
privacyIDEA Documentation

Release 2.3

Cornelius Kölbel

June 23, 2015

1	Table of Contents	3
1.1	Overview	3
1.2	Installation	3
1.3	First Steps	12
1.4	Configuration	17
1.5	Tokenview	47
1.6	Userview	52
1.7	Policies	56
1.8	Audit	72
1.9	Client machines	72
1.10	Application Plugins	75
1.11	Tools	79
1.12	Import	80
1.13	Code Documentation	82
1.14	Frequently Asked Questions	173
2	Indices and tables	175
	HTTP Routing Table	177
	Python Module Index	179

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*)

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 14.04 LTS and
- ready made [Debian Packages](#) for Debian Wheezy.

If you want to upgrade from a privacyIDEA 1.5 installation please read [Upgrading](#).

1.2.1 Python Package Index

You can install privacyidea on usually any Linux distribution in a python virtual environment 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
```

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.py create_enckey  
pi-manage.py create_signkey
```

Create the database and the first administrator:

```
pi-manage.py createdb  
pi-manage.py admin add admin admin@localhost
```

Now you can run the server for your first test:

```
pi-manage.py runserver
```

Depending on the database you want to use, you may have to install additional packages.

1.2.2 Ubuntu Packages

There are ready made debian packages for Ubuntu 14.04 LTS. These are available in a public ppa repository ¹, so that the installation will automatically resolve all dependencies. Install it like this:

```
add-apt-repository ppa:privacyidea/privacyidea  
apt-get update  
apt-get install python-privacyidea privacyideadm
```

Optionally you can also install necessary configuration files to run privacyIDEA within the Nginx Webserver:

```
apt-get install privacyidea-nginx
```

Alternatively you can install privacyIDEA running in an Apache webserver:

```
apt-get install privacyidea-apache2
```

After installing in Nginx or Apache2 you only need to create your first administrator and you are done:

```
pi-manage.py admin add admin admin@localhost
```

Now you may proceed to *First Steps*.

¹ <https://launchpad.net/~privacyidea>

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 like privacyIDEA. To install this module run:

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

For further details see *FreeRADIUS Plugin*.

SimpleSAMLphp

Starting with 1.4 privacyIDEA also supports SAML via a plugin for simpleSAMLphp². The simpleSAMLphp service does not need to run on the same machine like the privacyIDEA server.

To install it on a Ubuntu 14.04 system please run:

```
apt-get install privacyidea-simplesamlphp
```

For further details see *simpleSAMLphp Plugin*.

PAM

privacyIDEA also comes with a PAM library to add two factor authentication to any Linux system. You can run one central privacyIDEA server and configure all other systems using the PAM library to authenticate against this privacyIDEA.

To install it on a Ubuntu 14.04 system please run:

```
apt-get install privacyidea-pam
```

For further details see *Pluggable Authentication Module*.

OTRS

OTRS is an important Open Source Ticket Request System. It is written in Perl and privacyIDEA provides an authentication plugin to authenticate at OTRS with two factors.

To install it on Ubuntu 14.04 please run:

```
apt-get install privacyidea-otrs
```

For further details and configuration see *OTRS*.

1.2.3 Debian Packages

You can install privacyIDEA on debian Wheezy either via the *Python Package Index* or with a ready made Wheezy package.

The available Wheezy package `privacyidea-venv_2.1~dev0_amd64.deb` contains a complete virtual environment with all necessary dependent modules. To install it run:

```
dpkg -i privacyidea-venv_2.1~dev0_amd64.deb
```

² <https://github.com/privacyidea/privacyidea/tree/master/authmodules/simpleSAMLphp>

This will install privacyIDEA into a virtual environment at `/opt/privacyidea/privacyidea-venv`.

You can enter the virtual environment by:

```
source /opt/privacyidea/privacyidea-venv/bin/activate
```

Running privacyIDEA with Apache2 and MySQL

You need to create and fill the config directory `/etc/privacyidea` manually:

```
cp /opt/privacyidea/privacyidea-venv/etc/privacyidea/dictionary \
/etc/privacyidea/
```

Create a config `/etc/privacyidea/pi.cfg` like this:

```
# Your database
SQLALCHEMY_DATABASE_URI = 'mysql://pi:password@localhost/pi'
# This is used to encrypt the auth_token
SECRET_KEY = 'choose one'
# This is used to encrypt the admin passwords
PI_PEPPER = "choose one"
# 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_LOGFILE = '/var/log/privacyidea/privacyidea.log'
#CRITICAL = 50
#ERROR = 40
#WARNING = 30
#INFO = 20
#DEBUG = 10
PI_LOGLEVEL = 20
```

You need to create the above mentioned logging directory `/var/log/privacyidea`.

You need to create the above mentioned database with the corresponding user access:

```
mysql -u root -p -e "create database pi"
mysql -u root -p -e "grant all privileges on pi.* to 'pi'@'localhost' \
identified by 'password'"
```

With this config file in place you can create the database tables, the encryption key and the audit keys:

```
pi-manage.py createdb
pi-manage.py create_enckey
pi-manage.py create_audit_keys
```

Now you can create the first administrator:

```
pi-manage.py admin add administrator email@domain.tld
```

The system is set up. You now only need to configure the Apache2 webserver.

The Apache2 needs a wsgi script that could be located at `/etc/privacyidea/piapp.wsgi` and look like this:

```
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")
```

Finally you need to create a Apache2 configuration `/etc/apache2/sites-available/privacyidea.conf` which might look like this:

```
WSGIPythonHome /opt/privacyidea/privacyidea-venv
<VirtualHost _default_:443>
    ServerAdmin webmaster@localhost
    # You might want to change this
    ServerName localhost

    DocumentRoot /var/www
    <Directory />
        # For Apache 2.4 you need to set this:
        # Require all granted
        Options FollowSymLinks
        AllowOverride None
    </Directory>

    # We can run several instances on different paths with different configurations
    WSGIScriptAlias / /etc/privacyidea/piapp.wsgi
    #
    # The daemon is running as user 'privacyidea'
    # This user should have access to the encKey database encryption file
    WSGIDaemonProcess privacyidea processes=1 threads=15 display-name=%{GROUP} user=privacyidea
    WSGIProcessGroup privacyidea
    WSGIPassAuthorization On

    ErrorLog /var/log/apache2/error.log

    LogLevel warn
    LogFormat "%h %l %u %t %>s \"%m %U %H\" %b \"%{Referer}i\" \"%{User-agent}i\"" privacyIDEA
    CustomLog /var/log/apache2/ssl_access.log privacyIDEA

    # SSL Engine Switch:
    # Enable/Disable SSL for this virtual host.
    SSLEngine on

    # If both key and certificate are stored in the same file, only the
    # SSLCertificateFile directive is needed.
    SSLCertificateFile /etc/ssl/certs/privacyideaserver.pem
    SSLCertificateKeyFile /etc/ssl/private/privacyideaserver.key

    <FilesMatch "\.(cgi|shtml|phtml|php)$">
        SSLOptions +StdEnvVars
    </FilesMatch>
    <Directory /usr/lib/cgi-bin>
        SSLOptions +StdEnvVars
    </Directory>
    BrowserMatch ".*MSIE.*" \
        nokeepalive ssl-unclean-shutdown \
        downgrade-1.0 force-response-1.0

</VirtualHost>
```

The configuration assumes, a user `privacyidea`, which you need to create:

```
useradd -r -m privacyidea
```

The files in `/etc/privacyidea` and the logfiles in `/var/log/privacyidea/` should be restricted to this user.

1.2.4 Upgrading

Upgrade From privacyIDEA 2.x to 2.3

In 2.3 the priority of resolvers in realms was added.

You need to update the database models:

```
pi-manage.py db stamp 4f32a4e1bf33 -d path/to/migrations
pi-manage.py db upgrade -d path/to/migrations
```

Note: You need to specify the path to the migrations scripts. This could be `/usr/lib/privacyidea/migrations`.

Note: When upgrading with the Ubuntu LTS packages, the database update is performed automatically.

Upgrade From privacyIDEA 1.5

Warning: privacyIDEA 2.0 introduces many changes in database schema, so at least perform a database backup!

Stopping Your Server

Be sure to stop your privacyIDEA server.

Upgrade Software

To upgrade the code enter your python virtualenv and run:

```
pip install --upgrade privacyidea
```

Configuration

Read about the configuration in the *The Config File*.

You can use the old *enckey*, the old *signing keys* and the old *database uri*. The values can be found in your old ini-file as `privacyideaSecretFile`, `privacyideaAudit.key.private`, `privacyideaAudit.key.public` and `sqlalchemy.url`. Your new config file might look like this:

```
config_path = "/home/cornelius/tmp/pi20/etc/privacyidea/"
# This is your old database URI
# Note the three slashes!
SQLALCHEMY_DATABASE_URI = "sqlite:/// " + config_path + "token.sqlite"
# This is new!
SECRET_KEY = 't0p s3cr3t'
# This is new
#This is used to encrypt the admin passwords
```

```

PI_PEPPER = "Never know..."
# This is used to encrypt the token data and token passwords
# This is your old encryption key!
PI_ENCFEILE = config_path + 'enckey'
# These are your old signing keys
# This is used to sign the audit log
PI_AUDIT_KEY_PRIVATE = config_path + 'private.pem'
PI_AUDIT_KEY_PUBLIC = config_path + 'public.pem'

```

To verify the new configuration run:

```
pi-manage.py create_enckey
```

It should say, that the enckey already exists!

Migrate The Database

You need to upgrade the database to the new database schema:

```
pi-manage.py db upgrade -d lib/privacyidea/migrations
```

Note: In the Ubuntu package the migrations folder is located at `/usr/lib/privacyidea/migrations/`.

Create An Administrator

With privacyIDEA 2.0 the administrators are stored in the database. The password of the administrator is salted and also peppered, to avoid having a database administrator slip in a rogue password.

You need to create new administrator accounts:

```
pi-manage.py addadmin <email-address> <admin-name>
```

Start The Server

Run the server:

```
pi-manage.py runserver
```

or add it to your Apache or Nginx configuration.

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`.

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_LOGFILE = '....'
# PI_LOGLEVEL = 20
# PI_INIT_CHECK_HOOK = 'your.module.function'
```

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.

If you are using a config file other than `/etc/privacyidea/pi.cfg` you need to set the environment variable:

```
export PRIVACYIDEA_CONFIGFILE=/your/config/file
```

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](#).

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 like this:

```
PI_LOGCONFIG = "/etc/privacyidea/logging.cfg"
```

Such a configuration could look like this:

```
[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=NOTSET
handlers=file
```

The file structure follows ³ and can be used to define additional handlers like logging errors to email addresses.

Note: In this example a mail handler is defined, that will send emails to certain email Addresses, if an ERROR occurs.

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.

³ <https://docs.python.org/2/library/logging.config.html#configuration-file-format>

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

1.3 First Steps

You installed privacyIDEA successfully according to *Installation* and created an administrator using the command `pi-manage.py admin` as e.g. described in *Ubuntu Packages*.

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 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.py 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 `SUPERUSER_REALM` in `inifile_superuser` 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 may 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*)

1.3.2 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`.

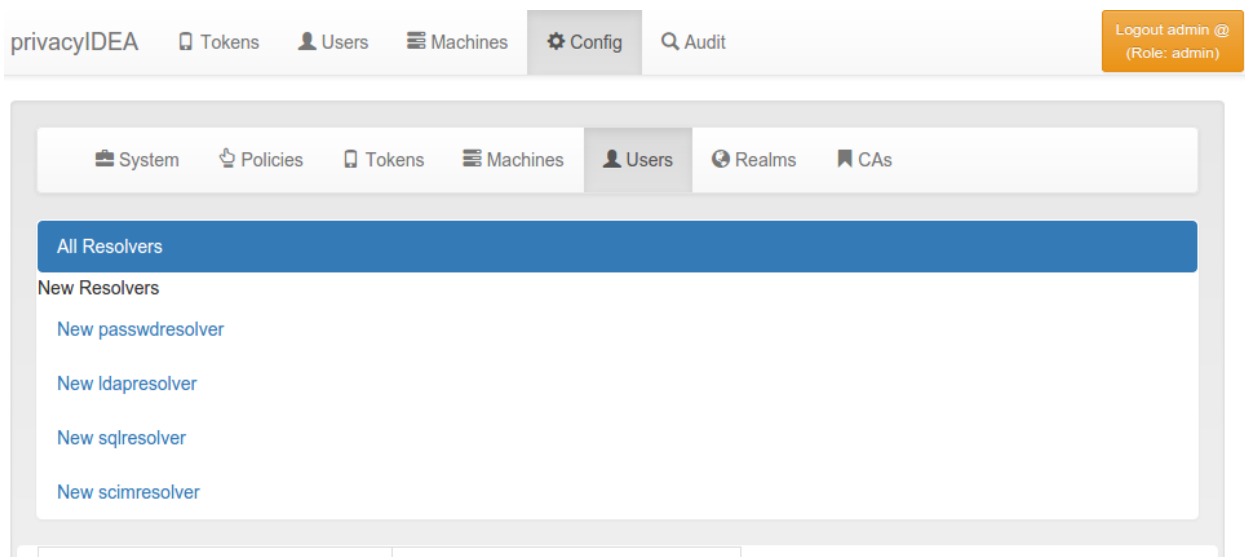


Fig. 1.1: Create the first `UserIdResolver`

- Choose *New passwdresolver* and
- Enter the name “myusers”.
- Save it.

You just created your first connection to a user source.

Create a Realm

User sources are grouped together 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`.

The screenshot shows the privacyIDEA web interface. The top navigation bar includes 'privacyIDEA', 'Tokens', 'Users', 'Machines', 'Config', and 'Audit'. A 'Logout admin @ (Role: admin)' button is in the top right. The main content area has a sub-navigation bar with 'System', 'Policies', 'Tokens', 'Machines', 'Users', 'Realms', and 'CAs'. The 'Users' tab is active, showing 'All Resolvers' and 'New Resolvers' links. Below these are links for 'New passwdresolver', 'New ldapresolver', 'New sqlresolver', and 'New scimresolver'. The 'Edit Passwd Resolver myusers' form is displayed with two input fields: 'Resolver name' containing 'myusers' and 'File name' containing '/etc/passwd'. A 'Save resolver' button is at the bottom right of the form.

Fig. 1.2: Create the first UserIdResolver

The screenshot shows the privacyIDEA web interface with the 'Realms' tab active. It displays 'All Realms' and a 'Clear default realm' link. Below is a table for creating a new realm:

Default	Realm name	resolvers	
<input type="checkbox"/>	<input type="text" value="realm1"/>	<input checked="" type="checkbox"/> myusers 1 (passwdresolver)	<input type="button" value="Create realm"/>

Fig. 1.3: Create the first Realm

privacyIDEA
Tokens
Users
Machines
Config
Audit
Logout admin @
(Role: admin)

All users

Select Realm
realm1

Quick links
[Edit realms](#)

total users: 52

First Previous 1 2 3 4 Next Last

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
root							118

Fig. 1.4: The users from /etc/passwd

Congratulation! You created your first realm.

You are now ready to enroll a token to a user. Read [Enrolling your first token](#).

1.3.3 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*

The screenshot shows the 'Enroll a new token' dialog in the privacyIDEA web interface. The top navigation bar includes 'Tokens', 'Users', 'Machines', 'Config', and 'Audit' tabs. A sidebar on the left shows 'All tokens' and 'Enroll Token' button. The main content area is titled 'Enroll a new token' and contains a dropdown for 'HOTP: event based One Time Passwords'. Below this, there is a checkbox for 'Generate OTP Key on the Server' and a text box for 'OTP length' (6). The 'Hash algorithm' is set to 'sha1'. The 'Assign token to user' section includes a dropdown for 'Realm' (realm1) and a text box for 'Username' ([0] root (root)). The 'PIN' section has two text boxes, one for the PIN (test) and one for the OTP value. An 'Enroll Token' button is located at the bottom right of the dialog.

Fig. 1.5: 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”.

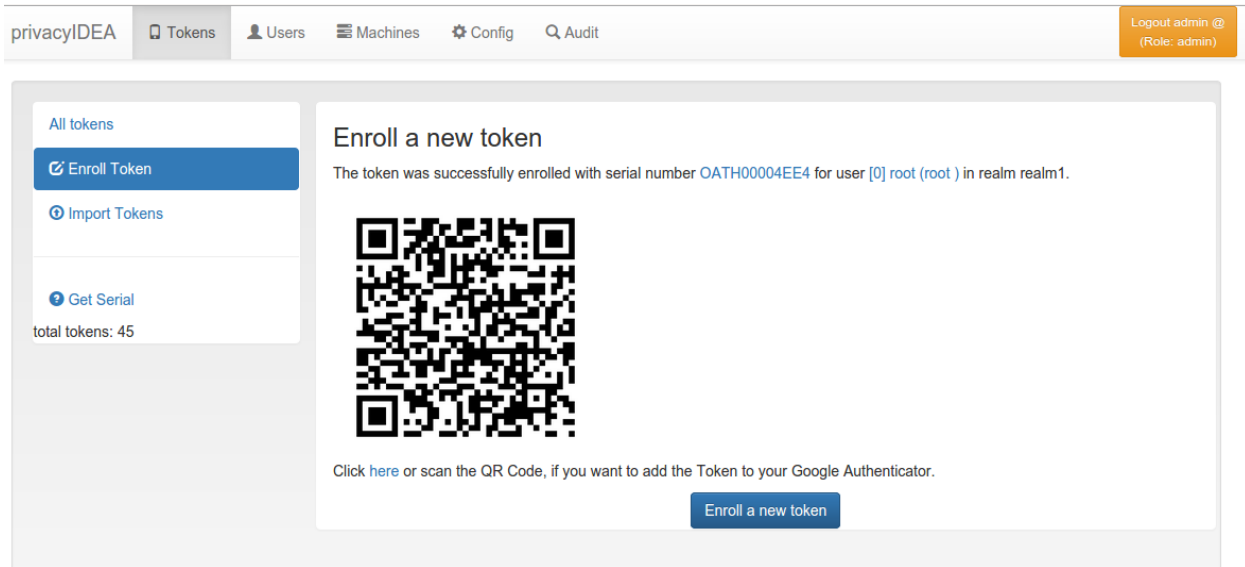


Fig. 1.6: Enrollment Success

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

- 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 chapter [Configuration](#) to get a deeper insight in the configuration possibilities.

1.4 Configuration

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

It also contains a shortcut to the policy tab (see [Policies](#)).

1.4.1 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

privacyIDEA
Tokens
Users
Machines
Config
Audit
Logout admin @ (Role: admin)

All tokens
Token OATH00004EE4
Enroll Token
Import Tokens
Lost Token
Get Serial
total tokens: 45

Token details for OATH00004EE4

[View token in Audit log](#)

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	<ul style="list-style-type: none"> count_auth: 1 count_auth_success: 1 hashlib: sha1 	Edit
Realms	<ul style="list-style-type: none"> realm1 	Edit

[Resync Token](#)

[Set PIN](#)

[Test token](#)
matching 1 tokens

Assigned User

Username	root	Unassign User
----------	------	-------------------------------

Fig. 1.7: Test the Token

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-pwresolver-user`.

Create a flat file like this:

```
privacyidea-create-pwresolver-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 ⁴.

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 ⁵.

Example:

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

⁴ <http://blogs.technet.com/b/ad/archive/2009/07/06/referral-chasing.aspx>

⁵ <https://github.com/cannatag/ldap3/blob/master/docs/manual/source/servers.rst#server-pool>

pivacyIDEA
Token View
User View
Config
Audit
Logout admin @ (Role: admin)

System
Policies
Tokens
Machine Resolvers
User Resolvers
User Realms

All Resolvers
New Resolvers
New passwdresolver
New ldapresolver
New sqlresolver

Create a new LDAP Resolver

Resolver name resolver1

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

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

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

Bind Password topsecret
Bind Type Simple

Timeout (seconds) 5
Size Limit 500

Preset OpenLDAP
Preset Active Directory

☐ No anonymous referral chasing

Loginname Attribute sAMAccountName

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

User Filter (&(sAMAccountName=%s)(objectClass=person))

Attribute mapping { "username": "sAMAccountName", "phone": "telephoneNumber", "mobile": "mobile", "

UID Type DN

Test LDAP Resolver
Save resolver

Fig. 1.8: LDAP resolver configuration

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 `DOMAINNAMEusername`.

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

The `searchfilter` and the `userfilter` are used for forward and backward search the object in LDAP.

The `searchfilter` is used to list all possible users, that can be used in this resolver.

The `userfilter` is used to find the LDAP object for a given loginname. This is why the `userfilter` contains the python string replacement parameter `%s`, which will be filled with the given loginname to find the LDAP object.

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

- username,
- phone,
- mobile,
- email,
- surname,
- givenname.

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`.

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.

