit=None*)
This method returns the total number of audit entries in the audit store

search(*search_dict*, *page_size=15*, *page=1*, *sortorder='asc'*, *timelimit=None*)
This function returns the audit log as a Pagination object.

Parameters **timelimit** (*timedelta*) – Only audit entries newer than this timedelta will be searched

search_query(*search_dict*, *page_size=15*, *page=1*, *sortorder='asc'*, *sortname='number'*, *timelimit=None*)
This function returns the audit log as an iterator on the result

Parameters **timelimit** (*timedelta*) – Only audit entries newer than this timedelta will be searched

Monitoring

Base class

class privacyidea.lib.monitoringmodules.base.**Monitoring**(*config=None*)

add_value(*stats_key*, *stats_value*, *timestamp*, *reset_values=False*)
This method adds a measurement point to the statistics key “stats_key”. If reset_values is set to True, all old values of this stats_key are deleted.

Parameters

- **stats_key** – Identifier of the stats
- **stats_value** – measured value
- **timestamp** (*timezone aware datetime*) – the timestamp of the measurement
- **reset_values** – boolean to indicate the reset

Returns None

delete(*stats_key*, *start_timestamp*, *end_timestamp*)
Delete all entries of the stats_key for the given time frame. The start_timestamp and end_timestamp are also deleted.

Parameters

- **stats_key** – Identifier of the stats
- **start_timestamp** (*timezone aware datetime*) – beginning of the time frame
- **end_timestamp** (*timezone aware datetime*) – end of the time frame

Returns number of deleted entries

get_keys()
Return a list of the available statistic keys.

Returns list of identifiers

get_last_value(*stats_key*)
returns the last value of the given stats_key in time. :param stats_key: The identifier of the stats :return: a string value.

get_values(*stats_key*, *start_timestamp=None*, *end_timestamp=None*)
Return a list of tuples of (timestamp, value) for the requested stats_key.

Parameters

- **stats_key** – Identifier of the stats
- **start_timestamp** (*timezone aware datetime*) – start of the time frame
- **end_timestamp** (*timezone aware datetime*) – end of the time frame

Returns

SQL Statistics module

class privacyidea.lib.monitoringmodules.sqlstats.**Monitoring**(*config=None*)

add_value(*stats_key*, *stats_value*, *timestamp*, *reset_values=False*)
This method adds a measurement point to the statistics key "stats_key". If reset_values is set to True, all old values of this stats_key are deleted.

Parameters

- **stats_key** – Identifier of the stats
- **stats_value** – measured value
- **timestamp** (*timezone aware datetime*) – the timestamp of the measurement
- **reset_values** – boolean to indicate the reset

Returns None

delete(*stats_key*, *start_timestamp*, *end_timestamp*)
Delete all entries of the stats_key for the given time frame. The start_timestamp and end_timestamp are also deleted.

Parameters

- **stats_key** – Identifier of the stats
- **start_timestamp** (*timezone aware datetime*) – beginning of the time frame
- **end_timestamp** (*timezone aware datetime*) – end of the time frame

Returns number of deleted entries

get_keys()

Return a list of all stored keys. :return:

get_last_value(*stats_key*)

returns the last value of the given stats_key in time. :param stats_key: The identifier of the stats :return: a string value.

get_values(*stats_key*, *start_timestamp=None*, *end_timestamp=None*, *date_strings=False*)

Return a list of tuples of (timestamp, value) for the requested stats_key.

Parameters

- **stats_key** – Identifier of the stats
- **start_timestamp** (*timezone aware datetime*) – start of the time frame
- **end_timestamp** (*timezone aware datetime*) – end of the time frame

Returns

Machine Resolvers

Machine Resolvers are used to find machines in directories like LDAP, Active Directory, puppet, salt, or the /etc/hosts file.

Machines can then be used to assign applications and tokens to those machines.

Base class

class privacyidea.lib.machines.base.**BaseMachineResolver**(*name*, *config=None*)

static get_config_description()

Returns a description what config values are expected and allowed.

Returns dict

get_machine_id(*hostname=None*, *ip=None*)

Returns the machine id for a given hostname or IP address.

If hostname and ip is given, the resolver should also check that the hostname matches the IP. If it can check this and hostname and IP do not match, then an Exception must be raised.

Parameters

- **hostname** (*basestring*) – The hostname of the machine
- **ip** (*netaddr*) – IP address of the machine

Returns The machine ID, which depends on the resolver

Return type basestring

get_machines(*machine_id=None*, *hostname=None*, *ip=None*, *any=None*, *substring=False*)

Return a list of all machine objects in this resolver

Parameters **substring** – If set to true, it will also match search_hostnames,

that only are a subnet of the machines hostname. :type substring: bool :param any: a substring that matches EITHER hostname, machineid or ip :type any: basestring :return: list of machine objects

load_config(*config*)

This loads the configuration dictionary, which contains the necessary information for the machine resolver to find and connect to the machine store.

Parameters **config** (*dict*) – The configuration dictionary to run the machine resolver

Returns None

static testconnection(*params*)

This method can test if the passed parameters would create a working machine resolver.

Parameters **params** –

Returns tupple of success and description

Return type (bool, string)

Hosts Machine Resolver

class privacyidea.lib.machines.hosts.**HostsMachineResolver**(*name*, *config=None*)

classmethod get_config_description()

Returns a description what config values are expected and allowed.

Returns dict

get_machine_id(*hostname=None*, *ip=None*)

Returns the machine id for a given hostname or IP address.

If hostname and ip is given, the resolver should also check that the hostname matches the IP. If it can check this and hostname and IP do not match, then an Exception must be raised.

Parameters

- **hostname** (*basestring*) – The hostname of the machine
- **ip** (*netaddr*) – IP address of the machine

Returns The machine ID, which depends on the resolver

Return type basestring

get_machines(*machine_id=None*, *hostname=None*, *ip=None*, *any=None*, *substring=False*)

Return matching machines.

Parameters

- **machine_id** – can be matched as substring
- **hostname** – can be matched as substring
- **ip** – can not be matched as substring
- **substring** (*bool*) – Whether the filtering should be a substring matching
- **any** (*basestring*) – a substring that matches EITHER hostname, machineid or ip

Returns list of Machine Objects

load_config(*config*)

This loads the configuration dictionary, which contains the necessary information for the machine resolver to find and connect to the machine store.

