Comparative analysis - Web-based Identity Management Systems
1. Introduction

This report presents an overview and comparative analysis of some of the latest and most relevant Web-based Identity Management technologies currently available on the market: OpenID, OAuth, OpenID Connect and UMA.

The paper is divided into two main sections: The first section provides an introduction and overview to each technology and the second part provides a comparison chart of the technologies studied along three main axes: Standard Status, Use Cases and Security Features.
2. Current State of the art

The broad number and variety of authentication and authorization systems existing in the market creates a great confusion, not only among its potential users, but also among software developers at the time of deciding the technology to adopt.

This feeling is emphasized by the fact that indeed, some of the existing systems have very significant differences in terms of their security approach, philosophy and scope.

However, all the solutions presented share a common goal: To secure and facilitate the life of the end user when accessing and managing his/her web resources.

This is done by:

- Minimizing the number of passwords to be remembered by users in order to access their resources in a secured way.
- Allowing for the adoption of Single-Sign on mechanisms.
- Providing access management mechanisms that allow for user-controlled share of resources among different providers.

2.1. OpenID

OpenID [OpenID 1] is an authentication protocol designed for web environment with the spirit of minimizing the creation of multiple identities to log into personal web sites. This is achieved by allowing users to log into a site by making use of the account information created in a different site.

The designers of OpenID wanted to simplify the life for end users while minimizing the risk associated with the fact that users cannot remember enough distinct passwords to use for every different account that they manage on the web.

Another advantage of OpenID is that it simplified the life of web developers by providing them with a simple and external Single Sign-On solution that would take care of user authentication and that, at the same time, would also facilitate the registration of new uses by minimizing their effort in doing so.

OpenID has been adopted by a broad number of web developers and users. On December 2009 there were over one billion OpenID-enabled user accounts and 9 million websites using OpenID [OpenID 2].

Although the OpenID 2.0 specification was published on December 2007, according to [Tapiador 1], by the year 2011, the deployment of this version was still significantly lower than the deployment of versions 1.0 and 1.1 (97.32% even though version 1.0 presented a very fundamental security problem), with respect to version 2.0 with a 43.54% of the existing identifiers (40.86% of the identifiers supported both versions of the protocol).

In this paper we will first present the most up-to-date v.2.0 of the protocol and, from its model, we will then comment on the previous two versions.

However, it is important to remark that currently this protocol is considered to be obsolete.

2.1.1. Protocol overview

Figure 1 depicts an overview of the OpenID 2.0 protocol.
The main flow defined by the OpenID 2.0 protocol presents the following steps:

1. End User (Resource Owner) requests information from Relying Party via the User Agent (Browser).
2. Init request is forwarded by Browser onto Relying Party.
3. Relying Party forwards the white list of OpenID Providers (Authorization Servers) that it accepts to the Browser. The Relying Party could also use a generic approach by just requesting the user for his/her OpenID identifier and make use of a black list of not accepted providers to filter the request.
4. Browser presents the list to the End User and requests for his/her OpenID identifier.
5. User makes a choice of an OpenID Provider and/or introduces his OpenID identifier to login.
6. Browser forwards this information to the Relying Party.
7. Relying Party uses the OpenID identifier to look up for the OP Endpoint (OpenID Provider Endpoint) to initiate the request. This discovery process can be done via the Yadis protocol (41.32% of cases according to [Tapiador 1]), HTML-based discovery (91.53%) or XRI resolution (practically inexistent).
8. The specification recommends that a Relying Party requests for the establishment of an association with the OP Endpoint by requesting a key to exchange information.
9. The OP Endpoint may generate a shared key or a private-public key pair.
10. The OP Endpoint returns the key to the Relying Party. In case of using a shared key, the specification mandates that it should be exchanged via TLS.
11. The Relying Party sends an indirect request for User Authentication to the OP Endpoint. Version 1.x only allows for GET HTTP redirection usually lim-
The Alexandra Institute • Aabogade 34 • DK-8200 Aarhus N • T +45 70 27 70 12 • www.alexandra.dk

12. As result of the indirect request, the OP validates the request from the Relying Party and forwards its request for authentication to the Browser.

13. The Browser requests the End User for his/her authentication credentials.

14. The End User presents his/her credentials and consent to the Browser. Normally, the End User’s consent should only refer to whether the user accepts that the OpenID Provider authenticates him/her to login into the Client Application. But from v2.0 of the protocol, the addition of the extensions mechanism also allows for the exchange of user information from the OpenID Provider to the Client Application. However, the specification does not mention anything about whether the End User should consent for this exchange of information or how should s/he do so. The result is that several implementations just take for granted that the End User will agree in such information exchange without even informing the End User that this exchange of information is going to take place before s/he gives consent for the authentication process to take place.

15. The credentials and consent from the End User are sent to the OP in return of its request.

16. The login information is validated by the OP and then, it generates a signed nonce for the response to avoid replay attacks. This element has been introduced as normative on the 2.0 specification.

17. The response is forwarded to the Relying Party via the use of indirect communication.

18. The Relying Party validates the response provided by the OP and, as result of the indirect request, forwards its result to the Browser.

19. The Browser presents the result of the Authentication to the End User.

In order to ensure that the End User participates in the authentication protocol, the authentication and response messages are exchanged indirectly between the RP and OP by passing through the Browser.

The protocol also takes into account other alternative situations. This is the main reason for showing some important parameters as conditional ones (shown in brackets in Figure 1).

One alternative flow exploits the Single Sign-On features of the protocol by not asking again for the authentication of a user who has been previously authenticated.

As already mentioned, version 2.0 also introduces the provision of the Extensions mechanism. Extensions are useful for providing extra information about an authentication request or response allowing, for example, the provision of certain attributes from the user’s profile.

2.1.2. Protocol Analysis

Sharing identity information among several providers using OpenID raises a few security issues, such as:

- Privacy concerns: An OpenID provider will know every site visited by each user using their credentials. However, on the other hand, the relying party providers usually minimize the information requested from the user when...
they request it from the OpenID provider in contrast to what happens when a user is requested to fill out a form [Bellamy-McIntyre 1].

- **Centralized risk:** And OpenID account is much more valuable to hackers because it opens the door to many different sites. The more popular the OpenID provider, the more lucrative it may become for hackers.

- **Co-relation:** Several service providers accepting the same OpenID account can get together to co-relate the user’s information/activity.

- **Session swapping:** Given the fact that an OpenID session is not bound to a client, a web attacker may force the client to initialize a session authenticated as himself in order to obtain sensitive information from the victim if the user does not realize that he is logged in as the attacker.

- **Phishing attacks:** The user is under the threat of a phishing attack when being redirected from the Relying Party to the Identity Provider. Several anti-phishing solutions have been proposed (see for example [Lee 1]) based on the use of a second medium to authenticate the Identity Provider by the user. However, none of these methods has been adopted in a standard form yet.

- **Man-in-the-middle attacks:** The protocol does not mandate the use of SSL, although it is strongly recommended. However, even if this protocol is set, recent studies show that in most cases it is badly implemented [Georgiev 1]. A formal analysis of the protocol undertaken by [San-Tsai 2] shows that the protocol does not guarantee the authenticity and integrity of the authentication request and it proposes a simple and scalable mitigation technique to this problem based on the introduction of a shared key between the Browser and the Relying Party during the authentication process.

- **Buffer overflow and code injection:** The OpenID identifier is a simple URL. In many cases, an attacker can easily prepare an OpenID identifier aimed to hijack the contents of the service provider.

- **Replay attacks:** Even though the protocol is designed to avoid this problem, in practice, many service providers do not use TLS and do not check for nonces (only v2.0 mandates its use and it also suggests that they should be signed in order to avoid for their malicious replacement as pointed out by [Hyun-Kyung 1]). Therefore, most of the sites are not protected against replay attacks.

Probably, as pointed out by [Bellamy-McIntyre 1], the main problem of OpenID is that the resulting standard is too open. This may facilitate its initial adoption but it also complicates its integration.

