Trusted CI: The NSF Cybersecurity Center of Excellence

Globus Auth First Principles Vulnerability Assessment

July 7, 2020

Final Report

Distribution: Public

Benjamin Kinzer[1], Elisa Heymann[2], Barton P. Miller[3]

[1] Student Researcher, bkinzer@cs.wisc.edu
[2] Software Assurance Lead, elisa@cs.wisc.edu
[3] Co-PI, bart@cs.wisc.edu

About Trusted CI

The mission of Trusted CI is to provide the NSF community with a coherent understanding of cybersecurity, its importance to computational science, and what is needed to achieve and maintain an appropriate cybersecurity program[4].

Acknowledgments

Trusted CI's engagements are inherently collaborative; the authors would like to thank the Globus team, specifically Mattias Lidman, Stuart Martin, Rachana Ananthakrishnan and Sam Claassens for the collaborative effort that made this document possible.

This document is a product of the Center for Trustworthy Scientific Cyberinfrastructure (Trusted CI). Trusted CI is supported by the National Science Foundation under grant 1920430. For more information about the Center for Trustworthy Scientific Cyberinfrastructure please visit: https://trustedci.org/. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Using & Citing this Work

Cite this work using the following information:

Benjamin Kinzer, Elisa Heymann, Barton P. Miller. "TrustedCI: The NSF Cybersecurity Center of Excellence Globus Auth Report". TrustedCI: The NSF Cybersecurity Center of Excellence. June 2020.

This work is available on the web at the following URL: http://hdl.handle.net/2142/106617

[4] https://trustedci.org/mission

Table of Contents

List of Figures

Executive Summary

Trusted CI collaborated with the Globus team to assess the security of Globus Auth service. Globus Auth provides an identity and access management platform for securing services and clients in the research and education ecosystem. It serves to broker authentication and authorization interactions between end-users, identity providers, resource servers (services), and clients (including web, mobile, desktop, and command line applications, and other services). Leveraging federated identities and acting as an authorization server as defined in the OAuth2 standard, Globus Auth provides a security layer for users and applications to access protected resources.

Trusted CI collaborated with the Globus Auth team to assess the security of their identity and access management platform service. The goal of Globus Auth is to link services and clients in the research and education community by providing an authorization service. Globus Auth acts as an authorization server in the OAuth2 standard and uses the specification in order to link users with their protected data.

We conducted an in-depth vulnerability assessment of Globus Auth by applying the First Principle Vulnerability Assessment (FPVA)[5] methodology. Our FPVA analysis started by mapping out the architecture and resources of the system, paying attention to trust and privilege used across the system, and identifying the high value assets in the system. From there we performed a detailed code inspection of the parts of the code that have access to the high value assets.

We assessed the Globus Auth version with tag “release/production/2019-08-21.0” using a Docker container and sample clients sent to us by the Globus Auth team. The sample clients had IDs, secrets, whitelisted redirect URLs, and approved scopes like a real client would have, but we created all client requests to Globus Auth manually. The assessment covered the core Globus Auth code, which includes most of the APIs covered in the Globus Auth API Reference[6]. We focused our effort on APIs implementing the OAuth2 standard. We collected the results from each step of the FPVA methodology to form this report for the Globus Auth team at the end of the engagement.

This report also includes a discussion of the parts of Globus Auth that we inspected but where no apparent issues were found. Though it is impossible to certify that code is free of vulnerabilities, we have increased our confidence in the security of those sections of the code. We provide detailed explanations in order to back this confidence.

[5] James A. Kupsch, Barton P. Miller, Eduardo César, and Elisa Heymann, “First Principles Vulnerability Assessment”, 2010 ACM Cloud Computing Security Workshop (CCSW), Chicago, IL, October 2010.
[6] https://docs.globus.org/api/auth/reference/

Overall, our team found no security issues in the Globus Auth code, however we made several recommendations (in Section 6.1) to further increase security based on findings from our assessment. None of the recommendations are for security flaws, but ways in which Globus Auth can detect and prevent attacks faster and more effectively.

1 Overview

This document describes the engagement between Trusted CI and Globus that occurred from July 2019 to February 2020. The goals of the engagement were to evaluate the technology and architecture of the Globus Auth software, and perform a code-level security review of the Globus Auth software.

1.1 Background

Globus Auth is an authorization and authentication service created by Globus to broker secure interactions between users, client applications and services[7]. The service allows users to use their federated identity, and optionally link identities from multiple identity providers, providing
authentication context and credentials needed for protected resources to make authorization decisions. Globus Auth implements the OAuth2[8] and OpenID Connect[9] specifications to provide the security needed for interactions across users,identity providers, clients, and resource servers. Tokens are sent to clients by Globus Auth after a user has authenticated with an identity provider and the tokens can then be used to authenticate with the resource servers for access to protected resources.
Including dependencies and the Docker container used to run Globus Auth, the application contains about 1,010,000 lines of code not counting comments and whitespace. Excluding the Docker container, Globus Auth is 163,262 lines of code and without dependencies, there are 30,920 lines of code, including 21,255 lines of Python. The remaining lines are composed of 3,859 lines of XML, 2,615 lines of SQL, 2,093 lines of CSS, and 1,098 lines of YAML.

1.2 Methodology

The Trusted CI engagement for Globus Auth started on July 1, 2019. This engagement focused on performing FPVA on Globus Auth. The engagement was originally scheduled to end in December but ended in February because of changes in staff part way through the

[7] https://docs.globus.org/api/auth/specification
[8] https://tools.ietf.org/html/rfc6749
[9] https://openid.net/specs/openid-connect-core-1_0.html

engagement. Our FPVA assessment focused on the APIs featured in the Globus Auth API Reference, especially the authorization and token endpoints which are key to the OAuth2 flow.

Globus Auth provided the Trusted CI team with a Docker container for the assessment on July 9, 2019. We received an update on August 9, 2019 that allowed us to use Google as an identity provider. On August 19, 2019 the Docker container was updated to include two Python scripts to illustrate how clients and resource servers can make requests to Globus Auth. In September, logging into Globus Auth using Globus ID as an identity provider stopped working, so the Globus Auth team provided a workaround by logging in with Google on October 23, 2019.

There were some limitations that came with this Docker environment. The Docker container does not display some aspects of a client such as how a client using other grant types described in the OAuth2 RFC besides authorization codes are created and the differences between client types such as native vs. browser-based. This Docker implementation also did not show how identity providers are registered with Globus Auth, so only Globus ID and Google could be used as identity providers with the container.

2 Overview of First Principles Vulnerability Assessment

First Principles Vulnerability Assessment (FPVA) is an analyst-centric (manual) methodology that aims to focus the analyst's attention on the part of the software system and its resources that are most likely to contain vulnerabilities that would provide access to high-value assets. FPVA finds new threats to a system and is not dependent on a list of known threats. The FPVA methodology consists of five steps for evaluating a given piece of software.

1. **Architectural Analysis:** determine the major structural components of the system and how they interact. At this point, we produce architectural diagrams that illustrate the structure of the system. The primary deliverables of this step are Figures 1 and 2.
2. **Resource Identification:** identify key resources accessed by each component. Examples of these resources include files, databases, logs and devices. The Resource Diagrams we produced illustrate these resources and their connection to system components. The primary deliverables of this step are Figures 3, 4, and 5.
3. **Trust and Privilege Analysis:** identify the trust assumptions about each component, answering such questions as how are they protected and who can access them? Associated with trust is describing the privilege level at which each executable component runs.
4. **Component Evaluation:** examine relevant components in depth. A key aspect of the FPVA methodology is that this evaluation is guided by information obtained in the first

three steps. This helps to prioritize the work so that high value targets are evaluated first. Any vulnerabilities identified as well as all other work done during this step is logged for inclusion in the final report.

5. **Dissemination of Results:** a final report is prepared that includes the deliverables mentioned above as well as an outline of the work completed. We include recommendations as well as areas that have been investigated but no bugs or vulnerabilities were found. We then disseminate the final report to the requesting parties (i.e., the lead of the development team).

We adhered to these steps in the Globus Auth engagement. We note that Globus Auth is a large and complex software system, so no time-limited assessment activity will be able to find all possible sources of insecurity. Regular assessments of the software will help maintain its security. We also note here that ongoing attention to the security of the external software on which Globus Auth depends is necessary to keep up the application’s safety.

