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 common web token exchange flow plus a HTTP basic authentication. As such, it may easily map to Cloud computing SaaS paradigms, enabling services integration into a single authorization-centric 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

THE question of how we effectively and securely identify people and enable them to perform tasks is one of the most fundamental ones of our time. An even increasing number of Internet applications require schemes enabling people to use digital identity credentials with more than one service provider (so called relying parties). By using these schemes offering consistent identification procedures, governments, financial institutions and telecommunication providers may collaborate to create nation-wide digital ecosystems bringing key services to their citizens (e.g., providing bank or insurance services, healthcare, national ID cards or driver licenses verification, travel documents, etc.). 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”, generally run by external “Identity Providers” managing user identities. By signing in with one of 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 implementation of these schemes, OAuth [1], each application (“Client”) 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 (or metadata) 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, 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 triangulation is that the token must be protected from any potential attacker that may use it to gain dishonest access to protected resources.

With the raising of OpenID Connect 1.0 [2] and the JSON Web Token (JWT) [3] as the 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 lead some researchers believe that it constitutes the second mostly used approach for identifying users at server side (the first being the traditional password-based authentication) [4]. However, many aspects related to the security of JWT and OpenID Connect are still under discussion. In this paper, after deepening into some security issues today known and understanding their mitigation measures (section 2 and 3), we redesign (section 4) the whole authorization process by using Attribute-Based Encryption (ABE). ABE cryptographically enforces distributed access control across the three different involved parties, while the resulting protocol may be mapped into a familiar OpenID Connect “Implicit Flow”, implementing the original triangulation procedure between a Client, an Authorization Server and a Resource Server (section 5). We present a prototype implementation and its evaluation in section 6 and our conclusions in section 7.

2 RELATED WORKS

This section provides an overview of the main known vulnerabilities of web tokens together with a summary of relevant works which were intended to improve their security. With the exception of [5], at the present time, only

• 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

few works are available in the mainstream research literature while a relevant debate is ongoing on less “conventional” sources including Standard Development Organizations, communities of developers, fintech companies, 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 must necessarily be considered. When confidential tokens are sent, it is strongly discouraged - although possible - to expose them en-clair in URL query-string parameters. They are better transmitted in a HTTP “Authorization” request header, possibly under TLS. This was actually the reason why OpenID Connect “Authorization Code Grant” flow was introduced, replacing the original “Implicit Grant” flow for most applications.

When tokens are returned to a browser, 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. Even if in principle XSS and CSRF attacks may work in both cases [6], 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). 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.

To address confidentiality and authenticity, the Javascript Object Signing and Encryption (JOSE) expert group has defined a set of methods for the Json Web Token in RFC 7515 “JSON Web Signature” (JWS) [7] and RFC 7516 “JSON Web Encryption” (JWE) [8]. RFC 7518 “JSON Web Algorithms” (JWA) [9] defines a list of cryptographic algorithms that can be used for JWS and JWE. The specifications define content signature and encryption for JWT issuers, to prevent token modification attacks on the token presenter’s side.

Regarding the cryptographic framework, in [11] 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, predictable vectors may result in a vulnerability to chosen plaintext attacks, while for the AES-GCM algorithm, reusing the same nonce may completely compromise authenticity. According to the author, in high-volume

multi-server environments these circumstances are not theoretical but likely to happen.

Regarding implementations, in addition, some flaws were detected in common libraries implementing JOSE (now collected and publicly available in reports from main literature [11], [12], [13]). 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 key). By exploiting a flaw present in some elliptic curve cryptographic libraries, an attacker may give 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. Then, by repeating this attack using several smaller order curves, and finally combining the different remainders, the attacker may finally obtain the recipient’s private key, compromising the whole cryptography [14].

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 [15] “Decryption/Signature verification input is always under attacker’s control” in JOSE.

It has been observed [4] that a Client usually cannot change the information contained in the token until its expiration. As the token remains the same in almost all the interactions between the client and the server, 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 that updates token’s signature at each request. This technique however 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. Unfortunately, the use of a shared symmetric key for digital signature represents an intrinsic vulnerability, because any services, using the shared key, might play the role of an attacker by generating a new valid JWT for any other user, as noted in [17]. This latter 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 level¹, at the cost of a much

¹ The authors consider the two NIST defined security levels (#1 and

#3) for quantum cryptography which compares the needed strength

larger key size and signature size.

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 [18]. 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 needed. The author remarks that JWT, as a standalone mechanism, is not an appropriate mean for maintaining session data.

However, JWT is rarely used by its own. Rather it is mostly used in conjunction with OAuth 2.0 and its derived specification Open ID Connect (OIDC) [2]. The original idea behind this specification was a simple triangulation between a Client, an Authorization Server and a Resource Server. 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, due to potential attacks discussed in the previous section. Therefore a slightly more complex flow named “Authorization Code Flow” is used in real world cases. Figure 1 illustrates an implementation of this flow by a well-known identity and platform provider² to give third parties access to users’ profile information as well as to API functionalities. The flow implements the “Authorization Code Flow”, described in the specifications of OpenID Connect 1.0 as a simple identity layer on top of the OAuth 2.0 protocol using RESTful API and optimized for web and mobile applications

As a prerequisite, a 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 (“State”), which should be unique to later allow the Client to verify the authenticity of the following incoming request from the Authorization Server (step 3) 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 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 a signed ID token and signed access token embedding the Nonce to prevent reply attacks (optionally also a “refresh token” granting access to API functionalities after the access token expiration and without authenticating the user again).

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

Even if this flow proves to be formally correct (section 3), unfortunately, checking the validity of a token, by its own, does not provide the Resource Server with any assurance on whether the token presenter is the legitimate client or an attacker. To this end, the recent RFC 7800 “Proof-of-Possession Key Semantics for JSON Web Tokens” [19] describes a method allowing a JWT presenter to claim the legitimate possession of a particular “proof-of-possession”

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

key and a recipient to cryptographically verify this proof. The specification explicitly refers to an OAuth 2.0 setup and 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

to break symmetric block cipher (respectively AES-128 for level #1 and AES-192 for level #3) by key search attacks.

² More details about this flow can be found in Google Identity Plat-

form online documentation, available at <https://developers.google.com/identity/protocols/OpenIDConnect#authenticatingtheuser> (last accessed October 2020)

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

In the symmetric key use case, conceptually similar to the use of Kerberos tickets, the JWT issuer, the recipient and the JWT 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 decrypting the response using the symmetric key.

Note however that the symmetric use case suffers from the same intrinsic vulnerability highlighted in [17], thus it is less viable in real world scenarios, where proposals have focused on techniques leveraging on asymmetric cryptography. For instance, in “Demonstration of Proof-of-Possession at the Application Layer” Internet draft [20], the Client self signs its public key and presents it as a “DPoP HTTP header” to the Authorization Server in order to receive a signed token certifying the client’s public key “thumbprint” [21]. Proof of possession is achieved by presenting later the received token together with the DPoP (including a signed data piece to avoid reply attacks) to the Resource Server, with the latter comparing the signing key with the one certified in the token. [22] provides a summary of other protocols implementing a possible proof-of-possession, the only one recently standardized (February 2020) being “OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens” [23] and using TLS mutual authentication instead of a self-signed public key by the client (the Authorization Server associates the fingerprint of the client’s public key from the TLS stack with the access tokens).

Note that neither DPoP nor mTLS Certificate-Bound are resistant to XSS vulnerabilities, where an attacker is allowed to execute code in the context of a browser-based client application (in case of DPoP, the specification works correctly as long as “the private key is non-extractable (as is possible with W3C Web Cryptography API [24])”). As stated in DPoP specifications, “at this application layer there is most likely no feasible defense against this [XSS] threat except generally preventing XSS”.

A further issue affects the Authorization Code Grant flow. If the code is leaked, an attacker can exchange it for the final token if other protection measures (e.g. a Client’s password) are not used at the token exchange endpoint. Or the attacker can inject the leaked code in a second session

by the same Client (for example, on behalf of another user) to associate the original token to the second session, thus altering authorization grants. First examples of code leakages were observed in mobile devices, where the attacker can register a malicious application and a custom URI scheme on behalf of a legitimate application with the aim of receiving authorization codes intended for this latter. However, this vulnerability also affects browsers (e.g., by XSS) [22] and any device where the attacker is able to gain authorization code by inspecting HTTP log information. The current mitigation measure is to rely on a secret (“code verifier”) which is generated by the Client and sent in a “transformed version” to the Authorization Server when asking for the code; later the Client sends the secret together with the original code in order to obtain the token. Applying the “transformation”, the Authorization Server can verify that the request does not come from an attacker (who does not know the secret) but from the legitimate secret holder. In a variant known as Identity Provider mix-up attack [5], authorization code leakage may happen when the attacker makes the Client believe the user wants to exchange code for token at a fake Authorization Server token endpoint, controlled by herself. The currently suggested mitigation measure is for the Authorization Server to indicate the correct token exchange endpoint to the Client by sending back, together with the code, the “[code and token] issuer information” either in an ID Token returned with the code (if present) or as a separate parameter inside the HTTP response returning the token [25].

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

3 ASSESSING OIDC SECURITY WITH OFMC

In this section we review the previously mentioned attacks using a formal model and a model checker. To the best of our knowledge, at the time of writing, the only formal model available for OIDC is [5]. The model is very comprehensive and was successfully used to discover several attacks (including the mentioned Identity Provider

Mix-up attack). However, this model does not allow automation and requires manual theorem proofs. A different approach to formal analysis of security properties is using an automatic model-checker. In our work, we rely on the last release (year 2020) of the symbolic model-checker OFMC [26], because of two main reasons: 1) models and security goals may be formally specified using an easy specification language such as AnB [27] and 2) the checker supports vertical channel composition in protocol specifications.

