
privacyIDEA Documentation

Release 2.2

Cornelius Kölbel

April 09, 2015

1	Overview	3
2	Installation	5
2.1	Python Package Index	5
2.2	Ubuntu Packages	6
2.3	Debian Packages	6
3	Upgrade From privacyIDEA 1.5	11
3.1	Stopping Your Server	11
3.2	Upgrade Software	11
3.3	Configuration	11
3.4	Migrate The Database	12
3.5	Create An Administrator	12
3.6	Start The Server	12
4	The Config File	13
5	The WSGI Script	15
6	Login to the Web UI	17
6.1	Login for normal users	17
6.2	Login for administrators	17
7	Configuration	19
7.1	UserIdResolvers	19
7.2	Realms	23
7.3	System Config	25
7.4	Token configuration	27
7.5	privacyIDEA setup tool	29
8	Tokenview	33
8.1	Token Details	33
9	Userview	39
9.1	User Details	40
10	Policies	43
10.1	Admin policies	43
10.2	User Policies	46

10.3	Authentication policies	50
10.4	Authorization policies	52
10.5	Enrollment policies	53
10.6	WebUI Policies	55
11	Audit	59
11.1	Cleaning up entries	60
12	Client machines	61
12.1	SSH	61
12.2	LUKS	62
12.3	Offline	62
13	Application Plugins	63
13.1	Pluggable Authentication Module	63
13.2	FreeRADIUS Plugin	63
13.3	simpleSAMLphp Plugin	64
14	Tools	67
14.1	Get Serial by OTP value	67
14.2	Copy token PIN	67
14.3	Check Policy	68
14.4	Export token information	68
14.5	Export audit information	68
15	Import	69
15.1	SafeNet XML	69
15.2	OATH CSV	69
15.3	Yubikey CSV	70
16	The Code Documentation	71
16.1	API level	71
16.2	LIB level	71
16.3	DB level	71
17	Indices and tables	145
	HTTP Routing Table	147
	Python Module Index	149

As always the privacyIDEA documentation is work in progress.

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!

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 [The 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.

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 14.04LTS and
- ready made *Debian Packages* for Debian Wheezy.

If you want to upgrade from a privacyIDEA 1.5 installation please read *Upgrade From privacyIDEA 1.5*.

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.

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 privacyideaadm
```

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
```

2.2.1 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*.

2.2.2 SimpleSAMLphp

Starting with 1.4 privacyIDEA also supports SAML via a plugin for simpleSAMLphp ². 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*.

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
```

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

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
```

2.3.1 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.

Upgrade From privacyIDEA 1.5

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

3.1 Stopping Your Server

Be sure to stop your privacyIDEA server.

3.2 Upgrade Software

To upgrade the code enter your python virtualenv and run:

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

3.3 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_ENCFILE = 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!

3.4 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/`.

3.5 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>
```

3.6 Start The Server

Run the server:

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

or add it to your Apache or Nginx configuration.

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
```

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.

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.

Login to the Web UI

privacyIDEA has one login form for users to login and for administrators to login.

The login screen can contain a dropdown box with the list of existing realms. The dropdown box can be configured in the *System Config* dialog on the *GUI settings* tab.

You should enter your username and choose the right realm. If the dropdown box is not displayed, then you need to append the realm to the username like `username@realm`.

6.1 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.

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*)

6.2 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.

Configuration

The configuration menu can be used to define `useridresolvers` and realms, set the system config and the token config. It also contains a shortcut to the policy tab (see *Policies*).

7.1 UserIdResolvers

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

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

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

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

- Flatfile resolver,
- LDAP resolver,
- SQL resolver,
- SCIM resolver. (**TODO**): Not yet migrated.

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.

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.

7.1.1 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
```

7.1.2 LDAP resolver

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

The screenshot shows the 'Create a new LDAP Resolver' page in the privacyIDEA web interface. The top navigation bar includes 'System', 'Token View', 'User View', 'Config', and 'Audit'. A 'Logout admin @ (Role: admin)' button is in the top right. The left sidebar shows a tree view with 'All Resolvers', 'New Resolvers', 'New passwordresolver', 'New ldapresolver' (highlighted), and 'New sqlresolver'. The main content area is titled 'Create a new LDAP Resolver' and contains the following fields:

- Resolver name:** resolver1
- Server URI:** ldap://privacyidea.server1, ldap://privacyidea.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 (dropdown menu)
- Timeout (seconds):** 5
- Size Limit:** 500

Below these fields are two buttons: 'Preset OpenLDAP' and 'Preset Active Directory'. A checkbox labeled 'No anonymous referral chasing' is checked. The 'Advanced configuration' section includes:

- 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

At the bottom are two buttons: 'Test LDAP Resolver' and 'Save resolver'.

Figure 7.1: LDAP resolver configuration

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

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

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

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`.

7.1.3 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:

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

privacyIDEA
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

Figure 7.2: *SQL resolver configuration*

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.

7.1.4 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 OSIAM², 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`.

7.2 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. Therefore the users need to authenticate with their username and the realm like this:

```
user@realm
```

7.2.1 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.

² <http://www.osiam.org>

7.2.2 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.

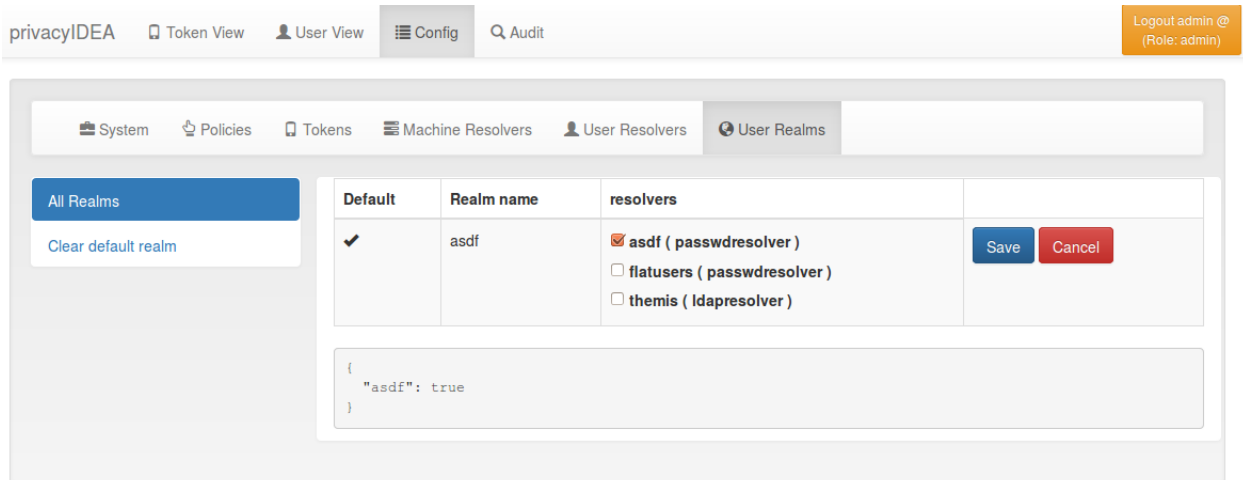
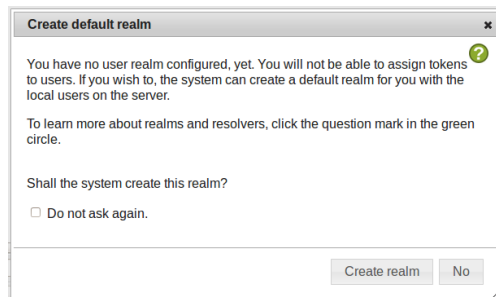


Figure 7.3: Edit a realm

7.2.3 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.

7.3 System Config

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

Figure 7.4: The system config

7.3.1 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.

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.

7.3.2 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.

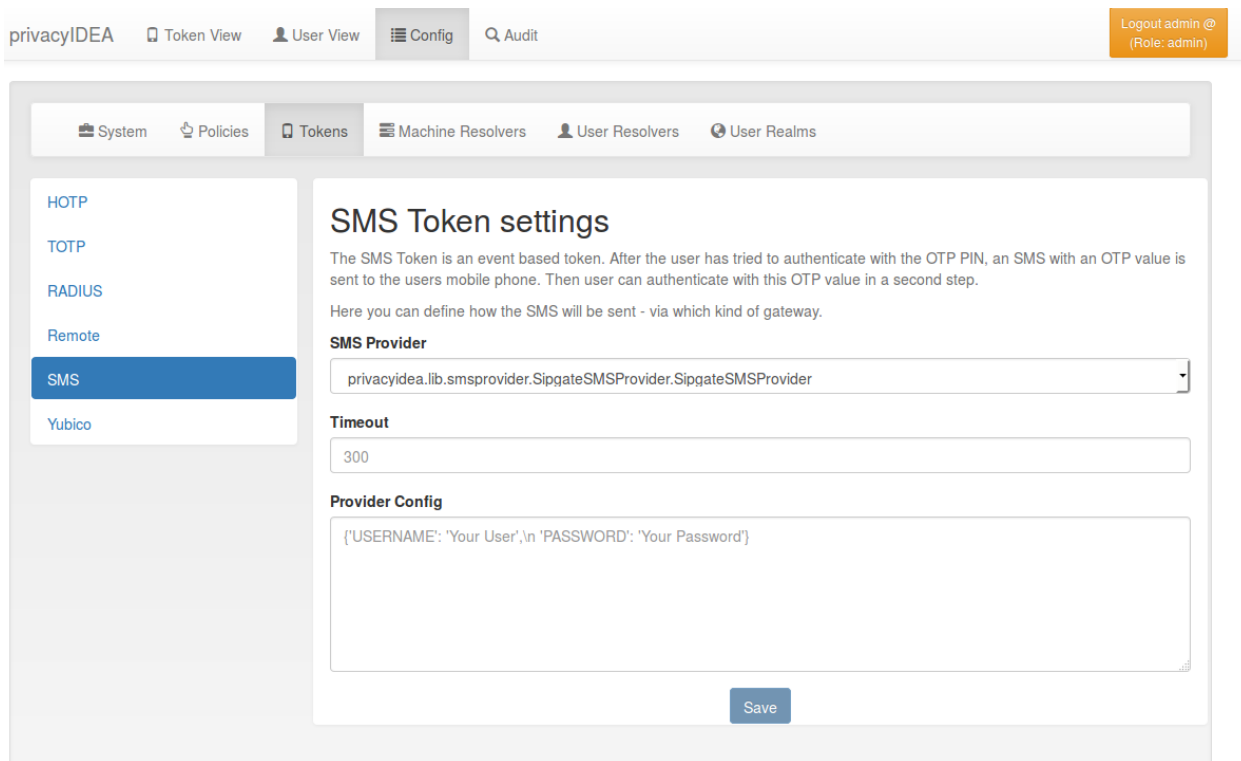
default QR suite is the OCRA suite that is set for a QR token during enrollment if no OCRA suite is specified. **(TODO):** Not migrated, yet.

7.3.3 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.

7.4 Token configuration

Each token type can provide its own configuration dialog.



The screenshot shows the privacyIDEA web interface. At the top, there is a navigation bar with links for "Token View", "User View", "Config", and "Audit". The "Config" link is active. Below the navigation bar, there is a sidebar with a list of token types: HOTP, TOTP, RADIUS, Remote, SMS (selected), and Yubico. The main content area is titled "SMS Token settings". It contains a description of the SMS Token, a section for defining the SMS gateway, a dropdown menu for the "SMS Provider" (set to "privacyidea.lib.smsprovider.SipgateSMSProvider.SipgateSMSProvider"), a "Timeout" field (set to 300), and a "Provider Config" text area containing the configuration string: `{'USERNAME': 'Your User', 'PASSWORD': 'Your Password'}`. A "Save" button is located at the bottom right of the configuration area.

Figure 7.5: Token Configuration: SMS

7.4.1 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 an API call or by authenticating with only the OTP PIN:

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.

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

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.

7.5 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:

