Programmable authentication workflows in iRODS

Stefan Wolfsheimer
SURF
Utrecht, The Netherlands
stefan.wolfsheimer@surf.nl

Claudio Cacciari SURF
Utrecht, The Netherlands
claudio.cacciari@surf.nl

Harry Kodden SURF
Utrecht, The Netherlands
harry.kodden@surf.nl

ABSTRACT

iRODS (Integrated Rule-Oriented Data System) [1] supports various authentication methods such as native authentication (username and password), GSI, Kerberos, and OpenID. New authentication methods are implemented as shared libraries that need to be installed on client and server sides. Client libraries such as python-irodsclient may need to be patched to support any new authentication protocol.

A universal implementation that supports all authentication flows is clearly favored over managing combinations of client and server libraries and flows. The PAM (Pluggable Authentication Module) [2] mechanism is a way to implement and customize authentication flows on the server without needing to adjust the software that uses this mechanism. Existing PAM libraries may be combined to implement flows featuring branches, multiple-factor authentication, and much more. The PAM mechanism is already supported by iRODS but the current version of the plugin is restricted to the standard flow only (username and password). We have implemented an authentication plugin for iRODS 4.3.0 "`pam_interactive`" that enables the flexibility of fully-fledged PAM authentication flows beyond the standard case.

SURF, the Dutch cooperative association of educational and research institutions, will use that implementation to offer new features to iRODS users. Two scenarios are especially relevant: the support of the SURF Access Management Provider (SRAM), which allows multiple Identity Providers to authenticate a user with iRODS, and the support of Multi-Factor Authentication (MFA) directly at iRODS level, which is often required for sensitive data management.

Keywords

PAM stack, authentication, plugin, OIDC.

INTRODUCTION

Linux-PAM [2] is a mechanism that aims at standardizing user authentication workflows. The mechanism is flexible such that it is possible to support a number of different authentication methods and combinations of them. PAM supports four management groups: account management, authentication, password management, and session management. The scope of this paper and the implemented iRODS plugin is the authentication flow only. The present paper is a follow-up of the work described in a paper presented at UGM 2019 [6], where a similar approach was adopted. In the previous implementation, the complexity of the flow was encapsulated by an additional web component required in front of the iRODS Catalog Provider which increased the overhead and limits the flexibility.

System administrators can mix and match from a variety of *PAM-modules* to implement authentication flows of arbitrary complexity [3]. PAM-modules are layered on a stack which is processed from top to bottom. Finally, a status is returned indicating the success or failure of the authentication flow. Each module itself returns a status code. A control value, which is assigned to each layer, indicates a criterion of how status codes are to be handled. For

iRODS UGM 2022 July 5-8, 2022, Leuven, Belgium

example, if a module with control value `sufficient` returns status code `success`, the stack terminates with success (state "authenticated"). While a failed module assigned `required` causes the whole flow to fail.

PAM-Modules are shared libraries that bridge the communication between directory services, user databases, flat files, etc. with the PAM framework [4].

In order to enable applications with PAM, application developers need to implement the user-facing parts of the authentication flow (e.g., retrieving login information from the user) and delegate the flow control to the PAM library [5]. This approach is described below in the PAM flows sections. In the section Usage, we discuss a few configuration patterns and examples using the flexible `pam_python` module.

IMPLEMENTATION

PAM flows

The authentication procedure of a PAM-enabled application is controlled by the PAM library. The process can be seen as a state machine defined by the PAM configuration. The user interaction is realized by callback functions that are passed from the application to the PAM library. The function is called on each transition that requires user interaction (e.g. querying users' credentials, printing a message on the screen, etc.). The PAM architecture with each component is shown in Figure 1.

