Attribute-Based Encryption for Access Control in (Real World) Cloud Ecosystems

G. Bartolomeo, CNIT

Abstract— We introduce a distributed, fine-granuled, policy-based resource access control protocol leveraging on Attribute-Based Encryption and OpenID Connect. We show how the resulting protocol may simplify and secure the whole access control procedure from the authorization issuer to the resource server, natively providing the desired properties of confidentiality, integrity, proof of possession and antiforgery.

1 Introduction

REAL-world Cloud ecosystems massively use token-based authorization. One of the most popular token format is JSON Web Tokens (JWT), often used in conjunction with OAuth 2.0 and OpenID Connect (OIDC). However, many security aspects are still under discussion. This paper proposes the introduction of Attribute-Based Encryption (ABE), and shows how this choice may streamline OIDC, achieving a very simple and secure access control protocol.

2 Related Works

The original idea behind OAuth 2.0 and its derived specification OIDC is a simple triangulation between a Client, an Authorization Server and a Resource Server implemented on top of the HTTP(S) protocol. After trying to access a resource, the Client is redirected to the Authorization Server, who prompts the user for login & consent. Assumed the user logs in and grants the proper authorization, the server issues a successfully authentication response and redirects to a client provided URI, appending the issued token as a parameter.

This simple flow, known as "Implicit Grant", is known to be vulnerable to token leakage, therefore a slightly more complex flow named "Authorization Code Flow" is used in real world cases. In this latter, a third-party provider registers its Client, obtaining a "Client-id" and a "Secret". When a user needs to be identified, the Client generates a unique session token ("State"). After establishing a TLS server authentication connection with the Authorization Server, the Client sends a HTTP GET request specifying the resource to be accessed ("Scope"), the redirection URL of the server that will receive the response, an anti-forgery session token ("State"), and a Nonce to protect the server against replay attacks. After the user performs login & consent with the Authorization server, the Client is sent back to the redirection URL by a HTTP GET request which includes the "State" parameter, plus a "Code" parameter, which is a one-time authorization code later exchanged for an "ID token" (i.e., a token containing the requested user information) and an the access token (the actual authorization credentials). This exchange happens through a HTTP POST request which includes also the "Client-id" and the "Secret" preassigned to the client application. The HTTP POST response contains a signed ID token and signed access token embedding the Nonce to prevent reply attacks. The Client finally retrieves user profile information from the ID token and may present the access token to the Resource Server in a subsequent API call, by including it in the HTTP "Authorization" request header. As this flow does not provide the Resource Server with any assurance on whether the token presenter is the legitimate client or an attacker, recent RFC 7800 "Proof-of-Possession Key Semantics for JSON Web Tokens" describes a method allowing a Client to present a cryptographically verifiable "proof-of-possession" together with the token.

3 Our Formal Analysis

A very comprehensive formal model for OIDC is FKS[1]. The model was successfully used to discover several attacks. However, it does not allow automation, requiring manual theorem proofs. For our investigations, instead, we used the symbolic model-checker OFMC[2] and constructed our model by relying on only few kinds of channels, abstracting some properties of the HTTP(S) protocol:

- [A] •→• B denotes a "secure" (confidential and weakly authentic) channel between [A] and B, i.e., [A] can be sure that only B can read the message, while B can be sure it comes from [A]. Symbol [A] is used instead of A to mean that A does not disclose her real identity but a pseudonym [A] staying constant during the session (which is the case for HTTPS without mutual authentication).
- A → B denotes an unencrypted, unauthenticated channel between the A and B, such as plain HTTP is[3].

A number of attacks may come from web attackers, i.e. attackers who can listen to and send messages from inside a browser – including those performed through XSS – but

[1] D. Fett, R. Küsters, and G. Schmitz: The web sso standard OpenID Connect: In-depth formal security analysis and security guidelines. In 2017 IEEE 30th Computer Security Foundations Symposium (CSF), pp. 189–202, Aug 2017.

[2] The reader may refer to OFMC overview and manual for details. Available at https://www.imm.dtu.dk/~samo/OFMC-tutorial.pdf last accessed on 26th May 2023.

