Swauth Documentation

Release 1.3.0

OpenStack, LLC

Jul 27, 2018

Contents

An Auth Service for Swift as WSGI Middleware that uses Swift itself as a backing store. Docs at: https://swauth.readthedocs.io/ or ask in #openstack-swauth on freenode IRC (archive: http://eavesdrop.openstack.org/irclogs/%23openstack-swauth/).

Source available at: https://github.com/openstack/swauth

See also https://github.com/openstack/keystone for the standard OpenStack auth service.

CHAPTER 1

Overview

Before discussing how to install Swauth within a Swift system, it might help to understand how Swauth does it work first.

1. Swauth is middleware installed in the Swift Proxy's WSGI pipeline.
2. It intercepts requests to `/auth/` (by default).
3. It also uses Swift's authorize callback and acl callback features to authorize Swift requests.
4. Swauth will also make various internal calls to the Swift WSGI pipeline it's installed in to manipulate containers and objects within an `AUTH_.auth` (by default) Swift account. These containers and objects are what store account and user information.
5. Instead of #4, Swauth can be configured to call out to another remote Swauth to perform #4 on its behalf (using the swauth_remote config value).
6. When managing accounts and users with the various `swauth-` command line tools, these tools are actually just performing HTTP requests against the `/auth/` end point referenced in #2. You can make your own tools that use the same *API*.
7. In the special case of creating a new account, Swauth will do its usual WSGI-internal requests as per #4 but will also call out to the Swift cluster to create the actual Swift account.
    (a) This Swift cluster callout is an account PUT request to the URL defined by the `swift_default_cluster` config value.
    (b) This callout end point is also saved when the account is created so that it can be given to the users of that account in the future.
    (c) Sometimes, due to public/private network routing or firewalling, the URL Swauth should use should be different than the URL Swauth should give the users later. That is why the `default_swift_cluster` config value can accept two URLs (first is the one for users, second is the one for Swauth).
    (d) Once an account is created, the URL given to users for that account will not change, even if the `default_swift_cluster` config value changes. This is so that you can use multiple clusters with the same Swauth system; `default_swift_cluster` just points to the one where you want new users to go.

(e) You can change the stored URL for an account if need be with the `swauth-set-account-service` command line tool or a POST request (see *API*).

CHAPTER 2

Install

1. Install Swauth with `sudo python setup.py install` or `sudo python setup.py develop` or via whatever packaging system you may be using.
2. Alter your `proxy-server.conf` pipeline to have `swauth` instead of `tempauth`:

   Was:

   ```
   [pipeline:main]
   pipeline = catch_errors cache tempauth proxy-server
   ```

   Change To:

   ```
   [pipeline:main]
   pipeline = catch_errors cache swauth proxy-server
   ```

3. Add to your `proxy-server.conf` the section for the Swauth WSGI filter:

   ```
   [filter:swauth]
   use = egg:swauth#swauth
   set log_name = swauth
   super_admin_key = swauthkey
   default_swift_cluster = <your setting as discussed below>
   ```

   The `default_swift_cluster` setting can be confusing.

   (a) If you're using an all-in-one type configuration where everything will be run on the local host on port 8080, you can omit the `default_swift_cluster` completely and it will default to `local#http://127.0.0.1:8080/v1`.

   (b) If you're using a single Swift proxy you can just set the `default_swift_cluster = cluster_name#https://<public_ip>:<port>/v1` and that URL will be given to users as well as used by Swauth internally. (Quick note: be sure the `http` vs. `https` is set right depending on if you're using SSL.)

   (c) If you're using multiple Swift proxies behind a load balancer, you'll probably want `default_swift_cluster = cluster_name#https://`

   `<load_balancer_ip>:<port>/v1#http://127.0.0.1:<port>/v1` so that Swauth gives out the first URL but uses the second URL internally. Remember to double-check the `http` vs. `https` settings for each of the URLs; they might be different if you're terminating SSL at the load balancer.