Parameters **config** (*dict*) – The configuration dictionary to run the machine resolver

Returns None

static testconnection(*params*)

Test if the given filename exists.

Parameters **params** –

Returns

PinHandler

This module provides the PIN Handling base class. In case of enrolling a token, a PIN Handling class can be used to send the PIN via Email, call an external program or print a letter.

This module is not tested explicitly. It is tested in conjunction with the policy decorator init_random_pin in tests/test_api_lib_policy.py

Base class

class privacyidea.lib.pinhandling.base.**PinHandler**(*options=None*)

A PinHandler Class is responsible for handling the OTP PIN during enrollment.

It receives the necessary data like

- the PIN
- the serial number of the token
- the username
- all other user data:
  - given name, surname
  - email address
  - telephone
  - mobile (if the module would deliver via SMS)
- the administrator name (who enrolled the token)

send(*pin*, *serial*, *user*, *tokentype=None*, *logged_in_user=None*, *userdata=None*, *options=None*)

Parameters

- **pin** – The PIN in cleartext
- **user** (*user object*) – the owner of the token
- **tokentype** (*basestring*) – the type of the token
- **logged_in_user** (*dict*) – The logged in user, who enrolled the token
- **userdata** (*dict*) – Handler-specific user data like email, mobile…
- **options** (*dict*) – Handler-specific additional options

Returns True in case of success

Return type bool

1.15.3 DB level

On the DB level you can simply modify all objects.

The database model

class `privacyidea.models.`**`Admin`**(***kwargs*)

The administrators for managing the system. To manage the administrators use the command pi-manage.

In addition certain realms can be defined to be administrative realms.

Parameters

- **`username`** (*`basestring`*) – The username of the admin
- **`password`** (*`basestring`*) – The password of the admin (stored using PBKDF2, salt and pepper)
- **`email`** (*`basestring`*) – The email address of the admin (not used at the moment)

class `privacyidea.models.`**`Audit`**(*action='', success=0, serial='', token_type='', user='', realm='', resolver='', administrator='', action_detail='', info='', privacyidea_server='', client='', loglevel='default', clearance_level='default', policies=''*)

This class stores the Audit entries

class `privacyidea.models.`**`AuthCache`**(*username, realm, resolver, authentication, first_auth=None, last_auth=None*)

class `privacyidea.models.`**`CAConnector`**(*name, catype*)

The table "caconnector" contains the names and types of the defined CA connectors. Each connector has a different configuration, that is stored in the table "caconnectorconfig".

class `privacyidea.models.`**`CAConnectorConfig`**(*caconnector_id=None, Key=None, Value=None, caconnector=None, Type='', Description=''*)

Each CAConnector can have multiple configuration entries. Each CA Connector type can have different required config values. Therefor the configuration is stored in simple key/value pairs. If the type of a config entry is set to "password" the value of this config entry is stored encrypted.

The config entries are referenced by the id of the resolver.

class `privacyidea.models.`**`Challenge`**(*serial, transaction_id=None, challenge='', data='', session='', validitytime=120*)

Table for handling of the generic challenges.

`get`(*timestamp=False*)

return a dictionary of all vars in the challenge class

Parameters **`timestamp`** (*`bool`*) – if true, the timestamp will given in a readable format 2014-11-29 21:56:43.057293

Returns dict of vars

`get_otp_status`()

This returns how many OTPs were already received for this challenge. and if a valid OTP was received.

Returns tuple of count and True/False

Return type tuple

`is_valid`()

Returns true, if the expiration time has not passed, yet. :return: True if valid :rtype: bool

set_data(*data*)

set the internal data of the challenge :param data: unicode data :type data: string, length 512

class privacyidea.models.**ClientApplication**(***kwargs*)

This table stores the clients, which sent an authentication request to privacyIDEA. This table is filled automatically by authentication requests.

class privacyidea.models.**Config**(*Key*, *Value*, *Type=''*, *Description=''*)

The config table holds all the system configuration in key value pairs.

Additional configuration for realms, resolvers and machine resolvers is stored in specific tables.

class privacyidea.models.**EventCounter**(*name*, *value=0*, *node=''*)

This table stores counters of the event handler “Counter”.

Note that an event counter name does *not* correspond to just one, but rather *several* table rows, because we store event counters for each privacyIDEA node separately. This is intended to improve the performance of replicated setups, because each privacyIDEA node then only writes to its own “private” table row. This way, we avoid locking issues that would occur if all nodes write to the same table row.

decrease()

Decrease the value of a counter. :return:

increase()

Increase the value of a counter :return:

class privacyidea.models.**EventHandler**(*name*, *event*, *handlermodule*, *action*, *condition=''*, *ordering=0*, *options=None*, *id=None*, *conditions=None*, *active=True*, *position='post'*)

This model holds the list of defined events and actions to this events. A handler module can be bound to an event with the corresponding condition and action.

get()

Return the serialized eventhandler object including the options

Returns complete dict

Rytpe dict

class privacyidea.models.**EventHandlerCondition**(*eventhandler_id*, *Key*, *Value*, *comparator='equal'*)

Each EventHandler entry can have additional conditions according to the handler module

class privacyidea.models.**EventHandlerOption**(*eventhandler_id*, *Key*, *Value*, *Type=''*, *Description=''*)

Each EventHandler entry can have additional options according to the handler module.

class privacyidea.models.**MachineResolver**(*name*, *rtype*)

This model holds the definition to the machinestore. Machines could be located in flat files, LDAP directory or in puppet services or other…

The usual MachineResolver just holds a name and a type and a reference to its config

class privacyidea.models.**MachineResolverConfig**(*resolver_id=None*, *Key=None*, *Value=None*, *resolver=None*, *Type=''*, *Description=''*)

Each Machine Resolver can have multiple configuration entries. The config entries are referenced by the id of the machine resolver

class privacyidea.models.**MachineToken**(*machineresolver_id=None*, *machineresolver=None*, *machine_id=None*, *token_id=None*, *serial=None*, *application=None*)

The MachineToken assigns a Token and an application type to a machine. The Machine is represented as the tuple of machineresolver.id and the machine_id. The machine_id is defined by the machineresolver.