Another important concern is the lack of security measures enforced by the standard, which results in implementations with very varied degrees of security. While at the same time, the user experience resulting from these implementations may seem very similar. As pointed out by [Bellamy-McIntyre 1], one way to minimize these problems is by white-listing the OP sites that a Relying Party allows End Users to give access to and/or white-listing the Relying Party sites that an OP may trust.

### 2.2. OAuth

OAuth 1.0 [OAuth 1] was designed to tackle a slightly different problem. When a user needs to make use of the information and resources that s/he initially stored in a service provider A over a different service provider B, instead of transferring those resources to the new provider B, via this protocol
s/he can give permissions for controlled access of the resources that s/he has on service provider A to the service provider B.

Therefore, OAuth could be taken as a complement to OpenID because OAuth puts its emphasis on Authorization whereas OpenID puts it on Authentication.

Following the OAuth naming conventions, the idea is to create an anonymous Authorization Code that the end user – Resource Owner – will generate via the identity provider – Authorization Server – in charge of maintaining and authenticating the information of the user so that it authorizes the consumer application – Client – to access resources kept in a controlled manner by the service provider – Resource Server - acting as relying party. The Client makes use of the Authorization Code to obtain an Access Token that will be used to gain access to the resources granted by the Resource Owner.

Unfortunately, the OAuth 1.0 specification was not properly detailed and this resulted in important differences among each implementation of the protocol. Therefore, this situation introduced several important compatibility issues.

This situation gave rise to OAuth 2.0 [OAuth 2], which aims to become a de facto standard for personal web authorization. On the other hand, OAuth 2.0 extends the original OAuth 1.1 protocol – which mainly supports Web browsers – to smart clients and many other client application ranges (see the Implicit flow description of the protocol).

### 2.2.1. Protocol overview

The OAuth 2.0 protocol has three main parts, Client Registration, Resource Owner access grant and Client access.

Depending on the features of the Client Application, OAuth 2.0 specifies four possible flows to grant access to the Client Application: The Authorization Code Grant – which is optimized for confidential clients -, the Implicit Grant – which is optimized for public clients known to operate a particular redirection URI -, the Resource Owner Password Credentials Grant – which is suitable in cases where the Resource Owner has a trust relationship with the client such as a highly privileged application – and the Client Credentials Grant – which can be basically used when the client is requesting access to protected resources under its control.

The specification may be extended to cover additional flows.

However, in this paper we will only describe the Authorization Code Grant and the Implicit Grant because they are the most widely deployed.
### 2.2.1.1. Client Registration

Client registration is represented in Figure 2.

![OAuth 2.0 Client Registration Diagram](image)

**Figure 2** OAuth 2.0 - Client Registration

According to OAuth 2.0, first of all, a Client application should register its information with the Authorization Server via the Initiate Endpoint published by such server.

The specification does not exclude the use of unregistered clients, but it states that its use would require additional security analysis and that this is beyond its scope.

The specification does not mandate the time or manner to register a client application. For example, it can be done interactively via an HTML registration form, automatically by the client application itself or by the authorization server via some form of client discovery.

The Authorization Server may publish the following Endpoints:

- Initiate
- Authorize
- Gen_Token
- Refr_Token

On the other hand, the Resource Server should also publish all the API Endpoints required to access and manipulate its resources.

The information to register by the Client application through the Initiate Endpoint includes the client type (either public or confidential depending whether the client application is capable of maintaining the confidentiality of their credentials) and the client Redirection Endpoint in which the Authz Server will publish the authorization credentials to use by the client in order to get access to the resources from the Resource Owner.

The Client may also exchange other information such as: the application name, description, logo image, acceptance of legal terms, etc…

The Authorization Server generates and issues a unique Client Identifier under its own scope. This ClientID may be public.

The server may also return a secret key to be used for the exchange of confidential information, etc…
2.2.1.2. **Authz Code Grant**

The protocol flow for the Authz Code Grant is presented in Figure 3.

![OAuth 2.0 - Protocol flow for the Authz Code Grant](image)

The protocol for the OAuth 2.0 Authz Code grant is designed for Client Applications that are able to maintain the confidentiality of their credentials. Its main flow presents the following steps.

1. The Resource Owner initiates the flow by requesting access to the Client via the User Agent (Browser).
2. The Browser forwards the request to the Client application. If the Client application is linked to several Authorization Servers, here we may have a flow similar to the steps 3-6 shown in Figure 1 for the protocol OpenID.
3. The Client application redirects the request to the Authorization Server indicating in the `Response_type` parameter that it wishes to use the Authz Code Grant flow.
4. The Authorization Server replies to the redirect request by sending an authentication request to the Browser.
5. The Browser requests the Resource Owner whether s/he accepts the requested authorization `Scope` and it also requests for the Authentication credentials of the Resource Owner. If the client omits the `Scope` parameter when requesting authorization, the authorization server must use default requirements. Although the presentation of the authorization scope is not covered by the specification, this is recognized as a very crucial aspect of the protocol because it is very important to ensure that the Resource Owner understands very well the consequences from his/her decision.
6. The Resource Owner returns the credentials and the scope of his/her authorization decision to the Browser.
7. The Browser forwards this information to the Authorization Server.
8. The Authorization Server authenticates the Resource Owner and establishes the scope of his/her decision by generating an Authz code bound to it. The specification mandates that the Authz code should expire shortly after being issued with a recommended maximum lifetime of 10 minutes.

9. The Authorization Server redirects the Authz code to the Client Application in a secured and confidential manner using the Ret_Code Endpoint indicated by the Client.

10. The Client requests an Access token by including the Authz code received in the previous step. The Client must not use the Authz code\(^1\) more than once. When making the request, the client authenticates with the Authorization Server. The specification recommends that the Client includes the State parameter – defined as an opaque value used by the client to maintain state between the request and callback, for example, a hash of the session cookie used to authenticate the user-agent – in order to minimize the threat of cross-site request forgery.

11. The Authorization Server authenticates the client and validates the Authz code. If validated, the Authorization Server generates an Access token – which are the credentials used to access protected resources - and, optionally and at the discretion of the Authorization Server, it also issues a Refresh token – which are credentials used to obtain Access tokens.

12. The Tokens are forwarded back to the Client via a direct response. The Token_type provides the client with the information required to successfully utilize the Access token to make a protected resource request and the specification states that the Client must not use an Access token if it does not understand the type.

13. If issued, the Refresh token has a much larger lifespan than the Access token and it is used to retrieve new Access tokens once they expire. This is done through a call to the Refr_Token Endpoint.

14. The Authorization Server generates a new Access Token and, if required and considered appropriate, another new Refresh Token as well. Given the fact that the refresh tokens are long-lasting credentials, they are bound to the Client to which they were issued and therefore, before generating the new tokens, the Authorization Server must authenticate the Client.

15. The new tokens are forwarded to the Client.

16. The Client accesses the protected resources by presenting the Access token to the Resource Server. The specification presents a brief example of the usage of the Bearer token or the MAC token. In fact, the adoption of the Bearer token, which is the most used, has caused enormous controversy among the editors of the specification because of its potential security flaws. At this point, the Resource Server must validate the Access token and ensure that its scope covers the requested resource.

17. The methods used by the Resource Server to validate the Access token are left unspecified on the RFC [OAuth 2], but they generally involve coordination between the Resource Server and the Authorization Server. In the case that the Authorization Server and the Resource Server are the same entity, this step is easily done. Otherwise, given the fact that this step is not specified, it may be an important source of incompatibility issues among different implementations.

\(^1\) The Authz code is used for obtaining User approval while the Access token is used to access Protected resources.
As indicated in [OAuth 3], Access tokens may be self-contained so that a resource server needs no further interaction with the authorization server or there may just be handles to information stored in the authorization server. The figure shows such case, which is being described in a draft that formalizes the description of such communication [OAuth 6].

18. [OAuth 6] defines the parameters to be returned from an introspection call that allow a Resource Server to learn all the information related to the Access Token issued by the Authz Server.

19. If everything is correct, the Resource Server executes the corresponding API call to retrieve/update the information from the Resource Owner.

20. The corresponding result is forwarded to the Client.

21. The Client returns the corresponding information to the Browser.

22. And the Browser finally displays the result of the call to the Resource Owner.

2.2.1.3. Implicit Grant

The protocol flow for the Implicit Grant is presented in Figure 4.

