Attribute-Based Encryption for Access Control in Cloud Ecosystems

DOI

10.36227/techrxiv.13546871.v1

Attribute-Based Encryption for Access Control in Cloud Ecosystems

G. Bartolomeo, CNIT

Abstract— We introduce a distributed, fine-granuled, policy-based resource access control protocol leveraging on Attribute-Based Encryption. The protocol secures the whole access control procedure from the authorization issuer to the resource server providing grant confidentiality, proof of possession, antiforgery and may be implemented through a developer familiar web token exchange flow plus a HTTP basic authentication flow. As such, it may map to Cloud computing SaaS paradigm, enabling microservices integration into a single, authorization-centric digital ecosystem, even across multiple identity domains. We also present the results of a performance evaluation on a first prototype implementation.

Index Terms— D.4.6.a Access controls, D.4.6.b Authentication, D.4.6.c Cryptographic controls, C.1.4.a Distributed architectures, C.2.6.f Standards

1 Introduction

As recently outlined[1], the question of how we effectively and securely identify people and enable them to perform secure tasks is one of the fundamental ones of our time. An even increasing number of applications require schemes enabling people to use digital identity credentials with more than one service provider (so called relying parties). By using these schemes, governments, financial institutions and telecommunication providers may collaborate to create nation-wide digital ecosystems offering key services to their citizens (e.g., providing bank or insurance services, healthcare, national ID cards or driver licenses verification, travel documents, etc.) through consistent identification procedures. In common Cloud computing “Software as a Service” (SaaS) implementations, for instance, many applications are built on top of web API and obtain authorization credentials as a result of their users' signing in with an “Authorization server” [1], generally run by external "Identity Providers" managing user identities. By signing in with these providers, the user effectively authorizes a transfer of credentials, delegating applications to access her data or to use API functionalities on behalf of her.

In a very widespread approach, each application receives from an Authorization server a “token” containing the authorization credentials and presents it to a “Resource server”, in order to access resources (i.e., getting user’s data or using API functionalities). Checking tokens, each Resource server can perform authorization decisions “autonomously”. At first glance, this approach might seem a cheap and convenient communication mechanism, respecting the independence principle of a canonical microservice architecture: in a typical flow, in fact, service A may include data to be exchanged in a token (signed to prove its integrity, encrypted to prevent content disclosure and provided with a nonce and/or an expiration date to avoid replay attacks), store it on a client-side, e.g., a web browser, and wait for service B to retrieve it. The latter, after retrieving the token, may simply check its validity, decrypt it and finally extract its content. However, the drawback of this simple procedure is that the token must be protected from anyone - other than by the legitimate presenter and the intended recipient - that may potentially use it to gain illegitimate access to protected resources.

With the raising of OpenID Connect 1.0 [2] and RFC 7519 “JSON Web Token (JWT)” [3] as a de facto standard for token-based authentication, in the last few years the number of identity providers have exponentially increased. JSON Web Tokens (JWT) is today so extensively used to even lead some researchers believe that it constitutes the second mostly used approach for identifying user’s clients at server side (the first being the traditional session-based authentication provided by Web servers!) [ref]. However, many aspects related to the security behind the use of JWT are still under discussion. In this paper, after deepening into some of them (section 2), we introduce an abstract protocol for securing token exchange. The protocol uses Attribute Based Encryption (ABE) to effectively distribute access control across the different involved nodes, and, differently from existing Identity-centric approaches, shifts the focus to authorization attributes themselves (leaving the user identifier just one of the attributes needed by the process). Meaning and implications of this choice is discussed in section 3. Section 4 proposes a mapping into a concrete “flow” exploiting familiar HTTP features. We present a prototype implementation and an evaluation in section 5 and finally our conclusions and plans for future work in section 6.

• *G.* Bartolomeo is with the Consorzio Nazionale Interuniversitario delle Telecomunicazioni, c/o Università di Roma Tor Vergata, Dipartimento di Ingegneria dell’Informazione, via del Politecnico 1, Rome (Italy). E-mail: giovanni.bartolomeo AT uniroma2.it

[1] See Goode Intelligence on digital identity [GD1].

2 Related Works

This section provides a brief overview of the main known vulnerabilities about web tokens together with a summary of the main relevant works which were intended to improve their security. Surprisingly enough, at the present time, only few works are available in the mainstream research, while a relevant debate is ongoing on less "conventional" sources including Standard Development Organizations, communities of developers, fintech companies[2], hackers' blogs, social networks.

Tokens are prone to man-in-the-middle attacks; therefore, they must be transmitted to an authenticated recipient over end-to-end channels secured using cryptographic algorithms, e.g., TLS 1.2+ and HTTPS.

But, even if a secure transport is used, tokens are exposed to possible attacks from malicious processes executed on the client where the application runs. As many clients are web applications, any vulnerability in a browser potentially becomes an important issue to address. XSS and CSRF attacks in particular have to be considered. When confidential tokens are sent, it is strongly discouraged - although possible - to expose them en-clair in URI query-string parameters. They are better transmitted in a HTTP "Authorization" request header. When tokens are returned, they may be stored as a cookie - set with the "Set-cookie" HTTP directive - or in the browser's local storage, using the HTTP "Authorization" response header. The two approaches differ significantly in terms of security. In fact, in principle XSS and CSRF attacks may work in both cases [4], however consolidated protection mechanisms exist and are widely accepted for cookies ("HttpOnly", "secure", "path" and "domain" flags may be used to provide various levels of protection); Therefore, proper middleware implementations on the browser side may mitigate the related risks. Instead, storing the token in the browser's local storage - often a preferred option due to space constraints in cookies - is unfortunately prone to even very simple XSS attacks due to the lack of confidentiality protection for the browser's local storage.

To address confidentiality and authenticity, the Javascript Object Signing and Encryption (JOSE) expert group has defined a set of signing and encryption methods for JWT in RFC 7515 "JSON Web Signature" (JWS) [5] and RFC 7516 "JSON Web Encryption" (JWE) [6]. RFC 7518 "JSON Web Algorithms" (JWA) [7] defines a list of cryptographic algorithms that can be used for JWS and JWE. The specifications define content signature and encryption for JWT issuers, in order to prevent token modification by an attacker on the token presenter's side.

Regarding the cryptographic framework, in [8] the author observes that some of the proposed symmetric encryption methods require either a random initialization vector or a unique nonce. For the AES-CBC with HMAC algorithm, predicable vectors may result in a vulnerability to chosen plaintext attacks, while for the or AES-GCM algorithm using a nonce even just one more time may completely compromise authenticity. In high-volume multi-server environments these circumstances are not theoretical but likely to happen.

Regarding implementations, in addition, some flaws were existing in common libraries implementing JOSE. They were identified, responsibly disclosed, and now collected and publicly available in reports from main literature [9], [10], [11]. Potential attacks exploiting these flaws rely on forged tokens. For instance, the ECDH-ES algorithm allows two parties to derive an ephemeral shared secret, to be used as a token's content encryption key (or as a wrapper for an additional symmetric content encryption algorithm key). By exploiting a flaw present in some elliptic curve cryptographic libraries, an attacker may present to the recipient a forged token encrypted using a smaller order curve and have it correctly accepted (decrypted by the recipient) or alternatively refused. Repeating this test several times, the attacker may be able to recover the secret key modulo the smaller order used in the crafted curve. Furthermore, by repeating this attack using several smaller order curves, and finally combining the different remainders (Chinese Remainder Theorem), one may finally obtain the recipient's private key, compromising the whole cryptography [12].

