EPJ Web of Conferences **251**, 02036 (2021) https://doi.org/10.1051/epjconf/202125102036
CHEP 2021

Secure Command Line Solution for Token-based Authentication

Dave Dykstra[1,1], *Mine* Altunay[2], *Jeny* Teheran[2]

[1]Scientific Computing Division, Fermilab, Batavia, IL, USA
[2]Core Computing Division, Fermilab, Batavia, IL, USA

Abstract. The WLCG is modernizing its security infrastructure, replacing X.509 client authentication with the newer industry standard of JSON Web Tokens (JWTs) obtained through the Open ID Connect (OIDC) protocol. There is a wide variety of software available using the standards, but most of it is for Web browser-based applications and doesn't adapt well to the command line-based software used heavily in High Throughput Computing (HTC). OIDC command line client software did exist, but it did not meet our requirements for security and convenience. This paper discusses a command line solution we have made based on the popular existing secrets management software from Hashicorp called **vault**. We made a package called **htvault-config** to easily configure a vault service and another called **htgettoken** to be the vault client. In addition, we have integrated use of the tools into the **HTCondor** workload management system, although they also work well independent of **HTCondo**r. All of the software is open source, under active development, and ready for use.

1 Introduction

When the Worldwide LHC Computing Grid (WLCG) was built, X.509 client authentication was chosen as the basic method of distributing authorization around the world, in particular X.509 proxy certificates. Although X.509 is ubiquitous for authenticating servers to clients, it was never very much used by the industry to authenticate clients to servers. As a result, much of the software used for authentication and authorization had to be maintained by the grid High Throughput Computing (HTC) community. This limited the amount of effort that could be put into improvements and made much of the software less convenient to use than it could be.

Since that time, the industry has converged on new standards for authentication and authorization, in particular OAuth 2.0 [1] for obtaining authorization tokens, its Open ID Connect (OIDC) [2] specialization for HTTP, and JSON Web Tokens (JWTs) [3]. Much off-the-shelf software is available that supports these standards. JWTs are very attractive because they are much more flexible in what they authorize than X.509 proxy certificates were, so they can be made to be very limited and therefore less of a risk if they are ever stolen. JWTs also have the scalability needed by HTC in that, like X.509 proxy certificates, they can be fully verified by the end clients. That enables them to be replicated to many independent jobs without requiring all of them to contact a server to verify authenticity.

The primary disadvantage of applying the use of these new standards to HTC is that they are designed for web browser-based applications, and many of our applications are based on the command line. That means that on the client side there is very little software available

1 Corresponding author: dwd@fnal.gov

for the command line, and that it inconveniently requires the use of a web browser for at least the initial authentication and authorization by the user.

In cooperation with the WLCG Authorization Working Group, we evaluated [4] the best existing command line OIDC client tool **oidc-agent** [5] and found that it didn't meet our security requirements and especially our convenience requirements. We didn't want a new system that was much less convenient or any less secure than our existing system. Our current system for obtaining and storing X.509 proxy certificates [6] is completely hidden from most of the users, authenticating with Kerberos, and it stores relatively long-lived credentials in a separate secured server called **MyProxy** [7]. Those longer-lived credentials are used by the **HTCondor** [8] workload management system to send updated short-lived credentials with computing jobs. The equivalent long-lived credential in OAuth is called a refresh token, and oidc-agent stores that encrypted on disk and unencrypted in memory. That means that in addition to the web browser interaction always required by OIDC to obtain a refresh token, with oidc-agent each user has to keep the encrypted refresh token secure and enter in a passphrase whenever a background process needs to be restarted. The users also have to go through an additional one-time step to register the oidc-agent command as a client of the OIDC token issuer.

Instead, we found that we could use an existing, popular general purpose and open source secrets management software package called **vault** [9] from Hashicorp to store the refresh tokens of all the users from multiple Virtual Organizations (VOs). Vault already supported OIDC and Kerberos and has a very flexible plugin architecture and REST API. The rest of this paper describes the solution based on vault that we came up with. The solution has very similar security as our existing system, and its convenience is also very similar to our existing system except that users are required to authenticate with their web browsers once.

2 Architecture and design

The components of our solution are (1) vault, (2) a vault configurator package we made called **htvault-config** [10], and (3) a vault client we made called **htgettoken** [11]. Vault is registered as an OIDC client (or multiple clients) of a token issuer (or multiple token issuers) and holds the OIDC client id and secret as part of its configuration. Vault in this solution takes on the role that web applications or portals play in a typical OIDC flow, shared by many users with web browsers. Once a refresh token is stored in vault, htgettoken can continue to obtain new OAuth access tokens (that is, JWTs) completely from the command line.

