B2Safe module administrator guide

Table of Contents

Updated to the version 2.1 of the module.

Install

See <module_home>/install.txt .

Configuration

Authorization

- iRODS 3.3.x is required.
- in case you are using the version 3.3.0 please apply the patch in "rsExecCmd.patch" placed the dir "patches".
- in the rule file "*eudat.re*": there are two rules called "*EUDATAuthZ*" and "*getAuthZParameters*". The "*EUDATAuthZ*" calls an external python script placed in `<irods_home>/server/bin/cmd` and called "*authZ.manager.py*". Which requires a configuration file placed in `<irods_home>/modules/B2SAFE/cmd` and called "*authz.map.json*". The script provides just a couple of methods: "test" and "check", which returns a boolean value of True if the authorization is granted, False otherwise. The authorization decision is based on the file "*authz.map.json*", which contains triplets (subject, action, target) called assertions. So, for example, passing to the script in input a request like:

```
testuser#testzone,
read,
/<irods_home>/modules/B2SAFE/cmd/credentials
```

It will be accepted if the json file contains:

```
"assertion 1":
  { "subject":
     [ "testuser#testzone" ],
    "action":
     [ "read" ],
    "target":
```

```
        [ "/<irods_home>/modules/B2SAFE/cmd/credentials" ]
      }
```

Or even:

```
    "assertion 1":
       { "subject":
          [ "*#testzone" ],
         "action":
          [ "read" ],
         "target":
          [ "/<irods_home>/modules/B2SAFE/cmd/*" ]
       }
```

Because it supports the wild characters in the same way a shell do.

- in the rulebase file "*core.re*" the hook should be configured using the patch "*corere.patch*" placed in the folder "patches" of the module.
- The entry point for rules specific for certain external executables should be called inside the "*EUDATAuthZ*" as fall back.

Logging

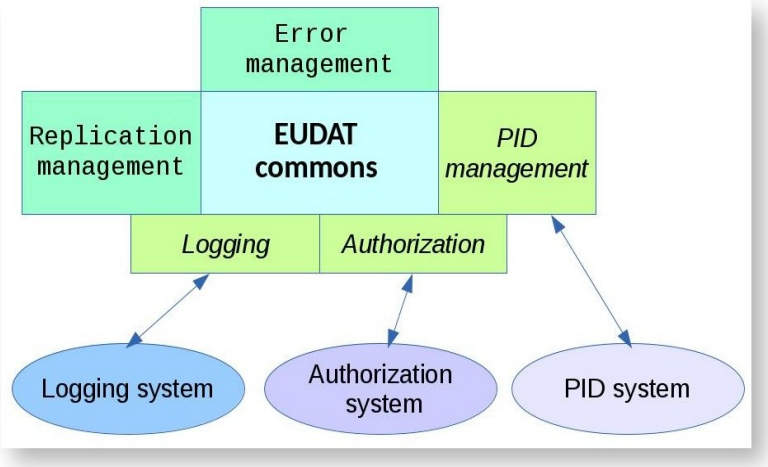
We are considering here `/<irods_home>/modules/B2SAFE/cmd/log.manager.conf`.
Just configure the logging level (INFO, DEBUG, ERROR) and the path to the logging directory:

```
{
"log_level": "DEBUG",
"log_dir": "/<irods_home>/modules/B2SAFE/log",
}
```

Changelog

See <module_home>/docs/changelog.txt

Architecture

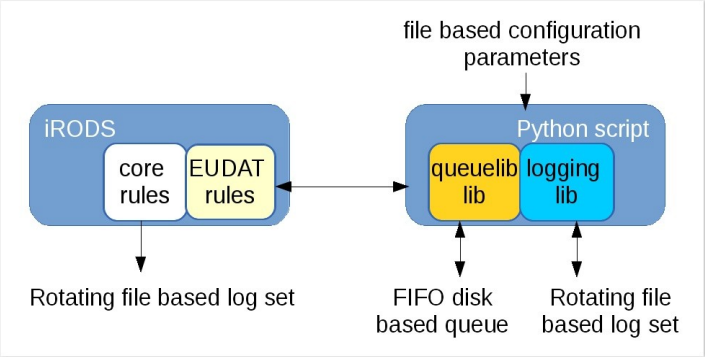


The architecture of the module is organized into small sub-modules. It includes the following major components:

1. EUDAT commons

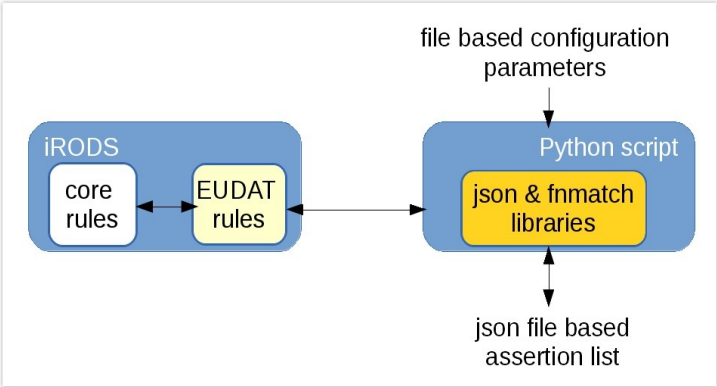
It is a rule set containing the basic functions for processing replication and control-files of each single data object.