[3] Other alternatives are possible, and we refer to the OFMC manual for further details on channel composition

cannot listen to traffic intended for other processes and cannot spoof their own address. On the contrary, OFMC is based on the classical Dolev-Yao intruder model assumptions where the attacker can intercept, eavesdrop and forge any message (without, however, being able to break cryptography). While the Dolev-Yao model is not able to capture all the complexity of a complete and subtle formal model such as FKS, we can still model the attacker as effectively able to operate from inside the user's device, by relaxing assuntions on some channels and modelling them as unsecure, in order to capture at least some (yet unspecific) web attacks. Thus, we assume that, even if HTTPS is actually used, some steps in the model may use unsecure channels, allowing the attacker to read and alter some URL parameters, redirect the browser to websites other than the intended one, perform CSRFs, etc. We also leave out from our model user's login and consent. Clearly, the model does not capture specific web attacks (e.g., exploiting a wrong use of redirect messages without stripping sensitive parameters) nor attacks on end user interfaces (e.g., brute force against their password) and may even lead to unfeasible or unrealistic attacks (that we will discard). Nevertheless, it may give a useful insight of the properties of an Authorization Code flow.

Our model specifies the following actions: after a Client (C) connects to a Resource Provider (RS) in order to get access to a given resource (Scope), it is returned a session parameter (e.g., in a cookie) and redirected to an Authorization Server (as). For the sake of simplicity, we assume this latter is constant and cannot be changed by the user, i.e. a single Identity Provider is used[4]. The Client connects to the Authorization Server passing Scope, State and Nonce parameters (authorization request). After user login & consent (not modeled), the Client is returned an authorization response, containing a token, which is a trapdoor function of passed parameters: code(Scope,State,Nonce). Next, the Client exchanges the code for a token at the token endpoint (part of the Authorization Server) using a secure channel. The token is digitally signed by the Authorization Server and contains the list of resources the user has authorized access to. Finally, the Client goes back to the Resource Provider to spend the token.

It is very relevant to look at the token model. The returned token is indeed a function of the code, which, in turn is a function of the original Scope, State and Nonce parameters. The token thus contains the resources the user has authorized access to and the Nonce parameter. However, while the Client *knows in advance* the Nonce parameter, but has no information on which resources the user has authorized access to, the Resource Server *does not know in advance any of these*. It is blind to this information: when the token is presented, the Resource Server does not know whether and which resources the user has authorized access to, nor whether the Nonce value is correct, even if this information is *written* in the token[5]. The Resource Server limits to trust the token signature.

As regarding desired security objectives, we specify the following:
i) The Resource Server must authenticate the Client using, other than the session parameter, the information contained in the token that it is able to acknowledge; ii) Viceversa, the Client authenticates the Resource Server on the specific data it returns, which, furthermore, should be kept confidential; iii) The Client authenticates the Authorization Server authorization response by checking the State parameters and both agreeing on the returned (user authorized) code and Scope[6]; iv) at the token endpoint, the Authorization Server must authenticate the Client via its registered identity and password and the code parameter[7]. To study the effect of parameters leakage and injection (which in real world may happen in several ways), we investigate the effect of having authorization request and response transmitted under unsecure channels[8]. Running the model checker up to 2 simultaneous runs, we found, as expected, the very well-known code injection attack. Without protecting any of the two messages, Client's parameters injection is possible making the Client retrieving a wrong

```
Protocol: OIDC_AuthCodeFlow

Types: Agent    as, #Authorization Server (constant)
                 RS, #Resource Server
                  C; #Client
       Function pw, #shared password between as and C
                pk, #as public key
          resource, #user authorized resource to be accessed
              code, #authorization code
              hash; #PKCE trapdoor function
      Number Scope, #OIDC AuthCodeFlow parameter
             State, #OIDC AuthCodeFlow parameter
             Nonce, #OIDC AuthCodeFlow parameter
              Data, #RS returns this data on successfulauthorization
           Session, #shared session between RS and C
          Verifier; #Verifier used in PKCE

Knowledge:
           as: as,pk(as),inv(pk(as)),C,pw(as,C),code,resource,hash;
           RS: RS,as,pk(as);
            C: C,RS,pk(as),pw(as,C),hash,pk(C),inv(pk(C));
         where RS!=C, RS!=as, C!=as

Actions:
C->RS:Scope
RS*->C:Scope,as,Session

[C]*->*as: RS,Scope,State,Nonce,hash(Verifier)
as->[C]: State,code(Scope,State,Nonce),Scope

[C]*->*as:C,pw(as,C),code(Scope,State,Nonce),Verifier,
         pk(C),{RS,Scope,pk(C)}inv(pk(C)) #DPoP proof
as*->[C]:{resource(code(Scope,State,Nonce)),C,as,RS,Nonce,pk(C)}inv(pk(as)),
          code(Scope,State,Nonce) #this channel is no more confidential

[C]*->*RS:{resource(code(Scope,State,Nonce)),C,as,RS,Nonce,pk(C)}inv(pk(as)),
          Session,{RS,Scope,pk(C)}inv(pk(C))
RS*->*[C]:Data,Session

Goals:
RS authenticates C on RS,resource(code(Scope,State,Nonce)),
                      C,as,Session,{RS,Scope,pk(C)}inv(pk(C))
C authenticates RS on Data
Data secret between RS,C
C authenticates as on State,Scope,code(Scope,State,Nonce)
as weakly authenticates C on C,pw(as,C),code(Scope,State,Nonce)
```

