A Secure Token-Based Communication for Authentication and Authorization Servers

Conference Paper *in* Lecture Notes in Computer Science · November 2016

DOI: 10.1007/978-3-319-48057-2_17

4 authors, including:

Christian Huber
SAL Silicon Austria Labs
11 PUBLICATIONS **39** CITATIONS
SEE PROFILE

Markus Jäger
Pro2Future GmbH
35 PUBLICATIONS **111** CITATIONS
SEE PROFILE

Josef Küng
Johannes Kepler University Linz
180 PUBLICATIONS **620** CITATIONS
SEE PROFILE

A secure Token-based Communication for Authentication and Authorization Servers

Jan Kubovy[2], Christian Huber[1](✉), Markus Jäger[1], and Josef Küng[1]

[1] Institute for Application Oriented Knowledge Processing (FAW)
Faculty of Engineering and Natural Sciences (TNF)
Johannes Kepler University (JKU), Linz, Austria
{chuber, mjaeger, jkueng}@faw.jku.at,
WWW home page: http://www.faw.jku.at
[2] Informations- u. Prozesstechnik, Anwendungen, Eigenentwicklungen
Stadtwerke München GmbH, München, Germany
kubovy.jan@swm.de,
WWW home page: https://www.swm.de

Abstract. Today, software projects often have several independent subsystems which provide resources to clients. To protect all subsystems from unauthorized access, the mechanisms proposed in the OAuth2.0 framework and the OpenID standard are often used. The communication between the servers, described in the OAuth2.0 framework, must be encrypted. Usually, this is achieved using Transport Layer Security (TLS), but administrators can forget to activate this protocol in the server configuration. This makes the whole system vulnerable. Neither the developer, nor the user of the system is able to check whether the communication between servers is safe. This paper presents a way to ensure secure communication between authentication-, authorization-, and resource servers without relying in on a correct server configuration. For this purpose, this paper introduces an additional encryption of the transmitted tokens to secure the transmission independently from the server configuration. Further this paper introduces the Central Authentication & Authorization System (CAAS), an implementation of the OpenID standard and the OAuth2.0 framework that uses the token encryption presented in this paper.

Keywords: OpenID, OAuth2.0, security, authentication, authorization, token, encryption

1 Introduction

This work is based on and partially implements *OpenID* [1] and *OAuth2.0* [2]. It presents a cloud environment with several independent actors, clients, resource-, authorization- and authentication servers. Within the system, a user can access data and services from resource servers. To obtain access, the user has to prove his identity to the authentication server. Further, the authorization server has to confirm that this specific user has permission to access the resource.

The OAuth2.0 framework requires secure communication between the involved parties. Otherwise an attacker could gain access to secured resources. To prevent this, a cryptographic protocol has to be enabled by the server administrator, explicitly [2, 3]. According to the *SSL Pulse* project, most Internet sites use no or inadequate security protocols [4]. A user may verify that the connection of his client is secure, but he can not check whether the communication between the servers is secure. Therefore, our approach introduces an additional encryption of the sensitive information on the application level.

This paper starts by discussing some related work in section 2. In section 3 a common definition and explanation of tokens, different kinds of tokens and their attributes follows. Section 4 introduces an implementation of a security framework and its actors. Afterwards, section 5 discusses some major use cases of this implementation and how a faulty server configuration can be exploited. A solution to this problem is presented in section 6 by illustrating a major use case that uses encrypted tokens. Next, section 7 presents the introduced framework as a security layer in an agricultural environment. Finally, the paper concludes the results of this work in section 8.

2 Related work

In the publication [5] the authors analyze the *Oauth2.0* framework regarding possible security problems. Among others, they could identify several possible attacks which exploit an insufficient transport layer encryption (TLS). Unfortunately, the authors did not present a solution to this problem.

To reduce the harm of such attacks, [6] suggests to reduce the scope associated with an access token and to use a short expiration time for access tokens. However, the document does not provide an alternative to the TLS .

The *OpenID Connect* specification provides an identity authentication for OAuth2.0 systems. Although, OpenID Connect allows to use encrypted tokens for the user authentication [7], it does not define the communication for the authorization. As a consequence, the OAuth2.0 system that uses the OpenID Connect, may use plaintext tokens to authorize resource accesses.

