DynaFed Technical Report

Louise Davies
louise.davies@stfc.ac.uk

August 6, 2018

Abstract

This document aims to record all of the knowledge gained from Louise Davies' 6 month graduate project that started January 2018. It documents the efforts made to create a new JSON configuration format to control authorisation for DynaFed users, the new Python plugin that was created to be able to authenticate and authorise users using LDAP, the efforts put into exploring Shibboleth as an authentication and authorisation option and instructions on how to install the new plugin, set up the configuration for it and how to use the management script to more easily maintain the JSON configuration files.

1 LDAP plugin

1.1 Config JSON specification

1.1.1 Analysing the existing solution

The original `gridmap_auth.py` plugin used a `authdb.json` file in order to specify permissions for paths in DynaFed. The JSON structure was suited for the exact task that the plugin needed to perform.

```
"scd":
{
 "disk":
 {
  "role":
  {
   "scduser": "rwld"
  },
  "clientname":
  {
   "/C=UK/O=eScience/OU=CLRC/L=RAL/CN=alastair dewhurst": "rwld",
   "/C=UK/O=eScience/OU=CLRC/L=RAL/CN=louise davies": "rwld"
  },
  "remoteaddr":
  {
   "127.0.0.1": "rwld"
  }
 }
},
```

Figure 1: The original, authdb.json file structure

Unfortunately, this exactness of its fit for purpose makes the format inflexible and inextensible to other use cases. Reasons for why the format is hard to be used generally are detailed below.

- Some configuration logic is hard coded into the Python plugin (e.g. the base federation prefix, waiving authorization checks on the base path)

- JSON assumes a basic path structure of `/VO/bucket`, meaning longer paths or protection on sub-buckets is not supported
- JSON only allows for three criteria to authorize users: role, clientname and remoteaddr. These are all checked for explicitly by name in the Python code, so filtering on other criteria is impossible without modifying the Python code
- More complicated expressions of requirements are not supported, so one can only specify permissions for a single attribute and can't express "the user must have both this attribute and this attribute" aka no AND and OR logic, instead it uses a straight mapping of attribute to permissions.

1.1.2 The new solution

```
{
 "endpoints": [
  {
   "endpoint_path": "/ldap/example",
   "allowed_attributes": [
    {
     "attribute_requirements": [
      {
       "attribute": "email",
       "value": "louise.davies@stfc.ac.uk"
      }
     ],
     "permissions": "rlw"
    },
    {
     "attribute_requirements": [
      {
       "attribute": "company",
       "value": "STFC"
      },
      {
       "attribute": "department",
       "value": "Sc"
      }
     ],
     "permissions": "rl"
    }
   ],
   "allowed_ip_addresses": [
    {
     "ip": "127.0.0.1",
     "permissions": "rlwd"
    }
   ]
   "propogate_permissions": true
  }
 ]
 "server": "fed.cclrc.ac.uk",
 "prefix": "/myfed"
}
```

Figure 2: New config

Figure 2 shows the new format created that aims to solve the shortcomings of the authdb format when trying to generalise. Rather than assuming the LDAP server or federation prefix, these are explicitly stated in the config at the top level. It would be easy to extend and add additional parameters to this top level, as the code already written only assumes structure in the `endpoints` field. Another change is that `endpoints` can specify any endpoint path and does not assume a fixed level of folder structure. Whilst nesting could have had advantages when matching paths and be less verbose, it would become cumbersome to parse for both machines and users trying to edit the file. Since the endpoints have nested elements themselves, things with different meanings would be on the same indentation level which would be confusing to readers. It would also be less resilient to changes in structures, whereas manually specifying paths would just involve editing the path names.

Each endpoint has a couple of parameters. Firstly, it has an `endpoint_path`. This is simply the path that it is protected within DynaFed. The `prefix` will be prepended to endpoint paths. The reason the base federation prefix was separated is to ease the ability to change the federation prefix without having to change all of the paths and to match DynaFed's internal idea of a path. Additionally, the original authdb format does not include the endpoint path in the config either, it instead parses it out in the Python plugin, so it makes the conversion from old to new easier.

