Solutions for non-web OAuth 2.0 authorisation at CERN

Asier Aguado Corman[1,*], *Jack* Henschel[1,], *Hannah* Short[1,**], and *Sebastian* Lopienski[1,***]

[1]CERN

Abstract. The need for Single Sign-On solutions in command line interfaces is not new to CERN. Different technologies have been introduced and internal solutions have been implemented to allow users to authenticate to remote servers or applications from their console interfaces. In the case of web services, the most common approach was to use cookie-based authentication, for which an internal tool was developed and made available for all the CERN user community. As the authorisation infrastructure evolved and started to fully support the OAuth 2.0 standard, as well as two-factor authentication (2FA), using the internal tool started to show its limitations. In this work, we present the past and present (OAuth-compliant) solutions, and compare them by looking at the advantages and disadvantages we have found. We also present a case study of a service, OpenShift, that implements this new authentication solution for their users.

1 Introduction

CERN is migrating its Authentication and Authorisation Infrastructure to Keycloak [1], which allows us to fully support OAuth 2.0 (Open Authorisation 2.0, hereafter OAuth for short, as previous versions are obsolete [2] and OAuth 2.1 is in draft stage) as well as SAML (Security Assertion Markup Language). OAuth has become increasingly popular over the past 5 years, and the vast majority of our authenticated applications now rely on OAuth. Large web service providers have started to support — or even recommend — OAuth as the authorization mechanism for web APIs during the last decade [3]. Many workflows for researchers and engineers at CERN are completed on the command line by interacting with web-based APIs from this environment. With the current popularity and maturity of OAuth, finding user friendly, secure mechanisms to provision OAuth tokens on a CLI is essential to support users and avoid insecure workarounds.

2 Previous solution: cookie-based authentication

This process was developed for the previous, Active Directory Federation Services based system and works in the same way with Keycloak. Keycloak supports Kerberos authentication through SPNEGO [4]: this allows us to use Kerberos to keep the same session as in the terminal and avoid any prompt for credentials. Here is an example of how this kind of

[*]e-mail: asier@aguado.email

[**]e-mail: hannah.short@cern.ch

[***]e-mail: sebastian.lopienski@cern.ch

authentication would work in practice for a user: first they obtain a Kerberos TGT (Ticket Granting Ticket) in their terminal session:

```
kinit
```

Then they are able to run our internal tool to export the SSO cookies and other session cookies from a website into a text file:

```
auth-get-sso-cookie -u https://the-target-api.cern.ch -o cookies.txt
```

Finally they can use this file to call the target website using a tool like curl:

```
curl -L -b cookies.txt https://the-target-api.cern.ch/foobar
```

This login flow is often seen as quick and transparent for the user, as they only need a Kerberos TGT. This is a very common authentication mechanism in command line environments and no external web browser is needed. The negative side of this tool is that it relies on web parsing, as the Kerberos implementation is not part of the authorisation protocol but part of the Keycloak login forms. This makes it dependent on the SSO website structure, therefore small modifications (which may regularly occur) can break the tool. In addition, it is difficult to reuse the tool outside CERN or replace it with open source tools that use authentication and authorization standards supported by Keycloak. As standards evolve and more authentication options become available, depending on this tool for authentication can limit the options for our users. Finally, this solution does not support two-factor authentication, which is increasingly widely used at CERN.

3 Present solution: OAuth 2.0 Device Authorization Grant

As two-factor authentication was being rolled out to secure accounts from the organisation, the authentication process was becoming more complex and showing the limitations of the cookie-based approach. Our internal tool was unusable for many users who configured WebAuthn devices for two-factor authentication. Therefore we needed a new authentication mechanism that meets the following requirements:

- Can be used with TOTP (Time-Based One-Time Password Algorithm) [5] and WebAuthn [6] second factor authentication protocols.
- Is standards compliant and does not depend on web parsing for the authentication forms.
- Works inside remote terminal sessions (e.g. through SSH).