In contrast to the OpenID Connect, our approach enforces encrypted token not only for the authentication of users but also for the authorization of resource accesses.

Similar Open-Source projects like *Apache Oltu* [8], *Restlet Framework* [9] and *APIs* [10] do not address the issue of weakly secured server communications.

An alternative approach that could be adopted for secure token communication is the blockchain technology. Blockchain technology is a decentralized approach. It was developed for transactions of electronic money within the *Bitcoin* network [11, 12]. The downside of this technology is the increased implementation effort for new clients in the cloud. Further network traffic in this method would be higher, because every client additionally has to broadcast its token transactions to the so called minors.

3 Tokens

Tokens are often hardware-items for identifying and authenticating user. They also can be software-based artifacts of permission granting systems, where multi-path authentication algorithms are used. In this work, the token concept of RFC 6750 [13] is used and implemented with the basic authentication concept of RFC 2617 [14]. Typically tokens are shared between multiple components, so every token has to be unique in its usage, autonomous and should not be repeated. Tokens themselves contain only implicit information – they carry information about the owner of the token (e.g. the user it was created for), the purpose of the token (e.g. for a user authentication), how long the token is valid etc.

Referring to the RFCs, the properties of different types of tokens are shown in table 1:

perishable_token: is used to validate a single action; afterwards the token will be invalidated; it is secret and must be transferred via secure communication.

session_token: is valid for one specific session and can be used several times within this session; valid time span has to be renewed if token was used; session token must be transmitted over TLS.

access_token: can be used multiple times but cannot be renewed; its derivation must be verifiable; its transmission is encrypted.

refresh_token: can be used only once (must be invalidated after its use); like the `access_token` its derivation must be verifiable; the transmission of the token is encrypted.

Type	Valid	Reuse	Renew	New on use
perishable_token	24 h	No	No	No
session_token	720 h	Yes	Yes	No
access_token	1 h	Yes	No	No
refresh_token	∞	No	No	Yes

Table 1. Token types

4 Central Authentication & Authorization System (CAAS)

The Central Authentication & Authorization System (CAAS) is a security framework developed by the *FAW* (Institute for Application Oriented Knowledge Processing). The CAAS environment runs at least one authentication server and one authorization server. These servers handle access privileges for arbitrary users and clients to resources of one or more resource servers.

The authentication server checks the identity of users, whereas the authorization server manages privileges of users. A user uses a user-agent to access resources over a client or directly. Considering a web access, the web browser is a user-agent and the server providing the web page is a client accessing resources. The corresponding resource server asks the authorization server whether the user is authorized to access the resources. Because the authorization server cannot check the identity of the user, it asks the authentication server whether the user is who he claims to be. Therefore, the user has to prove his identity to the authentication server.

4.1 Authentication server (ANS)

An Authentication Server (ANS) is responsible for the user management. Users can create accounts and reset their password. A user account can be in one of the following states: *activated*, *suspended*, *resumed* or *deleted*.

The main purpose of the ANS is to verify the identity of a user to the authorization server as described in the OpenID specification in [1]. Therefore, the server asks the user to prove his identity. Such identity prove can either be a password, a session token, or a perishable token (see section 3). A session token is generated by the ANS and shared with a user that already authenticated himself with a password. This way the user does not have to send his password every time he has to authenticate.

Authenticated users can obtain an `authorization code`, which is basically a perishable token. Using this `authorization code` as described in the OAuth 2.0 standard, the user can provide information to the authorization server which only this user and the ANS know. The authorization server checks the identity of the user by letting the ANS validate the `authorization code`. Therefore, the user does not have to send his password or its session token to the authorization server to prove his identity.

4.2 Authorization server (AZN)

The Authorization Server (AZN) administrates and provides information about which user has permission to access which resource. For this purpose it implements the OAuth2.0 protocol. Therefore, the server manages a so called `ScopeTree` whose nodes represent subjects and objects. In this context a subject can be a user, a client, or a group. An object is a resource or a group of resources provided by a resource server. A subject can have one or more permissions on an object.

The `ScopeTree` fulfills the following conventions:

- All clients must be listed in the `/clients` scope.
- Users must be listed in an `[ans_id]` scope.
- For every user in the system a link in the `/users` scope must exist.

