A Comparison of Authentication Protocols for Unified Client Applications

1st Floris Breggeman
Semantics, Cybersecurity & Services
University of Twente
Enschede, Netherlands
f.t.breggeman@student.utwente.nl

2nd Mohammed El-hajj
Semantics, Cybersecurity & Services
University of Twente
Enschede, Netherlands
m.elhajj@utwente.nl

3rd Florian Hahn
Semantics, Cybersecurity & Services
University of Twente
Enschede, Netherlands
f.w.hahn@utwente.nl

2023 International Symposium on Networks, Computers and Communications (ISNCC) | 979-8-3503-3559-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISNCC58260.2023.10323861

***Abstract*—OAuth, LDAP, forward authentication, and proxy authentication are protocols that allow a service to use a third party for user authentication. We compare these protocols on a variety of aspects, concluding that OAuth offers the greatest end-user convenience, forward and proxy authentication are the easiest to implement for the client application, and LDAP puts restrictions on how to identify the end-users. We then demonstrate a single data structure that can be used to store a client application in all four protocols, overcoming their disparate ways of functioning.**

1. Introduction

Services that require users to be authenticated often rely on third parties for user authentication; this allows organizations to maintain only a single directory with all of their users, and users to authenticate to multiple services with a single set of credentials. There are several protocols that allow for communication with an authentication service. This paper compares four authentication protocols: OAuth, LDAP, forward authentication, and proxy authentication. It then demonstrates a single data structure that can store a client application in any of the four protocols; this data structure is used by DAS[2, 1], an authentication service developed specifically for domestic self-hosting, i.e. the practice of end-users hosting software from their own homes for their own household. The use of a single datastructure has simplified the implementation of DAS, and allowed for easily creating a unified user interface for administering interaractions using all 4 protocols.

The following paragraph may be used to disambiguate the usage of the words printed in bold in this paper. In the **authentication** process an **end user** wants to prove their identity to a **client application**. The **client application** has outsourced this to an **authentication provider**, using an **authentication protocol** to communicate with it. The **authentication provider** then uses an **authentication scheme** to identify the **end user**, likely involving **credentials**, and uses the **authentication protocol** to communicate the identity of the **end user** to the **client application**.

We have found one paper that has a significant relationship to this topic. Mengyi et al.[7] propose an augmentation of Shibboleth, a SAML authentication provider, adding support for OpenID Connect and theoretical support for any other authentication protocol. It does so not by changing the authentication provider itself, but by creating a separate component to interoperate between SAML and other protocols.

2. Protocols

This section will describe and compare the different authentication protocols, on both their theoretical specification and their practical implementations. In order to do so, it will explain the registration process, i.e. what information the client application and the authentication provider need to know about each other, followed by explaining a basic authentication flow, and conclude with any other relevant information required for understanding the protocol.

2.1. Oauth

This section will describe OAuth 2.0; OAuth 1.0 can be considered deprecated for all practical purposes. For more information on OAuth 2.0, including implementation details, the author recommends [11].

OAuth 2.0 has no single practical specification, but rather a collection of separate RFCs[4, 6, 8, 5] which don't necessarily cover all implementation details. Regardless, it seems almost all implementations have converged to a compatible protocol.

It should be noted that while OAuth can be used as an authentication protocol, it is not strictly intended as such. Instead, it was originally intended as an *authorization* protocol, i.e. a protocol that grants access rights to client applications, instead of an authentication protocol, which proves the identity of the end user to the client application. This section will also explain the way in which a dedicated authentication provider and a client application that only requires authentication interact.

2.1.1. Client Application Registration. For a client application to connect to the OAuth service, it must first be registered with the OAuth provider. This will generate a Client ID, and optionally a Client Secret. OAuth authentication involves redirecting the browser from the client application to the authentication provider and back; the URIs to which the user should be redirected back should also be registered with the authentication provider.

2.1.2. Basic Authentication Flow. The process of authentication is illustrated in figure 1 First, the end user accesses the login page from the client application. If the end user is not authenticated, it will redirect to the authentication provider, using a URI that contains several GET parameters, most notably the application's Client ID.

