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

Giovanni Bartolomeo

CNIT

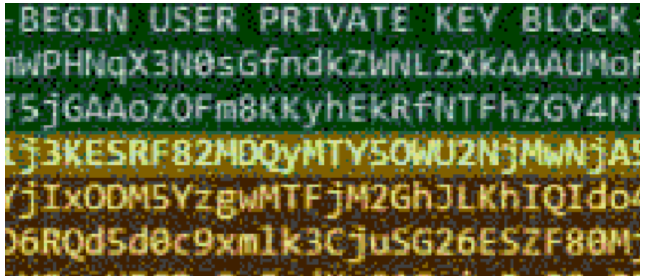
(giovanni.bartolomeo AT uniroma2.it)

Good Morning!

1. About myself: https://www.linkedin.com/in/giovannibartolomeo/

consorzio nazionale interuniversitario per le telecomunicazioni

2. About CNIT: https://www.cnit.it/



3. Some resources about this work: https://github.com/netgroup/abe4jwt

Why this work?

1. Today, OAuth2/OpenID Connect 1.0 is the most common auth mechanism on the web, just after static username/password auth

2. Broken Access Control is OSWAP#1 Application Security Risk in 2021

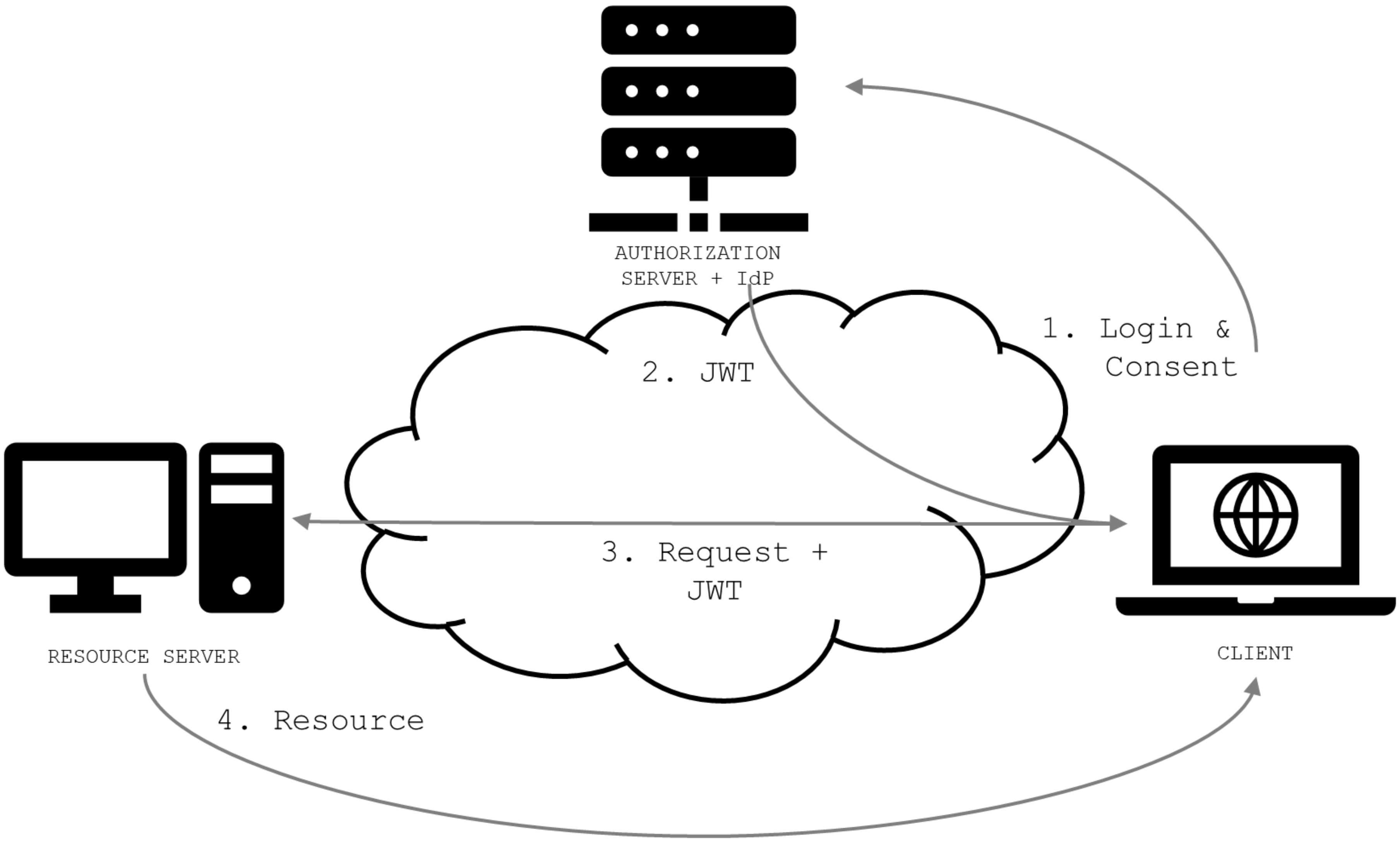
3. OAuth2 specs increasing their complexity as soon as new vulnerabilities are found