It is worthy to note that this attack, known as "Invalid Curve Attack", other than a bug in the library implementation, exploits the unaware complicity of the victim which acts as an oracle for the attacker, because, as observed in [13] "Decryption/Signature verification input is always under attacker's control" in JOSE.

Despite proper encryption techniques and correct implementation may be applied, an attacker may still be able to misuse tokens to perform replay attacks. Possible countermeasures to reply attacks include the use of periodically refreshed expiration time flags, one-time passwords, nonces and blacklists. All these approaches, explicitly defined in the JWT specifications, however, have their drawbacks [14] (the same reference provides further criticism about the use of tokens to store session data, as they limit accountability and result in an essential inability for the service provider to invalidate sessions when appropriate or needed. The author concludes stating that JWT, as a standalone mechanism, seems not suitable for maintaining session data).

It has been observed [15] that a Client usually cannot change the information contained in the token until its expiration. As the token remains the same in most all the request and response between client and server interaction, an attacker might predict tokens' value, or a Client might be still allowed to access a protected resource even after the corresponding user's role has been revoked. To mitigate this effect, the authors propose a secret sharing technique

[2] See for example the work of Grupo Santander available at https://gruposantander.github.io/digital-trust-docs/ (last accessed October 2020).

between Client and the (Resource) Server that updates token's signature at each request. This technique does not solve the problem of transferring the authorization between multiple Resource Servers, which is addressed in [16]. This paper introduces a dialog between the Client and multiple servers in order to share the permission obtained by the Authorization Server among an array of Resource Servers. To prevent reply attacks, tokens are invalidated as soon as they are used (or when they expire). This technique leverages on digitally signed tokens, using a symmetric key, to be shared with participating servers. But the use of a shared symmetric key for digital signature represents an intrinsic vulnerability, as any services, using the key, might generate a new valid JWT for any other user [17]. The same paper explores quantum-resistant cryptography in JWT, considering DILITHIUM and qTESLA (two lattice-based standard candidate schemes) and comparing their performance against RSA for digital signature. DILITHIUM shown the best performance, maintaining smaller performance loss when scaling to the upper NIST security level[3], at the cost of a much larger key size and signature size.

Checking the validity of a token, unfortunately, does not address the underlying problem of stating whether the token presenter is the legitimate one or rather it is an impersonating attacker. To this end, the recent RFC 7800 "Proof-of-Possession Key Semantics for JSON Web Tokens" [18] describes a method allowing a JWT presenter to claim the legitimate possession of a particular "proof-of-possession" key and a recipient to cryptographically verify this proof. The specification defines two different cases, one using public key cryptography, the other based on symmetric cryptography.

In the first use case the JWT presenter generates a public/private key pair and sends the public key to the issuer which creates a JWT containing the public key, signs it for integrity protection and sends it back to the presenter. To demonstrate the possession of the private key, the presenter signs a nonce using its private key and transmits it to the recipient (by an extra interaction). After checking the integrity of the presented token, the JWT recipient is able to verify that it is interacting with the genuine presenter by extracting the public key from the token and using it to verify the transmitted nonce signature.

In the symmetric key use case, conceptually similar to the use of Kerberos tickets, the JWT issuer, the recipient and the JTW presenter share a symmetric key. Similarly, to the previous case, the presenter, after receiving from the issuer a digital signed JWT containing an (encrypted) copy of the symmetric key and transmitting it to the recipient, uses the symmetric key in a challenge-response protocol with the recipient. The recipient may thus know that it is interacting with the genuine presenter by checking the response against the symmetric key. The symmetric use case however suffers from the same intrinsic vulnerability highlighted in [17].

3 Our Contribution

To understand the rationale behind the proposed flow, let us consider a typical "flow" implementing token-based authentication. The flow in Fig. 1 is used by a well-known identity and platform provider[4] to give third parties access to their users' profile information as well as to their API functionalities. The flow implements the "Authorization Code Flow" described in the specifications of OpenID Connect 1.0, a simple identity layer on top of the OAuth 2.0 protocol using RESTful API and optimized for web and mobile applications.

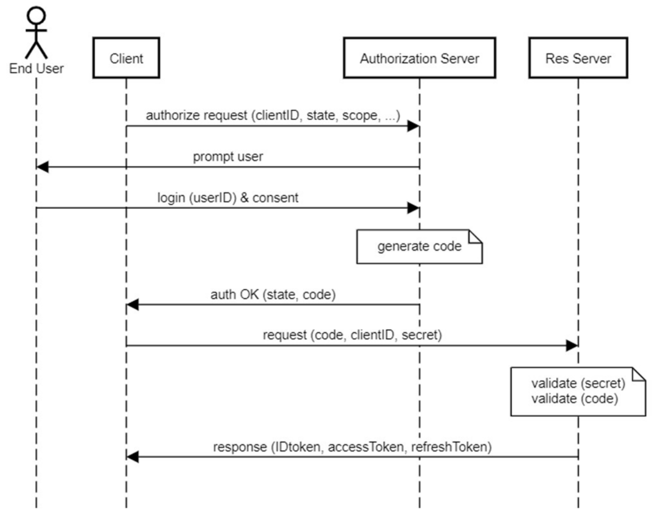

Fig. 1. A real-world implementation of OAuth2.0 as OpenID Connect 1.0 Authorization Code flow using Json Web Tokens.

As a prerequisite, the third-party provider registers its client, obtaining a "client-id" and a "secret". The following steps then apply:

1) The Client generates a session token, which should be unique in order to later allow the third-party provider to verify the authenticity of the request and prevent CSRF attacks.

2) 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, the session token ("state"), and a nonce to protect the server against replay attacks. The user is asked to login and to provide her consent to allow the third-party provider to access her profile data.

3) The response is sent back to the redirection URL as a HTTP GET request, and includes the anti-forgery "state" parameter in the query string, plus a "code" parameter, which is a one-time authorization code later exchanged for

[3] The authors consider the two NIST defined security levels (#1 and #3) for quantum cryptography which compares the strength to breaking symmetric block cipher (respectively AES-128 for level #1 and AES-192 for level #3) by key search attacks.

[4] More details about this flow can be found in Google Identity Platform online documentation, available at https://developers.google.com/identity/protocols/OpenIDConnect#authenticatingtheuser (last accessed October 2020)

an "ID token" (i.e., a token containing the requested user information) and an the access token (the actual authorization credentials).

4) 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 the ID token and the access token (optionally also a "refresh token" granting access to API functionalities after the access token expiration and without authentication the user again).

5) The Client retrieves user profile information from the ID token and may present the access token (or the refresh token) in any subsequent API call, by including it in each HTTP "Authorization" request header, to access services from the Resource Server.

Figure 2 shows the additional steps that would be needed in case a proof of possession [18] would be required.