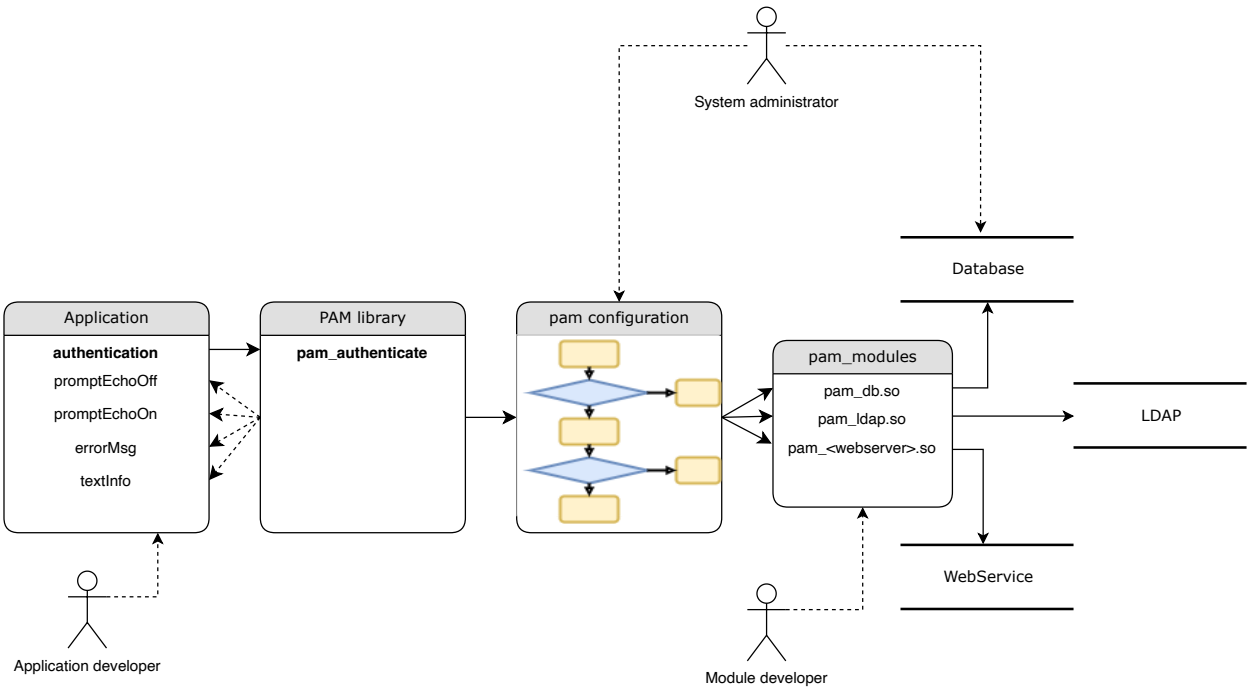

Figure 1. PAM configuration: PAM enabled application, PAM library, PAM stack configuration, PAM modules, and services

PAM flows over the network

Enabling applications with PAM is straightforward when all components, the PAM library, the PAM configuration, and the application are installed on the same host. The situation is more complicated for client-server systems such as iRODS. In this case, the PAM library and the configuration are installed on the server, while the user interaction is realized on the client-side. This implies that the callback functions invoked by the PAM library need to wait for user

input on another host during their lifetimes. On the other hand, the iRODS API is implemented as a request-response model, where the client drives the communication between the components by requesting resources from the server. A callback to the client from the server is not directly supported in such protocols.