4. Using predicate encryption instead of traditional token signature we can achieve a simpler and more effective design

The ideal Oauth flow (**Implicit Grant**).

Interesting part of the protocol under investigation is the **authz req and res**, which happens through http GET cross site/server side requests.

Implicit is unsecure as parameters are carried *en clair* as URL in the authz req/res.

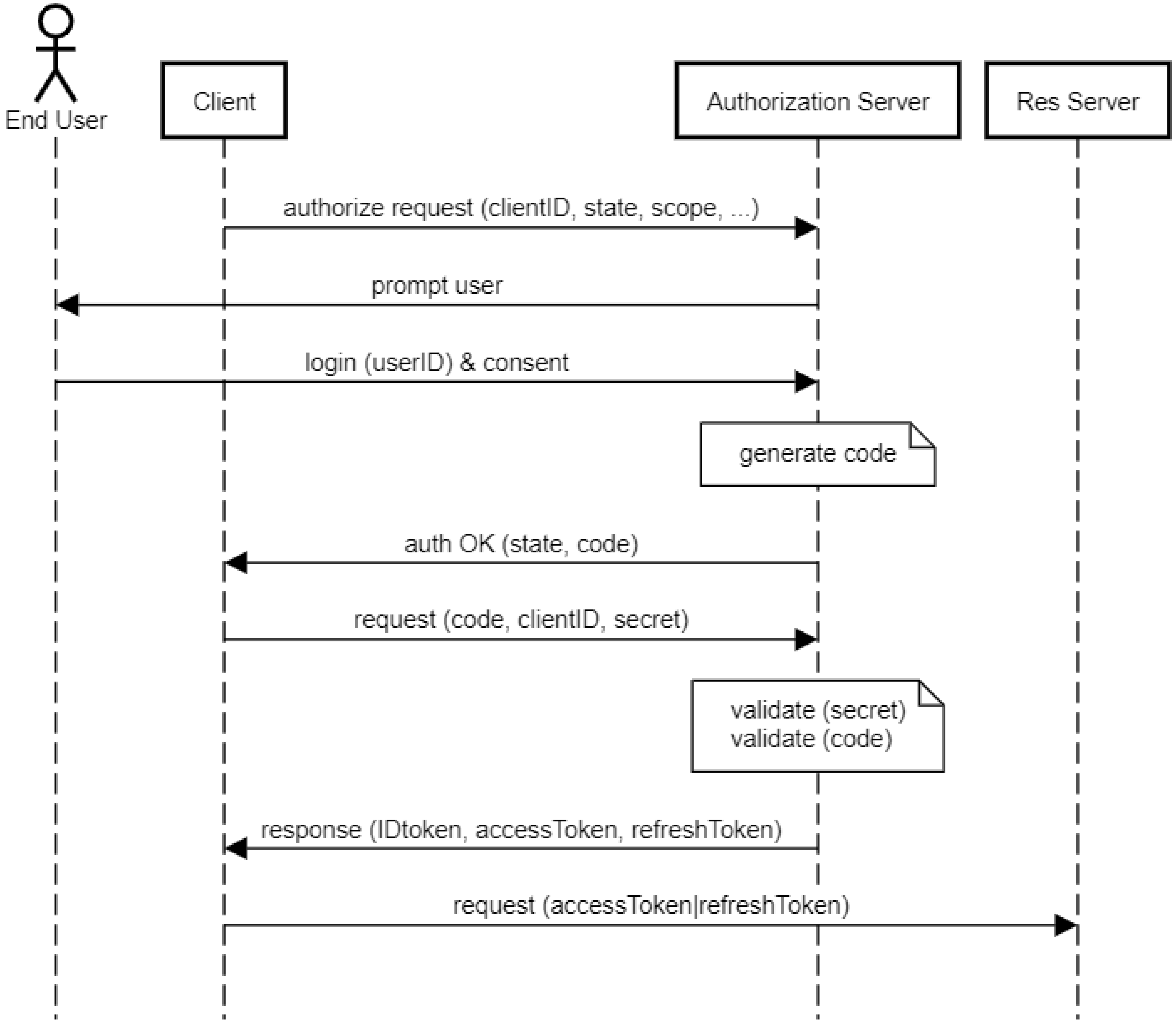


1 OpenID: Implict Grant & Authorization Code Flow 2 3 4 5

Here is a real-world implementation of OAuth2.0 as OpenID Connect 1.0 **Authorization Code flow** using Json Web Tokens.

Instead of a token, the authz res returns a code which is later exchanged for a token at the token endpoint.

Several vulnerabilities related to code/parameter injections.



1 2 Formal model using A&B syntax and results from OFMC 3 4 5

We investigated OIDC formal correctness of Authz Code Flow using Opensource Fixedpoint Model Checker [1]

Dolev-Yao style model, indeed, much less comprehensive than Fett, Küsters, and Schmitz's [2]:

- use a fixed AS
- does not model end user interface (Login&Consent not modeled)
- does not capture web specific attacks
- does not provide native support for strong Client authentication (just Client's username/pw)

Investigation under various initial conditions:

- Original Authentication Code Grant Flow
- A nonce is returned in the token
- RFC 7636 Proof Key for Code Exchange (PKCE) for OAuth2.0
- Request object signature
- Demonstrating Proof of Possession (draft-ietf-oauth-dpop-04)

1. S. Mödersheim. Algebraic properties in Alice and Bob notation. In International Conference on Availability, Reliability and Security (ARES 2009), pages 433–440, 2009.
2. 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.

Our model
Attempt #1
(Nonce is not returned in the token)

Results…
Attacker may impersonate AS and return (inject) a (previously obtained) wrong code to the Client

```
Actions:

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

[C] ->as   : RS,Scope,State,Nonce #authz req
as  ->[C]  : State,code(Scope,State,Nonce),Scope #authz res

[C]*->*as  : C,pw(as,C),code(Scope,State,Nonce)
as* ->*[C]: {resource(code(Scope,State,Nonce)),
             C,as,RS)}inv(pk(as)),#this is the access token
             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) #may be violated by
injection!
as weakly authenticates C on C,pw(as,C),code(Scope,State,Nonce) #confidential Client
```

Our model
Attempt #2 (with Nonce)