   Also see the `proxy-server.conf-sample` for more config options, such as the ability to have a remote Swauth in a multiple Swift cluster configuration.

4. Be sure your Swift proxy allows account management in the `proxy-server.conf`:

   ```
   [app:proxy-server]
   ...
   allow_account_management = true
   ```

   For greater security, you can leave this off any public proxies and just have one or two private proxies with it turned on.

5. Restart your proxy server `swift-init proxy reload`
6. Initialize the Swauth backing store in Swift `swauth-prep -K swauthkey`
7. Add an account/user `swauth-add-user -A http[s]://<host>:<port>/auth/ -K swauthkey -a test tester testing`
8. Ensure it works `swift -A http[s]://<host>:<port>/auth/v1.0 -U test:tester -K testing stat -v`

If anything goes wrong, it's best to start checking the proxy server logs. The client command line utilities often don't get enough information to help. I will often just `tail -F` the appropriate proxy log (`/var/log/syslog` or however you have it configured) and then run the Swauth command to see exactly what requests are happening to try to determine where things fail.

General note, I find I occasionally just forget to reload the proxies after a config change; so that's the first thing you might try. Or, if you suspect the proxies aren't reloading properly, you might try `swift-init proxy stop`, ensure all the processes died, then `swift-init proxy start`.

Also, it's quite common to get the `/auth/v1.0` vs. just `/auth/` URL paths confused. Usual rule is: Swauth tools use just `/auth/` and Swift tools use `/auth/v1.0`.

CHAPTER 3

Web Admin Install

1. If you installed from packages, you'll need to cd to the webadmin directory the package installed. This is `/usr/share/doc/python-swauth/webadmin` with the Lucid packages. If you installed from source, you'll need to cd to the webadmin directory in the source directory.
2. Upload the Web Admin files with `swift -A http[s]://<host>:<port>/auth/v1.0 -U .super_admin:.super_admin -K swauthkey upload .webadmin .`
3. Open `http[s]://<host>:<port>/auth/` in your browser.

CHAPTER 4

Swift3 Middleware Compatibility

Swift3 middleware support has to be explicitly turned on in conf file using *s3_support* config option. It can easily be used with swauth when *auth_type* in swauth is configured to be *Plaintext* (default):

```
[pipeline:main]
pipeline = catch_errors cache swift3 swauth proxy-server

[filter:swauth]
use = egg:swauth#swauth
super_admin_key = swauthkey
s3_support = on
```

The AWS S3 client uses password in plaintext to compute HMAC signature When *auth_type* in swauth is configured to be *Sha1* or *Sha512*, swauth can only use the stored hashed password to compute HMAC signature. This results in signature mismatch although the user credentials are correct.

When *auth_type* is **not** *Plaintext*, the only way for S3 clients to authenticate is by giving SHA1/SHA512 of password as input to it's HMAC function. In this case, the S3 clients will have to know *auth_type* and *auth_type_salt* beforehand. Here is a sample configuration:

```
[pipeline:main]
pipeline = catch_errors cache swift3 swauth proxy-server

[filter:swauth]
use = egg:swauth#swauth
super_admin_key = swauthkey
s3_support = on
auth_type = Sha512
auth_type_salt = mysalt
```

Security Concern: Swauth stores user information (username, password hash, salt etc) as objects in the Swift cluster. If these backend objects which contain password hashes gets stolen, the intruder will be able to authenticate using the hash directly when S3 API is used.

CHAPTER 5

Contents

5.1 LICENSE

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

5.2 Implementation Details

The Swauth system is a scalable authentication and authorization system that uses Swift itself as its backing store. This section will describe how it stores its data.