The OAuth 2.0 protocol for the Implicit Grant is designed for Client applications that can’t guarantee the confidentiality of their credentials\(^2\), such as user agent-based applications or native applications executed on the device used by the Resource Owner.

1-7. These steps are very similar to the ones presented in Figure 3 for the Authz Code flow. The main difference is that the Client application will redirect the request to the Authorization Server indicating in the Response_type parameter that it wishes to use the Implicit Grant flow. On the other hand, the communication protocol to establish between the Browser and the Client will depend on the case. In some cases, the Browser and the Client may be integrated into a single application.

\(^2\) These applications can’t keep the confidentiality of their credentials because the client credentials must be distributed with the copy of the application, which inevitably compromises them.
8. The Authorization Server generates the Access token directly without issuing any previous Authz token. Note that in this flow there is no option to have Refresh tokens either.


10. The Browser retains the URI fragment with the encoding of the Access token locally.

11. Notice that when the Browser forwards the redirection to the Web-Hosted Client Resource, it does not transfer it the URI fragment with the encoding of the Access token.


13. The Browser executes the script and extracts the Access token together with any other optional parameters.

14. The Browser transfers the Access token to the Client.

15-20. These steps are very similar to the ones presented in the steps 16-22 of the Authz Code flow which may also include the coordination protocol description described in [OAuth 6].

2.2.2. Protocol Analysis

The main problems of the OAuth 2.0 specification are:

- Lack of standardization: The OAuth 2.0 specification includes more than 50 instances of the keyword SHOULD which, in many cases, should be changed to MUST for the protocol to become more secure and interoperable. But even if this was corrected, the specification is far too open and does not cover several important details - such as the format of the Access Token and the Authorization Credential -, which results in several incompatibility issues. In fact, the OAuth 2.0 specification acknowledges this situation and justifies their decision to define such an open framework with the expectation that future work will define prescriptive profiles and extensions to achieve full interoperability.

Another problem is the fact that the final security of a solution depends on the implementation details resulting from several actors – the Authorization Server, the Resource Server and the Client. Therefore, in most cases, the main commercial actors – such as Google, Facebook or Yahoo - have created their own implementation of the Authorization Server and Resource Server associated with their own particularly library API to be used by Client implementers in order to access their own resources. This approach, although in first instance may facilitate the life of the Client implementers, it has also been the way to tight them up to a particular implementation of the protocol.

- Difficulty to achieve a secure server implementation: As shown by [San-Tsai 1], although the protocol is inherently secure, an empirical analysis of existing implementations of the protocol showed that in fact, it is very difficult to achieve a secure and guaranteed implementation given its open specification and the quantity of security factors to take into account. The paper [San-Tsai 1] shows that one of the main sources of error comes from the incorrect use of bearer tokens, an element that caused a lot of controversy among the editors of the specification when it was introduced in the 2.0 specification.
However, from the perspective of the client developer, OAuth 2.0 is much easier to implement than OAuth 1.0. This seems to have been the main driving force driving the main changes in OAuth 2.0, giving the fact that simplifying the life of the client developer to achieve a secure implementation of the protocol was one of the main factors that could allow for a fast, broad and secure adoption of the protocol.

- Lack of security transparency: From the Resource Owner’s perspective, it is not possible to know or understand the security guarantees offered by an implementation of the protocol even though this may differ substantially among particular implementations.

- Lack of orthogonal user experience: On the other hand, depending on the user interface designed by the Authorization Server, the Resource Owner’s experience may vary significantly in terms of flexibility and clarity at the time of deciding the scope of the level of access that should be granted to a Client application requesting access to the Resource Owner’s resources.

The result is a specification which is very difficult to adopt in terms of security, but particularly in terms of interoperability. Therefore, only the most important cloud identity providers offer a reliable implementation of the protocol by offering a set of API libraries that can theoretically be easily adopted by Resource Server and Client developers to allow access from their Authorization Server.

However, once adopted, the resulting implementation is tight to a particular Authorization Service and therefore, to allow access from a different Authorization Provider, the implementation needs to be redesigned in order to incorporate the libraries from the new provider.

On the other hand, OAuth is already widely used and deployed on the web. However, OpenID Connect has the potential to become a more widely deployed standard for Resource Authorization in the future.

### 2.3. OpenID Connect

The OpenID Connect 1.0 specification [OpenID Connect 1] defines the OpenID Connect protocol as a simple identity layer on top of the OAuth 2.0 protocol.

Indeed, OpenID Connect 1.0 is basically a redefinition of OAuth 2.0 resulting from the merge and enhancement of OpenID and OAuth 2.0 by providing authentication information of the user while trying to overcome the main problems detected with the adoption of OAuth 2.0.

The result is the provision of a complete standard framework designed to build secure and compatible OAuth implementations.

#### 2.3.1. Protocol overview

The OpenID Connect protocol basically keeps the same structure as OAuth 2.0 protocol making special emphasis on the security of the protocol and the End-User Authentication.

and therefore, it can also divided in three main parts: Client Registration, Resource Owner access grant and Client access.

Depending on the context of the Client Application, OpenID Connect 1.0 introduces three possible flows: The Authorization Code Flow, the Implicit Flow or the Hybrid Flow.
The characteristics of the three flows are summarized by the standard on the following table:

<table>
<thead>
<tr>
<th>Property</th>
<th>Authz Code Flow</th>
<th>Implicit Flow</th>
<th>Hybrid Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>All access tokens returned from Authz Endpoint</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>All access tokens returned from Token Endpoint</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Tokens not revealed to User Agent</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Client can be authenticated</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Refresh Token possible</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Communication in one round trip</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Most communication server-to-server</td>
<td>yes</td>
<td>no</td>
<td>varies</td>
</tr>
</tbody>
</table>

2.3.1.1. Client registration

The OpenID Connect Client Registration protocol is defined in [OpenID Connect 5].

The specification mandates that the Client registration should always be done dynamically via the @Initiate registration Endpoint published by the OpenID Provider.

The @Client_Registration endpoint is a protected resource used by the Client applications to register their service (see Figure 5). These requests may be limited – restricted to the registration of only authorized clients –, or open up to a certain extend – in this case, at least there should be some measures in place to limit the potential harm of denial-of-service attacks.

Figure 5 presents the flow for Client Registration.

1. The Client may use the OpenID discovery protocol [OpenID Connect 4] - which is mainly based in WebFinger [WebFinger 1] – to find the location of the OpenID Provider @Initiate Endpoint. The Client may also use any other out-of-band mechanism to find this information.
2. The Client sends an HTTPS Post request with all the parameters to register in the form of a JSON object.

The flow used is determined by the “response_type” value contained in the Authorization Request according to the following table:

<table>
<thead>
<tr>
<th>&quot;response_type_value</th>
<th>Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>Authorization Code Flow</td>
</tr>
<tr>
<td>id_token</td>
<td>Implicit Flow</td>
</tr>
<tr>
<td>id_token token</td>
<td>Implicit Flow</td>
</tr>
<tr>
<td>code id_token</td>
<td>Hybrid Flow</td>
</tr>
<tr>
<td>code token</td>
<td>Hybrid Flow</td>
</tr>
<tr>
<td>code id_token token</td>
<td>Hybrid Flow</td>
</tr>
</tbody>
</table>

The OpenID Provider generates a new ClientID, the Registration Access Token to use by the Client when requesting further operations and possibly a secret key to secure the communications between the Client and the OpenID Provider.

3. This information is forwarded together with the endpoints to use by the Client in further operations as a result of the @Initiate call.

2.3.1.2. End-User Authentication

OpenID Connect returns the result of the Authentication in a secure manner so that the Client can rely on it. For this reason, in this case the Client is also named the Relying Party.

2.3.1.2.1. Authentication using Authorization Code Flow

The default flow for OpenID Connect, Authz Code Flow [OpenID Connect 2] is depicted in Figure 6.
1-7. End-User Authentication: These steps are quite similar to the ones corresponding to the OAuth 2.0 Authz Code flow presented in Figure 3. But the OpenID Connect specification is very precise in defining the End-User Authentication by defining the parameters to exchange, particularly in terms of the description of the Scope and Display to present to the End User in order to try to ensure that s/he is aware of all the exchange of information that s/he is going to give consent to. The Scope is returned from the Resource Owner if the Scope accepted by the Resource Owner differs from the Scope that was initially required.