3 Architectural Analysis

Based upon our study of the Globus Auth documentation, testing environment, and code, we identified the attack surface and produced High Level Communication Flow and an Architectural diagrams seen in Figures 1 and 2. Among other things, these diagrams show how Globus Auth handles HTTPS requests and responses. We elaborate more on our findings in the following subsections.

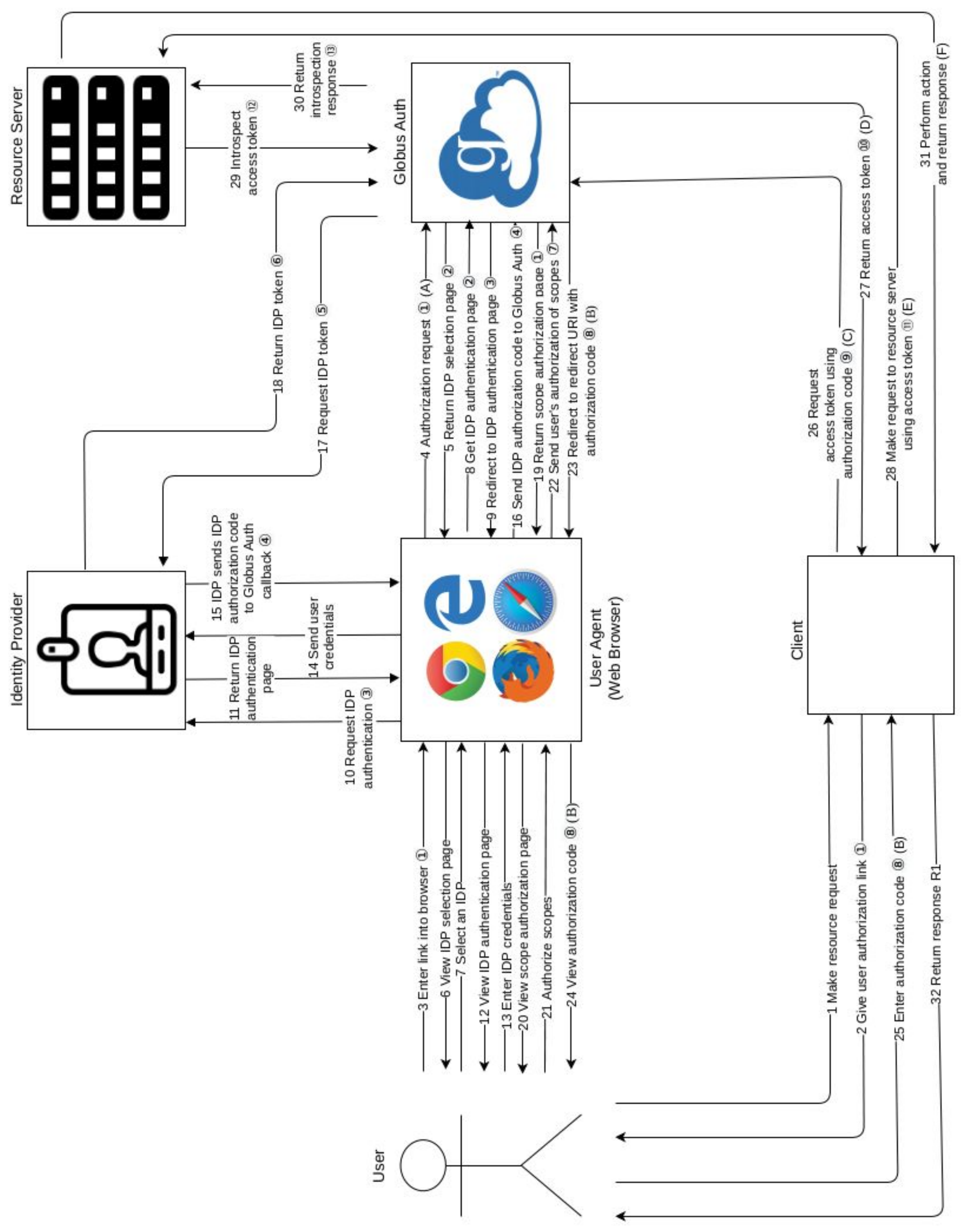
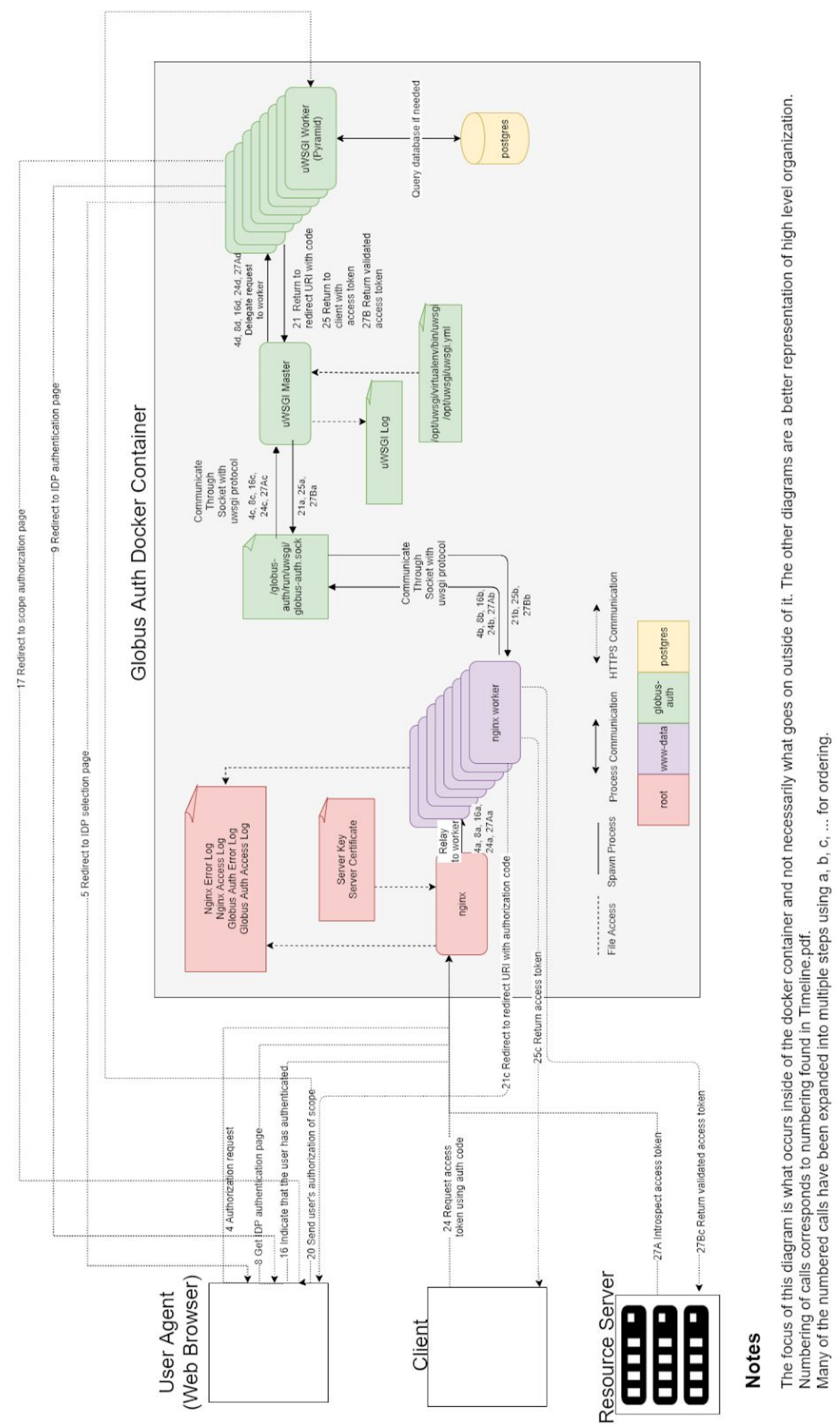

Resource Server
Globus Auth
Identity Provider
User Agent (Web Browser)
Client
User
1 Make resource request
2 Give user authorization link ①
3 Enter link into browser ①
4 Authorization request ① (A)
5 Return IDP selection page ②
6 View IDP selection page
7 Select an IDP
8 Get IDP authentication page ②
9 Redirect to IDP authentication page ③
10 Request IDP authentication ③
11 Return IDP authentication page
12 View IDP authentication page
13 Enter IDP credentials
14 Send user credentials
15 IDP sends IDP authorization code to Globus Auth callback ④
16 Send IDP authorization code to Globus Auth ④
17 Request IDP token ⑤
18 Return IDP token ⑥
19 Return scope authorization page ①
20 View scope authorization page
21 Authorize scopes
22 Send user's authorization of scopes ⑦
23 Redirect to redirect URI with authorization code ⑧ (B)
24 View authorization code ⑧ (B)
25 Enter authorization code ⑧ (B)
26 Request access token using authorization code ⑨ (C)
27 Return access token ⑩ (D)
28 Make request to resource server using access token ⑪ (E)
29 Introspect access token ⑫
30 Return introspection response ⑬
31 Perform action and return response (F)
32 Return response R1

Figure 1. High Level Communication Flow Diagram.