Note: You can access Swauth's internal .auth account by using the account:user of .super_admin:.super_admin and the super admin key you have set in your configuration. Here's an example using *st* on a standard SAIO: `st -A http://127.0.0.1:8080/auth/v1.0 -U .super_admin:.super_admin -K swauthkey stat`

At the topmost level, the auth system has its own Swift account it stores its own account information within. This Swift account is known as self.auth_account in the code and its name is in the format self.reseller_prefix + ".auth". In this text, we'll refer to this account as <auth_account>.

The containers whose names do not begin with a period represent the accounts within the auth service. For example, the <auth_account>/test container would represent the "test" account.

The objects within each container represent the users for that auth service account. For example, the <auth_account>/test/bob object would represent the user "bob" within the auth service account of "test". Each of these user objects contain a JSON dictionary of the format:

```
{"auth": "<auth_type>:<auth_value>", "groups": <groups_array>}
```

The *<auth_type>* specifies how the user key is encoded. The default is *plaintext*, which saves the user's key in plaintext in the *<auth_value>* field. The value *sha1* is supported as well, which stores the user's key as a salted SHA1 hash. Note that using a one-way hash like SHA1 will likely inhibit future use of key-signing request types, assuming such support is added. The *<auth_type>* can be specified in the swauth section of the proxy server's config file, along with the salt value in the following way:

```
auth_type = <auth_type>
auth_type_salt = <salt-value>
```

Both fields are optional. auth_type defaults to *plaintext* and auth_type_salt defaults to "swauthsalt". Additional auth types can be implemented along with existing ones in the authtypes.py module.

The *<groups_array>* contains at least two groups. The first is a unique group identifying that user and it's name is of the format *<user>:<account>*. The second group is the *<account>* itself. Additional groups of *.admin* for account administrators and *.reseller_admin* for reseller administrators may exist. Here's an example user JSON dictionary:

```
{"auth": "plaintext:testing",
 "groups": [{"name": "test:tester"}, {"name": "test"}, {"name": ".admin"}]}
```

To map an auth service account to a Swift storage account, the Service Account Id string is stored in the *X-Container-Meta-Account-Id* header for the <auth_account>/<account> container. To map back the other way, an <auth_account>/.account_id/<account_id> object is created with the contents of the corresponding auth service's account name.

Also, to support a future where the auth service will support multiple Swift clusters or even multiple services for the same auth service account, an <auth_account>/<account>/.services object is created with its contents having a JSON dictionary of the format:

```
{"storage": {"default": "local", "local": <url>}}
```

The "default" is always "local" right now, and "local" is always the single Swift cluster URL; but in the future there can be more than one cluster with various names instead of just "local", and the "default" key's value will contain the primary cluster to use for that account. Also, there may be more services in addition to the current "storage" service right now.

Here's an example .services dictionary at the moment:

```
{"storage":
    {"default": "local",
     "local": "http://127.0.0.1:8080/v1/AUTH_8980f74b1cda41e483cbe0a925f448a9"}}
```

But, here's an example of what the dictionary may look like in the future:

```
{"storage":
    {"default": "dfw",
     "dfw": "http://dfw.storage.com:8080/v1/AUTH_8980f74b1cda41e483cbe0a925f448a9",
     "ord": "http://ord.storage.com:8080/v1/AUTH_8980f74b1cda41e483cbe0a925f448a9",
     "sat": "http://ord.storage.com:8080/v1/AUTH_8980f74b1cda41e483cbe0a925f448a9"},
 "servers":
    {"default": "dfw",
     "dfw": "http://dfw.servers.com:8080/v1/AUTH_8980f74b1cda41e483cbe0a925f448a9",
     "ord": "http://ord.servers.com:8080/v1/AUTH_8980f74b1cda41e483cbe0a925f448a9",
     "sat": "http://ord.servers.com:8080/v1/AUTH_8980f74b1cda41e483cbe0a925f448a9"}}
```