Figure 1 shows an example of a `ScopeTree`. In the `/clients` scope is one client registered, `[client_id]`. For this client the AZN knows the public key and which scope nodes the client or its users are allowed to access. This example tree is paired with one ANS, `[ans_id]`. Therefore, the AZN knows the public key of `[ans_id]`. The AZN updates the user scope automatically when a request refers to a user that was not found in the scope tree. For this purpose, the AZN requests a list of all users from the registered ANS. Both users in the ANS scope have a link pointing to them in the `/users` scope.

Finally, the example lists some objects. Within the `[country_code]` scope is the object node `[object_id]` registered. This object has 2 sub objects, `[sub_object_id_1]` and `[sub_object_id_2]`. If a subject (e.g. `[user_id_1]`) has permissions for `object_id`, it has has implicitly the same permissions on the sub objects. The inherited permissions on a sub object can be extended or removed. Because of this it is possible to create complex permission structures.

```
/
|-- /clients
|    |-- /[client_id]
|-- /users
|    |-- /[user_id_1] [LINK]
|    |    -> /^[ans_id]/[user_id_1]
|    |-- /[user_id_2] [LINK]
|    |    -> /^[ans_id]/[user_id_2]
|-- /[ans_id]
|    |-- /[user_id_1]
|    |-- /[user_id_2]
|-- /[country_code]/[object_id]
|    |-- /[sub_object_id_1]
|    |-- /[sub_object_id_2]
| ...
```

Fig. 1. Example `ScopeTree`

Permissions are stored as triples comprising of *subject*, *object* and a permission string. The *SCRUDL* scheme, used for the permission string, distinguishes the following six privileges:

- **(S)EARCH** in a requested scope and its subtree.
- **(C)REATE** a new object or resource in the requested scope.
- **(R)EAD** an existing object (including its attributes) in the requested scope.
- **(U)PDATE** or modify an existing object (including modification, deletion and creation of attributes) in the requested scope.
- **(D)ELETE** an existing object in the requested scope.
- **(L)IST** existing objects in a requested scope but not in its sub trees.

Figure 2 illustrates the format of the permission string. Every privilege has its fixed position in the string. If the letter representing the privilege is present,

the privilege is granted. A '.' character instead of the letter means that the corresponding privilege of the parent scope is inherited. The '-' character denies the privilege it is representing, even if the privilege was inherited from the parent node. Example:

[..RU.-]: grants READ and UPDATE permission, SEARCH, CREATE, DELETE remain unchanged and the LIST permission is denied

[.C-.D.]: grants CREATE and DELETE permission, SEARCH, UPDATE, LIST remain unchanged and the READ permission is denied

```
permissions := ('S' | '.' | '-')('C' | '.' | '-')
               ('R' | '.' | '-')('U' | '.' | '-')
               ('D' | '.' | '-')('L' | '.' | '-')
```

Fig. 2. Permission format

The example in figure 3 illustrates a permission definition that belongs to the ScopeTree in figure 1. In the example the subject refers to the scope node /[ans_id]/[user_id_1] and the object refers to the scope node /[country_code]/[object_id]. As a consequence, the authorization server grants READ and UPDATE privileges for the user [user_id_1] to the object [object_id].

Since there is no permission definition for the child nodes, the user inherits there READ and UPDATE privileges for the child nodes.

```
Subject: /[ans_id]/[user_id_1]
Object: /[country_code]/[object_id]
Permissions: [..RU..]
```

Fig. 3. Permission example

5 Accessing resources

This section explains some major work flows of the CAAS based on OAuth2.0 framework [2].

5.1 Obtaining permissions

To obtain an authorization code, a User-Agent has to send an Authorization Request as illustrated in figure 4 (a). Because the User-Agent is not authenticated, the AZN replies an Unauthorized Response (b). This response contains

an URL to the ANS where the user should authenticate himself. After the user is successfully authenticated (c), the ANS responses with an `Authentication Response` that contains a perishable token (d). With this token the `User-Agent` can send an `Approved Authorization Request` (e). The AZN forwards the token to the ANS (f), which validates the token (afterwards the token is invalid for further requests) and replies an `Authentication Response` (g). At this point the AZN knows the identity of the user and sends back an `Authorization Code Response` if the user is allowed to access the resource (h). This `Authorization Code Response` contains a perishable token that is used to obtain an access token and a refresh token.