Fig. 1. OpenID Connect Authorization Code Flow implementing Proof Key for Code Exchange (PKCE) and Demonstrating Proof-of-Possession (DPoP), modeled in AnB specifications language.

[4] This implies that the checker will not be able to detect attacks such as Identity Provider Mix-up.

[5] In some implementations, this check is possible by using a back channel between the Resource Provider and the Authorization Server (implementing an "Introspection endpoint"). However, this is not part of the standard OAuth 2.0 protocol.

[6] Note that parameter Scope is returned as well in the authorization response. This is not needed by the OIDC specifications, however it is used in our model to avoid the checker detecting a trivial reply attack with the response message.

[7] The usual reccomandation is that the code parameter is one-time-usable, "short-living" (max 10 minutes lifespan) and "sufficiently" random. In our formal model, we relax this assumption and accept that code parameters may even be occasionally reused.

[8] In fact, while they may be transmitted under HTTPS, the presence of a potential untrusted element in the user agent (e.g., a malicious script in the web browser able to read the browser's URL bar or access the browser's logs) may make these connections formally unsecure.

code. Later, this makes the Client retrieving a different token than the one expected, allowing access to a different resource than the intended one. Noticeably, this happens through a violation of the *third* security goal (not the first one!) because the Resource Server does not formally know whether and which resources the user has authorized access to (hence, no violation of the first goal). If either only the authorization request or the authorization response is protected, a code or Nonce injection is respectively possible. However, assumed the Client properly checks the returned Nonce in the token, the flow alts without any prejudice to the user other than a denial of service.
As an alternative to the use of a Nonce, we explored the effect of introducing PKCE. PKCE is the ability for a Client to send a "Challenge" to the authorization endpoint that the Authorization Server may later verify, through a "Verifier", sent to the Token endpoint. We assume that the Challenge is simply a hash of the Verifier parameter and make the returned authorization code a trapdoor function of it[9]: code(Scope,State,Nonce, hash(Verifier)). This time an attack is only possible when neither the authorization request nor the response is protected. The attacker alters the Client parameter but presents the correct PKCE Challenge to obtaing a wrong code, which is later exchanged for a (wrong) token by the Client[10].

We finally investigated the effects of leaking a token. This clearly affects the *first* goal, as an attacker may immediately use the token to access a protected resource from the Resource Server. Introducing the Proof-of-Possession as described in a popular Internet draft[11], will finally result in a secure flow till 2 simultanous runs. This result holds as long as the presented token signature is over sufficient parameters to make it prevent reply attacks (in the model reported in Figure 1 we included the Client's public key, the Resource Server address, the Scope parameter).

4 Introducing Attribute-Based Encryption

Note that the use of PKCE creates a session between the Authorization Endpoint and the Token Endpoint, by relying on the Client; similarly, a DPoP proof creates a session between the Token Endpoint and the Resource Server. PKCE and DPoP, essentially, exactly solve the same problem: they create a distributed session between the Client, the Authorization Server and the Resource Server, where the semantics specified inside a JWT shall be enforced. Can we redesign a flow by handling over most part of verifications, decisions and enforcement to the cryptographic layer? End-to-end cryptography may be introduced to avoid attacker intrusion between the Authorization Server and the Client. In Identity-Based Encryption (IBE) any party may generate a public key from a known identity value (i.e., an ASCII string or an URI) – assumed they know a "Master Public Key" (MPK) provided by a Key Generator. Private keys are issued by the Generator and shipped to the parties. In our case, assumed the Generator is implemented inside the Authorization Server, each Client may register once with its own URI and receive its Client private key. The Authorization Server may later ship encrypted tokens to any Client, knowing its registration URI. IBE therefore may be used to implement end-to-end encryption between the Authorization Server and the Client. More in general, CiphertextPolicy Attribute-Based Encryption (CP-ABE) enables secret keys to be associated with a set of attributes and ciphertexts being calculated using an "access policy" over attributes. A client can decrypt a ciphertext if there is a "match" between the policy and its own set of attributes embedded in its key. Leveraging on CP-ABE, we assume an Authorization Server is able to execute the ABE set-up algorithm for CP-ABE, generating the corresponding master public key *MPK* and master secret key *MSK*. The Authorization Server is also able to generate client's secret keys based on a set of attributes and to perform CP-ABE encryption. Also, we assume that the Resource Server is able to encrypt data using CP-ABE (i.e., it knows the master public key *MPK* generated by the Authorization Server). Finally, we assume that the Client has received from the Authorization Server – just once at startup, using a secure channel – a Client's key $k=SK_{MSK,\{c\}}$, which is a CP-ABE key generated by the server using the Client's identifier[12] c as a single attribute (IBE-style fashion):