Lastly, the tokens themselves are stored as objects in the *<auth_account>/.token_[0-f]* containers. The names of the objects are the token strings themselves, such as *AUTH_tked86bbd01864458aa2bd746879438d5a*. The exact *.token_[0-f]* container chosen is based on the final digit of the token name, such as *.token_a* for the token *AUTH_tked86bbd01864458aa2bd746879438d5a*. The contents of the token objects are JSON dictionaries of the format:

```
{"account": <account>,
 "user": <user>,
 "account_id": <account_id>,
 "groups": <groups_array>,
 "expires": <time.time() value>}
```

The *<account>* is the auth service account's name for that token. The *<user>* is the user within the account for that token. The *<account_id>* is the same as the *X-Container-Meta-Account-Id* for the auth service's account, as described above. The *<groups_array>* is the user's groups, as described above with the user object. The "expires" value indicates when the token is no longer valid, as compared to Python's time.time() value.

Here's an example token object's JSON dictionary:

```
{"account": "test",
 "user": "tester",
 "account_id": "AUTH_8980f74b1cda41e483cbe0a925f448a9",
 "groups": [{"name": "test:tester"}, {"name": "test"}, {"name": ".admin"}],
 "expires": 1291273147.1624689}
```

To easily map a user to an already issued token, the token name is stored in the user object's *X-Object-Meta-Auth-Token* header.

Here is an example full listing of an <auth_account>:

```
.account_id
    AUTH_2282f516-559f-4966-b239-b5c88829e927
    AUTH_f6f57a3c-33b5-4e85-95a5-a801e67505c8
    AUTH_fea96a36-c177-4ca4-8c7e-b8c715d9d37b
.token_0
.token_1
.token_2
.token_3
.token_4
.token_5
.token_6
    AUTH_tk9d2941b13d524b268367116ef956dee6
.token_7
.token_8
    AUTH_tk93627c6324c64f78be746f1e6a4e3f98
.token_9
.token_a
.token_b
.token_c
.token_d
.token_e
    AUTH_tk0d37d286af2c43ffad06e99112b3ec4e
.token_f
    AUTH_tk766bbde93771489982d8dc76979d11cf
reseller
    .services
    reseller
test
    .services
    tester
    tester3
test2
    .services
    tester2
```

5.3 swauth

5.4 swauth.middleware

5.5 Swauth API

5.5.1 Overview

Swauth has its own internal versioned REST API for adding, removing, and editing accounts. This document explains the v2 API.

Authentication

Each REST request against the swauth API requires the inclusion of a specific authorization user and key to be passed in a specific HTTP header. These headers are defined as `X-Auth-Admin-User` and `X-Auth-Admin-Key`.

Typically, these values are `.super_admin` (the site super admin user) with the key being specified in the swauth middleware configuration as `super_admin_key`.

This could also be a reseller admin with the appropriate rights to perform actions on reseller accounts.

Endpoints

The swauth API endpoint is presented on the proxy servers, in the "/auth" namespace. In addition, the API is versioned, and the version documented is version 2. API versions subdivide the auth namespace by version, specified as a version identifier like "v2".

The auth endpoint described herein is therefore located at "/auth/v2/" as presented by the proxy servers.

Bear in mind that in order for the auth management API to be presented, it must be enabled in the proxy server config by setting `allow_account_managment` to `true` in the `[app:proxy-server]` stanza of your proxy-server.conf.

Responses

Responses from the auth APIs are returned as a JSON structure. Example return values in this document are edited for readability.

5.5.2 Reseller/Admin Services

Operations can be performed against the endpoint itself to perform general administrative operations. Currently, the only operations that can be performed is a GET operation to get reseller or site admin information.

Get Admin Info

A GET request at the swauth endpoint will return reseller information for the account specified in the `X-Auth-Admin-User` header. Currently, the information returned is limited to a list of accounts for the reseller or site admin.