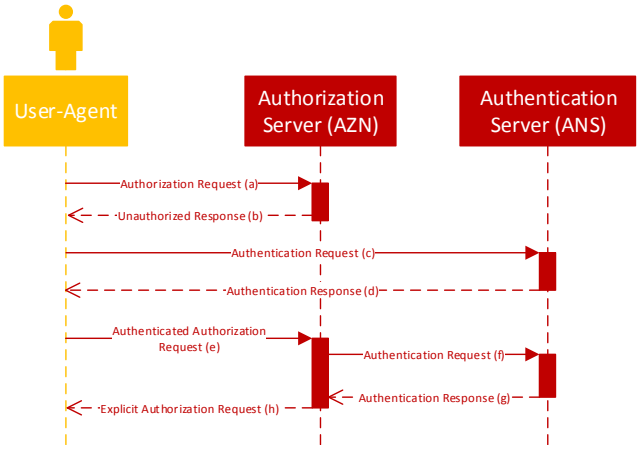

Fig. 4. User Authorization Flow [2]

5.2 Permission delegation

The permission delegation concept is based on the usage of bearer tokens as described in [13].

Figure 5 shows the process of obtaining and using access tokens and refresh tokens. The process starts with the `User-Agent` requesting a service from a `Client` (a). An `Authorization Challenge` responded by the `Client` contains a list of all scope permissions the `Client` needs to provide the service, as well as an URL to the responsible AZN (b). Using this `Authorization Challenge` the `User-Agent` obtains access to the required resources as demonstrated in subsection 5.1 (c-g).

The authorization code from the AZN is redirected as a Bearer token to the `Client` (h) and then forwarded back to the AZN (i). This way the user delegates his permissions on the resource to the `Client`. After validating the authorization

code, the code is invalidated and the AZN replies with an access token and a refresh token (j). These tokens are used as bearer tokens to authorize resource access to the `Client` [13]. According to [2] the access token can be leaked, for example, by delegating it to a malicious `Client` (e.g. due to phishing). Therefore, the access token expires after a specified time to limit the harm of a stolen access token. Until the access token is expired, the `Client` can use it to access the resource (k, l). If the access token expires, the `Client` can request a new one from the AZN by providing the refresh token (m, n). The use of refresh tokens is described as optional in [2]. In case the refresh token is invalid, the `Client` proceeds with sending the `Authentication Challenge` to the `User-Agent` again (b).