Globus Auth Docker Container
uWSGI Worker (Pyramid)
postgres
Query database if needed
4d, 8d, 16d, 24d, 27Ad Delegate request to worker
21 Return to redirect URI with code
25 Return to client with access token
27B Return validated access token
/opt/uwsgi/virtualenv/bin/uwsgi /opt/uwsgi/uwsgi.yml
uWSGI Master
uWSGI Log
Communicate Through Socket with uwsgi protocol
4c, 8c, 16c, 24c, 27Ac
21a, 25a, 27Ba
/globus-auth/run/uwsgi/globus-auth.sock
Communicate Through Socket with uwsgi protocol
4b, 8b, 16b, 24b, 27Ab
21b, 25b, 27Bb
nginx worker
Relay to worker
4a, 8a, 16a, 24a, 27Aa
nginx
Server Key Server Certificate
Nginx Error Log Nginx Access Log Globus Auth Error Log Globus Auth Access Log
File Access
Spawn Process
Process Communication
HTTPS Communication
root
www-data
globus-auth
postgres
17 Redirect to scope authorization page
9 Redirect to IDP authentication page
5 Redirect to IDP selection page
21c Redirect to redirect URI with authorization code
25c Return access token
4 Authorization request
8 Get IDP authentication page
16 Indicate that the user has authenticated
20 Send user's authorization of scope
24 Request access token using auth code
27A Introspect access token
27Bc Return validated access token
User Agent (Web Browser)
Client
Resource Server

Notes

The focus of this diagram is what occurs inside of the docker container and not necessarily what goes on outside of it. The other diagrams are a better representation of high level organization.
Numbering of calls corresponds to numbering found in Timeline.pdf.
Many of the numbered calls have been expanded into multiple steps using a, b, c, ... for ordering.

Figure 2. Architectural Diagram.

3.1 Attack Surface

Figure 1 outlines a typical flow communication through Globus Auth and its external entities. Details on how to interpret this diagram can be found in Appendix A. From this diagram, we identified four types of external entities that interact with Globus Auth:

- The client is an application that makes resource requests on behalf of users. Clients connect with Globus Auth to retrieve a token representing authorization.
- The resource server hosts protected resources and responds to resource requests made by clients. Resource servers use Globus Auth to validate such requests from clients.
- The identity provider checks the user's credentials and tells Globus Auth when a user has successfully been authenticated.
- A web browser acts as an intermediary user agent between the client, user, identity provider, and Globus Auth when authenticating a user.

The connections that these entities use to interact with Globus Auth make up the attack surface. Clients, resource servers, and identity providers all communicate with Globus Auth by creating a connection with Nginx, which acts as a web server for Globus Auth. HTTPS requests sent to and received from Nginx are the first point on Globus Auth's attack surface. A web browser can send and receive HTTPS requests to Globus Auth but a user can also provide input to various fields during authorization and client creation on the Globus Auth website. These fields are another important piece of the attack surface.

3.2 Architecture Diagram

Figure 2 outlines the internal architecture of Globus Auth and how components interact with one another. There are four main components: Nginx, uWSGI, Pyramid, and PostgreSQL.

Upon startup of the Globus Auth Docker container, Nginx is the first component to be run. Nginx is an HTTPS server. Nginx uses the SSL certificate (`server.crt`) and key (`server.key`). Nginx spawns workers that process incoming HTTPS requests and send outgoing responses. Requests and errors are logged in `/var/log/nginx/globusauth.access.log` and `/var/log/nginx/globusauth.error.log` as they occur.

Nginx workers pass HTTPS requests to uWSGI, a service that implements the Web Server Gateway Interface (WSGI), a protocol to pass requests from Nginx to the Python application. uWSGI logs its requests separately from Nginx. The service creates its own uWSGI workers. `/opt/uwsgi/uwsgi.yml` is used to configure uWSGI.

The Globus Auth application is written in Python and uses the Pyramid web framework. uWSGI workers process requests and execute the appropriate Python functions in the code base. These functions include requesting authorization for a user and requesting a token on behalf of a user.

Some Python functions make queries to PostgreSQL, the database of Globus Auth. SQL commands within the Python functions are used to retrieve, modify, and store information in the database which holds information such as access tokens, client secrets, and redirect urls.

HTTPS responses follow this same path in the opposite direction.

4 Resource Identification

Following the production of architectural diagrams, we identified the files with key resources accessed by the components. We used this information to produce the resource diagrams in Figures 3, 4, and 5. For each file, we inspected its permissions, as well as where and how it is used by Globus Auth.

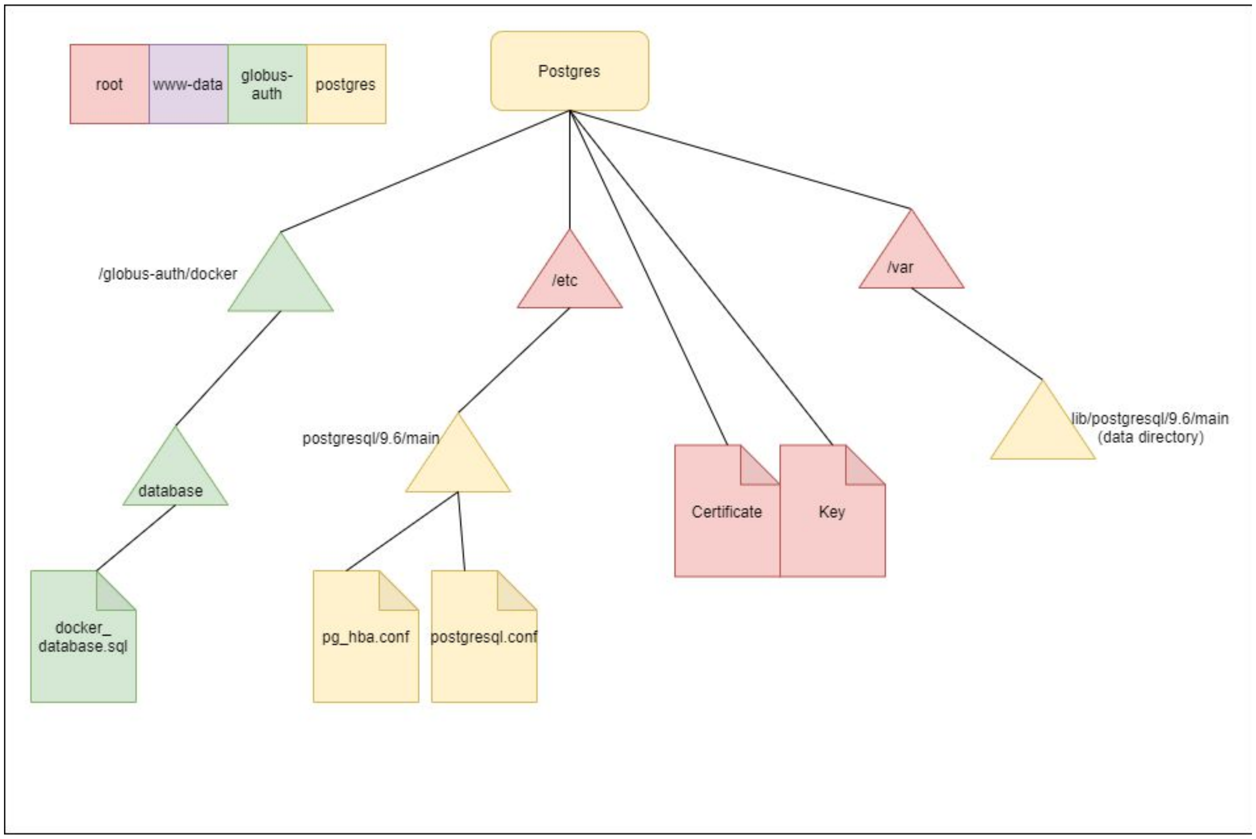


Figure 3. PostgreSQL Resource Diagram.