2. Logging



It is built on two different data structures: “queue” and “log”, and based on the python-script `/<irods_home>/modules/B2SAFE/cmd/log.manager.py` to do different operations (push, pop, queuesize, log) on the aforementioned data structures. This logging mechanism is used for sorting out only the important information, supporting the monitoring of the data transfers and of the pid management operations. It works as a replication tracking system, which allows to manage the failed transfers in an asynchronous way.

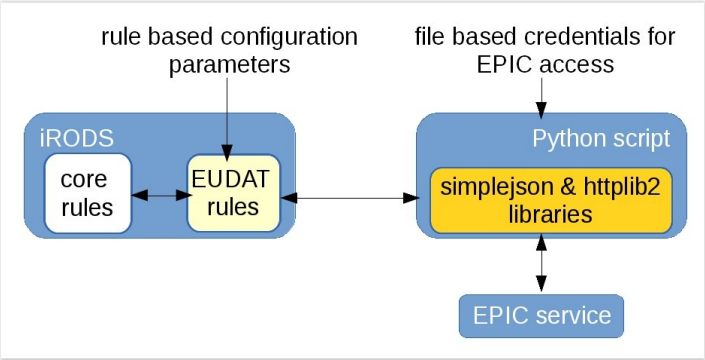
3. Authorization



The authorization decision is based on the file "*authz.map.json*", which contains triplets (subject, action, target) called assertions: they represent who are allowed to do what.

4. PID management:

This rule set contains the rules to perform the main operations related to the PID service. It is based on the script `<irods_home>/modules/B2SAFE/cmd/epicclient.py`; which relies on the configuration file `<irods_home>/modules/B2SAFE/cmd/credentials`.



5. Replication Management

This rule set includes functions used to transfer a single data object or a whole collection, together with integrity check mechanisms.

6. Error Management

The last rule set includes functions able to process errors during data transfer. It can detect errors about checksum and size inconsistencies, PID processing and data object ownernership.

API (EUDAT rules)

Commons

EUDATiCHECKSUMretrieve(*path, *checksum)

Get an existent checksum from iCAT

EUDATiCHECKSUMget(*path, *checksum)

Get, if exist or create if not, a checksum from iCAT

EUDATgetObjectTimeDiff(*filePath, *age)

Calculate the difference between the creation time and the modification time of an object (in seconds).

EUDATfileInPath(*path,*subColl)

Check if a file is in a given path

EUDATCreateAVU(*Key,*Value,*Path)

Create a metadata triplet on iCAT

EUDATiDSSfileWrite(*DSSfile)

Create an object containing a list of PIDs related to a specific collection

Logging

EUDATLog(*message, *level)

Log an event

EUDATQueue(*action, *message, *number)

Log a failure to a FIFO queue

Authorization

EUDATAuthZ(*user, *action, *target, *response)

Authorization policy decision point

PID management

EUDATCreatePID(*parent_pid, *path, *ror, *iCATCache, *newPID)

Create PID

EUDATSearchPID(*path, *existing_pid)

Search PID

Replication management

EUDATUpdateLogging(*status_transfer_success, *path_of_transfered_file, *target_transfered_file, *cause)

Log a transfer event to the log file and, if it is a failure, to the FIFO queue

EUDATCheckError(*path_of_transfered_file,*target_of_transfered_file)

Perform error checks about the transfer

Error management

EUDATCatchErrorChecksum(*source,*destination)

Catch error with Checksum

EUDATCatchErrorSize(*source,*destination)

Catch error Size of file

EUDATSearchPIDchecksum(*path, *existing_pid)

Search PID by checksum

EUDATUpdatePIDWithNewChild(*parentPID, *childPID)

Update PID record field 10320/LOC

EUDATGetRorPid(*pid, *ror)

Get PID record field RoR's value

EUDATeiPIDeiChecksumMgmt(*path, *PID, *ePIDcheck, *iCATuse, *minTime)

Create or update a PID, including checksum

EUDATiPIDcreate(*path, *PID)

Create a PID as iCAT metadata

EUDATiFieldVALUEretrieve(*path, *FNAME, *FVALUE)

Get a metadata value from iCAT

EUDATePIDcreate(*path, *PID)

Create a PID as EPIC service record

EUDATePIDsearch(*field, *value, *PID)

Search a PID on the EPIC service

EUDATeCHECKSUMupdate(*PID)

Update the PID record field checksum

EUDATeURLupdate(*PID, *newURL)

Update the PID record field URL

EUDATePIDremove(*path)

Delete a PID

EUDATeiPIDeiChecksumMgmtColl(*sourceColl)

Walk through the collection. For each object, it creates a PID and stores its value and the object checksum in the iCAT.