Results…
Without protecting → injection of parameters into the authz req, wrong token returned.
Protecting:
Only the authz req → code injection
Only the authz res → DoS by Nonce injection (*detectable if the Client checks Nonce in the returned token*).

```
Actions:

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

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

[C]*->*as  : C,pw(as,C),code(Scope,State,Nonce)
as*  ->*[C]: {resource(code(Scope,State,Nonce)),
             C,as,RS,Nonce)}inv(pk(as)),#returned access token now includes a nonce
             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)
```

Our model
Attempt #3 (RFC 7636 PKCE is inroduced)

Results…
Not protecting messages: attacker may alter the parameters in authz req but presents the correct PKCE Challenge to obtaing a wrong code, which is injected in the flow and later exchanged for a (wrong) token by the Client.

```
Actions:

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

[C]  ->as   : RS,Scope,State,Nonce,hash(Verifier) #PKCE Challenge
as   ->[C]  : State,code(Scope,State,Nonce,hash(Verifier)),Scope #code "embeds" PKCE

[C]*->*as  : C,pw(as,C),code(Scope,State,Nonce,hash(Verifier)),Verifier #PKCE Verifier
as*  ->*[C]: {resource(code(Scope,State,Nonce,hash(Verifier))),C,as,RS)}inv(pk(as)),
             Verifier

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

```
Goals:

RS authenticates C on RS,resource(code(Scope,State,Nonce,hash(Verifier))),C,as,
   Session
C authenticates RS on Data
Data secret between RS,C
C authenticates as on State,Scope,code(Scope,State,Nonce,hash(Verifier))
as weakly authenticates C on C,pw(as,C),code(Scope,State,Nonce,hash(Verifier)),
   Verifier
```

Our model
Attempt #4 (RFC 7636 PKCE + signing authz req)

Results…
Simply providing authenticity of the authz req – even without encrypting its content – finally results in a safe flow!

Protecting authz res only works too.

```
Actions:

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

C*   ->as   : RS,Scope,State,Nonce,hash(Verifier) #PKCE Challenge + req signature
as   ->C    : State,code(Scope,State,Nonce,hash(Verifier)),Scope #code "embeds" PKCE

[C]*->*as  : C,pw(as,C),code(Scope,State,Nonce,hash(Verifier)),Verifier #PKCE Verifier
as* ->*[C]: {resource(code(Scope,State,Nonce,hash(Verifier)),C,as,RS) }inv(pk(as)),
            Verifier

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

```
Goals:

RS authenticates C on RS,resource(code(Scope,State,Nonce),hash(Verifier)),C,as,
   Session
C authenticates RS on Data
Data secret between RS,C
C authenticates as on State,Scope,code(Scope,State,Nonce,hash(Verifier))
as weakly authenticates C on C,pw(as,C),code(Scope,State,Nonce,hash(Verifier)),
   Verifier
```

Our model
Modelling sender-constrained token.

Several proposed approaches (see draft-ietf-oauth-security-topics)

We investigated DPoP (popular draft)