Figure 3 shows the resources used by PostgreSQL. The PostgreSQL database is a key resource protected by Globus Auth. It contains information such as tokens, authorization codes, and client secrets. The database is queried by various Python methods using SQLAlchemy[10] (an SQL toolkit) and the data is stored in `/var/lib/postgresql/9.6/main/`. All files in this data directory have permissions that allow reading and writing by only the Postgres user. Upon creation of the database in the Docker container, a privileged database user named "myapp" is created that is used by Globus Auth for all queries of the database. Tables and data are added to the database using `/globus-auth/docker/database/docker_database.sql`. The configuration files are found in `/etc/postgresql/9.6/main/`. The general configuration file is `postgresql.conf` and client authentication is specified by `pg_hba.conf`. PostgreSQL uses SSL and has a certificate and key. The certificate is `ssl-cert-snakeoil.pem` and the key is `ssl-cert-snakeoil.key`. The permissions of these files do not allow them to be written to by `other` users and the database cannot be read by these users.

[10] https://www.sqlalchemy.org

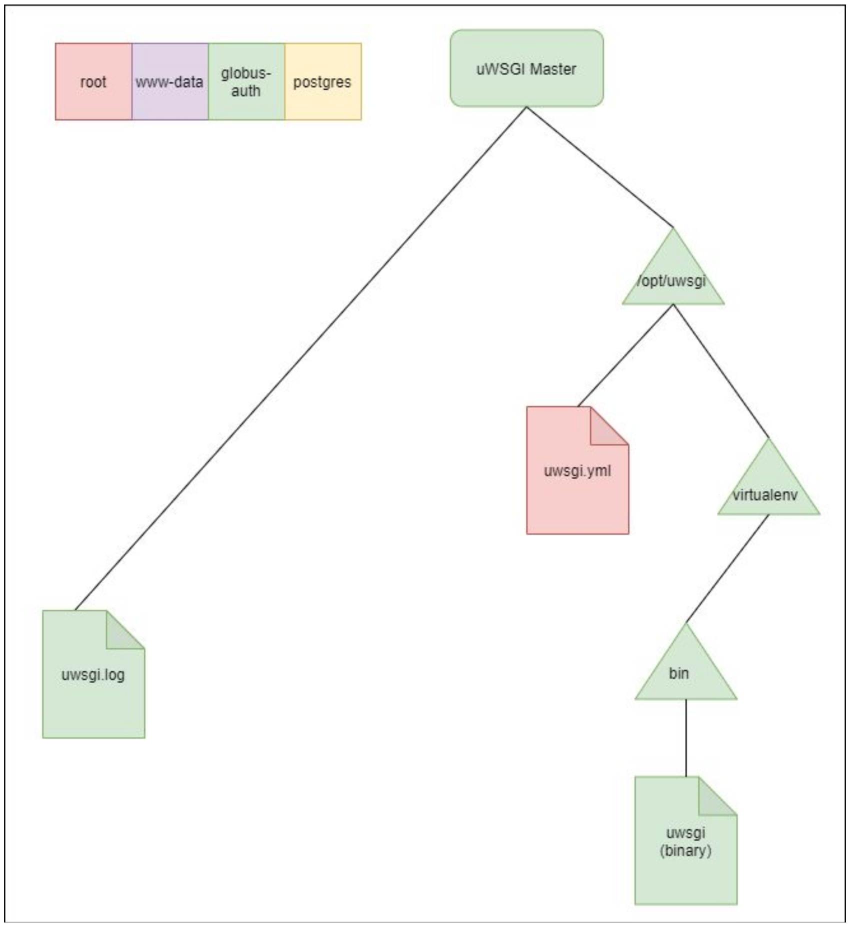


Figure 4. uWSGI Resource Diagram.

Figure 4 describes the resources used by uWSGI. `/opt/uwsgi/virtualenv/bin/uwsgi` is the uWSGI binary. `/opt/uwsgi/uwsgi.yml` configures uWSGI. As mentioned in the Architectural Analysis section, uWSGI also logs HTTPS requests and responses in the `uwsgi.log` file. The binary can be read and executed by `other` users and the other files can only be read by `other` users.

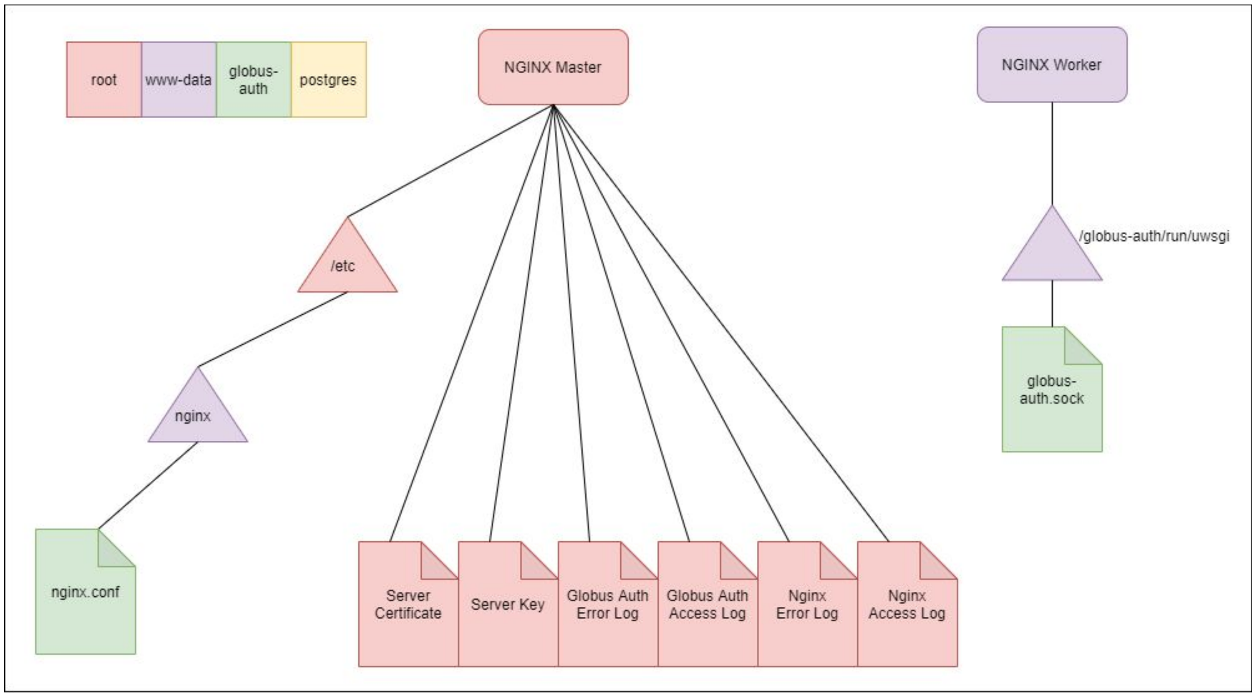


Figure 5. Nginx Resource Diagram.

Finally, the resources used by Nginx are depicted by Figure 5. Nginx uses SSL to communicate with external entities. The SSL certificate and key are held in the `server.crt` and `server.key` files.. Nginx logs HTTPS requests and responses in the `/var/log/nginx/` folder. The `/etc/nginx/nginx.conf` file configures Nginx. Requests and responses also pass through a socket found in `/globus-auth/run/uwsgi/globus-auth.sock`. The socket is not readable, writeable, or executable by `other` users. All other Nginx files are only readable by `other` users.