The `propogate_permissions` field states whether the permissions states for the current path should propagate to any child path. The default is True (and `propogate_permissions` can be omitted when one wants the default, although for clarity it has been explicitly stated in the example), in that any paths under `/ldap/example` are protected by the same access controls. So, unless `/ldap/example/test.txt` has it's own endpoint specified in the config file, it will use the permissions stated within `/ldap/example`. This seems like what one would want as the default, in that protecting a directory protects all the files underneath with the same protections unless a file specifies its own permissions. The problem came when considering protecting the overall federation and allowing users to see the highest levels of folders, but not have permissions to files underneath. You require read permission to be able to see a directory, and list permission to be able to list its contents. However, just because we want all users to be able to list all the possible folders at the top level directory does not mean they should have read and list permissions to the entire federation. Since both the ability to propagate permissions and not have them propagate are required, a boolean for each endpoint was created to support both functions. This also means a couple of endpoints of essentially meaningless config are required in each config file to allow access to the top level folders like `/`, `/ldap` and `/x509` (how we split up the different authentication mechanisms), and any other folders you want all users to have read and/or list permissions to, with empty `attribute_requirements` (specifying a match on any user) and `propogate_permissions` set to `false`. In the original authdb format these top level paths were exempted from authorisation checks implicitly in the Python code, whereas the new format rquires that they are stated explicitly in the config.

`allowed_ip_addresses` is there to allow for access to specific IP addresses. Since this is the only client information we usually get other than clientname, and how it is different from the idea of a "user" and the attributes associated with users, it was decided to keep the allowed IP list separate from the allowed attributes list. It is a list of objects with the field `ip` which specifies the address and the field `permissions` that specify the permissions that IP address should have.

`allowed_attributes` is the most complex field in the config. As stated in the shortcomings of the authdb structure, there was no way to specify any sort of logical expressions on attributes. However, one can easily imagine requiring to specify that a user is both within a specific company with a specific role. This requires a logical AND mechanism, which would also mean we require an OR mechanism, since we still want to allow for multiple ways to access the same endpoint. This was done by creating the `allowed_attributes` structure, which is a list of attribute sets that specify the attributes required to satisfy that attribute set and the permissions granted to those who satisfy the set. This essentially means that separate items in the `allowed_attributes` list are ORed, and attributes in the `attribute_requirements` list are ANDed.

1.1.3 The revised new solution

The new configuration worked well. However, when the authdb format was first converted to its equivalent in the new format it went from approximately 200 lines to approximately 700 lines. Some of this extra verbosity was expected, but there were areas that were more verbose than they needed to be. The new method effectively optimises single AND statements, in that these require very little structure. However, as seen in Figure 1, the most common real use case is simply a list of allowed attributes, or an OR statement. An OR statement in the new format requires an attribute set per attribute, which means that permissions get repeated and more brackets are needed for the structure. It is not possible to swap the structures around and have an AND list of ORs, so a new approach was needed to reduce verbosity.

```
{
 "endpoints": [
  {
   "endpoint_path": "/ldap/test/authorised"
   "allowed_ip_addresses": [],
   "allowed_attributes": [
    {
     "attribute_requirements": {
      "or": [
       {
        "attribute": "cn",
        "value": "mnf98541"
       },
       {
        "attribute": "cn",
        "value": "other_username"
       }
      ]
     },
     "permissions": "rlwd"
    },
    {
     "attribute_requirements":{
      "attribute": "department",
      "value": "Sc"
     },
     "permissions": "rl"
    }
   ],
   "propogate_permissions": true
  },
 ],
 "server": "fed.cclrc.ac.uk",
 "prefix": "/myfed"
}
```

Figure 3: Revised new config

Figure 3 shows this new approach. Rather than `attribute_requirements` being a list of attribute sets, it is now an object I refer to as an attribute condition. In the simplest case, this is just an attribute name and value, but it can be an object detailing the logical operator (OR or AND), and a list of attribute conditions. This is a recursive structure, so within an attribute condition list, the entries can also be attribute conditions, meaning that complex logic like ORs of AND statements or ANDs of OR statements can be constructed, or they can just be attribute-value statements. Importantly, this means there is only a minimal level of nesting and less repetition of permissions in the use case of a simple OR statement with lots of attributes that can satisfy the OR. This is also extensible, as other operators such as NOT, NOR, NAND and XOR could be introduced if needed, as well as non-logical operators like mathematical operations (comparing an attribute vs a fixed value). Since the attribute conditions need to be parsed recursively the Python plugin code had to change to accommodate parsing this, but when testing timings there are no significant differences between the timings of this version of the plugin and the previous one that parsed the other format, provided that the level of nesting is reasonable.