Results…

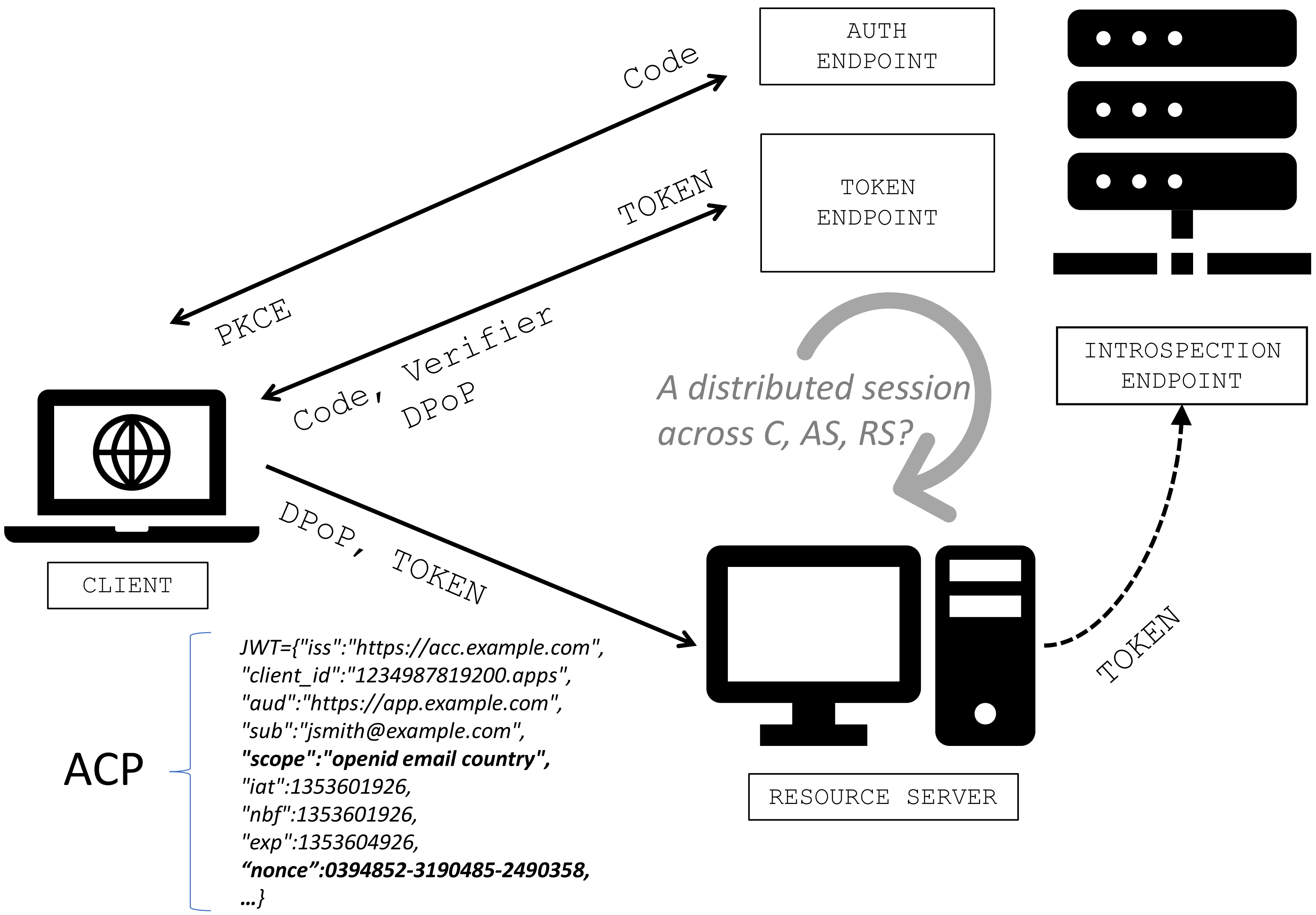
Safe if DPoP sign is over *sufficient* parameters (at least pk(C), RS, Scope) to avoid reply attacks.

```
Actions:

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

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

[C]*->*as  : C,pw(as,C),code(Scope,State,Nonce,hash(Verifier)),Verifier,
             {pk(C)}inv(pk(C)) #Self signed DPoP pk(C)
as*->[C]   : #Token returned en clair to investigate token leakage effects…
             {resource(code(Scope,State,Nonce,hash(Verifier))),
             C,as,RS,Nonce,pk(C)}inv(pk(as)),
             code(Scope,State,Nonce,hash(Verifier))
[C]*->*RS  : {resource(code(Scope,State,Nonce,hash(Verifier))),C,as,
             RS,Nonce,pk(C)}inv(pk(as)),Session,
             {ath(resource(code(Scope,State,Nonce,hash(Verifier))),C,as,
             RS,Nonce,pk(C)),RS,Scope}inv(pk(C)) #This is the DPop proof
RS* ->*[C]: Data,Session
```

```
Goals:

RS authenticates C on RS,resource(code(Scope,State,Nonce),hash(Verifier)),C,as,Session
C authenticates RS on Data
Data secret between RS,C
C authenticates as on State,Scope,code(Scope,State,Nonce,hash(Verifier))
as weakly authenticates C on C,pw(as,C),code(Scope,State,Nonce,hash(Verifier)),Verifier
```

So what?