Valid return codes:

- 200: Success
- 403: Invalid X-Auth-Admin-User/X-Auth-Admin-Key
- 5xx: Internal error

Example Request:

```
GET /auth/<api version>/ HTTP/1.1
X-Auth-Admin-User: .super_admin
X-Auth-Admin-Key: swauthkey
```

Example Curl Request:

```
curl -D - https://<endpoint>/auth/v2/ \
-H "X-Auth-Admin-User: .super_admin" \
-H "X-Auth-Admin-Key: swauthkey"
```

Example Result:

```
HTTP/1.1 200 OK

{ "accounts":
  [
    { "name": "account1" },
    { "name": "account2" },
    { "name": "account3" }
  ]
}
```

5.5.3 Account Services

There are API request to get account details, create, and delete accounts, mapping logically to the REST verbs GET, PUT, and DELETE. These actions are performed against an account URI, in the following general request structure:

```
METHOD /auth/<version>/<account> HTTP/1.1
```

The methods that can be used are detailed below.

Get Account Details

Account details can be retrieved by performing a GET request against an account URI. On success, a JSON dictionary will be returned containing the keys *account_id*, *services*, and *users*. The *account_id* is the value used when creating service accounts. The *services* value is a dict that represents valid storage cluster endpoints, and which endpoint is the default. The 'users' value is a list of dicts, each dict representing a user and currently only containing the single key 'name'.

Valid Responses:

- 200: Success
- 403: Invalid X-Auth-Admin-User/X-Auth-Admin-Key
- 5xx: Internal error

Example Request:

```
GET /auth/<api version>/<account> HTTP/1.1
X-Auth-Admin-User: .super_admin
X-Auth-Admin-Key: swauthkey
```

Example Curl Request:

```
curl -D - https://<endpoint>/auth/v2/<account> \
-H "X-Auth-Admin-User: .super_admin" \
-H "X-Auth-Admin-Key: swauthkey"
```

Example Response:

```
HTTP/1.1 200 OK

{ "services":
  { "storage":
    { "default": "local",
      "local": "https://<storage endpoint>/v1/<account_id>" }
  },
  "account_id": "<account_id>",
  "users": [ { "name": "user1" },
             { "name": "user2" } ]
}
```

Create Account

An account can be created with a PUT request against a non-existent account. By default, a newly created UUID4 will be used with the reseller prefix as the account ID used when creating corresponding service accounts. However, you can provide an X-Account-Suffix header to replace the UUDI4 part.

Valid return codes:

- 200: Success
- 403: Invalid X-Auth-Admin-User/X-Auth-Admin-Key
- 5xx: Internal error

Example Request:

```
PUT /auth/<api version>/<new_account> HTTP/1.1
X-Auth-Admin-User: .super_admin
X-Auth-Admin-Key: swauthkey
```

Example Curl Request:

```
curl -XPUT -D - https://<endpoint>/auth/v2/<new_account> \
-H "X-Auth-Admin-User: .super_admin" \
-H "X-Auth-Admin-Key: swauthkey"
```

Example Response:

```
HTTP/1.1 201 Created
```

Delete Account

An account can be deleted with a DELETE request against an existing account.

Valid Responses:

- 204: Success
- 403: Invalid X-Auth-Admin-User/X-Auth-Admin-Key
- 404: Account not found
- 5xx: Internal error

Example Request:

```
DELETE /auth/<api version>/<account> HTTP/1.1
X-Auth-Admin-User: .super_admin
X-Auth-Admin-Key: swauthkey
```

Example Curl Request:

```
curl -XDELETE -D - https://<endpoint>/auth/v2/<account> \
-H "X-Auth-Admin-User: .super_admin" \
-H "X-Auth-Admin-Key: swauthkey"
```

Example Response:

```
HTTP/1.1 204 No Content
```

5.5.4 User Services

Each account in swauth contains zero or more users. These users can be determined with the 'Get Account Details' API request against an account.

Users in an account can be created, modified, and detailed as described below by apply the appropriate REST verbs to a user URI, in the following general request structure:

```
METHOD /auth/<version>/<account>/<user> HTTP/1.1
```

The methods that can be used are detailed below.