Given these requirements, we have chosen the OAuth 2.0 Device Authorization Grant [7]. This standard OAuth authentication mechanism can be initiated in any device or remote session, and users can log in using a web browser session from another device, or in the same device. In our case we use a public client, so that no secrets have to be distributed to each user of the service. The authentication flow works as shown in Figure 1: when the user runs a command, a client-side script requests an authentication token from the SSO, using OAuth2 Device Grant and Proof Key for Code Exchange (PKCE). Then the user is presented with the device code returned by the SSO, which they have to introduce in the login web interface or alternatively open a direct URL in any web browser. The script periodically sends polling requests to the SSO, which will result in a successful response that contains the OAuth2 token once the user has introduced the correct code in the SSO. Finally, the user or a client-side application will be able to use this token to access an OAuth2 protected HTTP resource.

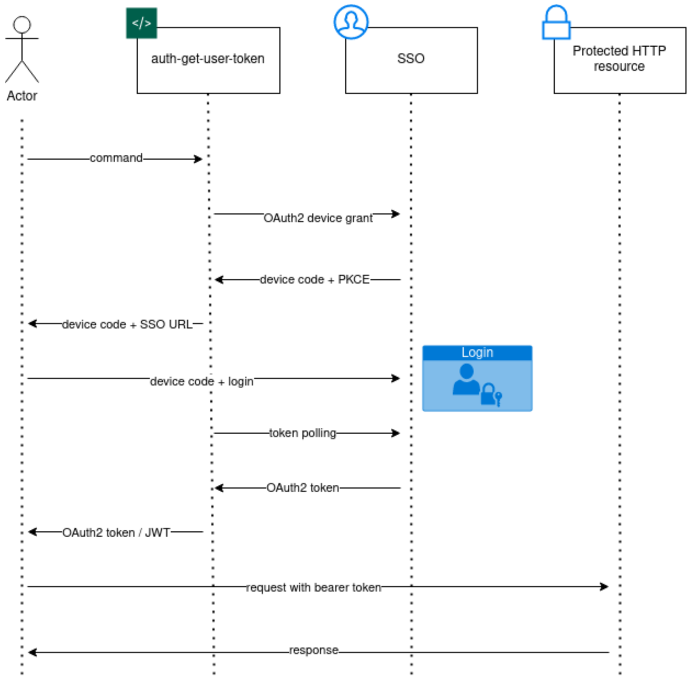


Figure 1. User authentication flow using the OAuth2.0 Device Authorization Grant

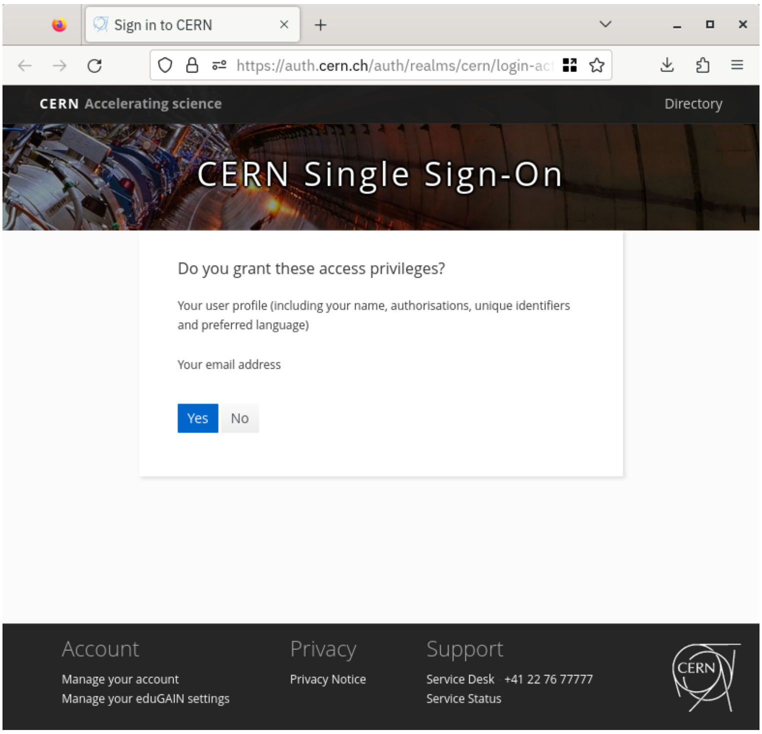
The following snippet shows practical example of a terminal session from the user's point of view: they start by running a command; in our case "auth-get-user-token". Then they are asked to open a URL in their web browser and the script waits for a login confirmation.

```
$ auth-get-user-token -c myapi -x -o token.txt
CERN SINGLE SIGN-ON

On your tablet, phone or computer, go to:
https://auth.cern.ch/auth/realms/cern/device
and enter the following code:
KFRX-JXIV

You may also open the following link directly and follow the instructions:
https://auth.cern.ch/auth/realms/cern/device?user_code=KFRX-JXIV

Waiting for login...
```

Then the user normally logs in with their web browser window, or opens the direct link in a web browser that already has an open session to authorise access for the console session, as shown in Figure 2.