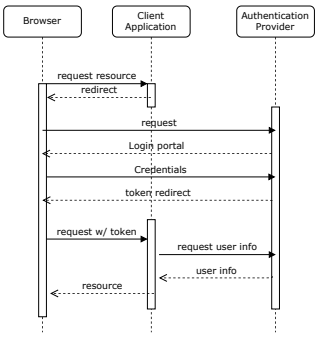

Figure 1. A schematic overview of the basic OAuth flow

The authentication provider then verifies the identity of the end user. Afterward, the authentication provider redirects the user back to the redirect URI that was registered by the application, appending a code to this URI as a GET parameter.

To resolve this code, the client application must exchange the code for an access token with the authentication provider. The request to do this will contain the code, the client id, and the client secret. It will return an access token, which must be included in any requests to the authentication provider's main API. It should be noted that the code can only be used in a short window, and only once.

To get information on the user, the client application can query the authentication provider, which should provide an API endpoint that returns some data on the user. Querying this endpoint requires an access token. Section 2.1.3 defines the format of this endpoint.

2.1.3. OpenID Connect. OpenID Connect is an authentication protocol that builds upon OAuth[13]. It offers several ways to utilize OAuth for authentication services, introduces extensions to the OAuth protocol, and incorporates a standardized discovery protocol. Notably, OpenID Connect defines two methods for obtaining user information via OAuth. The first method closely resembles the standard OAuth flow, and OpenID Connect prescribes a standardized `userinfo` endpoint for client applications to fetch user information. To enhance efficiency, OpenID Connect introduces an extension to the OAuth protocol called an ID token. This involves the authentication server generating an asymmetric key pair and publishing the public key. Instead of sending an access token in the usual OAuth flow, an OpenID Connect server also sends an ID token, containing user information and signed with the generated private key. This allows the client application to verify the data's authenticity without needing to directly contact the authentication provider. In standard OAuth flows, securing the callback URI (the URI to which the user is redirected after authentication) is essential for security, as it prevents malicious agents from obtaining an access token without being able to spoof the URI and its associated certificate. In the context of OpenID Connect, registering the callback URI becomes even more crucial. This is because the client application can acquire information from the ID Token, which is obtained without a client secret. Attackers could use this information to spoof the user login process, potentially leading to phishing attacks on end-users who believe they are logged into the legitimate application.

2.2. LDAP

The Lightweight Directory Access Protocol (LDAP) is a directory database that is commonly used to store user information. It is designed merely as a database, but a specific procedure allows client applications to verify user credentials against the database. In this procedure, unlike in OAuth, the client application is still responsible for gathering the credentials. It can then use these credentials to verify user authentication and gather user information.

2.2.1. Database Structure. Within a directory database, a hierarchical structure resembling a tree is employed, where each individual entry encompasses its own data while simultaneously serving as a parent entity to other entries [15]. These entries store information in various fields, with each field possessing a unique name and accommodating a specific type of data. Additionally, entries can be subject to constraints imposed by a class, establishing a template dictating the mandatory and optional fields that an entry must possess. Moreover, entries can be readily identified through their Distinguishing Name (DN), which comprises a collection of attributes enabling the differentiation of entries within the offspring of their parent, resembling the concept of a file path. This organization and identification structure contributes to the efficient management and retrieval of data within the directory database.

In order to use LDAP for authentication, there must be some sort of entry class that represents a user for authentication[9]. This object class must have at least two fields: firstly, some sort of unique identifier that is memorable to humans, such as a username or an email address, so that the client application can connect the entry to the user credentials. Secondly, a password field, so that the LDAP server can authenticate for this entity. It is recommended that this password field is a hash[14].

2.2.2. Client Application Registration and Configuration. It is possible to connect to an LDAP server with or without client application credentials, and an LDAP server could be set up such that unauthenticated client applications can authenticate users. For security reasons, however, it is advisable that a new set of client application credentials is generated for each client application, and that only authenticated client applications can access any user data.

The client application must be configured in order to be able to interact with the specific LDAP data formats used by the authentication provider. Specifically, it needs to know how to identify and search for entries that represent users, and which fields of these entries contain what information about the end user.

2.2.3. Authentication Flow. The authentication process follows a series of steps, as illustrated in figure 2. After the client application has gathered the user credentials, it establishes a connection with the authentication provider, using its own client id and secret if applicable. Subsequently, it performs a search for user entries that match

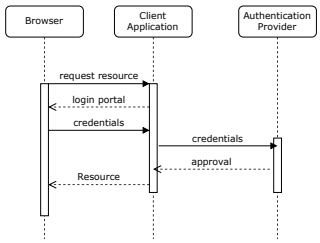

Figure 2. A schematic overview of the events in LDAP authentication

the identifying information provided in the credentials, in order to test whether the given username matches an existing user, and in order to gather any information on the user not provided by the credentials. If matching entries are found, the client application utilises the DN (Distinguished Name) of the corresponding entry, along with the user-provided password, to authenticate with the authentication provider. Successful authentication indicates that the end-user has been identified and authenticated.