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 privacyIDEA attribute. The known attributes are:

pivacyIDEA
Token View
User View
Config
Audit
Logout admin @ (Role: admin)

System
Policies
Tokens
Machine Resolvers
User Resolvers
User Realms

All Resolvers
New Resolvers
New passwdresolver
New ldapresolver
New sqlresolver

Create a new SQL Resolver

Resolver name

Driver

Server **Port**

Database

User

Password

Wordpress
OTRS
Tine 2.0
Owncloud

Table **Limit**

Mapping

Where statement

Database Encoding

Connection Parameters

Test SQL Resolver
Save Resolver

Fig. 1.9: SQL resolver configuration

- `userid`,
- `username`,
- `phone`,
- `mobile`,
- `email`,
- `givenname`,
- `surname`.

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

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 OSIAM⁶, the “Open Source Identity & 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.

Userinformation is then retrieved from the resource server.

The available attributes for the `Attribute` mapping are:

- `username`,
- `givenname`,
- `surname`,
- `phone`,
- `mobile`,
- `email`.

1.4.2 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

⁶ <http://www.osiam.org>

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.

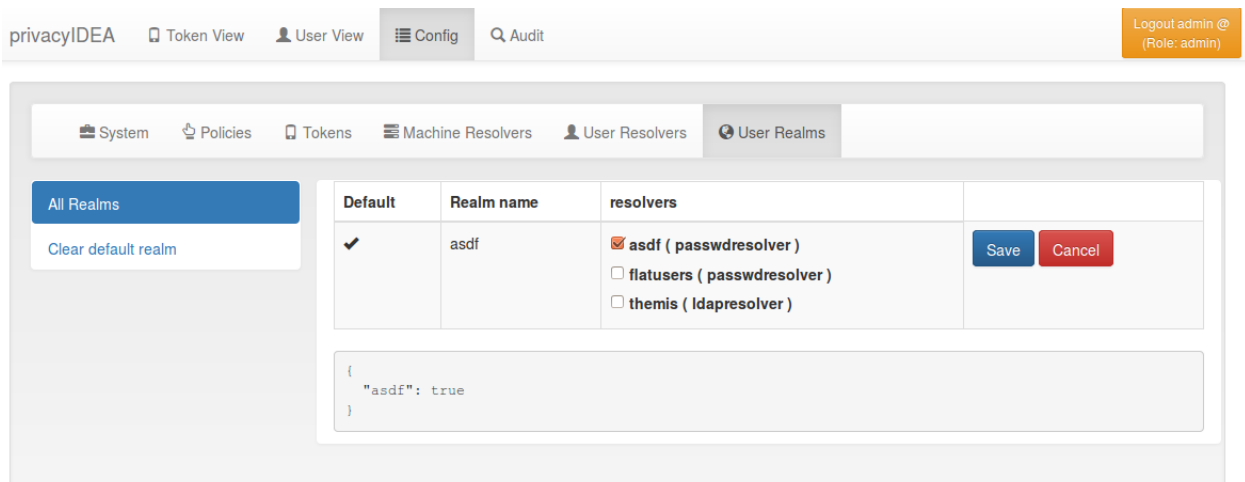


Fig. 1.10: *Edit a realm*

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:

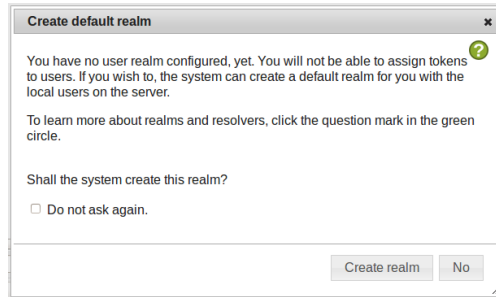
A user “administrator” is located in a resolver “users” which contains all Active Directory users. And the “administrator” is located in a resolver “admins”, which contains all users in the Security Group “Domain Admins” from the very same domain. Both resolvers are in the realm “AD”, “admins” with priority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



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.

1.4.3 System Config

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

Settings

`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.

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

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

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.

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

`Auto resync` defines if the system should try to resync a token if a user provides a wrong OTP value. If checked, the system remembers the OTP value and if during `auto resync timeout` the user tries to authenticate again with the next, successive OTP value, the system tries to resync this token with the two given OTP values.

`Pass on user not found` let the system return a successful authentication response if the authenticating user does not exist in the system.

pivacyIDEA
Token View
User View
Config
Audit
Logout admin @ (Role: admin)

System
Policies
Tokens
Machine Resolvers
User Resolvers
User Realms

☐ Use @ sign to split the username and the realm.
☐ Increase the failcounter if the wrong PIN was entered.
☐ Prepend the PIN in front of the OTP value . Otherwise it will be post pended.
☐ Include SAML attributes in the authentication response.
☐ Automatic resync during authentication
Auto resync timeout 300
☐ Pass, if the user does not exist
☐ Pass, if the user has no token
Override Authorization Clients 127.0.0.1, 10.0.0.8
These client IP addresses or subnets are allowed to masquerade as another client.

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

☐ Display Help Button on Login Screen.
☐ Display Realm Box on Login Screen.

Save System Config

Fig. 1.11: The system config

Warning: Use with care and only if you know what you are doing!

`Pass on user no token` let the system return a successful authentication response if the authenticating user exists in the system but has no token assigned.

Warning: Use with care and only if you know what you are doing! Since the user could remove all his tokens in selfservice and then have free rides forever.

`Override Authentication client` is important with client specific policies (see [Policies](#)) and RADIUS servers. In case of RADIUS the authenticating client for the privacyIDEA system will always be the RADIUS serve, 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. This field takes a comma seperated list of IP addresses.

`maximum concurrent OCRA challenges` defines how many OCRA requests for a single OCRA token are allowed to be active simultaneously. **(TODO):** Not migrated, yet.

`OCRA challenge timeout` defines how many seconds an OCRA challenge is kept active. The response must be sent within this timeout. **(TODO):** Not migrated, yet.

Token default settings

Misc settings

`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 lateron, you need to change this at the tokeninfo dialog.

`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 increas till this maxfailcount. Even if more failed authentication request occur, the failcounter will not increase anymore.

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

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

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

`DefaultChallengeValidityTime` is the timeout for a challenge response authentication.

OCRA settings

`default OCRA suite` is the OCRA suite that is set for an OCRA token during enrollment if no OCRA suite is specified. **(TODO):** Not migrated, yet.

`default QR suite` is the OCRA suite that is set for a QR token during enrollment if no OCRA suite is specified. **(TODO):** Not migrated, yet.

GUI settings

The login window of the WebUI may display a dropdown box with all realms. You might hide this dropdown box, if you do not want to tell the world which realms are defined on your system. If you check `display realm select box` the list of all realms including the special realm *admin* for the administrators from the superuser file will be displayed in the login form.

1.4.4 Tokens

Supported Tokens

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

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` 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` token. These tokens are initialized at the factory and you get a seed file, that you need to import to privacyIDEA.

Smartphone Apps

Google Authenticator. The Google Authenticator is working well in `hotp_token` and `totp_token` 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` 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.

Supported Tokentypes

At the moment the following tokentypes are supported:

- `hotp_token` - event base One Time Password tokens based on [RFC4225](#).
- `totp_token` - time based One Time Password tokens based on [RFC6238](#).
- `mOTP` - time based One Time Password tokens for mobile phones based on an a [public Algorithm](#).
- `password` - A password token used for *losttoken* scenario.
- *Registration* - A special token type used for enrollment scenarios (see *Registration Code*).
- `Simple Pass` - A token that only consists of the Token PIN.
- *Certificates* - A token that represents a client certificate.
- *SSH Keys* - An SSH public key that can be managed and used in conjunction with the *Client machines* concept.
- *Remote* - A virtual token that forwards the authentication request to another privacyIDEA server.
- *RADIUS* - A virtual token that forwards the authentication request to a RADIUS server.
- *SMS* - A token that sends the OTP value to the mobile phone of the user.
- `email_token` - A token that sends the OTP value to the EMail address of the user.
- *Yubico* - A Yubikey hardware that authenticates against the Yubico Cloud service.
- *Yubikey* - A Yubikey hardware initialized in the AES mode, that authenticates against privacyIDEA.
- `Daplug` - A hardware OTP token similar to the Yubikey.

The Tokentypes:

Certificates

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.

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.

Enroll a new token

Certificate: Enroll an x509 Certificate Token. ▼

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

Token data

Generate Request **Upload Request** Upload Certificate

CA Connector

myCA ▼

Certificate Signing Request (PEM)

Paste the Certificate Signing Request

Fig. 1.12: Upload a certificate signing request

Enroll a new token

Certificate: Enroll an x509 Certificate Token. ▼

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

Token data

Generate Request Upload Request Upload Certificate

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.

Open new tab to create certificate request

Fig. 1.13: Generate a certificate signing request

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



privacyIDEA Certificate Request

CA Connector: myCA

Key strength 2048 (High Grade) ▼

Senden

Fig. 1.14: Download or install the client certificate

Afterwards the user can install the certificate into the browser.

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.

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.

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

The screenshot shows a web form titled "Enroll a new token". At the top, there is a dropdown menu with the selected option "RADIUS: Forward authentication request to a RADIUS server". Below this, a text block explains: "The RADIUS token forwards the authentication request to another RADIUS server. You can choose if the PIN should be stripped and checked locally." Under the heading "Token data", there is a checkbox labeled "Check the PIN locally" which is currently unchecked. The next section, "RADIUS Server", contains a text input field with the placeholder text "your.radius.server:1812" and a note below it stating "The RADIUS server may include the port number." Following this is the "RADIUS User" section with an empty text input field. The final section is "RADIUS Secret", also with an empty text input field. At the bottom left of the form, there are small navigation icons: a left arrow, a right arrow, and a refresh icon.

Fig. 1.15: *Enroll a RADIUS token*

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 privacyIDEA 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

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. 1.16: Enroll a Remote token

and mangle the password.

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 token on *localhost* thus adding a second user to the same token.

SMS

The token type *sms* sends the OTP value via an SMS service. You can configure the SMS service in *SMS OTP Token*.

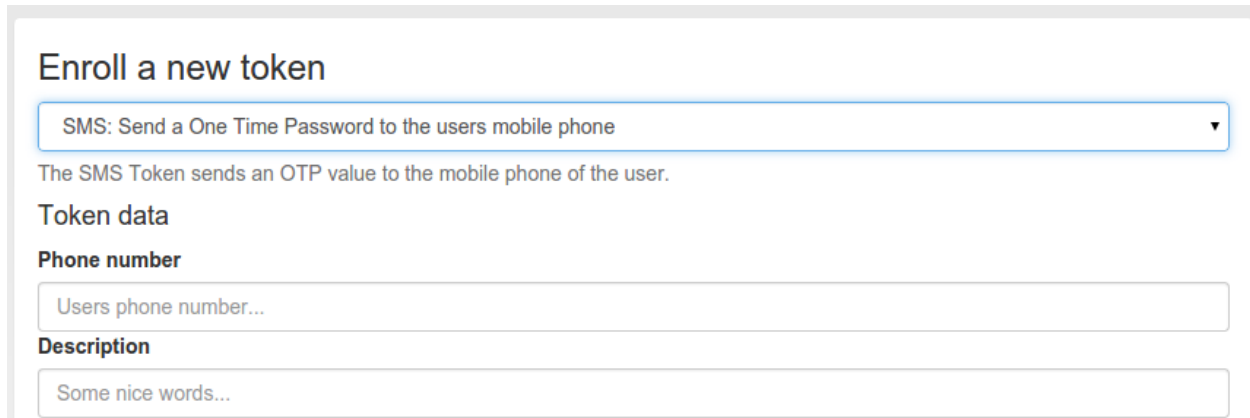


Fig. 1.17: Enroll an SMS token

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.

For a more detailed insight see the code documentation *SMS 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 *Client machines* concept.

A user or the administrator can upload the public SSH key and assign to a user.

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.

Yubico

The token type *yubico* authenticates against the Yubico Cloud mode. You need to configure this at *Yubico Cloud mode*.

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.

All tokens

Enroll Token

Import Tokens

Get Serial

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
AAAAB3NzaC1yc2EAAAABPszIM/dwBH4A6yKcSDv5+DqvYsZjYMwdNj9ldxaidYo4ohohgvpPGjamGsXKQlaDmeOREpH2Fc/0eZWG5vAzz
sw7/qCp2ydnZISLIJ6sdjDoNybhH4iq8hZyGtAeHN7fESc1MGkJ/eTkxD2v4lFP5MbGJOlbmy+JR56TuqKo/de9AnytvtqrMTD3+Y5ac4aZ7kSs
ufbOvaV1FI2+1wvJ2D64xeJXE90naGJzTFVleqQ330jw== corny@az.local
```

Description

corny@az.local

Assign token to user

Realm

privacyidea-demo.intranet

Username

start typing a username

Fig. 1.18: Enroll an SSH key token

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...

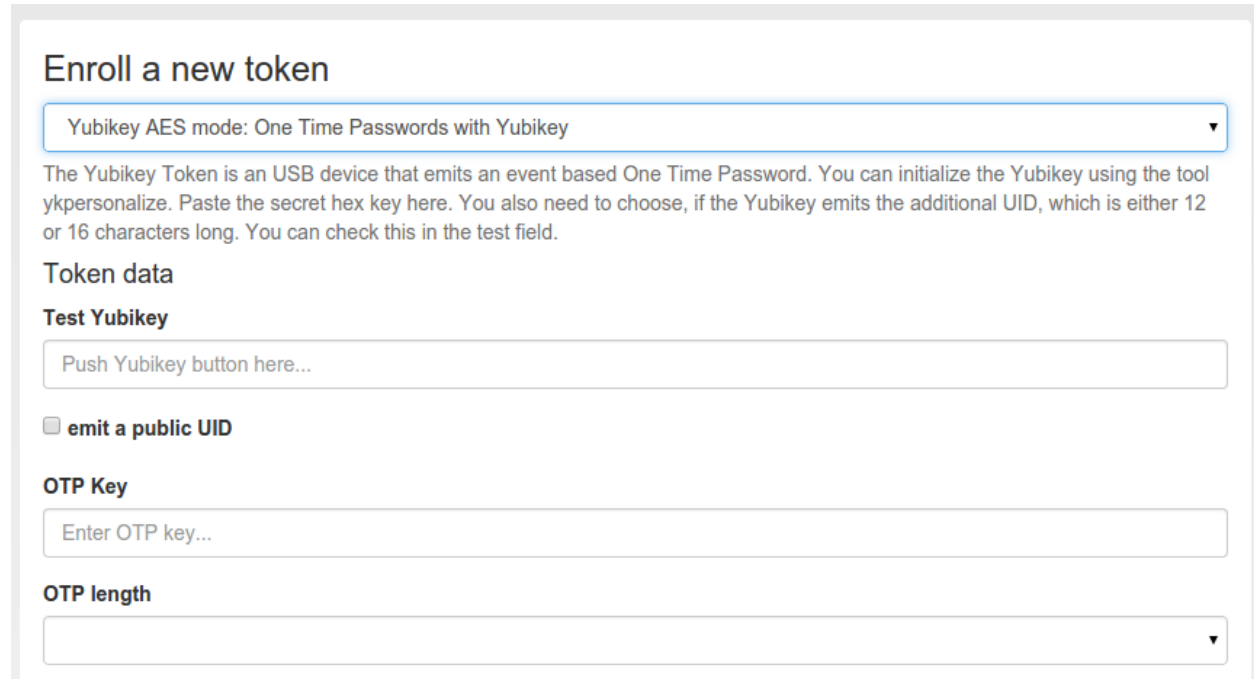
Assign token to user

Fig. 1.19: Enroll a Yubico token

Yubikey

The Yubikey is initialized with privacyIDEA and works in Yubicos own AES mode. It outputs a 44 digit OTP value. 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.



The screenshot shows a web interface titled "Enroll a new token". At the top, there is a dropdown menu with the selected option "Yubikey AES mode: One Time Passwords with Yubikey". Below this, a text block explains: "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." Below the text is a section titled "Token data" containing a "Test Yubikey" button with the label "Push Yubikey button here...". There is a checkbox labeled "emit a public UID" which is currently unchecked. Below this is an "OTP Key" section with a text input field labeled "Enter OTP key...". At the bottom is an "OTP length" section with a dropdown menu.

Fig. 1.20: *Enroll a Yubikey AES mode token*

You can use this dialog to enroll a Yubikey AES mode token, if you have initialized the yubikey with the external *ykpersonalize* tool.

Note: However, 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
```

Token configuration

Each token type can provide its own configuration dialog.

In this configuration dialog you can define default values for these token types.

Email OTP Token

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:

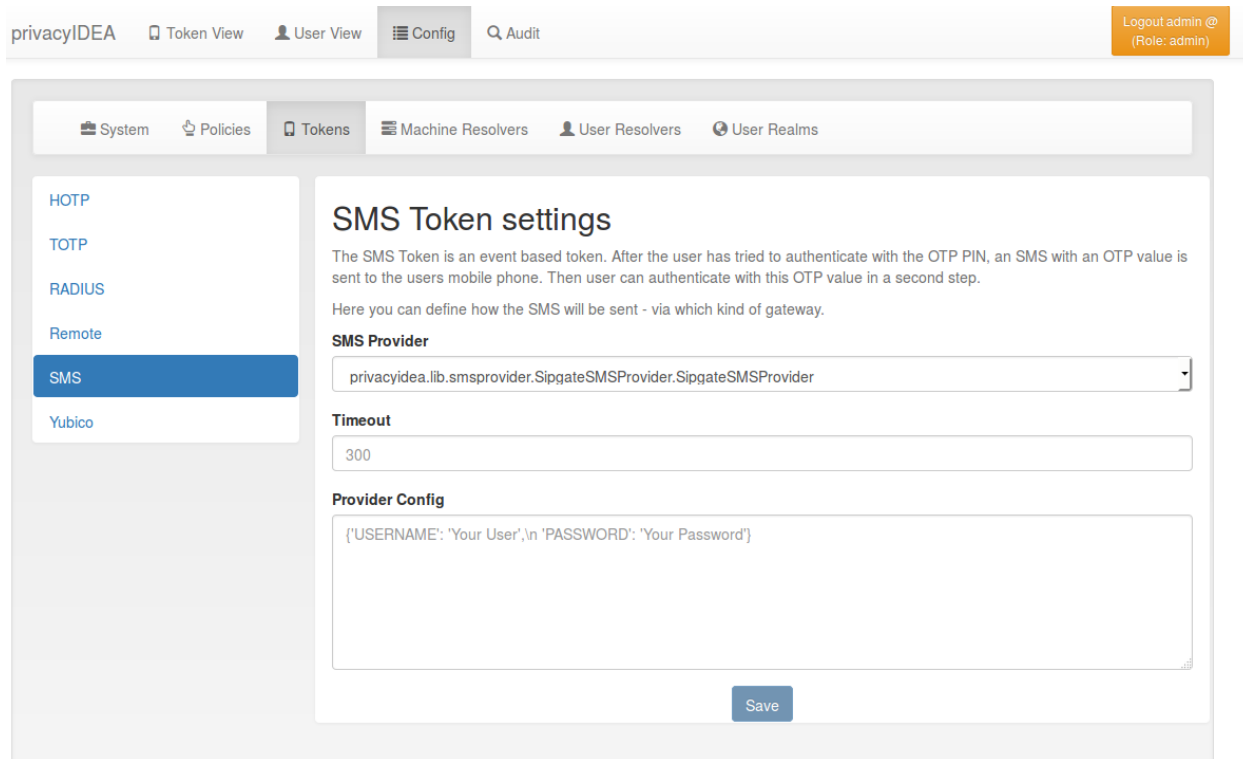


Fig. 1.21: Token Configuration: SMS

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.

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 You can configure the mail parameters for the Email Token centrally at Config -> Tokens -> Email.

Mail Server

The name or IP address of the mail server that is used to send emails.

Port

The port of the mail server.

Mail User

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.

Mail User Password

The password of the mail username to send emails.

pivacyIDEA
Tokens
Users
Machines
Config
Audit
Logout admin @ (Role: admin)

System
Policies
Tokens
Machines
Users
Realms
CAs

HOTP
TOTP
RADIUS
Remote
SMS
Email
Yubico

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.

Mail Server

themis.az.local

Port

25

Mail User

admin

Mail User Password

....

Mail Sender Address

corny@cornelinux.de

OTP validity time

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

120

☐ Use TLS

Fig. 1.22: Email Token configuration

Mail Sender Address

The mail address of the mail sender. This needs to correspond to the *Mail User*.

OTP validity time

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.

Use TLS

Whether the mail server should use TLS.

SMS OTP Token

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.

In the field `SMS_provider` you can enter the SMS provider module, you wish to use. In the *empty* field hit the arrow-down key and you will get a list of the ready made modules.

In the `SMS_configuration` text area you can enter the configuration, which contents is very much dependant on the selected provider module.

The HTTP and the Sipgate module provide a preset-button, which give you an idea of the configuration.

HTTP provider The HTTP provider can be used for any SMS gateway that provides a simple HTTP POST or GET request.

The following parameters can be used:

URL

This is the URL for the gateway.

HTTP_Method

Can be GET or POST.

USERNAME and PASSWORD

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

SMS_PHONENUMBER_KEY

This is the name of the HTTP parameter that holds the mobile phone number of the recipient.

SMS_TEXT_KEY

This is the name of the HTTP parameter that holds the SMS text.

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.

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.

PROXY

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

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.