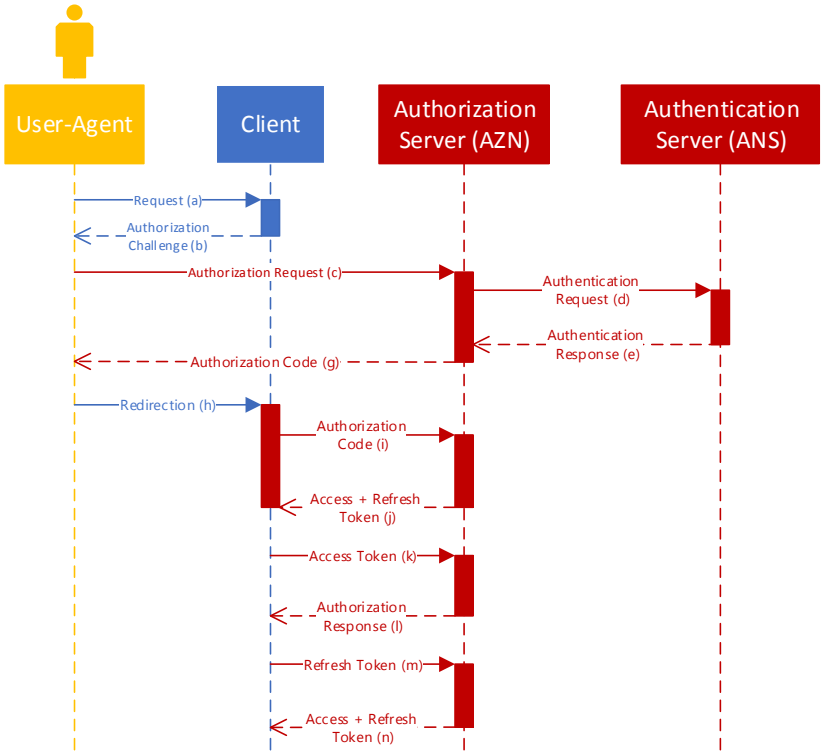

Fig. 5. Access/Refresh Token Sequence Flow

This flow relies on secure communication between the single components. Although, the user can ensure that the communication between the `User-Agent` and other components is secured by the TLS protocol, an unsafe communication

between `Clients`, AZNs and ANSs would be unrecognized. This would allow an attacker to steal the access token and get access to the requested resource.

5.3 Resource access

This subsection extends the example in subsection 5.2 where the `Clients` require some resources from one or more `Resource Servers` to fulfill their task. Figure 6 illustrates this work flow. After the `User-Agent` sent the request (a), the `Client` tries to request the necessary resource (b). The `Client` does not need to try to request resources for which it has no authorization. Because the request (c) is not authorized, the AZN and further the `Resource Server` deny the access (d, e). Therefore, the `Client` returns an `Authorization Challenge` to the `User-Agent` (f). Like in figure 5, the `User-Agent` obtains an `Authorization Code` (g-i) which is used by the `Client` to get an access token from the AZN
(i, k). Now, the `Client` can request the resource by use of the access token (l). The `Resource Server` uses the access token to obtain an `Access Granted`

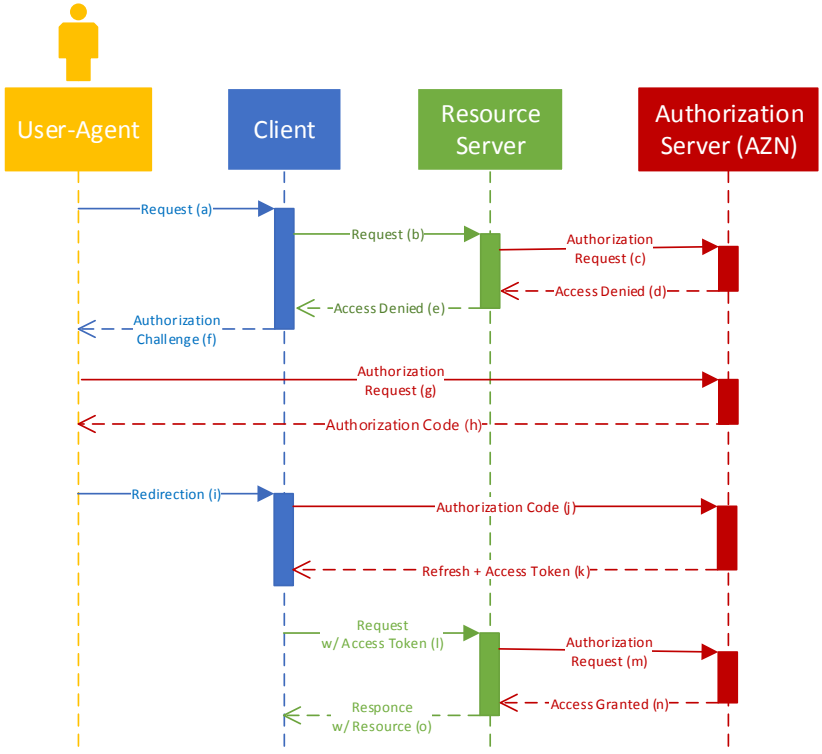

Fig. 6. Resource Access Sequence Flow

response from the AZN (m). After the `Resource Server` received the `Access Grant` (m), he returns the resource to the `Client` (o).

6 Secure communication

In the previous sections, this paper presented the CAAS, its components and how they communicate with each other in token based manner. Further, a possible vulnerability of such implementations of `OAuth2.0` and `OpenID` have been discussed. This section presents a solution to this problem.

By encrypting and/or signing the tokens before every transmission, the system is able to ensure secure communication between the single components independent from environmental influences like the server configuration. To demonstrate this procedure, figure 7 illustrates the complete authorization flow from the initial user request to the final resource response.