2.3. Forward Authentication

The forward authentication protocol relies on a reverse proxy to verify with the authentication provider that a request has been authenticated before forwarding the request to the client application. When proxying requests to the client application, the reverse proxy sets headers that the client application can use to identify the user.

Note that all the client application needs to do is read the user information from the HTTP headers it receives. Even if the client application does not support this authentication protocol, this would only mean it is unable to identify the user; the user would still have to be authenticated before they are able to make any requests.

2.3.1. Client Application Registration and Configuration. The reverse proxy plays a crucial role in establishing a robust authentication protocol (see section 2.3.3). Its configuration involves listening to the desired publicly accessible connection and efficiently forwarding incoming requests. One key aspect is the reverse proxy's ability to communicate effectively with the authentication provider, enabling seamless integration for authentication processes. A significant concern in securing this authentication protocol is to isolate the client application within the reverse proxy, preventing direct access to the client application from external sources in order to mitigate impersonation attacks that can arise from the manipulation of headers. When the client application supports the protocol and can extract user information from incoming HTTP headers, configuration is required to establish the mapping between headers and the corresponding user data. In scenarios where the authentication provider serves multiple client applications utilising the protocol, and each application has its own set of authorised users, it is essential to configure the authentication provider with the `Host` header value (i.e. domain) of each client application. This ensures that the authentication process remains tailored to each client application. By configuring the reverse proxy, client application, and authentication provider effectively, a secure and streamlined authentication protocol can be established. This approach safeguards the system against unauthorised access and potential impersonation.

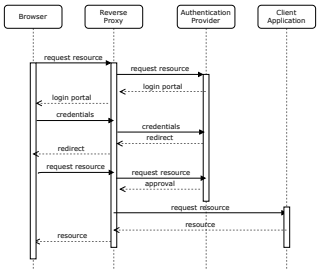

Figure 3. The sequence of events in forward authentication

2.3.2. Basic Authentication Flow. The process of forward authentication is depicted in figure 3. When a user's browser intends to send a request to the client application, the reverse proxy is the first to receive the request and forwards it to the authentication provider. If the user has not yet been authenticated, the authentication provider notifies the reverse proxy by returning an *unauthorized* status code. Instead of proxying the request to the client application, the reverse proxy responds by presenting the login page to the user's browser or redirecting to it. The user then authenticates themselves with the authentication provider. Once the authentication process is completed, the authentication provider instructs the user's browser to resend the initial request. With a successfully authenticated user, the authentication provider returns an *ok* status code along with some headers containing user-related information. The reverse proxy now proceeds to proxy the original request, including the additional headers received from the authentication provider, to the client application.

2.3.3. Integration with Reverse-Proxies. To implement forward authentication, a reverse proxy must be integrated and configured in a precise manner. This involves configuring the reverse proxy to initially forward all requests to the authentication provider. Additionally, it needs to be set up to redirect users to the appropriate login page in case of failed authentication and to correctly pass the relevant headers from the authentication provider to the client application. It is important to note that not all reverse proxies can be configured to support these requirements. A prominent example of a reverse proxy that lacks support for forwarding incoming requests to an authentication provider is Apache [3]. An alternative solution that is compatible with Apache is discussed in section 2.4.

2.4. Proxy Authentication

Proxy authentication offers similar functionality to forward authentication but does not require support from a pre-existing reverse proxy. It does so by turning the authentication provider into a reverse proxy which only proxies authenticated requests. We have not been able to find another implementation of this protocol, and suspect that the DAS is the first.

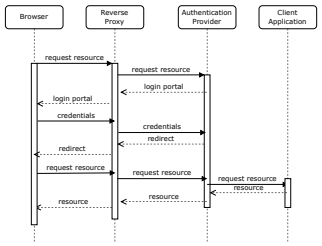

Figure 4. The sequence for proxy authentication

2.4.1. Client Registration. The authentication provider needs to know where to proxy authenticated requests to. If the authentication provider is providing proxy authentication services for multiple client applications, it needs some way to connect the incoming request to a particular client application.