The htvault-config package (described more fully in section 3) does all the configuration of vault, in particular defining the OIDC token issuers and roles within each issuer. There is expected to be a token issuer defined for each VO, although they may be mapped to a common issuer. Different roles within an issuer map to a different list of scopes requested from an issuer. The package additionally configures storage paths in vault (similar to paths on a filesystem) to store a refresh token for each issuer +role+user combination. Users are granted access to all their refresh tokens when they authenticate with either OIDC or Kerberos, by use of a vault token generated by vault. (Vault supports many other ways of getting new vault tokens as well, and we will probably support more in the future; in particular we want to add one using ssh-agent authorized keys.) If a client reads from a vault path where a refresh token is stored, vault contacts the token issuer for a new access token.

The htgettoken tool (described more fully in section 4) controls the flows for obtaining tokens. It carries out all the necessary steps to obtain a new access token, and it is designed to be run from a shared job submission server. When a working vault token is not already stored on the machine where it is invoked, htgettoken obtains a vault token and stores it unencrypted on that machine. In order to follow the spirit of security guidelines established for X.509 short-lived user credentials [12], htgettoken will only store unencrypted vault

tokens that last 1 million seconds (~11.5 days) or less, default 1 week. When no refresh token has yet been stored in vault, htgettoken requests vault to use the OIDC flow, which includes using a web browser. For the rest of the week on that machine, htgettoken uses that vault token to obtain access tokens. When the vault token expires, or if the user logs in to a different machine, htgettoken uses Kerberos to obtain a new 1 week vault token.

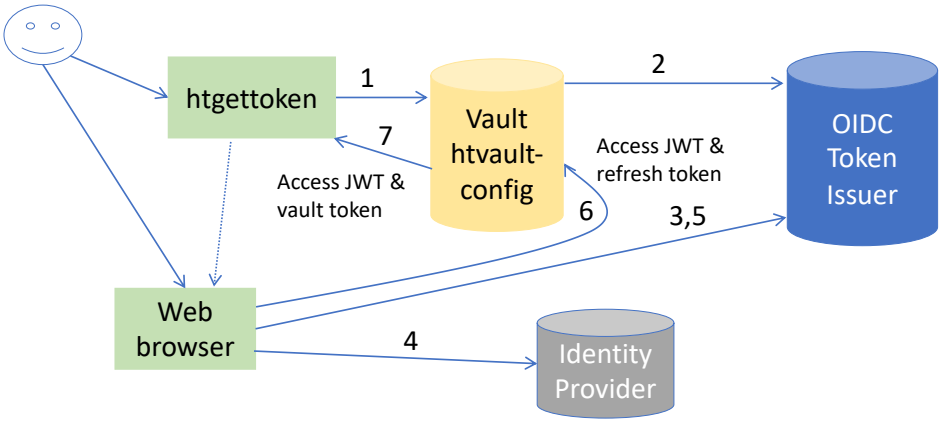

Figure 1. OIDC flow with vault and htgettoken

Figure 1 shows the flow that happens when obtaining an initial refresh token. The example includes a token issuer that authenticates through federated identity, with a separate identity provider. The steps are as follows:

1. Since no vault token exists, htgettoken contacts vault to do the OIDC flow.
2. Vault contacts the token issuer to start a transaction. Vault replies to htgettoken with a URL for the user's web browser that includes enough information to continue the flow.
3. The user's web browser contacts the token issuer and the user chooses the institution they want to authenticate with.
4. The token issuer redirects the web browser to the identity provider, where the user logs in.
5. The identity provider redirects the web browser back to the token issuer for final approval and granting of permission
6. The token issuer redirects the browser back to vault along with the access token and refresh token.
7. Vault returns the access token and a vault token to htgettoken.

3 htvault-config

The htvault-config package makes it convenient for anyone to install and configure vault for use with htgettoken. In addition to configuration, it includes two compiled golang vault plugins. One of the plugins, vault-plugin-auth-jwt [13], comes standard from Hashicorp with vault, but we have added a couple pull request patches that have not yet been accepted by the plugin maintainers although they have indicated approval. Someday we should be able to remove this plugin from our package. The other plugin, vault-plugin-secrets-oauthapp [14],

comes from Puppet Labs. We use the latter plugin to store refresh tokens and convert them into access tokens.

Configuring vault with the htvault-config rpm is quite simple. The configuration is done through a series of YAML files that the administrator can split up as desired. Only a small number of configuration items are needed for each token issuer, and any number of roles can be defined that each map to a set of scopes to request. The WLCG common JWT profile [15] is supported.