```
privacyidea-setup-tui
```

This will bring you to this start screen.

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

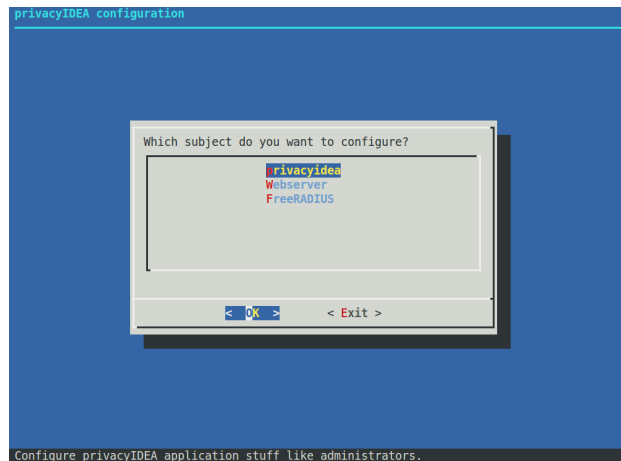


Figure 7.6: Start screen of the appliance setup tool.

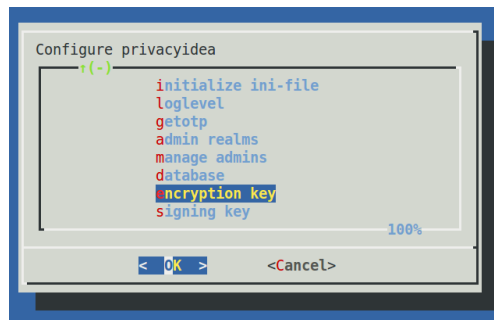


Figure 7.7: Configure privacyidea

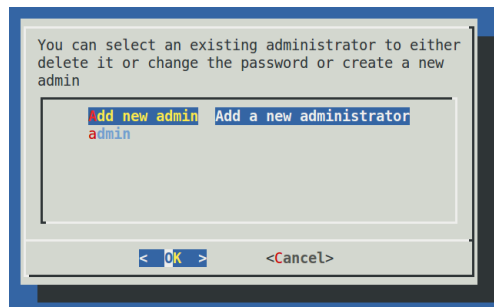


Figure 7.8: You can create new token administrators, delete them and change their passwords.

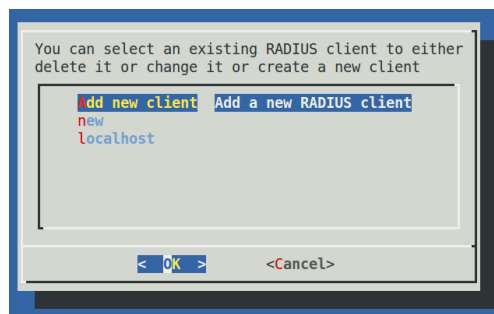


Figure 7.9: In the FreeRADIUS settings you can create and delete RADIUS clients.

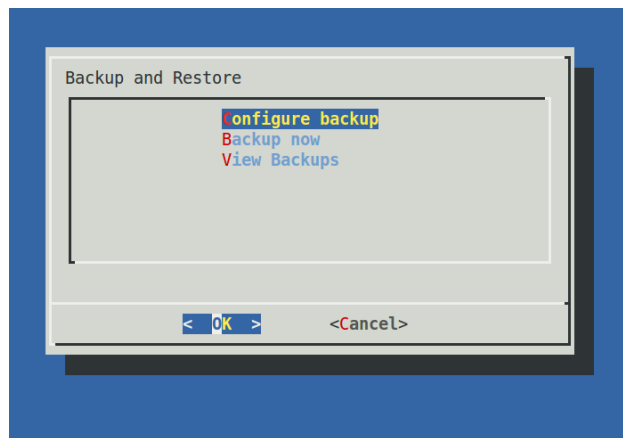
All changes done in this setup tool are directly read from and written to the corresponding configuration files. The setup tool parses the original nginx and freeradius configuration files. So there is no additional place where this data is kept.

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

7.5.1 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.



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`.

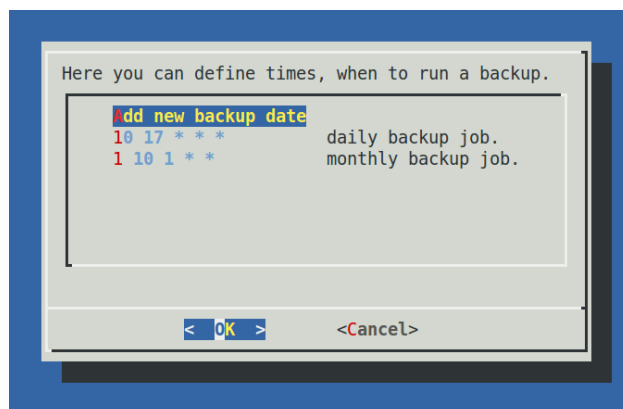


Figure 7.10: Scheduled backup

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.

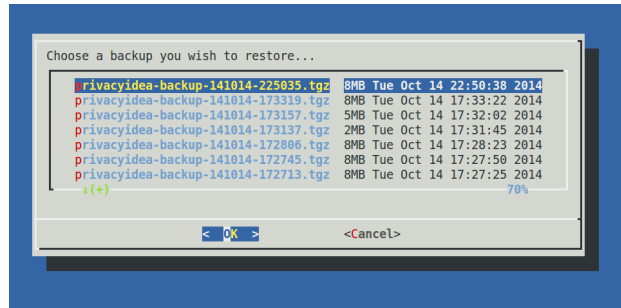


Figure 7.11: All available backups

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.

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.

serial	type	active	description	failcounter	user	realm
OATH000FB1E	hotp	active		0	cornelius	asdf
OATH00019B35	hotp	active		0	root	asdf
PIMO0000671B	motp	active		0		
SSHK0000FA9F	sshkey	active		0		
TOTP000067E9	totp	active		0	root	asdf

Figure 8.1: *Token View*

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

8.1 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](#).

8.1.1 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.

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	

Figure 8.2: Token Detail

The lost token is deactivated.

8.1.2 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.

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

this can be time consuming!

8.1.3 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 algorith 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

(TODO)

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

Tokeninfo - Valid end

(TODO)

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.

8.1.4 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.

8.1.5 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.

8.1.6 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.

8.1.7 enroll

For this function you do not need to select any token. But you can select a user on the userview, then the token to be enrolled will be directly assigned to this user.

When enrolling the token, you can choose, which token type you want to enroll. After enrolling the token, you can set a OTP PIN for this token.

8.1.8 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.

8.1.9 unassign

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

8.1.10 enable

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

8.1.11 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.

8.1.12 set PIN

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

8.1.13 Reset Failcounter

If a user locked his token, since he entered wrong OTP values or wrong OTP PINs, the fail counter has reached the max 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.

8.1.14 delete

Selecting tokens and clicking the button `delete` will remove the tokens from the database. The token information can not be recovered. But all events that occurred with this token still remain in the audit log.

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. The top navigation bar includes 'privacyIDEA', 'Token View', 'User View' (active), 'Config', and 'Audit'. A 'Logout admin @ (Role: admin)' button is in the top right. On the left, there's a sidebar with 'All users', a 'Select Realm' dropdown (showing 'asdf'), 'Quick links', 'Edit realms', and 'total users: 52'. The main area displays a table of users with pagination controls (First, Previous, 1, 2, 3, 4, Next, Last). The table has columns: username, surname, givenname, email, phone, mobile, description, and Id.

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	Köbel	Cornelius	cornelius.koelbel@netknights.it	+49 561 3166797	+49 151 2960 1417		1009
corny							1003
franzi		franzi					1000

Figure 9.1: 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.

As privacyIDEA only reads users from user sources the actions you can perform on the users are limited.

Note: You can not create or modify users in privacyIDEA!

9.1 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 links for 'All users', 'User cornelius' (selected), 'Quick links', 'Edit realms', and 'total users: 52'. The main content area displays 'Details for user cornelius in realm asdf' with a 'View user in Audit log' button. Below this, user details are listed in two columns: Username (cornelius), Email (cornelius.koelbel@netknights.it), Given name (Cornelius), Phone (+49 561 3166797), Surname (Kölbel), and Mobile (+49 151 2960 1417). A table titled 'Tokens for user cornelius' shows one token with serial OATH0000FB1E, type hotp, active status, window 10, and other attributes. An 'Enroll New Token' button is present. Below the table, the 'Assign a new token' section includes input fields for Serial, PIN, and Repeat password, with an 'Assign Token' button.

Figure 9.2: *User Details.*

You see a list of the users tokens and change to the *Token Details*.

9.1.1 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.

9.1.2 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.

9.1.3 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.

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:

10.1 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 *rest_token*, *rest_system*, *rest_realm* and *rest_resolver*.

Admin policies are implemented as decorators in *Policy Module* and *Policy Decorators*.

The `user` in the admin policies refers to the administrator.

All administrative actions also refer to the defined realm. Meaning an administrator may have many rights in one 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*:

10.1.1 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.

10.1.2 enable

type: bool

The `enable` action allows the administrator to activate disabled tokens.

10.1.3 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.

10.1.4 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.

10.1.5 setOTPPIN

type: bool

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

10.1.6 setMOTPPIN

type: bool

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

10.1.7 resync

type: bool

If the `resync` action is defined, the administrator is allowed to resynchronize a token.

10.1.8 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.

10.1.9 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.

10.1.10 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.

10.1.11 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.

10.1.12 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.

10.1.13 checkstatus

type: bool

If the `checkstatus` action is defined, the administrator is allowed to check the status of open challenge requests.

10.1.14 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.

10.1.15 getserial

type: bool

If the `getserial` action is defined, the administrator is allowed to calculate the token serial number for a given OTP value.

10.1.16 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!

10.1.17 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.

10.1.18 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.

10.1.19 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.

10.2 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 `rest_token` 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*:

10.2.1 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.

10.2.2 assign

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.

10.2.3 disable

type: bool

The user is allowed to disable his own tokens. Disabled tokens can not be used to authenticate.

10.2.4 enable

type: bool

The user is allowed to enable his own tokens.

10.2.5 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.

10.2.6 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.

10.2.7 resync

type: bool

The user is allowed to resynchronize the token if it has got out of synchronization.

10.2.8 reset

type: bool

The user is allowed to reset the failcounter of the token.

10.2.9 setOTPPIN

type: bool

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

10.2.10 setMOTPPIN

type: bool

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

10.2.11 getotp

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

type: bool

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

10.2.12 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.

10.2.13 otp_pin_minlength

type: integer

range: 0 - 31

This is the minimum required PIN the user must use when setting the OTP PIN.

10.2.14 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**)

10.2.15 activateQR

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

type: bool

The user is allowed to enroll a QR token.

10.2.16 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.

10.2.17 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.

10.2.18 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.

10.2.19 auditlog

type: bool

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

10.2.20 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.

10.3 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 *rest_validate* and are checked using *Policy Module* and *Policy Decorators*.

The following actions are available in the scope *authentication*:

10.3.1 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.

10.3.2 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.

10.3.3 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.

10.3.4 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.

10.3.5 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>`.

Note: You need to put the text into quotes, otherwise it will be split

and interpreted as list of words.

10.3.6 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.

10.3.7 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.

10.3.8 challenge_response

(TODO): not yet migrated.

type: string

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

10.4 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 *rest_validate* and are checked using *Policy Module* and *Policy Decorators*.

The following actions are available in the scope *authorization*:

10.4.1 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.

10.4.2 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.

10.4.3 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.

10.4.4 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.

10.4.5 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.

10.5 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 *rest_token* and specially the *init* and *assign*-methods.

Technically the decorators in [API Policies](#) are used.

The following actions are available in the scope *enrollment*:

10.5.1 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.

10.5.2 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.

10.5.3 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.

10.5.4 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.

10.5.5 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.

10.5.6 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.

10.5.7 lostTokenPWLen

type: int

This is the length of the generated password for the lost token process.

10.5.8 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)*.

10.5.9 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.

10.6 WebUI Policies

10.6.1 login_mode

type: string

allowed values: “userstore”, “privacyIDEA”

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.

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.

10.6.2 logout_time

type: int

Set the timeout, after which a user in th WebUI will be logged out. The dafault timeout is 30 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.

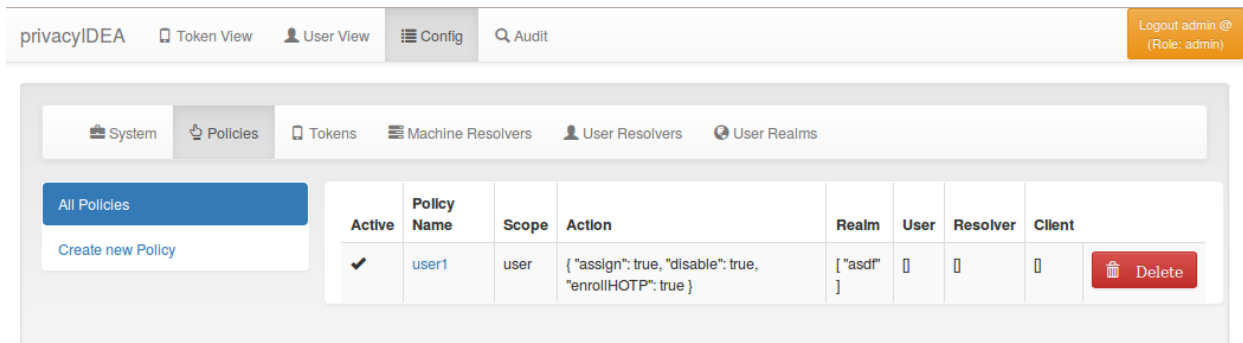


Figure 10.1: Policy Definition

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.

Audit

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

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

First
Previous
1
2
3
4
5
6
7
8
9
10
Next
Last

Download Audit log
Download file
268 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/	1		**		admin			127.0.0.1	realm: [""]	OK	FAIL		
267	Feb 21, 2015 8:50:54 AM	GET /token/	1		**		admin			127.0.0.1	realm: [""]	OK	OK		
266	Feb 21, 2015 8:49:18 AM	GET /policy/defs	1				admin			127.0.0.1		OK	OK		
265	Feb 21, 2015 8:49:18 AM	GET /resolver/	1				admin			127.0.0.1		OK	OK		
264	Feb 21, 2015 8:49:18 AM	GET /realm/	1				admin			127.0.0.1		OK	OK		
263	Feb 21, 2015 8:49:17 AM	GET /policy	1				admin			127.0.0.1	name = None, realm = None, scope = None	OK	OK		

Figure 11.1: Audit Log

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

11.1 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.

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 application require different authentication items.

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

You need to assign an application and a token to a client machine. Each application type can work with certain token types and each application type can use additional parameters.

Note: Not all tokens work well with all applications!

12.1 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.

12.2 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 --cleanslot --name salt-minion
```