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

```
{ "URL" : "http://api.clickatell.com/http/sendmsg",
  "PARAMETER" : {
    "user": "YOU",
    "password": "YOUR PASSWORD",
    "api_id": "YOUR API ID"
  },
  "SMS_TEXT_KEY": "text",
  "SMS_PHONENUMBER_KEY": "to",
  "HTTP_Method": "GET",
  "RETURN_SUCCESS" : "ID"
}
```

This will construct 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”.

Sipgate provider The sipgate provider connects to <https://samurai.sipgate.net/RPC2> and takes only two arguments `USERNAME` and `PASSWORD`. The arguments have to be passed in a dictionary like this:

```
{ "USERNAME" : "youruser",
  "PASSWORD" : "yourpassword" }
```

Note: You need to use double quotes around the values.

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

Yubico Cloud mode

The Yubico Cloud mode sends the One Time Password emitted by the yubikey to the Yubico Cloud service.

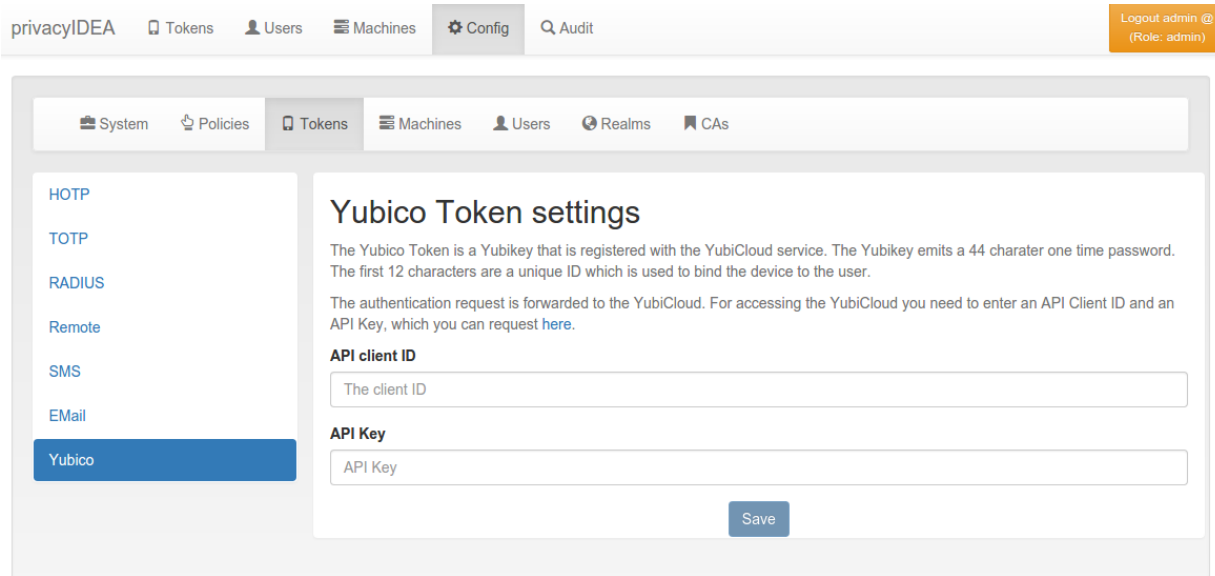


Fig. 1.23: *Configure the Yubico Cloud mode*

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.

You can get your own API key at ⁷.

1.4.5 CA Connectors

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

You can define connections to Certifcate 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.

1.4.6 privacyIDEA setup tool

(TODO): Not yet migrated.

Starting with 1.3.3 privacyIDEA comes with a graphical setup tool to manage your token administrators and RADIUS clients. Thus you will get a kind of appliance experience. To install all necessary components read appliance.

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

⁷ <https://upgrade.yubico.com/getapikey/>.

privacyIDEA
Tokens
Users
Machines
Config
Audit
Logout admin @
(Role: admin)

System
Policies
Tokens
Machines
Users
Realms
CA

All CA Connectors
New Connectors
New LOCAL CA Connector

Edit Local CA Connector myCA

Connector name	myCA
OpenSSL config file	/home/cornelius/src/privacyidea/tests/testdata/ca/openssl.cnf
CA Certificate	/home/cornelius/src/privacyidea/tests/testdata/ca/cacert.pem
CA Key	/home/cornelius/src/privacyidea/tests/testdata/ca/cakey.pem
Working Directory	/home/cornelius/src/privacyidea/tests/testdata/ca/
Certificate Signing Request Directory	/home/cornelius/src/privacyidea/tests/testdata/ca/
Certificate Directory	/home/cornelius/src/privacyidea/tests/testdata/ca/

Save resolver

Fig. 1.24: A local CA definition

privacyIDEA
Tokens
Users
Machines
Config
Audit
Logout admin @
(Role: admin)

All tokens

Enroll Token
Import Tokens
Get Serial

total tokens: 38

Enroll a new token

Certificate: Enroll an x509 Certificate Token.

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

Token data

Generate Request
Upload Request
Upload Certificate

CA Connector

myCA

Certificate Signing Request (PEM)

Paste the Certificate Signing Request

Assign token to user

Fig. 1.25: Enrolling a certificate token

```
privacyidea-setup-tui
```

This will bring you to this start screen.

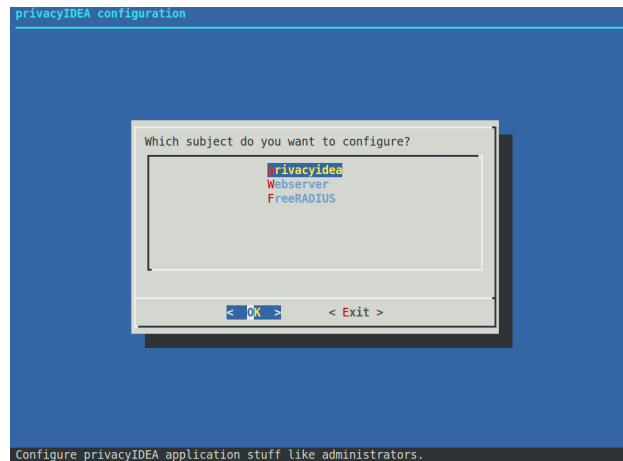


Fig. 1.26: Start screen of the appliance setup tool.

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

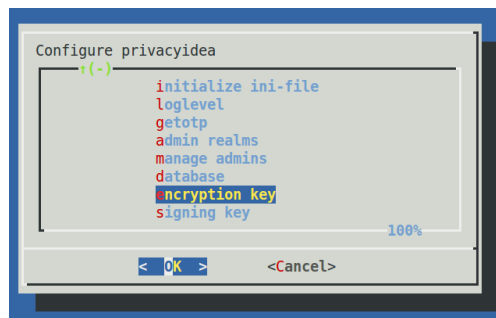


Fig. 1.27: Configure privacyidea

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!

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.

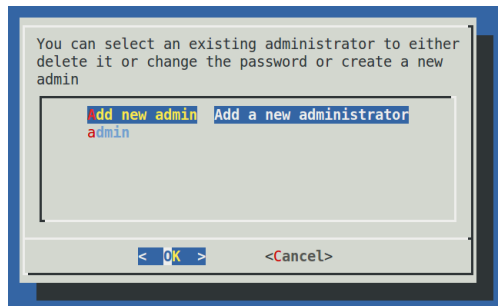


Fig. 1.28: You can create new token administrators, delete them and change their passwords.

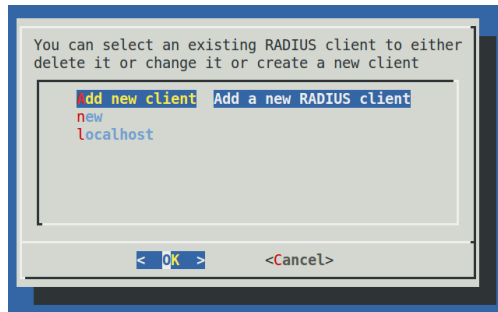
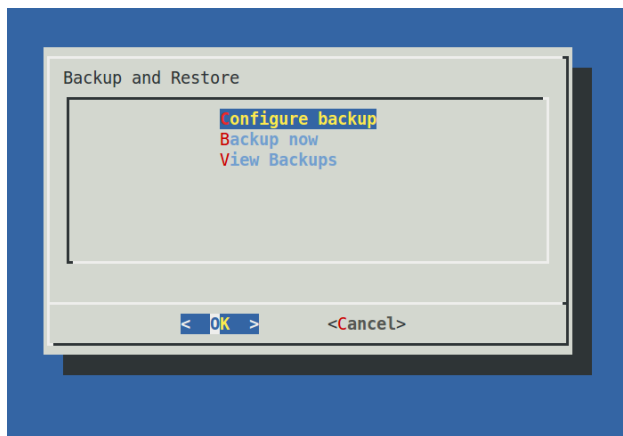


Fig. 1.29: In the FreeRADIUS settings you can create and delete RADIUS clients.



Scheduled 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*.

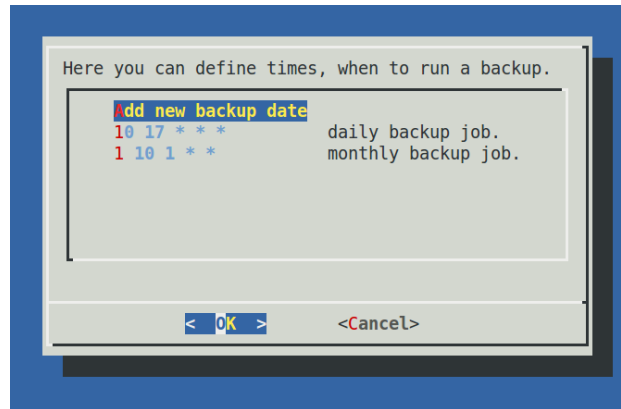


Fig. 1.30: Scheduled backup

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.

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.

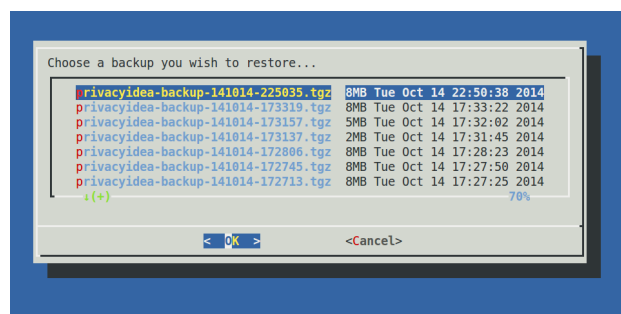


Fig. 1.31: All available backups

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.

1.5 Tokenview

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.

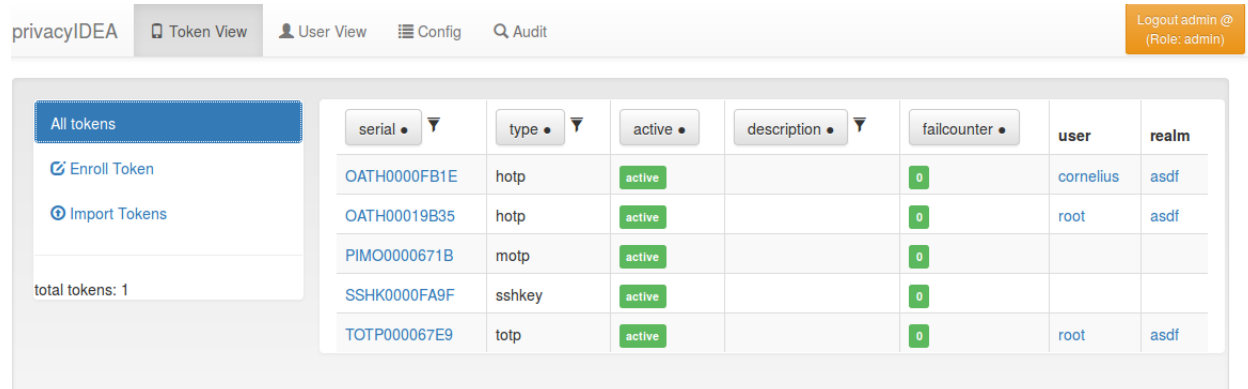


Fig. 1.32: Token View

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

1.5.1 Token Details

The Token Details give you more information about the token and also let the administrator perform specific tasks for this token.

At the bottom you see the assigned user. You can click on the username and change to the [User Details](#).

Lost token

When a user has lost a token, the administrator or the user can create a temporary password token for the user to login.

The administrator has to select the token that was lost and click the button `Lost token`. A new token of type `PW` is generated. The OTP PIN of the old token is automatically copied to the new token. Thus the administrator does not know the OTP PIN, while the user can use his old PIN.

A long password is displayed to the administrator and the administrator can read this password to the user. The user now can authenticate with his old OTP PIN and the long password.

The lost token is deactivated.

Get Serial

The administrator can enter a OTP value that was generated by an unknown token. Then the serial number for the corresponding token is search and displayed.

privacyIDEA
Token View
User View
Config
Audit
Logout admin @ (Role: admin)

All tokens
Token OATH0000FB1E
Enroll Token
Import Tokens
total tokens: 5

Token details for OATH0000FB1E

[View token in Audit log](#)

Type	hotp	Delete
Active	active	Disable
Maxfall	10	Edit
Fail counter	0	
OTP Length	6	
Count	3	
Count Window	10	Edit
Sync Window	1000	Edit
Description		Edit
Info	{ "hashlib": "sha1" }	
Realms	• asdf	Edit

[Resync Token](#)

[Set PIN](#)

[Test token](#)

Assigned User

Username	cornelius	Unassign User
Realm	asdf	
Resolver	asdf	
User Id	1009	

Fig. 1.33: Token Detail

Note: Since OTP values for all matching tokens need to be calculated,

this can be time consuming!

Token settings

You can change the following token settings.

MaxFail and FailCount

If the login fail counter reaches the `MaxFail` the user can not login with this token anymore. The Failcounter `FailCount` has to be reset to zero.

TokenDesc

The token description is also displayed in the tokenview. You can set a description to make it easier to identify a token.

CountWindow

The `CountWindow` is the look ahead window of event based tokens. If the user pressed the button on an event based token the counter in the token is increased. If the user does not use this otp value to authenticate, the server does not know, that the counter in the token was increased. This way the counter in the token can get out of sync with the server.

SyncWindow

If a token was out of sync (see `CountWindow`), then it needs to be synchronized. This is done by entering two consecutive OTP values. The server searches these two values within the next `CountWindow` (default 1000) values.

OtpLen

This is the length of the OTP value that is generated by the token. The password that is entered by the user is splitted according to this length. 6 or 8 characters are splitted as OTP value and the rest is used as static password (OTP PIN).

Hashlib

The HOTP algorithm can be used with SHA1 or SHA256.

Tokeninfo - Auth max

The administrator can set a value how often this token may be used for authentication. If the number of authentication try exceed this value, the token can not be used, until this `Auth max` value is increased.

Note: This way you could create tokens, that can be used only once.

Tokeninfo - Auth max success

The administrator can set a value how often this token may be used to successfully authenticate.

Tokeninfo - Valid start

A timestamp can be set. The token will only be usable for authentication after this start time.

Tokeninfo - Valid end

A timestamp can be set. The token can only be used before this end time.

Note: This way you can create temporary tokens for guests or short time or season employees.

Resync Token

The administrator can select one token and then enter two consecutive OTP values to resynchronize the token if it was out of sync.

set token realm

A token can be assigned to several realms. This is important if you have administrators for different realms. A realm administrator is only allowed to see tokens within his realms. He will not see tokens, that are not in his realm. So you can assign a token to realm A and realm B, thus the administrator A and the administrator B will be able to see the token.

get OTP

If the corresponding getOTP policy (*Policies*) is set, the administrator can get the OTP values of a token from the server without having the token with him.

Note: Of course this is a potential backdoor, since the administrator could login as the user/owner of this very token.

enroll

You can enroll a token either from the Token View or from the *User Details*. When enrolling a token from the User Details the token is directly assigned to the user.

If you enroll the token from the token view, you can select a user, to whom the token will be assigned.

When enrolling a token, you can select the token type and according to the token type other necessary information.

assign

This function is used to assign a token to a user. Select a realm and start typing a username to find the user, to whom the token should be assigned.

unassign

In the token details view you can unassign the token. After that, the token can be assigned to a new user.

enable

If a token is disabled, it can be enabled again.

disable

Tokens can be disabled. Disabled tokens still belong to the assigned user but those tokens can not be used to authenticate. Disabled tokens can be enabled again.

set PIN

You can set the OTP PIN or the mOTP PIN for tokens.

privacyIDEA
Tokens
Users
Machines
Config
Audit
Logout admin @
(Role: admin)

All tokens
Enroll Token
Import Tokens
Get Serial
total tokens: 2

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

Assign token to user

Realm

defrealm

Username

start typing a username

PIN

Type a password

Repeat password

Enroll Token

Fig. 1.34: Token enrollment dialog

Reset Failcounter

If a user locked his token, since he entered wrong OTP values or wrong OTP PINs, the fail counter has reached the mail failcount. The administrator or help desk user can select those tokens and click the button `reset failcounter` to reset the fail counter to zero. The tokens can be used for authentication again.

delete

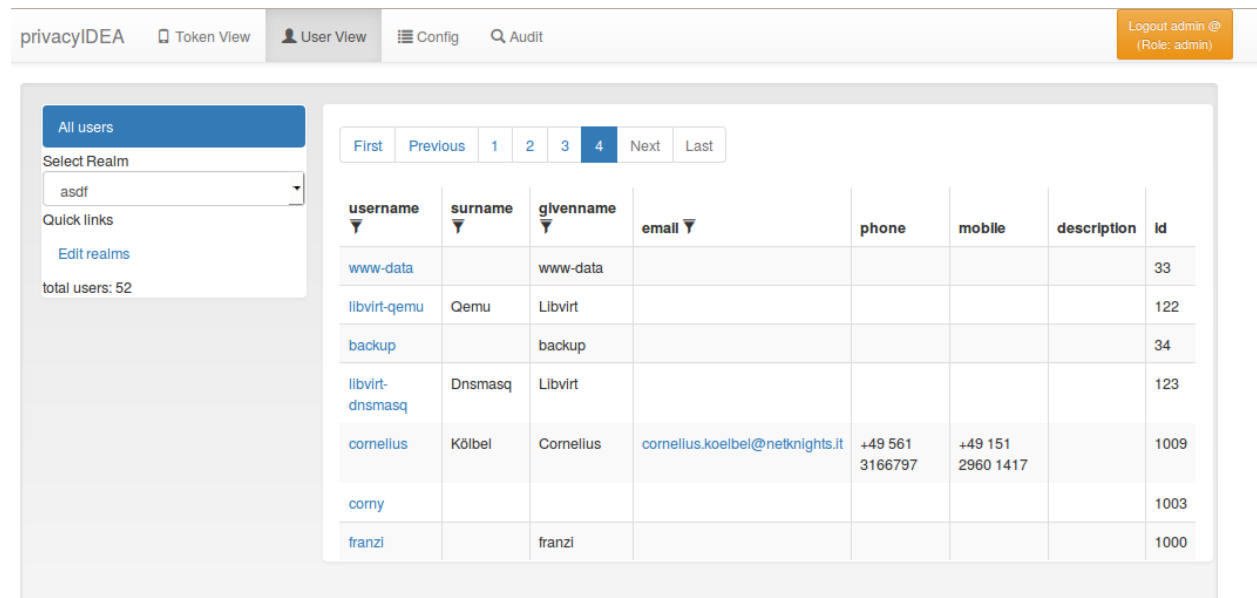
Deleting a token will remove the token from the database. The token information can not be recovered. But all events that occurred with this token still remain in the audit log.

1.6 Userview

The administrator can see all users 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. **(TODO)** No migrated, yet.



The screenshot shows the privacyIDEA User View interface. At the top, there is a navigation bar with links for Token View, User View (active), Config, and Audit. A user is logged in as 'admin' with the role 'admin'. On the left, there is a sidebar with a 'Select Realm' dropdown menu showing 'asdf' and a 'total users: 52' indicator. The main area displays a table of users with columns: username, surname, givenname, email, phone, mobile, description, and id. The table is paginated, showing users 1 through 4 of 52. The users listed are: www-data, libvirt-qemu, backup, libvirt-dnsmasq, cornelius, corny, and franzi.

username	surname	givenname	email	phone	mobile	description	id
www-data		www-data					33
libvirt-qemu	Qemu	Libvirt					122
backup		backup					34
libvirt-dnsmasq	Dnsmasq	Libvirt					123
cornelius	Kölbel	Cornelius	cornelius.koelbel@netknights.it	+49 561 3166797	+49 151 2960 1417		1009
corny							1003
franzi		franzi					1000

Fig. 1.35: User View. List all users in a realm.

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`.