This can be an n:m mapping.

class privacyidea.models.**MachineTokenOptions**(*machinetoken_id*, *key*, *value*)

This class holds an Option for the token assigned to a certain client machine. Each Token-Clientmachine-Combination can have several options.

class privacyidea.models.**MethodsMixin**

This class mixes in some common Class table functions like delete and save

class privacyidea.models.**MonitoringStats**(*timestamp*, *key*, *value*)

This is the table that stores measured, arbitrary statistic points in time.

This could be used to store time series but also to store current values, by simply fetching the last value from the database.

class privacyidea.models.**PasswordReset**(*recoverycode*, *username*, *realm*, *resolver=''*, *email=None*, *timestamp=None*, *expiration=None*, *expiration_seconds=3600*)

Table for handling password resets. This table stores the recoverycodes sent to a given user

The application should save the HASH of the recovery code. Just like the password for the Admins the appliaction shall salt and pepper the hash of the recoverycode. A database admin will not be able to inject a rogue recovery code.

A user can get several recoverycodes. A recovery code has a validity period

Optional: The email to which the recoverycode was sent, can be stored.

class privacyidea.models.**PeriodicTask**(*name*, *active*, *interval*, *node_list*, *taskmodule*, *ordering*, *options=None*, *id=None*)

This class stores tasks that should be run periodically.

property aware_last_update

Return self.last_update with attached UTC tzinfo

get()

Return the serialized periodic task object including the options and last runs. The last runs are returned as timezone-aware UTC datetimes.

Returns complete dict

save()

If the entry has an ID set, update the entry. If not, create one. Set `last_update` to the current time. :return: the entry ID

set_last_run(*node*, *timestamp*)

Store the information that the last run of the periodic job occurred on `node` at `timestamp`. :param node: Node name as a string :param timestamp: Timestamp as UTC datetime (without timezone information) :return:

class privacyidea.models.**PeriodicTaskLastRun**(*periodictask_id*, *node*, *timestamp*)

Each PeriodicTask entry stores, for each node, the timestamp of the last successful run.

property aware_timestamp

Return self.timestamp with attached UTC tzinfo

save()

Create or update a PeriodicTaskLastRun entry, depending on the value of `self.id`. :return: the entry id

class privacyidea.models.**PeriodicTaskOption**(*periodictask_id*, *key*, *value*)

Each PeriodicTask entry can have additional options according to the task module.

save()

Create or update a PeriodicTaskOption entry, depending on the value of `self.id` :return: the entry ID

class privacyidea.models.**Policy**(*name*, *active=True*, *scope=''*, *action=''*, *realm=''*, *adminrealm=''*, *adminuser=''*, *resolver=''*, *user=''*, *client=''*, *time=''*, *priority=1*, *check_all_resolvers=False*, *conditions=None*)

The policy table contains policy definitions which control the behaviour during

- enrollment
- authentication
- authorization
- administration
- user actions

get(*key=None*)

Either returns the complete policy entry or a single value :param key: return the value for this key :type key: string :return: complete dict or single value :rytpe: dict or value

get_conditions_tuples()

Returns a list of 5-tuples (section, key, comparator, value, active).

set_conditions(*conditions*)

Replace the list of conditions of this policy with a new list of conditions, i.e. a list of 5-tuples (section, key, comparator, value, active).

class privacyidea.models.**PolicyCondition**(***kwargs*)

as_tuple()

Returns the condition as a tuple (section, key, comparator, value, active)

class privacyidea.models.**PrivacyIDEAServer**(***kwargs*)

This table can store remote privacyIDEA server definitions

class privacyidea.models.**RADIUSServer**(***kwargs*)

This table can store configurations of RADIUS servers. https://github.com/privacyidea/privacyidea/issues/321

It saves * a unique name * a description * an IP address a * a Port * a secret * timeout in seconds (default 5) * retries (default 3)

These RADIUS server definition can be used in RADIUS tokens or in a radius passthru policy.

save()

If a RADIUS server with a given name is save, then the existing RADIUS server is updated.

class privacyidea.models.**Realm**(*realm*)

The realm table contains the defined realms. User Resolvers can be grouped to realms. This very table contains just contains the names of the realms. The linking to resolvers is stored in the table "resolverrealm".

class privacyidea.models.**Resolver**(*name*, *rtype*)

The table "resolver" contains the names and types of the defined User Resolvers. As each Resolver can have different required config values the configuration of the resolvers is stored in the table "resolverconfig".

class privacyidea.models.**ResolverConfig**(*resolver_id=None*, *Key=None*, *Value=None*, *resolver=None*, *Type=''*, *Description=''*)

Each Resolver can have multiple configuration entries. Each Resolver type can have different required config values. Therefor the configuration is stored in simple key/value pairs. If the type of a config entry is set to "password" the value of this config entry is stored encrypted.

The config entries are referenced by the id of the resolver.

class privacyidea.models.**ResolverRealm**(*resolver_id=None*, *realm_id=None*, *resolver_name=None*, *realm_name=None*, *priority=None*)

This table stores which Resolver is located in which realm This is a N:M relation

class privacyidea.models.**SMSGateway**(*identifier*, *providermodule*, *description=None*, *options=None*)

This table stores the SMS Gateway definitions. See https://github.com/privacyidea/privacyidea/wiki/concept:-Delivery-Gateway

It saves the * unique name * a description * the SMS provider module

All options and parameters are saved in other tables.

as_dict()

Return the object as a dictionary

Returns complete dict

Rytpe dict

delete()

When deleting an SMS Gateway we also delete all the options. :return:

property option_dict

Return all connected options as a dictionary

Returns dict

class privacyidea.models.**SMSGatewayOption**(*gateway_id*, *Key*, *Value*, *Type=None*)

This table stores the options and parameters for an SMS Gateway definition.

class privacyidea.models.**SMTPServer**(***kwargs*)

This table can store configurations for SMTP servers. Each entry represents an SMTP server. EMail Token, SMS SMTP Gateways or Notifications like PIN handlers are supposed to use a reference to to a server definition. Each Machine Resolver can have multiple configuration entries. The config entries are referenced by the id of the machine resolver

get()

Returns the configuration as a dictionary

class privacyidea.models.**Subscription**(***kwargs*)

This table stores the imported subscription files.

get()

Return the database object as dict :return:

class privacyidea.models.**TimestampMethodsMixin**

This class mixes in the table functions including update of the timestamp

class privacyidea.models.**Token**(*serial*, *tokentype=''*, *isactive=True*, *otplen=6*, *otpkey=''*, *userid=None*, *resolver=None*, *realm=None*, ***kwargs*)

The table "token" contains the basic token data like

- serial number
- assigned user
- secret key…

while the table "tokeninfo" contains additional information that is specific to the tokentype.

del_info(*key=None*)

Deletes tokeninfo for a given token. If the key is omitted, all Tokeninfo is deleted.