While the Client *knows in advance* the Nonce, but has no information on which resources the user has authorized access to, *the Resource Server does not know none of these,* even if this information is *written* in the token. The whole system trust relies on *the token signature* (sometimes introspection endpoint is used).



Modern crypto may come into help…

Public Key Crypto	Identity-Based Encryption[1]	Attribute-Based Encryption[2-4]
$z=\{x\}_{pk(a)}$	$z=\{x\}_{mpk,\text{"receiver"}}$	$z=\{x\}_{mpk,(a\wedge b)\vee c}$
$x=\{z\}^{-1}_{sk(a)}$	$x=\{z\}^{-1}_{mpk,sk(\text{"receiver"})}$	$x=\{z\}^{-1}_{mpk,sk(\{a,b\})}$
Solves key-distribution problem (pk is publicly available)	*Many randomized secrets keys for one set of MPK, MSK*	*Combines IBE with SSS [2] and monotonic span trees [3,4]*
	Public keys "replaced" by plain strings	*A fine-granuled content access policy implemented in crypto!*
	A KMS distributes MPK and generates secret keys	*Many other math properties…*

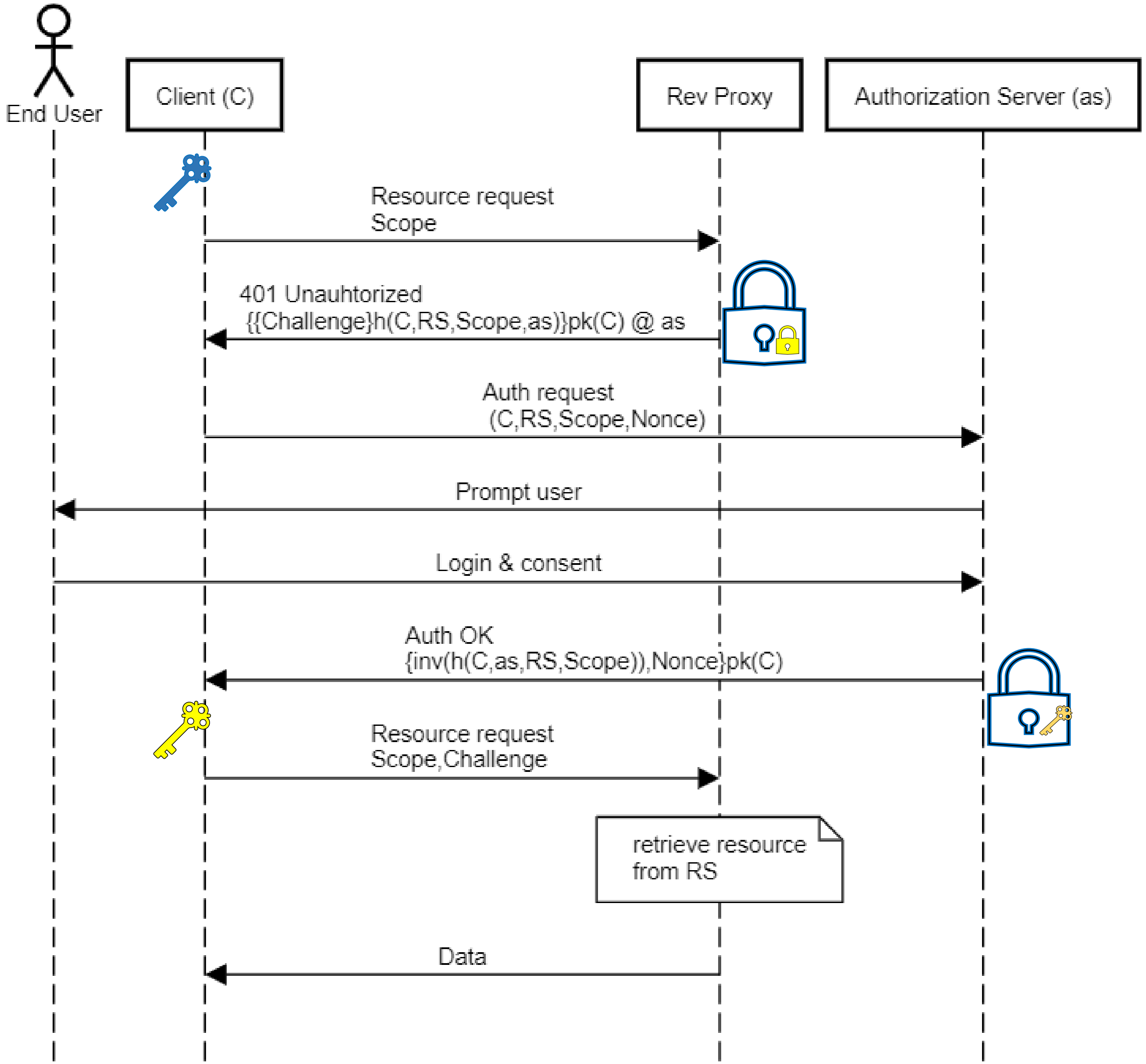
1. A. Shamir. Identity-based cryptosystems and signature schemes. In Proceedings of CRYPTO 84 on Advances in cryptology, pages 47–53. Springer-Verlag New York, Inc., 1984.
2. A. Sahai and B. Waters. Fuzzy identity-based encryption. In EUROCRYPT, pages 457-473, 2005.
3. V. Goyal, O. Pandey, A. Sahai, 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.
4. J. Bethencourt, A. Sahai, B. Waters: "Ciphertext-policy attribute-based en-cryption", Proceedings of the 2007 IEEE Symposium on Security and Privacy, SP'07, pages 32-334. Washington, DC, USA, IEEE Computer Society.