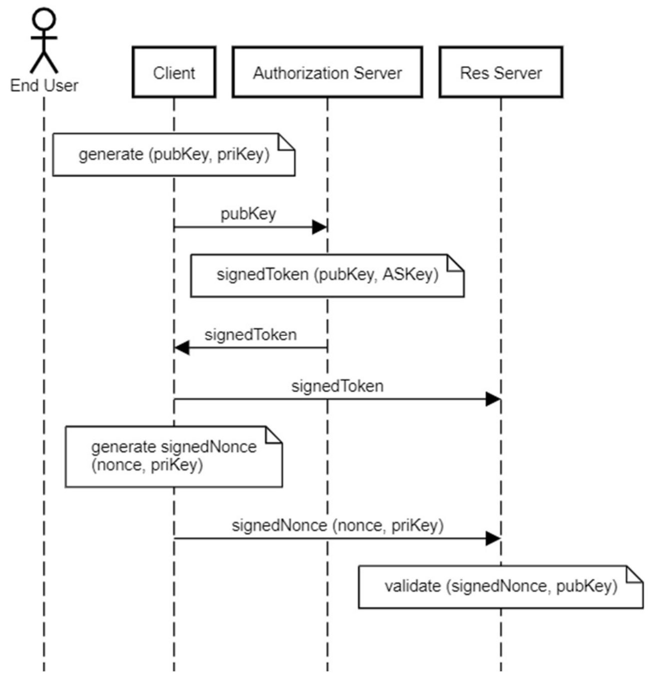
End User
Client
Authorization Server
Res Server
generate (pubKey, priKey)
pubKey
signedToken (pubKey, ASKey)
signedToken
signedToken
generate signedNonce (nonce, priKey)
signedNonce (nonce, priKey)
validate (signedNonce, pubKey)

Fig. 2. A flow implementing RFC 7800 "Proof-of-Possession Key Semantics for JSON Web Tokens" (using public key cryptography).

The flow works thanks to the one-time authorization code released by the Authorization Server and exchanged for an "ID token" and an access token at the Resource Server. But how the Resource Server is aware of the "code" parameter? In some implementations, the Resource Server may use an "introspection endpoint" to verify that the authorization code has really been issued by the Authorization Server, that it is still "active" (not expired or revoked) and what is the authorization context in which the token was granted. The OAuth 2.0 core specification, however, doesn't define a specific method, but mentions that it requires coordination between the Resource Server and Authorization Server. In some scenarios, both endpoints are part of the same platform, and can share this information internally (e.g., via a common database). In larger distributed systems, where the two endpoints are on different servers, the two servers may communicate either by own proprietary protocols or by a standard protocol described in RFC7662 "OAuth 2.0 Token Introspection" [19]. In any case, the two servers remain tightly coupled.

This situation seems reasonable in those "digital ecosystems" where the Authorization Server and the Resource Server are run by a single "Platform Provider"[5] and, generally, the Authorization Server is run by the same organization controlling the Identity Provider. Nevertheless, scenarios are possible where one or more identity provider(s) does not manage directly resources, which are instead owned by third parties[6]. These scenarios would require a common access control feature mechanism independent from any legacy resource provider, and from any identity provider. Finding such a solution would imply an integration of existing identity providers and resource providers into an authorization-centric ecosystem. This is a shift of our focus from the topic of identity attributes sharing to a general perspective of controlling user's data processing and reuse across different and independent organizations in even "wider digital ecosystems" than the ones developed, today, around single platform providers. Those new, wider ecosystems would be rather developed around new kind of businesses, similarly to today's Certification Authorities.

The basic idea to improve the OpenID Connect 1.0 "Authorization Code Flow" to enable such a change is simple, and consists in replacing the authorization "code" (at step 3 above) with a token, confidentially protected by the Authorization Server and containing a cryptographic implementation of "data use statement" policies[7] conveying the

[5] The Platform Provider model we are illustrating is an equivalent of the "Device Platform Provider" model described in ISO/IEC 19944 except that we do not necessarily assume the presence of a physical device extending the platform functionalities - as this latter may consist of a simple browser rather than of a hardware device. Device platforms are generally more secure and some attacks are not applicable (for example, device platforms, unless hijacked, provide application data isolation, resulting in "apps" less vulnerable to XSS attacks). However, an even increasing number of raising "digital ecosystems" (e.g. Milan's Expo 2015 platform, acronym "E015" [E015]) are today based on cheaper web platforms rather than on physical device platform, therefore they represent an interesting target to look at.

[6] For instance, the Italian Public Administration runs several different non communicating IT systems at all levels (national and government agencies, ministries, regional and local authorities). Recently a common Single Sign-On protocol (acronym "SPID", [SPID]) has been introduced, allowing citizens to authenticate toward these systems. Given this scenario, could we include an authorization feature enabling anyone of these legacy IT systems to share (under their consent) the same citizen's information with any other legacy system, assumed they properly authenticated using the SPID protocol?

[7] A number of works have addressed the relationship between tokens

information on what consent the user has given to the Client. As the verification relies on cryptography, and not on software, this step could be distributed among the Authorization Server and each Resource Server without requiring a backward channel (e.g., the "token introspection endpoint").

Widely using cryptographic schemes known as "Attribute Based Encryption" on behalf of traditional public key schemes, we seek for a protocol achieving the same effect of OpenID Connect Authorization Code Flow on the wider digital ecosystem above, with in addition a "native" proof of possession (without relying on the extra passages described in RFC 7800). This second goal is made possible thanks to the ABE embedded access control characteristic. In particular, we rely on one (ephemeral) key generation, containing users' permissions and consent, and two encryptions, one with the purpose of both securing the ephemeral key and ensuring that only the legitimate Client can read it, the second to prove that the Client effectively owns the key.

A third goal is to get rid of the inherent vulnerability in JWT highlighted in [13], due to decryption or signature verification on the sever. Reverting the protocol, decryption is always performed by the Client, whereas the Resource Server (or alternatively a proxy on its behalf) implements a challenge-response authentication.

For the sake of simplicity and given its ubiquitous availability, the implementation relies on the very same built-in authentication mechanism provided by the HTTP protocol. But before going into details, a brief overview of ABE is needed to introduce the abstract protocol backing end the whole flow.

4 Attribute-Based Encryption

The recent release of ETSI TS 103 532: "CYBER; Attribute Based Encryption for Attribute Based Access Control" is a step toward the spread of a common standard for ABE based solutions. The specifications however do not provide a "protocol" rather they consist of a toolkit of primitives that may be used to integrate ABE into existing protocols. Following, we just provide a brief description of the two originally defined Attribute Based Encryption (ABE) schemes, and let the reader refer to [20] for a formal definition of the ABE standard together with the description of several possible implementation algorithms and variants.
As known, a symmetric-key scheme allows two users with a pre-shared secret to securely exchange confidential data by encryption. While symmetric-key schemes usually present very good efficiency, they are not suitable for many applications, the primary drawback being that both users must share a secret before they can securely communicate. Asymmetric-key encryption schemes solve this problem using a public and private key pair. The user wishing to receive encrypted data usually generates the key pair and publishes the public key while keeping secret the private one. Anyone can encrypt data to her, using the public key, while she is the only one who can decrypt, using the private key. Today widely used for several applications (including encrypted email and secure web sessions), these schemes however lack the expressiveness needed for more advanced data sharing mechanisms. In 1985 Identity-Based Encryption (IBE) [21] brought the ability to encrypt a message to a user without knowing her public key, just her identity attribute. Twenty years later, in 2005, Brent Waters and Amit Sahai laid the conceptual foundations for a wider attribute-based encryption scheme [22]. The concept of Attribute-Based Encryption - actually derived from Identity-Based Encryption, and better refined over several following publications - enables both secret keys and ciphertexts to be associated with a set of attributes or a policy over attributes. A client is able to decrypt a ciphertext if there is a "match" between his secret key and the ciphertext. Mathematically, the ABE construction uses monotonic tree access structure. In a monotonic tree access structure, each non-leaf node of the tree is called a "threshold gate". Each leaf is associated with an attribute. The semantics of Boolean operators "AND" and "OR" can be implemented through gates using appropriate "gate thresholds". This way, a monotonic tree access structure may be read an "access policy".
In Key Policy ABE (KP-ABE) [23], a monotonic tree access structure is encoded into the user's secret key. The ciphertext is computed with respect to a master public key and a set of descriptive attributes. A client can decrypt the ciphertext if and only if the secret key it has been issued encodes an access structure matching the attributes used to compute the ciphertext (the secret key's access structure therefore specifies "which" ciphertext the key holder is allowed to decrypt). The construction consists of four algorithms:

1. *Setup(l, U):* The setup algorithm takes the global parameter l and an attribute universe U description as input and outputs the public parameters MPK and a master key MSK.
2. *Key-Gen(MSK, A):* Key generation happens by taking as input the master secret key MSK and an access structure A over a set of attributes S that describe the key. The output is a secret key $SK=SK_{MSK,A}$.
3. *Encrypt(MPK, M, S):* The algorithm takes as input a message M, a set of attributes S and the public key MPK. The output is the ciphertext $CT=\{M\}_{MPK,S.}$
4. *Decrypt(CT, SK):* Dually, the decryption takes as input the ciphertext CT that was encrypted under the set of attributes S and the secret key SK - representing the access structure A - generated at step 2. The output is the original message M if and only

and resource access management in Cloud computing, proposing different ad-hoc architectures to cope with policy management, tokens/policies synchronization issues, scalability with the number of users/resources, accountability and several related threats. Here, however, we refer to ISO/IEC 19444 specifications which describe how a Platform Provider may present complex "data use statements" to users enabling them authorization decisions about their data being accessed and used by third parties.

if S satisfies the access policy A: $M=\{CT\}^{-1}_{SK}$

Similarly, in Ciphertext Policy ABE (CP-ABE) [24] a monotonic tree access structure representing an access policy is encoded into the ciphertext, while the client's secret key is computed with respect to a set of attributes the client has been issued. A client is able to decrypt the ciphertext with a given key if and only if there is an assignment of attributes from the secret key to nodes of the tree such that the tree is satisfied (the construction consists of four algorithms similar to the above ones, here omitted due to space constraints).

Several proposals have focused on ABE as a cryptographic technique to protect the confidentiality of data resources, especially on the Cloud where users do not physically own the storage. However, they present the following limitations: 1) the need to re-encrypt the data as a consequence of an attribute revocation (in the past, proxy re-encryption techniques have been proposed to cope with this problem [25]); or, alternatively, to introduce techniques to allowing the decryption procedure to take into account possible attribute revocation, at the cost, however, of maintaining an attribute revocation list as a part of the decryption process [26]). 2) The inherent risk of leaving the ciphertext, containing actual user's data, under the attacker's control for an unlimited amount of time.

In this paper, we argue instead that ABE may be fruitfully adopted in conjunction with legacy resource confidentiality protection solutions to implement a distributed attribute-based access control. Brent Waters observed in [24] that in KP-ABE, the encrypting party exerts no control over who has access to the data it encrypts, except by its choice of descriptive attributes for the data. Rather, it trusts that the key-issuer issues the appropriate keys to grant or deny access to the appropriate users. The "intelligence" is assumed to be with the key issuer, and not the encryptor. This formulation makes KP-ABE appealing in scenarios where access to a given resource shall be granted based on policy over a set of the attributes the resources is labelled with. Fortunately, this is often the case in several OAuth scenarios, including the popular OpenId Connect 1.0 flow presented above.

5 Protocol and Flow

5.1 Protocol

In the following, we assume that a secure channel exists between the Client *Cl* and the Authorization Server *AS* as well as between the Client and the Resource Server *RS*.

We further assume an Authorization Server is able to execute the ABE set-up algorithm for KP-ABE, generating the corresponding master public key *MPK* and master secret key *MSK*. The Authorization Server is also able to generate keys based on a given policy and to perform KP-ABE encryption.

We also assume that the Resource Server is able to encrypt data using KP-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 a Client's key $k=SK_{MSK,A'}$, which is a KP-ABE key generated, as in Identity Based Encryption, by the server using the Client's identifier[8] c as a single attribute in the access structure A':

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

$$AS \rightarrow Cl: k = SK_{MSK,A'} \tag{2}$$

The Client's key shall be shipped from the server to the Client once using a secure channel.

These prerequisites assumed, the protocol begins with the Client requesting the Resource Server to access a protected resource on behalf of an end-user.

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

Where u is the user's identifier and r is the target resource identifier.

The Resource Server generates a secret x for the resource to be accessed and encrypts it using the following set S of attributes:

$$S = \{i, c, a, u, r, t\} \tag{4}$$

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.

The resulting ciphertext $\{x\}_{MPK,S}$ is further encrypted to the client, using a second set of attribute S', made of only one attribute, i.e. the Client's identifier.

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

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

The ciphertext z and the issuer i are returned to the Client.

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

The user hence is prompted to authenticate with the Authorization Server through any supported identity provider 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 KP-ABE secret key $e=SK_{MSK,A}$ (henceforth ephemeral key) from the following access structure

$$A = i \wedge c \wedge a \wedge u \wedge (r \vee r' \vee \ldots \vee r^n) \wedge (t < f) \tag{8}$$

Where $r, r', \ldots, r^n$ are the identifiers associated to the resource(s) the user has authorized access (should include r), t is the timestamp attribute and f is an expiration time for the ephemeral key.

Finally, the key e is encrypted to the Client and the ciphertext p is returned to the Client:

[8] 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.

$$AS \rightarrow Cl: p = \{e\}_{MPK,S'} \quad (9)$$

Using its key k, the Client decrypts the ciphertext and obtains the ephemeral key e.

$$e = \{p\}_k^{-1} \quad (10)$$

Owing both its own key k=$SK_{MSK,A'}$ and the ephemeral key e=$SK_{MSK,A}$, the Client can decrypt the secret:

$$x = \{\{z\}_k^{-1}\}_e^{-1} \quad (11)$$

The Client finally repeats the original request to the Resource Server, this time presenting the decrypted secret:

$$Cl \rightarrow RS: \{u, r, c, x\} \quad (12)$$

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

5.2 Flow

The above protocol may be implemented into a flow which is very similar to OpenID Connect 1.0 Authorization Code Flow. Some practical considerations (but not limitating assumptions) follows.

In order to get advantage, as much as possible, of any legacy infrastructure, a Reverse Proxy is used to communicate with the Resource Server. This scenario is quite common, as reverse proxies are used in DMZ to protect resource servers, and, at a geographical scale, may be implemented on an Edge Network to guarantee both performance and security. In addition, this approach has the advantage of leaving untouched the Resource Server itself, delegating any operation related to the flow and related security to a proxy and enabling seamless integration within existing infrastructures.

We also observes that it is common, in RESTful services, to include the user's identifier into the URI used to identify a resource, therefore the Client's request may be limited to contain two pieces of information: the resource to be accessed (including the user's identifier) r_u and the client's identifier c. This consideration enables to use a built-in HTTP mechanism, the HTTP BASIC Authentication [27], to implement the challenge-response authentication described above.