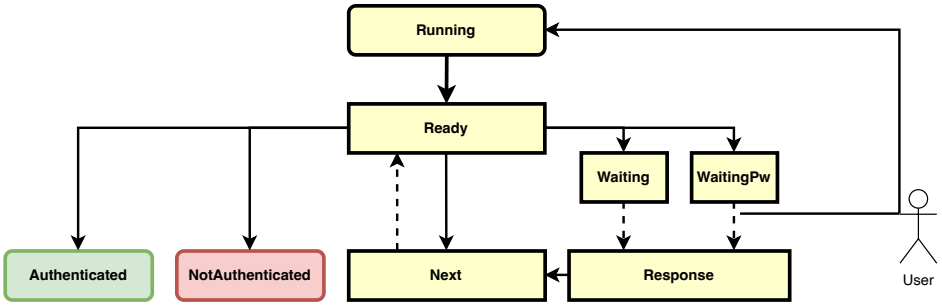

Figure 2. State diagram of the PAM flow

To overcome this limitation we have designed the PAM workflow as a state machine (see Figure 2). The conversation is triggered by a user wanting to login to iRODS (e.g. using `iinit`). The initial state is **Running** which is immendiatly turned to **Ready**. A transition from **Ready** to one of the states **Waiting**, **WaitingPw**, **Next**, **Authenticated** or **NotAuthenticated** is driven by the PAM configuration. As suggested by the names, the states **Authenticated** and **NotAuthenticated** refer to the final states of successful and unsuccessful authentication, respectively. The transition from **Ready** to next **Next** is silent or accompanied by a message that is printed on the screen of the client. After the transition from **Ready** to **Waiting** or **WaitingPw** a message is printed and the user is expected to type a response. The transition from **Waiting** and **WaitingPw** to **Response** is triggered by the client after the user has provided the response. The transition from **Next** to **Ready** is triggered by the client to indicate the readiness for another iteration. The transitions triggered by the client are indicated as dashed lines. Since those transitions depend on user input, the callback functions cannot simply return a value back to the PAM library. Instead, instances of the functions remain idle waiting for a response from the client. Technically, this behavior is implemented by a condition variable that is active during the lifetime of the callback function.

The sequence diagram in Figure 3 illustrates a simple PAM conversation over the network. There are four components involved:

- The iRODS client (icommands)
- The iRODS server
- The `pam_interactive` plugin
- The PAM library.

The blue boxes indicate the lifetimes of the callbacks. Notice that the transition from **Waiting** to **Response** will wake up a condition variable.

USAGE

State persistency

After the user has successfully authenticated using the PAM stack, a temporary password is generated which is valid for one hour by default. This password is used by `icommands` to authenticate against the server. After the expiration period of the password, the PAM authentication is again executed whenever a user invokes a `icommands`. The responses of the last conversation are locally cached and repeated. Below, we describe alternative flows.