EUDATiRORupdate(*source, *pid)

Add the ROR field of the PID of the object to iCAT

EUDATeParentUpdate(*PID, *PFName, *PFValue)

Update the EUDAT ROR or PPID field in the PID record

EUDATTransferSingleFile(*path_of_transfered_file,*target_of_transfered_file)

Transfer a single file

EUDATTransferUsingFailLog(*buffer_length)

Retry to perform a certain number of failed transfers queued in the FIFO queue

EUDATCheckReplicas(*source, *destination)

Check whether two files are available and identical and trigger replication if they are not

EUDATTransferCollection(*path_of_transfered_coll,*target_of_transfered_coll,*incremental,*recursive)

Transfer a whole collection

EUDATProcessErrorUpdatePID(*updfile)

Process error update PID at Parent_PID. It will be processed during replication_workflow, called by updateMonitor.

EUDATCatchErrorDataOwner(*path,*status)

Catch error Data Owner if user is not owner of Data from *path

Best Practices

Authorization

If you want to implement an ACL for the execution of an external command, such as a python script, a

C code executable or a shell command, you can use the iRODS hook:

```
acPreProcForExecCmd(*cmd, *args, *addr, *hint) {
    if (*cmd != "authZ.manager.py") {
        EUDATAuthZ("$userNameClient#$rodsZoneClient",
                   *cmd, *args, *response);
    }
}
```

This hook can be put in the ruleset `<irods_home>/server/config/reConfig/core.re` .

Then in the file `<irods_home>/modules/B2SAFE/cmd/authz.map.json` can be added the suitable assertions. So for example if the objective is to implement:

`Only user guybrush#MIslandZone can execute the python script <irods_home>/server/bin/cmd/drink_grog.py`

Then just add the following assertion in the authorization map:

```
{ "subject": [ "guybrush#MIslandZone" ],
  "action":  [ "<irods_home>/server/bin/cmd/drink_grog.py" ],
  "target":  [ "*" ]
}
```

But if you want a more fine-grained ACL, you can also specify the allowed input arguments:

`Only user guybrush#MIslandZone can execute the python script <irods_home>/server/bin/cmd/drink_grog.py -in acid_battery`

```
{ "subject": [ "guybrush#MIslandZone" ],
  "action":  [ "<irods_home>/server/bin/cmd/drink_grog.py" ],
  "target":  [ "-in acid_battery" ]
}
```

In principle, the same mechanism can be applied directly to filter the execution of every rule. For example, adding a line before the rule invocation in this way:

```
acPostProcForPut {
    EUDATAuthZ("$userNameClient#$rodsZoneClient",
               "EUDATTransferSingleFile", “*”, *response);
    EUDATTransferSingleFile(*path,*replicaPath);
}
```

And the related assertion in the map:

```
{ "subject": [ "user#CompanyZone" ],
  "action":  [ "EUDATTransferSingleFile" ],
  "target":  [ "*" ]
}
```

However the authorization mechanism implies a certain overhead so it should be used carefully.

Replication

In order to replicate a single file the best option is the rule:

```
EUDATTransferSingleFile(*path_of_transfered_file,
```

```
                            *target_of_transfered_file)
```
contained in the ruleset `<B2SAFE_module>/rulebase/replication.re` .
This rule already includes integrity checks (based on object's size and checksum), proper registration and update of PIDs associated to the source and to the target, and a logging mechanism to track replication errors.

In case the replication source is a collection, then it is more suitable the following rule:
```
EUDATTransferCollection(*path_of_transfered_coll,
                            *target_of_transfered_coll,
                            *incremental,*recursive)
```
Which is included in the aforementioned ruleset.
It relies on `EUDATTransferSingleFile`, hence it has its same features.
However this rule is quite demanding in terms of memory consumption, therefore in case of collection containing more than 1000 objects, it is advised to execute it in delayed mode (see `<B2SAFE_module>/rules/testtransfercollection.r`), for example:
```
delay("<EF>1s REPEAT UNTIL SUCCESS OR 10 TIMES</EF>") {
        EUDATTransferCollection(*Path,*replicaPath,
                                bool("true"),bool("true"));
}
```
Note the two boolean flags set to true which mean respectively “perform it in a incremental way” and “perform it in a recursive way”. The different options available for the delay mode are listed here: https://irods.sdsc.edu/index.php/Rule_Execution_modes .
In this way it will be possible to recover the replication, even after a failure due to out of memory errors.

Performance

In order to improve the performance in case of multiple processes requesting iRODS resources is advisable to change the following line in the rule set *core.re*:

```
# 17) acSetReServerNumProc - This rule set the policy for the number of processes
# to use when running jobs in the irodsReServer. The irodsReServer can now
# muli-task such that one or two long running jobs cannot block the execution
# of other jobs. One function can be called:
#    msiSetReServerNumProc(numProc) - numProc can be "default" or a number
#    in the range 1-4. numProc will be set to 1 if "default" is the input.
#
#acSetReServerNumProc {msiSetReServerNumProc("default"); }
```

With the next one:

```
acSetReServerNumProc {msiSetReServerNumProc("4"); }
```