2.4.2. Authentication Flow. The authentication process is depicted in Figure 4. When a request is received by the ingress reverse proxy (e.g. Apache or Nginx) it does not directly forward the request to the client application. Instead, the reverse-proxy routes the request to a specific socket on which the authentication provider is listening. The authentication provider then evaluates whether the request originates from an authenticated user or not, using the `Host` header to distinguish the client application to which the request was originally addressed. Following the principles introduced in section 2.3.2, if the request is unauthenticated, the authentication provider provides a means for the user to authenticate themselves. Once the user completes the authentication process, the authentication provider redirects them back to their original request. Only when the authentication provider receives an authenticated request, it proceeds to forward this request to the client application, ensuring that access is granted solely to authenticated users. This process ensures the secure and controlled flow of requests within the authentication protocol.

2.5. Protocol Comparison

Firstly, the features will be described. Comparison based on these features can be found in table 1:

- **Interface:** Which party has control over the interface where the user logs in? This feature decides how much freedom the authentication provider has in the used authentication scheme.
- **Communication:** How the client application and the authentication provider communicate. This functionality elucidates the significant architectural distinctions among the protocols, shedding light on their respective strengths and limitations. For instance, protocols that rely on browser-based communication are restricted to scenarios where a browser is available. This highlights the contextual constraints and considerations associated with such protocols.
- **Data format:** Which data format data is primarily exchanged in. This feature highlights the time in which a protocol was designed, as well as its intended audience.
- **Registration:** The requirement for the authentication provider to have prior knowledge of a specific client application before it can be utilised. This aspect reflects the level of trust that must be established between the authentication provider and the client application for the protocol to operate effectively. In protocols where client application registration is necessary, the authentication provider only serves clients that it already trusts. This feature also becomes significant when the authentication provider needs to adapt its behaviour based on the specific client application being authenticated.
- **Awareness:** The need for the client application to be aware of the authentication provider it is using. This feature emphasises the distinct advantage of the reverse proxy setup, as it eliminates the requirement for the client application to have awareness of the authentication provider. This capability enables the protocol to offer authentication services to client applications that may not inherently support a centralised authentication service.
- **Range:** The *furthest* type of connectivity that can be used to communicate between the authentication provider and the client application. This connectivity can range from the two applications being located on the same machine to being separated by an internet connection. This aspect highlights the limitations of protocols that do not necessitate registration, as they may not provide robust support for scenarios where the authentication provider and client application are not in close proximity.

2.6. Authentication Schemes

This section will briefly discuss what authentication schemes are compatible with all protocols. For OAuth and Reverse Proxy, the authentication provider needs to provide an HTTP server that authenticates the user's browser. This means that any authentication scheme that works in a web browser can be used.

An LDAP client, however, will only make two requests to the authentication provider. This flow itself was only intended for authentication via username/password, but this does not mean that this is the only way in which it could be used. The fact that the LDAP client asks the authentication provider to verify the password instead of verifying the password itself, allows the authentication provider to perform steps that are not quite the same as verifying a password, and the string in the password field does not have to be a password.

It should be noted that LDAP authentication is compatible with a wide variety of client applications, not limited to web browsers. As such, the deployment of any authentication schemes that require modification of the client application, such as automatic insertion of cryptographic keys, would feasibly only be compatible with the select subset of client applications that interact with web browsers and a very limited set of other popular client applications. Some two-factor authentication systems could still be used over LDAP, either by appending some data to the password or by performing an out-of-band verification while the authentication provider appears

TABLE 1. A COMPARISON OF THE DIFFERENT COMMUNICATION PROTOCOLS

PROTOCOL	Interface	Communication	Data Format	Registration	Awareness	Range
OAuth	Provider	Browser, optionally HTTP	JSON	Mandatory	Yes	Internet
LDAP	Client	TCP	Custom Binary	Recommended	Yes	Internet
Forward Auth	Provider	HTTP Proxy	HTTP Headers	Optional	No	Local Network
Proxy Auth	Provider	HTTP Proxy	HTTP Headers	Mandatory	No	Local Network

to be verifying the password. The only common protocols that can be used without requiring the end-user to purchase additional hardware or modify the client software are single-use authentication codes, or Time-based One Time Passwords (TOTP). With single-use authentication codes, the user is given a number of pre-generated codes that they should store, using each code only once as a two-factor authentication method. With TOTP, the user uses a smartphone app to generate a six-digit code used for authentication, which is based on a key generated by the authentication provider and the current time; the code is valid for 30 seconds[10]. Since TOTP offers more security and more usability[12], the method of choice for a 2-factor authentication system over LDAP should be to have the user prepend or append a 2-factor authentication code to the password entered in the password field.