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

1.6.1 User Details

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

The screenshot shows the 'User View' tab in the privacyIDEA interface. The left sidebar contains a list of users, with 'User cornelius' selected. The main content area displays the details for user 'cornelius' in realm 'asdf'. The details include Username (cornelius), Email (cornelius.koelbel@netknights.it), Given name (Cornelius), Surname (Kölbel), Phone (+49 561 3166797), and Mobile (+49 151 2960 1417). Below the details is a table of tokens for user 'cornelius'. The table has columns: serial, type, Active, window, description, failcounter, maxfail, and otpplen. One token is listed with serial 'OATH0000FB1E', type 'hotp', and an active status. Below the table is a button 'Enroll New Token'. At the bottom, there is a section 'Assign a new token' with input fields for 'Serial', 'PIN', and 'Repeat password', and an 'Assign Token' button.

privacyIDEA Token View User View Config Audit Logout admin @ (Role: admin)

All users User cornelius Quick links Edit realms total users: 52

Details for user cornelius in realm asdf View user in Audit log

Username
cornelius

Email
cornelius.koelbel@netknights.it

Given name
Cornelius

Phone
+49 561 3166797

Surname
Kölbel

Mobile
+49 151 2960 1417

Tokens for user cornelius

serial	type	Active	window	description	failcounter	maxfail	otpplen
OATH0000FB1E	hotp	active	10		0	10	6

Enroll New Token

Assign a new token

Serial
start typing a serial number of a token that is not assigned, yet.

PIN
Type a password
Repeat password

Assign Token

Fig. 1.36: User Details.

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](#).

1.6.2 Manage Users

Since version 2.4 pivacyIDEA 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**.

Fig. 1.37: 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.

Fig. 1.38: 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*.

Fig. 1.39: Add a new user.

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.

Simple local users setup

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

```
pi-manage.py 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.py 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.

1.7 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.

This way it is easy to define administrative rights for big groups of administrative users like help desk users in the IT department.

The screenshot shows the 'Edit Policy admins' form in the privacyIDEA web interface. The top navigation bar includes links for System, Policies, Tokens, Machines, Users, Realms, and CAs. The left sidebar shows 'All Policies' and a 'Create new Policy' button. The main form area is titled 'Edit Policy admins' with a 'Disable' button. The form contains the following fields:

- Policy Name:** A text input field containing 'admins'. Below it, a note states: 'If you change the name of the policy, it will create a new policy with the new name!'
- Scope:** A dropdown menu with 'admin' selected.
- Admin-Realm:** A dropdown menu with 'superuser' selected.
- Action:** A dropdown menu with 'adduser, assign, auditlog, caconnectordelete, caconnectorwrite, configdelete' selected.
- User-Realm:** A dropdown menu with 'None selected'.
- User-Resolver:** A dropdown menu with 'None selected'.
- Admin:** A text input field containing 'admin, superuser'.
- Client:** A text input field containing '10.0.0.0/8, 110.0.0.124'.

At the bottom right of the form is a blue button with a plus icon and the text '+ Create Policy'.

Fig. 1.40: Admin scope provides and additional field ‘admin realm’.

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`, `user:frank`, `action:enable` and `realm:sales` means that the administrator *frank* is allowed to enable tokens in the 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*:

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.

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.

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.

setOTPPIN

type: bool

If the `setOTPPIN` action is defined, the administrator is allowed to set the OTP PIN of a token.

setMOTPPIN

type: bool

If the `setMOTPPIN` action is defined, the administrator is allowed to set the mOTP PIN of an mOTP token.

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.

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.

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

(TODO) Not yet migrated.

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

(TODO) Not yet migrated.

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.

getotp

(TODO) Not yet migrated.

type: bool

If the `getserial` action is defined, the administrator is allowed to retrieve OTP values for a given token.

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 assign 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.

setOTPPIN

type: bool

The user is allowed to set the OTP PIN for his tokens.

setMOTPPIN

type: bool

The user is allowed to set the mOTP PIN of mOTP tokens.

getotp

(**TODO**): not yet migrated.

type: bool

The user is allowed to retrieve OTP values from a token.

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.

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.

c are letters matching [a-zA-Z].

n are digits matching [0-9].

s are special characters matching [.,;_-<>+*!/()=?\$\$%&#~^].

Example: The policy action `otp_pin_contents=cn`, `otp_pin_minlength=8` would require the user 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` would still mean, that the OTP PIN needs to contain letters and digits and it must not contain any other characters.

`-cn` (subtraction)

test1234 would be a valid OTP PIN, but *test12\$\$* and *testABCS* would not be valid OTP PINs. The later since it does not contain digits, the first (*test12\$\$*) since it does contain a special character (`$`), which it should 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.

(TODO) grouping and subtraction are not implemented, yet.

Note: You can change these character definitions in the `privacyidea.ini` file using `privacyideaPolicy.pin_c`, `privacyideaPolicy.pin_n` and `privacyideaPolicy.pin_s`. **(Not migrated, yet)**

activateQR

(TODO): not yet migrated.

type: bool

The user is allowed to enroll a QR token.

max_count_dpw

(TODO): not yet migrated.

type: integer

This works together with the `getotp` action. This is the maximum number of OTP values the user may retrieve from DPW tokens.

max_count_hotp

(TODO): not yet migrated.

type: integer

This works together with the `getotp` action. This is the maximum number of OTP values the user may retrieve from HOTP tokens.

max_count_totp

(TODO): not yet migrated.

type: integer

This works together with the `getotp` action. This is the maximum number of OTP values the user may retrieve from TOTP tokens.

auditlog

type: bool

This action allows the user to view and search the audit log for actions with his own tokens.

getserial

(TODO): not yet migrated.

type: bool

This action allows the user to search for the serial number of an unassigned token by entering an OTP value.

updateuser

type: bool

If the `updateuser` action is defined, the user is allowed to change his attributes in the user store.

Note: This is not implemented in the Web UI at the moment.

Note: To be able to edit the attributes, the resolver must be defined as editable.

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: bool

If the user has no token assigned, he will be authenticated against the UserIdResolver, i.e. he needs to provide the LDAP- or SQL-password.

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.

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.

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>`.

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. You can use the tags `<otp>` and `<serial>`.

Default: `<otp>`

emailsubject

type: string

This is the subject of the Email sent by the Email Token. You can use the tags `<otp>` and `<serial>`.

Default: Your OTP

emailautosend

type: bool

If set, a new OTP Email will be sent, when successfully authenticated with an Email Token.

qrtanurl

(TODO): not yet migrated.

type: string

This is the URL for the half automatic mode of the QR token. To this URL the TAN/OTP value will be pushed.

challenge_response

(TODO): not yet migrated.

type: string

This is a list of token types for which challenge response can be used during authentication.

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 comma separated list of case insensitive tokentypes.

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.

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.py 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](#).

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.

max_token_per_user

type: int

Limit the maximum number of tokens per user in this realm.

Note: If you do not set this action, a user may have unlimited tokens assigned.

tokenlabel

type: string

This sets the label for a newly enrolled Google Authenticator. Possible tags to be replaced are <u> for user, <r> for realm and <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.

Warning: If you are only using <u> 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: <u>”, which would result in “user: ”.

autoassignment

type: bool

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 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.
-

Warning: 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: At the moment this randomly generated PIN is not used. It could be used to be sent via a PIN letter in the future.

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

Example:

The action *lostTokenPWLen=10, lostTokenPWContents=Cns* could generate a password like *AC#!/49MK)*.

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.

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.py` 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.

logout_time

type: int

Set the timeout, after which a user in the 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.

You can define as many policies as you wish to. The logic of the policies in the scopes is additive.

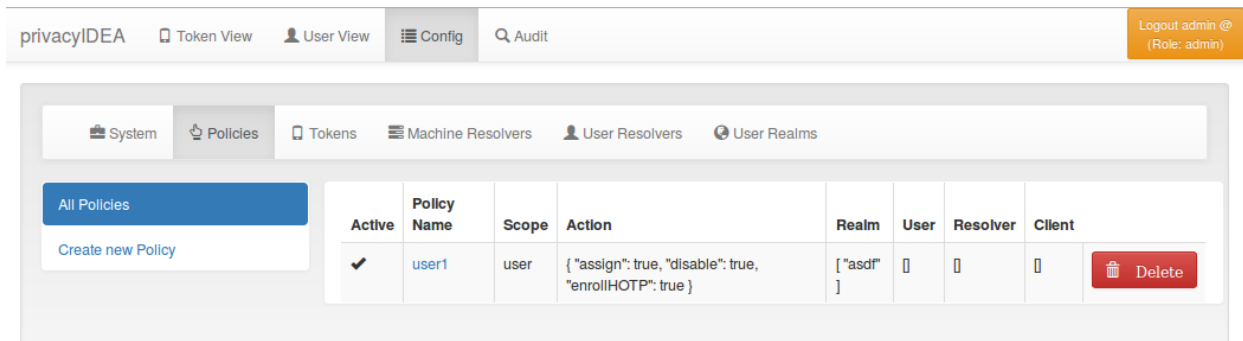


Fig. 1.41: Policy Definition

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.

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.

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 (like 10.2.0.0/16, 192.168.0.1).

time

Not used, yet.

Note: Policies can be active or inactive. So be sure to activate a policy to get the desired effect.

1.8 Audit

The system provides a sophisticated audit log, that can be viewed in the WebUI.

privacyIDEA comes with an SQL audit module. (see [Audit log](#))

1.8.1 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.

You can specify a *highwatermark* and a *lowwatermark*. To clean up the audit log table, you can call the `sqlaudit` module at the command line:

```
python privacyidea/lib/auditmodules/sqlaudit.py \
    -f config/privacyidea.ini.example \
    --low=5000
    --high=10000
```

This will, if there are more than 10.000 log entries, clean all old log entries, so that only 5000 log entries remain.

1.9 Client 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 applications require different authentication items.

Therefore privacyIDEA can define application types. At the moment privacyIDEA knows the applications `luks`, `offline` and `ssh`. You can write your own application class, which is defined in [Application Class](#).

privacyIDEA

Token View

User View

Config

Audit

Logout admin @
(Role: admin)

FirstPrevious12345678910NextLast

Download Audit logDownload file268 entries found.

number	date	action	success	action detail	serial	token type	administrator	user	realm	client	info	sig_check	missing_line	clearance	log level
268	Feb 21, 2015 8:50:54 AM	GET /token/			**		admin			127.0.0.1	realm: [""]				
267	Feb 21, 2015 8:50:54 AM	GET /token/			**		admin			127.0.0.1	realm: [""]				
266	Feb 21, 2015 8:49:18 AM	GET /policy/defs					admin			127.0.0.1					
265	Feb 21, 2015 8:49:18 AM	GET /resolver/					admin			127.0.0.1					
264	Feb 21, 2015 8:49:18 AM	GET /realm/					admin			127.0.0.1					
263	Feb 21, 2015 8:49:17 AM	GET /policy					admin			127.0.0.1	name = None, realm = None, scope = None				

Fig. 1.42: Audit Log

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!

1.9.1 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
nossllcheck=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 **nossllcheck** to

true, otherwise you are vulnerable to man in the middle attacks.

1.9.2 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.

1.9.3 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.

1.10 Application Plugins

privacyIDEA comes with application plugins. These are plugins for applications like PAM, OTRS, Apache2, FreeRADIUS or simpleSAMLphp 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*.

1.10.1 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 with a ready made debian package for Ubuntu or just use the source code file. It is a python module, that requires `pam-python`.

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, than 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.

1.10.2 FreeRADIUS Plugin

If you want to install the FreeRADIUS Plugin on Ubuntu 14.04 LTS this can be easily done, since there is a ready made package (see [FreeRADIUS](#)).

If you want to run your FreeRADIUS server on another distribution, you may download the module at ⁸.

Then you need to configure your FreeRADIUS site and the perl module. The latest FreeRADIUS plugin uses the /validate/check REST API of privacyIDEA.

You need to configure the perl module in FreeRADIUS modules/perl to look something like this:

```
perl {
    module = /usr/share/privacyidea/freeradius/privacyidea_radius.pm
}
```

Your freeradius enabled site config should contain something like this:

```
authenticate {
    Auth-Type Perl {
        perl
    }
    digest
    unix
}
```

While you define the default authenticate type to be Perl in the users file:

```
DEFAULT Auth-Type := Perl
```

Note: The privacyIDEA module uses other perl modules that were not thread safe in the past. So in case you are using old perl dependencies and are experiencing thread problems, please start FreeRADIUS with the -t switch. (Everything works fine with Ubuntu 14.04 and Debian 7.)

You can test the RADIUS setup using a command like this:

```
echo "User-Name=user, Password=password" | radclient -sx yourRadiusServer \
    auth topsecret
```

Note: Do not forget to configure the clients.conf accordingly.

Read more about radius_and_realms or rlm_perl_ini.

1.10.3 simpleSAMLphp Plugin

You can install the plugin for simpleSAMLphp on Ubuntu 14.04 LTS (see [SimpleSAMLphp](#)) or on any other distribution using the source files from ⁹.

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
```

⁸ <https://github.com/privacyidea/privacyidea/tree/master/authmodules/FreeRADIUS>

⁹ <https://github.com/privacyidea/privacyidea/tree/master/authmodules/simpleSAMLphp>


```

    * A port can be added by a colon
    * Required.
    */
    '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',
                           ),
),

```

1.10.4 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.

1.10.5 OTRS

There are two plugins for OTRS. For OTRS version 4.0 and higher use *privacyIDEA-4_0.pm*.

This perl module needs to be installed to the directory `Kernel/System/Auth`.

On Ubuntu 14.04 LTS you can also install the module using the PPA repository and installing:

```
apt-get install privacyidea-otrs
```

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.

1.10.6 Apache2

The Apache plugin uses `mod_wsgi` and `redis` to provide a basic authentication on Apache2 side and validating the credentials against privacyIDEA.

On Ubuntu 14.04 LTS you can easily install the module from the PPA repository by issuing:

```
apt-get install privacyidea-apache-client
```

To activate the OTP authentication on a “Location” or “Directory” you need to configure Apache2 like this:

```
<Directory /var/www/html/secret_dir>
    AuthType Basic
    AuthName "Protected Area"
    AuthBasicProvider wsgi
    WSGIAuthUserScript /usr/share/pyshared/privacyidea_apache.py
    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 as the successful authentication. 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. This way the fix password component of the user will get exposed!

1.10.7 Further plugins

You can find further plugins for Dokuwiki, Wordpress, Contao and Django at ¹⁰.

1.11 Tools

(TODO): Not yet migrated.

The menu `tools` contains some helpful tools to manage your tokens.

1.11.1 Get Serial by OTP value

Here you can enter an OTP value and have the system identify the token. This can be useful if the printed serial number on the token can not be read anymore or if the hardware token has not serial number printed on it at all.

You can choose the

- tokentype
- whether the token is an assigned token or not
- and the realm of the token

to set limits to the token search.

Warning: The system needs to get to all tokens and calculate the next (default) 10 OTP values. Depending on the number of tokens you have in the system this can be very time consuming!

1.11.2 Copy token PIN

Here you can enter a token serial number of the token from which you want to copy the OTP PIN and the serial number of the token to which you want to copy it.

This function is also used in the lost token scenario.

But the help desk can also use it if the administrator enrolls a new token to the user and

1. the user can not set the OTP PIN and
2. the administrator should not set or know the OTP PIN.

Then the administrator can create a second token for the user and copy the OTP PIN (which only the user knows) of the old token to the new, second token.

¹⁰ <https://github.com/cornelinux?tab=repositories>

1.11.3 Check Policy

If you have complicated policy settings you can use this dialog to determine if the policies behave as expected. You can enter the scope, the real, action user and client to “simulate” e.g. an authentication request.

The system will tell you if any policy is triggered.

1.11.4 Export token information

Here you can export the list of the tokens to a CSV file.

Note: In the resolver you can define additional fields, that are usually not used by privacyIDEA. But you can add those fields to the export. Thus you can e.g. add special LDAP attributes in the list of the exported tokens.

1.11.5 Export audit information

Here you can export the audit information.

Warning: You should limit the export to a number of audit entries. As the audit log can grow very big, the export of 20.000 audit lines could result in blocking the system.

1.12 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.

1.12.1 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 ¹¹.

1.12.2 Yubikey CSV

Here you can import the CSV file that is written by the Yubikey personalization tool ¹². privacyIDEA can import all Yubikey modes, either Yubico mode or HOTP mode.

Note: There is an annoying drawback of the personalization tool: If you initialize several HOTP yubikeys it will not write the serial number to the file.

1.12.3 PSKC

The *Portable Symmetric Key Container* is specified in ¹³. 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.

¹¹ <http://tools.ietf.org/html/rfc6287#section-6>

¹² <http://www.yubico.com/products/services-software/personalization-tools/use/>

¹³ <https://tools.ietf.org/html/rfc6030>

1.13 Code Documentation

The code roughly has three levels.

1.13.1 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.

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

{
  "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": "pivacyIDEA 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. :return:

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.

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

{
  "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:

- lib/token/check_user_pass (user, passw, options)
- lib/token/check_token_list(list, passw, user, options)
- lib/tokenclass/authenticate(pass, user, options)
- lib/tokenclass/check_pin(pin, user, options)
- lib/tokenclass/check_otp(otpval, options)

IN case if authenticating a serial number:

- lib/token/check_serial_pass(serial, passw, options)

- lib/token/check_token_list(list, passw, user, options)
- lib/tokenclass/authenticate(pass, user, options)
- lib/tokenclass/check_pin(pin, user, options)
- lib/tokenclass/check_otp(otpval, options)

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,
    "value": { "auth": true,
              "username": <loginname>,
              "realm": ....,
              "surname": ....,
              "givenname": .....,
              "mobile": ....,
              "phone": ....,
              "email": ....
            }
  },
  "version": "privacyIDEA unknown"
}
```

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": {
      "auth": true,
      "username": <loginname>,
      "realm": ....,
      "surname": ....,
      "givenname": .....,
      "mobile": ....,
      "phone": ....,
      "email": ....
    }
  },
  "version": "privacyIDEA unknown"
}

```

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`.

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.
- **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 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"
}

```

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`.

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.
- **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 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"
}

```

System endpoints

The code of this module is tested in `tests/test_api_system.py`

POST /system/setDefault

method: system/set

description: define default settings for tokens. These default settings are used when new tokens are generated. The default settings will not affect already enrolled tokens.

arguments:

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]

returns:

a json result with a boolean “result”: true

exception: if an error occurs an exception is serialized and returned

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

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

Parameters

- **pairs** (*key-value*) – pair of &keyname=value pairs

Return a json result with a boolean “result”: true

GET /system/

This endpoint either returns all config entries or only the value of the one config key.

Parameters

- **key** – The key to return.

Return A json response or a single value, when queried with a key.

Rtype json or scalar

GET /system/ (key)

This endpoint either returns all config entries or only the value of the one config key.

Parameters

- **key** – The key to return.

Return A json response or a single value, when queried with a key.

Rtype json or scalar

DELETE `/system/` (*key*)

delete a configuration key * if an error occurs an exception is serializedsetConfig and returned

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`

Return a json result with True, if the given values can create a working resolver and a description.

GET `/resolver/`

returns a json list of all resolver.

Parameters

- **type** (*basestring* ("I")) – Only return resolvers of type (like passwdresolver..)
- **editable** – If only editable resolvers should be returned

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** (*basestring*) – the name of the resolver.
- **type** – the type of the resolver. Valid types are passwdresolver,

ldapresolver, sqlresolver, scimresolver :type type: string :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

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

GET `/resolver/` (*resolver*)

This function retrieves the definition of a single resolver. It can be called via `/system/getResolver?resolver=` or via `/resolver/<resolver>`

Parameters

- **resolver** – the name of the resolver

Return a json result with the configuration of a specified resolver

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

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 take 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": "resol_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.

Parameters

- **realm** – The unique name of the realm
- **resolvers** – A comma separated list of unique resolver names or a

list object :type resolvers: string or list :param priority: Additional parameters priority.<resolvername> define the

priority of the resolvers within this realm.

Return a json result with a list of Realms

In the result it returns a list of added resolvers and a list of resolvers, that could not be added.

Example request:

To create a new realm “newrealm”, that consists of the resolvers “resol_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=resol_with_realm, reso2_with_realm
priority.resol_with_realm=1
priority.reso2_with_realm=2

```

Example response:

```

HTTP/1.1 200 OK
Content-Type: application/json

```



```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": {
      "added": ["resol_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",
  "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 administrators.

To see how to authenticate read [Authentication endpoints](#).

POST /token/unassign

Unsign 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 “value”: True.

Rtype json object

POST /token/copyuser

Copy the token user from one token to the other.

You can call the function like this: POST /token/copyuser?from=<serial>&to=<something>

Parameters

- **from** – the serial number of the single, from where you want to

copy the pin. :type from: basestring :param to: the serial number of the single, from where you want to copy the pin. :type to: basestring :return: returns value=True in case of success :rtype: bool

POST /token/disable

Disable a single token or all the tokens of a user. Disabled tokens can not be used to authenticate but can be enabled again.

You can call the function like this: POST /token/disable?serial=<serial> POST /token/disable?user=<user>&realm=<realm> POST /token/disable/<serial>

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.

You can call the function like this: POST /token/copypin?from=<serial>&to=<something>

Parameters

- **from** – the serial number of the single, from where you want to

copy the pin. :type from: basestring :param to: the serial number of the single, from where you want to copy the pin. :type to: basestring :return: returns value=True in case of success :rtype: bool

POST /token/assign

Assign a token to a user. The required arguments are serial, user and realm.

Return In case of success it returns “value”: True.

Rtype json object

POST /token/enable

Enable a single token or all the tokens of a user.

You can call the function like this: POST /token/enable?serial=<serial> POST /token/enable?user=<user>&realm=<realm> POST /token/enable/<serial>

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

Resync the OTP token by providing two consecutive OTP values.

You can call the function like this: POST /token/resync?serial=<serial>&otp1=<otp1>&otp2=<otp2> POST /token/resync/<serial>?otp1=<otp1>&otp2=<otp2>

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

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.

You can call the function like this: POST /token/setpin?serial=<serial>&userpin=<userpin>&sopin=<sopin> POST /token/setpin/<serial>?userpin=<userpin>&sopin=<sopin>

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

Reset the failcounter of a single token or of all tokens of a user.

You can call the function like this: POST /token/reset?serial=<serial> POST /token/reset?user=<user>&realm=<realm> POST /token/reset/<serial>

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.

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** – required: the serial number/identifier of the token
- **description** – A description for the token
- **pin** – the pin of the user pass
- **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 oder sha256

ocra arguments: for generating OCRA Tokens type=ocra you can specify the following parameters: :param
ocrasuite: if you do not want to use the default

ocra suite OCRA-1:HOTP-SHA256-8:QA64

Parameters

- **sharedsecret** – if you are in Step0 of enrolling an OCRA/QR token the sharedsecret=1 specifies, that you want to generate a shared secret
- **activationcode** – if you are in Step1 of enrolling an OCRA token you need to pass the activation code, that was generated in the QRTAN-App

Return a json result with a boolean “result”: true

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,iVBORw0KGgoAAAANSUheUgAAAcIAAA",
      "value": "otpauth://hotp/mylabel?secret=GEZDGNBVGY3TQOJQGEZDGNBVGY3TQOJQ&counter=0"
    },
    "oathurl": {
      "description": "URL for OATH token",
      "img": "<img width=250 src='data:image/png;base64,iVBORw0KGgoAAAANSUheUgAAAcIAAA",
      "value": "oathtoken:///addToken?name=mylabel&lockdown=true&key=3132333435363738393031"
    },
    "otpkey": {
      "description": "OTP seed",
      "img": "<img width=200 src='data:image/png;base64,iVBORw0KGgoAAAANSUheUgAAAUoAAA",
      "value": "seed://3132333435363738393031323334353637383930"
    },
    "serial": "OATH00096020"
  },
  "id": 1,
```

```

    "jsonrpc": "2.0",
    "result": {
      "status": true,
      "value": true
    },
    "version": "pivacyIDEA unknown"
  }

```

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

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.