In fact, we construct our model by relying on two kinds of channels, modelling some properties of the HTTP(S) protocol³:

- $[A] \bullet \rightarrow \bullet B$ 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).

- $A \rightarrow B$ denotes an unencrypted, unauthenticated channel between the A and B , such as plain HTTP is.

Other alternatives are possible, and we refer the reader to OFMC manual for further details on channel composition. [5] suggests that 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 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 [28] 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 the one presented in [5], we can still model the attacker as effectively able to operate from inside the user’s device, by relaxing assumptions 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, the model specifies an unsecure channel, 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 [5]) 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.

We report the model in AnB syntax and output from OFMC in Annex I. The 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 used⁴. The Client connects to the Authorization Server passing $Scope$, $State$ and $Nonce$ parameters. After user login & consent (not modeled), the Client is returned a code, which is a function of passed parameters: $code(Scope, State, Nonce)$ ⁵. In order to study the effect of code leakage and injection (which in real world may happen in a number of ways), the code is returned under an unsecure channel. 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 none 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 token⁶. The Resource Server limits to trust the token signature.

As regarding the security objectives, we specify the following:

- 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 (i.e., the token issuer, the token intended audience – i.e. itself – the token legitimate owner, i.e., the Client, the token signature).
- Viceversa, the Client authenticates the Resource Server on the specific data it returns, which, furthermore, should be kept confidential.
- The Client authenticates the Authorization Server authorization code response by checking the $State$ parameters and both agreeing on the returned (user

³ Strictly speaking, OFMC does not assume a concrete protocol implementation, but relies on abstractions. Channel abstractions however are useful approximation of transport protocols and have been traditionally used to model TCP and TLS flows in the verification of complex protocols, under the assumptions provided in [44] and generally satisfied by a large class of transport protocols. Note however that some properties of TLS, namely strong authentication (preventing session replies), are not modeled in the OFMC abstraction (and if this property has to be checked, OFMC users have to implement their own in the model specification).

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

⁵ Parameter $Scope$ is returned as well. This is not needed by the flow, however it is used in the model to avoid the checker detecting a trivial reply attack with the response message (as aforementioned, in OFMC channels are abstractions and, contrary to real world transport protocols, do not implement session reply protection by their own).

⁶ As discussed above, 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.

authorized) Scope and code.

- At the token endpoint, the Authorization Server must authenticate the Client via its registered identity and password and the code parameter. The usual recommendation is that the code parameter is one-time-usable, “short-living” (max 10 minutes lifespan) and “sufficiently” random [29]. In our model, we relax this assumption and accept that code parameters may even be occasionally reused (thus, this security goal is a “weak” authentication, similar to Lowe’s non-injective agreement [30]).

Running the model checker up to 2 simultaneous runs, we found, as expected, the very well-known code injection attack. Once the code is leaked to an attacker, the attacker, acting as a fake authorization endpoint, or replacing a message parameter coming from an honest endpoint, may “inject” it on another Client’s session. 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 know whether and which resources the user has

no attack was found – unless the Client’s password at the token endpoint is compromised.

We then incremented the model with the addition of the PKCE [31], i.e. the ability, for the Client, to send a code to the authorization endpoint that the Authorization Server may later verify, through a “Verifier”, sent by the Client to the Token endpoint. If the Client does not check the Nonce parameter returned in the token, a code injection is still possible (still violating the *third* security goal), however, properly checking the Nonce contained in the token makes the flow secure till 2 simultaneous runs (by using PKCE this property holds even in case no password is used at the Token endpoint).

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 Demonstration of Proof-of-Possession as described in the current draft [20], will finally result in a secure flow till 2 simultaneous runs, as long as the presented signature is over sufficient parameters to make it prevent reply attacks (in the model reported in Figure 3 we included the Client’s public key, the Resource Server address, the Scope parameter).