We have found that some positive aspects of this authentication flow are:

- Application development can become more simple by using stateless JWT (JSON Web Tokens) instead of session cookies.
- JWT provides built-in access control through roles, which are configurable by application managers.



Sign in to CERN
https://auth.cern.ch/auth/realms/cern/login-ac
CERN Accelerating science
Directory
CERN Single Sign-On

Do you grant these access privileges?

Your user profile (including your name, authorisations, unique identifiers and preferred language)

Your email address

Yes
No
Account
Manage your account
Manage your eduGAIN settings
Privacy
Privacy Notice
Support
Service Desk +41 22 76 77777
Service Status
CERN

Figure 2. Web browser window after opening the authorisation URL from console output.

- No dependencies on Kerberos; users can authenticate from environments that don't have Kerberos installed.
- The URL to open can easily be converted to a QR code that can be scanned by a secondary device if the CLI environment does not have access to a browser.

On the negative side, we are aware that the previous authentication flow is more convenient for some users, especially because the OAuth workflow asks the user to open a browser window outside the terminal. This is an extra step in some cases: Kerberos is often used as a Single Sign-On protocol in some console environments, where enforcing web authentication results in double authentication in practice.

4 Case study: OpenShift

The OpenShift Kubernetes distribution powers CERN's private Platform-as-a-Service (PaaS). It enables users to run their applications in a containerized, cloud native environment. The platform is fully self-service based, meaning users have the possibility to perform administrative tasks within their project (Kubernetes namespace) autonomously. These tasks, such as creating new workloads, updating configuration or monitoring the status of the application, can be performed with the OpenShift Web Console which offers convenient access and is very beginner friendly. The OpenShift Web Console is integrated with CERN's aforementioned Single Sign-On infrastructure via OIDC / OAuth 2.0.

However, advanced tasks and scripting are only possible with the "oc" OpenShift command line client, which is the equivalent of Kubernetes' "kubectl" command line tool. With the introduction of two-factor authentication (2FA) at CERN this tool has become cumbersome to use because it is not compatible with 2FA login flows (OTP codes or hardware authentication devices). One workaround involves opening the OpenShift Web Console, generating and copying an authentication token and then finally passing it to the “oc” CLI — inefficient and not user friendly.

For this reason we set out to develop a helper tool that would allow users to authenticate from the command line. The Device Authorization Grant seemed ideal for this use case because it has no requirements about the environment it is running in, apart from the fact that it needs to be able to show text. In particular, it does not require confidential client credentials, can be run on remote devices and is independent from the kind of multi factor authentication.

We built a first prototype with the aforementioned “auth-get-user-token” command line tool to confirm the feasibility of the approach. This also allowed us to better understand the required steps for the full login flow. This prototype was then refined and packaged into a “oc” / “kubectl” plugin, which allows users to connect to the OpenShift platform by typing “oc sso-login <cluster_name>”.

```
$ oc sso-login paas
Open the following link to log in:
  https://auth.cern.ch/auth/realms/cern/device?user_code=SQYL-ZVBU
Waiting for login...
Logged into "https://api.paas.cern.ch" as "username" using the
token provided.

You have access to 42 projects. You can list all projects with
’oc projects’
```

Behind the scenes several tokens are being exchanged to perform the login, as shown in Figure 3. First the Device Authorization Grant is used to obtain an access token for a “public” OAuth 2.0 client, as described in the Section 3. Since the OpenShift platform uses a separate, confidential OAuth client for authentication, an OAuth Token Exchange request [8] needs to be performed from the “public” to the “confidential” client, which yields another access token. The Kubernetes API, which is part of the OpenShift platform, can not be used directly with OAuth access tokens however. Therefore, a third step needs to be performed to exchange the OAuth token with a Kubernetes Bearer token. Finally, this bearer token is stored locally in the “kubeconfig” file so the OpenShift command line client “oc” automatically uses it to authenticate each request sent to the API server. This token automatically expires after a couple of hours to avoid leaking long-lived credentials. The login helper tool detects expired authentication tokens and re-authenticates the user.