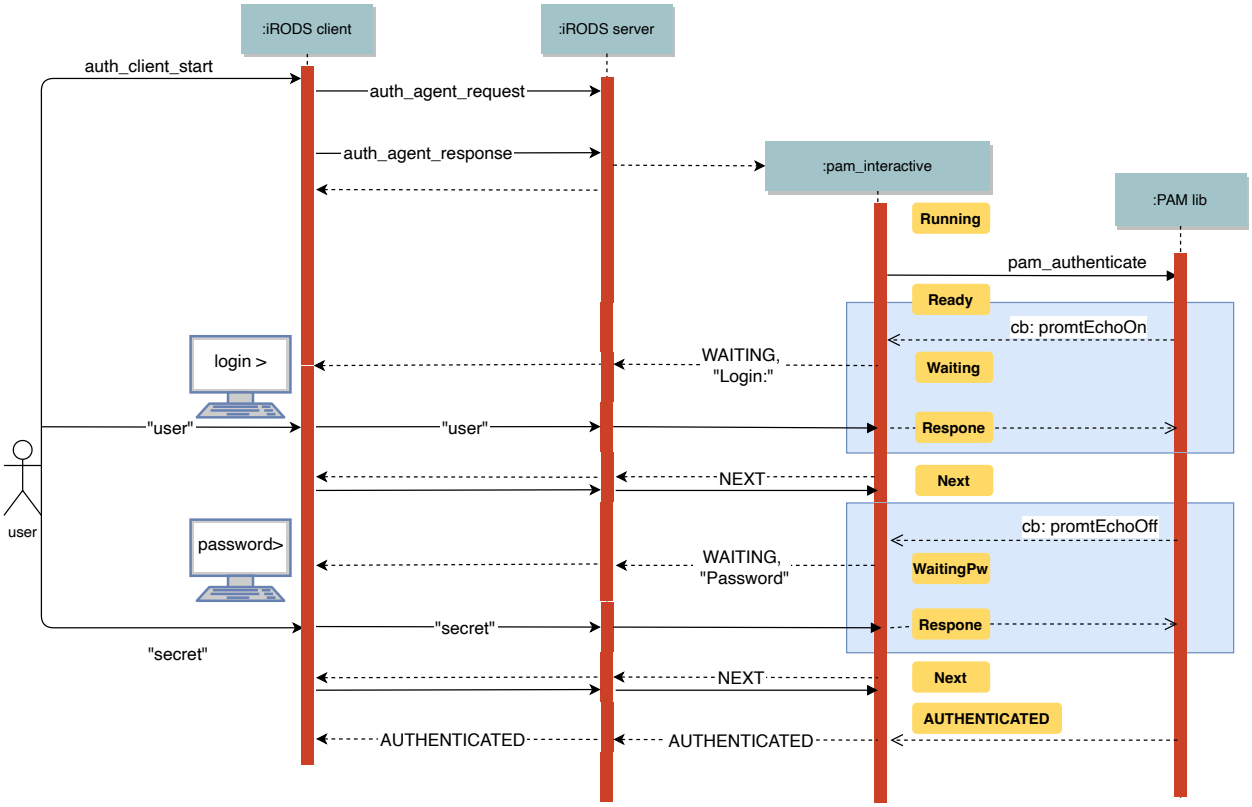

Figure 3. State diagram of the PAM flow

The expiration time of the password can be overridden by the user via the time-to-live option (`-ttl`, in hours) of `iinit`. The iRODS server administrator can set the range of valid values for the TTL value in the server configuration file (`/etc/irods/server_config.json`):

```
"plugin_configuration": {
  "authentication": {
    "pam_interactive" : {
      "password_min_time": 3600,
      "password_max_time": 7200
    }
  }
}
```

Notice that in the current implementation of iRODS, the smallest granularity of the TTL is in hours. The values in the server configuration are given in seconds to support future versions with smaller granularity.

Prototyping with `pam_python`

In this section, we discuss the ability of the `pam_interactive` plugin from the perspective of a PAM module developer and a PAM system administrator. In order to keep the discussion illustrative and generic, we use the `pam_python` [7] module. In contrast to many other modules, this module does not rely on specific backends or user databases. `pam_python` is convenient for

- implementing prototypes of PAM modules for novel backends
- illustrating PAM flows and
- implementing regression tests

However, the pace of the development is relatively slow. Thus, there is no guarantee that the software will be supported in the future.

Consider, for example, the following PAM stack (`/etc/pam.d/irods`), which uses the `pam_python.so` module with the *required* control variable:

```
auth required  pam_python.so /etc/pam.d/simple.py
```

The user will be successfully authenticated when the `pam_sm_authenticate` function defined in the python module returns `PAM_SUCCESS`. The following implementation mimics the authentication against a simple user database.

```
USERS_DB={
    'ayub': 'pw',
    'mara': 'ACtoRPHI',
    'noah': 'NgPOWArs'
}
def pam_sm_authenticate(pamh, flags, argv):
    msg = pamh.conversation(pamh.Message(pamh.PAM_PROMPT_ECHO_ON, "login:"))
    pwd_msg = pamh.conversation(pamh.Message(pamh.PAM_PROMPT_ECHO_OFF, "password:"))
    login = msg.resp
    password = pwd_msg.resp
    if login in USERS_DB and password == USERS_DB[login]:
        return pamh.PAM_SUCCESS
    return pamh.PAM_AUTH_ERR
```

In a real-life application, the dictionary would be replaced by a user directory, such as LDAP or another database.

Next, we illustrate how one would enable a second factor required to successfully log in. This can be realized by adding a second required layer on the pam stack

```
auth required  pam_python.so /etc/pam.d/simple.py
auth required  pam_python.so /etc/pam.d/2fa.py
```

Now, both layers are required. The user needs to enter their regular credentials before being asked for a one-time PIN generated by a key generator. The module of the second layer can be implemented as follows:

```
def pam_sm_authenticate(pamh, flags, argv):
    msg = pamh.conversation(pamh.Message(pamh.PAM_PROMPT_ECHO_ON, "pin:"))
    pin = msg.resp
    if pin == "1234":
        return pamh.PAM_SUCCESS
    else:
        return pamh.PAM_AUTH_ERR
```

According to the implementation, the correct PIN is the fixed value 1234. In real-life applications, this stub should obviously be replaced by a real validation. Notice that the logic is entirely driven by the backend and controlled by the systems administrator. In contrast to other iRODS authentication methods, new policies (e.g. enabling a second factor) can be rolled out without changing the local client configurations and actively supporting users.