For more information please see the man page of this tool.

12.3 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.

Application Plugins

privacyIDEA comes with application plugins. These are plugins for applications like PAM, OTRS, FreeRADIUS or simpleSAMLphp which enable these application to authenticate users against privacyIDEA.

13.1 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`.

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.

13.2 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 ¹.

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
}
```

¹ <https://github.com/privacyidea/privacyidea/tree/master/authmodules/FreeRADIUS>

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 perl module is not thread safe, so you need to start FreeRADIUS with the -t switch.

13.3 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 ².

Follow the simpleSAMLphp instructions to configure your authsources.php. A usual configuration will look like this:

```
'example-privacyidea' => array(
    'privacyidea:privacyidea',

    /*
     * The name of the privacyidea server and the protocol
     * A port can be added by a colon
     * Required.
     */
    '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.
     */
    /*
```

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

```
'attributemap' => array('username' => 'samlLoginName',  
                        'surname' => 'surName',  
                        'givenname' => 'givenName',  
                        'email' => 'emailAddress',  
                        'phone' => 'telePhone',  
                        'mobile' => 'mobilePhone',  
                        ),  
,
```

Tools

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

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

14.1 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 go 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!

14.2 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.

14.3 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.

14.4 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.

14.5 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.

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 or *Yubikey CSV* files.

15.1 SafeNet XML

SafeNet XML (former Aladdin) files are usually shipped with SafeNet eToken PASS.

Each token is contained in a <Token> tag and the seed is contained in a <seed> tag. These files were originally used to import HOTP-SHA1 tokens.

Note: Today usually SafeNet tokens are shipped with a .dat file. If you get such a file, please contact the mailing list.

15.2 OATH CSV

This is a very simple CSV file to import HOTP, TOTP or OATH tokens. You can also convert your seed easily to this file format, to import the tokens.

The file format looks like this:

```
<serial>, <seed>, <type>, <otp length>, <time step>
```

For OCRA tokens it looks like this:

```
<serial>, <seed>, OCRA, <ocra suite>
```

serial is the serial number of the token that will also be used to identify the token in the database. Importing the same serial number twice will overwrite the token data.

seed is the secret key, that is used to calculate the OTP value. The seed is provided in a hexadecimal notation. Depending on the length either the SHA1 or SHA256 hash algorithm is identified.

type is either HOTP, TOTP or OCRA.

otp length is the length of the OTP value generated by the token. This is usually 6 or 8.

time step is the time step of TOTP tokens. This is usually 30 or 60.

ocra suite is the ocra suite of the OCRA token according to ¹.

¹ <http://tools.ietf.org/html/rfc6287#section-6>

The Code Documentation

The code roughly has three levels.

16.1 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.

16.2 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.

16.3 DB level

On the DB level you can simply modify all objects.

16.3.1 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.