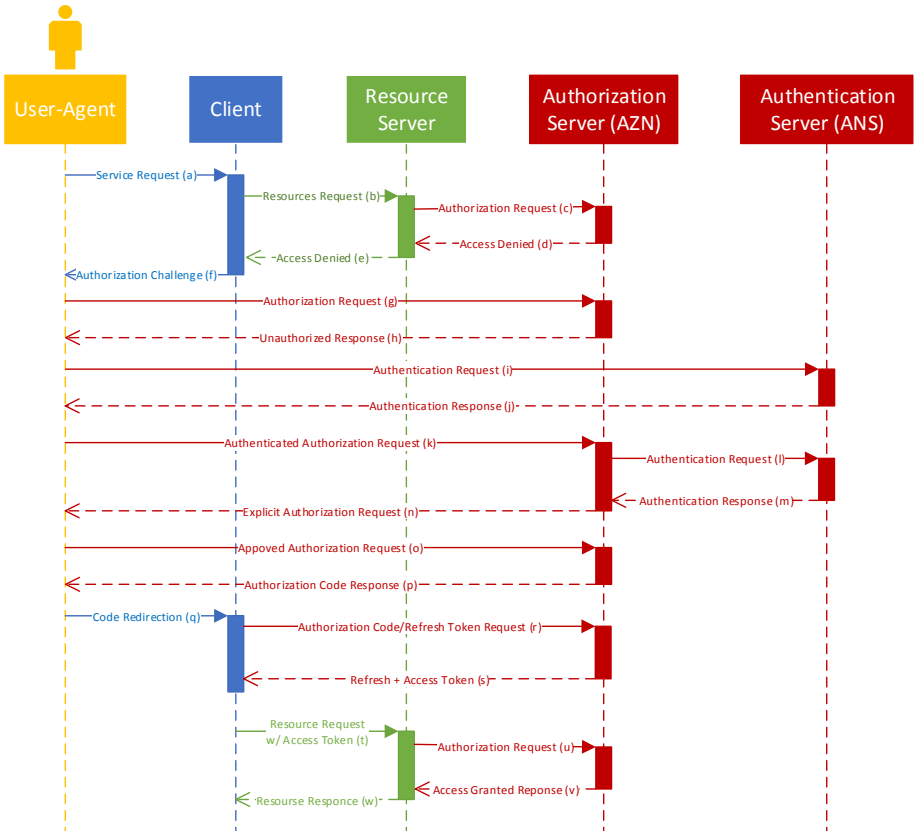

Fig. 7. Complete delegation flow[2].

(a) The user requests a service provided by a `Client`. If the `Client` needs resources from one or more `Resource Servers` and has access tokens for those resources it will try to fetch those.

(b) The `Client` asks one or more `Resource Servers` for the resources it needs for its job. If the `Client` has no authorization information it will skip the steps (c) and (d).

(c) The `Resource Server` asks the `Authorization Server` if it can provide the requested resources to the `Client`.

(d) At this point, the `Authorization Server` denies the request (invalid/expired access token or no access token was provided).

(e) The `Resource Server` forwards the response to the `Client`.

(f) The `Client` issues an `Authorization Challenge` and returns it to the user. Such `Authorization Challenge` must contain the `client_id` and a list of requested scope nodes.

(g) The user forwards this to the `Authorization Server`. This can happen automatically using redirects [14].

(h) Since the user is not even authenticated yet, the `Authorization Server` replies with another `Authorization Challenge` but for an `Authentication Server`. It uses the `client_id` of the `Authorization Server` in this `Authorization Challenge`.

(i) The user forwards this to the `Authentication Server`, with his credentials. Those can be user_id/password, session token, etc. There will be at least 2 rounds between the `Authentication Server` and the user. First the user will get an `Unauthorized` response. In the next round, he provides his credentials. Before sending the credentials, the user can check if his browser is connected using the TLS protocol with an valid certificate. This way, the user ensures that he does not disclose his credentials.

(j) In case the second round will pass, the `Authentication Server` issues a specific access code for the `Authorization Server` and encrypts it with the `Authorization Server`'s public key. Therefore, only the `Authorization Server` can use this token to validate the identity of the user.