3. Datastructure

TABLE 2. THE FIELDS OF THE UNIFIED DATA STRUCTURE, AND FOR WHICH PROTOCOLS THEY ARE USED

Field	Type	OAuth	LDAP	Forward	Proxy
Type	String	C	C	C	C
Id	Binary	✓	✓	✗	✗
Secret	String	✓	✓	✗	✗
Name	String	C	C	C	C
URL	String	C	C	✓	✓
Destination	String	✗	✗	✗	✓

This paper defines a data structure that can be used to store a client application in all four of the protocols. This data structure has the form of a singular database table, with not all columns being used for all protocols. Details on the schema can be seen in table 2. In this table, a checkmark (✓) represents the field is used, a cross (✗) represents it is not, and a C (C) represents the field can be used for display purposes (mainly in the administrative interface), but is not required for the functioning of the protocol.

Which protocol the client application was configured for is indicated by the type field. This field is only used for display purposes; any registered client application could use any protocol so long as all the required fields for that protocol are present.

There are four fields that are always filled in the client id, the client secret, the client name, and the client URL. The Client Id is a randomly generated binary value that functions as the primary key of the table and is as such useful for the authentication provider itself. A hexadecimal representation of the Client ID is used by OAuth and LDAP. The client secret is a randomly generated string used by OAuth and LDAP. The name is used solely for display purposes. The URL is used for display purposes by all types of client applications but is also used as a means of identification for client applications that use forward or proxy authentication. Additionally, there is the destination field, which is used solely for proxy authentication, as the location for proxy requests.

There is additionally the callback URIs table, which is used solely to store callback URIs for OAuth client applications, in order to enhance their security.

Note that for all protocols, the data structure only needs to be consulted once an authentication request arrives; as the data structure is stored in a database and therefore safe for use in concurrent environments, this means all modification of this data structure can be done at runtime without compromising the integrity of the data.

4. Implementation

This section will describe some notable details about how DAS uses the data structure as defined in section 3 to provide authentication services in the four protocols.

4.1. OAuth

The fields in the data structure can be trivially mapped to the properties of an OAuth client application, allowing both OAuth 2.0 and OpenId Connect to be implemented according to specification.

In DAS, all callback URIs have to be registered. To make it slightly easier for system administrators to register the callback URI, the error message that is displayed to the user when the callback URI is invalid contains instructions on how to register the callback URI.

4.2. LDAP

The implementation of the LDAP component is quite different from a standard LDAP server: while user data can be accessed via the LDAP wire protocol, it can not be modified via this protocol in any way, and the data does not actually reside in a directory structure. This means there are impactful decisions about how to translate the single database table that stores user data to the directory structure expected by the client application. DAS ignores all but the least significant parts of location (i.e. it uses only the first part of the DN). This applies to everything in the client request, including the name the client uses to authenticate itself (the Bind DN). The advantage of this approach is greater support for misconfigured and ill-behaving clients, as any valid username or client id will resolve regardless of which directory is searched. The Bind operation applies to both clients and users; DAS distinguishes the two by whether an `id` is provided (for clients) or a `username`. The directory that users appear to be in for any results do not depend on the directory used by the input, but on a value defined in the configuration file. Any filter in a search query that refers to an attribute of the entity that does not exist is ignored. This is done because many implementations filter not just on

the username of the user, but also on the `objectClass` attribute, which would normally distinguish user entities from other types of entities in the system. Since DAS will only return user entities under any circumstance, this attribute can be simply ignored. This does not apply to the `present` filter type, which returns true if and only if the attribute is part of the users schema.

4.3. Forward Authentication

The DAS endpoint that handles the authentication check is extremely simple: it simply checks if the user has a session, and if so, returns an OK response with some headers that contain values from this session. As this endpoint is called on every request to the client application, speed is important; in order to account for this, user data is cached in the session, meaning old data might persist upon user update.