```
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, **kwargs*)
 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='', 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, **kws*)

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. 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.ResolverRealm` (*resolver_id=None, realm_id=None, resolver_name=None, realm_name=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, otplen=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.and(*clauses)`
Produce a conjunction of expressions joined by AND.

E.g.:

```
from sqlalchemy import and_

stmt = select([users_table]).where(
    and_(
        users_table.c.name == 'wendy',
        users_table.c.enrolled == True
    )
)
```

The `and_()` conjunction is also available using the Python `&` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
    (users_table.c.name == 'wendy') &
    (users_table.c.enrolled == True)
)
```

The `and_()` operation is also implicit in some cases; the `Select.where()` method for example can be invoked multiple times against a statement, which will have the effect of each clause being combined using `and_()`:

```
stmt = select([users_table]).\
    where(users_table.c.name == 'wendy').\
    where(users_table.c.enrolled == True)
```

See also:

`or_()`

`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

16.3.2 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!

`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.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 user information

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

```
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 (*passwd, 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

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

This method verifies if there is a matching challenge for the given passwd 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
- **passwd** (*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_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 initialiaction 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:l 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** – tbdf.

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

Split the password into the token PIN and the OTP value

take the given password and split it into the PIN and the OTP value. The splitting can be dependent of certain policies. The policies may depend on the user.

Each token type may define its own way to slit the PIN and the OTP value.

Parameters

- **passw** – the password to split
- **user** (*User object*) – The user/owner of the token
- **options** (*dict*) – can be used be the token types.

Returns tuple of pin and otp value

Returns tuple of (split status, pin, otp value)

Return type tuple

status_validation_fail ()

callback to enable a status change, if auth failed

status_validation_success ()

callback to enable a status change, if auth succeeds

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, **kwargs)
```

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.and_(*clauses)
```

Produce a conjunction of expressions joined by AND.

E.g.:

```
from sqlalchemy import and_

stmt = select([users_table]).where(
    and_(
        users_table.c.name == 'wendy',
        users_table.c.enrolled == True
    )
)
```

The `and_()` conjunction is also available using the Python `&` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
    (users_table.c.name == 'wendy') &
    (users_table.c.enrolled == True)
)
```

The `and_()` operation is also implicit in some cases; the `Select.where()` method for example can be invoked multiple times against a statement, which will have the effect of each clause being combined using `and_()`:

```
stmt = select([users_table]).\
    where(users_table.c.name == 'wendy').\
    where(users_table.c.enrolled == True)
```

See also:

`or_()`

```
privacyidea.lib.token.assign_token(*args, **kwargs)
```

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) :param epoch_start: unix time start date (used with time based tokens) :param epoch_end: unix time end date (used with time based tokens) :param curTime: Simulate the server time :type curTime: datetime :param timestamp: Simulate the server time (unix time in seconds) :type timestamp: int

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 servertime 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** – Get either assigned (True) or unassigned (False) tokens.

If None get all tokens. :type assigned: bool :param user: Filter for the Owner of the token :type user: User Object :param serial: The serial number of the token :type serial: basestring :param active: Whether only active (True) or inactive (False) tokens should be returned :type active: bool :param resolver: filter for the given resolver name :type resolver: basestring :param 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. :param count: If set to True, only the number of the result and not the list is returned. :type count: bool

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** – 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. :type sortby: A Token column or a string. :param sortdir: Can be “asc” (default) or “desc” :type sortdir: basestring :param psize: The size of the page :type psize: int :param page: The number of the page to view. Starts with 1 ;-) :type page: int :return: dict with tokens, prev, next and count :rtype: 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` – initialization parameters like: serial (optional) type (optional, default=hotp) otpkey

type param: dict :param user: the token owner :type user: User Object :param tokenrealms: the realms, to which the token should belong :type tokenrealms: list

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 resetted :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 server time 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 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.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.

ASSIGN = `'assign'`

AUDIT = `'auditlog'`

AUTHITEMS = `'fetch_authentication_items'`

AUTOASSIGN = `'autoassignment'`

COPYTOKENPIN = `'copytokenpin'`

COPYTOKENUSER = `'copytokenuser'`

DELETE = `'delete'`

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 = 'tokentealms'
TOKENTYPE = 'tokentype'
UNASSIGN = 'unassign'
USERLIST = 'userlist'

```

```
class privacyidea.lib.policy.ACTIONVALUE
```

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

```

NONE = 'none'
TOKENPIN = 'tokenpin'
USERSTORE = 'userstore'

```

```
class privacyidea.lib.policy.LOGINMODE
```

This is the list of possible values for the login mode.

```
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)

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 :return: A list of the allowed tokentypes :rtype: list

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

Return the policies of the given filter values

Parameters

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

Returns list of policies

Return type list of dicts

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.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_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.

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, passw, options=None)`

This decorator checks, if the user does exist at all. If the user does exist, the wrapped function is called.

The wrapped function is usually `token.check_user_pass`, which takes the arguments (user, passw, options={})

Parameters

- `wrapped_function` –
- `user_object` –
- `passw` –

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

Returns Tuple of True/False and reply-dictionary

```
privacyidea.lib.policydecorators.auth_user_has_no_token(wrapped_function,
                                                         user_object,      passw,
                                                         options=None)
```

This decorator checks if the user has a token at all. If the user has a token, the wrapped function is called.

The wrapped function is usually token.check_user_pass, which takes the arguments (user, passw, options={})

Parameters

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

Returns Tuple of True/False and reply-dictionary

```
privacyidea.lib.policydecorators.auth_user_passthru(wrapped_function, user_object,
                                                    passw, options=None)
```

This decorator checks the policy settings of ACTION.PASSTHRU. If the authentication against the userstore is not successful, the wrapped function is called.

The wrapped function is usually token.check_user_pass, which takes the arguments (user, passw, options={})

Parameters

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

Returns Tuple of True/False and reply-dictionary

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

Decorator to decorate the lib.token.lost_token function. Depending on ACTION.LOSTTOKENVALID, ACTION.LOSTTOKENPWCONTENTS, ACTION.LOSTTOKENPWLEN it sets the check_otp parameter, to signal how the lostToken should be generated.

Parameters

- **wrapped_function** – Usually the function lost_token()
- **args** – argument “serial” as the old serial number
- **kwds** – keyword arguments like “validity”, “contents”, “pw_len”

kwds[“options”] contains the flask g

Returns calls the original function with the modified “validity”,

“contents” and “pw_len” argument

```
class privacyidea.lib.policydecorators.libpolicy(decorator_function)
```

This is the decorator wrapper to call a specific function before a library call in contrast to prepolicy and postpolicy, which are to be called in API Calls.

The decorator expects a named parameter “options”. In this options dict it will look for the flask global “g”.

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

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
- **kwargs** – keyword arguments like options and !check_otp!

kwargs["options"] contains the flask g :return: calls the original function with the modified “check_otp” argument

16.3.3 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
```

```
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

```
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={}*)

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

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** (*string*) – 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

loadConfig (*config*)

Load the config from conf.

Parameters **config** (*dict*) – The configuration from the Config Table

The information which config entries we need to load is taken from manage.js: function
save_ldap_config

```
'#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 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

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

16.3.4 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 *rest_auth*.

Now you can take a look at the several REST endpoints. This REST API is used to authenticate the users.

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": "privacyIDEA unknown"
}
```

This REST API is used to authenticate the users.

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.

Authentication endpoints

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
```

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)`

Validate endpoints

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.

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.

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"
}
```

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

System endpoints

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

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

Resolver endpoints

The resolver endpoints is a subset of the system endpoint.

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** – Only return resolvers of type (like passwdresolver..)

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

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

Realm endpoints**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 :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 “reso1_with_realm” and “reso2_with_realm” call:

```
POST /realm/newrealm HTTP/1.1
Host: example.com
Accept: application/json
Content-Length: 26
Content-Type: application/x-www-form-urlencoded

resolvers=reso1_with_realm, reso2_with_realm
```

Example response:

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

{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": {
      "added": ["reso1_with_realm", "reso2_with_realm"],
      "failed": []
    }
  },
  "version": "privacyIDEA unknown"
}
```

DELETE /realm/ (realm)

This call deletes the given realm.

Parameters

- **realm** – The name of the realm to delete

Return a json result with value=1 if deleting the realm was successful

Example request:

```
DELETE /realm/realm_to_delete HTTP/1.1
Host: example.com
Accept: application/json
```

Example response:

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

{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
```

```

    "value": 1
  },
  "version": "privacyIDEA unknown"
}

```

Default Realm endpoints

These endpoints are used to define the default realm, retrieve it and delete it.

DELETE /defaultrealm

This call deletes the default realm.