(k) The user forwards this encrypted access code to the `Authorization Server`, which decrypts it, signs it and uses it to obtain an access token and a refresh token from the `Authentication Server`. These tokens are transmitted encrypted with the `Authorization Server`'s public key as the access code before.

(l) The user authenticates himself to the `Authorization Server` directly using the token the `Authentication Server` issued to the user. The `Authorization Server` will then be able to check with the `Authentication Server` if the user is still valid and the authentication was not revoked by using the access token/refresh token issued to it by the `Authentication Server`. For the transmission of the access token/refresh token, these tokens must be encrypted by the `Authentication Server`'s public key. Therefore, the `Authentication Server` an `Authorization Server` can be sure the tokens are not leak to a third party.

(m) The `Authentication Server` confirms or denies the request. For the confirmation, the server encrypts the already encrypted access token with its private key and returns it to the `Authorization Server`. This way, the `Authorization Server` can verify that the request confirmation was issued by the `Authentication Server`.

(n) The `Authorization Server` checks the original `Authorization Challenge` and renders a confirmation page to the user, where he reviews the permission delegation. It issues a perishable token with this response, which should be used to approve the request.

(o) If the user approves, he does so by sending the generated perishable token with the approved scope node back to the `Authorization Server`. Before sending the perishable token, the user can check if his browser is connected using the TLS protocol with an valid certificate. This way, the user ensures that he does not send the token to a third party in case of a Man-in-the-middle attack.

(p) If the perishable token was valid the `Authorization Server` issues an access code for the `Client` and encrypts it with the `Client`'s public key. Therefore, only the `Client` is able request an access token using this code.

(q) The user forwards the encrypted access code to the `Client`.

(r) The `Client` decrypts the access code and uses it to obtain an access token and a refresh token directly from the `Authorization Server`. For this request, the access code must be encrypted with the `Authorization Server`'s public key. Therefore, the `Client` an `Authorization Server` can be sure the token is not leak to a third party.

(s) The `Authorization Server` issues a new access token and refresh token, encrypts both with the `Client`'s public key and sends them to the `Client`. Only the `Client` able to use these token to request a resource.

(t) The `Client` uses the access token encrypts with its `Authorization Server`'s public key, every time it needs to access the given `Resource Server`. This

way, the `Authorization Server` can verify that the access token was was not leak to a third party.

(u) The `Resource Server` uses this signed access token to check with the `Authorization Server` if token is valid.

(v) The `Authorization Server` confirms or denies the request. For the confirmation, the server encrypts the already encrypted access token with its private key and returns it to the `Resource Server`. This way, the `Resource Server` can verify that the request confirmation was issued by the `Authorization Server`.

(w) The `Resource Server` makes the requested resources available to the `Client`.

In this work, the asymmetric Rivest-Shamir-Adleman (RSA) cryptosystem is used to prevent insecure token transmissions due to a faulty server configuration [15]. For the encryption we suggest keys with a minimum bit-length of 3072-bit. According to the NIST this key size is equivalent to a 128-bit symmetric key, which is acceptable for usage beyond 2030 [16].

7 Application in the agricultural domain

The CLAFIS (Crop, Livestock and Forests Integrated System for Intelligent Automation) project [17] is a research project, involving 12 partners from different countries and diversified fields of excellence, funded by the European Union. The project's goal is, to support people, who are working in the agricultural domain. On every step, the farmer should be assisted by modern technologies, via automation and analyzing data for predicting the agricultural returns. There are several sensors used, which provide real time data into the CLAFIS cloud for the knowledge processing system. With the results, e.g. regulation steps for machinery can be generated or predictions of the current disease pressure. Those results can be accessed by the farmer live via mobile devices.

There are many components, which have to communicate with each other, that a system like CLAFIS is able to work properly. In this context, security is a very important topic. On the one hand, valuable data must not be accessed by unauthorized parties, on the other hand, each component of the system must be able to trust every other component. The CAAS was designed to ensure these requirements in the CLAFIS.