You can call the function like this: POST /token/set?serial=<serial>&description=<something> POST /token/set/<serial>?hashlib=<hash> POST /token/set?user=<username>&realm=<realm>&sync_window=100

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/

The method is called at /token Display the list of available tokens.

Parameters

- **serial** – Display the token data of this single token. You can do a

not strict matching by specifying a serial like “OATH”. :param type: Display only token of type. You can do a non strict matching by specifying a tokentype like “otp”, to filter hotp and totp tokens. :param user: display tokens of this user :param viewrealm: takes a realm, only the tokens in this realm will be displayed :param description: Display token with this kind of description :type description: basestring :param sortby: sort the output by column :param sortdir: asc/desc :param page: request a certain page :param assigned: Only return assigned (True) or not assigned (False) tokens :param pagesize: limit the number of returned tokens :param user_fields: additional user fields from the userid resolver of the owner (user) :param outform: if set to “csv”, then 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.

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
- **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/disable/ (serial)

Disable a single token or all the tokens of a user. Disabled tokens can not be used to authenticate but can be enabled again.

You can call the function like this: POST /token/disable?serial=<serial> POST /token/disable?user=<user>&realm=<realm> POST /token/disable/<serial>

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/enable/ (serial)

Enable a single token or all the tokens of a user.

You can call the function like this: POST /token/enable?serial=<serial> POST /token/enable?user=<user>&realm=<realm> POST /token/enable/<serial>

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)

Resync the OTP token by providing two consecutive OTP values.

You can call the function like this: POST /token/resync?serial=<serial>&otp1=<otp1>&otp2=<otp2> POST /token/resync/<serial>?otp1=<otp1>&otp2=<otp2>

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*)

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.

You can call the function like this: POST /token/setpin?serial=<serial>&userpin=<userpin>&sopin=<sopin>
POST /token/setpin/<serial>?userpin=<userpin>&sopin=<sopin>

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*)

Reset the failcounter of a single token or of all tokens of a user.

You can call the function like this: POST /token/reset?serial=<serial> POST /token/reset/<serial>
ken/reset?user=<user>&realm=<realm> POST /token/reset/<serial>

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/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 /to-
ken/realm/<serial>?realms=<hash>

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/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.

Parameters

- **filename** (*basestring*) – The name of the token file, that is imported
- **type** (*basestring*) – The file type. Can be “aladdin-xml”, “oathcsv” or “yubikeycsv”.

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

Parameters

- **serial** (*basestring*) – the serial number of the lost token.

Return returns value=dictionary in case of success

Rtype bool

POST /token/set/ (*serial*)

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

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.

You can call the function like this: POST /token/set?serial=<serial>&description=<something> POST /token/set/<serial>?hashlib=<hash> POST /token/set?user=<username>&realm=<realm>&sync_window=100

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

DELETE /token/ (*serial*)

Delete a token by its serial number. There are three different ways to delete a token.

DELETE /token/<serial>

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

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 K\u00f6lbel,,+49 151 2960 1417,+49 561 3166797,cornelius.koel",
        "email": "cornelius.koelbel@netknights.it",
        "givenname": "Cornelius",
        "mobile": "+49 151 2960 1417",
        "phone": "+49 561 3166797",
        "surname": "K\u00f6lbel",
        "userid": "1009",
        "username": "cornelius",
        "resolver": "name-of-resolver"
      }
    ]
  },
  "version": "privacyIDEA unknown"
}
```

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/

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.

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.

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"
      }
    }
  },
  "version": "pivacyIDEA unknown"
}
```

GET /policy/defs

This is a helper function that returns the POSSIBLE policy definitions, that can be used to define your policies.

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

this function is used to retrieve the policies that you defined. It can also be used to export the policy to a file.

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.

JSON Parameters

- **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": "pivacyIDEA unknown"
}

```

POST /policy/disable/ (*name*)

Disable a given policy by its name. :param name: The name of the policy :return: ID in the database

POST /policy/enable/ (*name*)

Enable a given policy by its name. :param name: Name of the policy :return: ID in the database

GET /policy/export/ (*export*)

this function is used to retrieve the policies that you defined. It can also be used to export the policy to a file.

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.

JSON Parameters

- **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/import/ (*filename*)

This function is used to import policies from a file.

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"
}

```

GET /policy/defs/ (*scope*)

This is a helper function that returns the POSSIBLE policy definitions, that can be used to define your policies.

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

POST /policy/ (*name*)

Creates a new policy that defines access or behaviour of different actions in privacyIDEA

Parameters

- **name** (*basestring*) – name of the policy

JSON Parameters

- **scope** – the scope of the policy like “admin”, “system”, “authentication” or “selfservice”
- **adminrealm** – Realm 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** (*string with wild cards or list of strings*) – The policy is valid for these users
- **time** – on which time does this policy hold
- **client** (*IP address with subnet*) – for which requesting client this should be

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 “poll” is created.

```
POST /policy/poll HTTP/1.1
Host: example.com
Accept: application/json

scope=admin
realm=realm1
action=enroll, disable
```

Example response:

```
HTTP/1.0 200 OK
Content-Length: 354
Content-Type: application/json

{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": {
      "setPolicy poll": 1
    }
  },
  "version": "pivacyIDEA unknown"
}
```

GET /policy/ (*name*)

this function is used to retrieve the policies that you defined. It can also be used to export the policy to a file.

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.

JSON Parameters

- **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"
}
```

DELETE /policy/ (name)

This deletes the policy of the given name.

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 “poll” is created.

```
DELETE /policy/poll 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": "pIvacyIDEA unknown"
}
```

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 :param resolver: identify the machine by the machine ID and the resolver name :param serial: identify the token by the serial number :param application: the name of the application like “luks” or “ssh”.

Parameters not listed will be treated as additional options.

Return

GET /machine/authitem

This fetches the authentication items for a given application and the given client machine.

Parameters

- **challenge** – 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.

Parameters

- **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 :param resolver: identify the machine by the machine ID and the resolver name :param serial: identify the token by the serial number :param 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",
  "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 :param 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": "908asljdass90ad0",
        "hostname": [ "flavon.example.com", "test.example.com" ],
        "ip": "1.2.3.4",
        "resolver_name": "machineresolver1"
      },
      {
        "id": "1908209x48x2183",
        "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 :param resolver: identify the machine by the machine ID and the resolver name :param serial: identify the token by the serial number :param 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" }
```

GET /machine/authitem/ (*application*)

This fetches the authentication items for a given application and the given client machine.

Parameters

- **challenge** – 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.

Parameters

- **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"
}
```

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

1.13.2 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 known, yet.

`realm.py` contains functions to deal with realm. Realms are a list of several resolvers. So prior to both 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 webservice!

This code is tested in `tests/test_lib_user.py`

```
privacyidea.lib.user.User(*args, **kws)
```

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.

```
privacyidea.lib.user.create_user(*args, **kws)
```

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”, “given-name”, “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

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_info(*args, **kws)
```

return the detailed information for a user in a resolver

Parameters

- **userid** (*string*) – The id of the user in a resolver
- **resolvername** – The name of the resolver

Returns a dict with all the userinformation

Return type dict

```
privacyidea.lib.user.get_user_list(*args, **kws)
```

```
privacyidea.lib.user.get_username(*args, **kws)
```

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.split_user(*args, **kws)
```

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, 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.

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. 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 Authentication 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 response:

```

HTTP/1.1 200 OK
Content-Type: application/json

{
  "detail": {
    "certificate": "...PEM..."
  },
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": true
  },
  "version": "privacyIDEA unknown"
}

```

classmethod `get_class_info(*args, **kws)`

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

classmethod `get_class_prefix()`

classmethod `get_class_type()`

get_init_detail (**args, **kws*)

At the end of the initialization we return the certificate

update (*param*)

This method is called during the initialization process. :param param: parameters from the token init :type param: dict :return: None

Daplug Token

class `privacyidea.lib.tokens.daplugtoken.DaplugTokenClass(*args, **kws)`

daplug token class implementation

check_otp (**args, **kws*)

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 (**args, **kws*)

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

classmethod get_class_info (**args, **kws*)

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

classmethod get_class_prefix ()

classmethod get_class_type ()

get_multi_otp (**args, **kws*)

get_otp (**args, **kws*)

resync (**args, **kws*)

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 (*passwd, user=None, options=None*)

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 (**args, **kws*)

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 (**args, **kws*)

create a challenge, which is submitted to the user

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

classmethod **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.

classmethod **get_class_prefix** ()

classmethod **get_class_type** ()

return the generic token class identifier

update (**args, **kws*)

update - process initialization parameters

Parameters **param** (*dict*) – dict of initialization parameters

Returns nothing

HOTP Token

class `privacyidea.lib.tokens.hotptoken.HotpTokenClass` (**args, **kws*)
hotp token class implementation

check_otp (**args, **kws*)

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 (**args, **kws*)

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

classmethod get_class_info (**args, **kws*)

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

classmethod get_class_prefix ()

Return the prefix, that is used as a prefix for the serial numbers. :return: oath

classmethod get_class_type ()

return the token type shortname

Returns 'hotp'

Return type string

get_init_detail (**args, **kws*)

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 (**args, **kws*)

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 is 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

Returns tuple of status: boolean, error: text and the OTP dictionary

get_otp (*args, **kws)
return the next otp value

Parameters **curTime** – Not Used in HOTP

Returns next otp value and PIN if possible

Return type tuple

get_sync_timeout ()
get the token sync timeout value

Returns timeout value in seconds

Return type int

hashlib

is_challenge_request (*args, **kws)
check, if the request would start a challenge

- default: if the passw contains only the pin, this request would trigger a challenge

- in this place as well the policy for a token is checked

Parameters

- **passw** – password, which might be pin or pin+otp
- **options** – dictionary of additional request parameters

Returns returns true or false

resync (*args, **kws)
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 (*args, **kws)
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** (*args, **kws)

check_otp (*args, **kws)

validate the token otp against a given otpvalue

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

classmethod 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

classmethod get_class_prefix ()

classmethod get_class_type ()

get_init_detail (*args, **kws)

to complete the token normalisation, the response of the initialization should be build by the token specific method, the getInitDetails

update (*args, **kws)

update - process initialization parameters

Parameters **param** (*dict*) – dict of initialization parameters

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

check_otp (*args, **kws)

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

classmethod get_class_info (*args, **kws)

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

classmethod `get_class_prefix()`

classmethod `get_class_type()`

set_otplen (**args, **kws*)

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

RADIUS Token

class `privacyidea.lib.tokens.radius.token.RadiusTokenClass` (*db_token*)

check_otp (**args, **kws*)

run the RADIUS request against the RADIUS server

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_pin_local

lookup if pin should be checked locally or on radius host

Returns bool

classmethod `get_class_info` (**args, **kws*)

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

classmethod `get_class_prefix()`

classmethod `get_class_type()`

split_pin_pass (**args, **kws*)

Split the PIN and the OTP value. Only if it is locally checked and not remotely.

update (*param*)

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=register
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,
    "value": true
  },
  "version": "privacyIDEA unknown"
}
```

check_otp (**args*, ***kws*)

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

classmethod `get_class_info` (**args*, ***kws*)

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

classmethod `get_class_prefix` ()

classmethod `get_class_type` ()

get_init_detail (*args, **kws)

At the end of the initialization we return the registration code.

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 (*args, **kws)

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

check_pin_local

lookup if pin should be checked locally or on remote host

Returns bool

classmethod **get_class_info** (*args, **kws)

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

classmethod **get_class_prefix** ()

return the token type prefix

classmethod `get_class_type()`

return the class type identifier

is_challenge_request (*args, **kws)

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.sms.token.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 siggate 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,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": true
  },
  "version": "privacyIDEA unknown"
}
```

check_otp (*args, **kws)

check the otpval of a token against a given counter and the window

Parameters **passwd** (*string*) – the to be verified passwd/pin

Returns counter if found, -1 if not found

Return type int

create_challenge (*args, **kws)

create a challenge, which is submitted to the user

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

classmethod 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.

classmethod `get_class_prefix()`

classmethod `get_class_type()`

return the generic token class identifier

is_challenge_request (*args, **kws)

check, if the request would start a challenge

if the passw contains only the pin, this request would trigger a challenge

in this place as well the policy for a token is checked

Parameters

- **passw** – password, which might be pin or pin+otp
- **user** – The authenticating user
- **options** – dictionary of additional request parameters

Returns returns true or false

update (*args, **kws)

process initialization parameters

Parameters **param** (*dict*) – dict of initialization parameters

Returns nothing

SPass Token

class `privacyidea.lib.tokens.spasstoken.SpasmTokenClass` (*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 (*args, **kws)

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 == 0

classmethod `get_class_info` (*args, **kws)

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

classmethod `get_class_prefix()`

classmethod `get_class_type()`

is_challenge_request (*passw*, *user*, *options=None*)

The spass token does not support challenge response :param passw: :param user: :param options: :return:

is_challenge_response (*passw*, *user*, *options=None*, *challenges=None*)

update (*param*)

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.

classmethod `get_class_info` (**args, **kws*)

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

classmethod `get_class_prefix` ()

classmethod `get_class_type` ()

get_sshkey (**args, **kws*)

returns the public SSH key

Returns SSH pub key

Return type string

update (*param*)

The key holds the public ssh key and this is required

The key probably is of the form “ssh-rsa BASE64 comment”

TOTP Token

class `privacyidea.lib.tokens.totptoken.TotpTokenClass` (**args, **kws*)

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 (**args, **kws*)

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

classmethod `get_class_info (*args, **kws)`
 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

classmethod `get_class_prefix ()`

Return the prefix, that is used as a prefix for the serial numbers. :return: TOTP

classmethod `get_class_type ()`

return the token type shortname

Returns 'totp'

Return type string

get_multi_otp (**args, **kws*)

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 server time
- **timestamp** (*epoch time*) – Simulate the server time

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

hashlib

resync (**args, **kws*)

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

timeshift

timestep

timewindow

update (*args, **kwargs)

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

Yubico Token

class privacyidea.lib.tokens.yubicotoken.YubicoTokenClass (db_token)

check_otp (*args, **kwargs)

Here we contact the Yubico Cloud server to validate the OtpVal.

classmethod **get_class_info** (*args, **kwargs)

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

classmethod **get_class_prefix** ()

classmethod **get_class_type** ()

update (param)

Yubikey Token

class privacyidea.lib.tokens.yubikeytoken.YubikeyTokenClass (db_token)

The Yubikey Token in the Yubico AES mode

check_otp (*args, **kwargs)

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 (*args, **kwargs)

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.

classmethod `get_class_info (*args, **kws)`
 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.

classmethod `get_class_prefix ()`

classmethod `get_class_type ()`

is_challenge_request (**args, **kws*)

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

class `privacyidea.lib.tokenclass.TokenClass (*args, **kws)`

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:

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

challenge_janitor ()

Just clean up all challenges, for which the expiration has expired.

Returns None

check_auth_counter ()

This function checks the count_auth and the count_auth_success. If the count_auth is less than count_auth_max and count_auth_success is less than count_auth_success_max 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_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 (*args, **kws)

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_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.

del_tokeninfo (key=None)

delete_token ()

delete the database token

enable (enable=True)

get_QRimage_data (response_detail)

FIXME: Do we really use this?

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

classmethod get_class_info (key=None, ret='all')

classmethod get_class_prefix ()

classmethod `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()

get_failcount()

classmethod `get_hashlib(hLibStr)`

Returns a hashlib function for a given string :param hLibStr: the hashlib :type hLibStr: string :return: the hashlib :rtype: function

get_init_detail (*params=None, user=None*)

to complete the token normalisation, the response of the initialisation should be build by this token specific method.

`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 (**args, **kws*)

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: list

get_serial ()

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 ()

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

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

get_vars (*save=False*)

return the token state as dicts with keys like type, token, mode... :return: token as dict

inc_count_auth ()

Increase the counter, that counts authentications - successful and unsuccessful

inc_count_auth_success ()

Increase the counter, that counts successful authentications

inc_failcount ()

inc_otp_counter (*args, **kws)

Increase the otp counter and store the token in the database :param counter: the new counter value. If counter is given, than

the counter is increased by (counter+1) If the counter is not given, the counter is increased by +1

Parameters **reset** (*bool*) – reset the failcounter if set to True

Returns the new counter value

is_active ()

is_challenge_request (passwd, 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 `passwd` 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!

Parameters

- **passwd** (*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 (passwd, 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

- **passwd** (*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

reset ()

Reset the failcounter

resync (otp1, otp2, options=None)

save ()

Save the database token

set_count_auth (*count*)

Sets the counter for the occurred login attempts 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_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*)

Set the list of the realms of a token. :param realms: realms the token should be assigned to :type realms: list

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 (*user, report=None*)

Set the user attributes (uid, resolvername, resolvertype) of a token.

Parameters

- **user** – a User() object, consisting of loginname and realm
- **report** – tbf.

Returns None

set_user_identifiers (*uid, resolvername, resolvertype*)

(was setUid) Set the user attributes of a token :param uid: The user id in the user source :param resolvername: The name of the resolver :param resolvertype: The type of the resolver :return: None

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 “%d/%m/%y %H:%M”

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 “%d/%m/%y %H:%M”

if the format is wrong, the method will throw an exception

split_pin_pass (*passwd, 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

- **passwd** – 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

update (*param*, *reset_failcount=True*)

Update the token object

Parameters **param** – a dictionary with different params like keysize, description, genkey, otp-key, pin

Type param: dict

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(*args, **kws)`

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(*args, **kws)`

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

Returns True if the token was assigned, in case of an error an exception

is thrown :rtype: bool

`privacyidea.lib.token.auto_assign_token(*args, **kws)`

This function is called to auto_assign a token to the user.

If the user does not have a token, yet, the not assigned tokens in his realm are searched if they match the given passw.

Parameters

- **user** (*User object*) – The user, who is authenticating
- **passw** (*basestring*) – The given password (pin + otp)
- **pin** –
- **param** (*dict*) – additional parameters

Returns True or False and detailed reply information

Return type bool, dict

`privacyidea.lib.token.check_serial(*args, **kws)`

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** – Serial 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(*args, **kws)`

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(*args, **kws)`

this takes a list of token objects and tries to find the matching token for the given passw. In 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
- **option** – additional parameters, which are passed to the token

Returns tuple of success and optional response

Return type (bool, dict)

`privacyidea.lib.token.check_user_pass(*args, **kws)`

This function checks the otp for a given user. It is called by the API /validate/check and simplecheck

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(*args, **kws)`

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(*args, **kws)`

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(*args, **kws)`

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_tokenclass_object(*args, **kws)`

(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.enable_token(*args, **kws)`

Enable or disable a token. 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(*args, **kws)`
generate a serial for a given tokentype

Parameters

- **tokentype** – the token type prefix is done by a lookup on the tokens
- **prefix** – A prefix to the serial number

Returns serial number

Return type string

`privacyidea.lib.token.get_all_token_users(*args, **kws)`

return a dictionary with all tokens, that are assigned to users. This returns a dictionary with the key being the serial number of the token and the user information as dict.

Returns dictionary of serial numbers

Return type dict

`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(*args, **kws)`

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 server time
- **timestamp** (*int*) – Simulate the server time (unix time in seconds)

Returns dictionary of otp values

Return type dictionary

`privacyidea.lib.token.get_num_tokens_in_realm(*args, **kws)`

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_otp(*args, **kws)`

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.

Parameters

- **serial** – serial number of the token
- **current_time** (*datetime*) – a fake server time for testing of TOTP token

Returns tuple with (result, pin, otpval, passw)

Return type tuple

`privacyidea.lib.token.get_realms_of_token(*args, **kws)`

This function returns a list of the realms of a token

Parameters `serial` (*basestring*) – the serial number of the token

Returns list of the realm names

Return type list

`privacyidea.lib.token.get_serial_by_otp(*args, **kws)`

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(*args, **kws)`

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(*args, **kws)`

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

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(*args, **kws)`

Returns the tokentype of a given serial number

Parameters `serial` (*string*) – the serial number of the to be searched token

Returns tokentype

Return type string

`privacyidea.lib.token.get_tokenclass_info(*args, **kws)`

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(*args, **kws)`

(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 serial number of the token
- **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.

Returns A list of tokenclasses (lib.tokenclass)

Return type list

`privacyidea.lib.token.get_tokens_in_resolver(*args, **kws)`

Return a list of the token objects, 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(*args, **kws)`

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, whos token should be displayed
- **serial** –
- **active** –
- **resolver** –
- **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 ;-)

Returns dict with tokens, prev, next and count

Return type dict

`privacyidea.lib.token.get_tokenserial_of_transaction(*args, **kws)`
get the serial number of a token from a challenge state / transaction

Parameters **transaction_id** (*basestring*) – the state / transaction id

Returns the serial number or None

Return type basestring

`privacyidea.lib.token.init_token(*args, **kws)`
create a new token or update an existing token

Parameters

- **param** (*dict*) – initialization parameters like: serial (optional) type (optional, default=hotp) otpkey
- **user** (*User Object*) – the token owner
- **tokenrealms** (*list*) – the realms, to which the token should belong

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 Returns None, if the token does not exist.

Parameters **serial** (*basestring*) – The serial number of the token

Returns True or False

Return type bool

`privacyidea.lib.token.is_token_owner(*args, **kws)`
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(*args, **kws)`
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 :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(*args, **kws)`

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

Returns The number of deleted token

Return type int

`privacyidea.lib.token.reset_token(*args, **kws)`

Reset the failcounter :param serial: :param user: :return: The number of tokens, that were reset :rtype: int

`privacyidea.lib.token.resync_token(*args, **kws)`

Resynchronize the token of the given serial number by searching the otp1 and otp2 in the future otp values.

Parameters

- **serial** (*basestring*) – token serial number
- **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.set_count_auth(*args, **kws)`

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 modify
- **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(*args, **kws)`

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

- **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(*args, **kws)`

Set the default values for the token with the given serial number :param serial: token serial :type serial: basestring :return: None

`privacyidea.lib.token.set_description(*args, **kws)`

Set the description of a token

Parameters

- **serial** (*basestring*) – The serial number of the token
- **description** (*int*) – 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_hashlib(*args, **kws)`

Set the hashlib in the tokeninfo. Can be something like sha1, sha256...

Parameters

- **serial** (*basestring*) – The serial number of the token
- **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(*args, **kws)`

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
- **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(*args, **kws)`

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
- **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(*args, **kws)
```