A comparison between the new solution and the revised new solution can be found in Figure 20 at the end of the document.

1.2 Authorisation Plugin

1.2.1 Plugin code description

There are two stages to the authorisation plugin: what needs to be performed when the plugin is loaded by DynaFed on start up and what needs to be performed when a user makes a request. When the plugin is first loaded, the JSON config file is parsed and a list of all the endpoint paths is created, as well as setting up a persistent connection to the LDAP server and creating the LDAP entry cache. The rest of the plugin logic is performed when the user makes a request to DynaFed and the `is_allowed` function is called. There are essentially three important parts to the way the authorisation plugin authorises a user in the `is_allowed` function.

First, we call `auth_info_for_path`. Since the path the user is requesting might not have its authorisation information explicitly stated in the config file (instead it may inherit permissions from its parent/ancestor paths), the `auth_info_for_path` function takes the path that is being requested and returns the correct permission information. This is done by testing to see if the actual path itself if within the path list, otherwise the end component of the path is removed and we search for information on the parent of the previous path. For example, for the path `/ldap/test/authorised/Smudge.jpg`, we first test the path itself, then `/ldap/test/authorised`, `/ldap/test`, `/ldap` and finally `/`.

Once we have the relevant authorisation information for the path, we first check that we aren't violating `propogate_permissions`. We then check to see if the IP address of the client matches any of the IP addresses that may be listed within the authorisation information.

After that, the plugin performs the LDAP search. First the cache is checked to see if it contains an entry corresponding to the clientname and retrieves it if it exists. If it doesn't, then a LDAP Search request is sent to the server that retrieves all of the attributes associated with the user. This is then stored in the cache for later use.

Once we have the LDAP attributes, we then need to resolve the attribute conditions. This is done by calling `process_condition`, which recursively evaluates the attribute condition by seeing if it is an AND conditon (and thus checking that all sub attribute conditions are satisfied), OR condition (and thus checking that at least one of the sub conditions is satisfied) or an attribute-value pair which is compared against the LDAP entry. This then returns either True or False to indicate whether a match has occurred and this then then used to grant or deny access alongside checking the access mode matches the permission string.

1.2.2 Installation instructions

In order to set up the authorisation plugin to work with DynaFed, the plugin itself (`ldap_auth.py`) and the JSON configuration file (`ldap_auth.json`) must be placed in the DynaFed configuration folder, which is located at `/etc/ugr/conf.d`. The plugin then needs to be compiled by running `python -m py_compile ldap_auth.py`. The plugin uses some non-standard library plugins that needed to be installed on the Python instance, so `pip install ldap3 cachetools` needs to be run to install the relevant modules. `yum install python-pip` may be needed to install `pip` if your server does not already have it installed.

Next, `/etc/ugr/ugr.conf` must be modified to tell DynaFed where to find the plugin. Adding the line `glb.authorizationplugin[]: /usr/lib64/ugr/libugrauthplugin_python27.so ldapauth ldap_auth isallowed` to this file tells DynaFed the Python interpreter path, the name of the authorisation plugin (feel free to change this), the name of the Python file without the `.py` extension and the name of the function to call to authorise.

1.3 Apache configuration

In order to set up the Apache server to be able to authenticate users by both STFC LDAP and certificates, the `/etc/httpd/conf.d/zlcgdm-ugr-dav.conf` file needs to be modified to change how Apache authenticates. I will specify and explain all the changes required from the base DynaFed `zlcgdm-ugr-dav.conf` file so as to be a reference for all configuration, even though some of these changes (specifically setting up the certificate authentication) is also documented on TWiki.

Initially, `mod_ldap` needs to by installed by running `yum install mod_ldap`. The next instructions all need to have their lines placed outside any Location directives, typically a good place is at the top of the file. The modules may be loaded by other Apache configuration files, but to be sure adding the lines `LoadModule ldap_module modules/mod_ldap.so` and `LoadModule authnz_ldap_module modules/mod_authnz_ldap.so` loads the relevant modules needed. In my experience, I was only able to get the SSL for LDAP working when using the lines `LDAPTrustedMode SSL` and `LDAPVerifyServerCert off`. You can also use a plain connection (aka don't specify that we want to use SSL), or you can do what I do and just not verify the server certificate. I tried using `LDAPTrustedGlobalCert` and supplied the STFC Root Authority certificate, but this resulted in errors.