Parameters **key** – searches for the given key to delete the entry

Returns

get(*key=None*, *fallback=None*, *save=False*)

simulate the dict behaviour to make challenge processing easier, as this will have to deal as well with 'dict only challenges'

Parameters

- **key** – the attribute name - in case of key is not provided, a dict of all class attributes are returned
- **fallback** – if the attribute is not found, the fallback is returned
- **save** – in case of all attributes and save==True, the timestamp is converted to a string representation

get_hashed_pin(*pin*)

calculate a hash from a pin Fix for working with MS SQL servers MS SQL servers sometimes return a '<space>' when the column is empty: ''

Parameters **pin** (`str`) – the pin to hash

Returns hashed pin with current pin_seed

Return type str

get_info()

Returns The token info as dictionary

get_realms()

return a list of the assigned realms :return: realms :rtype: list

get_user_pin()

return the userPin :rtype : the PIN as a secretObject

set_hashed_pin(*pin*)

Set the pin of the token in hashed format

Parameters **pin** (`str`) – the pin to hash

Returns the hashed pin

Return type str

set_info(*info*)

Set the additional token info for this token

Entries that end with ".type" are used as type for the keys. I.e. two entries sshkey="XYZ" and sshkey.type="password" will store the key sshkey as type "password".

Parameters **info** (`dict`) – The key-values to set for this token

set_pin(*pin*, *hashed=True*)

set the OTP pin in a hashed way

set_realms(*realms*, *add=False*)

Set the list of the realms. This is done by filling the tokenrealm table. :param realms: realms :type realms: list :param add: If set, the realms are added. I.e. old realms are not

deleted

set_so_pin(*soPin*)

For smartcards this sets the security officer pin of the token

:rtype : None

update_otpkey(*otpkey*)

in case of a new hOtpKey we have to do some more things

update_type(*typ*)

in case the previous has been different type we must reset the counters But be aware, ray, this could also be upper and lower case mixing…

class privacyidea.models.**TokenInfo**(*token_id*, *Key*, *Value*, *Type=None*, *Description=None*)

The table "tokeninfo" is used to store additional, long information that is specific to the tokentype. E.g. the tokentype "TOTP" has additional entries in the tokeninfo table for "timeStep" and "timeWindow", which are stored in the column "Key" and "Value".

The tokeninfo is reference by the foreign key to the "token" table.

class privacyidea.models.**TokenOwner**(*token_id=None*, *serial=None*, *user_id=None*, *resolver=None*, *realm_id=None*, *realmname=None*)

This tables stores the owner of a token. A token can be assigned to several users.

class privacyidea.models.**TokenRealm**(*realm_id=0*, *token_id=0*, *realmname=None*)

This table stores to which realms a token is assigned. A token is in the realm of the user it is assigned to. But a token can also be put into many additional realms.

save()

We only save this, if it does not exist, yet.

class privacyidea.models.**UserCache**(*username*, *used_login*, *resolver*, *user_id*, *timestamp*)

privacyidea.models.**cleanup_challenges**()

Delete all challenges, that have expired.

Returns None

privacyidea.models.**get_machineresolver_id**(*resolvername*)

Return the database ID of the machine resolver :param resolvername: :return:

privacyidea.models.**get_machinetoken_id**(*machine_id*, *resolver_name*, *serial*, *application*)

Returns the ID in the machinetoken table

Parameters

- **machine_id** (*basestring*) – The resolverdependent machine_id
- **resolver_name** (*basestring*) – The name of the resolver
- **serial** (*basestring*) – the serial number of the token
- **application** (*basestring*) – The application type

Returns The ID of the machinetoken entry

Return type int

privacyidea.models.**get_token_id**(*serial*)

Return the database token ID for a given serial number :param serial: :return: token ID :rtpye: int

privacyidea.models.**save_config_timestamp**(*invalidate_config=True*)

Save the current timestamp to the database, and optionally invalidate the current request-local config object. :param invalidate_config: defaults to True

1.16 Frequently Asked Questions

1.16.1 Customization

There are several different ways to customize the UI of privacyIDEA.

Templates

You can change the HTML templates of the web UI as follows. You can create a copy of the orignial templates, modify them and use rewrite rules of your webserver to call your new, modified templates.

This way updates will not affect your modifications.

All HTML views are contained in:

```
static/components/<component>/views/<view>.html
```

You can find them on *GitHub <https://github.com/privacyidea/privacyidea/tree/master/privacyidea/static>* or at the according location in your installation.

Follow these basic steps:

1. Create a new location, where you will keep your modifications safe from updates. You should create a directory like */etc/privacyidea/customization/* and put your modified views in there.
2. Activate the rewrite rules in your web server. E.g. in the Apache configuration you can add entries like:

   ```
   RewriteEngine On
   RewriteRule "/static/components/login/views/login.html"  \
        "/etc/privacyidea/customization/mylogin.html"
   ```

   and apply all required changes to the file *mylogin.html*.

   Note: In this case you need to create a `RewriteRule` for each file, you want to modify.

3. Now activate `mod_rewrite` and reload apache2.

Warning: Of course - if there are functional enhancements or bug fixes in the original templates - your template will also not be affected by these.

Translating templates

The translation in privacyIDEA is very flexible (see *Setup translation*). But if you change the templates the normal translation with PO files can get a bit tricky.

Starting with privacyIDEA 3.0.1 you can use the scope variable `browserLanguage` in your custom templates.

You can print the browser language like this `{{ browserLanguage }}`.

And you can display text in different languages in `divs` like this:

```
<div ng-show="browserLanguage === 'de'">
    Das ist ein deutscher Text.
</div>
<div ng-show="browserLanguage === 'en'">
    This is an English text.
</div>
```

Themes

You can adapt the style and colors by changing CSS. There are at least two ways to do this.