5 Trust and Privilege Analysis

There is implicit trust in any process running as `root` or resource owned by `root`. Any communication from an unprivileged process to a privileged process needs to be screened, and any data read from a `root` owned resource needs to be checked to make sure that private data is not released.

In Globus Auth, the Docker container and the Nginx master process run as `root` and uwsgi as the user `globus-auth`. When `root` starts Nginx, the user www-`data` is set as the owner of the nginx worker processes. The `postgres` user manages all processes relating to PostgreSQL.

Globus Auth's important resources are owned by `root`, `globus-auth`, or `postgres` as shown in Figures 3, 4, and 5.

6 Component Evaluation

This section describes the areas of focus for the component analysis step of our assessment. In this step, we performed code inspection looking for weaknesses that could be exploited.

6.1 Recommendations

Section 6.1 outlines areas that the Globus Auth team could address to enhance the security of Globus Auth. We found no vulnerabilities in the code. These are recommendations we make based on our exploration of Globus Auth.

6.1.1 Resource Owner Password Credentials Grant

Summary

The Resource Owner Password Credentials grant type places too much trust in the client and should not be used.

Description

The Resource Owner Password Credentials grant type is described in the OAuth2 RFC and removes much of the safety provided by the Authorization Code grant type. In the password grant type, users send their credentials to clients, which in turn send them to Globus Auth along with a client id and secret in exchange for a token. The problem with this process is that it strips power away from the user. Globus Auth must believe that the user was the one who triggered this flow initially, and that the user approved all of the scopes requested. Globus Auth cannot trust that this client is not maliciously acting as the user unless it is owned and operated by Globus Auth itself. Furthermore, this grant type follows the same flow as phishing attacks, making it easier for users to fall for these attacks in the future.

Result and Suggestion

Globus does not make this grant type available to all clients, and has a policy of vetting and allowing a few clients to use this. At present a dozen clients are allowed to use this grant type. We recommend Globus continue to restrict the use, periodically audit the use and need, and minimize the risk to the user by making few exceptions."

6.1.2 Refresh Token Rotation

Summary

Implementing refresh token rotation would help clients detect token theft faster.

Description

Currently, clients can use the same refresh token repeatedly to request new access tokens. While this is an acceptable practice, refresh token rotation has clear benefits. Refresh token rotation means invalidating previous refresh tokens and sending a new one upon each use. If a refresh token is stolen from a client and is used by either the client or the attacker, it will be invalidated. Then, when the second entity tries to use the refresh token, an error will be returned. This will help notify the client that their token has been stolen and that they need to act on the situation. Without refresh token rotation, the client may never discover the theft of a token.

Result and Suggestion

The Globus Auth team should consider implementing refresh token rotation. This means that every time a refresh token is used, it is revoked by Globus Auth and a new refresh token is created along with the new access token.

6.1.3 Scope Removal Denial of Service

Summary

This attack could occur not because Globus Auth is vulnerable, but because a client is vulnerable to CSRF attacks. We list it here, however, because Globus Auth does have a method of prevention by notifying the user when scopes have been removed in authorization requests.

After a user is authenticated, an attacker could use CSRF on a client's page to make an authorization request, this time with fewer scopes. Globus Auth will process this request and the end result will be a token associated with the user with fewer scopes. When the client uses this token to request a resource, an error will be returned instead of the resource.

Description

Assume that the victim is a registered user with the client App which uses resource servers A and B for data. The attacker has managed to use CSRF to insert an image element into the client's webpage to create an authorization request with Globus Auth when the user visits that page, something like:

<img
src="https://localhost/v2/oauth2/authorize?client_id=1f292832-7fbe-4351-800a-a670eaf2c2c7&redirect_uri=https://app.com/redirect&scope=A&response_type=code&state=app_state\">

1. User goes through authorization flow as normal through an authorization request such as

https://localhost/v2/oauth2/authorize?client_id=1f292832-7fbe-4351-800a-a670eaf2c2c7&redirect_uri=https://app.com/redirect&scope=A+B&response_type=code&state=app_state\. Note the scopes requested are A and B. The flow ends with a token with scopes A and B for the user held by the client.

2. The user visits the client page the attacker manipulated which sends an authorization request to Globus Auth such as https://localhost/v2/oauth2/authorize?client_id=1f292832-7fbe-4351-800a-a670eaf2c2c7&redirect_uri=https://app.com/redirect&scope=A&response_type=code&state=app_state\. Note the scope requested is just for A.
3. Globus Auth sees the request and sees that the user is already authenticated and has authorized these scopes so it returns an authorization code to the proper redirect.
4. The client's redirect handles the code return and makes a request for a token with it.
5. Globus Auth returns a token for the user with a scope for A.
6. The user requests data held by resource server B.
7. The client sends the user's token to server B.
8. Server B sees that the token does not have the proper scope and returns an error to the client.

Result and Mitigation

This would result in the user receiving an error when requesting data they should have access to. Globus Auth has the opportunity to mitigate this denial of service attack by checking to see if scopes have been removed from previously authenticated users in the `allowed_to_bypass_consent_prompt` method in `/globus-auth/repo/globus/auth/oauth/views.py`. If they have, the user should be shown the scope authorization page again.
The assessment team acknowledges that a consent page has the potential problem of confusing users. We emphasize the responsibility of applications/clients to prevent this kind of issues.

6.1.4 Authorization Code Hashing

Summary

Hashing authorization codes in the database would provide an extra layer of security.

Description

Globus Auth uses strong hashing methods to store valuable information such as access tokens and client secrets. It would be wise to consider hashing authorization codes as well since they act in a similar way to access tokens. Hashing codes makes them more secure in timing attacks as discussed in section 6.2.2. Codes are still protected due to the fact that an attacker would also need to steal a client secret in order to use them. In addition, authorization codes have short expiration times and are often used quickly.

Result and Suggestion

Since authorization codes are a key part of the OAuth2 flow, hashing them in the database would make theft harder. Hashing is already implemented for client secrets and tokens so implementation would not be difficult.

6.1.5 Implicit Grant OpenID Connect

Summary

OpenID should be implemented for implicit grant types. Note that this is an opportunity for additional features to support applications that use implicit grant, and not a security concern.

Description

OpenID Connect allows resource servers to gain information on how, why, and when access tokens are issued by Globus Auth. While OpenID Connect is implemented for other grant types in Globus Auth, it does not work as specified for implicit grants. Implicit grants are a different type of access flow described in the OAuth2 RFC where clients get access tokens immediately after authentication instead of an authorization code. In order to get id tokens as part of the OpenID Connect flow, a client must set the value of the `response_type` parameter to `id_token` in the token request. Doing so in Globus Auth causes an unauthorized response type error to be returned. We could not find any other way to get an id token for implicit grant clients.

Result and Suggestion

If id token information would be useful for implicit grant clients, this capability should be implemented. Globus Auth would need to check if the `id_token` value is present and if so also return the appropriate OpenID Connect token as is done with authorization code grants. The assessment team emphasizes that this is not a security concern, but an opportunity to support applications that use implicit grants better.

6.2 Other Globus Auth Components Evaluated

We evaluated other components of the system and did not find any vulnerabilities. Section 6.2 covers these areas. Though it is impossible to guarantee that code is free of vulnerabilities, we have increased confidence in the security of these parts of the code.

6.2.1 PostgreSQL

Summary

The PostgreSQL database is one of the most important resources belonging to Globus Auth. We looked at how data is stored and how the data is accessed.

Description

For data storage, we looked at the PostgreSQL data tables to see which pieces of data are hashed. For data access, we used strings in fields on the Globus Auth website to try to create injections in the SQL commands. We also looked into SQLAlchemy that Globus Auth uses to write its SQL commands in Python functions.

Result

For storage, client secrets and token signatures stored in the database are hashed using SHA-2, the current industry standard. For access, we found that none of our attempts at injections were effective because SQLAlchemy automatically escapes meta-characters, preventing SQL injections. The Globus Auth team should keep SQLAlchemy and their hashing algorithm up to date as technology advances and new vulnerabilities are found in them.

6.2.2 OAuth2 Authorization Code Flow

Summary

We verified that Globus Auth met many of the requirements of the OAuth2 standard, including TLS/SSL, redirect URL handling, scopes, and authorization code and token usage. We also explored many common attacks performed against OAuth2 authorization servers including attacks found in RFC 6819[11].