The package installs a configuring systemd service so starting or restarting only the vault automatically configures it. Reconfiguration without restarting vault can be done by restarting the htvault-config service separately.

Vault has a standard builtin high-availability feature, and htvault-config makes it easy to use that by running 3 machines as a single high availability service. Configuration is done by simply defining the name of the cluster, the name of the master machine of the cluster, and the names of the two peer machines. The full configuration is only applied on the master machine. Service continues if one or two of the machines are down, although reconfiguring can only be done when the master machine is up.

4 htgettoken

The htgettoken command is implemented in python3. Some of the libraries it uses were not readily available as rpm packages, so the rpm package bundles all needed python code into a standalone binary using PyInstaller [16]. As a result, there are no python dependencies on the package itself.

The htgettoken command has many options as detailed in the command's own usage help and in its man page. The only required option is the one that specifies the address of the vault server, although an issuer is also usually specified and often a role is as well. There are also options to specify the audience and a list of scopes. Those options can only be used to reduce the privileges of an access token; maximal privileges are defined in the vault configuration and the token issuer makes sure to only issue privileges allowed to that user.

Vault tokens are opaque hashes, so they do not contain discoverable information in them such as an expiration time. The only way to find out if they are valid is to try using them. For that reason, the default behavior of htgettoken is to obtain an access token using the following algorithm:

1. If a vault token exists, try to use it to retrieve an access token.
2. If no access token has been successfully retrieved, try to use Kerberos to obtain a vault token, and if that works write out the vault token and try to use it to read an access token.
3. If no access token has been successfully retrieved, try to use OIDC authentication to obtain a vault token and to store a refresh token in vault, and if that works write out the vault token and try to use it to retrieve an access token.
4. If an access token has been successfully retrieved write it out, otherwise return an error.

By default, access tokens are written to the disk according to the WLCG bearer token discovery specification [17] which prefers writing to a location ($XDG_RUNTIME_DIR) that disappears when a user logs out. Vault tokens by default follow a similar naming scheme except in /tmp because they last longer than access tokens.

5 HTCondor integration

We also integrated our solution with HTCondor. This was based on the existing HTCondor solution for SciTokens [18] which has its own custom OIDC client implementation. Users specify the issuers and roles of tokens they want to be sent along with their jobs and optionally may also specify reduced audiences and scopes they want in tokens. Vault tokens corresponding to all the tokens a user requests are automatically obtained by a condor_submit command, stored by condor, and used to obtain short-lived (default 1 hour) access tokens that are sent to jobs and automatically renewed.

All the components of the integration are installed with a **condor-credmon-vault** rpm that is part of the condor package. The two main components are **condor_vault_storer** and **condor_credmon_vault**.

condor_submit invokes the condor_vault_storer bash script which runs htgettoken to obtain the corresponding vault tokens. If possible, htgettoken will get them silently, but if the refresh token is not yet available htgettoken prompts the user to authenticate through a web browser. Those vault tokens last for 4 weeks and are sent to the condor_credd process to manage. htgettoken is then called again to exchange the 4 week vault token for a 1 week vault token and that is stored unencrypted on the local disk. The longer vault token is needed because jobs can be queued and run for a long time.

condor_credmon_vault is a python daemon that runs alongside condor_credd and exchanges the vault tokens for access tokens periodically, by default every 20 minutes. It directly invokes a simple vault API, without using htgettoken. Other HTCondor components then push those access tokens to the jobs.

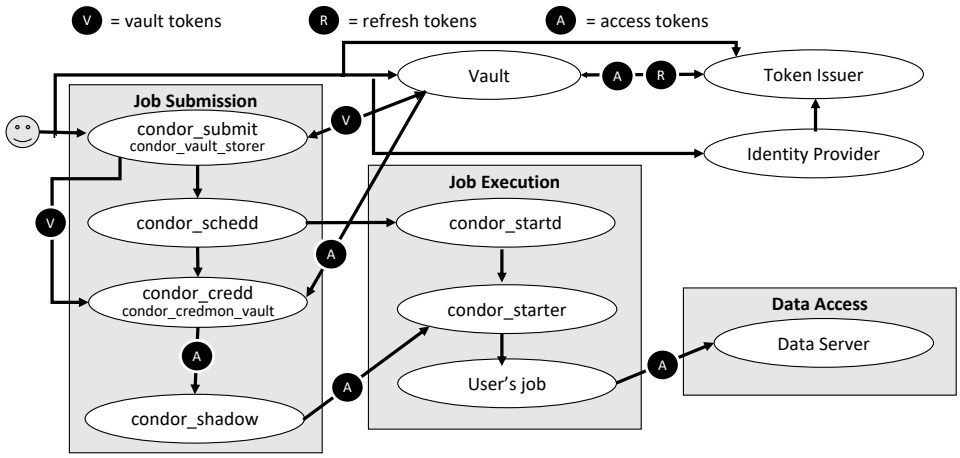

Figure 2. Vault integration components in a full HTCondor deployment

Figure 2 shows the new components in the context of a full HTCondor deployment and shows the places that tokens are exchanged.