Providing your own stylesheet in the config file

You can create your own CSS file to adapt the look and feel of the Web UI. The default CSS is the bootstrap CSS theme. Using `PI_CSS` in `pi.cfg` you can specify the URL of your own CSS file. The default CSS file url is */static/contrib/css/bootstrap-theme.css*. The file in the file system is located at *privacyidea/static/contrib/css*. You might add a directory *privacyidea/static/custom/css/* and add your CSS file there.

Use web server rewrite modules

Again you can also use the Apache rewrite module to replace the original css file:

```
RewriteEngine On
RewriteRule "/static/contrib/css/bootstrap-theme.css"  \
     "/etc/privacyidea/customization/my.css"
```

A good stating point might be the themes at http://bootswatch.com.

Note: If you add your own CSS file, the file *bootstrap-theme.css* will not be loaded anymore. So you might start with a copy of the original file.

Use web server substitute module

You can also use the substitute module of the Apache webserver. It is not clear how much performance impact you get, since this module can scan and replace any text that is delivered by the web server.

If you for example want to replace the title of the webpages, you could do it like this:

```
<Location "/">
    AddOutputFilterByType SUBSTITUTE text/html
    Substitute "s/>privacyidea Authentication System</>My own 2FA system</ni"
</Location>
```

Logo

The default logo is located at `privacyidea/static/css/privacyIDEA1.png`. If you want to use your own logo, you can put youf file "mylogo.png" just in the same folder and set

> PI_LOGO = "mylogo.png"

in the `pi.cfg` config file.

Page title

You can configure the page title by setting `PI_PAGE_TITLE` in the `pi.cfg` file.

Menu

The administrator can adapt the menu of the web UI using policies or of course web server rewrite rules. The original menu is located in `static/templates/menu.html`.

Note that policies are also dependent on the client IP, this way different clients could see different menus.

Read more about it at the web UI policies at the *custom_menu*.

Headers and Footers

The administrator can change the header and footer of each page. We call this the baseline of the web UI. The original baseline is contained in `static/templates/baseline.html`. You can use a web UI policy to change this baseline or - of course - could use the web server rewrite module.

Read more about changing it via the web UI policies at *custom_baseline*.

Tokenwizard

You can add additional HTML elements above and underneath the enrollment wizard pages. Read the *Token Enrollment Wizard* and *tokenwizard* to learn more about those code snippets.

Token customization

Some tokens allow a special customization.

The paper token allows you to add CSS for styling the printed output and add additional headers and footers. Read more about it at the paper token *Customization*.

1.16.2 How can I create users in the privacyIDEA Web UI?

So you installed privacyIDEA and want to enroll tokens to the users and are wondering how to create users.

privacyIDEA can read users from different existing sources like LDAP, SQL, flat files and SCIM.

You very much likely already have an application (like your VPN or a Web Application…) for which you want to increase the logon security. Then this application already knows users. Either in an LDAP or in an SQL database. Most web applications keep their users in a (My)SQL database. And you also need to create users in this very user database for the user to be able to use this application.

Please read the sections *UserIdResolvers* and usersview for more details.

But you also can define and editable SQL resolver. I.e. you can edit and create new users in an SQL user store.

If you do not have an existing SQL database with users, you can simple create a new database with one table for the users and according rows.

1.16.3 So what's the thing with all the admins?

privacyIDEA comes with its own admins, who are stored in a database table `Admin` in its own database (*The database model*). You can use the tool `pi-manage` to manage those admins from the command line as the system's root user. (see *Installation*)

These admin users can logon to the WebUI using the admin's user name and the specified password. These admins are used to get a simple quick start.

Then you can define realms (see *Realms*), that should be administrative realms. I.e. each user in this realm will have administrative rights in the WebUI.

Note: You need to configure these realms within privacyIDEA. Only after these realms exist, you can raise their rights to an administrative role.

Note: Use this carefully. Imagine you defined a resolver to a specific group in your Active Directory to be the pricacyIDEA admins. Then the Active Directory domain admins can simply add users to be administrator in privacyIDEA.

You define the administrative realms in the config file `pi.cfg`, which is usually located at `/etc/privacyidea/pi.cfg`:

```
SUPERUSER_REALM = ["adminrealm1", "super", "boss"]
```

In this case all the users in the realms "adminrealm1", "super" and "boss" will have administrative rights in the WebUI, when they login with this realm.

As for all other users, you can use the *login_mode* to define, if these administrators should login to the WebUI with their userstore password or with an OTP token.

1.16.4 What are possible rollout strategies?

There are different ways to enroll tokens to a big number of users. Here are some selected high level ideas, you can do with privacyIDEA.

Autoenrollment

Using the *autoassignment* policy you can distribute physical tokens to the users. The users just start using the tokens.

Registration Code

If your users are physically not available and spread around the world, you can send a registration code to the users by postal mail. The registration code is a special token type which can be used by the user to authenticate with 2FA. If used once, the registration token get deleted and can not be used anymore. While logged in, the user can enroll a token on his own.

Automatic initial synchronization

Hardware TOTP tokens may get out of sync due to clock shift. HOTP tokens may get out of sync due to unused key presses. To cope with this you can activate *AutoResync*.

But if you are importing hardware tokens, the clock in the TOTP token may already be out of sync and you do not want the user to authenticate twice, where the first authentication fails.

In this case you can use the following workflow.

In the TOTP token settings you can set the `timeWindow` to a very high value. Note that this timeWindow are the seconds that privacyIDEA will search for the valid OTP value *before* and *after* the current time. E.g. you can set this to 86400. This way you allow the clock in the TOTP token to have drifted for a maximum of one day.

As you do not want such a big window for all authentications, you can automatically reset the `timeWindow`. You can achieve this by creating an event definition:

- event: *validate_check*
- handler: *token*
- condition: * tokentype=TOTP * count_auth_success=1
- action=set tokeninfo * key=*timeWindow* * value=*180*

This way with the first successful authentication of a TOTP token the `timeWindow` of the TOTP token is set to 180 seconds.

1.16.5 How can I translate to my language?

The web UI can be translated into different languages. The system determines the preferred language of you browser and displays the web UI accordingly.

At the moment “en” and “de” are available.

1.16.6 What are possible migration strategies?

You are already running an OTP system like RSA SecurID, SafeNet or Vasco (alphabetical order) but you would like to migrate to privacyIDEA.

There are different migration strategies using the *RADIUS* token or the RADIUS *passthru* policy.