4 OUR CONTRIBUTION

As emerged from the above analysis, the Resource Server has no way to know whether the access to resources contained in the token were granted on the current run (i.e., user’s session with the Resource Server). The Resource Server even does not know whether the token presenter is the legitimate owner. In OpenID Connect, code injection is one of the most dangerous attack and, as seen, may be avoided using an explicit Nonce inside the token (assumed the Client’s password is kept secure over the time; or a PKCE may be used as an alternative to Client’s password). However, this solves only part of the problem, as the Resource Server does not know whether the Nonce parameter is a valid one. An Introspection Endpoint may be used to verify that the access token has been really issued by the Authorization Server, that it is still “active” (not expired or revoked) and what authorization context produced the token itself. The OAuth 2.0 core specifications, however, does not 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 information internally (e.g., via a common database). In larger distributed systems, where the two endpoints are on different machines, maybe on different domains, the two servers may communicate by some proprietary protocols or by the standard protocol described in RFC7662 “OAuth 2.0 Token Introspection” [32]. In any case, the two servers need to remain tightly coupled⁷.

```

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 successful authorization
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. 3. OpenID Connect Authorization Code Flow implementing Proof Key for Code Exchange (PKCE) and Demonstrating Proof-of-Possession (DPoP), modeled in AnB specifications language. This model was run using the Opensource Fixedpoint Model Checker (OFMC) and proved to be secure till 2 simultaneous runs.

authorized access to (hence, no violation of the first goal).

We also repeated the test making the Client explicitly checking the Nonce parameter contained in the returned token (as in the OpenID Connect specifications). This time,

⁷ This situation seems reasonable in those “digital ecosystems” where the Authorization Server and the Resource Server are shared by a single “Platform Provider” (ISO/IEC 19944) and, generally, where 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 directly manage resources, which are instead owned and

controlled by third parties. 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). Few years ago, a common Single Sign-On protocol (“il Sistema Pubblico di Identità Digitale (SPID)”, see <https://www.spid.gov.it/>) has been intro-

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 (Figure 4). Incidentally, we note that a conceptually similar need was present in the Google’s (flawed) variant of Web Browser SAML 2.0 Single Sign On profile [33] where “*the attack originates from one of the simplifications [made by Google] that deprives the authentication assertion of both the ID and SP fields*” (ID being the initial session identifier shared by the Client and the Resource Server).

While one simple solution could be to assign the Nonce parameter a value directly computed from the initial Session parameter (so that a Resource Server may later check whether the token has been generated in the context of the current session, and not on another one)⁸, this would however bind the whole security to the generation and confidential handling of the initial Session parameter which might be a too weak assumption in some contexts.

Hence, despite mitigation measures like PKCE, DPoP, Introspection Endpoints exist, it is undeniable that they

Fig. 4. State of the art measures to secure OpenID Connect Authorization Code Flow. PKCE and DPoP essentially creates a shared distributed session between the Authorization Endpoint, the Token Endpoint, the Client and the Resource Server. An Introspection Endpoint may be used to ensure the token was really issued by the Authorization Server.

makes the original idea of a flow implementing a simple triangulation between the Client, the Authorization Server and the Resource Server more difficult to realize, consequently more problematic and costly to certificate in software implementations. One question to ask is whether we can redesign a flow by handling over most part of verifications, decisions and enforcement to the cryptographic

layer, by packaging everything inside a single “semantically understandable” JWT, as in the original idea. After all, existing flows already delegate many checks to cryptography via signature verification.

[22] claims that the original OpenID Connect Implicit Grant flow is vulnerable to token leakage and replay attacks because access tokens are directly issued in the authorization response and no viable “mechanism” exists to cryptographically bind tokens to a certain Client (making, in addition, token replay detection impossible for Resource Servers). However, this claim is inaccurate: such “mechanism” exists and is known as “Identity-Based Encryption” [34]. Therefore, we start from the idea of amending the original OpenID Connect “Implicit Grant” flow by replacing the token in clear with one confidentially protected by the Authorization Server and “encrypted to the Client”. In Identity-Based Encryption (IBE) schemes 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. An IBE scheme thus may be useful in cases like the one pointed out in [22] because it avoids the overhead due to preliminary public key distribution between parties (only one MPK needs to be known in advance). Private keys are issued by the Generator and shipped to the parties, i.e., 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 is used to create a confidential channel between the server and the Client, assumed this latter can keep confidential and non-extractable its secret key (the very same assumptions in DPoP and similar specifications).

As a second step, having realized that encryption based on identity is feasible – which avoids the need for a code for token exchange and any related threats – we wonder whether a whole access control protocol may be built on cryptographic basis. We seek for a protocol achieving the same effect of Authorization Code Flow with, in addition, a “native” proof of possession. This second goal is made possible thanks to schemes known as “Attribute-Based Encryption”. We use these schemes to achieve a cryptographic implementation of “data use statement” policies⁹ conveying the information on what consent the user has provided. In our proposal, we rely on one (ephemeral) key generation, containing users’ permissions and consent, and two encryptions, one with the purpose of 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.

duced, giving citizens one single way to authenticate and access these systems. Given this scenario, could we include an authorization feature enabling any of these legacy IT systems to share a citizen’s information it holds with potentially any other one, assumed that the citizen has been properly authenticated using the SPID protocol and provided her consent? This scenario would require a common access control feature across different platforms (with many resource providers platforms totally decoupled from the identity provider platform).

⁸ We wrote a model to check this assumption, however we were not able to validate it using OFMC in over 4 hours run. We report this model in Annex I and leave it available for researchers willing to deepen this solution.

⁹ We mutuate a terminology from ISO/IEC 19944 specifications which abstractly describe how a “Platform Provider” may present “data use statements” to users enabling them authorization decisions about their data being accessed and used by third parties.

A final goal is to get rid of the inherent vulnerability highlighted in [15], due to repeated signature verification by the server. Reverting the protocol, decryption is always performed by the Client, whereas the Resource Server (or alternatively a proxy on its behalf) implements a randomized 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.

5 PROTOCOL AND FLOW

5.1 A Brief Overview of Attribute-Based Encryption

Before going into details, a brief overview of ABE is needed to introduce the abstract protocol backing end our proposed flow. We just provide a history and a quick description of the two originally defined Attribute-Based Encryption (ABE) schemes, and leave the reader refer to [35] for further details on variants and their implementations.

A symmetric-key scheme allows two users with a pre-shared secret to securely exchange confidential data. While symmetric-key schemes usually present very good performances, 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 willing 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 emails and secure web sessions), these schemes however lack of the expressiveness needed for more advanced data sharing mechanisms. In 1985 Identity-Based Encryption (IBE) [34] brought the ability to encrypt a message to a user without knowing her public key, but just a public identity attribute, under the condition of sharing a public "master" key. Twenty years later, Brent Waters and Amit Sahai laid the conceptual foundation for a wider attribute-based encryption scheme [36]. The concept of Attribute-Based Encryption - 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 an access policy over attributes. A client can decrypt a ciphertext if there is a "match" between the policy and a corresponding set of attributes. Mathematically, the ABE construction is an application of Shamir's secret sharing [37] over monotonic tree access structures. 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 interpreted as an "access policy". In the (historically) first presented construction, Key Policy ABE (KP-ABE) [38], 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). A few years later this publication, Waters observed 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". He therefore presented a second construction, Ciphertext Policy ABE (CP-ABE) [39], where a monotonic tree representing an access structure is encoded into the ciphertext, allowing encrypting party to formulate a policy controlling encrypted data access, while the client's secret key is computed with respect to a set of attributes the client has been issued. A client can 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 monotonic tree such that the tree is satisfied.

The CP-ABE 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, S): Key generation happens by taking as input the master secret key MSK and a set of attributes S that describe the key. The output is a *randomized* secret key $SK=SK_{MSK,S}$.
3. *Encrypt*(MPK, M, A): The algorithm takes as input a message M , an access structure A over a set of attributes and the public key MPK . The output is the ciphertext $CT=\{M\}_{MPK,A}$.
4. *Decrypt*(CT, SK): Dually, decryption takes as input the ciphertext CT generated at step 3 and the secret key SK generated at step 2. The output is the original message M if and only if the set of attributes S satisfies the access structure A : $M=\{CT\}^{-1}_{SK}$, null otherwise.

In recent years, 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 some limitations:

- the need to re-encrypt the data as a consequence of an attribute revocation (proxy re-encryption techniques have been proposed to cope with this problem [40]); or, alternatively, to introduce techniques 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 [41]).
- The inherent risk of leaving the ciphertext, containing actual user's data, exposed to cryptanalysis attacks for an unlimited amount of time.
- The superpower of the key-issuer, able to generate keys and thus to potentially decrypt every ciphertext.

In this paper, we argue instead that ABE may be fruitfully

adopted in conjunction with legacy resource confidentiality protection solutions (in particular, OAuth 2.0/OpenID Connect) to implement a distributed attribute-based access control system.

5.2 Protocol

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 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 a Client's key $k=SK_{MSK,i}$, which is a CP-ABE key generated by the server using the Client's identifier¹⁰ c as a single attribute (IBE-style fashion):

$$S' = \{c\} \quad (1)$$

$$AS \rightarrow Cl: k = SK_{MSK,S'} = SK_{MSK,\{c\}} \quad (2)$$

The Authorization Server ships this key to the Client once at startup, using a secure channel.

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

$$Cl \rightarrow RS: \{u, r\} \quad (3)$$

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 access structure A :

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

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.

$$A' = \{c\} \quad (5)$$

$$z = \{\{x\}_{MPK,A}\}_{MPK,A'} = \{\{x\}_{MPK,A}\}_{MPK,\{c\}} \quad (6)$$

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

$$RS \rightarrow Cl: \{z, i\} \quad (7)$$

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

$$Cl \rightarrow AS: \{c, a, r, m\} \quad (8)$$

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

$$S = \{i, c, a, u, r, r', \dots, r^n, t\} \quad (9)$$

where $r, r', \dots, r^n$ 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:

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

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

$$\{e, m\} = \{p\}_k^{-1} \quad (11)$$

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

$$x = \{\{z\}_k^{-1}\}_e^{-1} \quad (12)$$

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

$$Cl \rightarrow RS: \{u, r, c, x\} \quad (13)$$

The Resource Server checks the secret presented by the Client, and, in case of a positive match, grants access to the requested resource¹¹.

5.3 Flow

The above protocol may be implemented into a flow which is very similar to OpenID Connect 1.0 Implicit Flow. Some practical considerations (but not limiting assumptions) follow.

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 security to a proxy, enabling seamless integration with existing infrastructures.

We also observe 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 easily use a built-in HTTP mechanism, the HTTP BASIC Authentication [42],

¹⁰ 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.

¹¹ 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.

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 be a simple domain name).

WWW-Authenticate: Basic realm = z@i

As in a traditional OpenID Connect 1.0 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 standard claims [3] describing the key (Figure 5). Using JOSE [8] - slightly modified to include CP-ABE encryption - the generated JWT is encrypted and returned to the redirection URL, together with the "state" parameter.

Differently from a traditional Authorization Code Flow

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

Fig. 5. The ephemeral key as a claim into a JWT, together with other standard claims.

(which would require a code-for-token exchange step) the code parameter is immediately decrypted by the Client using its secret key k , thus obtaining the ephemeral key, which is used afterward to decrypt the challenge and obtain the secret x . Finally, 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 6.

5.4 Formal Model and Checking with OFMC

In order to validate the proposed flow, we developed a model and checked it with OFMC. Given the lack of explicit support for ABE in OFMC, we emulated the ABE keys using the checker's available features (see Annex I and Figure 7). The model specifies the following actions: after a Client connects to a Resource Server, a Challenge encoding a Secret – as in equation (6) – is returned together with the address of the Authorization Server (we assume

the Challenge comes from the real Resource Server and not from an attacker by signing the challenge with the Resource Server's public key, in practice this happens in a TLS channel). Following, the Client is redirected to the Authorization Server as in a traditional OpenID Connect flow. After user's login and consent, an encrypted ephemeral key (computed as in equation (10)) is returned to the Client. All steps right now expose parameters under the attacker's control and do not need any secure channel (except the assumption on the authenticity of the Challenge). In the final step the Client decrypts the Challenge and presents the Secret to the Resource Server in order to obtain the requested resource.

The model specifies the following goals:

- The Resource Server must authenticate the Client using the Secret encrypted in the Challenge.
- Viceversa, the Client authenticates the Resource Server on the specific data it returns, which, furthermore, should be kept confidential.
- The Client authenticates the Authorization Server's authorization code response by checking the Scope and Nonce parameters.

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

Fig. 6. The proposed flow makes use of CP-ABE and includes a slightly modified implementation of OpenID Connect Implicit flow and an HTTP challenge-response authentication.

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 (Annex I). Using this countermeasure, the flow proves to be secure till two simultaneous runs.

5.4 Further Considerations

Our first consideration, as anticipated, is that in our proposed flow decryption always happens at Client's side and is never "under the attacker's control" (in the sense reported in [15]). Therefore, attacks based on cryptanalysis are more difficult. With this regard, we also highlight that, depending on the specific variant, an ABE implementation may be CPA-resistant, CCA-resistant (variant used for our evaluation in section 6) or quantum-resistant [35].

Several further considerations on the wide use of ABE features may apply.

i) ABE is used "IBE-style 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 our flow the CP-ABE ephemeral key, inside the returned JWT, is directly delivered to the Client inside a URL. To protect its confidentiality, and to 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 scripting), the token is encrypted to the Client, hence it is not accessible to potential attackers unless of disclosure of the Client's key.

Breaking Client's key by brute force would require an attacker continuously polling the Authorization Server in order to make it issue several ephemeral keys, perhaps bound to the same parameters such as time of the request, user and client identifiers, resource to be accessed. Other than being fully accountable by the Authorization Server, these attacks are unlikely to succeed, because the ABE key-issuer *randomically* generates a different valid key at each request.

To improve Client's key confidentiality, however, it is pragmatically suggested that a second temporal attribute is used inside the access structure that generates the key itself (equation (5)), so that the Client's key may expire and be regularly renewed. The Client may obtain fresh and uncompromised keys through secure distribution channels from the Authorization Server which, at Client's request, may provide rotated keys¹².

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.

OpenID Connects leverages on the semantics inside a JWT. However, as noted, not all parties are aware of that (we noted above that the Resource Server does not know whether and which resources the user has authorized access to, even if this information is *written* in the token.). UL-

```

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 successful authorization
       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. 7. Flow using OpenID Connect Implicit, Attribute-Based Encryption, challenge/response authentication, modeled in AnB specifications language. 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 simplification, 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 (is possible for an attacker to spoof any parameter of type Agent, as it happens with parameters of type Number). No attack found till two simultaneous runs.

timately, ABE deploys this common semantics to all parties.

For instance, the double decryption 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 ensures that the token presenter, currently owning the ephemeral key, is the legitimate Client. The Client's key (bound to the Client's identity) and the ephemeral key (bearing permissions) are generated at two different instants 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 or any other proof of possession protocol. In this case, equation (6) and (12) would be simply replaced by

$$z = \{x\}_{MPK_S} \quad (6bis)$$

(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.