Next, we need to set up the certificates. Uncomment the lines in `zlcgdm-ugr-dav.conf` that turn on the 443 port and sets up a VirtualHost on port 443. The `SSLVerifyClient` line needs to be commented out, as this activates it for the whole server and we only want to apply it to specific paths. Additionally, ensure that the paths to the `SSLCertificateFile`, `SSLCertificateKeyFile` and `SSLCACertificatePath` point to the correct certificates and files needed to support certificate authentication.

Next, within the `Location /myfed` section (where `/myfed` is the relevant base federation prefix), it may be needed to add the `+LegacyDNStringFormat` option. This option sets the formatting of the client certificate name to start with the C attribute first and uses slashes as separators, rather than listing the C attribute last and using commas as separators. This is useful if your authorization configuration file uses the legacy format.

Now, the two locations for the authentication have to be created. These are two Location directives that activate different authentication mechanisms that lie directly under the main federation. This means that users can navigate to the URL that matches the way they want to authenticate. Below is the set up for the LDAP authentication.

```
<Location /myfed/ldap>
    AuthType Basic
    AuthName "stfc"
    AuthBasicProvider ldap
    AuthLDAPURL "ldap://fed.cclrc.ac.uk/dc=fed,dc=cclrc,dc=ac,dc=uk?cn?sub?(objectClass=person)"
    Require valid-user
</Location>
```

Everything is fairly self explanatory except for the `AuthLDAPURL`. This combines both the URL for the LDAP server as well as describing how to bind user credentials. `dc=fed,dc=cclrc,dc=ac,dc=uk` is the base DN, which is where all searches should start from. `cn` specifies the attribute to search for — this means whatever attribute is used as the username in the LDAP server, which in this case is `cn`. `sub` is the scope of the search and is the default option and it tells the server that we want to search the whole directory tree for a match. `(objectClass=person)` is a filter, which is an extra attribute that must be satisfied by the search result.

```
<Location /myfed/x509>
    SSLVerifyClient optional
    SSLVerifyDepth  10
    Require ssl-verify-client
</Location>
```

Here, we active the SSL client certificate authentication. This is effectively the same as the code removed from the overall SSL block. `Require ssl-verify-client` makes it work better in conjunction with other authentication mechanisms.

1.4 Endpoint configuration

Creating endpoints is not very different than what is being done in DynaFed currently. The only change is the requirement to have duplicate endpoints, of which each is "mounted" to a different authentication prefix. An example endpoint is given in Figure 4, in which the S3 keys need to be inserted as well as the correct Echo bucket name, plugin name and path to mount to in DynaFed (all represented by `example` in Figure 4)

It may be the case that when trying to upload files using DynaFed that signing errors are given in response. This is due to the mismatch between the S3 v2 and v4 signing algorithm. Echo wants to use signature v4 but DynaFed by default will only use v2. This can be fixed by adding a region to the endpoint configuration file (e.g. `locplugin.plugin_name.s3.region: uk`). The region can be set to anything, but the presence of a region will cause it to use v4 instead, likely due to v4 of the algorithm requiring a region to be specified.

```
# Plugin for example bucket (ldap)
glb.locplugin[]: /usr/lib64/ugr/libugrlocplugin_s3.so example-ldap 15
   s3s://s3.echo.stfc.ac.uk/example
locplugin.example-ldap.xlatepfx: /ldap/example /
locplugin.example-ldap.s3.priv_key: INSERT_PRIVATE_KEY_HERE
locplugin.example-ldap.s3.pub_key: INSERT_PUBLIC_KEY_HERE
locplugin.example-ldap.writable: true
locplugin.example-ldap.s3.alternate: true
locplugin.example-ldap.ca_path: /etc/grid-security/certificates/
locplugin.example-ldap.s3.region: uk

# Plugin for example bucket (x509)
glb.locplugin[]: /usr/lib64/ugr/libugrlocplugin_s3.so example-x509 15
   s3s://s3.echo.stfc.ac.uk/example
locplugin.example-x509.xlatepfx: /x509/example /
locplugin.example-x509.s3.priv_key: INSERT_PRIVATE_KEY_HERE
locplugin.example-x509.s3.pub_key: INSERT_PUBLIC_KEY_HERE
locplugin.example-x509.writable: true
locplugin.example-x509.s3.alternate: true
locplugin.example-x509.ca_path: /etc/grid-security/certificates/
locplugin.example-x509.s3.region: uk
```