RADIUS token migration strategy

Configure your application like your VPN to authenticate against the privacyIDEA RADIUS server and not against the old deprecated RADIUS server.

Now, you can enroll a *RADIUS* token for each user, who is supposed to login to this application. Configure the RADIUS token for each user so that the RADIUS request is forwarded to the old RADIUS server.

Now you can start to enroll tokens for the users within privacyIDEA. After enrolling a new token in privacyIDEA you can delete the RADIUS token for this user.

When all RADIUS tokens are deleted, you can switch off the old RADIUS server.

For strategies on enrolling token see *What are possible rollout strategies?*.

RADIUS PASSTHRU policy migration strategy

Configure your application like your VPN to authenticate against the privacyIDEA RADIUS server and not against the old deprecated RADIUS server.

Starting with privacyIDEA 2.11 the *passthru* policy was enhanced. You can define a system wide RADIUS server. Then you can create a *authentication* policy with the passthru action pointing to this RADIUS server. This means that - as long as a user trying to authenticate - has not token assigned, all authentication request with this very username and the password are forwarded to this RADIUS server.

As soon as you enroll a new token for this user in privacyIDEA the user will authenticate with this very token within privacyIDEA an his authentication request will not be forwarded anymore.

As soon as all users have a new token within privacyIDEA, you can switch of the old RADIUS server.

For strategies on enrolling token see *What are possible rollout strategies?*.

1.16.7 Setup translation

The translation is performed using grunt. To setup the translation environment do:

```
npm update -g npm
# install grunt cli in system
sudo npm install -g grunt-cli

# install grunt in project directory
npm install grunt --save-dev
# Install grunt gettext plugin
npm install grunt-angular-gettext --save-dev
```

This will create a subdirectory *node_modules*.

To simply run the German translation do:

```
make translate
```

If you want to add a new language like Spanish do:

```
cd po
msginit -l es
cd ..
grunt nggettext_extract
msgmerge po/es.po po/template.pot > po/tmp.po; mv po/tmp.po po/es.po
```

Now you can start translating with your preferred tool:

```
poedit po/es.po
```

Finally you can add the translation to the javascript translation file `privacyidea/static/components/translation/translations.js`:

```
grunt nggettext_compile
```

Note: Please ask to add this translation to the Make directive *translation* or issue a pull request.

1.16.8 How can I setup HA (High Availability) with privacyIDEA?

privacyIDEA does not track any state internally. All information is kept in the database. Thus you can configure several privacyIDEA instances against one DBMS[1] and have the DBMS do the high availability.

Note: The passwords and OTP key material in the database is encrypted using the *encKey*. Thus it is possible to put the database onto a DBMS that is controlled by another database administrator in another department.

HA setups

When running HA you need to assure to configure the *pi.cfg* file on all privacyIDEA instances accordingly. You might need to adapt the `SQLALCHEMY_DATABASE_URI` accordingly.

Be sure to set the same `SECRET_KEY` and `PI_PEPPER` on all instances.

Then you need to provide the same encryption key (file *encKey*) and the same audit signing keys on all instances.

Using one central DBMS

If you already have a high available, redundant DBMS - like MariaDB Galera Cluster - which might even be addressable via one cluster IP address the configuration is fairly simple. In such a case you can configure the same `SQLALCHEMY_DATABASE_URI` on all instances.

Using MySQL master-master-replication

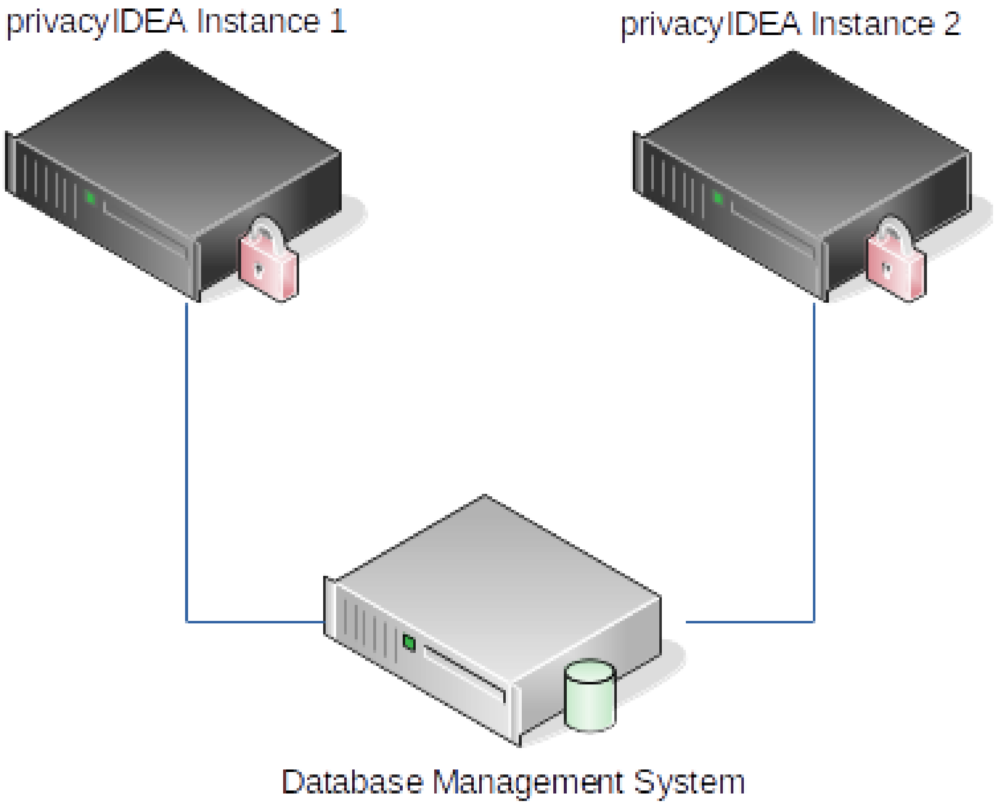
If you have no DBMS or might want to use a dedicated database server for privacyIDEA, you can setup one MySQL server per privacyIDEA instance and configure the MySQL servers to run in a master-master-replication.