¹² 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

$$x = \{z\}_k^{-1} \quad (12\text{bis})$$

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 removing encryption to the Client would imply an extra step to protect the confidentiality of the returned JWT, falling back to a traditional "Code for an Access Token" passage to protect the confidentiality of the returned JWT.

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 an access policy 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. It is this "reverted approach" that overcomes the need of a traditional token signature and consequently the issue presented in [15] ("*Decryption/Signature verification input is always under attacker's control*" in JOSE).

6 EVALUATION

6.1 Setup

To evaluate the proposed protocol, we implemented¹³ 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 frontend) 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, as suggested in section 5, 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 application server on the underlying layers. The popular Nimbus JOSE + JWT framework was extended with the addition of a KP-ABE/CP-ABE Encrypter and Decrypter, wrapping the OpenABE library by Zeutro¹⁴. 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 [43]). 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 an access policy, challenge response by decrypting the ciphertext on the Client. In lab environment, all servers were running on the same host as a separate Linux Ubuntu 20.04 LTS Subsystems 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-world 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. Ephemeral keys, instead, are returned (encrypted) to the Client upon a traditional OIDC request. We performed 500 key generation stress tests using a uniformly distributed variable number

TABLE 1
EPHEMERAL KEY GENERATION

Attributes (No.)	Key Length (base64 octects)	Key Length (bits)	Single thread		Multi-threading	
			Mean (ms)	Std dev. (ms)	Mean (ms)	Std dev. (ms)
1	828	4.968	80,23	12,90	364,04	183,67
2	976	5.856	79,70	10,98	406,12	181,13
3	1084	6.504	81,47	10,35	406,00	178,02
4	1228	7.368	80,36	9,29	324,24	178,38
5	1376	8.256	81,30	7,57	370,73	175,34
-	-	-	80,59	10,16	375,74	180,92

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

or resource attributes (as from equation (9)), varying the number n of resource attributes, in form of URI paths, from 1 to 5).

The average key generation time was 80,59 ms with a

¹³ Source code available on Github <https://github.com/netgroup/abe4jwt> (last accessed January 2021).

¹⁴ Available on Github under AGPL 3.0 license, OpenABE is compliant with ETSI TS 103 532 and provides a CCA-secure KP/CP-ABE

implementation: <https://github.com/zeutro/openabe> (last accessed October 2020).

standard deviation of 10,16 ms and a practically negligible effect due to the number of attributes (table 1). The key size was varying between 828 and 1376 Base64 octets, corresponding to 4.968-8.256 bits, much bigger than RSA keys and incomparably longer than ECDSA keys. We observe, however, that a simple key length comparison between ABE and other cryptographic algorithms would be unfair, as it would not consider that CP-ABE mathematically hides a full set of attributes inside each key (and implements a whole access control structure inside each ciphertext!). The average number of generated keys per second was 12,40 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 exactly 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 375,74 ms,

Fig. 8. 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).

but the values were more dispersed, with a very elevated standard deviation of 180,92 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 8), corresponding to the number of additional threads. Noticeably, the key generation frequency, measured at the end of the two experiments shown that multithreading is more “productive” (13,23 keys/s were generated in multithreading vs 12,35 keys/s in singlethreading) and measurements repeated at intermediate instants (every 100 key generation) confirmed these value.

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. Measurements on encryption time were performed with a 500 encryption stress tests, in both single and multithreading scenarios. The equation (6) was used when performing CP-

Fig. 9. Ciphertext size (vertical axis, in octets) as a function of plaintext size (horizontal axis, in octets), computed over two encryptions as from equation (6). The same access structure was used but with different URI paths identifying specific target resources (about 15 and 45 octets respectively). Original secret size varying between 10 and 59 octets. The ciphertext almost linearly grows with the plaintext size, with a fixed overhead of about 3200 octets and a grow ratio of 2,5:1 on each new octet added to the plaintext.

Fig. 10. 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 horizontal axis represents the time needed to complete a single test (in ms), the vertical axis the probability.

ABE encryption, with the same access structure but different resource attributes (i.e., different URI paths identifying a specific target resources at the server; each path length was in two different categories of about 15 and 45 octets respectively). The secret to encrypt was varying between 10 and 59 octets. 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 the 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 vary in length without affecting too much performance, but, as drawback, the ciphertext size suffers from the overhead due to the encapsulated key and results in a very long sequence of Base64 octets¹⁵. We found that the ciphertext almost linearly grows with the secret’s size, with a fixed overhead of about 3200 octets – not surprising, as aforementioned the ciphertext implements a whole access control structure – and a grow ratio of 2,5:1 on every new octet added to the

¹⁵ For greater efficiency, KEM only should be used, which however was

not directly exposed by the OpenABE CLI at the time of our experiment.

plaintext (Figure 9). On average, the single thread scenario reported an encryption time of 193,95 ms with a standard deviation of 21,48 ms. The multithreading experiment reported, as expected, less deterministic performances (mean 848,08 ms and standard deviation 412,65 ms) and, as in the key generation procedure, measured values were almost dispersed among five different “layers” (Figure 10). The encryption frequency was 5,15 encryptions/s in the single thread scenario and 5,80 encryptions/s in the multithreading one.

6.5 Decryption

Fig. 11. Client-Proxy interactions (sorted by their completion time, reported on the vertical axis in ms). The light blue line represents a single roundtrip in the challenge-response (i.e. a single request from the Client followed by a “Unauthorized” message from the Proxy), the green line represents the complete decryption time on the Client (while the orange line is a partial decryption achieved by Client’s secret key), the blue line represents the completion of the whole challenge-response procedure (including actual resource retrieval from a backend database).

TABLE 2
CLIENT-PROXY INTERACTIONS

	Single Roundtrip	Decryption	Completion
Mean	536,37	139,01	1769,40
Std dev.	66,44	22,60	83,44

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

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 keys, the Client, according to the HTTP Basic Authentication protocol, repeats the request to the proxy presenting the decrypted secret and receiving an answer (either access to the request resources or another “Unauthorized” message containing a new challenge). We perform a stress test on the proxy by invoking 500 sequential 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 completion time. Results are depicted in Figure 11 where tests have been sorted by completion time and summarized in Table 2. The average decryption time was 139,01 ms with a standard deviation of 22,60 ms, while the whole procedure completion – less meaningful as it includes resource retrieval operations at the backend database – took on average 1,77 s (standard deviation 0,08 s), and a single roundtrip 0,54 s (standard deviation 0,07 s). As network time was considered negligible, we interpreted these data as mostly due to the internal server operations. In almost all our experiments, the decryption time was an order of magnitude lower than the procedure completion time, and proved to be almost constant, witnessing the feasibility of a distributed access control protocol leveraging on ABE.

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 across the Internet, so called “digital ecosystems” typically developed around an identity provider, while users may maintain control through their devices. Identifying people and providing them proper authorizations even across these ecosystems is a fundamental issue which is becoming more and more critical as organizations progressively moves their business from the real to the virtual world.

Web tokens represent a popular developers’ choice for signaling and conveying session information in Cloud SaaS architectures, only second to the traditional authentication via username/password. The JSON Web Token (JWT) provides a standard token format that is widely used by application developers. 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.

Using a model checker, we analyzed handling of JWT in the popular OpenID Connect Authorization Code flow, understanding its vulnerabilities and existing state of the art mitigation measures. Despite these measures prove to be effective, they introduce additional complexity, making implementations more difficult and costly to certificate.

To achieve a simpler and effective software design, we developed a protocol combining the original OpenID Connect idea of a simple triangulation between involved parties with Attribute-Based Encryption, a cryptographic technique merging confidentiality protection with access control. By leveraging on ABE’s unique feature of generating randomized encryption keys from a chosen set of strings and regular expressions, our protocol natively introduces a distributed, fine-granule policy-based resource access control mechanism. While tokens do not enforce

policy by their own, but require a coordination of software components to do so, in our approach policy enforcement is automatically achieved through cryptographic schemes – not by software! – which only requires sharing of a single Master Public Key (just as in traditional public key cryptography). Results from our evaluation on a prototype implementation are encouraging and prove the viability of our approach.

The resulting protocol may work across different resource and identity providers, being potentially able to join different independent domains. Therefore this work entails a shift of focus from the problem of sharing identity attributes in the context of a single Identity/Platform provider to a more general perspective of controlling user's data processing and reuse across different and independent organizations in even wider digital ecosystems than the ones today deployed. Visionarily, we think that these new, wider ecosystems would probably be developed around new kind of businesses, similar to today's Certification Authorities, which already widely operate cryptographic services on the Internet.

ACKNOWLEDGMENT

Prof. Giuseppe Bianchi from Networking Group, University of Rome Tor Vergata is a computer and network security enthusiast and evangelist. Indeed, he was the main inspirator of this research.

The idea of a challenge-response authentication using an ABE-based access structure is owed to a discussion with cryptographers Dr. Pascal Pallier and Dr. Christoph Striecks, in the context of the ETSI STF529 project.

The author specially thanks all ETSI STF529 team members for their insightful ideas and suggestions.