Get User Details

User details can be retrieved by performing a GET request against a user URI. On success, a JSON dictionary will be returned as described:

```
{"groups": [  # List of groups the user is a member of
    {"name": "<act>:<usr>"},
        # The first group is a unique user identifier
    {"name": "<account>"},
        # The second group is the auth account name
    {"name": "<additional-group>"}
        # There may be additional groups, .admin being a
        # special group indicating an account admin and
        # .reseller_admin indicating a reseller admin.
 ],
 "auth": "<auth-type>:<key>"
 # The auth-type and key for the user; currently only
 # plaintext and sha1 are implemented as auth types.
}
```

For example:

```
{"groups": [{"name": "test:tester"}, {"name": "test"},
            {"name": ".admin"}],
 "auth": "plaintext:testing"}
```

Valid Responses:

- 200: Success
- 403: Invalid X-Auth-Admin-User/X-Auth-Admin-Key
- 404: Unknown account
- 5xx: Internal error

Example Request:

```
GET /auth/<api version>/<account>/<user> HTTP/1.1
X-Auth-Admin-User: .super_admin
X-Auth-Admin-Key: swauthkey
```

Example Curl Request:

```
curl -D - https://<endpoint>/auth/v2/<account>/<user> \
-H "X-Auth-Admin-User: .super_admin" \
-H "X-Auth-Admin-Key: swauthkey"
```

Example Response:

```
HTTP/1.1 200 Ok

{ "groups": [ { "name": "<account>:<user>" },
              { "name": "<account>" },
              { "name": ".admin" } ],
  "auth" : "plaintext:password" }
```

Create User

A user can be created with a PUT request against a non-existent user URI. The new user's password must be set using the `X-Auth-User-Key` header. The user name MUST NOT start with a period ('.'). This requirement is enforced by the API, and will result in a 400 error. Alternatively you can use `X-Auth-User-Key-Hash` header for providing already hashed password in format `<auth_type>:<hashed_password>`.

Optional Headers:

- `X-Auth-User-Admin: true`: create the user as an account admin
- `X-Auth-User-Reseller-Admin: true`: create the user as a reseller admin

Reseller admin accounts can only be created by the site admin, while regular accounts (or account admin accounts) can be created by an account admin, an appropriate reseller admin, or the site admin.

Note that PUT requests are idempotent, and the PUT request serves as both a request and modify action.

Valid Responses:

- 200: Success
- 400: Invalid request (missing required headers)
- 403: Invalid X-Auth-Admin-User/X-Auth-Admin-Key, or insufficient priv

- 404: Unknown account
- 5xx: Internal error

Example Request:

```
PUT /auth/<api version>/<account>/<user> HTTP/1.1
X-Auth-Admin-User: .super_admin
X-Auth-Admin-Key: swauthkey
X-Auth-User-Admin: true
X-Auth-User-Key: secret
```

Example Curl Request:

```
curl -XPUT -D - https://<endpoint>/auth/v2/<account>/<user> \
-H "X-Auth-Admin-User: .super_admin" \
-H "X-Auth-Admin-Key: swauthkey" \
-H "X-Auth-User-Admin: true" \
-H "X-Auth-User-Key: secret"
```

Example Response:

```
HTTP/1.1 201 Created
```

Delete User

A user can be deleted by performing a DELETE request against a user URI. This action can only be performed by an account admin, appropriate reseller admin, or site admin.