8-15. Access Token issuance: Apart from a much more complete specification than the one provided in OAuth 2.0, the main difference here is the introduction of the ID Token parameter, which is a very important and required element that had not been defined in OAuth 2.0. This element permits to ensure the security of the protocol because it provides for a much more robust usage of the Bearer Access Token by allowing for the identification of the issuer and subject of the access token – possibly including also many other optional claims – encoded in a JSON Web Token [JWT 1], a JSON object that is digitally signed using JSON Web Signature (JWS) and/or encrypted using JSON Web Encryption (JWE). (On the previous section, the message 3 may also provide a Request parameter which is very similar in structure to this ID Token parameter in order to provide more security guarantees to the request).

On top of this, the ID Token can also be used to return User Information.

2.3.1.2.2. Authentication using Implicit Flow

The main flow corresponding to the OpenID Connect Implicit Flow protocol [OpenID Connect 3] is depicted in Figure 7.
This protocol is very similar to the OAuth 2.0 Implicit Code flow shown in Figure 4 but introducing the main features just described for the OpenID Connect 1.0 Authz Code Flow.

However, it is interesting to notice that in this flow, the usage of the Nonce parameter is mandatory on the Authorization Request sent by the Client. The Nonce parameter is also a required component of the Id Token together with a hash value of the Access Token.

On the other hand, even though it may seem that the connection between the Browser and the Client/Relying Party is not specified, we should note that the Client/Relying Party represented here – and for the Hybrid Flow as well - may be often implemented as a javascript client running in a browser.

2.3.1.2.3. **Authentication using Hybrid Flow**

The Hybrid Flow returns some tokens from the Authorization Endpoint and others from the Token Endpoint mixing the previous two flows as presented in Figure 8.
Figure 8 Authentication in OpenID Connect 1.0 – Protocol for the Hybrid Flow
2.3.1.3. Client access

Once authentication has been performed, the Client may – if it does not already done so via the retrieval of ID Token parameters - retrieve User Info and/or User Claims by following the protocol shown in Error! Reference source not found..

This part introduces a few differences with respect to the OAuth 2.0 specification.

On one hand, it introduces the possibility to specify Claim requests. Claim requests are used to request that specific claims should be returned by indicating that a parameter is essential, optional or that it should correspond to a particular value from a set of values. Claim requests also allow for the specification of non-standard claim requests. They can also be set to request values in particular languages. The specification distinguishes also among the following types of claim values depending on how they have been retrieved:

- **Normal Claims**: Claims that are directly asserted by the OpenID Provider.
- **Aggregated Claims**: Claims that are asserted by a Claims Provider other than the OpenID Provider but are returned by OpenID Provider.
- **Distributed Claims**: Claims that are asserted by a Claims Provider other than the OpenID Provider but are returned as references by the OpenID Provider.

On the other hand, this part also includes a description for the management of the state of the connection [OpenID Connect 6] on an ongoing basis so that the End-User can log out from the Relying Party or the OpenID Provider and this information automatically passed over to the other service.

1. Request for the retrieval of User Information. Notice that OAuth 2.0 did not restrict in such a way this Endpoint, allowing a Resource Server to define its
The Alexandra Institute • Aabogade 34 • DK-8200 Aarhus N • T +45 70 27 70 12 • www.alexandra.dk

own @User_Info_API. However, OpenID Connect provides a specific End-Point that has to be accessed exclusively from the OpenID Provider.

2. In case of Session State management, normally the session starts after issuance of the End-User’s ID Token. At that moment, the Relying Party loads an invisible iframe that must know the ID of the OpenID Provider iframe in order to be able to post messages to check the End-User session state with the OpenID Provider.

3. The OpenID Provider should change the value of the session state returned to the Client under one of the following events:
   - The set of users authenticated to the browser changes.
   - The authorization status of Clients being used by the End-User changes.

4. By means of this EndPoint, a Relying Party can also notify the OpenID Provider that the End-User has logged out of the site, and might want to log out of the OpenID Provider as well.

5. The OpenID Provider will require of synchronization with the Resource Provider in order to provide values for aggregated or distributed claims. As reviewed in the case of the OAuth Auth Code grant, such synchronization may or not follow the protocol defined in [OAuth 6].

6. In this step, the OpenID Provider returns the User Information and requested claim values to the Relying Party.

7. In case of Distributed Claims, the Relying Party will need to retrieve the corresponding values from the Resource Provider User_EndPoint by using the Distributed Access Token provided on the Claims Info parameter returned by the OpenID Provider.

8. If everything is correct, the Resource Provider will return in this step the corresponding values for the Distributed Claims being requested.

9. In case of Implicit or Hybrid flows, the information retrieved is transferred from the Client to the User Agent.

10. Finally, the User Agent presents the corresponding values to the End-User.

2.3.2. Protocol Analysis

   This protocol tackles the main problems of the OAuth 2.0 specification:
   - Lack of standardization: The OpenID Connect 1.0 specification, while still being a work in progress, already provides for a much more complete and concise specification than the OAuth 2.0 allowing for much better interoperability among OpenID Providers, Resource Servers and Client Applications.
   - Difficulty to achieve a secure implementation: The introduction of the JSON Web Token data structures "Request" and "Id Token" together with the mandate of using TLS connection whenever possible, provides for a much more reliable implementation of the OAuth protocol simply based on the use of Bearer tokens in a more secure, efficient and simple to implement fashion than the one introduced by the OAuth 1.0 [OAuth 1] specification.
   - Lack of security transparency and orthogonal user experience: The introduction of the Display parameter, which describes how to present the authorization and consent information to the Resource Owner. However, we still feel that there is still room for improvement in this direction such as, for example, allowing to distinguishing among optional and mandatory parameters in the "acr_values" list or
finding better descriptions that could allow end users to understand better what they are really consenting to.

On the other hand, this protocol introduces a very interesting new feature:

- **Discovery protocol:** The introduction of the discovery protocol for Client and Resource Applications allows for a very dynamic and natural adoption of new services on the web.

This dynamic adoption requires for the implementation of a dynamic trust model. This dynamic trust model is based on Trust On First Use (TOFU).

The first time a user tries to authorize a new Client application or Resource Server, the OpenID Provider warns the user that the site that s/he is going to give access to is a new and unknown service.

Once the user has accepted once the authorization of access for this site, the OpenID Provider won’t warn the user anymore on the basis that s/he already accepted such site before.

This solution provides for the introduction of a Gray List – based on user decisions - on top of the White List – a list of predefined trusted partners – and Black List – sites officially recognized as potentially harmful – that the OpenID Provider may have in place by default.

We have to consider that OpenID Connect is a very recent specification that, although it may seem a promising direction for the near future, it still has to prove its full potential on a real scale.

2.4. **UMA**

OpenID Connect is not the only candidate to substitute OAuth in the future. User-Managed Access (UMA) is a project from the Kantara Initiative that is defined as a profile for OAuth 2.0 but with a much broader focus than OAuth 2.0 [UMA 1].

2.4.1. **Protocol overview**

UMA defines three phases to provide for controlled access to a resource:

1. **Protect a resource:** The Resource Owner decides to protect a Resource – which is identified by an URI address – through an Authorization Manager by providing some access policies.

2. **Get authorization:** The Relying Party – which does not have to be the same entity as the Resource Owner – requests for authorization to access the resource.

3. **Access a resource:** The Relying Party can get access to the resource.

UMA protocol is based on the work undertaken by some of the previous protocols:

- From OAuth 2.0 it adopts the token management schema.
- From OpenID Connect 1.0 it adopts the dynamic client registration.

---

[UMA 1]: MITRE Corporation has recently released an initial implementation of the protocol.
2.4.1.1. Client and Resource Server registration

UMA Client Registration is based on the OpenID Connect 1.0 dynamic registration (see Figure 5) but enhancing it by also allowing for the dynamic registration of resource servers. This protocol is being defined as an enhancement to the OAuth 2.0 protocol [UMA 2].