$$S' = \{c\} \tag{1}$$

$$AS \rightarrow Cl\colon k = SK_{MSK,S'} = SK_{MSK,\{c\}} \tag{2}$$

The protocol begins with the Client requesting the Resource Server to access a protected resource r on behalf of an end-user u.

$$Cl \rightarrow RS\colon \{u, r\} \tag{3}$$

where is the user's identifier and is the target resource identifier. The Resource Server generates a secret x for the resource to be accessed and encrypts it using the following access structure A:

$$A = i \wedge c \wedge a \wedge u \wedge r \wedge (t < f) \tag{4}$$

where i (for "issuer") is the identifier of the Authorization Server, a (for "audience") is the identifier associated to the Resource Server, t is a timestamp attribute and f is an expiration time. The resulting ciphertext $\{x\}_{MPK,A}$ is further encrypted to the Client, using A', an access structure made of only one attribute, i.e. the Client's identifier.

$$A' = \{c\} \tag{5}$$

$$z = \{\{x\}_{MPK,A}\}_{MPK,A'} = \{\{x\}_{MPK,A}\}_{MPK,\{c\}} \tag{6}$$

[9] From the specifications: "*Typically, the "code_challenge" and "code_challenge_method" values are stored in encrypted form in the "code" itself but could alternatively be stored on the server associated with the code. The server MUST NOT include the "code_challenge" value in client requests in a form that other entities can extract*".

[10] This scenario is called "Stronger Attacker Model" in Daniel Fett's articole "PKCE vs. Nonce: Equivalent or Not?", https://danielfett.de/2020/05/16/pkce-vs-nonce-equivalent-or-not/ last accessed on 26th December 2021.

[11] "OAuth 2.0 Demonstrating Proof-of-Possession", Internet draft, https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop last accessed on 26th May 2023.

[12] In practice, however, a second temporal attribute is used inside the access structure, so that the Client's key may be periodically – e.g. weekly, daily or hourly – renewed in order to improve security.

The ciphertext z and the identifier of the Authorization Server i are returned to the Client.

$$RS \rightarrow Cl: \{z, i\} \tag{7}$$

The Client issues a nonce m and redirects to the Authorization Server.

$$Cl \rightarrow AS: \{c, a, r, m\} \tag{8}$$

Following, the user is prompted to authenticate with the Authorization Server through any supported method and may authorize (or partially authorize) the Client's request ("login & consent" procedure). As a result of this authorization, the Authorization Server generates a corresponding CP-ABE secret key $e{=}SK_{MSK,S}$ (henceforth ephemeral key) from the following set of attributes

$$S = \{i, c, a, u, r, r', \ldots, r^n, t\} \tag{9}$$

where $r, r', \ldots, r^n$ are n different identifiers associated to the resource(s) the user has authorized access (should include r), and t is the timestamp attribute. Finally, the key e and the nonce m are encrypted to the Client and the ciphertext p is returned:

$$AS \rightarrow Cl: p = \{e, m\}_{MPK,A'} = \{e, m\}_{MPK,\{c\}} \tag{10}$$

Using its key k, the Client can now decrypt this ciphertext and obtain the nonce m and ephemeral key e.

$$\{e, m\} = \{p\}_k^{-1} \tag{11}$$

After checking the nonce m is correct, owing both its own key $k{=}SK_{MSK,\{c\}}$ and the ephemeral key $e{=}SK_{MSK,S}$, the Client can finally decrypt the secret:

$$x = \{\{z\}_k^{-1}\}_e^{-1} \tag{12}$$

Note that, thanks to the native end-to-end encryption, all steps right now do expose parameters under the attacker's control and do not need any security measure on the transmission channel (except ensuring the authenticity of the Challenge), making the flow suitable for redirection-based protocols like OIDC.