Description

We analyzed the following issues:

- TLS/SSL be used with all authorization server endpoints. Nginx and uWSGI use SSL to handle HTTPS requests.
- Globus Auth performs all necessary redirect URL validation outlined in the OAuth2 RFC.
- The redirect URLs on a client's whitelist were matched exactly.
- Inputting a bad redirect URL along with other bad parameters would send an error to the redirect. Therefore no open redirect is possible.
- URL fragments could not be included in redirect URLs.
- Other tokens associated with the user were automatically revoked after reusing an authorization code.
- Authorization codes expire.
- Bearer tokens are not sent as URL parameters.

[11] https://tools.ietf.org/html/rfc6819

- Tokens could not be guessed. SHA-2 is used to create token signatures, therefore making them sufficiently hard to guess.
- Clients are authenticated before using a refresh token and refresh tokens are properly validated according to the OAuth2 specification. The client ID and secret are checked to make sure a valid client is issuing the request and that the token belongs to the client. The refresh token is also searched for in the database to find a match. The `validate_scopes` method checks to see if the scopes requested when using a refresh token match the scopes of the original authorization request. If more scopes are requested, an error is returned. If fewer scopes are requested, no error is returned. The RFC is contradictory on this point. At the very end of section 6 of the RFC it states: "If a new refresh token is issued, the refresh token scope MUST be identical to that of the refresh token included by the client in the request." But it also states in section 1.5 that refresh tokens can be used to "obtain additional access tokens with identical or narrower scope". The assessment team wants to point out this inconsistency in the RFC.
- Ensured that token revocation responses were correct.
- Scopes could not be requested by clients without user permission and scopes could not be silently added.
- Attacks from RFC 6819 Section 6 are handled. Attack 4.4.1.12 is not handled; this attack involves sending large quantities of randomly generated authorization codes to client redirect URLs, resulting in a denial of service attack. The attack should be handled by clients as described in the RFC. Implementing rate limits could prevent legitimate users from authenticating and attackers could still get around the limit by sending fewer authorization codes to more clients. Section 6.2.6 addresses the Nginx configuration within Globus Auth to prevent denial of service attacks.

Result

No significant issues were found related to the above mentioned items.

6.2.3 OAuth2 Other Flows

Summary

We looked at less used OAuth2 flows such as client credentials, consent only, and implicit grant types. Dependent tokens and using email as an identity provider were explored as well.

Description

We analyzed the following issues:

- Clients are authenticated before receiving tokens in client credentials grants. The client credentials grant follows the normal authorization code grant flow starting from the token request. All checks normally performed during a token request, including the authorization of clients, are done with a client credentials grant.

- For consent only grants, an open redirect was not possible by inputting redirect URLs not on the client's whitelist. Those redirect URLs that had not been whitelisted for a client returned an error and did not route to the redirect URL.
- Only specified clients are allowed to use implicit grants.
- Refresh tokens cannot be sent in implicit grants.
- Malicious user agents are a threat to implicit grants. An attacker could do much worse than stealing a token in an implicit grant if they were able to take control of the user agent (steal the user's credentials directly for example).
- Implicit grant responses were non cacheable by looking at a response.
- All validations done for access tokens are performed for dependent tokens.
- Globus Auth uses email as an identity provider. This is available when linking identities. A randomized code is sent to the specified email using forgery protection and `sys.maxsize` for the length. Codes are validated and expired and used codes will not work.

Result

No significant issues were found related to the above mentioned items.

6.2.4 Clients

Summary

We explored how clients are registered and edited within Globus Auth.

Description

The `/developers` page of the Globus Auth website is used to create and edit clients. We investigated if an attacker could edit, delete, or create clients on behalf of an admin. We tried revoking tokens and secrets on behalf of a client. We investigated if collisions are possible in client ids.

Public clients must use Proof Key for Code Exchange (PKCE)[12] to authenticate with Globus Auth. PKCE are hashed codes sent with authorization and token requests when clients cannot keep secrets safe. We checked PKCE requirements found in the PKCE RFC to make sure Globus Auth performed this interaction correctly and in a safe manner. We made an authorization request without using PKCE with a public client.

Result

[12] https://tools.ietf.org/html/rfc7636

The `/developers` page requires a cookie to be sent from the browser to check if the user is allowed to remove admins and create, edit, or delete clients. Clients can use this page to revoke secrets and tokens that they have requested so they cannot be used anymore. Secret revocation requires the secret itself and token revocation requires both the token and the client's secret. These would need to be stolen by an attacker. Globus Auth creates a version 4 UUID which has around 122 randomly generated bits for each client. Globus returns a failed request on the improbable case of a duplicate id. Collisions are extremely unlikely to occur and that the times that they do occur in practice are because of bad seeds (poor implementations).

An error is returned if a public client does not use PKCE. Globus Auth verifies the code challenge length is between 43 and 128 characters and checks that the code verifier matches the code challenge. PKCE code challenges are associated with a specific authorization code.

6.2.5 Other Globus Auth Web Pages

Summary

Globus Auth implements many web pages that are outside of the OAuth2 specification scope.

Description

Introspection allows clients and resource servers to send tokens to Globus Auth in exchange for information about how the token was issued. We tried using tokens issued to other resource servers and revoked tokens.

We confirmed that an attacker cannot log a user out. We checked to see if any steps were skipped in the authorization flow after logging out.

Linking accounts allows users to use one login to access resources from multiple accounts. We made sure it was not possible to link or unlink an account of a user by an attacker.

We also checked what resources were retrieved when calling the `/sessions`, `/health`, `/identities`, `/consents`, `/whoami`, `/authcode`, `/userinfo`, `/identity_providers`, and `/rescind_named_grant` pages and saw if these resources could be stolen by an attacker.

Result

Using revoked tokens and tokens issued to other resource servers returned the correct message "`active: false`" during introspection to show the token was not active.

Logging out requires the user’s browser to present a cookie an attacker would need to steal. After going through the authorization process once and logging out, the user is not asked to approve previously approved consents during a second authorization request. This was not deemed a threat.

Linking and unlinking accounts requires the user’s cookie to be presented by the browser so all attacks here are not viable.

The `/consents` page displays what consents (scopes) the current user has approved and when they were granted. Users can also rescind consents by using this page. The browser again presents a cookie to display and rescind consents.

`/userinfo` is an endpoint described by the OpenID Connect specification. An attacker would need a token in order to use the endpoint.

All other pages display basic information and require a cookie from the browser to do so.

6.2.6 Miscellaneous Exploration

Summary

This section describes exploration we performed that does not fit in one of the above sections. These explorations include looking more closely at dependent libraries within Globus Auth and common functions in Python that could be problematic.

Description

We used the Common Vulnerabilities and Exposures website[13] to see if dependencies have dangerous vulnerabilities. We found version numbers of dependent libraries that Globus Auth uses and checked if there were newer versions with fewer vulnerabilities for PostgreSQL, Nginx, Pyramid, Jinja2, Mako, uWSGI and SQLAlchemy.

We looked more deeply at how Nginx is configured within Globus Auth to prevent denial of service attacks. We tested this configuration by creating idle connections and many requests from a single client in rapid succession.

The Python language itself can create vulnerabilities if certain functions are used unsafely. We checked to see if the Globus Auth repository made safe use of functions like `os.system`, `os.popen`, `exec`, `eval`, `execfile`, `input`, or `compile`.

[13] https://www.cvedetails.com/

Result

All checked Globus Auth dependent libraries had little to no vulnerabilities and were versions still being upkept by developers at the time of this report.

Globus Auth sets a connection timeout, rate limits on connections, and limits on requests through one connection in Nginx's config file. We successfully tested these were working with the mentioned tests.

The only vulnerable Python function present in the repository was `compile`. Globus Auth never passes strings as parameters created by the user which is safe. Instead they are static strings and regular expressions inside the repository.

Appendices

Appendix A: High Level Communication Flow Diagram Notes

Notes

- This flow is based on the Python test script given to us by Globus Auth.
- Uncircled numbers are used to number steps in flow. See Figure 6 for a different view of this.
- Circled numbers refer to what operation and data are being passed. See below for key.
- Letters correspond to letters in Figure 1 in the OAuth 2.0 RFC, which outline the basic OAuth flow.