Figure 4: Example DynaFed endpoint configuration for an S3 bucket

1.5 Misc.

1.5.1 Version 0.19.0 of LCGDM-DAV

If you are getting errors abotu invalud Authorization headers and you are using LCGDM-DAV 1.19.0, then you will either have to upgrade to 0.20.0 (might not be released on EPEL yet) or downgrade to 0.18.2, as due to an addition in 1.19.0 the HTTP Authorization header gets overwritten and thus HTTP Basic Auth doesn't work.

1.5.2 Enabling CORS on Echo buckets

In order for the DynaFed interface to be able to upload files to Echo, Cross-Origin Resource Sharing (CORS) rules must be added to each bucket in order for them to allow the PUT HTTP method as well as GET. A simple Python script that contains the correct CORSRule for a bucket and can add this rule to a bucket is given in Figure 5.

2 Shibboleth plugin

2.1 Setting up Shibboleth

The Shibboleth documentation in general is sufficient and requires very little deviation from. However, for completeness, the installation and setups steps will be written here.

The first step is to install Shibboleth. This uses steps from `https://wiki.shibboleth.net/confluence/display/SP3/RPMInstall`. First, the Shibboleth repo must be added to `/etc/yum.repod.d/`, the contents of `shibboleth.repo` are listed in Figure 6. Then, once the repository has been added, `yum install shibboleth` can be run to install Shibboleth.

After Shibboleth has been installed, the Shibboleth daemon needs to be run by running `/sbin/service shibd start`.

Next, the Shibboleth configuration for the file `/etc/shibboleth/shibboleth2.xml` needs to be configured to reflect your service. The `entityID` attribute in `<ApplicationDefaults>` needs to be changed to a unique URI, one that will be persistent and that you own the domain for. The rest of the configuration will be federation specific, and usually they will provide extensive instructions on how to set up the metadata for your service provider. The UK Access Management Federation for Education and Research is the one most useful to STFC, and the instructions to register as a service provider (which

```
#!/usr/bin/env python
import boto3

bucket = "test"
s3_server = "https://ceph-res-gw1.fds.rl.ac.uk"
dynafed_server = "https://dynafed.stfc.ac.uk"

session = boto3.session.Session()
s3_client = session.client(
  service_name="s3",
  endpoint_url=s3_server,
  verify=False
)

cors_rule = {
  "CORSRules": [
    {
      "AllowedMethods": ["GET", "PUT"],
      "AllowedOrigins": [dynafed_server],
      "MaxAgeSeconds": 3000
    }
  ]
}

s3_client.put_bucket_cors(Bucket=bucket, CORSConfiguration=cors_rule)
```

Figure 5: A script that applies the correct CORS rules to an S3 bucket

```
[shibboleth]
name=Shibboleth (CentOS_7)
# Please report any problems to https://issues.shibboleth.net
type=rpm-md
mirrorlist=https://shibboleth.net/cgi-bin/mirrorlist.cgi/CentOS_7
gpgcheck=1
gpgkey=https://download.opensuse.org/repositories/security:/shibboleth/
   CentOS_7/repodata/repomd.xml.key
enabled=1
```

Figure 6: The `shibboleth.repo` repository file

also includes configuration instructions) are found at `https://www.ukfederation.org.uk/content/Documents/Register2SP`. For testing before joining a federation, it is useful to use the TestShib service (`http://www.testshib.org/`).

2.2 Apache configuration

The rest of the configuration to get Shibboleth working with DynaFed is done in the Apache configuration files. Setting up the authentication for Shibboleth mostly just requires setting up the authentication endpoint similarly to the certificate and LDAP endpoints. The `ShibRequestSetting` is stating to Shibboleth to require a user to authenticate by setting the `requireSession` flag to be true.

```
<Location /myfed/shib>
    AuthType shibboleth
    ShibRequestSetting requireSession 1
    Require shib-session
</Location>
```

```
static int dav_shared_pass_env_var(void *rec, const char *key, const
    char *value)
{

  if (strncmp(key, "SHIB_", strlen("SHIB_")) == 0) {

    pass_ctx_t *ctx = (pass_ctx_t*)rec;

    const char *env_var_key = apr_pstrcat(ctx->pool, "env.", key, NULL)
        ;

    dmlite_any *any_value = dmlite_any_new_string(value);
    dmlite_any_dict_insert(ctx->creds->extra, env_var_key, any_value);
    dmlite_any_free(any_value);
  }

  return 1;
}

static void dav_shared_pass_env_vars(apr_pool_t *pool,
    dmlite_credentials *creds, request_rec *r)
{
  pass_ctx_t ctx = {creds, pool};
  apr_table_do(dav_shared_pass_env_var, &ctx, r->subprocess_env, NULL);
}
```