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] 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.
- [5] 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.
- [6] R. Dampousse, "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.
- [7] M. Jones, J. Bradley and N. Sakimura, IETF RFC 7515 "JSON Web Signature (JWS)".
- [8] M. Jones and J. Hildebrand, IETF RFC 7516 "JSON Web Encryption (JWE)".
- [9] M. Jones, IETF RFC 7515 "JSON Web Algorithms (JWA)".
- [10] Madden, N., "Misuse-resistant cryptography for JOSE/JWT". Position paper. 2018.
- [11] S. Langkemper, "Attacking JWT authentication", online <https://www.sjoerdlangkemper.nl/2016/09/28/attacking-jwt-authentication/> last accessed on 15th January 2020.
- [12] 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.
- [13] C.K. Ho. Descriptive Research for JWT Implementation as Session Data. March 2018. <http://doi.org/10.13140/RG.2.2.24623.02728>
- [14] 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.
- [15] 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>
- [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] S. Sloatweg, "Stop Using JWT for Sessions!", online <http://crypto.net/~joepie91/blog/2016/06/19/stop-using-jwt-for-sessions-part-2-why-your-solution-doesnt-work/> last accessed on 15th January 2020.
- [19] M. Jones, J. Bradley and H. Tschofenig, IETF RFC 7800 "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)".
- [20] D. Fett, J. Bradley, T. Lodderstedt, M. Jones, D. Waite, OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP), draft-ietf-oauth-dpop-04, 4th October 2021, <https://data-tracker.ietf.org/doc/html/draft-ietf-oauth-dpop>, last accessed on 10th October 2021.
- [21] M. Jones, N. Sakimura, RFC7638, JSON Web Key (JWK) Thumbprint, September 2015, draft-ietf-jose-jwk-thumbprint-08, <https://data-tracker.ietf.org/doc/html/rfc7638>, last accessed on 10th October 2021.
- [22] T. Lodderstedt, J. Bradley, A. Labunets, D. Fett, OAuth 2.0 Security Best Current Practice, draft-ietf-oauth-security-topics-18, 12 April 2021, last accessed on 10th October 2021.
- [23] B. Campbell, J. Bradley, N. Sakimura, T. Lodderstedt, OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens, February 2020, <https://data-tracker.ietf.org/doc/html/rfc8705>, last accessed on 10th October 2021.
- [24] M. Watson (Ed.), Web Cryptography API, W3C Recommendation 26 January 2017, <https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/>, last accessed on 10th October 2021.
- [25] K. Meyer zu Selhausen, D. Fett, OAuth 2.0 Authorization Server Issuer Identification, draft-ietf-oauth-iss-auth-resp-02, 8 June 2021, <https://data-tracker.ietf.org/doc/html/draft-ietf-oauth-iss-auth-resp-02>, last accessed on 10th October 2021.
- [26] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. International Journal of Information Security, 4(3):181–208, 2005.
- [27] S. Mödersheim. Algebraic properties in Alice and Bob notation. In International Conference on Availability, Reliability and Security (ARES 2009), pages 433–440, 2009.

- [28] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on information Theory*, 2(29), 1983.
- [29] L. Holtmann, "Single Sign-On Security: Security Analysis of real-life OpenID Connect Implementations.
- [30] G. Lowe. A hierarchy of authentication specification. In 10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA, pages 31–44. IEEE Computer Society, 1997.
- [31] IETF RFC 7636, N. Sakimura, J. Bradley, N. Agarwal, Proof Key for Code Exchange by OAuth Public Clients, <https://data-tracker.ietf.org/doc/html/rfc7636>, last accessed on 10th October 2021.
- [32] J. Richer (Ed.), IETF RFC7662 "OAuth 2.0 Token Introspection".
- [33] A. Armando, C. Carbone, L. Compagna, J. Cuellar, L. Tobarra. (2008). Formal analysis of SAML 2.0 web browser single sign-on. 1-10. Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008, Alexandria, VA, USA, October 27, 2008 DOI 10.1145/1456396.1456397.
- [34] A. Shamir. Identity-based cryptosystems and signature schemes. In Proceedings of CRYPTO 84 on Advances in cryptology, pages 47–53. Springer-Verlag New York, Inc., 1984.
- [35] ETSI TS 103 532: "CYBER; Attribute-Based Encryption for Attribute-Based Access Control".
- [36] A. Sahai and B. Waters. Fuzzy identity-based encryption. In EUROCRYPT, pages 457-473, 2005.
- [37] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
- [38] 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.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] J. Hallam-Baker, P. Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Stewart, L., IETF RFC 2617, "HTTP Authentication: Basic and Digest Access Authentication".
- [43] Ambrosin, M., Conti, M. and 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).
- [44] S. Mödersheim and L. Viganò. Sufficient conditions for vertical composition of security protocols. In Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, pages 435–446, New York, NY, USA, 2014. ACM

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 STF529 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).

ANNEX 1

OpenID Connect flow models and their analysis by Opensource Fixedpoint Model Checker.

```
#
# A model for OpenID Connect Authorization Code Flow implemented in AnB
# specification language. To use this file, download the checker:
# Open-Source Fixed-Point Model-Checker (OFMC)
# from https://www.imm.dtu.dk/~samo/
# (last access on 13th October 2021)
#
# 00_Auth_Code_noPKCE_noNonce_Flawed
#

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

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

Goals:

RS authenticates C on RS,resource(code(Scope,State,Nonce)),C,as,Session
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)
```

Listing 1. Modeling OpenID Connect Authentication Code Flow using Opensource Fixedpoint Model Checker (OFMC). Client's initial parameters (State, Nonce) are exposed to the attacker.

```

Open-Source Fixedpoint Model-Checker version 2020
INPUT:
  oidc.anb
SUMMARY:
  ATTACK_FOUND
GOAL:
  weak_auth
BACKEND:
  Open-Source Fixedpoint Model-Checker version 2020
STATISTICS:
  TIME 168 ms
  parseTime 4 ms
  visitedNodes: 28 nodes
  depth: 6 plies

ATTACK TRACE:
(x901,1) -> i: Scope(1)
i -> (x901,1): {Scope(1),as,x407}_inv(authChCr(i))

#attacker asks the as for a code on behalf of C
#assumed user authorizes, a code is returned
i -> (as,1): i,x306,x307,x308
(as,1) -> i: x307,code(x306,x307,x308),x306

#attacker impersonates as and return C a wrong code
(x901,1) -> i: i,Scope(1),State(3),Nonce(3)
i -> (x901,1): State(3),code(x306,x307,x308),Scope(1)

#C's code4token exchange
(x901,1) -> i: pseudonym(x901),{|x901,pw(as,x901),code(x306,x307,x308)|}_ (secChCr(pseudonym(x901),as))
i -> (as,1): pseudonym(x901),{|x901,pw(as,x901),code(x306,x307,x308)|}_ (secChCr(pseudonym(x901),as))
(as,1) -> i: {|resource(code(x306,x307,x308)),x901,as,i}_inv(pk(as)),code(x306,x307,x308)|}_ (secChCr(as,pseudonym(x901)))
i -> (x901,1): {|resource(code(x306,x307,x308)),x901,as,i}_inv(pk(as)),code(x306,x307,x308)|}_ (secChCr(as,pseudonym(x901)))

#C uses the wrong token
(x901,1) -> i: pseudonym(x901),{|resource(code(x306,x307,x308)),x901,as,i}_inv(pk(as)),x407|}_ (secChCr(pseudonym(x901),i))
i -> (x901,1): {|x812,x407|}_ (secChCr(i,pseudonym(x901)))

% Reached State:
%
% request(x901,as,pCasStateScopecodeScopeStateNonce,State(3),Scope(1),code(x306,x307,x308),1)
% contains(secretset(x901,1,pData),x901)
% contains(secretset(x901,1,pData),i)
% secrets(x812,secretset(x901,1,pData),i)
% request(x901,i,pCRSData,x812,1)
%
state_rC(x901,5,hash,pw(as,x901),pk(as),i,Scope(1),x407,as,Scope(1),as,x407,State(3),Nonce(3),i,Scope(1),State(3),Nonce(3),code(x306,x307,x308),State(3),code(x306,x307,x308),Scope(1),x901,pw(as,x901),code(x306,x307,x308),resource(code(x306,x307,x308)),{resource(code(x306,x307,x308)),x901,as,i}_inv(pk(as)),{resource(code(x306,x307,x308)),x901,as,i}_inv(pk(as)),code(x306,x307,x308),{resource(code(x306,x307,x308)),x901,as,i}_inv(pk(as)),x407,x812,x812,x407,1)
% state_rRS(x29,0,pk(as),as,1)
%
state_ras(as,2,hash,resource,code,pw(as,x901),x901,inv(pk(as)),pk(as),x308,x307,x306,i,i,x306,x307,x308,x307,code(x306,x307,x308),x306,code(x306,x307,x308),x901,pw(as,x901),code(x306,x307,x308),pseudonym(x901),{resource(code(x306,x307,x308)),x901,as,i}_inv(pk(as)),code(x306,x307,x308),1)
% witness(as,x901,pCasStateScopecodeScopeStateNonce,x307,x306,code(x306,x307,x308))
% witness(x901,as,pasCCpwasCcodeScopeStateNonce,x901,pw(as,x901),code(x306,x307,x308))

```

```
% wrequest(as,x901,pasCCpwasCodeScopeStateNonce,x901,pw(as,x901),code(x306,x307,x308),i)
% witness(x901,i,pRSCRSresourcecodeScopeStateNonceCasSession,i,resource(code(x306,x307,x308)),x901,as,x407)
```

Listing 2. Output from OFMC using Listing 1. Because she knows the initial Client's parameters, the attacker may impersonate the Authorization Server.

```
#
# A model for OpenID Connect Authorization Code Flow implemented in AnB
# specification language. To use this file, download the checker:
# Open-Source Fixed-Point Model-Checker (OFMC)
# from https://www.imm.dtu.dk/~samo/
# (last access on 13th October 2021)
#
# 01_Auth_Code_noPKCE_noNonce_Flawed
#

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

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

Goals:
RS authenticates C on RS,resource(code(Scope,State,Nonce)),C,as,Session
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)
```