Valid Responses:

- 200: Success
- 403: Invalid X-Auth-Admin-User/X-Auth-Admin-Key, or insufficient priv
- 404: Unknown account or user
- 5xx: Internal error

Example Request:

```
DELETE /auth/<api version>/<account>/<user> HTTP/1.1
X-Auth-Admin-User: .super_admin
X-Auth-Admin-Key: swauthkey
```

Example Curl Request:

```
curl -XDELETE -D - https://<endpoint>/auth/v2/<account>/<user> \
-H "X-Auth-Admin-User: .super_admin" \
-H "X-Auth-Admin-Key: swauthkey"
```

Example Response:

```
HTTP/1.1 204 No Content
```

5.5.5 Other Services

There are several other swauth functions that can be performed, mostly done via "pseudo-user" accounts. These are well-known user names that are unable to be actually provisioned. These pseudo-users are described below.

Set Service Endpoints

Service endpoint information can be retrived using the Get Account Details API method.

This function allows setting values within this section for the <account>, allowing the addition of new service end points or updating existing ones by performing a POST to the URI corresponding to the pseudo-user ".services".

The body of the POST request should contain a JSON dict with the following format:

```
{"service_name": {"end_point_name": "end_point_value"}}
```

There can be multiple services and multiple end points in the same call.

Any new services or end points will be added to the existing set of services and end points. Any existing services with the same service name will be merged with the new end points. Any existing end points with the same end point name will have their values updated.

The updated services dictionary will be returned on success.

Valid Responses:

- 200: Success
- 403: Invalid X-Auth-Admin-User/X-Auth-Admin-Key
- 404: Account not found
- 5xx: Internal error

Example Request:

```
POST /auth/<api version>/<account>/.services HTTP/1.0
X-Auth-Admin-User: .super_admin
X-Auth-Admin-Key: swauthkey

{"storage": { "local": "<new endpoint>" }}
```

Example Curl Request:

```
curl -XPOST -D - https://<endpoint>/auth/v2/<account>/.services \
-H "X-Auth-Admin-User: .super_admin" \
-H "X-Auth-Admin-Key: swauthkey" --data-binary \
'{ "storage": { "local": "<new endpoint>" }}'
```

Example Response:

```
HTTP/1.1 200 OK

{"storage": {"default": "local", "local": "<new endpoint>" }}
```

Get Account Groups

Individual user group information can be retrieved using the *Get User Details* API method.

This function allows retrieving all group information for all users in an existing account. This can be achieved using a GET action against a user URI with the pseudo-user ".groups".

The JSON dictionary returned will be a "groups" dictionary similar to that documented in the *Get User Details* method, but representing the summary of all groups utilized by all active users in the account.

Valid Responses:

- 200: Success
- 403: Invalid X-Auth-Admin-User/X-Auth-Admin-Key
- 404: Account not found
- 5xx: Internal error

Example Request:

```
GET /auth/<api version>/<account>/.groups
X-Auth-Admin-User: .super_admin
X-Auth-Admin-Key: swauthkey
```

Example Curl Request:

```
curl -D - https://<endpoint>/auth/v2/<account>/.groups \
-H "X-Auth-Admin-User: .super_admin" \
-H "X-Auth-Admin-Key: swauthkey"
```

Example Response:

```
HTTP/1.1 200 OK

{ "groups": [ { "name": ".admin" },
              { "name": "<account>" },
              { "name": "<account>:user1" },
              { "name": "<account>:user2" } ] }
```

5.6 swauth.authtypes

This module hosts available auth types for encoding and matching user keys. For adding a new auth type, simply write a class that satisfies the following conditions:

- For the class name, capitalize first letter only. This makes sure the user can specify an all-lowercase config option such as "plaintext" or "sha1". Swauth takes care of capitalizing the first letter before instantiating it.
- Write an encode(key) method that will take a single argument, the user's key, and returns the encoded string. For plaintext, this would be "plaintext:<key>"
- Write a match(key, creds) method that will take two arguments: the user's key, and the user's retrieved credentials. Return a boolean value that indicates whether the match is True or False.

`swauth.authtypes.`**`MAX_TOKEN_LENGTH = 5000`**

Maximum length any valid token should ever be.