As a last step, the Client repeats the original request to the Resource Server, this time presenting the decrypted secret x:

$$Cl \rightarrow RS: \{u, r, c, x\} \tag{13}$$

The Resource Server checks the secret presented by the Client, and, in case of a positive match, grants access to the requested resource[13].

Checking the above protocol with OFMC (Figure 2), the model specifies the following goals: i) The Resource Server must authenticate the Client using the Secret encrypted in the Challenge; ii) viceversa, the Client authenticates the Resource Server on the specific data it returns, which, furthermore, should be kept confidential; iii) The Client authenticates the Authorization Server's authorization code response by checking the Scope and Nonce parameters. Note that the model does not specify any Client's authentication at the Authorization Server, anyone can request an encrypted ephemeral key claiming any Client's identity (just as anyone can request an authorization code at an Authorization Endpoint). However, as it is possible to verify using OFMC, it is fundamental that the Nonce parameter is returned to and checked by the Client, otherwise a reply attack exploiting the returned ephemeral key might occur. Using this countermeasure, the flow proves to be formally secure (till two simultaneous runs).

```
Protocol: OIDC_Implicit_ABE

Types: Agent    as, #Authorization Server (constant)
               RS, #Resource Server
                C, #Client
            Scope; #Dummy agent modeling "Scope" parameter
       Function  h, #as generated ephemeral key
               pk, #as generated Client's key
         resource; #user authorized resource to be accessed
      Number State, #OIDC AuthCodeFlow parameter
             Code, #OIDC AuthCodeFlow parameter
            Nonce, #OIDC AuthCodeFlow parameter
        Challenge, #challenge-response parameter
             Data, #RS returns this data on successfulauthorization
          Session; #shared session between RS and C


Knowledge:
            as: as,C,RS,Scope,pk(C),inv(h(C,as,RS,Scope));
            RS: RS,as,C,Scope,
              #Apparently, RS must know pk(C),h(C,as,RS,Scope).
              #However, in ABE these are trivially computed
              #from ABE master public key.
              pk(C), h(C,as,RS,Scope);
            C: C,RS,pk(C),inv(pk(C)),h,Scope;
            where RS!=C, RS!=as, C!=as

Actions:
C->RS: Scope
RS*->C: as,{{Challenge}h(C,as,RS,Scope)}pk(C)

C->as: C,RS,Scope,Nonce
as->C: {inv(h(C,as,RS,Scope)),Nonce}pk(C)

[C]*->*RS: Scope,Challenge,Session
RS*->*[C]: Data,Session

Goals:
C authenticates as on C,RS,Scope,Nonce
RS authenticates C on Challenge
C authenticates RS on Data
Data secret between RS,C
```

Fig. 2. Proposed OpenID Connect flow using Attribute-Based Encryption. Due to OFMC limitations, we did not consider master keys and key generations, rather we used two precomputed sets of traditional asymmetric keys, one for the client's key {pk(B), inv(pk(SP))} and the second one for the ephemeral key {h(B,idp,SP,Scope), inv(h(B,idp,SP,Scope))}. As a simplicifcation, the ephemeral key does not contain attributes related to the current user and time. Also, the parameter Scope is given the role of "Agent", as no key in OFMC can be associated to a parameter of type "Number" (nevertheless, the checker handles the Scope parameter as any other variable).

5 Conclusion and Future Work

Cloud computing has deeply changed our human society enabling a paradigm where data are processed on various distributed servers across the Internet, typpically developed around identity and platform provider. Secure access to this data is a primary problem. Using a model checker, we formally reviewed one of the most popular access control protocol, OpenID Connect. Then, we showed how the security of this protocol might be streamlined by the introduction of Attribute-Based Encryption. Due to space constraints, details, performance evaluation and a more extended state-of-the-art review were omitted in the compat edition of this paper. The reader is invited to refer to the full draft[14].

[13] To improve performance, the Resource Server may choose to setup a session with the Client and store the secret associated to the resource, the same mechanism may be used Client side until the expiration time is not elapsed.

[14] Available at https://t.co/naROuymJTh, last accessed on 26th May 2023.