Persistent client information

The user responses to the conversation are stored locally in a JSON document next to the scrambled password. They can be reused as default values when the user logs in again. However, in some cases, this behavior is not desirable. For example, it does not make sense to store and reuse the values of one-time passwords for second-factor authentications. On the other hand, some workflows may require storing and retrieving data without user interaction. In order to address these use cases, we have extended the standard protocol. The server can either send simple messages (as in the example above), or JSON payloads describing a set of operations. The message has the form of a JSON object with the following (optional) keys:

- `prompt`: a message to be printed on the screen,
- `default_path`: the JSON path to the default value,
- `patch`: a list of patches to be applied to the locally stored JSON document (The patches are implemented according to the specification RFC6902 [8, 9]. and
- `retrieve`: a JSON path to the locally stored JSON node to be sent back to the server

Example 1: `prompt` *and* `patch`

The following example, a modification of the `2fa.py` script from the previous section, illustrates the use of the `prompt` and `patch` fields. The PIN is returned but not stored locally because of the absence of the `patch` field. After successful authentication, a token is sent to the client and stored locally:

```
import json
import uuid

def pam_sm_authenticate(pamh, flags, argv):
    # just prompt, don't save the pin locally
    pin = {"prompt": "pin:"}
    msg = pamh.conversation(pamh.Message(pamh.PAM_PROMPT_ECHO_ON, json.dumps(pin)))
    pin = msg.resp

    if pin == "1234":
        token = str(uuid.uuid4())
        # save token on client, no prompt
        patch = {"patch": [{"op": "add",
                            "path": "/token",
                            "value": token}]}
        msg = pamh.conversation(pamh.Message(pamh.PAM_TEXT_INFO,
                                             json.dumps(patch)))

        return pamh.PAM_SUCCESS
    else:
        return pamh.PAM_AUTH_ERR
```

Example 2: `default_path`

Notice that it is also possible to query a message from the user and save it locally under a given path. For example, the following message queries a pin and stores it under the path `/pin` in the local JSON document. The next time the PIN is queried, the default value is take from the JSON document under the path `/pin`.

```
# prompt and save the pin locally
pin = {"prompt": "enter pin:",
       "default_path": "/pin",
       "patch": [{"op": "add",
                  "path": "/pin"}]}
```

Example 3: `retrieve`

Suppose we have stored the token under the path `/token`. Then the data can be retrieved with the payload `{"retrieve": "/token"}`. The following `pam_python` module requests the locally stored token from the client and returns the result

```
import json

def pam_sm_authenticate(pamh, flags, argv):
    payload = {"retrieve": "/token"}
    msg = pamh.conversation(pamh.Message(pamh.PAM_PROMPT_ECHO_ON, json.dumps(payload)))
    token = msg.resp
    payload_resp = {"prompt": "token={}".format(token)}
    pamh.conversation(pamh.Message(pamh.PAM_TEXT_INFO, json.dumps(payload_resp)))
    return pamh.PAM_SUCCESS
```

This example concludes the fundamental operations that can be applied to the local JSON document. PAM module developers and/or administrators can make use of them to implement complex flows.

In the following section, we turn to an application that has motivated the need of a flexible authentication flow.

OpenID Connect

Our need to support multi-factor authentication and federated identity management led us to choose the OpenID Connect protocol, which adds, on top of the OAuth2 authorization protocol, an authentication token that includes some basic user profile information. The main use case for the OIDC protocol is the authentication of a web application against an Identity Provider (IdP). But our users want to log in iRODS via the command line. In order to do that we have adapted one of the OIDC `flow`, the Authorization Code Flow (defined in OAuth 2.0 RFC 6749, section 4.1, [10]), as shown in Figure 4.

Clearly, it would not have been possible to implement that flow without the new authentication plugin. In fact, the user is presented with a challenge (the log in URL) and the server waits for a response. Beyond the Authorization Code Flow, we have added steps 13-15 to map the identity of the user to an iRODS account, using one of the available attributes, like, for example, the email address. In terms of the PAM python module, the function `pam_sm_authenticate` could be written in the way given in the APPENDIX.