Figure 7: Example functions to be added to `shared/security.c` to enable the passing of environment variables to DynaFed

Additionally, `ShibCompatValidUser On` may be required to be placed at the top level of the config file to not cause conflicts with the other authentication methods when specifying the `Require` directive.

2.3 Shibboleth attributes and DynaFed

Shibboleth stores user attributes in Apache server environment variables. Although they can be stored in request headers, this method is liable to header spoofing and requires more careful security set up to the point that the official recommended advice is to never use headers when your server supports environment variables. The problem is that DynaFed does not pass through server variables to the authorisation plugins, whereas it can pass through headers. If one wants to go into production with a Shibboleth authenticated system then it is heavily recommended to ask the DynaFed/LCGDM team to try and incorporate passing environment variables into the program.

Initial explorations on how to pass Apache environment variables show that they can be passed through in a similar manner to headers and query strings. This is performed in the LCGDM-DAV code, in the file `shared/security.c`. Figure 7 is an example of code that can place environment variables in the shared security context passed to DynaFed. It prefixes environment variables with the `env.` prefix, mirroring how headers are prefixed with the `http.` prefix. The only difficulty is filtering server environment variables to add to the security context, as concerned were raised when discussing this with Fabrizio Furano about too many variables being added and affecting performance, and that being able to filter what environment variables to add would help. However, since this code is in `shared` and the settings are parsed in individual folders such as `lcgdm_dav_ns`, it seems difficult to be able to pass a filter string such as `SHIB_` as done in Figure 7. The `SHIB_` prefix is appended to the environment variables by setting the `attributePrefix` attribute in `<ApplicationDefaults>` in `/etc/shibboleth/shibboleth2.xml`.

3 Management script

This section details the functionality and usage instructions for the `manage_ldap_json.py` script. The script was created with the aim to provide functions that allow quicker ways to edit and manage the JSON configuration file that controls the user access management for the DynaFed plugins.

The script is a command line tool that offers various subcommands that perform various operations pertaining to the JSON config file. Figure 8 shows a list of all the subcommands and a brief description of the functionality of each command.

Most functions require the name of a config file to operate on (all except `convert`) and this dictates the location of the configuration file to operate on. This file is given by using the `-f, --file` flag.

```
verify      Verify that the JSON file is valid.
list        List all endpoints in file
info        Get the configuration information for an endpoint
add         Add a new endpoint to the authorisation file
new_ldap    Create a new LDAP protected DynaFed bucket and generate all
    the configuration for it
new_x509    Create a new X509 protected DynaFed bucket and generate all
    the configuration for it
addusers    Add a new users to an endpoint in the authorisation file
remove      Remove a new endpoint to the authorisation file
server      Get the name of the LDAP server or provide a new URI and
   set a new LDAP server
prefix      Get the federation prefix for DynaFed or provide a new
   prefix. This will be prepended to all endpoints
convert     Convert an old style AuthDB json file into LDAP config json
```

Figure 8: A list of the valid subcommands and a brief description of their function. This is what is outputted when using the help flag.

3.1 `verify` command

The `verify` command checks that the config file is both valid JSON and that it is correctly formatted so that it can be read properly by the authorisation plugins. It takes no arguments and only takes the file option described at the beginning of this section that specifies the file to verify. It will output an error message that aims to point to the particular problem and part of the file the problem is located, but it cannot show line numbers since the JSON is not parsed line by line, but instead by objects. It will instead output indices pointing to the relevant structure.

```
python manage_ldap_json.py -f /etc/grid-security/ldap_auth.json verify
```

Figure 9: Example `verify` commands

3.2 `list` command

The `list` command simply prints all of the endpoints that are currently contained within the file. It takes no arguments or options other than the file option that every function can take.

```
python manage_ldap_json.py -f /etc/grid-security/x509_auth.json list
```