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 used: User object :param serial: If the serial is specified, the PIN for this very token will be set. :return: The number of PINs set (usually 1) :rtype: int

```
privacyidea.lib.token.set_pin_so(*args, **kws)
```

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
- **so_pin** – The Security Officer PIN

Returns The number of SO PINs set. (usually 1)

Return type int

```
privacyidea.lib.token.set_pin_user(*args, **kws)
```

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
- **user_pin** (*basestring*) – The user PIN

Returns The number of PINs set (usually 1)

Return type int

```
privacyidea.lib.token.set_realms(*args, **kws)
```

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.

Thus, setting realms=[] clears all realms assignments.

Parameters

- **serial** (*basestring*) – the serial number of the token
- **realms** (*list*) – A list of realm names

Returns the number of tokens, to which realms where added. As a serial

number should be unique, this is either 1 or 0. :rtype: int

```
privacyidea.lib.token.set_sync_window(*args, **kws)
```

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
- **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(*args, **kws)`

Set the validity period for the given token.

Parameters

- **serial** –
- **user** –
- **end** (*basestring*) – Timestamp in the format DD/MM/YY HH:MM

`privacyidea.lib.token.set_validity_period_start(*args, **kws)`

Set the validity period for the given token.

Parameters

- **serial** –
- **user** –
- **start** (*basestring*) – Timestamp in the format DD/MM/YY HH:MM

`privacyidea.lib.token.token_exist(*args, **kws)`

returns true if the token with the given serial number exists

Parameters **serial** – the serial number of the token

`privacyidea.lib.token.token_has_owner(*args, **kws)`

returns true if the token is owned by any user

`privacyidea.lib.token.unassign_token(*args, **kws)`

unassign the user from the token

Parameters **serial** – The serial number of the token to unassign

Returns True

Application Class

`privacyidea.lib.applications.MachineApplicationBase`

alias of 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.

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`.

class `privacyidea.lib.policy.ACTION`

This is the list of usual actions.

ADDUSER = `'adduser'`

APIKEY = `'api_key_required'`

ASSIGN = `'assign'`

AUDIT = `'auditlog'`

AUTHITEMS = `'fetch_authentication_items'`

AUTOASSIGN = `'autoassignment'`

CACONNECTORDELETE = `'caconnectordelete'`

CACONNECTORREAD = `'caconnectorread'`

CACONNECTORWRITE = `'caconnectorwrite'`

COPYTOKENPIN = `'copytokenpin'`

COPYTOKENUSER = `'copytokenuser'`

DELETE = `'delete'`

DELETEUSER = `'deleteuser'`

DISABLE = `'disable'`

ENABLE = `'enable'`

ENCRYPTPIN = `'encrypt_pin'`

GETSERIAL = `'getserial'`

IMPORT = `'importtokens'`

LOGINMODE = `'login_mode'`

LOGOUTTIME = `'logout_time'`

```

LOSTTOKEN = 'losttoken'
LOSTTOKENPWCONTENTS = 'losttoken_PW_contents'
LOSTTOKENPWLEN = 'losttoken_PW_length'
LOSTTOKENVALID = 'losttoken_valid'
MACHINELIST = 'machinelist'
MACHINERESOLVERDELETE = 'mresolverdelete'
MACHINERESOLVERWRITE = 'mresolverwrite'
MACHINETOKENS = 'manage_machine_tokens'
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'
PASSNOTOKEN = 'passOnNoToken'
PASSNOUSER = 'passOnNoUser'
PASSTHRU = 'passthru'
POLICYDELETE = 'policydelete'
POLICYWRITE = 'policywrite'
RESET = 'reset'
RESOLVERDELETE = 'resolverdelete'
RESOLVERWRITE = 'resolverwrite'
RESYNC = 'resync'
SERIAL = 'serial'
SET = 'set'
SETPIN = 'setpin'
SETREALM = 'setrealm'
SYSTEMDELETE = 'configdelete'
SYSTEMWRITE = 'configwrite'
TOKENLABEL = 'tokenlabel'
TOKENREALMS = 'tokenrealms'
TOKENTYPE = 'tokentype'
UNASSIGN = 'unassign'

```

UPDATEUSER = 'updateuser'

USERLIST = 'userlist'

class `privacyidea.lib.policy.ACTIONVALUE`

This is a list of usual action values for e.g. policy action-values like otpin.

DISABLE = 'disable'

NONE = 'none'

TOKENPIN = 'tokenpin'

USERSTORE = 'userstore'

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.PolicyClass`

The Policy_Object will contain all database policy entries for easy filtering and mangling. It will be created at the beginning of the request and is supposed to stay alive unchanged during the request.

get_action_values (*action, scope='authorization', realm=None, resolver=None, user=None, client=None, unique=False, allow_white_space_in_action=False*)

Get the defined action values for a certain action like scope: authorization action: tokentype

would return a list of the tokentypes

scope: authorization action: serial

would return a list of allowed serials

Parameters

- **unique** – if set, the function will raise an exception if more than one value is returned
- **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!

Returns A list of the allowed tokentypes

Return type list

get_policies (*name=None, scope=None, realm=None, active=None, resolver=None, user=None, client=None, action=None, adminrealm=None*)

Return the policies of the given filter values

Parameters

- **name** –
- **scope** –
- **realm** –
- **active** –
- **resolver** –
- **user** –

- **client** –
- **action** –
- **adminrealm** – This is the realm of the admin. This is only evaluated in the scope admin.

Returns list of policies

Return type list of dicts

ui_get_enroll_tokenypes (*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: clas-
sical 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 Password to the users
email address”, “sms”: “SMS: Send a One Time Password to the users mobile phone”, “cer-
tificate”: “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

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’

GETTOKEN = ‘gettoken’

USER = ‘user’

WEBUI = ‘webui’

`privacyidea.lib.policy.delete_policy` (**args, **kws*)

Function to delete one named policy

Parameters **name** – the name of the policy to be deleted

Returns the count of the deleted policies.

Return type int

`privacyidea.lib.policy.enable_policy` (**args, **kws*)

Enable or disable the policy with the given name :param name: :return: ID of the policy

`privacyidea.lib.policy.export_policies` (**args, **kws*)

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_static_policy_definitions(*args, **kws)`

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(*args, **kws)`

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(*args, **kws)`

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”, “system”, “authentication” :param action: A scope specific action or a comma seperated 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 :return: The database ID of the policy :rtype: int

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.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.check_base_action(request=None, action=None)`

This decorator function takes the request and verifies the given action for the SCOPE ADMIN or USER. :param req: :param action: :return: 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

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 in the SCOPE.USER. It is used to decorate the API functions.

The pin is investigated in the params as `pin = params.get("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.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_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>" }

It uses the policy SCOPE.ENROLL, ACTION.TOKENLABEL to set the tokenlabel of Smartphone tokens during enrollment and this fill the details of the response.

`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.

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.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. It 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

- **req** (*Request Object*) – The request that is intercepted during the API call
- **action** (*basestring*) – An optional Action

Returns Always true. Modified the parameter request

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.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_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.get_logout_time(request, response)`

This decorator is used in the /auth API to add configuration information like the logout time to the response.

:param request: flask request object :param response: flask response object :return: the response

`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 to call a specific function after the decorated function. The `postpolicy` decorator is to be used in the API calls.

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_otppin(wrapped_function, *args, **kwargs)`

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 `**kwargs`: `kwargs["options"]` contains the flask g :return: True or False

```
privacyidea.lib.policydecorators.auth_user_does_not_exist (wrapped_function,
                                                         user_object,    passwd,
                                                         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, passwd, options={ })

Parameters

- **wrapped_function** –
- **user_object** –
- **passwd** –
- **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,    passwd,
                                                         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, passwd, options={ })

Parameters

- **wrapped_function** –
- **user_object** –
- **passwd** –
- **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,
                                                         passwd, 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, passwd, options={ })

Parameters

- **wrapped_function** –
- **user_object** –
- **passwd** –
- **options** – Dict containing values for “g” and “clientip”

Returns Tuple of True/False and reply-dictionary

```
privacyidea.lib.policydecorators.config_lost_token (wrapped_function, *args,
                                                         **kwargs)
```

Decorator to decorate the lib.token.lost_token function. Depending on ACTION.LOSTTOKENINVALID, 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

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

classmethod getResolverClassDescriptor()

return the descriptor of the resolver, which is - the class name and - the config description

Returns resolver description dict

Return type dict

classmethod getResolverClassType()

provide the resolver type for registration

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

classmethod getResolverType()

getResolverType - return the type of the resolver

Returns returns the string 'Idapresolver'

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 empty string, if the user does not exist.

Parameters **loginName** (*string*) – The login name of the user

Returns The ID of the user

Return type string or int

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

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.”

param param: The parameters that should be saved as the resolver type param: dict return: returns True in case of success and a descriptive text rtype: 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

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

getResolverId ()
return the resolver identifier string, which in fact is filename, where it points to.

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

Returns the userId

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*)
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: string

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.

classmethod 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`

checkPass (*uid, password*)

This function checks the password for a given uid. - returns true in case of success - false if password does not match

classmethod **getResolverClassDescriptor** ()

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.

getUserId (*LoginName*)

resolve the loginname to the userid.

Parameters **LoginName** (*string*) – The login name from the credentials

Returns UserId as found for the LoginName

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*)

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

classmethod **get_serverpool** (*urilist, timeout*)

This create the serverpool for the ldap3 connection. The URI from the LDAP resolver can contain a comma seperated 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 seperated
- **timeout** (*float*) – The connection timeout

Returns Server Pool

Return type LDAP3 Server Pool Instance

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_userfilter': 'LDAPFILTER', '#ldap_mapping': 'USERINFO', '#ldap_uidtype': 'UIDTYPE',
'#ldap_noreferrals': 'NOREFERRALS', '#ldap_certificate': 'CACERTIFICATE',
```

classmethod `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.

This is taken from controllers/admin.py

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, LDAPFILTER, USERINFO, SIZELIMIT, NOREFERRALS, CACERTIFICATE, AUTHTYPE

Audit log

Base class

class `privacyidea.lib.auditmodules.base.Audit (config=None)`

add_to_log (param)

Add to existing log entry :param param: :return:

audit_entry_to_dict (audit_entry)

If the searchQuery returns an iteretor 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)

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_total (param, AND=True, display_error=True)

This method returns the total number of audit entries in the audit store

initialize (*args, **kws)

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.

log (*args, **kws)

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.

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 (*args, **kws)

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 (param, display_error=True, rp_dict=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, rp_dict)

This function returns the audit log as an iterator on the result

set ()

This function could be used to set certain things like the signing key. But maybe it should only be read from privacyidea.ini?

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. PI_AUDIT_SQL_URI

add_to_log (param)

Add new text to an existing log entry :param param: :return:

clear ()

Deletes all entries in the database table. This is only used for test cases! :return:

csv_generator (param=None, user=None)

Returns the audit log as csv file. :param config: The current flask app configuration :type config: dict
: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_total (param, AND=True, display_error=True)

This method returns the total number of audit entries in the audit store

log (*param*)

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

search (*search_dict*, *page_size=15*, *page=1*, *sortorder='asc'*)

This function returns the audit log as a Pagination object.

searchQuery (*search_dict*, *page_size=15*, *page=1*, *sortorder='asc'*, *sortname='number'*)

This function returns the audit log as an iterator on the result

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*)

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 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

classmethod `testconnection` (*params*)

This method can test if the passed parameters would create a working machine resolver.

Parameters `params` –

Returns tuple of success and description

Return type (bool, string)

Hosts Machine Resolver

class `privacyidea.lib.machines.hosts.HostsMachineResolver` (*name, config=None*)

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

classmethod `testconnection` (*params*)

Test if the given filename exists.

Parameters `params` –

Returns

1.13.3 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.py`.

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.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. Therefore 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` (**args, **kwargs*)

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.Config` (**args, **kwargs*)

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.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` (**args, **kwds*)

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.Policy` (*name, active=True, scope='', action='', realm='', admin-realm='', resolver='', user='', client='', time='', condition=0*)

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 :rtype: dict or value

class `privacyidea.models.Realm` (**args, **kwds*)

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.Token(serial, tokentype=u'', isactive=True, otpplen=6, otpkey=u'',
                               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 (**args, **kws*)

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: “

get_info ()

Returns The token info as dictionary

get_realms ()

return a list of the assigned realms :return: realms :rtype: list

get_user_pin (**args, **kws*)

return the userPin :rtype : the PIN as a secretObject

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*)

Set the list of the realms. This is done by filling the tokenrealm table. :param realms: realms :type realms: list

set_so_pin (*soPin*)

For smartcards this sets the security officer pin of the token

:rtype : None

split_pin_pass (*passwd*, *prepend=True*)

The password is split into the PIN and the OTP component. The token knows its length, so it can split accordingly.

Parameters

- **passwd** – The password that is to be split
- **prepend** – The PIN is put in front of the OTP value

Returns tuple of (res, pin, otpval)

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.TokenRealm` (*realm_id=0*, *token_id=0*, *realmname=None*)

This table stored to wich 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.

`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 :rtype: int

1.14 Frequently Asked Questions

1.14.1 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 itself does not manage users and therefore you do not need to create users.

You very much likely already have an application (like your VPN or a Web Application...) for which you want to increase the login 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.

So there is no sense in creating the user in the application **and** in privacyIDEA. Right?

This is why you can not create users in privacyIDEA but you only need to tell privacyIDEA where the users are located and you can start enrolling tokens to those users.

Please read the sections *UserIdResolvers* and *Userview* for more details.

1.14.2 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.py` to manage those admins from the command line as the system's root user. (see *Installation*)

These admin users can login 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: Use this carefully. Imagine you defined a resolver to a specific group in your Active Directory to be the privacyIDEA 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.

1.14.3 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.

Note: Some parts are marked as “**(TODO)** Not yet implemented”. These are components that have not been migrated from 1.5 to 2.0. If you are missing an important, not-yet-migrated part, drop us a note!

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](mailto:info(@)privacyidea.org) or go to the [Google group](#).

This will help us a lot to improve documentation to your needs.

Thanks a lot!

Indices and tables

- `genindex`
- `modindex`
- `search`

/application

GET /application/, 114

/audit

GET /audit/, 83

GET /audit/(csvfile), 83

/auth

GET /auth/rights, 84

POST /auth, 84

/defaultrealm

GET /defaultrealm, 94

POST /defaultrealm/(realm), 94

DELETE /defaultrealm, 93

/machine

GET /machine/, 112

GET /machine/authitem, 111

GET /machine/authitem/(application), 114

GET /machine/token, 112

POST /machine/token, 112

POST /machine/tokenoption, 111

DELETE /machine/token/(serial)/(machineid)/(resolver)/(application), 113

/machineresolver

GET /machineresolver/, 110

GET /machineresolver/(resolver), 111

POST /machineresolver/(resolver), 110

POST /machineresolver/test, 110

DELETE /machineresolver/(resolver), 111

/policy

GET /policy, 105

GET /policy/(name), 108

GET /policy/check, 103

GET /policy/defs, 104

GET /policy/defs/(scope), 107

GET /policy/export/(export), 106

POST /policy/(name), 107

POST /policy/disable/(name), 106

POST /policy/enable/(name), 106

POST /policy/import/(filename), 107

DELETE /policy/(name), 109

/realm

GET /realm/, 91

GET /realm/superuser, 91

POST /realm/(realm), 92

DELETE /realm/(realm), 93

/resolver

GET /resolver/, 90

GET /resolver/(resolver), 90

POST /resolver/(resolver), 90

POST /resolver/test, 90

DELETE /resolver/(resolver), 90

/system

GET /system/, 89

GET /system/(key), 89

POST /system/setConfig, 89

POST /system/setDefault, 88

DELETE /system/(key), 89

/token

GET /token/, 98

/token/(serial)

DELETE /token/(serial), 102

/token/assign

POST /token/assign, 95

/token/copypin

POST /token/copypin, 95

/token/copyuser

POST /token/copyuser, 94

/token/disable

POST /token/disable, [95](#)

POST /token/disable/(serial), [99](#)

/token/enable

POST /token/enable, [95](#)

POST /token/enable/(serial), [99](#)

/token/getserial

GET /token/getserial/(otp), [98](#)

/token/init

POST /token/init, [96](#)

/token/load

POST /token/load/(filename), [101](#)

/token/lost

POST /token/lost/(serial), [101](#)

/token/realm

POST /token/realm/(serial), [100](#)

/token/reset

POST /token/reset, [96](#)

POST /token/reset/(serial), [100](#)

/token/resync

POST /token/resync, [96](#)

POST /token/resync/(serial), [99](#)

/token/set

POST /token/set, [98](#)

POST /token/set/(serial), [101](#)

/token/setpin

POST /token/setpin, [96](#)

POST /token/setpin/(serial), [100](#)

/token/unassign

POST /token/unassign, [94](#)

/user

GET /user/, [102](#)

POST /user/, [102](#)

PUT /user/, [103](#)

DELETE /user/(resolvername)/(username),
[103](#)

/validate

GET /validate/check, [87](#)

GET /validate/samlcheck, [86](#)

POST /validate/check, [88](#)