Any request that enters the forward authentication endpoint at the authentication provider is a forwarded request that was originally made to the client application, meaning cookies originally set with the DAS interface will likely not propagate. As such, any unauthenticated request will be redirected to the special DAS endpoint under the DAS domain, which will be referred to as the session creation endpoint. This endpoint will check if a session has already been established, and refer to the login portal if not; after the user has logged in, the login portal will redirect back to the session creation endpoint. The session creation endpoint will then reuse the authorization code feature from OAuth, and create an authentication code that contains the necessary user info. It will then redirect the user back to the original request they made to the client application, but with the authorization code as a query parameter. The reverse-proxy endpoint will then redeem the authorization code and establish a session under the domain of the client application. This process is essentially a fully internal implementation of the OAuth implicit flow.

4.4. Proxy Authentication

Proxy authentication establishes a session with the user in the same way as forward authentication and reuses the session creation endpoint.

The proxy authentication code needs to know where to forward the request. The location must consist of a scheme (either HTTP or HTTPS) and a host, with optionally a port. While the location is stored in the `destination` field of the client application, querying the database for every request would lead to unacceptable latency, and as such an already parsed value cached.

4.5. TOTP

When using TOTP over LDAP, the client application, and by extension the login interface, are not aware that TOTP is being used, and will not communicate this to the user. Instead, the user is expected to remember this by themselves and append the TOTP code to their password (with no separator). To add to the inconvenience for the user, the client application does not display the reason for authentication failure, and as such the user can't be reminded to enter the TOTP code. Furthermore, if the authentication does not succeed, the user is unable to determine whether this is due to a wrong password or a wrong TOTP code. As such, TOTP over LDAP can be enabled by each user separately from TOTP and is disabled by default.

5. Discussion

This subsection will compare the four supported protocols as supported by DAS, discussing aspects that concern the practical deployment and have not yet been discussed in section 2.5.

The first of these aspects is the complexity of configuring a client application with the correct credentials. An OAuth application, in the best-case scenario, only needs to be informed of three values: the client id, the client secret, and the automatic discovery URL. However, not all applications use the automatic discovery URL, and in this case, the application needs to additionally be configured with the authorization, token, and user info URLs, as well as the scope, bringing the total amount of required configuration settings to 6. In either case, DAS will have to be additionally configured with the callback URL. An LDAP client needs to be configured with the URL of the LDAP server, the bind DN and password, the default search DN, object class, and filter, which is also 6 values; however, it should be noted that DAS ignores the search DN and object class. Forward and proxy authentication are considerably more complex to configure, as they require the configuration of the client application and the reverse proxy. In both cases, the client application needs to be configured with the name of the headers in which the user information will be transmitted; this usually amounts to two values, one for the username and one for the email. For forward authentication, the configuration of the reverse-proxy is by far the most complicated configuration discussed yet, and will usually require combining the configuration file suggested by the developer of the client application for the reverse-proxy with the one suggested by the authentication provider. For proxy authentication, this configuration can be simpler if the client application can be accessed entirely via an HTTP socket, and does not require further reverse-proxy configuration. If this is not possible, as is the case for e.g. PHP and UWSGI applications, a second reverse proxy will need to be configured between the authentication service and the client application, which is obviously the most complex to configure and may incur a significant performance penalty. In conclusion, OAuth is the simplest to configure in the best case, and slightly worse than LDAP in the worst case. Another important aspect is performance, which we define as the amount of time it takes the user to be authenticated to the client application, given that they are already authenticated with the authentication provider. In a domestic self-hosting environment, we can assume that the client application and the authentication provider are running on the same machine, but that any communication with the user must go over a network call, and as such other sources of latency are insignificant. LDAP requires one user request to the client application containing the user credentials

and is thereby the fastest. OAuth, forward authentication, and proxy authentication require two network requests: one towards DAS to generate an authentication code, and one towards the client application to retrieve this code. The OpenID Connect ID Token flow is slightly faster, as there is no further request from the client application to the authentication provider necessary, nor any internal lookups to resolve the tokens. As such, LDAP has the highest performance, followed by OAuth using the OpenID Connect flow, followed by a shared third place between forward authentication, proxy authentication, and base OAuth. Another important performance metric is the performance after the user has been authenticated. LDAP and OAuth don't affect this performance at all, as they are not active after user authentication, but forward and proxy authentication do still require the authentication provider to authenticate every request, necessarily introducing latency. From the above comparisons, it may look like LDAP is the most ideal protocol, as it is the easiest to configure and the most performant. However, it does suffer from a major drawback and inconvenience to the user, as it is the only protocol where the user has to enter their credentials for every client application, even if they are already authenticated with DAS. Furthermore, using TOTP in combination with LDAP leads to even more inconvenience, as described in section 4.5. Instead, OAuth should be considered the most ideal protocol, as it is unique in both allowing DAS to control the login portal, which avoids repeated logins and allows for intuitive TOTP, and not affecting the performance of the client application after the user has been authenticated.