Figure 10: Example `list` commands

3.3 `info` command

The `info` command takes an endpoint path as an argument and prints out the configuration information for that endpoint. This saves opening the file itself and getting confused by other endpoints listed in the file.

```
python manage_ldap_json.py -f /etc/grid-security/ldap_auth.json info /
   ldap/example
```

Figure 11: Example `info` commands

3.4 `add` command

The `add` command walks the user through creating a new endpoint from scratch and adding authorisation information to it. It takes the name of the endpoint path to be created as an argument. It then goes through a process of prompting the user for input via the command line to modify the endpoint configuration. The user is then presented with the finished endpoint and has to confirm that they want to add it to the file.

```
python manage_ldap_json.py -f /etc/grid-security/x509_auth.json add /
   x509/example
```

Figure 12: Example `add` commands

3.5 `new_ldap` command

The `new` command generates all the configuration needed to add a new LDAP bucket to DynaFed. It has three basic steps: first it creates the bucket (currently it doesn't do this, so you will have to create the bucket manually), then it creates the DynaFed endpoint configuration files and then it updates the LDAP JSON authorisation file to control access to the newly created bucket.

It takes the name of the new bucket to be created as a positional argument. It also requires the username attribute to be matched against for the LDAP configuration using the `-u, --username-attr` flag, in the STFC LDAP server this is `cn`. Next, you can supply a list of `r, --read-users` and `-w, --write-users` which will grant the usernames you supply `rl` and `rlwd` permissions respectively.

The name of the Ceph/Amazon S3 server the bucket is located in (`-c, --ceph-server`), and the public and private access keys (`--public` and `--private`) also have to be supplied so the script can generate the endpoint config that lets DynaFed access the bucket.

```
sudo python manage_ldap_json.py new_ldap newtestbucket --file /etc/grid
   -security/ldap_auth.json --read-users readuser1 readuser2 --write-
   users writeuser --username-attr cn --ceph-server https://ceph-res-
   gw1.fds.rl.ac.uk --public-key public --private-key private
sudo python manage_ldap_json.py new newtestbucket -f /etc/grid-security
   /ldap_auth.json -r readuser1 readuser2 -w writeuser -u cn -c https
   ://ceph-res-gw1.fds.rl.ac.uk --public-key public --private-key
   private
```

Figure 13: Example `new_ldap` commands

3.6 `new_x509` command

The `new` command generates all the configuration needed to add a new X509 bucket to DynaFed. It has three basic steps: first it creates the bucket (currently it doesn't do this, so you will have to create the bucket manually), then it creates the DynaFed endpoint configuration files and then it updates the X509 JSON authorisation file to control access to the newly created bucket.

It takes the name of the new bucket to be created as a positional argument. Next, you can supply a list of `r, --read-users` and `-w, --write-users` which will grant the usernames you supply `rl` and `rlwd` permissions respectively.

The name of the Ceph/Amazon S3 server the bucket is located in (`-c, --ceph-server`), and the public and private access keys (`--public` and `--private`) also have to be supplied so the script can generate the endpoint config that lets DynaFed access the bucket.

```
  sudo python manage_ldap_json.py new_x509 newtestbucket --file /etc/
     grid-security/x509_auth.json --read-users readuser1 readuser2 --
     write-users writeuser --username-attr cn --ceph-server https://
     ceph-res-gw1.fds.rl.ac.uk --public-key public --private-key
     private
  sudo python manage_ldap_json.py new newtestbucket -f /etc/grid-
     security/x509_auth.json -r readuser1 readuser2 -w writeuser -u cn
     -c https://ceph-res-gw1.fds.rl.ac.uk --public-key public --private
     -key private
```

Figure 14: Example `new_x509` commands

3.7 `addusers` command

The `addusers` command adds lists of users to a specific bucket in the config. It takes the endpoint name as a positional argument, and requires you to specify the `-u, --username-attr` flag so that the correct attribute name can be placed into the file. For STFC LDAP this should be `cn` and for x509 certificates this should be `clientname`. Next, you can supply a list of `r, --read-users` and `-w, --write-users` which will grant the usernames you supply `rl` and `rlwd` permissions respectively.

```
python manage_ldap_json.py -f /etc/grid-security/ldap_auth.json
   addusers /ldap/example --username-attr cn --read-users readuser1
   readuser2 --write-users writeuser1 writeuser2
python manage_ldap_json.py -f /etc/grid-security/x509_auth.json
   addusers /ldap/example -u clientname -r readuser1 readuser2 -w
   writeuser1 writeuser2
```