POST /validate/samlcheck, [86](#)

p

- [privacyidea.api](#), 82
- [privacyidea.api.application](#), 114
- [privacyidea.api.auth](#), 84
- [privacyidea.api.lib.postpolicy](#), 157
- [privacyidea.api.lib.prepolicy](#), 155
- [privacyidea.api.machine](#), 111
- [privacyidea.api.machineresolver](#), 110
- [privacyidea.api.policy](#), 103
- [privacyidea.api.realm](#), 91
- [privacyidea.api.resolver](#), 90
- [privacyidea.api.system](#), 88
- [privacyidea.api.token](#), 94
- [privacyidea.api.user](#), 102
- [privacyidea.api.validate](#), 85
- [privacyidea.lib](#), 115
 - [privacyidea.lib.auditmodules](#), 165
 - [privacyidea.lib.machines](#), 167
 - [privacyidea.lib.policy](#), 150
 - [privacyidea.lib.policydecorators](#), 158
 - [privacyidea.lib.resolvers](#), 160
 - [privacyidea.lib.token](#), 140
 - [privacyidea.lib.user](#), 115
- [privacyidea.models](#), 169

A

- ACTION (class in `privacyidea.lib.policy`), 151
- ACTIONVALUE (class in `privacyidea.lib.policy`), 153
- Active Directory, 17, 19
- Add User, 54, 59
- `add_init_details()` (`privacyidea.lib.tokenclass.TokenClass` method), 132
- `add_to_log()` (`privacyidea.lib.auditmodules.base.Audit` method), 165
- `add_to_log()` (`privacyidea.lib.auditmodules.sqlaudit.Audit` method), 166
- `add_tokeninfo()` (in module `privacyidea.lib.token`), 140
- `add_tokeninfo()` (`privacyidea.lib.tokenclass.TokenClass` method), 132
- `add_user()` (`privacyidea.lib.resolvers.UserIdResolver.UserIdResolver` method), 160
- ADDUSER (`privacyidea.lib.policy.ACTION` attribute), 151
- Admin (class in `privacyidea.models`), 169
- ADMIN (`privacyidea.lib.policy.SCOPE` attribute), 154
- admin accounts, 173
- admin policies, 56
- admin realm, 56
- API, 82
- `api_key_required()` (in module `privacyidea.api.lib.prepolicy`), 155
- APIKEY (`privacyidea.lib.policy.ACTION` attribute), 151
- appliance, 41
- Application Plugins, 75
- ASSIGN (`privacyidea.lib.policy.ACTION` attribute), 151
- `assign_token()` (in module `privacyidea.lib.token`), 140
- Audit, 72
- Audit (class in `privacyidea.lib.auditmodules.base`), 165
- Audit (class in `privacyidea.lib.auditmodules.sqlaudit`), 166
- AUDIT (`privacyidea.lib.policy.ACTION` attribute), 151
- AUDIT (`privacyidea.lib.policy.SCOPE` attribute), 154
- audit modules, 165
- `audit_entry_to_dict()` (`privacyidea.lib.auditmodules.base.Audit` method), 165
- AUTH (`privacyidea.lib.policy.SCOPE` attribute), 154
- `auth_otppin()` (in module `privacyidea.lib.policydecorators`), 158
- `auth_user_does_not_exist()` (in module `privacyidea.lib.policydecorators`), 158
- `auth_user_has_no_token()` (in module `privacyidea.lib.policydecorators`), 159
- `auth_user_passthru()` (in module `privacyidea.lib.policydecorators`), 159
- `authenticate()` (`privacyidea.lib.tokenclass.TokenClass` method), 132
- `authenticate()` (`privacyidea.lib.tokens.remotetoken.RemoteTokenClass` method), 125
- `authenticate()` (`privacyidea.lib.tokens.spasstoken.SpasmTokenClass` method), 128
- authenticating client, 27
- authentication policies, 64
- AUTHITEMS (`privacyidea.lib.policy.ACTION` attribute), 151
- authorization policies, 66
- AUTHZ (`privacyidea.lib.policy.SCOPE` attribute), 154
- `auto_assign_token()` (in module `privacyidea.lib.token`), 140
- AUTOASSIGN (`privacyidea.lib.policy.ACTION` attribute), 151
- `autoassign()` (in module `privacyidea.api.lib.postpolicy`), 157
- autoassignment, 69
- autoresync, 25

B

- Backup, 44
- BaseMachineResolver (class in `privacyidea.lib.machines.base`), 167

C

- CA, 29, 41
- CACConnector (class in `privacyidea.models`), 169
- CACConnectorConfig (class in `privacyidea.models`), 169

CACONNECTORDELETE (privacyidea.lib.policy.ACTION attribute), 151

CACONNECTORREAD (privacyidea.lib.policy.ACTION attribute), 151

caconnectors, 41

CACONNECTORWRITE (privacyidea.lib.policy.ACTION attribute), 151

Certificate Authority, 41

certificate token, 41

certificates, 29

CertificateTokenClass (class in privacyidea.lib.tokens.certificatetoken), 116

Challenge (class in privacyidea.models), 169

challenge_janitor() (privacyidea.lib.tokenclass.TokenClass method), 132

Change User Password, 54

check_auth_counter() (privacyidea.lib.tokenclass.TokenClass method), 133

check_base_action() (in module privacyidea.api.lib.prepolicy), 155

check_challenge_response() (privacyidea.lib.tokenclass.TokenClass method), 133

check_external() (in module privacyidea.api.lib.prepolicy), 155

check_failcount() (privacyidea.lib.tokenclass.TokenClass method), 133

check_max_token_realm() (in module privacyidea.api.lib.prepolicy), 156

check_max_token_user() (in module privacyidea.api.lib.prepolicy), 156

check_otp() (privacyidea.lib.tokenclass.TokenClass method), 133

check_otp() (privacyidea.lib.tokens.daplugtoken.DaplugTokenClass method), 117

check_otp() (privacyidea.lib.tokens.emailtoken.EmailTokenClass method), 119

check_otp() (privacyidea.lib.tokens.hotptoken.HotpTokenClass method), 119

check_otp() (privacyidea.lib.tokens.motptoken.MotpTokenClass method), 121

check_otp() (privacyidea.lib.tokens.passwordtoken.PasswordTokenClass method), 122

check_otp() (privacyidea.lib.tokens.radiustoken.RadiusTokenClass method), 123

check_otp() (privacyidea.lib.tokens.registrationtoken.RegistrationTokenClass method), 124

check_otp() (privacyidea.lib.tokens.remotetoken.RemoteTokenClass method), 125

check_otp() (privacyidea.lib.tokens.sms token.SmsTokenClass method), 127

check_otp() (privacyidea.lib.tokens.spas token.SpasmTokenClass method), 128

check_otp() (privacyidea.lib.tokens.totp token.TotpTokenClass method), 129

check_otp() (privacyidea.lib.tokens.yubicotoken.YubicoTokenClass method), 131

check_otp() (privacyidea.lib.tokens.yubikeytoken.YubikeyTokenClass method), 131

check_otp_exist() (privacyidea.lib.tokenclass.TokenClass method), 133

check_otp_exist() (privacyidea.lib.tokens.daplugtoken.DaplugTokenClass method), 118

check_otp_exist() (privacyidea.lib.tokens.hotptoken.HotpTokenClass method), 120

check_otp_exist() (privacyidea.lib.tokens.totp token.TotpTokenClass method), 129

check_otp_exist() (privacyidea.lib.tokens.yubikeytoken.YubikeyTokenClass method), 131

check_otp_pin() (in module privacyidea.api.lib.prepolicy), 156

check_pin() (privacyidea.lib.tokenclass.TokenClass method), 133

check_pin_local (privacyidea.lib.tokens.radius token.RadiusTokenClass attribute), 123

check_pin_local (privacyidea.lib.tokens.remotetoken.RemoteTokenClass attribute), 125

check_serial() (in module privacyidea.api.lib.postpolicy), 157

check_serial() (in module privacyidea.lib.token), 141

check_serial_pass() (in module privacyidea.lib.token), 141

check_token_init() (in module privacyidea.api.lib.prepolicy), 156

check_token_list() (in module privacyidea.lib.token), 141

check_token_upload() (in module privacyidea.api.lib.prepolicy), 156

check_tokentype() (in module privacyidea.api.lib.postpolicy), 157

check_token_pass() (in module privacyidea.lib.token), 141

check_validity_period() (privacyidea.lib.tokenclass.TokenClass method), 134

checkPass() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver method), 164

checkPass() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver method), 162

checkPass() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver method), 160

checkUserId() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver method), 160

- method), 162
- checkUserName() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver method), 163
- cleanup_challenges() (in module privacyidea.models), 172
- clear() (privacyidea.lib.auditmodules.sqlaudit.Audit method), 166
- Clickatel, 39
- client, 27
- client certificates, 29
- client machines, 72
- client policies, 60
- close() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver method), 161
- Config (class in privacyidea.models), 169
- config file, 9
- config_lost_token() (in module privacyidea.lib.policydecorators), 159
- configuration, 17
- copy_token_pin() (in module privacyidea.lib.token), 141
- copy_token_realms() (in module privacyidea.lib.token), 142
- copy_token_user() (in module privacyidea.lib.token), 142
- COPYTOKENPIN (privacyidea.lib.policy.ACTION attribute), 151
- COPYTOKENUSER (privacyidea.lib.policy.ACTION attribute), 151
- count window, 49
- create_challenge() (privacyidea.lib.tokenclass.TokenClass method), 134
- create_challenge() (privacyidea.lib.tokens.emailtoken.EmailTokenClass method), 119
- create_challenge() (privacyidea.lib.tokens.sms token.SmsTokenClass method), 127
- create_tokenclass_object() (in module privacyidea.lib.token), 142
- create_user() (in module privacyidea.lib.user), 115
- CSR, 29
- csv_generator() (privacyidea.lib.auditmodules.base.Audit method), 165
- csv_generator() (privacyidea.lib.auditmodules.sqlaudit.Audit method), 166
- D**
- DaplugTokenClass (class in privacyidea.lib.tokens.daplugtoken), 117
- database, 169
- DB2, 21
- debug, 9
- Debugging, 10
- default realm, 23
- del_info() (privacyidea.models.Token method), 171
- del_tokeninfo() (privacyidea.lib.tokenclass.TokenClass method), 134
- DELETE (privacyidea.lib.policy.ACTION attribute), 151
- Delete User, 59
- delete_policy() (in module privacyidea.lib.policy), 154
- delete_token() (privacyidea.lib.tokenclass.TokenClass method), 134
- delete_user() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver method), 161
- DELETEUSER (privacyidea.lib.policy.ACTION attribute), 151
- DISABLE (privacyidea.lib.policy.ACTION attribute), 151
- DISABLE (privacyidea.lib.policy.ACTIONVALUE attribute), 153
- DISABLE (privacyidea.lib.policy.LOGINMODE attribute), 153
- E**
- Edit User, 54, 59, 64
- Edit Users, 54
- Editable Resolver, 54
- Email policy, 66
- Email policy, 66
- Email subject, 66
- Email text, 66
- Email Token, 36
- EMAIL_ADDRESS_KEY (privacyidea.lib.tokens.emailtoken.EmailTokenClass attribute), 119
- EmailTokenClass (class in privacyidea.lib.tokens.emailtoken), 119
- ENABLE (privacyidea.lib.policy.ACTION attribute), 151
- enable() (privacyidea.lib.tokenclass.TokenClass method), 134
- enable_policy() (in module privacyidea.lib.policy), 154
- enable_token() (in module privacyidea.lib.token), 142
- encrypt_pin() (in module privacyidea.api.lib.prepolicy), 156
- ENCRYPTPIN (privacyidea.lib.policy.ACTION attribute), 151
- ENROLL (privacyidea.lib.policy.SCOPE attribute), 154
- enroll token, 50
- enrollment policies, 68
- export_policies() (in module privacyidea.lib.policy), 154
- external hook, 9
- F**
- failcount, 49
- FAQ, 173
- finalize_log() (privacyidea.lib.auditmodules.base.Audit method), 165

finalize_log() (privacyidea.lib.auditmodules.sqlaudit.Audit method), 166
 flatfile resolver, 19
 FreeIPA, 19
 FreeRADIUS, 75

G

gen_serial() (in module privacyidea.lib.token), 142
 Get Serial (Determine Serial by OTP), 47
 get() (privacyidea.models.Challenge method), 169
 get() (privacyidea.models.Policy method), 170
 get() (privacyidea.models.Token method), 171
 get_action_values() (privacyidea.lib.policy.PolicyClass method), 153
 get_all_token_users() (in module privacyidea.lib.token), 143
 get_as_dict() (privacyidea.lib.tokenclass.TokenClass method), 134
 get_audit_id() (privacyidea.lib.auditmodules.base.Audit method), 165
 get_class_info() (privacyidea.lib.tokenclass.TokenClass class method), 134
 get_class_info() (privacyidea.lib.tokens.certificatetoken.CertificateTokenClass class method), 117
 get_class_info() (privacyidea.lib.tokens.daplugtoken.DaplugTokenClass class method), 118
 get_class_info() (privacyidea.lib.tokens.emailtoken.EmailTokenClass class method), 119
 get_class_info() (privacyidea.lib.tokens.hotptoken.HotpTokenClass class method), 120
 get_class_info() (privacyidea.lib.tokens.motptoken.MotpTokenClass class method), 122
 get_class_info() (privacyidea.lib.tokens.passwordtoken.PasswordTokenClass class method), 122
 get_class_info() (privacyidea.lib.tokens.radiusTokenClass class method), 123
 get_class_info() (privacyidea.lib.tokens.registrationtoken.RegistrationTokenClass class method), 124
 get_class_info() (privacyidea.lib.tokens.remotetoken.RemoteTokenClass class method), 125
 get_class_info() (privacyidea.lib.tokens.smsTokenClass class method), 127
 get_class_info() (privacyidea.lib.tokens.spasTokenClass class method), 130

class method), 128
 get_class_info() (privacyidea.lib.tokens.sshkeytoken.SSHkeyTokenClass class method), 129
 get_class_info() (privacyidea.lib.tokens.totptoken.TotpTokenClass class method), 129
 get_class_info() (privacyidea.lib.tokens.yubicotoken.YubicoTokenClass class method), 131
 get_class_info() (privacyidea.lib.tokens.yubikeytoken.YubikeyTokenClass class method), 131
 get_class_prefix() (privacyidea.lib.tokenclass.TokenClass class method), 134
 get_class_prefix() (privacyidea.lib.tokens.certificatetoken.CertificateTokenClass class method), 117
 get_class_prefix() (privacyidea.lib.tokens.daplugtoken.DaplugTokenClass class method), 118
 get_class_prefix() (privacyidea.lib.tokens.emailtoken.EmailTokenClass class method), 119
 get_class_prefix() (privacyidea.lib.tokens.hotptoken.HotpTokenClass class method), 120
 get_class_prefix() (privacyidea.lib.tokens.motptoken.MotpTokenClass class method), 122
 get_class_prefix() (privacyidea.lib.tokens.passwordtoken.PasswordTokenClass class method), 123
 get_class_prefix() (privacyidea.lib.tokens.radiusTokenClass class method), 123
 get_class_prefix() (privacyidea.lib.tokens.registrationtoken.RegistrationTokenClass class method), 124
 get_class_prefix() (privacyidea.lib.tokens.remotetoken.RemoteTokenClass class method), 125
 get_class_prefix() (privacyidea.lib.tokens.smsTokenClass class method), 128
 get_class_prefix() (privacyidea.lib.tokens.spasTokenClass class method), 128
 get_class_prefix() (privacyidea.lib.tokens.sshkeytoken.SSHkeyTokenClass class method), 129
 get_class_prefix() (privacyidea.lib.tokens.totptoken.TotpTokenClass class method), 130

[get_class_prefix\(\)](#) (privacyidea.lib.tokens.yubicotoken.YubicoTokenClass class method), [131](#)
[get_class_prefix\(\)](#) (privacyidea.lib.tokens.yubikeytoken.YubikeyTokenClass class method), [132](#)
[get_class_type\(\)](#) (privacyidea.lib.tokenclass.TokenClass class method), [134](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.certificatetoken.CertificateTokenClass class method), [117](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.daplugtoken.DaplugTokenClass class method), [118](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.emailtoken.EmailTokenClass class method), [119](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.hotptoken.HotpTokenClass class method), [120](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.motptoken.MotpTokenClass class method), [122](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.passwordtoken.PasswordTokenClass class method), [123](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.radiusTokenClass class method), [123](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.registrationtoken.RegistrationTokenClass class method), [124](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.remotetoken.RemoteTokenClass class method), [125](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.smsTokenClass class method), [128](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.spasTokenClass class method), [128](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.sshkeytoken.SSHkeyTokenClass class method), [129](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.totptoken.TotpTokenClass class method), [130](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.yubicotoken.YubicoTokenClass class method), [131](#)
[get_class_type\(\)](#) (privacyidea.lib.tokens.yubikeytoken.YubikeyTokenClass class method), [132](#)
[get_config_description\(\)](#) (privacyidea.lib.machines.base.BaseMachineResolver class method), [167](#)
[get_count_auth\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [135](#)
[get_count_auth_max\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [135](#)
[get_count_auth_success\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [135](#)
[get_count_auth_success_max\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [135](#)
[get_count_window\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [135](#)
[get_dynamic_policy_definitions\(\)](#) (in module privacyidea.lib.token), [143](#)
[get_failcount\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [135](#)
[get_hashed_pin\(\)](#) (privacyidea.models.Token method), [171](#)
[get_hashlib\(\)](#) (privacyidea.lib.tokenclass.TokenClass class method), [135](#)
[get_info\(\)](#) (privacyidea.models.Token method), [171](#)
[get_init_detail\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [135](#)
[get_init_detail\(\)](#) (privacyidea.lib.tokens.certificatetoken.CertificateTokenClass method), [117](#)
[get_init_detail\(\)](#) (privacyidea.lib.tokens.hotptoken.HotpTokenClass method), [120](#)
[get_init_detail\(\)](#) (privacyidea.lib.tokens.motptoken.MotpTokenClass method), [122](#)
[get_init_detail\(\)](#) (privacyidea.lib.tokens.registrationtoken.RegistrationTokenClass method), [124](#)
[get_init_details\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [135](#)
[get_logout_time\(\)](#) (in module privacyidea.api.lib.postpolicy), [158](#)
[get_machine_id\(\)](#) (privacyidea.lib.machines.base.BaseMachineResolver method), [167](#)
[get_machine_id\(\)](#) (privacyidea.lib.machines.hosts.HostsMachineResolver method), [168](#)
[get_machineresolver_id\(\)](#) (in module privacyidea.models), [172](#)
[get_machines\(\)](#) (privacyidea.lib.machines.base.BaseMachineResolver method), [167](#)
[get_machines\(\)](#) (privacyidea.lib.machines.hosts.HostsMachineResolver method), [168](#)

method), 168

get_machinetoken_id() (in module privacyidea.models), 172

get_max_failcount() (privacyidea.lib.tokenclass.TokenClass method), 135

get_multi_otp() (in module privacyidea.lib.token), 143

get_multi_otp() (privacyidea.lib.tokenclass.TokenClass method), 135

get_multi_otp() (privacyidea.lib.tokens.daplugtoken.DaplugTokenClass method), 118

get_multi_otp() (privacyidea.lib.tokens.hotptoken.HotpTokenClass method), 120

get_multi_otp() (privacyidea.lib.tokens.totptoken.TotpTokenClass method), 130

get_num_tokens_in_realm() (in module privacyidea.lib.token), 143

get_otp() (in module privacyidea.lib.token), 143

get_otp() (privacyidea.lib.tokenclass.TokenClass method), 135

get_otp() (privacyidea.lib.tokens.daplugtoken.DaplugTokenClass method), 118

get_otp() (privacyidea.lib.tokens.hotptoken.HotpTokenClass method), 121

get_otp() (privacyidea.lib.tokens.totptoken.TotpTokenClass method), 130

get_otp_count() (privacyidea.lib.tokenclass.TokenClass method), 136

get_otp_count_window() (privacyidea.lib.tokenclass.TokenClass method), 136

get_otp_status() (privacyidea.models.Challenge method), 169

get_otplen() (privacyidea.lib.tokenclass.TokenClass method), 136

get_pin_hash_seed() (privacyidea.lib.tokenclass.TokenClass method), 136

get_policies() (privacyidea.lib.policy.PolicyClass method), 153

get_QRimage_data() (privacyidea.lib.tokenclass.TokenClass method), 134

get_realms() (privacyidea.lib.tokenclass.TokenClass method), 136

get_realms() (privacyidea.models.Token method), 171

get_realms_of_token() (in module privacyidea.lib.token), 143

get_serial() (privacyidea.lib.tokenclass.TokenClass method), 136

get_serial_by_otp() (in module privacyidea.lib.token), 144

get_serverpool() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver class method), 164

get_sshkey() (privacyidea.lib.tokens.sshkeytoken.SSHkeyTokenClass method), 129

get_static_policy_definitions() (in module privacyidea.lib.policy), 155

get_sync_timeout() (privacyidea.lib.tokens.hotptoken.HotpTokenClass method), 121

get_sync_window() (privacyidea.lib.tokenclass.TokenClass method), 136

get_token_by_otp() (in module privacyidea.lib.token), 144

get_token_id() (in module privacyidea.models), 172

get_token_owner() (in module privacyidea.lib.token), 144

get_token_type() (in module privacyidea.lib.token), 144

get_tokenclass_info() (in module privacyidea.lib.token), 144

get_tokeninfo() (privacyidea.lib.tokenclass.TokenClass method), 136

get_tokens() (in module privacyidea.lib.token), 145

get_tokens_in_resolver() (in module privacyidea.lib.token), 145

get_tokens_paginate() (in module privacyidea.lib.token), 145

get_tokenserial_of_transaction() (in module privacyidea.lib.token), 146

get_tokentype() (privacyidea.lib.tokenclass.TokenClass method), 136

get_total() (privacyidea.lib.auditmodules.base.Audit method), 165

get_total() (privacyidea.lib.auditmodules.sqlaudit.Audit method), 166

get_type() (privacyidea.lib.tokenclass.TokenClass method), 136

get_user() (privacyidea.lib.tokenclass.TokenClass method), 136

get_user_from_param() (in module privacyidea.lib.user), 115

get_user_id() (privacyidea.lib.tokenclass.TokenClass method), 136

get_user_info() (in module privacyidea.lib.user), 116

get_user_list() (in module privacyidea.lib.user), 116

get_user_pin() (privacyidea.models.Token method), 171

get_username() (in module privacyidea.lib.user), 116

get_validity_period_end() (privacyidea.lib.tokenclass.TokenClass method), 136

get_validity_period_start() (privacyidea.lib.tokenclass.TokenClass method), 136

get_vars() (privacyidea.lib.tokenclass.TokenClass method), 136

getResolverClassDescriptor() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver class method), 164

cyidea.lib.resolvers.LDAPIdResolver.IdResolver (method), 161
 class method), 164
 getResolverClassDescriptor() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver (class method), 163
 class method), 163
 getResolverClassDescriptor() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver (class method), 161
 class method), 161
 getResolverClassType() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver (class method), 161
 class method), 161
 getResolverDescriptor() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver (method), 161
 method), 161
 getResolverId() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver (method), 164
 method), 164
 getResolverId() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver (class in privacyidea.lib.resolvers.LDAPIdResolver), 164
 method), 163
 getResolverId() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver (class in privacyidea.lib.resolvers.PasswdIdResolver), 162
 method), 161
 getResolverType() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver (class method), 161
 class method), 161
 getSearchFields() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver (method), 163
 method), 163
 getserial, 59
 GETSERIAL (privacyidea.lib.policy.ACTION attribute), 151
 GETTOKEN (privacyidea.lib.policy.SCOPE attribute), 154
 getUserId() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver (method), 164
 method), 164
 getUserId() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver (method), 163
 method), 163
 getUserId() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver (method), 161
 method), 161
 getUserInfo() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver (method), 164
 method), 164
 getUserInfo() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver (method), 163
 method), 163
 getUserInfo() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver (method), 161
 method), 161
 getUserList() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver (method), 164
 method), 164
 getUserList() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver (method), 163
 method), 163
 getUserList() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver (method), 161
 method), 161
 getUsername() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver (method), 164
 method), 164
 getUsername() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver (method), 163
 method), 163
 getUsername() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver (method), 161
 method), 161