Return a json result with either 1 (success) or 0 (fail)

Example response:

```

{
  "id": 1,
  "jsonrpc": "2.0",
  "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)

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.

Token endpoint

POST /token/unassign

Unassign 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,iVBORw0KGgoAAAANSUhEUgAAAUoAA",
      "value": "seed://3132333435363738393031323334353637383930"
    },
    "serial": "OATH00096020"
  },
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": true
  },
  "version": "privacyIDEA 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 file 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?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 /token/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. You can call one of these URLs:

POST /token/delete?serial=<serial> DELETE /token/<serial> POST /token/delete/<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"
      }
    ]
  },
  "version": "privacyIDEA unknown"
}
```

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": "privacyIDEA 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”
- **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 “pol1” 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": "privacyIDEA 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 “poll” is created.

```
GET /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": {
      "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"
}

```

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 “pol1” is created.

```

DELETE /policy/pol1 HTTP/1.1
Host: example.com
Accept: application/json

```

Example response:

```

HTTP/1.0 200 OK
Content-Type: application/json

{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "status": true,
    "value": 1
  },
  "version": "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

16.3.5 Audit log

Base class

class `privacyidea.lib.auditmodules.base.AuditBase` (*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 iterator with elements that are not a dictionary, the audit module needs to provide this function, to convert the audit entry to a dictionary.

csv_generator (*param*)

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

16.3.6 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

Indices and tables

- *genindex*
- *modindex*
- *search*

/application

GET /application/, 139

/audit

GET /audit/, 109

GET /audit/(csvfile), 109

/auth

POST /auth, 110

/defaultrealm

GET /defaultrealm, 119

POST /defaultrealm/(realm), 119

DELETE /defaultrealm, 119

/machine

GET /machine/, 137

GET /machine/authitem, 136

GET /machine/authitem/(application), 138

GET /machine/token, 137

POST /machine/token, 136

POST /machine/tokenoption, 136

DELETE /machine/token/(serial)/(machineid)/(resolver)/(application), 138

/machineresolver

GET /machineresolver/, 135

GET /machineresolver/(resolver), 135

POST /machineresolver/(resolver), 135

POST /machineresolver/test, 135

DELETE /machineresolver/(resolver), 135

/policy

GET /policy, 129

GET /policy/(name), 133

GET /policy/check, 128

GET /policy/defs, 129

GET /policy/defs/(scope), 132

GET /policy/export/(export), 130

POST /policy/(name), 132

POST /policy/disable/(name), 130

POST /policy/enable/(name), 130

POST /policy/import/(filename), 131

DELETE /policy/(name), 134

/realm

GET /realm/, 117

POST /realm/(realm), 117

DELETE /realm/(realm), 118

/resolver

GET /resolver/, 116

GET /resolver/(resolver), 116

POST /resolver/(resolver), 116

POST /resolver/test, 116

DELETE /resolver/(resolver), 116

/system

GET /system/, 115

GET /system/(key), 115

POST /system/setConfig, 115

POST /system/setDefault, 114

DELETE /system/(key), 115

/token

GET /token/, 124

/token/(serial)

DELETE /token/(serial), 127

/token/assign

POST /token/assign, 120

/token/copypin

POST /token/copypin, 120

/token/copyuser

POST /token/copyuser, 120

/token/disable

POST /token/disable, [120](#)

POST /token/disable/(serial), [124](#)

/token/enable

POST /token/enable, [121](#)

POST /token/enable/(serial), [124](#)

/token/getserial

GET /token/getserial/(otp), [124](#)

/token/init

POST /token/init, [122](#)

/token/load

POST /token/load/(filename), [126](#)

/token/lost

POST /token/lost/(serial), [126](#)

/token/realm

POST /token/realm/(serial), [126](#)

/token/reset

POST /token/reset, [121](#)

POST /token/reset/(serial), [125](#)

/token/resync

POST /token/resync, [121](#)

POST /token/resync/(serial), [125](#)

/token/set

POST /token/set, [123](#)

POST /token/set/(serial), [126](#)

/token/setpin

POST /token/setpin, [121](#)

POST /token/setpin/(serial), [125](#)

/token/unassign

POST /token/unassign, [120](#)

/user

GET /user/, [127](#)

/validate

GET /validate/check, [113](#)

GET /validate/samlcheck, [112](#)

POST /validate/check, [114](#)

POST /validate/samlcheck, [112](#)

p

- `privacyidea.api`, 108
- `privacyidea.api.application`, 139
- `privacyidea.api.auth`, 110
- `privacyidea.api.lib.postpolicy`, 101
- `privacyidea.api.lib.prepolicy`, 99
- `privacyidea.api.machine`, 135
- `privacyidea.api.machineresolver`, 135
- `privacyidea.api.policy`, 128
- `privacyidea.api.realm`, 117
- `privacyidea.api.resolver`, 116
- `privacyidea.api.system`, 114
- `privacyidea.api.token`, 120
- `privacyidea.api.user`, 127
- `privacyidea.api.validate`, 111
- `privacyidea.lib`, 75
- `privacyidea.lib.auditmodules`, 139
- `privacyidea.lib.machines`, 141
- `privacyidea.lib.policy`, 95
- `privacyidea.lib.policydecorators`, 102
- `privacyidea.lib.resolvers`, 104
- `privacyidea.lib.token`, 85
- `privacyidea.lib.user`, 76
- `privacyidea.models`, 71

A

ACTION (class in `privacyidea.lib.policy`), 96
 ACTIONVALUE (class in `privacyidea.lib.policy`), 97
 Active Directory, 19, 20
 add_init_details() (`privacyidea.lib.tokenclass.TokenClass` method), 77
 add_to_log() (`privacyidea.lib.auditmodules.base.AuditBase` method), 139
 add_to_log() (`privacyidea.lib.auditmodules.sqlaudit.Audit` method), 140
 add_tokeninfo() (in module `privacyidea.lib.token`), 85
 add_tokeninfo() (`privacyidea.lib.tokenclass.TokenClass` method), 77
 Admin (class in `privacyidea.models`), 71
 ADMIN (`privacyidea.lib.policy.SCOPE` attribute), 98
 admin policies, 43
 and_() (in module `privacyidea.lib.token`), 85
 and_() (in module `privacyidea.models`), 74
 API, 108
 appliance, 29
 Application Plugins, 63
 ASSIGN (`privacyidea.lib.policy.ACTION` attribute), 96
 assign_token() (in module `privacyidea.lib.token`), 85
 Audit, 59
 Audit (class in `privacyidea.lib.auditmodules.sqlaudit`), 140
 AUDIT (`privacyidea.lib.policy.ACTION` attribute), 96
 AUDIT (`privacyidea.lib.policy.SCOPE` attribute), 98
 audit modules, 139
 audit_entry_to_dict() (`privacyidea.lib.auditmodules.base.AuditBase` method), 139
 AuditBase (class in `privacyidea.lib.auditmodules.base`), 139
 AUTH (`privacyidea.lib.policy.SCOPE` attribute), 98
 auth_otppin() (in module `privacyidea.lib.policydecorators`), 102
 auth_user_does_not_exist() (in module `privacyidea.lib.policydecorators`), 102
 auth_user_has_no_token() (in module `privacyidea.lib.policydecorators`), 103

auth_user_passthru() (in module `privacyidea.lib.policydecorators`), 103
 authenticate() (`privacyidea.lib.tokenclass.TokenClass` method), 77
 authenticating client, 26
 authentication policies, 50
 AUTHITEMS (`privacyidea.lib.policy.ACTION` attribute), 96
 authorization policies, 52
 AUTHZ (`privacyidea.lib.policy.SCOPE` attribute), 98
 auto_assign_token() (in module `privacyidea.lib.token`), 86
 AUTOASSIGN (`privacyidea.lib.policy.ACTION` attribute), 96
 autoassign() (in module `privacyidea.api.lib.postpolicy`), 101
 autoassignment, 54
 autoresync, 26

B

Backup, 31
 BaseMachineResolver (class in `privacyidea.lib.machines.base`), 141

C

Challenge (class in `privacyidea.models`), 71
 challenge_janitor() (`privacyidea.lib.tokenclass.TokenClass` method), 77
 check_auth_counter() (`privacyidea.lib.tokenclass.TokenClass` method), 77
 check_base_action() (in module `privacyidea.api.lib.prepolicy`), 99
 check_challenge_response() (`privacyidea.lib.tokenclass.TokenClass` method), 77
 check_max_token_realm() (in module `privacyidea.api.lib.prepolicy`), 99
 check_max_token_user() (in module `privacyidea.api.lib.prepolicy`), 100

- check_otp() (privacyidea.lib.tokenclass.TokenClass method), 78
 - check_otp_exist() (privacyidea.lib.tokenclass.TokenClass method), 78