Listing 3. Same model as in Listing 1 except that Client's initial parameters are now protected.

```
Open-Source Fixedpoint Model-Checker version 2020
Verified for 1 sessions
INPUT:
  oidc.anb
SUMMARY:
  ATTACK_FOUND
GOAL:
  weak_auth
BACKEND:
  Open-Source Fixedpoint Model-Checker version 2020
STATISTICS:
  TIME 13008 ms
  parseTime 4 ms
  visitedNodes: 3063 nodes
  depth: 9 plies

ATTACK TRACE:
(x1201,1) -> i: Scope(1)
(x1201,2) -> i: Scope(2)
i -> (x1201,1): {Scope(1), as, x407}_inv(authChCr(i))
i -> (x1201,2): {Scope(2), as, x607}_inv(authChCr(i))

#C(1) asks for code using Scope(1)
(x1201,1) -> i: pseudonym(x1201), {|i, Scope(1), State(3), Nonce(3)|}_ (secChCr(pseudonym(x1201), as))
i -> (as,1): pseudonym(x1201), {|i, Scope(1), State(3), Nonce(3)|}_ (secChCr(pseudonym(x1201), as))
(as,1) -> i: State(3), code(Scope(1), State(3), Nonce(3)), Scope(1)

#C(2) asks for code using Scope(2)
(x1201,2) -> i: pseudonym(x1201), {|i, Scope(2), State(5), Nonce(5)|}_ (secChCr(pseudonym(x1201), as))
i -> (as,2): pseudonym(x1201), {|i, Scope(2), State(5), Nonce(5)|}_ (secChCr(pseudonym(x1201), as))
(as,2) -> i: State(5), code(Scope(2), State(5), Nonce(5)), Scope(2)

#C(1) is returned a code associated to Scope(2)
i -> (x1201,1): State(3), code(Scope(2), State(5), Nonce(5)), Scope(1)

#C(1)'s code4token exchange
(x1201,1) -> i: pseudonym(x1201), {|x1201, pw(as, x1201), code(Scope(2), State(5), Nonce(5))|}_ (secChCr(pseudonym(x1201), as))
i -> (as,2): pseudonym(x1201), {|x1201, pw(as, x1201), code(Scope(2), State(5), Nonce(5))|}_ (secChCr(pseudonym(x1201), as))
(as,2) -> i: {|resource(code(Scope(2), State(5), Nonce(5))), x1201, as, i}_inv(pk(as)), code(Scope(2), State(5), Nonce(5))|}_ (secChCr(as, pseudonym(x1201)))
i -> (x1201,1): {|resource(code(Scope(2), State(5), Nonce(5))), x1201, as, i}_inv(pk(as)), code(Scope(2), State(5), Nonce(5))|}_ (secChCr(as, pseudonym(x1201)))

#C(1) presents the wrong token
(x1201,1) -> i: pseudonym(x1201), {|resource(code(Scope(2), State(5), Nonce(5))), x1201, as, i}_inv(pk(as)), x407|}_ (secChCr(pseudonym(x1201), i))
i -> (x1201,1): {|x1112, x407|}_ (secChCr(i, pseudonym(x1201)))

% Reached State:
%
% request(x1201, as, pCasStateScopecodeScopeStateNonce, State(3), Scope(1), code(Scope(2), State(5), Nonce(5)), 1)
% contains(secretset(x1201,1,pData), x1201)
% contains(secretset(x1201,1,pData), i)
% secrets(x1112, secretset(x1201,1,pData), i)
% request(x1201, i, pCRSData, x1112, 1)
```



```

%
state_rC(x1201,5,hash,pw(as,x1201),pk(as),i,Scope(1),x407,as,Scope(1),as,x407,State(3),Nonce(3),i,Scope(1),State(3),Nonce(3),code(Scope(
2),State(5),Nonce(5)),State(3),code(Scope(2),State(5),Nonce(5)),Scope(1),x1201,pw(as,x1201),code(Scope(2),State(5),Nonce(5)),re-
source(code(Scope(2),State(5),Nonce(5))),{resource(code(Scope(2),State(5),Nonce(5))),x1201,as,i}_inv(pk(as)),{re-
source(code(Scope(2),State(5),Nonce(5))),x1201,as,i}_inv(pk(as)),code(Scope(2),State(5),Nonce(5)),{re-
source(code(Scope(2),State(5),Nonce(5))),x1201,as,i}_inv(pk(as)),x407,x1112,x1112,x407,1)
% state_rRS(x34,0,pk(as),as,2)
% state_rC(x1201,2,hash,pw(as,x1201),pk(as),i,Scope(2),x607,as,Scope(2),as,x607,State(5),Nonce(5),i,Scope(2),State(5),Nonce(5),2)
%
state_ras(as,2,hash,re-
source,code,pw(as,x1201),x1201,inv(pk(as)),pk(as),Nonce(5),State(5),Scope(2),i,i,Scope(2),State(5),Nonce(5),pseudo-
nym(x1201),State(5),code(Scope(2),State(5),Nonce(5)),Scope(2),code(Scope(2),State(5),Nonce(5)),x1201,pw(as,x1201),code(Scope(2),State(5)
,Nonce(5)),{resource(code(Scope(2),State(5),Nonce(5))),x1201,as,i}_inv(pk(as)),code(Scope(2),State(5),Nonce(5)),2)
% state_rRS(x29,0,pk(as),as,1)
% state_ras(as,1,hash,resource,code,pw(as,x27),x27,inv(pk(as)),pk(as),Nonce(3),State(3),Scope(1),i,i,Scope(1),State(3),Nonce(3),pseudo-
nym(x1201),State(3),code(Scope(1),State(3),Nonce(3)),Scope(1),1)
% witness(as,x27,pCasStateScopecodeScopeStateNonce,State(3),Scope(1),code(Scope(1),State(3),Nonce(3)))
% witness(as,x1201,pCasStateScopecodeScopeStateNonce,State(5),Scope(2),code(Scope(2),State(5),Nonce(5)))
% witness(x1201,as,pasCCpwasCcodeScopeStateNonce,x1201,pw(as,x1201),code(Scope(2),State(5),Nonce(5)))
% wrequest(as,x1201,pasCCpwasCcodeScopeStateNonce,x1201,pw(as,x1201),code(Scope(2),State(5),Nonce(5)),i)
% witness(x1201,i,pRSCRSresourcecodeScopeStateNonceCasSession,i,resource(code(Scope(2),State(5),Nonce(5))),x1201,as,x407)

```

Listing 4. Output from OFMC using Listing 3. Code is leaked to the attacker who can later inject it into a second Client's session.

```

#
# A model for OpenID Connect Authorization Code Flow implemented in AnB
# specification language. To use this file, download the checker:
# Open-Source Fixed-Point Model-Checker (OFMC)
# from https://www.imm.dtu.dk/~samo/
# (last access on 13th October 2021)
#
# 02_Auth_Code_noPKCE_Nonce_OK
#

Protocol: OI DC_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 successful authorization
        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;
        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
as*->*[C]:{resource(code(Scope,State,Nonce)),C,as,RS,Nonce}inv(pk(as)),code(Scope,State,Nonce)

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

Goals:
RS authenticates C on RS,resource(code(Scope,State,Nonce)),C,as,Session
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)

```

Listing 5. The model in Listing 3 is amended in order to include the Nonce parameter inside the returned token, so that the Client can explicitly check it. No attack found using this model.

```

#
# A model for OpenID Connect Authorization Code Flow implemented in AnB
# specification language. To use this file, download the checker:
# Open-Source Fixed-Point Model-Checker (OFMC)
# from https://www.imm.dtu.dk/~samo/
# (last access on 13th October 2021)
#
# 03_Auth_Code_PKCE_noNonce_Flawed
#

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

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

Goals:
RS authenticates C on RS,resource(code(Scope,State,Nonce)),C,as,Session
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)

```

Listing 6. Same model as in Listing 5 except that Proof Key for Code Exchange (PKCE) is introduced. However, no Nonce is returned in the token.

```

Open-Source Fixedpoint Model-Checker version 2020
Verified for 1 sessions
INPUT:
  oidc.anb
SUMMARY:
  ATTACK_FOUND
GOAL:
  weak_auth
BACKEND:
  Open-Source Fixedpoint Model-Checker version 2020
STATISTICS:
  TIME 12040 ms
  parseTime 0 ms
  visitedNodes: 4560 nodes
  depth: 10 plies

ATTACK TRACE:
(x1301,1) -> i: Scope(1)
(x1301,2) -> i: Scope(2)
i -> (x1301,1): {Scope(1),as,x407}_inv(authChCr(i))
i -> (x1301,2): {Scope(2),as,x607}_inv(authChCr(i))

#Client(1) asks for code with Scope(1)
(x1301,1) -> i: pseudonym(x1301),{|i,Scope(1),State(3),Nonce(3),hash(Verifier(3))|}_ (secChCr(pseudonym(x1301),as))
i -> (as,1): pseudonym(x1301),{|i,Scope(1),State(3),Nonce(3),hash(Verifier(3))|}_ (secChCr(pseudonym(x1301),as))
(as,1) -> i: State(3),code(Scope(1),State(3),Nonce(3)),Scope(1)

#Client(2) asks for code with Scope(2), but the attacker returns the same code to both Clients
(x1301,2) -> i: pseudonym(x1301),{|i,Scope(2),State(5),Nonce(5),hash(Verifier(5))|}_ (secChCr(pseudonym(x1301),as))
i -> (as,2): pseudonym(x1301),{|i,Scope(2),State(5),Nonce(5),hash(Verifier(5))|}_ (secChCr(pseudonym(x1301),as))
(as,2) -> i: State(5),code(Scope(2),State(5),Nonce(5)),Scope(2)
i -> (x1301,1): State(3),code(Scope(2),State(5),Nonce(5)),Scope(1)
i -> (x1301,2): State(5),code(Scope(2),State(5),Nonce(5)),Scope(2)

#Client(2) code4token (incomplete)
(x1301,2) -> i: pseudonym(x1301),{|x1301,pw(as,x1301),code(Scope(2),State(5),Nonce(5)),Verifier(5)|}_ (secChCr(pseudonym(x1301),as))
i -> (as,2): pseudonym(x1301),{|x1301,pw(as,x1301),code(Scope(2),State(5),Nonce(5)),Verifier(5)|}_ (secChCr(pseudonym(x1301),as))
(as,2) -> i: {|{resource(code(Scope(2),State(5),Nonce(5))),x1301,as,i}_inv(pk(as)),code(Scope(2),State(5),Nonce(5))|}_ (secChCr(as,pseudonym(x1301)))