2.4.1.2. Authorization API Token (AAT) Grant

UMA defines two types of Client applications: The Resource-consumer Client applications and the Management Client applications.

Both types of Client applications require an Authorization API Token (AAT) Grant, which is delivered to them by the Authorization Server following the normal OAuth 2.0 protocol as shown in Figure 9.
2.4.1.3. **Protection API Token (PAT) Grant**

On the other hand, Resource Servers need to acquire a Protection API Token (PAT) Grant in order to be able to act as Resource Servers in front of the Authorization Server.

This is also done via the OAuth 2.0 protocol as shown in Figure 10.
2.4.1.4. Resource Protection

Before protecting a resource, the Resource Owner has to register it first so that, from that moment onwards, the Resource Server will protect such resource according to the policies defined on the Authorization Server.

This is done via the Resource Protection flow [UMA 2] as shown in Figure 11.

This flow shows that the Resource Server has to include a Management Console from which the Resource Owners define the security policies to control and manage the access to their resources.

The flow from Figure 11 shows that the information about the security policies set to protect a resource URI are kept by the Authorization Server.

Therefore, the Resource Management Console has to map the Resource_uri shown to the Resource Owner with the corresponding Rsc_id defined on the Authorization Server.

The security policies are managed on a RESTful way by using the methods PUT (to create or update a resource set description), GET (to list or read resource descriptions) and DELETE (to remove them).

---

*Figure 13 UMA – Main flow for the Resource Protection Management protocol*
2.4.1.5. Resource Authorization and Access

When a Requesting Party requires a resource, the Client application first needs to obtain a Requesting Party Token (RPT) Grant from the Authorization Server by following the protocol depicted in Figure 12.

![Diagram of Resource Authorization and Access](image)

Figure 14 UMA – Main flow for the RPT Grant

Once the Client application has got hold of the RPT Token, it has to retrieve a Permission Ticket. This Permission Ticket is generated by the Authorization Server but it is being retrieved via the Resource Server.

With the Permission Ticket, the Client application requests the corresponding Authz Code from the Authorization Server. But before providing the corresponding Authz Code, the Authorization Server has to authenticate the End user requesting access and validate that this user has effectively got the rights to access the requested resources.

The specification leaves quite open the process of gathering Claims from the End user. Figure 13 presents the flow corresponding to the OpenID Connect Claim Profile presented as example in [UMA 2].
Finally, once the Client retrieved the Authz_Code, the Requesting Party can obtain the requested resources from the Resource Server following to the protocol depicted in Figure 14.

Figure 15 UMA – Authorization Request

Figure 16 UMA – Resource Access
2.4.2. Protocol Analysis

The main features introduced by the UMA protocol are:

- **Off-line Authorization**
  Resource Owner’s set up permission policies via an administration console designed to allow them controlled access to their resources.

- **Centralized Authz Management control over several Resource Servers**
  The administration console allows them controlled access to all their resources – independently on the Resource Server in which they are located.

- **Provision of access to third parties**
  From the administration console, Resource Owners can provide access to other users (and not only to them-selves).

- **Standard to establish legal binding obligations among UMA Participants**
  Given the fact that UMA allows for the dynamic binding among many participants, there is an effort to establish a common legal framework that can be used as a base from which to solve any possible legal disputes [UMA 4].
3. Comparative Analysis

In this section we will compare different features among the following Authentication/Authorization technologies that we already introduced in the previous section: OpenID 2.0, OAuth 2.0, OpenID Connect 1.0 and UMA 1.0.

3.1. Standard status and current deployment

All the technologies presented in this study are still evolving. This section presents what is the current status of evolution and adoption for each of them.

3.1.1. Standardization Process

Given the fact that most of the technologies being presented are quite recent, several standard bodies are currently on the process of defining the specification for several of them.

- OpenID 2.0: OpenID specifications are developed by the community on technical specification mailing lists overseen by specs@openid.net.

OpenID 1.0 and OpenID 1.1 are currently considered to be obsolete specifications of the protocol. OpenID Authentication 2.0 is a finalized specification of the protocol.

Probably, as pointed out by [Bellamy-McIntyre 1], the main problem of OpenID is that the resulting standard is too open. This may facilitate its initial adoption but it also complicates its integration.

Therefore, even though nowadays OpenID is still widely used, newer and more secure protocols are rapidly taking over, and in fact, OpenID 2.0 is already considered obsolete.

- OAuth 2.0: OAuth 2.0 is a proposed RFC 6749 drafted by the oauth.net community and submitted on October 2012 to the IETF. This new proposal will probably soon replace and make obsolete RFC 5849, which defined the OAuth 1.0 protocol.

The OAuth 2.0 specification includes more than 50 instances of the keyword SHOULD which in many cases should be changed to MUST for the protocol to become more secure and interoperable. But even if this was corrected, the specification is far too open and does not cover several important details - such as the format of the Access Token and the Authorization Credential - which results in several incompatibility issues. In fact, the OAuth 2.0 specification acknowledges this situation and justifies their decision to define such an open framework with the expectation that future work will define prescriptive profiles and extensions to achieve full interoperability.

- OpenID Connect 1.0: The OpenID Connect specifications are developed by the OpenID Foundation (openid.net/connect), which is also behind OpenID. It is introduced as an authentication layer on top of OAuth 2.0, merging the experiences of OpenID into OAuth.

Its final specifications were released on February 25th 2014 [OpenID Connect 1] and it is thus very recent. Therefore, it still has to prove its full potential on a real scale.

The OpenID Connect provides for a much more complete and concise specification than OAuth 2.0. This allows for a much better interoperability among OpenID Providers, Resource Servers and Client Applications.
UMA 1.0: User Managed Access is a specification currently being developed by the Kantara Initiative, an Industry Consortia connecting Government, Industry and Research in Identity Management. The specifications related to the UMA protocol are being incubated in the Kantara Initiative with the intent to contribute the draft work to the IETF.

Even though UMA 1.0 is still a work in process, the Kantara Initiative seems to be trying to define a specification as complete as possible. Given the fact that UMA allows for the dynamic binding among many participants, it is even trying to include topics normally outside the scope effort of a technical specification. In this sense, there is an effort to establish a common legal framework that can be used as a base from which to solve any possible legal disputes [UMA 4].

3.1.2. Current deployment

The oldest technologies are the most successfully deployed but they may be replaced by the newest technologies as user awareness grows.

- OpenID 2.0: OpenID has been adopted by a broad number of web developers and users. On December 2009 there were over one billion OpenID-enabled user accounts and 9 million websites using OpenID [OpenID 2]. Although the OpenID 2.0 specification was published on December 2007, according to [Tapiador 1], by the year 2011 the deployment of this version was still significantly lower than the deployment of version 1.0 (97,32% even though this version presented a very fundamental security problem) and version 1.1 (43,54%).

- OAuth 2.0: OAuth is already widely used and deployed on the web and it has already been adopted by the most important social services although, given the fact that the 2.0 version of the protocol is very recent and its publication has caused some controversy issues, some of them are still relying on their 1.0 implementation of the protocol (e.g. yahoo).

- OpenID Connect 1.0: For this recent specification currently exists a first reference implementation named MITREid Connect [MITREid 1]. There are already a few commercial providers offering this protocol, the most remarkable being Google, Microsoft, Deutsche Telekom and PayPal.

- UMA 1.0: The Kantara Initiative is preparing an UMA interoperability and conformance feature tests. The current experimental implementations of the UMA specification include pilot projects from the Fraunhofer Institute, Cloud Identity, Gluu or the SMART project and they can be found at [UMA 5].

3.2. Use Cases

This section presents and compares a few of the main factors taken into account when defining a Use Case.

3.2.1. RESTful architectures

As indicated by the designer of the Representational State Transfer [REST 1], the REST architectural style emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

The features that distinguish a RESTful architecture are: Client-Server, Stateless Server, Cache Constraints, Uniform Interface, Layered System and Code-On-Demand. The concept of Stateless Server can be taken with
some flexibility; if the server has to keep some state, this should be minimal so that it should enable for horizontal scalability.