Upon an initial Client's HTTP request to the target resource r_u, which includes the Client's identifier and a void password into the request Authentication header:

```
Authentication: <BASE64(c:)>
```

the proxy server responds with a HTTP 407 Unauthorized message, containing a "realm" made of a concatenation between the computed ciphertext value z and the Authorization Server's identifier i (which may consist into a simple domain name).

```
WWW-Authenticate: Basic realm = z@i
```

As in a traditional OpenID Connect 1.0 Authorization Code Flow (step 1 and 2 described in Section 3), the Client redirects to the Authorization Server i. The Client generates a session token ("state"), unique in order to prevent CSRF attacks, and sends a HTTP GET request specifying the list of resources to authorize ("scope", which shall include the resource r), the redirection URL, the "state" token, and a nonce to protect the server against replay attacks. After the login & consent procedure takes place, the Authorization Server includes the ephemeral key as a claim into a JWT, together with other RFC 7519 and RFC8693 standard claims.

```
JWT={"iss":<i>,
"client_id":<c>,
"aud":<proxy-uri>,
"sub":<u>,
"scope":<[r, r', … , rⁿ]>,
"iat":<token-issued-time>,
"nbf":<token-issued-time>,
"exp":<f>,
"ephkey":<computed-ephemeral-key>
}
```

Using JOSE - slightly modified to include KP-ABE encryption - the generated JWT is encrypted and returned to the redirection URL inside the "code" parameter of the HTTP GET request, together with the "state" parameter.

Differently from a traditional "Server Flow" (which would require a code-for-token exchange step) the code

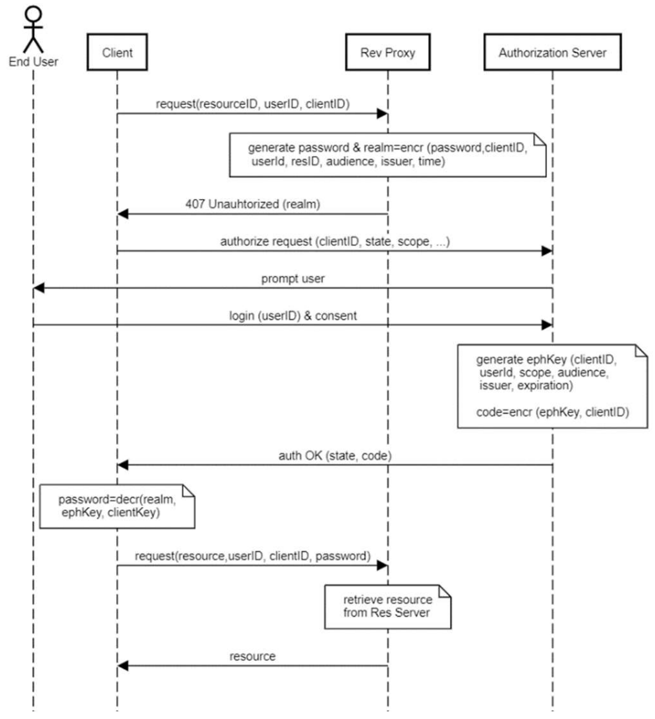
End User
Client
Rev Proxy
Authorization Server
request(resourceID, userID, clientID)
generate password & realm=encr (password,clientID, userId, resID, audience, issuer, time)
407 Unauhtorized (realm)
authorize request (clientID, state, scope, ...)
prompt user
login (userID) & consent
generate ephKey (clientID, userId, scope, audience, issuer, expiration)
code=encr (ephKey, clientID)
auth OK (state, code)
password=decr(realm, ephKey, clientKey)
request(resource,userID, clientID, password)
retrieve resource from Res Server
resource

Fig. 3. The proposed flow makes use of KP-ABE and includes a slightly modified implementation of OpenID Connect 1.0 Authorization Code flow and an HTTP challenge-response authentication.

parameter is immediately decrypted by the Client using its secret key k, thus obtaining the ephemeral key, which is used afterward to decrypt the challenging value and obtaining the random password x. Then, the Client may repeat the request to the proxy

```
Authentication: <BASE64(c:x)>
```

this time having granted the access to the target resource. The resulting flow is depicted in Figure 3.

5.3 Security Considerations

Our first consideration, as anticipated, is that decryption always happens at Client's side and is never under the attacker's control (in the sense reported in [13]). Therefore, attacks based on cryptanalysis are difficult. With this regard, we also highlight that, depending on the specific implemented variant, ABE implementations may be CPA-resistant, CCA-resistant (variant used for our evaluation in section 6) or quantum-resistant [20].

Several further considerations on the wide use of ABE in the aforementioned flow may apply.

i) ABE is used "IBE fashion" by encrypting to the Client the token generated by the Authorization Server and containing the ephemeral key.

A first ABE usage is to secure confidentiality of the returned JWT. In the aforementioned flow the KP-ABE ephemeral key, inside the returned JWT, is directly delivered to the client inside a URL. To protect its confidentiality, and prevent that the request can be modified on its way to the browser or by a malicious process running on the browser itself (e.g. through XSS or CRSF scripting) and later reuse it to impersonate the Client, the token is encrypted to the Client, hence it is not accessible to potential attackers unless of disclosure of the Client's key.

Brute force attacks against the ciphertext aimed at retieving the encrypted key and later use it to try an access to the target resource are unlikely to succeed, as the generated KP-ABE key is ephemeral and bound to a number of parameters such as the time of the request, the user identifier, the client identifier, the specific resource to be accessed. In addition, this kind of attacks would be fully accountable by the proxy server (on behalf of the Resource Server) which would probably react by invalidating such suspicious sessions and eventually blocking the source of malicious requests.

About the Client's key, instead, it is suggested that, pragmatically, a second temporal attribute is used inside the access structure that generates the key itself, so that the Client's key may expire and be regularly renewed.

ii) ABE is used for implementing a distributed access control mechanism, relying on cryptography and on the security of the Master Secret Key kept by the Authorization Server.

A question arises about the use of the double encryption technique. Given that the access policy from which the ephemeral key is generated already contains all the needed conditions to be satisfied to access the target resource, what is the meaning of a second encryption and why is it needed?

A naive answer would be that if the ciphertext were computed by a single encryption, i.e. using the ephemeral key only, a breach of this key would effectively compromise the system. Investigating more in depth, we realize that the double encryption does not only prove that the Authorization Server has effectively issued a grant to the Client to access the target resource, thus the Client is legitimate to access it, but it also ensuers that the token presenter, currently owning the ephemeral key, coincides with the legitimate Client. The Client's key (bound to the Client's identity) and the ephemeral key (bearing permissions) are generated at a different time and follow two separate routes to the Client. In decrypting the challenge, the common attribute contained in both keys (i.e., the Client's identifier) is implicitly checked to be the same (otherwise no decryption will happen) so to implement an actual proof of possession.

An alternative would be using a mutual TLS connection, which proves the identity of the presenter. In this case, equation (6) and (11) would be simply replaced by

$$z = \{x\}_{MPK,S} \tag{6bis}$$

$$x = \{z\}_k^{-1} \tag{11bis}$$

while the Client identifier c would be checked by the software powering the Resource Server against the credentials presented in the Client's certificate (equations (1) and (2) would be no more used). Note however that this choice would imply, in addition, a traditional "Code for an Access Token" passage to protect the confidentiality of the returned JWT (step 4 described in Section 3).

iii) ABE is also used to implement authenticity of the grant, overcoming traditional JWT signature.

Only the Authorization Server owns the master secret key and can issue valid keys to a Client. These keys are used to decrypt the ciphertext generated by a proxy server using the master public key and a set of attributes describing the grant. The proxy server may thus implicitly trust the authenticity of the requests coming from any Client able to decrypt the presented challenges. Note that this "inverted approach" overcomes the need of a traditional token signature and consequently the issue presented in [13] ("Decryption/Signature verification input is always under attacker's control" in JOSE).