Data Flow

① Client id, redirect URI, requested scopes, client state, response type (which is authorization code in this case)
② Client id, idp identity, callback uri (localhost/p/authenticate/callback), redirect URI (which now has the current path and query embedded in it for after authentication)
③ All items in 2, Globus Auth's client id, Globus Auth's redirect URI, Globus Auth's requested scopes, response type
④ All items in 2, requested scope, session state, and IDP authorization code
⑤ Token URI, IDP authorization code, Globus Auth's client id, Globus Auth's secret
⑥ IDP token
⑦ Same as 3 with the addition of action (indicates that the user approved the scopes)
⑧ Authorization code and client state
⑨ Authorization code, redirect URI, grant type, client id, and client secret
⑩ Access token and refresh token (if requested)
⑪ Access token
⑫ Access token, session info, client id, client secret
⑬ Access token, active (whether the token is valid), token scopes, client id of token, username, name, email, token expiration, token creation time, token issuer

Appendix B: Timeline Diagrams

We created timeline diagrams representing a normal flow through Globus Auth using an authorization code grant type. Figure 6 is based off of the Python script given to us by the Globus Auth team: `/globus-auth/repo/doc/globus_auth_app_and_services_docker.py`. Figure 7 is similar to Figure 6 except that it follows what would occur in a more realistic environment. Instead of copying authorization codes from a browser and pasting them into a terminal as in the Python script, users and clients communicate through a user agent. Diagrams contain notes, basic definitions, and a list of data passed in every step in order to help with understanding.

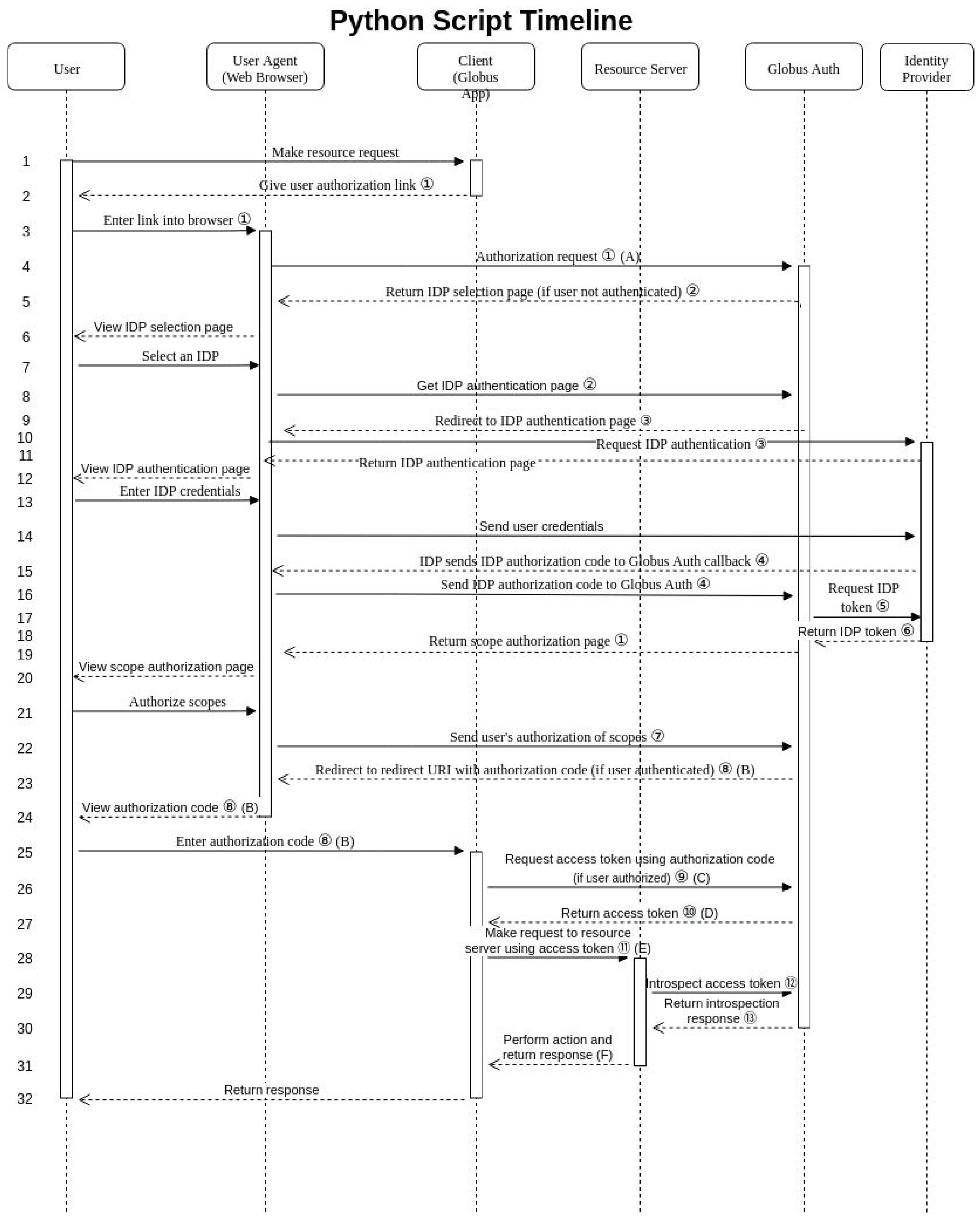

Python Script Timeline
User
User Agent (Web Browser)
Client (Globus App)
Resource Server
Globus Auth
Identity Provider
1 Make resource request
2 Give user authorization link ①
3 Enter link into browser ①
4 Authorization request ① (A)
5 Return IDP selection page (if user not authenticated) ②
6 View IDP selection page
7 Select an IDP
8 Get IDP authentication page ②
9 Redirect to IDP authentication page ③
10 Request IDP authentication ③
11 Return IDP authentication page
12 View IDP authentication page
13 Enter IDP credentials
14 Send user credentials
15 IDP sends IDP authorization code to Globus Auth callback ④
16 Send IDP authorization code to Globus Auth ④
17 Request IDP token ⑤
18 Return IDP token ⑥
19 Return scope authorization page ①
20 View scope authorization page
21 Authorize scopes
22 Send user's authorization of scopes ⑦
23 Redirect to redirect URI with authorization code (if user authenticated) ⑧ (B)
24 View authorization code ⑧ (B)
25 Enter authorization code ⑧ (B)
26 Request access token using authorization code (if user authorized) ⑨ (C)
27 Return access token ⑩ (D)
28 Make request to resource server using access token ⑪ (E)
29 Introspect access token ⑫
30 Return introspection response ⑬
31 Perform action and return response (F)
32 Return response

Figure 6. Python Script Timeline Diagram.

Notes

- This flow follows directly from the Python test script given to us by Globus Auth.
- Uncircled numbers are used to number steps in flow. See Figure 6 for a different view of this.
- Circled numbers refer to what operation and data are being passed. See below for key.
- Letters correspond to letters in Figure 1 in the OAuth 2.0 RFC, which outline the basic OAuth flow.

Definitions

Access Token: Credentials used to access protected resources.
Authorization Code: Credential representing identity provider's authorization used by client to obtain access token.
Client: Application making resource requests with authorization from the identity provider. Set up and run within the Python script `/globus-auth/repo/doc/globus_auth_app_and_services_docker.py` in this case.
Client ID: Unique identifier given to each client by Globus.
Client State: Used to maintain state of client on callback.
Grant Type: Type of authorization grant client uses in exchange for a token (authorization code in this case)
Identity Provider: Entity capable of granting access to a protected resource. Google is used here in the test script.
Introspection: Method for a resource server to query Globus to determine whether an access token is active as well as determine meta-information about the token if needed.
Redirect URI: URI the user agent is redirected to at the end of the authorization process. It is sent to Globus during the authorization request. The authorization code is appended to the end of the URI for redirection.
Resource Server: Server hosting protected resources, capable of accepting and responding to protected resource requests using access tokens. Set up and run within the Python script in this case.
Response Type: Type of authorization grant to be returned.
Scope: Specification for the access request. What data the user wants to access with the access request.
User Agent: Typically a web browser. Handles interaction between user, Globus, and identity provider in this test script. Normally interaction between the client and user would also first pass through the user agent.