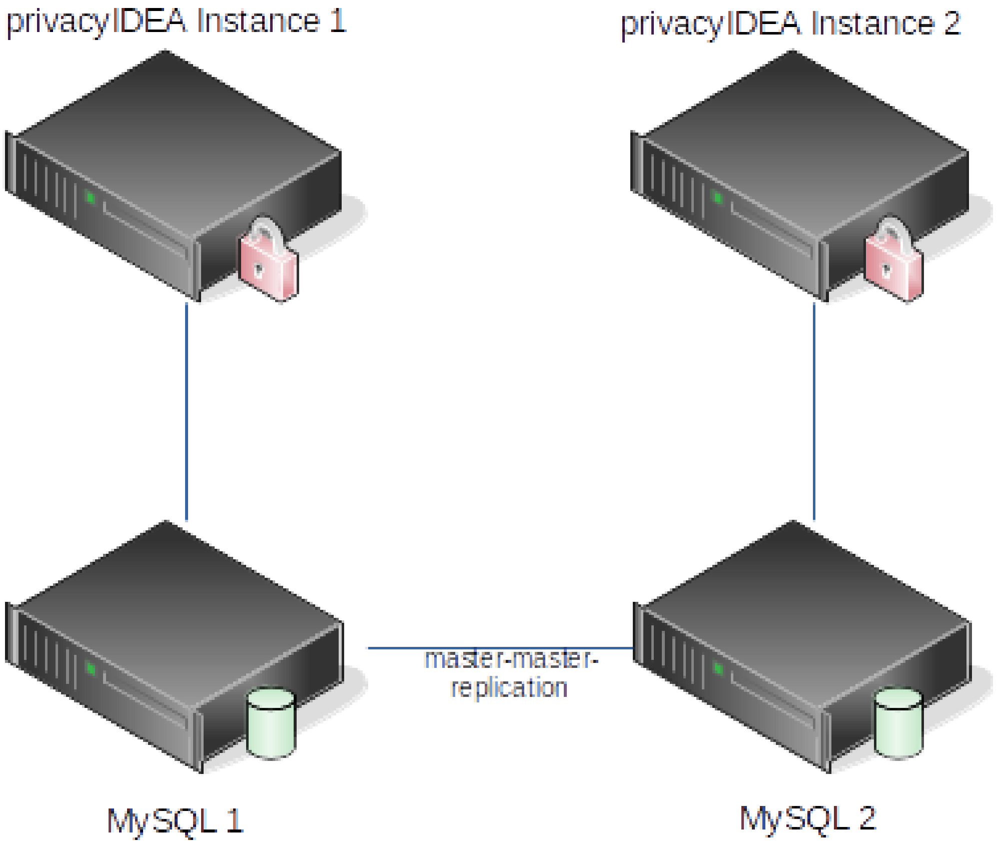
Note: The master-master-replication only works with two MySQL servers.

There are some good howtos out there like[2].

[1] Database management system

[2] https://www.digitalocean.com/community/tutorials/how-to-set-up-mysql-master-master-replication.


privacyIDEA Instance 1
privacyIDEA Instance 2
Database Management System


privacyIDEA Instance 1
privacyIDEA Instance 2
master-master-replication
MySQL 1
MySQL 2

1.16.9 MySQL database connect string

You can use the python package `MySQL-python` or `PyMySQL`.

`PyMySQL` is a pure python implementation. `MySQL-python` is a wrapper for a C implementation. I.e. when installing `MySQL-python` your python virtualenv, you also need to install packages like *python-dev* and *libmysqlclient-dev*.

Depending on whether you are using `MySQL-python` or `PyMySQL` you need to specify different connect strings in `SQLALCHEMY_DATABASE_URI`.

MySQL-python

connect string: `mysql://u:p@host/db`

Installation

Install a package *libmysqlclient-dev* from your distribution. The name may vary depending on which distritubtion you are running:

```
pip install MySQL-python
```

PyMySQL

connect string: `pymysql://u:p@host/db`

Installation

Install in your virtualenv:

```
pip install pymysql-sa
pip install PyMySQL
```

1.16.10 Are there shortcuts to use the Web UI?

I do not like using the mouse. Are there hotkeys or shortcuts to use the Web UI?

With version 2.6 we started to add hotkeys to certain functions. You can use '?' to get a list of the available hotkeys in the current window.

E.g. you can use `alt-e` to go to the *Enroll Token* Dialog and `alt-r` to actually enroll the token.

For any further ideas about shortcuts/hotkeys please drop us a note at github or the google group.

1.16.11 How to copy a resolver definition?

Creating a user resolver can be a time consuming task. Especially an LDAP resolver needs many parameters to be entered. Sometimes you need to create a second resolver, that looks rather the same like the first resolver. So copying or duplicating this resolver would be great.

You can create a similar second resolver by editing the exiting resolver and entering a new resolver name. This will save this modified resolver definition under this new name. Thus you have a resolver with the old name and another one with the new name.

1.16.12 Cryptographic considerations of privacyIDEA

Encryption keys

The encryption key is a set of 3 256bit AES keys. Usually this key is located in a 96 byte long file "enckey" specified by *PI_ENCFILE* in *The Config File*. The encryption key can be encrypted with a password.

The three encryption keys are used to encrypt

- data like the OTP seeds and secret keys stored in the *Token* table,
- password of resolvers to connect to LDAP/AD or SQL (stored in the *ResolverConfig* table)
- and optional additional values.

OTP seeds and passwords are needed in clear text to calculate OTP values or to connect to user stores. So these values need to be stored in a decryptable way.

Token Hash Algorithms

OTP values according to HOTP and TOTP can be calculated using SHA1, SHA2-256 and SHA2-512.

PIN Hashing

Token PINs are managed by privacyIDEA as the first of the two factors. Each token has its own token PIN. The token PIN is hashed with a seed with SHA2-256 and stored in the *Token* database table.

This PIN hashing is performed in *lib.crypto:hash*.

Administrator Passwords

privacyIDEA can manage internal administrators using *The pi-manage Script*. Internal administrators are stored in the database table *Admin*.

The password is stored using a PBKDF with SHA512 with 10023 rounds. The hash is salted and peppered. While the salt is stored in the *Admin* table created randomly for each admin password the pepper is unique for one privacyIDEA installation and stored in the pi.cfg file.

This way a database administrator is not able to inject rogue password hashes.

The admin password hashing is performed in *lib.crypto:hash_with_pepper*.

Audit Signing

The audit log is digitally signed. (see *Audit* and *The Config File*).