Obviously, this mechanism works unless the Client's key is compromised, which leads to the implicit drawback behind the use of any cryptographic technique: the need to cope with key management.

Key distribution and key revocation procedures may easily map to existing legacy mechanisms without introducing additional architectural elements. More specifically, to cope with possible breaches of confidentiality of Client's key, a timestamp mechanism (i.e., a second, temporal attribute used in the access structure generating the key) may be used. The Client may obtain fresh and uncompromised keys through secure distribution channels from the Authorization Server which, at Client's request, may provide updated keys[9]. As each Client's key is bound to Client's identity, a simple HTTPS post request to the Client's specified endpoint may ensure secure transmission of the key without requiring too much effort to key distribution infrastructure developers.

6 Evaluation

6.1 Setup

To evaluate the proposed protocol, we implemented[10] a simple interactive website featured with a protected area where users can login and post their messages. The different components used to implement the site are three independent nodes taking the roles of the Authorization Server (implementing login&consent), the Client (website front-end) and the Resource Server (storage service). In particular, users may log in through their own identity provider (to leave it simple, email providers were used, but in principle any user identification technology may be plugged-in) while an independent authorization service releases tokens containing permissions to perform various actions (post comments, modify user's profile, etc). The Client uses the acquired credentials to perform persistence operations on the Resource Server, which is a simple database exposing RESTful API. We prefer not to implement the challenge-response authentication needed by the protocol directly into the Resource Server, rather to mediate the Client access through a Reverse Proxy. The Reverse Proxy handles all the burden of the cryptographic procedure leaving unmodified the legacy server interface. This architectural choice is aimed at providing the greatest flexibility, potentially proving that any legacy HTTP service can be integrated via a Reverse Proxy without no (or just minor) modifications. It also enables full accountability of access requests.

All the services were implemented using Jakarta EE 1.7 and the Java API for RESTful Web Services (JAX-RS) and were running on OpenLiberty application servers, except for the Reverse Proxy (implemented by extending the Eclipse Jetty server), due to the restrictions imposed by the Liberty framework on the underlying layers. The popular Nimbus JOSE + JWT framework was extended with the addition of a KP-ABE Encrypter and Decrypter, wrapping the OpenABE library by Zeutro[11]. As OpenABE is a C++ library, the wrapping was implemented through the Java process builder interface, i.e. as an operating system call invoking the framework Command Line Interface (CLI) (note that an alternative approach would have been to implement a Java Native Interface wrapper as in [28]). This had an actual effect in a Java multithreading environment and consequently on performance evaluation, as the encryption, decryption and key generation operations are executed atomically outside the Java virtual machine.

Three types of performance evaluations were investigated: ephemeral key generation with a variable number of attributes, challenge setup on the proxy by encrypting a secret using attributes, challenge response by decrypting the ciphertext using a key-policy on the client. In lab environment, all servers were running as a localhost in a separate Linux Ubuntu 20.04 LTS Subsystem and as such the time spent in networking operations was considered negligible. Measurements were performed on the same machine, equipped with an Intel Core i5-6200U CPU at 2.30GHz. For real-word deployment, a "containerized" Docker version of the software has been released as well.

6.2 Key generation

Every key generation takes place on the Authorization Server. The Client's key is distributed to the Client at system startup, using a secure channel (we used a HTTPS post request to the Client's specified endpoint, however other approaches may apply). Ephemeral keys, instead, are returned (encrypted) to the Client upon a traditional "server flow" request. We performed 500 key generation stress tests using a typical Boolean expression as from formula (8) (Section 5.1), containing a uniformly distributed variable number or resource attributes (1-5 attributes) disjoint by an OR operator. The average key generation time was 60,14 ms with a standard deviation of 14,55 ms and a practically negligible effect due to the number of attributes (table 1). The key size was varying between 1.516 and 2.556 Base64 octects, corresponding to 9.096-15.336 bits, much bigger than RSA keys and incomparably longer than EC-DSA keys but comparable with quantum algorithms key paradigm, where an Authorization Server might well own a distribution channel to provide keys to its Clients.

TABLE 1
EPHEMERAL KEY GENERATION

			Single thread		Multi-theading	
Attributes	**Key Length (base64 octects)**	**Key Length (bits)**	**Mean**	**Std dev.**	**Mean**	**Std dev.**
1	1516	9096	57,80	3,16	289,57	152,81
2	1744	10464	59,79	3,69	298,74	149,22
3	2016	12096	58,83	3,48	291,87	149,06
4	2288	13728	60,63	3,60	294,44	148,64
5	2556	15336	60,42	3,03	327,89	162,56
-	-	-	**59,52**	**3,54**	**299,27**	**152,17**

Measurement on 500 key generation tests, using a Boolean expression containing a uniformly distributed variable number of attributes disjoint by an OR operator. Time is expressed in ms.

[9] In an ISO/IEC 19444 Platform Provider model (which formalizes a common practice in similar real-world architectures) the Platform Provider provides interfaces (e.g. API) allowing Third Party Service Providers to use and extend the platform services. Registered third parties may obtain secret keys to use API or have access to privileges (for example, a greater number of API calls for months). Despite based on a more loosely coupled paradigm, we however envisage a similar paradigm, where an Authorization Server might well own a distribution channel to provide keys to its Clients.

[10] Source code available on Github https://github.com/netgroup/abe4jwt (last accessed January 2021).

[11] Available on Github under AGPL 3.0 license, OpenABE is compliant with ETSI TS 103 532 and provides a CCA-secure KP-ABE implementation: https://github.com/zeutro/openabe (last accessed October 2020).

size reported in [17]. We observe, however, that a simple key length comparison between ABE and other cryptographic algorithms would be unfair, as it would not consider that ABE mathematically implements a whole access control structure inside each key. The average number of generated keys per second was 16,57 keys/s.

6.3 Multithreading

Despite a modern application server may handle hundreds of threads at a time, we considered that few of them would require actual key generation at the same time, so in order to investigate the effect of multithreading we limited to repeat the experiment with 5 concurrent key generating threads. The average generation time was 299,27 ms, but the values were more dispersed, with a very elevated standard deviation of 152,17 ms. As in single threading, this does not seem due to the number of attributes involved in the key generation (Table 1). In fact, the experiment evolution over the time shows an almost regular pattern occurring at about every 4 key generation cycles (Figure 4), corresponding to the number of additional threads. Noticeably, the key generation frequency, measured at the end of the two experiments is almost the same as in the single thread experiment (16,63 keys/s) and measurements repeated at intermediate instants (every 100 tests) confirmed this alignment. We interpreted this behavior as related to the virtual machine's strategy adopted when invoking operating system calls (serialization, we suppose) and believe that most of the time is spent by the virtual machine in multi-threading management (launching new threads, switching between them, verifying their conclusion) rather than in actual data processing. Our conclusion, compatible with that reported in [28] for mobile devices, was that native C++ operations are executed one order or magnitude faster than Java operations. This behavior seems to confirm that ABE-related operations do not represent a bottleneck in itself, rather they may be delayed when implemented on interpreted languages.