When the token is not valid, the authentication simply fails, but it would be possible to use a refresh token to automatically renew the expired one.

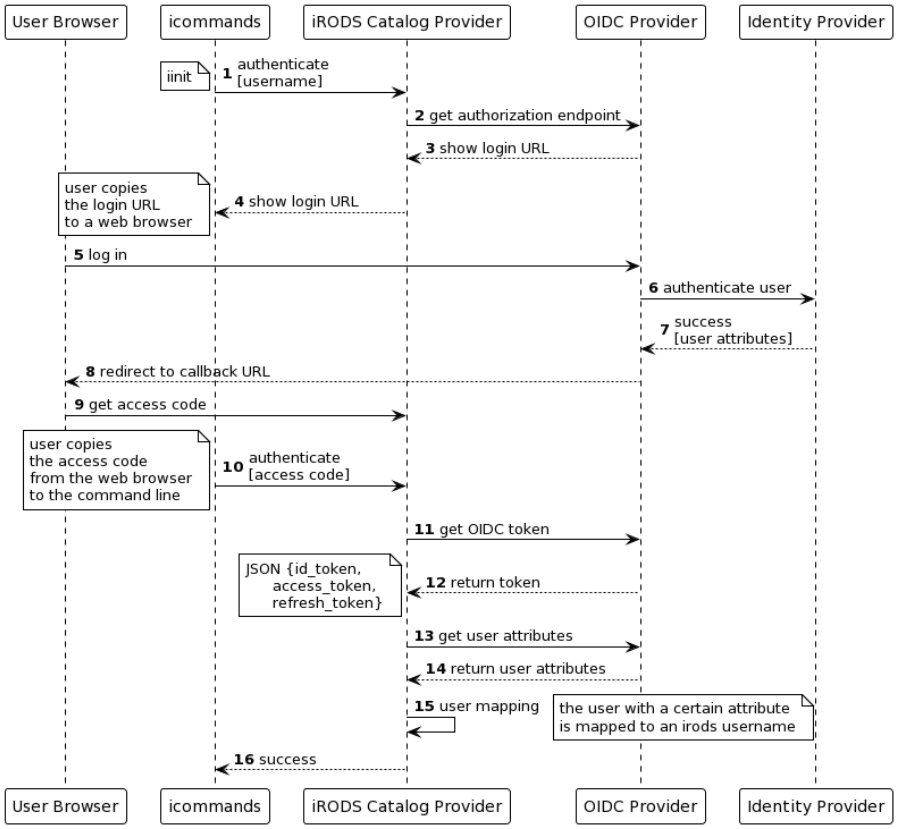

Figure 4. OIDC Authorization Code Flow: adapted for the interaction via command line

CONCLUSION

`pam_interactive` provides a backend-driven programmable and flexible authentication mechanism for iRODS. By simplified examples, we have illustrated solution patterns for programing multifactor authentication flows and token management.

The plugin supports a large range of authentication methods and customized flows because the conversation is not restricted to simple login-password credentials. The capability of storing information locally can be extended in future releases. One can think of JSON Web Token as a common technique to implement single signed-on in web-based applications. Adopting this technology to iRODS would improve the interoperability with other systems.

ACKNOWLEDGMENTS

We thank the members of the iRODS Authentication Working Group for the organization of regular meetings and open discussions. We also thank our colleague Maithili Kalamkar-Stam for critical proofreading on very short notice.