Data Flow

① Client id, redirect URI, requested scopes, client state, response type (which is authorization code in this case)
② Client id, idp identity, callback uri (localhost/p/authenticate/callback), redirect URI (which now has the current path and query embedded in it for after authentication)
③ All items in 2, Globus Auth's client id, Globus Auth's redirect URI, Globus Auth's requested scopes, response type
④ All items in 2, requested scope, session state, and IDP authorization code
⑤ Token URI, IDP authorization code, Globus Auth's client id, Globus Auth's secret
⑥ IDP token
⑦ Same as 3 with the addition of action (indicates that the user approved the scopes)
⑧ Authorization code and client state
⑨ Authorization code, redirect URI, grant type, client id, and client secret
⑩ Access token and refresh token (if requested)
⑪ Access token
⑫ Access token, session info, client id, client secret
⑬ Access token, active (whether the token is valid), token scopes, client id of token, username, name, email, token expiration, token creation time, token issuer

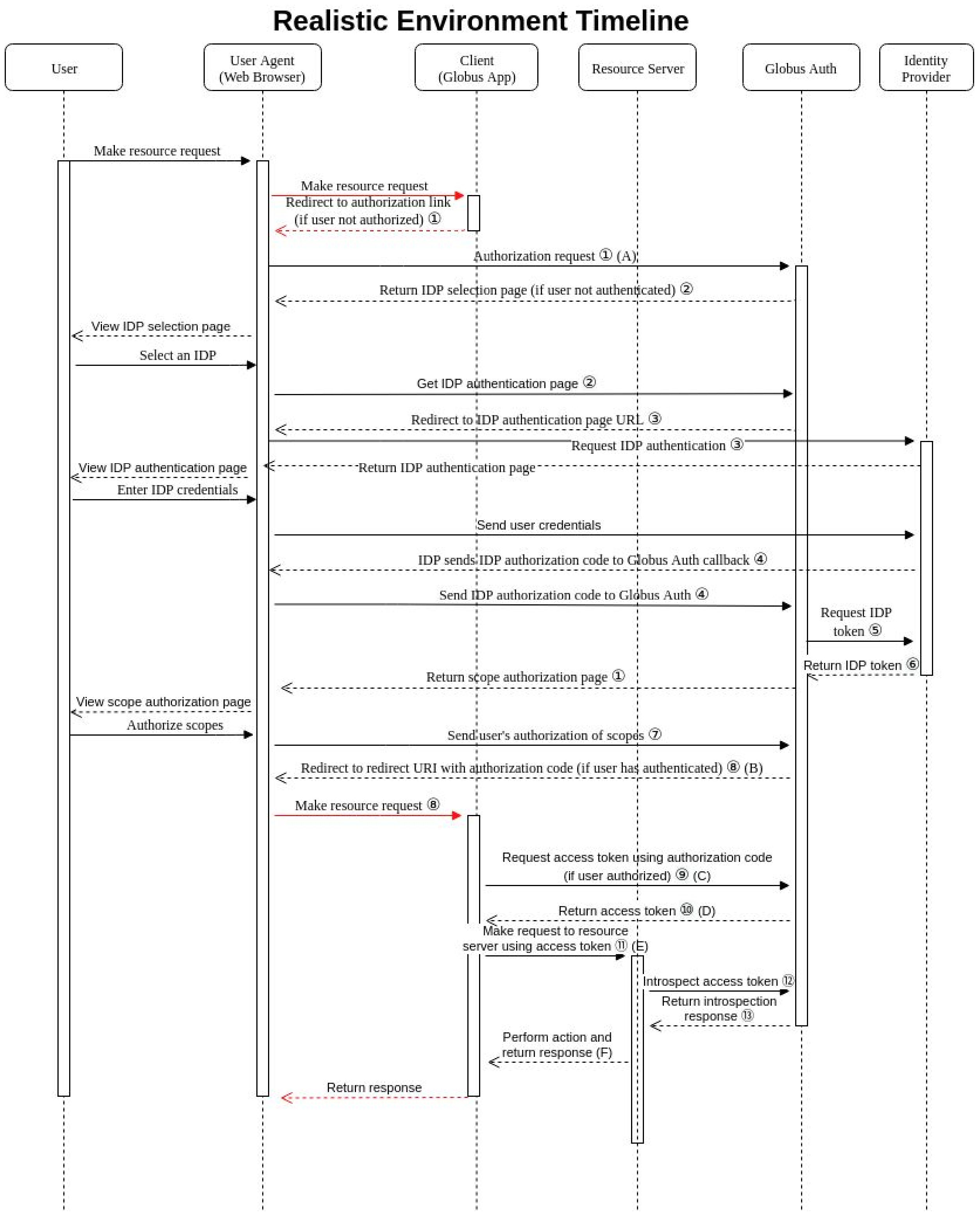

Realistic Environment Timeline
User
User Agent (Web Browser)
Client (Globus App)
Resource Server
Globus Auth
Identity Provider
Make resource request
Make resource request
Redirect to authorization link (if user not authorized) ①
Authorization request ① (A)
Return IDP selection page (if user not authenticated) ②
View IDP selection page
Select an IDP
Get IDP authentication page ②
Redirect to IDP authentication page URL ③
Request IDP authentication ③
View IDP authentication page
Return IDP authentication page
Enter IDP credentials
Send user credentials
IDP sends IDP authorization code to Globus Auth callback ④
Send IDP authorization code to Globus Auth ④
Request IDP token ⑤
Return IDP token ⑥
Return scope authorization page ①
View scope authorization page
Authorize scopes
Send user's authorization of scopes ⑦
Redirect to redirect URI with authorization code (if user has authenticated) ⑧ (B)
Make resource request ⑧
Request access token using authorization code (if user authorized) ⑨ (C)
Return access token ⑩ (D)
Make request to resource server using access token ⑪ (E)
Introspect access token ⑫
Return introspection response ⑬
Perform action and return response (F)
Return response

Figure 7. Realistic Timeline Diagram.

Notes

- Red arrows represent differences that would appear in a realistic environment that do not occur in the Python test script given to us.
- Circled numbers refer to what operation and data are being passed. See below for key.
- Letters correspond to letters in Figure 1 in the OAuth 2.0 RFC, which outline the basic OAuth flow.

Definitions

Access Token: Credentials used to access protected resources.
Authorization Code: Credential representing identity provider's authorization of the user used by client to obtain access token.
Client: Application making resource requests with authorization of the user from the identity provider.
Client ID: Unique identifier given to each client by Globus.
Client State: Used to maintain state of client on callback.
Grant Type: Type of authorization grant client uses in exchange for a token (authorization code in this case)
Identity Provider: Entity capable of granting access to a protected resource. Google is used here in the test script.
Introspection: Method for a resource server to query Globus to determine whether an access token is active as well as determine meta-information about the token if needed.
Redirect URI: URI the user agent is redirected to at the end of the authorization process. It is sent to Globus during the authorization request. The authorization code is appended to the end of the URI for redirection.
Resource Server: Server hosting protected resources, capable of accepting and responding to protected resource requests using access tokens. Set up and run within the Python script in this case.
Response Type: Type of authorization grant to be returned
Scope: Specification for the access request. What data the user wants to access with the access request.
User Agent: Typically a web browser. Handles interaction between user, Globus, and identity provider in this test script. Normally interaction between the client and user would also first pass through the user agent.

Data Flow

① Client id, redirect URI, requested scopes, client state, response type (which is authorization code in this case)

② Client id, idp identity, callback uri (localhost/p/authenticate/callback), redirect URI (which now has the current path and query embedded in it for after authentication)
③ All items in 2, Globus Auth's client id, Globus Auth's redirect URI, Globus Auth's requested scopes, response type
④ All items in 2, requested scope, session state, and IDP authorization code
⑤ Token URI, IDP authorization code, Globus Auth's client id, Globus Auth's secret
⑥ IDP token
⑦ Same as 3 with the addition of action (indicates that the user approved the scopes)
⑧ Authorization code and client state
⑨ Authorization code, redirect URI, grant type, client id, and client secret
⑩ Access token and refresh token (if requested)
⑪ Access token
⑫ Access token, session info, client id, client secret
⑬ Access token, active (whether the token is valid), token scopes, client id of token, username, name, email, token expiration, token creation time, token issuer