- OpenID 2.0: OpenID allows for the adoption of RESTful architectures because the OpenID Provider does not need to keep any state once a user has been logged in.

  In order to avoid the login from the user once s/he has already been logged in, the Relying Party may use cookies that, although are to be kept in the Browser, they may imply some minor management from the part of the Relying Party. However, in any case, this would not have any implication on the part of the OpenID Provider either.

- OAuth 2.0: In this case, the Authz Server should keep some temporary status’ information for the management of the Authz Codes and Refresh Tokens to generate. However, the information to be kept is minimal and temporary, still allowing for horizontal scalability of the services.

- OpenID Connect 1.0: In this respect, this protocol shares the same features as OAuth 2.0.

- UMA 1.0: In this case, we cannot really state that UMA accomplishes the feature of Stateless Server given the large quantity of information that the Authz Server has to keep and manage, including the security policies linked to all the resources being secured. However, even though the architecture of the overall system cannot be considered RESTful because the server side cannot be scaled in a RESTful way, this design still allows for the adoption of RESTful clients because it complies with the required parameters from a client perspective.

We should note that the latest architectures extend the concept of RESTful architectures to Lightweight Services by adding the following features: HTTP Transport, small and compact messages (usually JSON) and dynamic interoperability.

### 3.2.2. Web Service clients

The protocols being presented are mainly used from an Internet Browser. This is the reason why all these protocols make use of HTTP Redirect calls.

However, applications developed outside the scope of a Browser should be implemented as Web Service clients that can’t make use of such mechanism. Therefore, in order to allow for such type of applications, the protocols being presented should be adapted in some way to avoid requiring the usage of the HTTP Redirect calls.

- OpenID 2.0: [OpenID 3] presents a mechanism that can be used by Web Service clients to authenticate to OpenID providers.

- OAuth 2.0: [OAuth 4] provides a simple API interface for the Java platform in conformance with OAuth Core 1.0a specification.

- OpenID Connect 2.0: [OpenID Connect 8] provides a Java library for designing OpenID Connect client and server applications. [OpenID Connect 7] provides a reference implementation of the protocol for the Java platform.

- UMA 1.0: [UMA 5] presents the list of current implementations of the UMA protocol. However, for the moment there is no SDK for the development of Web Service client applications.
3.2.3. Mobile applications

Due to their limitations on processing power and security features, authentication and authorization protocols may define specific flows for mobile applications.

- OpenID 2.0: OpenID does not provide any particular support for mobile applications but it does not place any special requirements on them either. Therefore, an OpenID mobile application is exposed to the same threats as any other OpenID application.

- OAuth 2.0: The Implicit Grant flow is designed to allow the secure implementation of this protocol on applications that can’t guarantee the confidentiality of their credentials, such as mobile applications.

- OpenID Connect 1.0: This protocol defines the Implicit Client Profile which, in fact, corresponds with the Implicit Grant flow defined in OAuth 2.0. The adoption of JSON Web Token and JSON Web Signatures also facilitates their exchange and processing on mobile applications for this protocol4.

- UMA 1.0: This protocol also makes use of the OAuth 2.0 protocol to issue Authorization API Token and Protection API Token Grants and therefore, it recommends the adoption of the Implicit Client flow to issue these grants on mobile applications.

3.2.4. Single Sign-On capabilities

Single Sign-On (SSO) is the mechanism by which, once a user has logged on a service, s/he could enter into a different service without requiring him/her to login again.

Given that all the specifications analyzed provide for some sort of centralized authentication system, all of them have the potential for providing Single Sign-On services to their users. However, their real availability depends on each actual implementation features.

- OpenID 2.0: OpenID is about delegating authentication onto an OpenID Provider so that you can effectively login on multiple sites with one single set of credentials. This mechanism can “incidentally” provide for Single Sign-On in case that the second service you are trying to access supports the same OpenID Provider and that your browser supports 3rd party cookies. However, according to rule 3.3.6 of RFC 2965, this mechanism should be avoided in order to avoid potentially harmful uncontrolled interactions from malicious services.

- OAuth 2.0: Although OAuth stands for Authorization and not Authentication, indeed, many implementations of OAuth include both mechanisms at the same time. And, in fact, many important OAuth providers (such as Facebook, Twitter or Google) already incorporate “incidental” SSO service capabilities in their implementations for clients using the same OAuth provider in a similar manner as the one presented for OpenID. However, thanks to the introduction of the Access Token, this protocol is better protected (particularly when using MAC Access Tokens) towards the uncontrolled interactions from malicious services.

---

4 In opposition, SAML [SAML 1] can’t be properly supported by many mobile apps. This is due to the extensive length of SAML assertions, what forces them to be included on the HTTP POST message body instead of sending them on the message head. Unfortunately, due to the restrictions imposed by their environment, many mobile applications can’t process the message body.
- OpenID Connect 1.0: OpenID Connect enhances the security of Access Tokens and therefore, in terms of SSO adoption, it is equivalent or even more secure than OAuth 2.0.
- UMA 1.0: In these terms, UMA does not improve the OpenID Connect protocol.
3.2.5. **Dynamic Identity Federation**

Dynamic Identity Federation allows for a very dynamic and natural adoption of new services on the web. However, dynamic federation may also pose some security issues.

- **OpenID 2.0**: The OpenID identifier is a simple URL. This facilitates the dynamic establishment of the Authentication process by the Relying Party via an external and possibly unknown OpenID Provider.

- **OAuth 2.0**: This protocol requires that a Relying Party initially registers with the Server that it is going to use for Authorization purposes. This process is normally established statically during the deployment of the Relying Party application.

- **OpenID Connect 1.0**: This protocol offers a dynamic trust model based on Trust On First Use (TOFU) which consists of the following:
  - The first time a user tries to authorize a new Client application or Resource Server, the OpenID Provider warns the user that the site that s/he is going to give access to is a new and unknown service.
  - Once the user has accepted once the authorization of access for this site, the OpenID Provider won’t warn the user anymore on the basis that s/he already accepted such site before.
  
  This solution provides for the introduction of a Gray List – based on user decisions - on top of the White List – a list of predefined trusted partners – and Black List – sites officially recognized as potentially harmful – that the OpenID Provider may have in place by default.

- **UMA 1.0**: UMA Client Registration is based on the OpenID Connect 1.0 dynamic registration but also allowing for the dynamic registration of resource servers.

3.3. **Security features**

This section presents a few fundamental security features to take into account when adopting an Authentication/Authorization technology.

3.3.1. **Security Approach**

We can distinguish two very different security approaches: Organization and User-Centric.

3.3.1.1. **User-centric approach**

In this case, the resource server is only used to store personal information. Most of the personal information stored in the system is not necessarily verified by any external party. It is just managed by the user according to his/her interests.

Under this model, identity federation is managed by the user himself. Normally, s/he is under control of the personal information transferred among the identity information authority and the relying party.

This is the model followed by most of the technologies presented in this review: OpenID 2.0, OAuth 2.0, OpenID Connect 1.0 and UMA 1.0.

However, the UMA 1.0 protocol extends this boundary by allowing the management of Authorization credentials to third parties (other users or organizations).
3.3.1.2. Organizational approach

In this case, the resource server keeps the information from the organization and the organization keeps hold of verified personal information used to identify his staff.

A user presents his/her credentials in order to be authenticated by the system. Once in, s/he is given a controlled access to the resources of the system according to his/her role.

SAML [SAML 1] is a standard protocol designed to be tailored and eventually, adopted by any generic authorization/authentication protocol. However, even though this may be true, it was certainly designed with the spirit of providing a standard solution to manage Identity Federation and Single Sign-On capabilities in an organizational context.

SAML and WS* provide a framework to deploy Claim-based identity.

Claim-based Identity provides application developers with a common framework to manage all sorts of identities in the same way, independently from where they are coming from.

A Claim-based identity provides the application with the pertinent information required from each user presented always in the same format, independently from the identification means used by each user. The Identity Provider is the entity in charge of authenticating the user and the Security Token Issuer is in charge of generating the claims required by each application in the claim’s standard format established by the claims.

3.3.2. Credential management

Credential management takes care of the whole life cycle of a credential, from the time it is being issued until the time it expires or it is being revoked.