APPENDIX

OIDC authentication example code

```
def pam_sm_authenticate(pamh, flags, argv):
    try:
        user = pamh.get_user(None)
    except pamh.exception, e:
```

```
    user = None
    pamh.conversation(pamh.Message(pamh.PAM_ERROR_MSG, str(e.pam_result)))
if user == None:
    return pamh.PAM_USER_UNKNOWN

no_token = True
# Check if the user has a token
payload = json.dumps({ "retrieve": "/oauth2_access_token"})

token_msg = pamh.conversation(pamh.Message(pamh.PAM_PROMPT_ECHO_ON, payload))
if token_msg is not None and len(token_msg.resp.strip()) > 0:

    # Validate the token
    result = validate_token(user, token_msg.resp, INTROSPECT_URL, OIDC_USER_MAP)
    pamh.conversation(pamh.Message(pamh.PAM_TEXT_INFO, "Authentication: {}".format(result)))
    if (result.strip() == "Success"):
        no_token = False
        return pamh.PAM_SUCCESS
    else:
        pamh.conversation(pamh.Message(pamh.PAM_TEXT_INFO, "Invalid token: {}".format(result)))

if no_token:

    state = uuid.uuid4()
    # Get the login URL
    params = {"response_type": "code",
              "client_id": CLIENT_ID,
              "redirect_uri": REDIRECT_URI,
              "scope": "openid offline_access email eduperson_principal_name"
              "state": state}
    loginURL = AUTHORIZATION_EP + '?' + urlencode(params)

    # Copy it to the browser
    payload = {"prompt": "Copy the following URL to your web browser:"}
    pamh.conversation(pamh.Message(pamh.PAM_TEXT_INFO, json.dumps(payload)))

    payload = {"prompt": loginURL}
    pamh.conversation(pamh.Message(pamh.PAM_TEXT_INFO, json.dumps(payload)))

    # Copy back the callback string
    # request without saving answer locally
    payload = {"prompt": "Copy the callback string from your web browser here:"}
    callback_msg = pamh.conversation(pamh.Message(pamh.PAM_PROMPT_ECHO_ON, json.dumps(payload)))

    # Get a token
    token = oidc_get_token(callback_msg.resp, REDIRECT_URI, TOKEN_EP, BASE64CREDS)

    # Validate the token
    result = validate_token(user, token, INTROSPECT_URL, OIDC_USER_MAP)
    pamh.conversation(pamh.Message(pamh.PAM_TEXT_INFO, "Authentication: {}".format(result)))
```

```
    if (result.strip() == "Success"):
        # save a simple cookie on the client
        # display an optional message
        payload = {"prompt": "the cookie 'oauth2_access_token' has been updated",
                   "patch": [{"op": "add",
                              "path": "/oauth2_access_token",
                              "value": token}]}
        pamh.conversation(pamh.Message(pamh.PAM_TEXT_INFO, json.dumps(payload)))
        return pamh.PAM_SUCCESS
    else:
        return pamh.PAM_AUTH_ERR
```

REFERENCES

[1] Integrated Rule-Oriented Data System (iRODS) https://irods.org/

[2] Linux-PAM. http://www.linux-pam.org/ Visited last on 06.24.2022.

[3] Morgan, A.G., Kukuk, T.: The Linux-PAM System Administrators' Guide, Version 1.1.2, 31. (2010) http://www.linux-pam.org/Linux-PAM-html/Linux-PAM_SAG.html

[4] Morgan, A.G., Kukuk, T.: The Linux-PAM Module Writers' Guide http://www.linux-pam.org/Linux-PAM-html/Linux-PAM_MWG.html Version 1.1.2, 31. (2010)

[5] Morgan, A.G., Kukuk, T.: The Linux-PAM Application Developers' Guide http://www.linux-pam.org/Linux-PAM-html/Linux-PAM_ADG.html Version 1.1.2, 31. (2010)

[6] Cacciari, C., Muscianisi G., Carpené, M., D'Antonio, M. and Fiameni G. An authentication solution for iRODS based on the OpenID Connect protocol, iRODS User Group Meeting Proceedings (2019)

[7] Stuart, R.: pam_python: Write PAM modules in Python http://pam-python.sourceforge.net/ Version 1.0.8-1. (2020)

[8] Lohmann, N. et. al.: JSON for Modern C++ https://json.nlohmann.me/ Version 3.7.3 (2022)

[9] JavaScript Object Notation (JSON) Patch, RFC 6902 https://www.rfc-editor.org/info/rfc6902 (2013)

[10] The OAuth 2.0 Authorization Framework, RFC 6749 (2012) https://www.rfc-editor.org/info/rfc6749