This integration is in the OSG distribution of HTCondor-9.0.x, and is scheduled to be in HTCondor's own distribution beginning in the 9.1.x release series.

6 Support for automated processes

Getting the initial refresh token in this solution requires a web browser, so that raises a question about what to do to support automated processes that need to run without user intervention. The system supports that, in two ways.

One way is through a special capability of Kerberos. Getting Kerberos credentials typically requires interactive reauthentication by a user weekly, but many Kerberos installations also support "robot" credentials that may be used indefinitely. They are given a name with path subcomponents such as "user/type/machinename" where "user" is the ID of the user that owns them, "type" is some arbitrary category name, and "machinename" is the fully qualified domain name of the host machine that uses the credential. Since vault stores secrets by path names, OIDC authentication is set up to give users access to all vault subpaths under their IDs. When users use htgettoken to initially obtain a refresh token through the OIDC flow, they can tell htgettoken to store the refresh token in a subpath matching the robot credential name. From that point forward the Kerberos robot credential can be used to get a vault token with access to that subpath.

The alternative method to Kerberos is for the administrator of vault to manually create an indefinitely renewable vault token that provides access to a specific subpath in vault and to send the token to the owner of the automated process.

7 Conclusions

This solution for command line-based use of use of JSON Web Tokens has been designed to be convenient for users and to be secure while taking advantage of existing industry standard software where possible. It has not yet been put into production use, but it is fully functional. It is expected to evolve as production experience is gained while keeping the basic components the same. All the components are available as open source, and rpms are currently being distributed by the Open Science Grid.

Acknowledgements

This document was prepared using the resources of the Fermi National Accelerator Laboratory (Fermilab), a U.S. Department of Energy, Office of Science, HEP User Facility. Fermilab is managed by Fermi Research Alliance, LLC (FRA), acting under Contract No. DE-AC02-07CH11359.

References

1. *OAuth 2.0,* https://oauth.net/2/, accessed: 2021-06-02
2. *OpenID Connect,* https://openid.net/connect/, accessed: 2021-06-02
3. *JSON Web Token,* https://jwt.io/, accessed: 2021-06-02
4. *WLCG Authorization Working Group client tools technical investigation,* https://github.com/WLCG-AuthZ-WG/client-tools, accessed: 2021-06-02
5. *oidc-agent,* https://indigo-dc.gitbook.io/oidc-agent/, accessed: 2021-06-02
6. D. Dykstra, et. al., CISRC '16 Proceedings of the 11th Annual Cyber and Security Research Conference, 10 (2016), https://dl.acm.org/doi/10.1145/2897795.2897807, accessed: 2021-06-02
7. *MyProxy*, http://grid.ncsa.illinois.edu/myproxy/, accessed: 2021-06-02
8. *HTCondor*, https://research.cs.wisc.edu/htcondor/, accessed: 2021-06-02

9. *vault*, https://www.vaultproject.io/, accessed: 2021-06-02
10. *htvault-config*, https://github.com/fermitools/htvault-config/, accessed: 2021-06-02
11. *htgettoken*, https://github.com/fermitools/htgettoken/, accessed: 2021-06-02
12. *IGTF Short Lived Credential Service*, https://www.igtf.net/ap/slcs/, accessed: 2021-06-02
13. *vault-plugin-auth-jwt*, https://github.com/hashicorp/vault-plugin-auth-jwt/, accessed: 2021-06-02
14. *vault-plugin-secrets-oauthapp*, https://github.com/puppetlabs/vault-plugin-secrets-oauthapp/, accessed: 2021-06-02
15. *WLCG Common JWT Profiles*, https://github.com/WLCG-AuthZ-WG/common-jwt-profile/, accessed: 2021-06-02
16. *PyInstaller*, https://www.pyinstaller.org/, accessed: 2021-06-02
17. *Bearer token discovery*, https://github.com/WLCG-AuthZ-WG/bearer-token-discovery/, accessed: 2021-02-25
18. *SciTokens*, https://scitokens.org/, accessed: 2021-02-25