6. Conclusion

We set out to perform a comparative analysis of OAuth, LDAP, forward authentication, and proxy authentication, as well as create a single data structure that can store a client application in any of the protocols.

We conclude that OAuth offers the most technical flexibility, at the cost of the greatest complexity to implement and administer. LDAP offers the greatest runtime speed, and is also the only examined protocol that can authenticate clients that are not web browsers; this comes at the cost of user convenience by not being a single sign-on protocol, as well as limiting the potential authentication schemes. Forward authentication is the easiest to implement for the client application, but is only usable when the client application and authentication provider are on the same network, requires complex configuration by the system administrator in order to be secure, and also adds latency to every request. Proxy authentication is functionally equivalent to forward authentication but is less tightly coupled to the ingress reverse proxy, at the cost of increased latency. This comparison, to the best of our knowledge the first of its kind, can be used to make decisions about and guide further research into authentication protocols.

We also define a single data structure for storing all four of the protocols as used in DAS. The data structure is defined as a database table, of which some fields are not used for some protocols at all, and some are used to indicate information to the system administrator but not by all protocol implementations. To the best of our knowledge, this is the first time such a data structure has been defined.

This paper has only assessed protocols that are commonly used in domestic self-hosting. Future research may use our comparison and its criteria to asses and compare additional authentication protocols, such as SAML, PAM, or future protocols that have yet to be designed.

References

[1] Floris Breggeman. *An Authentication Service for Domestic Self-Hosting*. May 2023. URL: http://essay.utwente.nl/94995/.

[2] Floris Breggeman. *Das*. 2023. URL: https://github.com/florisbreggeman/das (visited on 2023-06-11).

[3] Apache Software Foundation. *Apache HTTP Server Project*. 2022. URL: https://httpd.apache.org/ (visited on 2022-01-13).

[4] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. RFC Editor, Oct. 2012, pp. 1–76. URL: https://datatracker.ietf.org/doc/html/rfc676749.

[5] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. RFC Editor, May 2015, pp. 1–30. URL: https://datatracker.ietf.org/doc/html/rfc7519.

[6] M. Jones and D. Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC 6750. RFC Editor, Oct. 2012, pp. 1–18. URL: https://datatracker.ietf.org/doc/html/rfc6750.

[7] Mengyi Li et al. "A Multi-protocol Authentication Shibboleth Framework and Implementation for Identity Federation". In: *Security and Privacy in Communication Networks*. 2018, pp. 81–101.

[8] T. Lodderstedt, M. McGloin, and P Hunt. *OAuth 2.0 Threat Model and Security Considerations*. RFC 6819. RFC Editor, Jan. 2013, pp. 1–71. URL: https://datatracker.ietf.org/doc/html/rfc6819.

[9] Connect2id Ltd. *LDAP user authentication explained*. 2022. URL: https://connect2id.com/products/ldapauth/auth-explained (visited on 2022-11-18).

[10] D. M'Raihi et al. *TOTP: Time-Based One-Time Password Algorithm*. RFC 6238. RFC Editor, May 2011, pp. 1–16. URL: https://www.rfc-editor.org/rfc/rfc6238.

[11] Aaron Pecki. *OAuth 2.0 Simplified*. Lulu Press Inc, 2018. ISBN: 9781387813650.

[12] Ken Reese et al. "A Usability Study of Five Two-Factor Authentication Methods". In: *SOUPS'19: Proceedings of the Fifteenth USENIX Conference on Usable Privacy and Security*. Ed. by Heather Richter Lipford. Aug. 2019.

[13] N. Sakimura et al. *OpenID Connect Core 1.0 incorporating errata set 1*. 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html (visited on 2022-11-17).

[14] K Zeilinga and OpenLDAP Foundation. *LDAP Authentication Password Schema*. RFC 3112. May 2001, pp. 1–9.

[15] K Zeilinga and OpenLDAP Foundation. *Lightweight Directory Access Protocol (LDAP): Directory Information Models*. RFC 4512. June 2006, pp. 1–52.