Figure 15: Example `addusers` commands

3.8 `remove` command

The `add` command deletes the authorisation information for an endpoint. It takes the name of the endpoint path to be created as an argument. The user is then presented with the endpoint information and has to confirm that they want to remove it from the file.

```
python manage_ldap_json.py -f /etc/grid-security/ldap_auth.json remove
   /ldap/example
```

Figure 16: Example `remove` commands

3.9 `server` command

The `server` command can either list the current value of the server field or you can supply a new value as an argument and it will change the value in the config file. The server field is used to supply the authorisation plugin the server URL it needs to connect to. For the LDAP plugin, this is the LDAP server. If the plugin does not require a server, then the field can be left blank.

```
python manage_ldap_json.py -f /etc/grid-security/ldap_auth.json server
python manage_ldap_json.py -f /etc/grid-security/ldap_auth.json server
   example-server.stfc.ac.uk
```

Figure 17: Example `server` commands

3.10 `prefix` command

The `server` command can either list the current value of the prefix field or you can supply a new value as an argument and it will change the value in the config file. The prefix field is used to specify the base federation prefix (e.g. `/myfed`, `/gridpp`) to the authorisation plugin. It will then automatically prepend all the paths with the federation prefix.

```
python manage_ldap_json.py -f /etc/grid-security/x509_auth.json prefix
python manage_ldap_json.py -f /etc/grid-security/x509_auth.json prefix
   /gridpp
```

Figure 18: Example `prefix` commands

3.11 `convert` command

The `convert` command converts the old `authdb.json` format into the new JSON config format. It takes two positional arguments, the first is the path of the `authdb.json` file to convert and the second is the path where you want to write the converted output to. It also requires two options: the base prefix which states the base federation prefix so it can be added to the config, and the bucket prefix which is prepended to all paths and is used to add an authentication prefix. Either of these two options can be specified with a leading slash or with the leading slash omitted.

```
python manage_ldap_json.py convert authdb.json x509_auth.json --base-
   prefix myfed --bucket-prefix x509
python manage_ldap_json.py convert --base-prefix /gridpp --bucket-
   prefix /x509 authdb.json x509_auth.json
```

Figure 19: Example `convert` commands

```
"allowed_attributes": [
 {
  "attribute_requirements": [
   {
    "attribute": "role",
    "value": "enmruser"
   }
  ],
  "permissions": "rwld"
 },
 {
  "attribute_requirements": [
   {
    "attribute": "clientname",
    "value": "andrew lahiff"
   }
  ],
  "permissions": "rwld"
 },
 {
  "attribute_requirements": [
   {
    "attribute": "clientname",
    "value": "alastair dewhurst"
   }
  ],
  "permissions": "rwld"
 },
 {
  "attribute_requirements": [
   {
    "attribute": "clientname",
    "value": "alexander dibbo"
   }
  ],
  "permissions": "rwld"
 },
 {
  "attribute_requirements": [
   {
    "attribute": "clientname",
    "value": "catalin condurache"
   }
  ],
  "permissions": "rwld"
 }
]
```

```
"allowed_attributes": [
 {
  "attribute_requirements": {
   "or": [
    {
     "attribute": "role",
     "value": "enmruser"
    },
    {
     "attribute": "clientname",
     "value": alastair dewhurst"
    },
    {
     "attribute": "clientname",
     "value": "catalin
        condurache"
    },
    {
     "attribute": "clientname",
     "value": "alexander dibbo"
    },
    {
     "attribute": "clientname",
     "value": "andrew lahiff"
    }
   ]
  },
  "permissions": "rwld"
 }
]
```

Figure 20: A comparison of the way to represent a list of attributes that all have the same permissions. The newer version is 18 lines shorter, as well as more explicit in the relationship that any of the attributes listed grant rlwd permissions. Since the most common use case is to grant users either rl (effectively read permissions) or rlwd (full control), then this means only two lists of users will be required and the permission string is only specified twice. Additionally, attribute-value pairs are closer together in the newer version with no other brackets separating them, making reading through the list easier for humans.