- OpenID 2.0: Authentication credentials are only used once immediately after being issued. Therefore, there is no need to provide any mechanism to manage them from the End User’s perspective.

- OAuth 2.0: OAuth does not define any credential management mechanisms because it recommends making use of the Authz Code and Access Tokens in a transient manner and as soon as possible. However, OAuth 2.0 also introduces the concept of Refresh Tokens, which are long-term credentials used to obtain new Access Tokens without requiring for a new login. However, given the fact that these Refresh Tokens may be subject to theft, a mechanism to revoke such tokens is specified in [OAuth 3].

- OpenID Connect 1.0: This protocol defines the same mechanism as the ones specified for OAuth 2.0.

- UMA 1.0: Credential management is at the core of the UMA protocol design. Users manage the whole live cycle of their credentials via the Resource Server Management Consoles.

3.3.3. Non-repudiation

Non-repudiation is the assurance that someone cannot deny something that s/he has previously done.

Non-repudiation normally makes sense only when there is an interaction of at least two different parties.
Given the fact that protocols OpenID, OAuth and OpenID Connect normally imply mainly a single user – the one who is trying to make use of resources placed on different services —, it does not make much sense to talk about non-repudiation in the context of those protocols.

- **UMA 1.0**: In this case, non-repudiation is guaranteed by providing signing mechanisms on the messages exchanged among the different parties involved in the protocol.

### 3.3.4. Mapping of “Stork” Authentication Levels

The Stork project [Stork 1] set a direction for the establishment of a pan-European Identity Management System.

In order to assess the quality of the authentication process, the Stork project defined four quality levels – defined as Quality of Authentication Assurance level, QAA - layered according to the impact of erroneous authentication [Stork 2]:

1. **Very low or negligible**: The authentication method either assures minimal confidence in the asserted identity or no confidence at all.
2. **Low impact**: Even if the claimants are not required to appear physically during the registration, their real-world identities must be validated and a token issued by a body subjected to a specific governmental agreement. Identity tokens must be delivered with accuracy and security guarantees. Sufficiently robust authentication protocols must be used during the electronic authentication phase.
3. **Substantial impact**: The registration of an identity is processed with methods that unambiguously and with a high level of certainty identify the claimant. The identity providers are supervised or accredited by the government. The credentials delivered are at least soft certificates or hard certificates. The authentication mechanisms used in the remote authentication phase are robust.
4. **Heavy impact**: The registration requires at least once (i.e., the very first time of the request but not for a later renewal) either the physical presence of the claimant or a physical meeting with the claimant. Furthermore, the identity provider must be a qualified entity according to the Annex II of the e-signature Directive. The certificates are hard certificates qualified according to the Annex I of the e-signature Directive. The most robust authentication mechanisms are used during the authentication phase.

Although ENISA (the European Network and Information Security Agency) acknowledges that the definition of the quality levels is a work in progress that should be further elaborated in future versions, it establishes that the principles of QAA levels are sound [Stork 3].

Therefore, we believe that it could be of interest to consider the inclusion of a QAA mechanism in the authentication/authorization protocols being studied in order to facilitate the future exchange of information with other European partners.

- **OpenID 2.0**: The QAA levels could be introduced in OpenID via its adoption through the extensions mechanism introduced in the v2.0 of the protocol [OpenID 1].

---

5 In this context we are not focusing on the interactions of the user with each service.
- OAuth 2.0: OAuth also introduced an extensions mechanism in its v2.0 that could be used to convey such information [OAuth 2].
- OpenID Connect 1.0: OpenID Connect 1.0 introduces the parameter "amr" on the ID Token structure that could easily be used to exchange this information [OpenID Connect 9].
- UMA 1.0: UMA could also make use of the extensions mechanism defined in OAuth 2.0 to convey this information.

3.3.5. Identity anonymity

This term refers to the state of an individual’s personal identity being publicly unknown.

- OpenID 2.0: The OpenID provider can trace the access to the different services that are being done from a single OpenID identifier.
  But on top of that, in most cases, a third party can easily intercept and trace the calls done from a single OpenID.
  And the last problem - but not the least - is that very often; the name of the OpenID identifier directly reveals the real name of the entity behind it.
- OAuth 2.0: The Authz Code and Access Tokens generated by the protocol provide for proper identity anonymity not allowing a third party to link the identity of the person accessing a service with his/her history usage (The standard indicates that Authz Codes must be used only once and that they should have a maximum lifetime of 10 minutes. It also indicates that Access Tokens should have a parameter indicating their expiration time).
- OpenID Connect 1.0: This protocol shares the same features as OAuth 2.0 in terms of Identity anonymity.
- UMA 1.0: This protocol extends the Authorization of a resource over time and different entities. Therefore, the resulting anonymity is improved because it is much more difficult than in the previous cases to establish a link between resource authorization and resource access.

However, as stated in [OAuth 5], the Authorization Server comes to be in possession of resource set information (such as names and icons) that may reveal information about the resource owner. Therefore, it is recommended to obscure resource set identifiers. But in any case, the relationship among the Authorization Server and the Resource Server is considered to accommodate in opposition with the relationship with the Client, which is entirely untrustworthy until authorization data is associated with its Requesting Party Token (RPT).
4. Summary and final conclusions

This paper compared the most prominent Identity Management technologies that are currently being designed and adopted on web applications.

The paper presented a detailed description of each technology providing a common framework from which they could easily be understood and compared.

The final part of the paper presented a comparative analysis focused on three main axes: Standard Status, Use Cases and Security Features which can be summarized on the following figures.

---

**Figure 17 Standard Status - Comparison Table**

<table>
<thead>
<tr>
<th>System</th>
<th>Stable version</th>
<th>Full specification</th>
<th>Current deployment</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenId 2.0</td>
<td>yes</td>
<td>no</td>
<td>Broadly spread</td>
</tr>
<tr>
<td>OAuth2.0</td>
<td>yes</td>
<td>no</td>
<td>Broadly spread</td>
</tr>
<tr>
<td>OpenId Connect 1.0</td>
<td>no</td>
<td>yes</td>
<td>Experimental/First commercial sites</td>
</tr>
<tr>
<td>UMA 1.0</td>
<td>no</td>
<td>yes</td>
<td>Experimental</td>
</tr>
</tbody>
</table>

**Figure 18 Use Cases – Comparison Table**

<table>
<thead>
<tr>
<th>System</th>
<th>RESTful client</th>
<th>Web Service</th>
<th>Mobile apps</th>
<th>Single Sign-On</th>
<th>Dynamic Identity Federation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenId 2.0</td>
<td>yes</td>
<td>yes</td>
<td>no specific support</td>
<td>yes - if same Provider</td>
<td>yes</td>
</tr>
<tr>
<td>OAuth2.0</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes - if same Provider</td>
<td>no</td>
</tr>
<tr>
<td>OpenId Connect 1.0</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes - if same Provider</td>
<td>yes</td>
</tr>
<tr>
<td>UMA 1.0</td>
<td>yes</td>
<td>not yet</td>
<td>yes</td>
<td>yes - if same Provider</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Figure 19 Security Features – Comparison Table**

<table>
<thead>
<tr>
<th>System</th>
<th>Security Approach</th>
<th>Credential management</th>
<th>Non-repudiation</th>
<th>&quot;Stork&quot; Authentication</th>
<th>Identity anonymity</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenId 2.0</td>
<td>User centric</td>
<td>no</td>
<td>not considered</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>OAuth2.0</td>
<td>User centric</td>
<td>basic</td>
<td>not considered</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>OpenId Connect 1.0</td>
<td>User centric</td>
<td>basic</td>
<td>not considered</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>UMA 1.0</td>
<td>Personal/organisational</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
5. Appendix A – NemID: the Danish Public Sign On and Digital Signature system

NemID is the common Danish system for communicating securely with public institutions as well as private and financial services. It is also used for net banking and for sign-on to a number of commercial services on the web. Particularly NemID can do identity confirmations for logins and digitally sign documents/transactions.

NemID is developed by Nets DanID [NemID 1] under contract to the Danish government (Digitaliseringsstyrelsen).