```

```

#Client(1)'s code4token: the attacker returns the wrong token
(x1301,1) -> i: pseudonym(x1301),{|x1301,pw(as,x1301),code(Scope(2),State(5),Nonce(5)),Verifier(3)|}_ (secChCr(pseudonym(x1301),as))
i -> (x1301,1): {|{resource(code(Scope(2),State(5),Nonce(5))),x1301,as,i}_inv(pk(as)),code(Scope(2),State(5),Nonce(5))|}_ (sec-
ChCr(as,pseudonym(x1301)))

#Client(1) uses the wrong token
(x1301,1) -> i: pseudonym(x1301),{|{resource(code(Scope(2),State(5),Nonce(5))),x1301,as,i}_inv(pk(as)),x407|}_ (secChCr(pseudo-
nym(x1301),i))
i -> (x1301,1): {|x1213,x407|}_ (secChCr(i,pseudonym(x1301)))

% Reached State:
%
% request(x1301,as,pCasStateScopecodeScopeStateNonce,State(3),Scope(1),code(Scope(2),State(5),Nonce(5)),1)
% contains(secretset(x1301,1,pData),x1301)
% contains(secretset(x1301,1,pData),i)
% secrets(x1213,secretset(x1301,1,pData),i)
% request(x1301,i,pCRSData,x1213,1)
%
% state_rC(x1301,5,hash,pw(as,x1301),pk(as),i,Scope(1),x407,as,Scope(1),as,x407,State(3),Nonce(3),Veri-
fier(3),i,Scope(1),State(3),Nonce(3),hash(Verifier(3)),hash(Veri-
fier(3)),code(Scope(2),State(5),Nonce(5)),State(3),code(Scope(2),State(5),Nonce(5)),Scope(1),x1301,pw(as,x1301),code(Scope(2),State(5),N
once(5)),Verifier(3),resource(code(Scope(2),State(5),Nonce(5))),{resource(code(Scope(2),State(5),Nonce(5))),x1301,as,i}_inv(pk(as))},{re-
source(code(Scope(2),State(5),Nonce(5))),x1301,as,i}_inv(pk(as)),code(Scope(2),State(5),Nonce(5))},{re-
source(code(Scope(2),State(5),Nonce(5))),x1301,as,i}_inv(pk(as)),x407,x1213,x1213,x407,1)
% state_rRS(x34,0,pk(as),as,2)
%
% state_ras(as,2,hash,resource,code,pw(as,x1301),x1301,inv(pk(as)),pk(as),hash(Veri-
fier(5)),Nonce(5),State(5),Scope(2),i,i,Scope(2),State(5),Nonce(5),hash(Verifier(5)),pseudo-
nym(x1301),State(5),code(Scope(2),State(5),Nonce(5)),Scope(2),code(Scope(2),State(5),Nonce(5)),Veri-
fier(5),x1301,pw(as,x1301),code(Scope(2),State(5),Nonce(5)),Verifier(5),{re-
source(code(Scope(2),State(5),Nonce(5))),x1301,as,i}_inv(pk(as)),code(Scope(2),State(5),Nonce(5)),2)
%
% state_rC(x1301,3,hash,pw(as,x1301),pk(as),i,Scope(2),x607,as,Scope(2),as,x607,State(5),Nonce(5),Veri-
fier(5),i,Scope(2),State(5),Nonce(5),hash(Verifier(5)),hash(Veri-
fier(5)),code(Scope(2),State(5),Nonce(5)),State(5),code(Scope(2),State(5),Nonce(5)),Scope(2),x1301,pw(as,x1301),code(Scope(2),State(5),N
once(5)),Verifier(5),2)
% state_rRS(x29,0,pk(as),as,1)
%
% state_ras(as,1,hash,resource,code,pw(as,x27),x27,inv(pk(as)),pk(as),hash(Veri-
fier(3)),Nonce(3),State(3),Scope(1),i,i,Scope(1),State(3),Nonce(3),hash(Verifier(3)),pseudo-
nym(x1301),State(3),code(Scope(1),State(3),Nonce(3)),Scope(1),1)
% witness(as,x27,pCasStateScopecodeScopeStateNonce,State(3),Scope(1),code(Scope(1),State(3),Nonce(3)))
% witness(as,x1301,pCasStateScopecodeScopeStateNonce,State(5),Scope(2),code(Scope(2),State(5),Nonce(5)))
% witness(x1301,as,pasCCpwasCcodeScopeStateNonce,x1301,pw(as,x1301),code(Scope(2),State(5),Nonce(5)))
% wrequest(as,x1301,pasCCpwasCcodeScopeStateNonce,x1301,pw(as,x1301),code(Scope(2),State(5),Nonce(5)),i)
% witness(x1301,i,pRSCRSresourcecodeScopeStateNonceCasSession,i,resource(code(Scope(2),State(5),Nonce(5))),x1301,as,x407)

```

Listing 7. Output from OFMC using Listing 6. Without Nonce parameter checking, code injection is still possible.

```

#
# A model for OpenID Connect Authorization Code Flow implemented in AnB
# specification language. To use this file, download the checker:
# Open-Source Fixed-Point Model-Checker (OFMC)
# from https://www.imm.dtu.dk/~samo/
# (last access on 13th October 2021)
#
# 04_Auth_Code_PKCE_Nonce_OK
#
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 successful authorization
    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;
    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
as*->*[C]:{resource(code(Scope,State,Nonce)),C,as,RS,Nonce}inv(pk(as)),code(Scope,State,Nonce)

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

Goals:
    RS authenticates C on RS,resource(code(Scope,State,Nonce)),C,as,Session
    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)

```

Listing 8. Same model as in Listing 6 but including a Nonce inside the returned token. PKCE is used as well. Checking this model results in a safe flow.

```

#
# A model for OpenID Connect Authorization Code Flow implemented in AnB
# specification language. To use this file, download the checker:
# Open-Source Fixed-Point Model-Checker (OFMC)
# from https://www.imm.dtu.dk/~samo/
# (last access on 13th October 2021)
#
# 05_Auth_Code_PKCE_DPOP_OK
#

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)

```

Listing 9. Same model as in Listing 8 using the Demonstrating Proof-of-Possession (DPoP) protocol. Even if token may be leaked to an attacker (it is now transmitted under a no more confidential channel), no attack is found, as long as the presented signature is over sufficient parameters to make it prevent reply attacks (the signature is over RS and Scope parameters).

```

#
# A model for OpenID Connect Authorization Code Flow implemented in AnB
# specification language. To use this file, download the checker:
# Open-Source Fixed-Point Model-Checker (OFMC)
# from https://www.imm.dtu.dk/~samo/
# (last access on 13th October 2021)
#
# 06_Auth_Code_hash(session)_Flawed
#

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
        nonce, #OIDC AuthCodeFlow parameter (computed from Session)
    Number Scope, #OIDC AuthCodeFlow parameter
    State, #OIDC AuthCodeFlow parameter
    Data, #RS returns this data on successful authorization
    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,nonce;
    where RS!=C, RS!=as, C!=as

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

[C]*->*as: RS,Scope,State,nonce(Session) #nonce is computed from Session
as->[C]: State,code(Scope,State,nonce(Session)),Scope

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

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

Goals:
RS authenticates C on RS,resource(code(Scope,State,nonce(Session))),C,as,Session
C authenticates RS on Data
Data secret between RS,C
C authenticates as on State, Scope,code(Scope,State,nonce(Session))
as weakly authenticates C on C,pw(as,C),code(Scope,State,nonce(Session))

```

Listing 10. Same model as in Listing 5. Nonce is now calculated from Session. Note that no PKCE nor DPoP is used.

```

Open-Source Fixedpoint Model-Checker version 2020
INPUT:
    oidc.anb
SUMMARY:
    ATTACK_FOUND
GOAL:
    weak_auth
BACKEND:
    Open-Source Fixedpoint Model-Checker version 2020
STATISTICS:
    TIME 156 ms
    parseTime 8 ms
    visitedNodes: 16 nodes
    depth: 6 plies

#i can simply intercept the token and use it with the RS
ATTACK TRACE:
(x902,1) -> i: Scope(1)
i -> (x901,1): Scope(1)
(x901,1) -> i: {Scope(1),as,Session(2)}_inv(authChCr(x901))