5 Conclusion

Authentication mechanisms in command-line interfaces remain essential for many users of web applications. Previously, cookie-based workarounds were mostly used, but they delegate user authentication to Kerberos. This has shown its limitations when we started to introduce modern standards for two-factor authentication, and when API developers preferred using JWT or OAuth to identify their users.

In order to have a standards compliant approach that supports WebAuthn for two-factor authentication and works in different terminal environments, we have proposed to use OAuth 2.0 Device Authorization Grant in Section 3. This authentication flow has many advantages

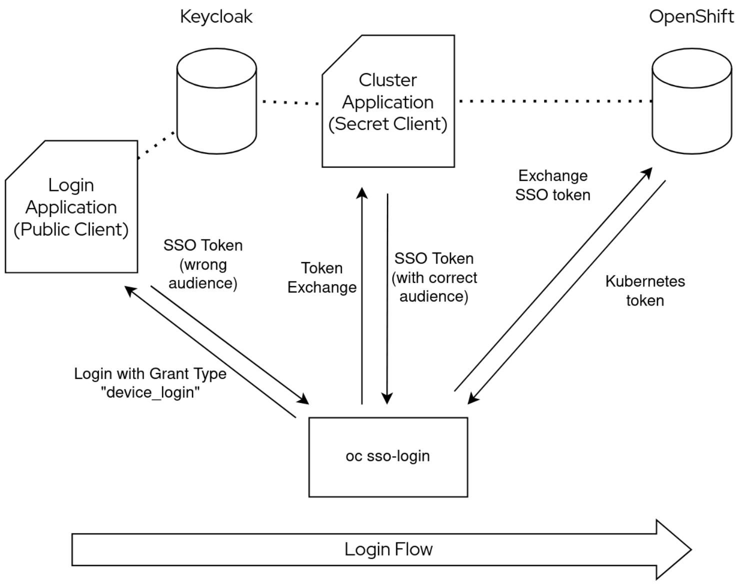

Keycloak
OpenShift
Cluster Application (Secret Client)
Login Application (Public Client)
Exchange SSO token
SSO Token (wrong audience)
Token Exchange
SSO Token (with correct audience)
Kubernetes token
Login with Grant Type "device_login"
oc sso-login
Login Flow

Figure 3. Visualization of the login flow that connects users to the OpenShift platform.

over previous solutions, but we have shown that it could result in double authentication in environments that still use Kerberos.

Finally, Section 4 presented a case study of CERN's private Platform-as-a-Service based on OpenShift, which has implemented this authentication mechanism for the "oc" command line client. In this case it was possible to use a public client to get credentials for a confidential client by using Token Exchange.

References

[1] Aguado Corman, Asier, Fernández Rodríguez, Daniel, Georgiou, Maria V., Rische, Julien, Schuszter, Ioan Cristian, Short, Hannah, Tedesco, Paolo, EPJ Web Conf. **245**, 03012 (2020)

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

[3] *Using OAuth 2.0 to Access Google APIs* (2023), accessed 04-09-2023, https://developers.google.com/identity/protocols/oauth2

[4] *Keycloak Server Administration Guide* (2023), accessed 04-09-2023, https://www.keycloak.org/docs/latest/server_admin/#_kerberos

[5] D. M'Raihi, J. Rydell, M. Pei, S. Machani, *TOTP: Time-Based One-Time Password Algorithm*, RFC 6238 (2011), https://www.rfc-editor.org/info/rfc6238

[6] J. Hodges, G. Mandyam, M.B. Jones, *Registries for Web Authentication (WebAuthn)*, RFC 8809 (2020), https://www.rfc-editor.org/info/rfc8809

[7] W. Denniss, J. Bradley, M.B. Jones, H. Tschofenig, *OAuth 2.0 Device Authorization Grant*, RFC 8628 (2019), https://www.rfc-editor.org/info/rfc8628

[8] M.B. Jones, A. Nadalin, B. Campbell, J. Bradley, C. Mortimore, *OAuth 2.0 Token Exchange*, RFC 8693 (2020), https://www.rfc-editor.org/info/rfc8693