The audit log can be handled by different modules. privacyIDEA comes with an SQL Audit Module.

For signing the audit log the SQL Audit Module uses the RSA keys specified with the values *PI_AUDIT_KEY_PUBLIC* and *PI_AUDIT_KEY_PRIVATE* in *The Config File*.

By default the installer generates 2048bit RSA keys.

The audit signing is performed in *lib.crypto:Sign.sign* using SHA2-256 as hash function.

1.16.13 Policies

How to disable policies?

I create an evil admin policy and locked myself out. How can I disable a policy?

You can use the *pi-manage* command line tool to list, enable and disable policies. See

> pi-manage policy -h

How do policies work anyway?

Policies are just a set of definitions. These definitions are ment to modify the way privacyIDEA reacts on requests. Different policies have different **scopes** where they act.

admin policies define, what an administrator is allowed to do. These policies influence endpoints like `/token`, `/realm` and all other endpoints, which are used to configure the system. (see *Admin policies*)

user policies define, how the system reacts if a user is managing his own tokens. (see *User Policies*)

authentication and *authorization* policies influence the */validate/* endpoint (*Validate endpoints*).

The *Authentication policies* define if an authentication request would be successful at all. So it defines how to really check the authentication request. E.g. this is done by defining if the user has to add a specific OTP PIN or his LDAP password (see *otppin*).

The *Authorization policies* decide, if a user, who would authentication successfully is *allowed* to issue this request. I.e. a user may present the right credentials, but he is not allowed to login from a specific IP address or with a not secure token type (see *tokentype*).

How is this technically achieved?

At the beginning of a request the complete policy set is read from the database into a policy object, which is a singleton of PolicyClass (see *Policy Module*).

The logical part is performed by policy decorators. The decorators modify the behaviour of the above mentioned endpoints.

Each policy has its own decorator. The decorator can be used on different functions, methods, endpoints. The decorators are implemented in api/lib/prepolicy.py and api/lib/postpolicy.py.

PrePolicy decorators are executed at the beginning of a request, PostPolicy decoratros at the end of the request.

A policy decorator uses one of the methods get_action_value or get_policies.

get_policies is used to determine boolean actions like *passOnNoToken*.

get_action_value is used to get the defined value of non-boolean policies like *otppin*.

All policies can depend on IP address, user and time. So these values are taken into account by the decorator when determining the defined policy.

Note: Each decorator represents one policy and defines its own logic i.e. checking filtering for IP address and fetching the necessary policy sets from the policy object.

1.16.14 Performance considerations

You can test performace using the apache bench from the apache utils. Creating a simple pass token for a user, eases the performance testing.

Then you can run

> ab -l -n 200 -c 8 -s 30 'https://localhost/validate/check?user=yourUser&pass=yourPassword'

The performance depends on several aspects like the connection speed to your database and the connection speed to your user stores.

Processes

You should run several processes and threads. You might start with the number of processes equal to the number of your CPU cores. But you should evaluate, which is the best number of processes to get the highest performance.

Config caching

Starting with privacyIDEA 2.15 privacyIDEA uses a Cache per instance and process to cache system configuration, resolver, realm and policies.

As the configuration might have been changed in the database by another process or another instance, privacyIDEA compares a cache timestamp with the timestamp in the database. Thus at the beginning of the request privacyIDEA reads the timestamp from the database.

You can configure how often the timestamp should be read using the pi.cfg variable `PI_CHECK_RELOAD_CONFIG`. You can set this to seconds. If you use this config value to set values higher than 0, you will improve your performance. But: other processes or instances will learn later about configuration changes which might lead to unexpected behaviour.

Logging

Choose a logging level like WARNING or ERROR. Setting the logging level to INFO or DEBUG will produce much log output and lead to a decrease in performance.

Response

You can strip the authentication response, to get a slight increase in performace, using the policy `no_details_on_success`.

Clean configuration

Remove unused resolvers and policies. Have a realm with several resolvers is a bit slower than one realm with one resolver. Finding the user in the first resolver is faster than in the last resolver. Although e.g. the LDAP resolver utilizes caching.

Also see *What happens in the tokenview?*.

1.16.15 What happens in the tokenview?

A question which comes up often is why you can not view hundrets of tokens in the tokenview. Well - you are doing - you are just paging through the list ;-)

Ok, here it what happens in the tokenview.

The tokenview fetches a slice of the tokens from the token database. So, if you configure the tokenview to display 15 tokens, only 15 tokens will be fetched using the `LIMIT` and `OFFSET` mechanisms of SQL.

But what really influences the performance is the user resolver part. privacyIDEA does not store username, givenname or surname of the token owner. The token table only contains a "pointer" to the user object in the userstore. This pointer consists of the userresolver ID and the user ID in this resolver. This is useful, since the username or the surname of the user may change. At least in Germany the givenname only changes in very rare cases.

This means that privacyIDEA needs to contact the userstore, to resolve the user ID to a username and a surname, givenname. Now you know that you will create 100 LDAP requests, if you choose to display 100 tokens on one page.

Although we are doing some LDAP caching, this will not help with new pages.

We very much recommend using the search capabilities of the tokenview.

1.16.16 How to mitigate brute force and lock tokens

For each failed authentication attempt privacyIDEA will increase a fail counter of a token. If the maximum allowed fail counter is reached, authentication with this token is not possible anymore. The token get a timestamp mark, when the maximum fail counter was reached. Starting with version 2.20 the administrator can define a timeout in minutes. If this timestamp is more than these specified minutes ago, an authentication attempt with a correct PIN will reset the fail counter. See *Automatically clearing Failcounter*.

The failcounter avoids brute force attacks which guess passwords or OTP values. Choose a failcounter clearing timeout, which is not too long. Otherwise brute force would also lock the token of the user forever.

Another possibility to mitigate brute force is to define an `authorization` policy with the action `auth_max_fail`. This will check, if there are too many failed authentication requests during the specified time period. If there are, even a successful authentication will fail. This technique uses the audit log, to search for failed authentication requests. See *auth_max_fail*.

If you are missing any information or descriptions file an issue at github (which would be the preferred way), drop a note to info(@)privacyidea.org or go to the Community Forum.

This will help us a lot to improve documentation to your needs.

Thanks a lot!

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

HTTP ROUTING TABLE

INDEX