```

```

i -> (x902,1): {Scope(1),as,Session(2)}_inv(authChCr(x901))

(x902,1) -> i: pseudonym(x902),{|x901,Scope(1),State(3),nonce(Session(2))|}_ (secChCr(pseudonym(x902),as))
i -> (as,1): pseudonym(x902),{|x901,Scope(1),State(3),nonce(Session(2))|}_ (secChCr(pseudonym(x902),as))
(as,1) -> i: State(3),code(Scope(1),State(3),nonce(Session(2))),Scope(1)
i -> (x902,1): State(3),code(Scope(1),State(3),nonce(Session(2))),Scope(1)

(x902,1) -> i: pseudonym(x902),{|x902,pw(as,x902),code(Scope(1),State(3),nonce(Session(2))|}_ (secChCr(pseudonym(x902),as))
i -> (as,1): pseudonym(x902),{|x902,pw(as,x902),code(Scope(1),State(3),nonce(Session(2))|}_ (secChCr(pseudonym(x902),as))
(as,1) -> i: {{resource(code(Scope(1),State(3),nonce(Session(2))))},x902,as,x901,nonce(Session(2))}_inv(pk(as)),code(Scope(1),State(3),nonce(Session(2))}_inv(authChCr(as))

i -> (x901,1): i,{|{resource(code(Scope(1),State(3),nonce(Session(2))))},x902,as,x901,nonce(Session(2))}_inv(pk(as)),Session(2)}_ (secChCr(i,x901))
(x901,1) -> i: {|Data(7),Session(2)}_ (secChCr(x901,i))

% Reached State:
%
% request(x901,x902,pRSCRSresourcecodeScopeStatenonceSessionCasSession,x901,resource(code(Scope(1),State(3),nonce(Session(2))))},x902,as,Session(2),1)
% state_rRS(x901,2,pk(as),as,Scope(1),Session(2),Scope(1),as,Session(2),nonce(Session(2)),x902,resource(code(Scope(1),State(3),nonce(Session(2))))}, {resource(code(Scope(1),State(3),nonce(Session(2))))},x902,as,x901,nonce(Session(2))}_inv(pk(as)), {resource(code(Scope(1),State(3),nonce(Session(2))))},x902,as,x901,nonce(Session(2))}_inv(pk(as)),Session(2),i,Data(7),Data(7),Session(2),1)
% witness(x901,x902,pCRSData,Data(7))
% secrets(Data(7),secrecyset(x901,1,pData),i)
% contains(secrecyset(x901,1,pData),x901)
% contains(secrecyset(x901,1,pData),x902)
% state_ras(as,2,hash,resource,code,pw(as,x902),x902,inv(pk(as)),pk(as),nonce(Session(2)),State(3),Scope(1),x901,x901,Scope(1),State(3),nonce(Session(2))),pseudonym(x902),State(3),code(Scope(1),State(3),nonce(Session(2))),Scope(1),code(Scope(1),State(3),nonce(Session(2))),x902,pw(as,x902),code(Scope(1),State(3),nonce(Session(2))), {resource(code(Scope(1),State(3),nonce(Session(2))))},x902,as,x901,nonce(Session(2))}_inv(pk(as)),code(Scope(1),State(3),nonce(Session(2))),1)
% state_rC(x902,3,nonce,hash,pw(as,x902),pk(as),x901,Scope(1),Session(2),as,Scope(1),as,Session(2),State(3),x901,Scope(1),State(3),nonce(Session(2)),nonce(Session(2)),code(Scope(1),State(3),nonce(Session(2))),State(3),code(Scope(1),State(3),nonce(Session(2))),Scope(1),x902,pw(as,x902),code(Scope(1),State(3),nonce(Session(2))),1)
% witness(as,x902,pCasStateScopecodeScopeStatenonceSession,State(3),Scope(1),code(Scope(1),State(3),nonce(Session(2))))
% witness(x902,as,pasCCpwasCcodeScopeStatenonceSession,x902,pw(as,x902),code(Scope(1),State(3),nonce(Session(2))))
% wrequest(as,x902,pasCCpwasCcodeScopeStatenonceSession,x902,pw(as,x902),code(Scope(1),State(3),nonce(Session(2))),i)

```

Listing 11. Output from OFMC using Listing 10. The flow is not safe, if the Session parameter is leaked to the attacker.

```

#
# A model for OpenID Connect Authorization Code Flow implemented in AnB
# specification language. To use this file, download the checker:
# Open-Source Fixed-Point Model-Checker (OFMC)
# from https://www.imm.dtu.dk/~samo/
# (last access on 13th October 2021)
#
# 07_Auth_Code_hash(session)
#

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
        nonce; #OIDC AuthCodeFlow parameter (computed from Session)
Number Scope, #OIDC AuthCodeFlow parameter
        State, #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,nonce;
        where RS!=C, RS!=as, C!=as

Actions:

C->RS:Scope
RS*->*C:Scope,as,Session #Perhaps OK as long as Session is kept confidential

[C]*->*as: RS,Scope,State,nonce(Session) #nonce is computed from Session
as->[C]: State,code(Scope,State,nonce(Session)),Scope

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

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

Goals:

RS authenticates C on RS,resource(code(Scope,State,nonce(Session))),C,as,Session
C authenticates RS on Data
Data secret between RS,C
C authenticates as on State, Scope,code(Scope,State,nonce(Session))
as weakly authenticates C on C,pw(as,C),code(Scope,State,nonce(Session))

```

Listing 12. Same model as in Listing 10 but assuming that the Session parameter is kept confidential. This flow is probably secure (no attack returned by OFMC during over 4h running, however not yet fully verified for 2 simultaneous runs).

```

#
# A model for OpenID Connect Authorization Code Flow implemented in AnB
# specification language. To use this file, download the checker:
# Open-Source Fixed-Point Model-Checker (OFMC)
# from https://www.imm.dtu.dk/~samo/
# (last access on 13th October 2021)
#
# 08_Implicit_ABE+Challenge_flawed
#

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 AuthFlow parameter
    Code, #OIDC AuthFlow parameter
    Nonce, #OIDC AuthFlow parameter
    Challenge, #challenge-response parameter
    Data, #RS returns this data on successfull authorization
    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))}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

```

Listing 13. Flow using OpenID Connect Implicit, Attribute-Based Encryption, challenge/response authentication.

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 $\{h(B, idp, SP, Scope), inv(h(B, idp, SP, Scope))\}$. 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 OFMC handles the Scope parameter as any other variable).

```

Open-Source Fixedpoint Model-Checker version 2020
INPUT:
    oidc.anb
SUMMARY:
    ATTACK_FOUND
GOAL:
    weak_auth
BACKEND:
    Open-Source Fixedpoint Model-Checker version 2020
STATISTICS:
    TIME 80 ms
    parseTime 4 ms
    visitedNodes: 45 nodes
    depth: 4 plies

ATTACK TRACE:
(x701,1) -> i: x32
i -> (as,1): x701,i,x32,x307

```

```

#the attacker grabs an ephemeral key from as, and returns it to C
(as,1) -> i: {inv(h(x701,as,i,x32))}_ (pk(x701))
i -> (x701,1): {as,{x505}_ (h(x701,as,i,x32))}_ (pk(x701))}_ inv(authChCr(i))
(x701,1) -> i: x701,i,x32,Nonce(3)
i -> (x701,1): {inv(h(x701,as,i,x32))}_ (pk(x701))

(x701,1) -> i: pseudonym(x701),{x32,x505,Session(4)}_ (secChCr(pseudonym(x701),i))
i -> (x701,1): {x609,Session(4)}_ (secChCr(i,pseudonym(x701)))

% Reached State:
%
% request(x701,as,pCasCRSScopeNonce,x701,i,x32,Nonce(3),1)
% request(x701,i,pCRSData,x609,1)
% secrets(x609,secrecyset(x701,1,pData),i)
% contains(secrecyset(x701,1,pData),i)
% contains(secrecyset(x701,1,pData),x701)
%
state_rC(x701,4,x32,h,inv(pk(x701)),pk(x701),i,{x505}_ (h(x701,as,i,x32))}_ (pk(x701)),as,{x505}_ (h(x701,as,i,x32))}_ (pk(x701)),Nonce(3),x701,i,x32,Nonce(3),x505,inv(h(x701,as,i,x32)),{inv(h(x701,as,i,x32))}_ (pk(x701)),Session(4),x32,x505,Session(4),x609,x609,Session(4),1)
% state_rRS(x28,0,h(x701,as,x28,x32),pk(x701),x32,x701,as,1)
% state_ras(as,1,inv(h(x701,as,i,x32)),pk(x701),x32,i,x701,x307,x701,i,x32,x307,{inv(h(x701,as,i,x32))}_ (pk(x701)),1)
% witness(as,x701,pCasCRSScopeNonce,x701,i,x32,x307)
% witness(x701,i,pRSCChallenge,x505)

```

Listing 14. Output from OFMC using Listing 13. The flow is not safe, as the attacker may impersonate the Authorization Server and returns an ephemeral key to the Client.

```

#
# A model for OpenID Connect Authorization Code Flow implemented in AnC
# specification language. To use this file, download the checker:
# Open-Source Fixed-Point Model-Checker (OFMC)
# from https://www.imm.dtu.dk/~samo/
# (last access on 13th October 2021)
#
# 09_Implicit_ABE+Challenge_OK
#

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 successful authorization
        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

```

Listing 15. Same model as in Listing 13, but including the Nonce parameter in the returned ciphertext. The Client is able to check that the message is from the legitimate Authorization Server and not from an attacker. No attack was found till two simultaneous runs.