`class` `swauth.authtypes.`**`Plaintext`**

Bases: `object`

Provides a particular auth type for encoding format for encoding and matching user keys.

This class must be all lowercase except for the first character, which must be capitalized. encode and match methods must be provided and are the only ones that will be used by swauth.

`encode`(*key*)

Encodes a user key into a particular format. The result of this method will be used by swauth for storing user credentials.

Parameters **`key`** – User's secret key

Returns A string representing user credentials

match(*key*, *creds*, ***kwargs*)
Checks whether the user-provided key matches the user's credentials

Parameters

- **key** – User-supplied key
- **creds** – User's stored credentials
- **kwargs** – Extra keyword args for compatibility reason with other auth_type classes

Returns True if the supplied key is valid, False otherwise

validate(*auth_rest*)
Validate user credentials whether format is right for Plaintext

Parameters **auth_rest** – User credentials' part without auth_type

Returns Dict with a hash part of user credentials

Raises **ValueError** – If credentials' part has zero length

class swauth.authtypes.**Sha1**
Bases: `object`

Provides a particular auth type for encoding format for encoding and matching user keys.

This class must be all lowercase except for the first character, which must be capitalized. encode and match methods must be provided and are the only ones that will be used by swauth.

encode(*key*)
Encodes a user key into a particular format. The result of this method will be used by swauth for storing user credentials.

If salt is not manually set in conf file, a random salt will be generated and used.

Parameters **key** – User's secret key

Returns A string representing user credentials

encode_w_salt(*salt*, *key*)
Encodes a user key with salt into a particular format. The result of this method will be used internally.

Parameters

- **salt** – Salt for hashing
- **key** – User's secret key

Returns A string representing user credentials

match(*key*, *creds*, *salt*, ***kwargs*)
Checks whether the user-provided key matches the user's credentials

Parameters

- **key** – User-supplied key
- **creds** – User's stored credentials
- **salt** – Salt for hashing
- **kwargs** – Extra keyword args for compatibility reason with other auth_type classes

Returns True if the supplied key is valid, False otherwise

validate(*auth_rest*)
Validate user credentials whether format is right for Sha1

Parameters **auth_rest** – User credentials' part without auth_type

Returns Dict with a hash and a salt part of user credentials

Raises **ValueError** – If credentials' part doesn't contain delimiter between a salt and a hash.

class swauth.authtypes.**Sha512**
Bases: object

Provides a particular auth type for encoding format for encoding and matching user keys.

This class must be all lowercase except for the first character, which must be capitalized. encode and match methods must be provided and are the only ones that will be used by swauth.

encode(*key*)
Encodes a user key into a particular format. The result of this method will be used by swauth for storing user credentials.

If salt is not manually set in conf file, a random salt will be generated and used.

Parameters **key** – User's secret key

Returns A string representing user credentials

encode_w_salt(*salt*, *key*)
Encodes a user key with salt into a particular format. The result of this method will be used internal.

Parameters

- **salt** – Salt for hashing
- **key** – User's secret key

Returns A string representing user credentials

match(*key*, *creds*, *salt*, ***kwargs*)
Checks whether the user-provided key matches the user's credentials

Parameters

- **key** – User-supplied key
- **creds** – User's stored credentials
- **salt** – Salt for hashing
- **kwargs** – Extra keyword args for compatibility reason with other auth_type classes

Returns True if the supplied key is valid, False otherwise

validate(*auth_rest*)
Validate user credentials whether format is right for Sha512

Parameters **auth_rest** – User credentials' part without auth_type

Returns Dict with a hash and a salt part of user credentials

Raises **ValueError** – If credentials' part doesn't contain delimiter between a salt and a hash.

swauth.authtypes.**validate_creds**(*creds*)
Parse and validate user credentials whether format is right

Parameters **creds** – User credentials

Returns Auth_type class instance and parsed user credentials in dict

Raises **`ValueError`** – If credential format is wrong (eg: bad auth_type)

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index