For example: the Disease Pressure Model (DPM – calculation model for predicting the risk of the outbreak of a disease on a specific field), is working with data from several external systems and sensors. The implementation of the DPM is already secured by CAAS. Therefore, every component has to be authorized before it is able to access the DPM. Furthermore the components can trust on the certainty of the values provided by the DPM.

8 Conclusion

This paper presented the token-based authentication and authorization system, CAAS that implements the `OpenId` platform [1] and the `OAuth2.0` framework [2].

Afterwards, it showed the major use cases and discussed the importance of the Transport Layer Security (TLS) [18] for `OAuth2.0` systems. This leads to the result that a false configured server reveals a critical security flaw. During operation it is not possible for the user or the security framework to determine the missing security layer.

Therefore, a token encryption was introduced that secures the tokens and furthermore the resources of the system even if the TLS protocol is not activated. This mechanism prevents `OAuth2.0` based system from being exploited due to a deployment failure.

Finally, the use of the introduced CAAS and the token encryption within an agricultural domain was stated and the importance of secure authentication and authorization scheme was outlined.

Acknowledgement

This work was done as part of several projects by the authors during their stay at the Institute for Application Oriented Knowledge Processing at the Johannes Kepler University in Linz.

The research leading to these results has partly received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no.604659.

References

1. Recordon, D., Reed, D.: OpenID 2.0: A Platform for User-centric Identity Management. In: Proceedings of the Second ACM Workshop on Digital Identity Management. DIM '06, New York, NY, USA, ACM (2006) 11–16
2. Hardt, D.: The OAuth 2.0 Authorization Framework. RFC 6749, RFC Editor (October 2012)
3. The Apache Software Foundation: SSL/TLS Configuration HOW-TO. `https://tomcat.apache.org/tomcat-8.0-doc/ssl-howto.html#Introduction_to_SSL` (2016) Accessed: 3 Sep 2016.
4. Trustworthy Internet Movement: SSL Pulse - Survey of the SSL Implementation of the Most Popular Web Sites. `https://www.trustworthyinternet.org/ssl-pulse` Accessed: 3 Sep 2016.
5. Yang, F., Manoharan, S.: A security analysis of the OAuth protocol. In: Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on. (Aug 2013) 271–276
6. Lodderstedt, T., McGloin, M., Hunt, P.: OAuth 2.0 Threat Model and Security Considerations. RFC 6819, RFC Editor (January 2013)
7. Sakimura, N., Bradley, J., Jones, M.B., de Medeiros, B., Mortimore, C.: OpenID Connect Core 1.0. The OpenID Foundation (2014) S3

8. The Apache Software Foundation: Apache Oltu: An OAuth Open Source framework. https://cwiki.apache.org/confluence/display/OLTU/Index (2013) Accessed: 3 Sep 2016.
9. Restlet Inc.: RestLet Framework. https://restlet.com/technical-resources/restlet-framework/guide/2.3/extensions/oauth (2016) Accessed: 3 Sep 2016.
10. Harsta, O.: OAuth-Apis: OAuth Authorization as a Service. https://github.com/OAuth-Apis/apis (2012-2016) Accessed: 3 Sep 2016.
11. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf (2008) Accessed: 3 Sep 2016.
12. Travis, P.: The Bitcoin Revolution: An Internet of Money. Travis Patron (2015) Accessed: 3 Sep 2016.
13. Jones, M.B., Hardt, D.: The OAuth 2.0 Authorization Framework: Bearer Token Usage. RFC 6750, RFC Editor (October 2012)
14. Franks, J., Hallam-Baker, P.M., Hostetler, J.L., Lawrence, S.D., Leach, P.J., Luotonen, A., Stewart, L.C.: HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, RFC Editor (June 1999)
15. RSA Security: Information Security, Governance, Risk, and Compliance - EMC. http://www.rsa.com (2014) Accessed: 3 Sep 2016.
16. Barker, E., Barker, W., Burr, W., Polk, T., Smid, M., Zieglar, L.: NIST Special Publication 800-57 Revision 4 Recommendation for Key Management Part 1: General. http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4 (2016)
17. CLAFIS Project: CLAFIS: Crop, livestock and forests integrated system for intelligent automation. http://www.clafis-project.eu (2013-2016) EU Seventh Framework Programme NMP.2013.3.0-2.
18. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor (August 2008)