 - check_otp_pin() (in module privacyidea.api.lib.prepolicy), 100
 - check_pin() (privacyidea.lib.tokenclass.TokenClass method), 78
 - check_serial() (in module privacyidea.api.lib.postpolicy), 101
 - check_serial() (in module privacyidea.lib.token), 86
 - check_serial_pass() (in module privacyidea.lib.token), 86
 - check_token_init() (in module privacyidea.api.lib.prepolicy), 100
 - check_token_list() (in module privacyidea.lib.token), 86
 - check_token_upload() (in module privacyidea.api.lib.prepolicy), 100
 - check_tokentype() (in module privacyidea.api.lib.postpolicy), 101
 - check_user_pass() (in module privacyidea.lib.token), 87
 - check_validity_period() (privacyidea.lib.tokenclass.TokenClass method), 79
 - checkPass() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver method), 107
 - checkPass() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver method), 105
 - checkPass() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver method), 104
 - checkUserId() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver method), 106
 - checkUserName() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver method), 106
 - cleanup_challenges() (in module privacyidea.models), 75
 - clear() (privacyidea.lib.auditmodules.sqlaudit.Audit method), 140
 - Clickatel, 28
 - client, 26
 - client machines, 61
 - client policies, 46
 - close() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver method), 104
 - Config (class in privacyidea.models), 72
 - config file, 13
 - config_lost_token() (in module privacyidea.lib.policydecorators), 103
 - configuration, 19
 - copy_token_pin() (in module privacyidea.lib.token), 87
 - copy_token_realms() (in module privacyidea.lib.token), 87
 - copy_token_user() (in module privacyidea.lib.token), 87
 - COPYTOKENPIN (privacyidea.lib.policy.ACTION attribute), 96
 - COPYTOKENUSER (privacyidea.lib.policy.ACTION attribute), 96
 - count window, 35
 - create_challenge() (privacyidea.lib.tokenclass.TokenClass method), 79
 - create_tokenclass_object() (in module privacyidea.lib.token), 87
 - csv_generator() (privacyidea.lib.auditmodules.base.AuditBase method), 139
 - csv_generator() (privacyidea.lib.auditmodules.sqlaudit.Audit method), 140
- ## D
- database, 71
 - DB2, 21
 - default realm, 23
 - del_info() (privacyidea.models.Token method), 73
 - del_tokeninfo() (privacyidea.lib.tokenclass.TokenClass method), 79
 - DELETE (privacyidea.lib.policy.ACTION attribute), 96
 - delete_policy() (in module privacyidea.lib.policy), 98
 - delete_token() (privacyidea.lib.tokenclass.TokenClass method), 79
 - DISABLE (privacyidea.lib.policy.ACTION attribute), 96
- ## E
- ENABLE (privacyidea.lib.policy.ACTION attribute), 96
 - enable() (privacyidea.lib.tokenclass.TokenClass method), 79
 - enable_policy() (in module privacyidea.lib.policy), 99
 - enable_token() (in module privacyidea.lib.token), 88
 - encrypt_pin() (in module privacyidea.api.lib.prepolicy), 100
 - ENCRYPTPIN (privacyidea.lib.policy.ACTION attribute), 96
 - ENROLL (privacyidea.lib.policy.SCOPE attribute), 98
 - enroll token, 36
 - enrollment policies, 53
 - export_policies() (in module privacyidea.lib.policy), 99
- ## F
- failcount, 35
 - finalize_log() (privacyidea.lib.auditmodules.base.AuditBase method), 139
 - finalize_log() (privacyidea.lib.auditmodules.sqlaudit.Audit method), 141
 - flatfile resolver, 19
 - FreeIPA, 20
 - FreeRADIUS, 63
- ## G
- gen_serial() (in module privacyidea.lib.token), 88

Get Serial (Determine Serial by OTP), 35

get() (privacyidea.models.Challenge method), 71

get() (privacyidea.models.Policy method), 72

get() (privacyidea.models.Token method), 73

get_action_values() (privacyidea.lib.policy.PolicyClass method), 98

get_all_token_users() (in module privacyidea.lib.token), 88

get_as_dict() (privacyidea.lib.tokenclass.TokenClass method), 79

get_audit_id() (privacyidea.lib.auditmodules.base.AuditBase method), 139

get_class_info() (privacyidea.lib.tokenclass.TokenClass class method), 79

get_class_prefix() (privacyidea.lib.tokenclass.TokenClass class method), 79

get_class_type() (privacyidea.lib.tokenclass.TokenClass class method), 79

get_config_description() (privacyidea.lib.machines.base.BaseMachineResolver class method), 141

get_count_auth() (privacyidea.lib.tokenclass.TokenClass method), 79

get_count_auth_max() (privacyidea.lib.tokenclass.TokenClass method), 79

get_count_auth_success() (privacyidea.lib.tokenclass.TokenClass method), 79

get_count_auth_success_max() (privacyidea.lib.tokenclass.TokenClass method), 79

get_count_window() (privacyidea.lib.tokenclass.TokenClass method), 79

get_dynamic_policy_definitions() (in module privacyidea.lib.token), 88

get_failcount() (privacyidea.lib.tokenclass.TokenClass method), 79

get_hashed_pin() (privacyidea.models.Token method), 73

get_hashlib() (privacyidea.lib.tokenclass.TokenClass class method), 80

get_info() (privacyidea.models.Token method), 73

get_init_detail() (privacyidea.lib.tokenclass.TokenClass method), 80

get_init_details() (privacyidea.lib.tokenclass.TokenClass method), 80

get_logout_time() (in module privacyidea.api.lib.postpolicy), 101

get_machine_id() (privacyidea.lib.machines.base.BaseMachineResolver method), 141

get_machine_id() (privacyidea.lib.machines.hosts.HostsMachineResolver method), 142

get_machineresolver_id() (in module privacyidea.models), 75

get_machines() (privacyidea.lib.machines.base.BaseMachineResolver method), 141

get_machines() (privacyidea.lib.machines.hosts.HostsMachineResolver method), 142

get_machinetoken_id() (in module privacyidea.models), 75

get_max_failcount() (privacyidea.lib.tokenclass.TokenClass method), 80

get_multi_otp() (in module privacyidea.lib.token), 88

get_multi_otp() (privacyidea.lib.tokenclass.TokenClass method), 80

get_num_tokens_in_realm() (in module privacyidea.lib.token), 88

get_otp() (in module privacyidea.lib.token), 89

get_otp() (privacyidea.lib.tokenclass.TokenClass method), 80

get_otp_count() (privacyidea.lib.tokenclass.TokenClass method), 80

get_otp_count_window() (privacyidea.lib.tokenclass.TokenClass method), 80

get_otp_status() (privacyidea.models.Challenge method), 72

get_otplen() (privacyidea.lib.tokenclass.TokenClass method), 80

get_pin_hash_seed() (privacyidea.lib.tokenclass.TokenClass method), 80

get_policies() (privacyidea.lib.policy.PolicyClass method), 98

get_QRimage_data() (privacyidea.lib.tokenclass.TokenClass method), 79

get_realms() (privacyidea.lib.tokenclass.TokenClass method), 80

get_realms() (privacyidea.models.Token method), 73

get_realms_of_token() (in module privacyidea.lib.token), 89

get_serial() (privacyidea.lib.tokenclass.TokenClass method), 80

get_serial_by_otp() (in module privacyidea.lib.token), 89

get_static_policy_definitions() (in module privacyidea.lib.policy), 99

get_sync_window() (privacyidea.lib.tokenclass.TokenClass method), 81

get_token_by_otp() (in module privacyidea.lib.token), 89

get_token_id() (in module privacyidea.models), 75

get_token_owner() (in module privacyidea.lib.token), 89

get_token_type() (in module privacyidea.lib.token), 90

[get_tokenclass_info\(\)](#) (in module `privacyidea.lib.token`), [90](#)
[get_tokeninfo\(\)](#) (`privacyidea.lib.tokenclass.TokenClass` method), [81](#)
[get_tokens\(\)](#) (in module `privacyidea.lib.token`), [90](#)
[get_tokens_in_resolver\(\)](#) (in module `privacyidea.lib.token`), [90](#)
[get_tokens_paginate\(\)](#) (in module `privacyidea.lib.token`), [90](#)
[get_tokenserial_of_transaction\(\)](#) (in module `privacyidea.lib.token`), [91](#)
[get_tokentype\(\)](#) (`privacyidea.lib.tokenclass.TokenClass` method), [81](#)
[get_total\(\)](#) (`privacyidea.lib.auditmodules.base.AuditBase` method), [140](#)
[get_total\(\)](#) (`privacyidea.lib.auditmodules.sqlaudit.Audit` method), [141](#)
[get_type\(\)](#) (`privacyidea.lib.tokenclass.TokenClass` method), [81](#)
[get_user\(\)](#) (`privacyidea.lib.tokenclass.TokenClass` method), [81](#)
[get_user_from_param\(\)](#) (in module `privacyidea.lib.user`), [76](#)
[get_user_id\(\)](#) (`privacyidea.lib.tokenclass.TokenClass` method), [81](#)
[get_user_info\(\)](#) (in module `privacyidea.lib.user`), [76](#)
[get_user_list\(\)](#) (in module `privacyidea.lib.user`), [76](#)
[get_user_pin\(\)](#) (`privacyidea.models.Token` method), [74](#)
[get_username\(\)](#) (in module `privacyidea.lib.user`), [76](#)
[get_validity_period_end\(\)](#) (`privacyidea.lib.tokenclass.TokenClass` method), [81](#)