- Two-factor authentication

Each NemID user has a user ID and a password and a cardboard keycard with one-time PIN codes. Authentication is based on two factors: The user has to demonstrate both knowledge of the password and possession of the key card, by entering one of the one-time pin codes upon request.

![Figure 20 A cardboard key card with one-time pin codes](image)

Login is done through either a Java applet, which is downloaded from and signed by Nets, or a recently implemented JavaScript solution. The user experiences of the two versions are identical, except that the JavaScript version can be run on platforms without java, such as mobile devices or the Chrome browser.

![Figure 21 The NemID login applet](image)
Central key store

NemID is based on PKI technology, and for each user there is a public certificate and a corresponding secret key kept by Nets on a key server with high security. A user can use his secret key to sign data that can then be verified using the certificate, or to decrypt data, which has been encrypted with the certificate. Signing a challenge with your NemID secret key can be used for authentication. The user can also download plug-ins to use with common e-mail programs to communicate securely using NemID. However, very few users use NemID for encryption.

Using NemID

NemID is primarily used by public institutions and banks, but it is also available for commercial service providers. Benefits for service providers include:

1. It is widely adopted by Danes (> 90% has a NemID) and is trusted by the users [NemID 2]. Furthermore, banks, service providers and public institutions are very pleased with the solution.
2. It is based on two-factor-authentication, it is much safer than authentication based only on username and password,
3. The user information is verified by the Danish government and is hence very reliable.

A service provider has to pay a fee per login (1.01 DKK) or a yearly fee per unique user (3.14 DKK) [NemID 3]. For information on how to implement NemID as a service provider can be found on Nets DanID’s web site [NemID 4].

Security and criticism

NemID is a reliable and safe solution, and since it is based on two-factor authentication, it is much safer than authentication based only on username and password. However, it has been the target of some attacks: In 2013, a series of DDoS attacks [NemID 5] targeted towards the server hosting the Java applet caused instability of the system, and there has also been phishing attacks based on fake e-mails claiming to be sent by Nets or the user’s bank that directs users to fake websites to trick the user into giving away his credentials [NemID 6]. There has also been some criticism of the architecture of the system, including the use of Java as the user’s platform [NemID 7], and that the Java applet used for authentication requires full permission [NemID 8].

Nets has targeted much of this critique, by implementing measures to avoid DDoS attacks in the future and replacing the Java applet with a JavaScript solution, and much of the critique is hence not relevant anymore.
6. Appendix B - Definition of key terms

One of the main problems in the field of digital identity has been the lack of agreed terminology.

Therefore, for the scope of this report, we decided to adopt some of the definitions presented in a standard [ISO/IEC 1] recently published.

- **Entity**: Item inside or outside an information and communication technology system, such as a person, an organization, a device, a subsystem, or a group of such items that has recognizably distinct existence.

- **Resource owner**\(^6\): An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

- **Identity**: Set of attributes related to an entity.

- **Attribute**: Characteristic or property of an entity that can be used to describe its state, appearance, or other aspects.

- **Identity information**: Set of values of attributes optionally with any associated metadata in an identity. This is the information pertaining to a particular entity in a domain.

- **Domain**: Environment where an entity can use a set of attributes for identification and other purposes.

- **Resource server**\(^7\): The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

- **Identifier**: Identity information that unambiguously distinguishes one entity from another one in a given domain.

- **Identity management**: Processes and policies involved in managing the lifecycle and value, type and optional metadata of attributes in identities known in a particular domain.

- **Enrolment**: Process to make an entity known within a particular domain.

- **Identity register**: Repository of identities for different entities.

- **Verification**: Process to determine that presented identity information associated with a particular entity is applicable for the entity to be recognized in a particular domain at some point in time.

- **Identity registration**: Process of recording an entity’s identity information in an identity register.

\(^6\) Definition provided by RFC 6749 – The OAuth Authorization Framework

\(^7\) Definition provided by RFC 6749 – The OAuth Authorization Framework
• **Identification**: Process of recognizing an *entity* in a particular *domain* as distinct from other *entities*.

• **Identity assurance**: Level of assurance in the result of *identification*. This concept expresses the level of confidence in provenance, integrity and applicability of *identity information* and its maintenance.

• **Identity proofing / Initial entity authentication**: Particular form of *authentication* based on *identity evidence* that is performed as the condition for enrolment.

• **Identity evidence**: *Identity information* for an *entity* required for authentication of that *entity*.

• **Credential**: Representation of an *identity*. A *credential* is typically made to facilitate data *authentication* of the *identity information* in the *identity* it represents. A *credential* can be a username, username with a password, a PIN, a smartcard, a token, a fingerprint, a passport, etc.

• **Authentication**: Formalized process of *verification* that, if successful, results in an *authenticated identity* for an *entity*.

• **Authenticated identity**: *Identity information* for an *entity* created to record the result of *authentication*.

• **Authentication level of assurance**: Level of assurance in the result of *authentication*. Successful *authentication* of an *entity* in a *domain*, at a specific level of assurance, gives a *relying party* confidence in the correctness and applicability of the verification result. International Standard ISO/IEC 29115 Error! Reference source not found. specifies four levels of authentication assurance and the criteria and guidelines for achieving each of the levels defined.

• **Relying party**: *Entity* that relies on the *verification* of *identity information* for a particular *entity*.

• **Client**: An application making protected resource requests on behalf of the resource owner and with its authorization.

• **Identity assertion**: Statement by an *identity information authority* used by a *relying party* for *authentication*.

• **Identity information authority**: *Entity* related to a particular *domain* that can make provable statements on the validity and/or correctness of one or more *attribute* values in an *identity*.

• **Identity federation**: Agreement between two or more *domains* specifying how *identity information* will be exchanged and managed for cross-domain identification purposes.

---

8 Definition provided by RFC 6749 – The OAuth Authorization Framework
• **Single-sign-on identity**: Identity that includes a single identity assertion that can be verified by a relying party in multiple domains.

• **Authorization**: Process by which a system determines what level of access should have a particular authenticated entity to a secured resource controlled by the system.

• **Authorization server**: The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

It is important to notice that an entity may have multiple identities, each identity relating to at least one domain.

---

9 Given that this concept was not introduced in the standard Error! Reference source not found., this is the definition adopted by the authors of this article.

10 Definition provided by RFC 6749 – The OAuth Authorization Framework
7. Bibliography


[ENISA 1] “Mapping security services to authentication levels”, S. Górniak et al., ENISA, March 2011.


[NemID 3] http://www.nets.eu/dk-da/Produkter/Sikkerhed/NemID-tjenesteudbyder/Pages/Priser.aspx (in Danish)


[OpenID 1] "OpenID Authentication 2.0 – Final", specs@OpenID.net, 5 December 2007, http://openid.net/specs/openid-authentication-2_0.html


[OpenID Connect 1] "OpenID Connect Core 1.0", specs@OpenID.net, 25 February 2014, http://openid.net/specs/openid-connect-core-1_0.html

[OpenID Connect 2] "OpenID Connect Basic Client Profile 1.0" - draft 28 (work in progress), specs@OpenID.net, 5 July 2013, http://openid.net/specs/openid-connect-basic-1_0.html

[OpenID Connect 3] "OpenID Connect Implicit Client Profile 1.0" - draft 11 (work in progress), specs@OpenID.net, 5 July 2013, http://openid.net/specs/openid-connect-implicit-1_0.html

[OpenID Connect 4] "OpenID Connect Discovery 1.0", specs@OpenID.net, 25 February 2014, http://openid.net/specs/openid-connect-discovery-1_0.html

[OpenID Connect 5] "OpenID Connect Dynamic Client Registration 1.0", specs@OpenID.net, 25 February 2014, http://openid.net/specs/openid-connect-registration-1_0.html

[OpenID Connect 6] "OpenID Connect Session Management 1.0" - draft 19 (work in progress), specs@OpenID.net, 25 February 2014, http://openid.net/specs/openid-connect-session-1_0.html


[OpenID Connect 9] "OpenID Connect Messages 1.0" - draft 20 (work in progress), specs@OpenID.net, 5 July 2013, http://openid.net/specs/openid-connect-messages-1_0-20.html