- method), 128
 - is_challenge_request() (privacyidea.lib.tokens.spasstoken.SpasmTokenClass method), 128
 - is_challenge_request() (privacyidea.lib.tokens.yubikeytoken.YubikeyTokenClass method), 132
 - is_challenge_response() (privacyidea.lib.tokenclass.TokenClass method), 137
 - is_challenge_response() (privacyidea.lib.tokens.spasstoken.SpasmTokenClass method), 128
 - is_token_active() (in module privacyidea.lib.token), 146
 - is_token_owner() (in module privacyidea.lib.token), 146
 - is_valid() (privacyidea.models.Challenge method), 169
- ## J
- JSON Web Token, 82
 - JWT, 82
- ## L
- LDAP, 17
 - LDAP resolver, 19
 - libpolicy (class in privacyidea.lib.policydecorators), 160
 - library, 115
 - load_config() (privacyidea.lib.machines.base.BaseMachineResolver method), 167
 - load_config() (privacyidea.lib.machines.hosts.HostsMachineResolver method), 168
 - loadConfig() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver method), 164
 - loadConfig() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver method), 163
 - loadConfig() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver method), 162
 - loadFile() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver method), 163
 - log() (privacyidea.lib.auditmodules.base.Audit method), 165
 - log() (privacyidea.lib.auditmodules.sqlaudit.Audit method), 166
 - log_token_num() (privacyidea.lib.auditmodules.base.Audit method), 166
 - Logging, 10
 - login mode, 70
 - Login Policy, 70
 - login_mode() (in module privacyidea.lib.policydecorators), 160
 - LOGINMODE (class in privacyidea.lib.policy), 153
 - LOGINMODE (privacyidea.lib.policy.ACTION attribute), 151
 - loglevel, 9
 - logout time, 71
 - LOGOUTTIME (privacyidea.lib.policy.ACTION attribute), 151
 - Lost token, 47
 - lost token, 70
 - lost_token() (in module privacyidea.lib.token), 146
 - LOSTTOKEN (privacyidea.lib.policy.ACTION attribute), 151
 - LOSTTOKENPWCONTENTS (privacyidea.lib.policy.ACTION attribute), 152
 - LOSTTOKENPWLEN (privacyidea.lib.policy.ACTION attribute), 152
 - LOSTTOKENVALID (privacyidea.lib.policy.ACTION attribute), 152
- ## M
- Machine Resolvers, 167
 - MachineApplicationBase (in module privacyidea.lib.applications), 150
 - MACHINELIST (privacyidea.lib.policy.ACTION attribute), 152
 - MachineResolver (class in privacyidea.models), 169
 - MachineResolverConfig (class in privacyidea.models), 170
 - MACHINERESOLVERDELETE (privacyidea.lib.policy.ACTION attribute), 152
 - MACHINERESOLVERWRITE (privacyidea.lib.policy.ACTION attribute), 152
 - Machines, 72
 - MachineToken (class in privacyidea.models), 170
 - MachineTokenOptions (class in privacyidea.models), 170
 - MACHINETOKENS (privacyidea.lib.policy.ACTION attribute), 152
 - maxfail, 49
 - MAXTOKENREALM (privacyidea.lib.policy.ACTION attribute), 152
 - MAXTOKENUSER (privacyidea.lib.policy.ACTION attribute), 152
 - MethodsMixin (class in privacyidea.models), 170
 - Migration, 31
 - mock_fail() (in module privacyidea.api.lib.prepolicy), 157
 - mock_success() (in module privacyidea.api.lib.prepolicy), 157
 - MotpTokenClass (class in privacyidea.lib.tokens.motptoken), 121
 - MySQL, 21
- ## N
- no_detail_on_fail() (in module privacyidea.api.lib.postpolicy), 158
 - no_detail_on_success() (in module privacyidea.api.lib.postpolicy), 158

- NODETAILFAIL (privacyidea.lib.policy.ACTION attribute), 152
- NODETAILSUCCESS (privacyidea.lib.policy.ACTION attribute), 152
- NONE (privacyidea.lib.policy.ACTIONVALUE attribute), 153
- Novell eDirectory, 19
- O**
- OATH CSV, 80
- offline, 75
- offline_info() (in module privacyidea.api.lib.postpolicy), 158
- OpenLDAP, 19
- Oracle, 21
- OTP length, 49
- OTPPIN (privacyidea.lib.policy.ACTION attribute), 152
- OTPPINCONTENTS (privacyidea.lib.policy.ACTION attribute), 152
- OTPPINMAXLEN (privacyidea.lib.policy.ACTION attribute), 152
- OTPPINMINLEN (privacyidea.lib.policy.ACTION attribute), 152
- OTPPINRANDOM (privacyidea.lib.policy.ACTION attribute), 152
- OTRS, 5, 75
- out of sync, 49
- override client, 27
- overview, 3
- P**
- PAM, 5, 75
- pass on user no token, 25
- pass on user not found, 25
- PASSNOTOKEN (privacyidea.lib.policy.ACTION attribute), 152
- PASSNOUSER (privacyidea.lib.policy.ACTION attribute), 152
- passOnNoToken, 65
- passOnNoUser, 65
- passthru, 65
- PASSTHRU (privacyidea.lib.policy.ACTION attribute), 152
- PasswordTokenClass (class in privacyidea.lib.tokens.passwordtoken), 122
- Penrose, 19
- pi-manage, 173
- pip install, 4
- policies, 56
- Policy (class in privacyidea.models), 170
- PolicyClass (class in privacyidea.lib.policy), 153
- POLICYDELETE (privacyidea.lib.policy.ACTION attribute), 152
- POLICYWRITE (privacyidea.lib.policy.ACTION attribute), 152
- PostgreSQL, 21
- postpolicy (class in privacyidea.api.lib.postpolicy), 158
- prepolicy (class in privacyidea.api.lib.prepolicy), 157
- PRIVACYIDEA (privacyidea.lib.policy.LOGINMODE attribute), 153
- privacyidea.api (module), 82
- privacyidea.api.application (module), 114
- privacyidea.api.auth (module), 82, 84
- privacyidea.api.lib.postpolicy (module), 157
- privacyidea.api.lib.prepolicy (module), 155
- privacyidea.api.machine (module), 111
- privacyidea.api.machineresolver (module), 110
- privacyidea.api.policy (module), 103
- privacyidea.api.realm (module), 91
- privacyidea.api.resolver (module), 90
- privacyidea.api.system (module), 88
- privacyidea.api.token (module), 94
- privacyidea.api.user (module), 102
- privacyidea.api.validate (module), 85
- privacyidea.lib (module), 115
- privacyidea.lib.auditmodules (module), 165
- privacyidea.lib.machines (module), 167
- privacyidea.lib.policy (module), 150
- privacyidea.lib.policydecorators (module), 158
- privacyidea.lib.resolvers (module), 160
- privacyidea.lib.token (module), 140
- privacyidea.lib.user (module), 115
- privacyidea.models (module), 169
- PSKC, 80
- R**
- RADIUS token, 31
- RadiusTokenClass (class in privacyidea.lib.tokens.radius token), 123
- read_keys() (privacyidea.lib.auditmodules.base.Audit method), 166
- Realm (class in privacyidea.models), 170
- realm administrator, 59
- realm autocreation, 25
- realm edit, 24
- realms, 23
- registration, 29
- RegistrationTokenClass (class in privacyidea.lib.tokens.registration token), 124
- Remote token, 32
- RemoteTokenClass (class in privacyidea.lib.tokens.remotetoken), 125
- remove_token() (in module privacyidea.lib.token), 147
- request, 29
- RESET (privacyidea.lib.policy.ACTION attribute), 152
- reset() (privacyidea.lib.tokenclass.TokenClass method), 137

reset_token() (in module `privacyidea.lib.token`), 147
 Resolver (class in `privacyidea.models`), 170
 resolver priority, 24
 ResolverConfig (class in `privacyidea.models`), 170
 RESOLVERDELETE (`privacyidea.lib.policy.ACTION` attribute), 152
 ResolverRealm (class in `privacyidea.models`), 170
 RESOLVERWRITE (`privacyidea.lib.policy.ACTION` attribute), 152
 REST, 82
 Restore, 44
 RESYNC (`privacyidea.lib.policy.ACTION` attribute), 152
 resync token, 50
 resync() (`privacyidea.lib.tokenclass.TokenClass` method), 137
 resync() (`privacyidea.lib.tokens.daplugtoken.DaplugTokenClass` method), 118
 resync() (`privacyidea.lib.tokens.hotptoken.HotpTokenClass` method), 121
 resync() (`privacyidea.lib.tokens.totptoken.TotpTokenClass` method), 130
 resync_token() (in module `privacyidea.lib.token`), 147
 resyncDiffLimit (`privacyidea.lib.tokens.totptoken.TotpTokenClass` attribute), 130
 RFC6030, 80
 rollout strategy, 173

S

SAML, 75
 save() (`privacyidea.lib.tokenclass.TokenClass` method), 137
 save() (`privacyidea.models.TokenRealm` method), 172
 SCIM resolver, 23
 scope, 56
 SCOPE (class in `privacyidea.lib.policy`), 154
 search() (`privacyidea.lib.auditmodules.base.Audit` method), 166
 search() (`privacyidea.lib.auditmodules.sqlaudit.Audit` method), 167
 search_query() (`privacyidea.lib.auditmodules.base.Audit` method), 166
 searchQuery() (`privacyidea.lib.auditmodules.sqlaudit.Audit` method), 167
 selfservice policies, 60
 SERIAL (`privacyidea.lib.policy.ACTION` attribute), 152
 SET (`privacyidea.lib.policy.ACTION` attribute), 152
 set() (`privacyidea.lib.auditmodules.base.Audit` method), 166
 set_count_auth() (in module `privacyidea.lib.token`), 147
 set_count_auth() (`privacyidea.lib.tokenclass.TokenClass` method), 137
 set_count_auth_max() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_count_auth_success() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_count_auth_success_max() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_count_window() (in module `privacyidea.lib.token`), 147
 set_count_window() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_data() (`privacyidea.models.Challenge` method), 169
 set_defaults() (in module `privacyidea.lib.token`), 148
 set_defaults() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_description() (in module `privacyidea.lib.token`), 148
 set_description() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_failcount() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_hashlib() (in module `privacyidea.lib.token`), 148
 set_hashlib() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_info() (`privacyidea.models.Token` method), 171
 set_init_details() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_max_failcount() (in module `privacyidea.lib.token`), 148
 set_maxfail() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_otp_count() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_otpkey() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_otplen() (in module `privacyidea.lib.token`), 148
 set_otplen() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_otplen() (`privacyidea.lib.tokens.passwordtoken.PasswordTokenClass` method), 123
 set_pin() (in module `privacyidea.lib.token`), 149
 set_pin() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_pin() (`privacyidea.models.Token` method), 171
 set_pin_hash_seed() (`privacyidea.lib.tokenclass.TokenClass` method), 138
 set_pin_so() (in module `privacyidea.lib.token`), 149
 set_pin_user() (in module `privacyidea.lib.token`), 149
 set_policy() (in module `privacyidea.lib.policy`), 155
 set_realm() (in module `privacyidea.api.lib.prepolicy`), 157
 set_realms() (in module `privacyidea.lib.token`), 149
 set_realms() (`privacyidea.lib.tokenclass.TokenClass` method), 138

- set_realms() (privacyidea.models.Token method), 171
 - set_so_pin() (privacyidea.lib.tokenclass.TokenClass method), 138
 - set_so_pin() (privacyidea.models.Token method), 171
 - set_sync_window() (in module privacyidea.lib.token), 149
 - set_sync_window() (privacyidea.lib.tokenclass.TokenClass method), 138
 - set_tokeninfo() (privacyidea.lib.tokenclass.TokenClass method), 138
 - set_type() (privacyidea.lib.tokenclass.TokenClass method), 138
 - set_user() (privacyidea.lib.tokenclass.TokenClass method), 139
 - set_user_identifiers() (privacyidea.lib.tokenclass.TokenClass method), 139
 - set_user_pin() (privacyidea.lib.tokenclass.TokenClass method), 139
 - set_validity_period_end() (in module privacyidea.lib.token), 150
 - set_validity_period_end() (privacyidea.lib.tokenclass.TokenClass method), 139
 - set_validity_period_start() (in module privacyidea.lib.token), 150
 - set_validity_period_start() (privacyidea.lib.tokenclass.TokenClass method), 139
 - SETPIN (privacyidea.lib.policy.ACTION attribute), 152
 - SETREALM (privacyidea.lib.policy.ACTION attribute), 152
 - setup tool, 41
 - setup() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver class method), 163
 - Sipgate, 39
 - SMS, 29
 - SMS automatic resend, 65
 - SMS Gateway, 39
 - SMS policy, 65
 - SMS text, 65
 - SMS Token, 39
 - SMS token, 34
 - SmsTokenClass (class in privacyidea.lib.tokens.smstoken), 126
 - Software Tokens, 28
 - SpassTokenClass (class in privacyidea.lib.tokens.spasstoken), 128
 - split_pin_pass() (privacyidea.lib.tokenclass.TokenClass method), 139
 - split_pin_pass() (privacyidea.lib.tokens.daplugtoken.DaplugTokenClass method), 118
 - split_pin_pass() (privacyidea.lib.tokens.radiusstoken.RadiusTokenClass method), 123
 - split_pin_pass() (privacyidea.models.Token method), 172
 - split_uri() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver class method), 165
 - split_user() (in module privacyidea.lib.user), 116
 - SQL resolver, 21
 - sqlite, 21
 - SSH Key, 29
 - SSH keys, 34
 - SSHkeyTokenClass (class in privacyidea.lib.tokens.sshkeytoken), 129
 - status_validation_fail() (privacyidea.lib.tokenclass.TokenClass method), 139
 - status_validation_success() (privacyidea.lib.tokenclass.TokenClass method), 139
 - superuser realm, 56
 - syncwindow, 49
 - system config, 25
 - SYSTEMDELETE (privacyidea.lib.policy.ACTION attribute), 152
 - SYSTEMWRITE (privacyidea.lib.policy.ACTION attribute), 152
- ## T
- testconnection() (privacyidea.lib.machines.base.BaseMachineResolver class method), 167
 - testconnection() (privacyidea.lib.machines.hosts.HostsMachineResolver class method), 168
 - testconnection() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver class method), 165
 - testconnection() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver class method), 162
 - timeshift (privacyidea.lib.tokens.totpoken.TotpTokenClass attribute), 130
 - timestep (privacyidea.lib.tokens.totpoken.TotpTokenClass attribute), 130
 - timewindow (privacyidea.lib.tokens.totpoken.TotpTokenClass attribute), 131
 - token, 3
 - Token (class in privacyidea.models), 170
 - token configuration, 36
 - token default settings, 25
 - token description, 49
 - token types, 29
 - token_exist() (in module privacyidea.lib.token), 150
 - token_has_owner() (in module privacyidea.lib.token), 150

TokenClass (class in privacyidea.lib.tokenclass), 132

TokenInfo (class in privacyidea.models), 172

TOKENLABEL (privacyidea.lib.policy.ACTION attribute), 152

TOKENPIN (privacyidea.lib.policy.ACTIONVALUE attribute), 153

TokenRealm (class in privacyidea.models), 172

TOKENREALMS (privacyidea.lib.policy.ACTION attribute), 152

TOKENTYPE (privacyidea.lib.policy.ACTION attribute), 152

tokenview, 47

tools, 79

TotpTokenClass (class in privacyidea.lib.tokens.totptoken), 129

U

ubuntu, 4

ui_get_enroll_tokentypes() (privacyidea.lib.policy.PolicyClass method), 154

UNASSIGN (privacyidea.lib.policy.ACTION attribute), 152

unassign_token() (in module privacyidea.lib.token), 150

update() (privacyidea.lib.tokenclass.TokenClass method), 139

update() (privacyidea.lib.tokens.certificatetoken.CertificateTokenClass method), 117

update() (privacyidea.lib.tokens.emailtoken.EmailTokenClass method), 119

update() (privacyidea.lib.tokens.hotptoken.HotpTokenClass method), 121

update() (privacyidea.lib.tokens.motptoken.MotpTokenClass method), 122

update() (privacyidea.lib.tokens.passwordtoken.PasswordTokenClass method), 123

update() (privacyidea.lib.tokens.radiusTokenClass method), 123

update() (privacyidea.lib.tokens.registrationToken.RegistrationTokenClass method), 125

update() (privacyidea.lib.tokens.remotetoken.RemoteTokenClass method), 126

update() (privacyidea.lib.tokens.smstoken.SmsTokenClass method), 128

update() (privacyidea.lib.tokens.spasstoken.SpassTokenClass method), 128

update() (privacyidea.lib.tokens.sshkeytoken.SSHkeyTokenClass method), 129

update() (privacyidea.lib.tokens.totptoken.TotpTokenClass method), 131

update() (privacyidea.lib.tokens.yubicotoken.YubicoTokenClass method), 131

update_otpkey() (privacyidea.models.Token method), 172

update_type() (privacyidea.models.Token method), 172

update_user() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver method), 162

UPDATEUSER (privacyidea.lib.policy.ACTION attribute), 152

USER (privacyidea.lib.policy.SCOPE attribute), 154

user policies, 60

User() (in module privacyidea.lib.user), 115

UserIdResolver (class in privacyidea.lib.resolvers.UserIdResolver), 160

useridresolvers, 17, 160

USERLIST (privacyidea.lib.policy.ACTION attribute), 153

Users, 59

USERSTORE (privacyidea.lib.policy.ACTIONVALUE attribute), 153

USERSTORE (privacyidea.lib.policy.LOGINMODE attribute), 153

userview, 52

V

virtual environment, 4

W

WEBUI (privacyidea.lib.policy.SCOPE attribute), 154

WebUI Login, 70

WebUI Policy, 70

Y

Yubico, 29

Yubico AES mode, 36

Yubico Cloud mode, 34, 41

YubicoTokenClass (class in privacyidea.lib.tokens.yubicotoken), 131

Yubikey, 29, 34, 36

Yubikey CSV, 80

YubikeyTokenClass (class in privacyidea.lib.tokens.yubikeytoken), 131