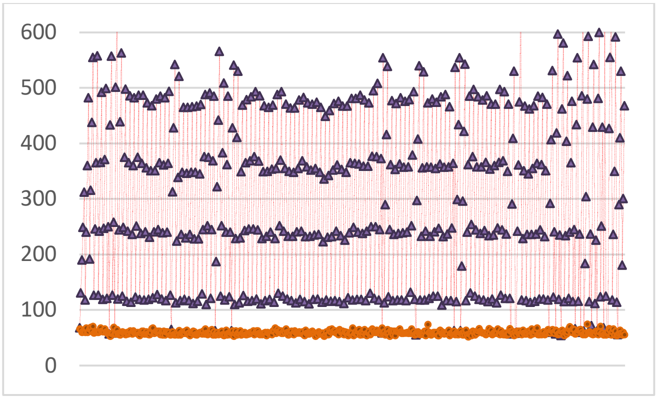

Fig. 4. Evolution of 500 key generation tests over the time. Dots represent the single thread scenario, triangles the multi-threading one. The vertical axis represents the time needed to complete a single test (in ms).

6.4 Encryption

Encryption procedure happens on the reverse proxy, by encrypting a random secret using attributes for which the legitimate Client has received a corresponding key policy. The formula (6) from Section 5.1 was used when performing KP-ABE encryption. We used the primitives provided by the CLI interface of OpenABE, which offers by default the framework's ABE CCA Scheme "Context". In turn, this "Context" uses the key derived from an underlying CCA Key Encapsulation Mechanism (KEM) Transform "Context" to encrypt plaintext of an arbitrary length using an authenticated encryption mode (AES-GCM). The benefit of this approach is that the secret's size may be variable but, as drawback, the ciphertext size suffers from the overhead due to the encapsulated key and results in a very long sequence of Base64 octects. The ciphertext almost linearly grows with the secret's size, with a fixed overhead of about 2150 octect and a grow ratio of 3:1 on every new octect added to the plaintext (Figure 5). Measurements on encryption time were performed with a 500 encryption stress tests, in both single thread and multiple threads scenarios. On average, the single thread scenario reported an encryption time of 123,94 ms with a standard deviation of 5,15 ms. The multithreading experiment reported, as expected, worse and less deterministic performances (mean 622,45 ms and standard deviation 288,65 ms) and, as in the key generation procedure, measured values were almost dispersed among five different "layers". The encryption frequency was 7,98 s^{-1} in the single thread scenario and 5,39 s^{-}

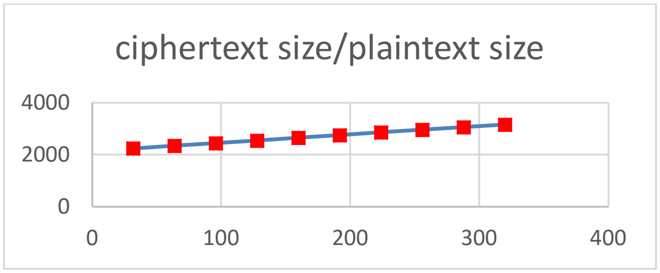

Fig. 5. Ciphertext size (vertical axis, in octects) as a function of plaintext size (horizontal axis, in octects). The ciphertext almost linearly grows with the plaintext size, with a fixed overhead of about 2150 octects and a grow ratio of 3:1 on each new octect added to the plaintext.

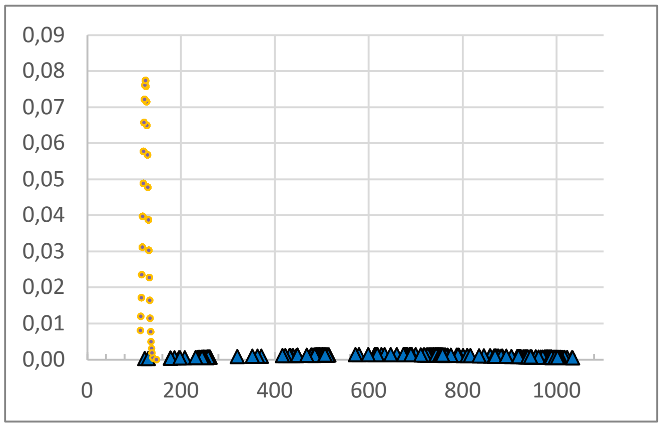

Fig. 6. Distribution of values representing the time needed to complete a single encryption test. Dots represent the single thread scenario, triangles the multi-threading one. The orizontal axis represents the time needed to complete a single test (in ms), the vertical axis the probability.

¹ in the multithreading one.

6.5 Decryption

The last set of measurements were performed on the Client. Decryption happens after the Client requests a resource for which no previous access has been granted or the grant has expired. The proxy server responds with an "Unauthorized" message, containing a ciphertext to be decrypted. After decrypting using its ephemeral key, the Client, according to the HTTP Basic Authentication protocol repeats the request to the proxy presenting the decrypted secret and receives an answer (either access to the request

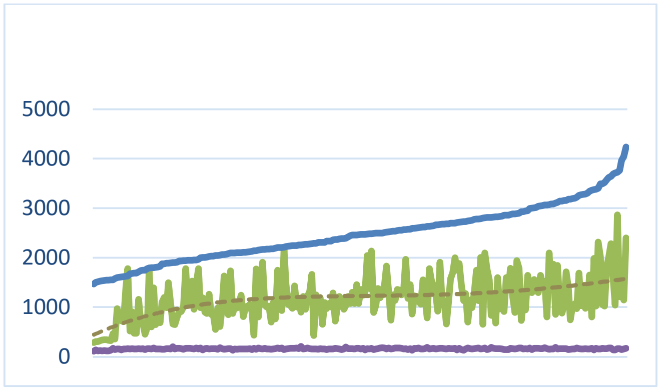

Fig. 7. Client-Proxy interactions (sorted by their completion time, reported on the vertical axis in ms). The purple line represents the decryption time on the Client, the green line represents a single roundtrip (i.e. a single request form the Client followed by a "Unauthorized" message from the Proxy), the blue line represents the completion time of the whole challenge-response authentication procedure.

resources or another "Unauthorized" message containing a new challenge). We perform a stress test on the proxy by invoking 250 requests. For each request we measured out, from the Client's standpoint: the time needed to receive a challenge, which include challenge generation on the proxy (we called this process "single roundtrip"), the decryption time itself, the overall procedure completion time. Results are depicted in Figure 7 were tests have been sorted by completion time and summarized in Table 2. The average completion time was 2,47 s (standard deviation 0,55 s), with a single roundtrip taking on average 1,18 s. As network time was considered negligible, we interpreted these data as mostly due to the internal server operations. In almost all the experiments, the single roundtrip time was lower than 1.500 s, while the decryption time itself was instead an order of magnitude lower (159 ms) and quite constant during the experiment (standard deviation 13,35 ms).

TABLE 2
CLIENT-PROXY INTERACTIONS

	Single Roundtrip	Decryption	Completion
mean	1.181,43	159,02	2.472,34
std dev	446,32	13,35	554,01

Average time and standard deviation of Client-Proxy interactions (in ms), considering a single roundtrip, the acutal decryption on the Client side and the completion time of the whole challenge-response authentication procedure.

7 Conclusion and Future Work

In less than one decade, Cloud computing has deeply changed our human society enabling a paradigm where data are processed on various distributed servers, part of so called "digital ecosystems" while users maintain control through their devices. Identifying people and providing them proper authorizations even across different digital ecosystems is a fundamental issue which is becoming more and more critical as organizations progressively moves their business from the real to the virtual environment.