[get_validity_period_start\(\)](#) (`privacyidea.lib.tokenclass.TokenClass` method), [81](#)
[get_vars\(\)](#) (`privacyidea.lib.tokenclass.TokenClass` method), [81](#)
[getResolverClassDescriptor\(\)](#) (`privacyidea.lib.resolvers.LDAPIdResolver.IdResolver` class method), [107](#)
[getResolverClassDescriptor\(\)](#) (`privacyidea.lib.resolvers.PasswdIdResolver.IdResolver` class method), [106](#)
[getResolverClassDescriptor\(\)](#) (`privacyidea.lib.resolvers.UserIdResolver.UserIdResolver` class method), [104](#)
[getResolverClassType\(\)](#) (`privacyidea.lib.resolvers.UserIdResolver.UserIdResolver` class method), [104](#)
[getResolverDescriptor\(\)](#) (`privacyidea.lib.resolvers.UserIdResolver.UserIdResolver` method), [104](#)
[getResolverId\(\)](#) (`privacyidea.lib.resolvers.LDAPIdResolver.IdResolver` method), [107](#)
[getResolverId\(\)](#) (`privacyidea.lib.resolvers.PasswdIdResolver.IdResolver` method), [106](#)
[getResolverId\(\)](#) (`privacyidea.lib.resolvers.UserIdResolver.UserIdResolver` method), [104](#)
[getResolverType\(\)](#) (`privacyidea.lib.resolvers.UserIdResolver.UserIdResolver` class method), [104](#)
[getSearchFields\(\)](#) (`privacyidea.lib.resolvers.PasswdIdResolver.IdResolver` method), [106](#)
[getserial](#), [45](#)
[GETSERIAL](#) (`privacyidea.lib.policy.ACTION` attribute), [96](#)
[GETTOKEN](#) (`privacyidea.lib.policy.SCOPE` attribute), [98](#)
[getUserId\(\)](#) (`privacyidea.lib.resolvers.LDAPIdResolver.IdResolver` method), [107](#)
[getUserId\(\)](#) (`privacyidea.lib.resolvers.PasswdIdResolver.IdResolver` method), [106](#)
[getUserId\(\)](#) (`privacyidea.lib.resolvers.UserIdResolver.UserIdResolver` method), [105](#)
[getUserInfo\(\)](#) (`privacyidea.lib.resolvers.LDAPIdResolver.IdResolver` method), [107](#)
[getUserInfo\(\)](#) (`privacyidea.lib.resolvers.PasswdIdResolver.IdResolver` method), [106](#)
[getUserInfo\(\)](#) (`privacyidea.lib.resolvers.UserIdResolver.UserIdResolver` method), [105](#)
[getUserList\(\)](#) (`privacyidea.lib.resolvers.LDAPIdResolver.IdResolver` method), [107](#)
[getUserList\(\)](#) (`privacyidea.lib.resolvers.PasswdIdResolver.IdResolver` method), [106](#)
[getUserList\(\)](#) (`privacyidea.lib.resolvers.UserIdResolver.UserIdResolver` method), [105](#)
[getUsername\(\)](#) (`privacyidea.lib.resolvers.LDAPIdResolver.IdResolver` method), [107](#)
[getUsername\(\)](#) (`privacyidea.lib.resolvers.PasswdIdResolver.IdResolver` method), [107](#)
[getUsername\(\)](#) (`privacyidea.lib.resolvers.UserIdResolver.UserIdResolver` method), [105](#)

H

[HostsMachineResolver](#) (class in `privacyidea.lib.machines.hosts`), [142](#)
[IdResolver](#) (class in `privacyidea.lib.resolvers.LDAPIdResolver`), [107](#)
[IdResolver](#) (class in `privacyidea.lib.resolvers.PasswdIdResolver`), [105](#)
[import](#), [69](#)
[IMPORT](#) (`privacyidea.lib.policy.ACTION` attribute), [96](#)
[import_policies\(\)](#) (in module `privacyidea.lib.policy`), [99](#)
[inc_count_auth\(\)](#) (`privacyidea.lib.tokenclass.TokenClass` method), [81](#)

[inc_count_auth_success\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [81](#)
[inc_failcount\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [81](#)
[inc_otp_counter\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [81](#)
[init_random_pin\(\)](#) (in module privacyidea.api.lib.prepolicy), [100](#)
[init_token\(\)](#) (in module privacyidea.lib.token), [91](#)
[init_tokenlabel\(\)](#) (in module privacyidea.api.lib.prepolicy), [100](#)
[initialize\(\)](#) (privacyidea.lib.auditmodules.base.AuditBase method), [140](#)
[initialize_log\(\)](#) (privacyidea.lib.auditmodules.base.AuditBase method), [140](#)
[is_active\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [81](#)
[is_challenge_request\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [81](#)
[is_challenge_response\(\)](#) (privacyidea.lib.tokenclass.TokenClass method), [82](#)
[is_token_active\(\)](#) (in module privacyidea.lib.token), [91](#)
[is_token_owner\(\)](#) (in module privacyidea.lib.token), [91](#)
[is_valid\(\)](#) (privacyidea.models.Challenge method), [72](#)

J

[JSON Web Token](#), [71](#)
[JWT](#), [71](#)

L

[LDAP](#), [19](#)
[LDAP resolver](#), [20](#)
[libpolicy](#) (class in privacyidea.lib.policydecorators), [103](#)
[library](#), [75](#)
[load_config\(\)](#) (privacyidea.lib.machines.base.BaseMachineResolver method), [142](#)
[load_config\(\)](#) (privacyidea.lib.machines.hosts.HostsMachineResolver method), [142](#)
[loadConfig\(\)](#) (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver method), [108](#)
[loadConfig\(\)](#) (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver method), [107](#)
[loadConfig\(\)](#) (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver method), [105](#)
[loadFile\(\)](#) (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver method), [107](#)
[log\(\)](#) (privacyidea.lib.auditmodules.base.AuditBase method), [140](#)
[log\(\)](#) (privacyidea.lib.auditmodules.sqlaudit.Audit method), [141](#)
[log_token_num\(\)](#) (privacyidea.lib.auditmodules.base.AuditBase method), [140](#)
[login mode](#), [55](#)
[Login Policy](#), [55](#)
[login_mode\(\)](#) (in module privacyidea.lib.policydecorators), [103](#)
[LOGINMODE](#) (class in privacyidea.lib.policy), [97](#)
[LOGINMODE](#) (privacyidea.lib.policy.ACTION attribute), [96](#)
[logout time](#), [56](#)
[LOGOUTTIME](#) (privacyidea.lib.policy.ACTION attribute), [96](#)
[Lost token](#), [33](#)
[lost token](#), [55](#)
[lost_token\(\)](#) (in module privacyidea.lib.token), [91](#)
[LOSTTOKEN](#) (privacyidea.lib.policy.ACTION attribute), [96](#)
[LOSTTOKENPWCONTENTS](#) (privacyidea.lib.policy.ACTION attribute), [96](#)
[LOSTTOKENPWLEN](#) (privacyidea.lib.policy.ACTION attribute), [96](#)
[LOSTTOKENVALID](#) (privacyidea.lib.policy.ACTION attribute), [96](#)

M

[Machine Resolvers](#), [141](#)
[MachineApplicationBase](#) (in module privacyidea.lib.applications), [95](#)
[MACHINELIST](#) (privacyidea.lib.policy.ACTION attribute), [96](#)
[MachineResolver](#) (class in privacyidea.models), [72](#)
[MachineResolverConfig](#) (class in privacyidea.models), [72](#)
[MACHINERESOLVERDELETE](#) (privacyidea.lib.policy.ACTION attribute), [96](#)
[MACHINERESOLVERWRITE](#) (privacyidea.lib.policy.ACTION attribute), [96](#)
[machines](#), [61](#)
[MachineToken](#) (class in privacyidea.models), [72](#)
[MachineTokenOptions](#) (class in privacyidea.models), [72](#)
[MACHINETOKENS](#) (privacyidea.lib.policy.ACTION attribute), [96](#)
[maxfail](#), [35](#)
[MAXTOKENREALM](#) (privacyidea.lib.policy.ACTION attribute), [97](#)
[MAXTOKENUSER](#) (privacyidea.lib.policy.ACTION attribute), [97](#)
[MethodsMixin](#) (class in privacyidea.models), [72](#)
[MySQL](#), [21](#)

N

[no_detail_on_fail\(\)](#) (in module privacyidea.api.lib.postpolicy), [101](#)

- no_detail_on_success() (in module privacyidea.api.lib.postpolicy), 102
- NODETAILFAIL (privacyidea.lib.policy.ACTION attribute), 97
- NODETAILSUCCESS (privacyidea.lib.policy.ACTION attribute), 97
- NONE (privacyidea.lib.policy.ACTIONVALUE attribute), 97
- Novell eDirectory, 20
- O**
- offline, 63
- offline_info() (in module privacyidea.api.lib.postpolicy), 102
- OpenLDAP, 20
- Oracle, 21
- OTP length, 35
- OTPPIN (privacyidea.lib.policy.ACTION attribute), 97
- OTPPINCONTENTS (privacyidea.lib.policy.ACTION attribute), 97
- OTPPINMAXLEN (privacyidea.lib.policy.ACTION attribute), 97
- OTPPINMINLEN (privacyidea.lib.policy.ACTION attribute), 97
- OTPPINRANDOM (privacyidea.lib.policy.ACTION attribute), 97
- OTRS, 63
- out of sync, 35
- override client, 26
- overview, 3
- P**
- PAM, 63
- pass on user no token, 26
- pass on user not found, 26
- PASSNOTOKEN (privacyidea.lib.policy.ACTION attribute), 97
- PASSNOUSER (privacyidea.lib.policy.ACTION attribute), 97
- passOnNoToken, 51
- passOnNoUser, 51
- passthru, 51
- PASSTHRU (privacyidea.lib.policy.ACTION attribute), 97
- Penrose, 20
- pip install, 5
- policies, 43
- Policy (class in privacyidea.models), 72
- PolicyClass (class in privacyidea.lib.policy), 98
- POLICYDELETE (privacyidea.lib.policy.ACTION attribute), 97
- POLICYWRITE (privacyidea.lib.policy.ACTION attribute), 97
- PostgreSQL, 21
- postpolicy (class in privacyidea.api.lib.postpolicy), 102
- prepolicy (class in privacyidea.api.lib.prepolicy), 100
- PRIVACYIDEA (privacyidea.lib.policy.LOGINMODE attribute), 97
- privacyidea.api (module), 108
- privacyidea.api.application (module), 139
- privacyidea.api.auth (module), 109, 110
- privacyidea.api.lib.postpolicy (module), 101
- privacyidea.api.lib.prepolicy (module), 99
- privacyidea.api.machine (module), 135
- privacyidea.api.machineresolver (module), 135
- privacyidea.api.policy (module), 128
- privacyidea.api.realm (module), 117
- privacyidea.api.resolver (module), 116
- privacyidea.api.system (module), 114
- privacyidea.api.token (module), 120
- privacyidea.api.user (module), 127
- privacyidea.api.validate (module), 111
- privacyidea.lib (module), 75
- privacyidea.lib.auditmodules (module), 139
- privacyidea.lib.machines (module), 141
- privacyidea.lib.policy (module), 95
- privacyidea.lib.policydecorators (module), 102
- privacyidea.lib.resolvers (module), 104
- privacyidea.lib.token (module), 85
- privacyidea.lib.user (module), 76
- privacyidea.models (module), 71
- R**
- read_keys() (privacyidea.lib.auditmodules.base.AuditBase method), 140
- Realm (class in privacyidea.models), 73
- realm administrator, 45
- realm autocreation, 24
- realm edit, 24
- realms, 23
- remove_token() (in module privacyidea.lib.token), 92
- RESET (privacyidea.lib.policy.ACTION attribute), 97
- reset() (privacyidea.lib.tokenclass.TokenClass method), 82
- reset_token() (in module privacyidea.lib.token), 92
- Resolver (class in privacyidea.models), 73
- ResolverConfig (class in privacyidea.models), 73
- RESOLVERDELETE (privacyidea.lib.policy.ACTION attribute), 97
- ResolverRealm (class in privacyidea.models), 73
- RESOLVERWRITE (privacyidea.lib.policy.ACTION attribute), 97
- REST, 108
- Restore, 31
- RESYNC (privacyidea.lib.policy.ACTION attribute), 97
- resync token, 36
- resync() (privacyidea.lib.tokenclass.TokenClass method), 82

resync_token() (in module privacyidea.lib.token), 92

S

SAML, 63

save() (privacyidea.lib.tokenclass.TokenClass method), 82

save() (privacyidea.models.TokenRealm method), 74

SCIM resolver, 23

scope, 43

SCOPE (class in privacyidea.lib.policy), 98

search() (privacyidea.lib.auditmodules.base.AuditBase method), 140

search() (privacyidea.lib.auditmodules.sqlaudit.Audit method), 141

search_query() (privacyidea.lib.auditmodules.base.AuditBase method), 140

searchQuery() (privacyidea.lib.auditmodules.sqlaudit.Audit method), 141

selfservice policies, 46

SERIAL (privacyidea.lib.policy.ACTION attribute), 97

SET (privacyidea.lib.policy.ACTION attribute), 97

set() (privacyidea.lib.auditmodules.base.AuditBase method), 140

set_count_auth() (in module privacyidea.lib.token), 92

set_count_auth() (privacyidea.lib.tokenclass.TokenClass method), 82

set_count_auth_max() (privacyidea.lib.tokenclass.TokenClass method), 82

set_count_auth_success() (privacyidea.lib.tokenclass.TokenClass method), 82

set_count_auth_success_max() (privacyidea.lib.tokenclass.TokenClass method), 82

set_count_window() (in module privacyidea.lib.token), 92

set_count_window() (privacyidea.lib.tokenclass.TokenClass method), 83

set_data() (privacyidea.models.Challenge method), 72

set_defaults() (in module privacyidea.lib.token), 93

set_defaults() (privacyidea.lib.tokenclass.TokenClass method), 83

set_description() (in module privacyidea.lib.token), 93

set_description() (privacyidea.lib.tokenclass.TokenClass method), 83

set_failcount() (privacyidea.lib.tokenclass.TokenClass method), 83

set_hashlib() (in module privacyidea.lib.token), 93

set_hashlib() (privacyidea.lib.tokenclass.TokenClass method), 83

set_info() (privacyidea.models.Token method), 74

set_init_details() (privacyidea.lib.tokenclass.TokenClass method), 83

set_max_failcount() (in module privacyidea.lib.token), 93

set_maxfail() (privacyidea.lib.tokenclass.TokenClass method), 83

set_otp_count() (privacyidea.lib.tokenclass.TokenClass method), 83

set_otpkey() (privacyidea.lib.tokenclass.TokenClass method), 83

set_otplen() (in module privacyidea.lib.token), 93

set_otplen() (privacyidea.lib.tokenclass.TokenClass method), 83

set_pin() (in module privacyidea.lib.token), 94

set_pin() (privacyidea.lib.tokenclass.TokenClass method), 83

set_pin() (privacyidea.models.Token method), 74

set_pin_hash_seed() (privacyidea.lib.tokenclass.TokenClass method), 83

set_pin_so() (in module privacyidea.lib.token), 94

set_pin_user() (in module privacyidea.lib.token), 94

set_policy() (in module privacyidea.lib.policy), 99

set_realm() (in module privacyidea.api.lib.prepolicy), 101

set_realms() (in module privacyidea.lib.token), 94

set_realms() (privacyidea.lib.tokenclass.TokenClass method), 83

set_realms() (privacyidea.models.Token method), 74

set_so_pin() (privacyidea.lib.tokenclass.TokenClass method), 83

set_so_pin() (privacyidea.models.Token method), 74

set_sync_window() (in module privacyidea.lib.token), 94

set_sync_window() (privacyidea.lib.tokenclass.TokenClass method), 83

set_tokeninfo() (privacyidea.lib.tokenclass.TokenClass method), 83

set_type() (privacyidea.lib.tokenclass.TokenClass method), 83

set_user() (privacyidea.lib.tokenclass.TokenClass method), 83

set_user_identifiers() (privacyidea.lib.tokenclass.TokenClass method), 83

set_user_pin() (privacyidea.lib.tokenclass.TokenClass method), 84

set_validity_period_end() (privacyidea.lib.tokenclass.TokenClass method), 84

set_validity_period_start() (privacyidea.lib.tokenclass.TokenClass method), 84

SETPIN (privacyidea.lib.policy.ACTION attribute), 97

SETREALM (privacyidea.lib.policy.ACTION attribute), 97

- setup tool, [29](#)
- setup() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver class method), [108](#)
- setup() (privacyidea.lib.resolvers.PasswdIdResolver.IdResolver class method), [107](#)
- Sipgate, [28](#)
- SMS automatic resend, [51](#)
- SMS Gateway, [28](#)
- SMS policy, [51](#)
- SMS text, [51](#)
- SMS token, [28](#)
- split_pin_pass() (privacyidea.lib.tokenclass.TokenClass method), [84](#)
- split_pin_pass() (privacyidea.models.Token method), [74](#)
- split_user() (in module privacyidea.lib.user), [76](#)
- SQL resolver, [21](#)
- sqlite, [21](#)
- status_validation_fail() (privacyidea.lib.tokenclass.TokenClass method), [84](#)
- status_validation_success() (privacyidea.lib.tokenclass.TokenClass method), [84](#)
- syncwindow, [35](#)
- system config, [25](#)
- SYSTEMDELETE (privacyidea.lib.policy.ACTION attribute), [97](#)
- SYSTEMWRITE (privacyidea.lib.policy.ACTION attribute), [97](#)
- T**
- testconnection() (privacyidea.lib.machines.base.BaseMachineResolver class method), [142](#)
- testconnection() (privacyidea.lib.machines.hosts.HostsMachineResolver class method), [142](#)
- testconnection() (privacyidea.lib.resolvers.LDAPIdResolver.IdResolver class method), [108](#)
- testconnection() (privacyidea.lib.resolvers.UserIdResolver.UserIdResolver class method), [105](#)
- token, [3](#)
- Token (class in privacyidea.models), [73](#)
- token configuration, [27](#)
- token default settings, [25](#)
- token description, [35](#)
- token_exist() (in module privacyidea.lib.token), [95](#)
- token_has_owner() (in module privacyidea.lib.token), [95](#)
- TokenClass (class in privacyidea.lib.tokenclass), [77](#)
- TokenInfo (class in privacyidea.models), [74](#)
- TOKENLABEL (privacyidea.lib.policy.ACTION attribute), [97](#)
- TOKENPIN (privacyidea.lib.policy.ACTIONVALUE attribute), [97](#)
- TokenRealm (class in privacyidea.models), [74](#)
- TOKENREALMS (privacyidea.lib.policy.ACTION attribute), [97](#)
- TOKENTYPE (privacyidea.lib.policy.ACTION attribute), [97](#)
- tokenview, [33](#)
- tools, [67](#)
- U**
- ubuntu, [6](#)
- UNASSIGN (privacyidea.lib.policy.ACTION attribute), [97](#)
- unassign_token() (in module privacyidea.lib.token), [95](#)
- update() (privacyidea.lib.tokenclass.TokenClass method), [84](#)
- update_otpkey() (privacyidea.models.Token method), [74](#)
- update_type() (privacyidea.models.Token method), [74](#)
- USER (privacyidea.lib.policy.SCOPE attribute), [98](#)
- user policies, [46](#)
- User() (in module privacyidea.lib.user), [76](#)
- UserIdResolver (class in privacyidea.lib.resolvers.UserIdResolver), [104](#)
- useridresolvers, [19](#), [104](#)
- USERLIST (privacyidea.lib.policy.ACTION attribute), [97](#)
- USERSTORE (privacyidea.lib.policy.ACTIONVALUE attribute), [97](#)
- USERSTORE (privacyidea.lib.policy.LOGINMODE attribute), [97](#)
- userview, [39](#)
- V**
- virtual environment, [5](#)
- W**
- WEBUI (privacyidea.lib.policy.SCOPE attribute), [98](#)
- WebUI Login, [55](#)
- WebUI Policy, [55](#)