Web tokens represent a popular developers' choice for signaling and conveying information in Cloud SaaS architectures, only second to the traditional session-based authentication; in particular, JSON Web Token is the most adopted approach. However, despite several cryptographic enhancements have been progressively introduced over the time, security of JWT-based protocols is still under discussion as threatened by several potential vulnerabilities, depending upon implementations and actual usages.

Through Attribute Based Encryption, a cryptographic technique combining confidentiality protection with access control, we introduced a simple protocol providing the main relevant security features and decoupling the Authorization Server function from legacy Resource Servers. With ABE leveraging on its unique feature of generating encryption keys from a chosen set of strings and regular expressions, the protocol natively introduces distributed fine-granule, policy-based resource access control suitable for Cloud computing SaaS scenarios. The resulting distributed authorization-centric mechanism may work even across different identity providers, thus being potentially able to join different domains into even wider digital ecosystems. Results from our evaluation on aa prototype implementation are encouraging and prove the viability of this approach.

Additional aspects like the use of "refresh tokens" or the handling of keys in Clients unable to maintain the confidentiality of their credentials ("public Clients" according to the definition contained in [1]) – which nevertheless represent an even more increasing number – have been deliberately left for further investigations.

Acknowledgment

The idea of a challenge-response authentication using an ABE-based access structure is owed to a discussion we had with cryptographers Dr. Pascal Pallier and Dr. Christoph Striecks, in the context of ETSI STF529. The author specially thanks them and all the STF team members for their insightful ideas.

References

[1] Hardt, D. (Ed.), IETF RFC 6749 "The OAuth 2.0 Authorization Framework".

[2] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", 8 November 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.

[3] M. Jones, J. Bradley and N. Sakimura, IETF RFC 7915 "JSON Web Token (JWT)".

[4] R. Damphousse, "Build Secure User Interfaces Using JSON Web Tokens (JWTs)", online https://stormpath.com/blog/build-secure-user-interfaces-using-jwts last accessed on 15th January 2020.

[5] M. Jones, J. Bradley and N. Sakimura, IETF RFC 7515 "JSON Web Signature (JWS)".

[6] M. Jones and J. Hildebrand, IETF RFC 7516 "JSON Web Encryption (JWE)".

[7] M. Jones, IETF RFC 7515 "JSON Web Algorithms (JWA)".

[8] Madden, N., "Misuse-resistant cryptography for JOSE/JWT". Position paper. 2018.

[9] S. Langkemper, "Attacking JWT authentication", online https://www.sjoerdlangkemper.nl/2016/09/28/attacking-jwt-authentication/ last accessed on 15th January 2020.

[10] D. Detering, J. Somorovsky, C. Mainka, V. Mladenov, J., Schwenk. On the (in-) security of JavaScript Object Signing and Encryption. In Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium (pp. 1-11). November 2017.

[11] C.K. Ho. Descriptive Research for JWT Implementation as Session Data. March 2018. http://doi.org/10.13140/RG.2.2.24623.02728

[12] A. Sanso, "Critical Vulnerability in JSON Web Encryption", March 13, 2017, online https://auth0.com/blog/critical-vulnerability-in-json-web-encryption/ last accessed on 15th January 2020.

[13] [9.b] Q. Nguyen, "Practical Cryptanalysis of Json Web Token and Galois Counter Mode's Implementations". Real World Crypto Conference 2017, online http://www.realworldcrypto.com/rwc2017

[14] S. Slootweg, "Stop Using JWT for Sessions!", online http://cryto.net/~joepie91/blog/2016/06/19/stop-using-jwt-for-sessions-part-2-why-your-solution-doesnt-work/ last accessed on 15th January 2020.

[15] Ahmed, S. and Mahmood, Q. (2019), "An authentication based scheme for applications using JSON web token", in 2019 22nd International Multitopic Conference (INMIC). IEEE, 2019. p. 1-6.

[16] Mestre, P., et al., "Securing RESTful Web Services using Multiple JSON Web Tokens", in Lecture Notes in Engineering and Computer Science: Proceedings of the World Congress on Engineering 2017, 2017, p. 418-423.

[17] Alkhulaifi, A and El-Alfy, E.S.M., "Exploring Lattice-based Post-Quantum Signature for JWT Authentication: Review and Case Study", in IEEE 91st Vehicular Technology Conference (VTC2020-Spring). IEEE, 2020, p. 1-5.

[18] M. Jones, J. Bradley and H. Tschofenig, IETF RFC 7800 "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)".

[19] Richer, J. (Ed.), IETF RFC7662 "OAuth 2.0 Token Introspection".

[20] ETSI TS 103 532: "CYBER; Attribute Based Encryption for Attribute Based Access Control".

[21] Adi Shamir. Identity-based cryptosystems and signature schemes. In Proceedings of CRYPTO 84 on Advances in cryptology, pages 47–53. Springer-Verlag New York, Inc., 1985.

[22] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In EUROCRYPT, pages 457-473, 2005.

[23] V. Goyal, O. Pandey, A. Sahai, B. Waters: "Attribute-based encryption for fine-grained access control of encrypted data", Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS '06, pages 8-98, New York, NY, USA, 2006. ACM.

[24] J. Bethencourt, A. Sahai, B. Waters: "Ciphertext-policy attribute-based encryption", Proceedings of the 2007 IEEE Symposium on Security and Privacy, SP'07, pages 32-334. Washington, DC, USA, IEEE Computer Society.

[25] Liang, K., Fang, L., Susilo, W., & Wong, D. S. (2013, September). A ciphertext-policy attribute-based proxy re-encryption with chosen-ciphertext security. In 2013 5th International Conference on Intelligent Networking and Collaborative Systems (pp. 552-559). IEEE.

[26] Horváth, M. (2015, January). Attribute-based encryption optimized for cloud computing. In International Conference on Current Trends in Theory and Practice of Informatics (pp. 566-577). Springer, Berlin, Heidelberg.

[27] J. Hallam-Baker, P. Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Stewart, L., IETF RFC 2617, "HTTP Authentication: Basic and Digest Access Authentication".

[28] Ambrosin, M., Conti, M., & Dargahi, T. (2015, May). On the feasibility of attribute-based encryption on smartphone devices. In Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems (pp. 49-54).

Additional references reported in note #1, #5 and #6:

[GD1] Goode Intelligence: "The Digital Identity Report – The Global Opportunities for Verified Citizen & Consumer Digital ID; Market & Technology Analysis, Adoption Strategies and Forecasts 2020-2025".

[E015] Zuccalà, M. and Verga, E.S. (2017), "Enabling Energy Smart Cities through Urban Sharing Ecosystems". Energy Procedia, Volume 111, March 2017, Pages 826-835. OI:10.1016/j.egypro.2017.03.245

[SPID] Agenzia per l'Italia Digitale, "Linee Guida OpenID Connect in SPID" (Italian only), https://docs.italia.it/AgID/documenti-in-consultazione/lg-openidconnect-spid-docs/it/bozza/index.html

Giovanni Bartolomeo received his PhD degree in Telecom Engineering from the University of Rome Tor Vergata, in 2014. He is currently serving as an IT Officer at the Italian Ministry of Justice. He was Team Leader in ETSI STF 529 and member of ETSI TC CYBER, ETSI TC HF, ETSI STF342, OASIS XDI TC. His research interests include semantics, data analytics, cybersecurity and data privacy. He has published several articles in peer-reviewed journals and conferences, and he is author of the book "Identification and Management of Distributed Data", Taylor & Francis, Boca Raton, FL, US (2013).