1 2 3 Present ABE and show its application to OIDC 4 5

Our **proposed flow** is a slightly modified implementation of OpenID Connect 1.0 Implicit Grant plus an HTTP challenge-response authentication

→

Problems 1, 2 and 3 wholly solved by crypto



Challenge:

$$\{\{x\}_{MPK,h}\}_{MPK,"client\_id"}$$

Where

$$h = iss \wedge client\_id \wedge aud \wedge user \wedge scope \wedge (t<exp)$$

Same Access Control Policy as in a JWT:

JWT={"iss":"https://acc.example.com",
"client_id":"1234987819200.apps",
"aud":"https://app.example.com",
"sub":"jsmith@example.com",
"scope":"openid email country",
"iat":1353601926,
"nbf":1353601926,
"exp":1353604926,
"nonce":0394852-3190485-2490358,
…}

Simpler and effective design, leveraging on e2e encryption

Straightforward to implement

Less certification costs

Access control decision enforced by math, not by code

```
Actions:

C   ->RS  : Scope
RS* ->C   : as,{{Challenge}h(C,as,RS,Scope)}pk(C) #401 Unauthorized

C   ->as  : C,RS,Scope,Nonce
as  ->C   : {inv(h(C,as,RS,Scope)),Nonce}pk(C) #JWT containing an ephemeral key and a
Nonce encrypted to C

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

Protocol proves to be formally correct with respect to the original goals

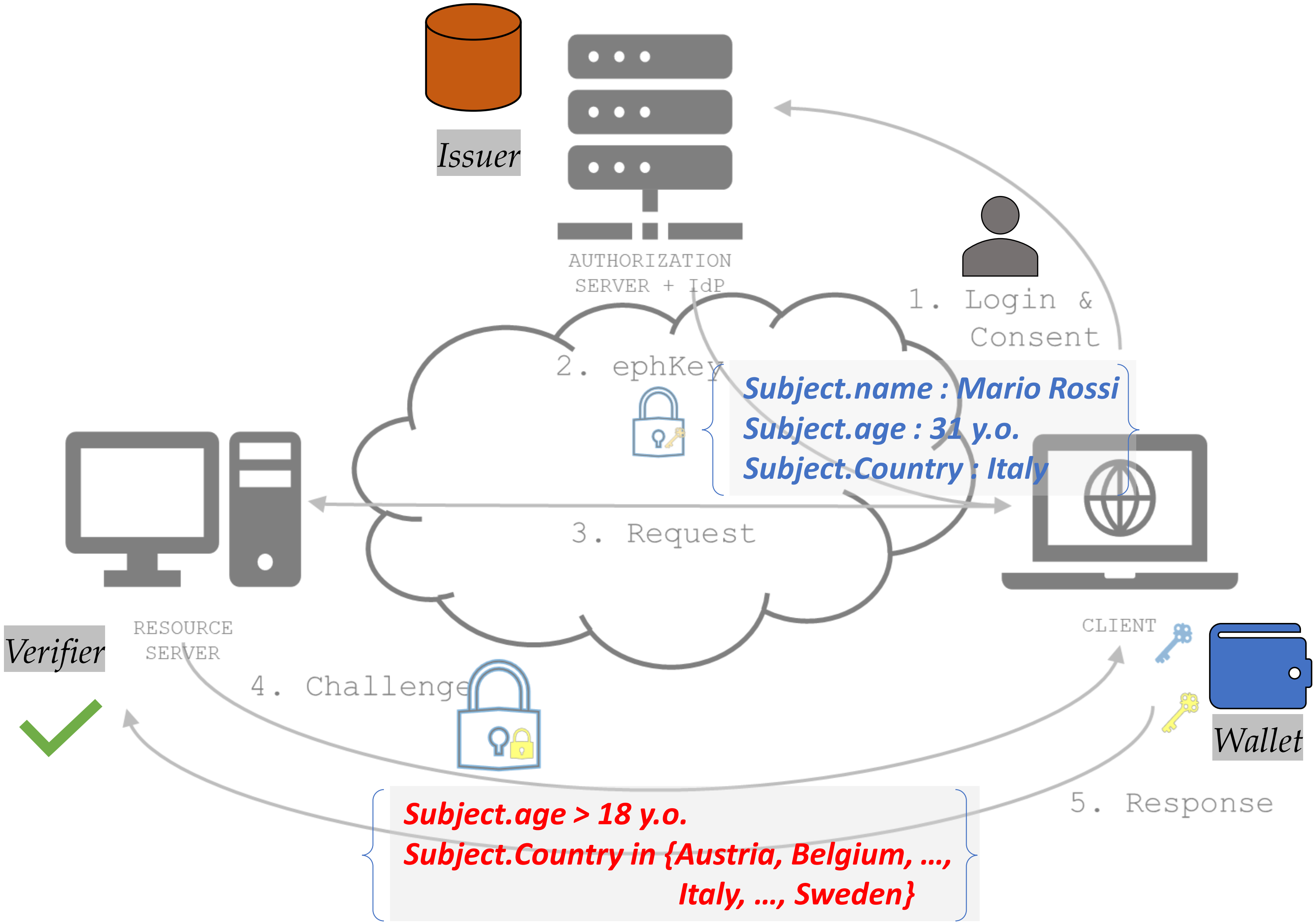
```
Goals:

C authenticates as on C,RS,Scope,Nonce #Nonce avoids reply attacks
RS authenticates C on Challenge
C authenticates RS on Data
Data secret between RS,C
```

Zero Knowledge using ABE challenge /response

A single ephKey may contain several attributes… →

BUT

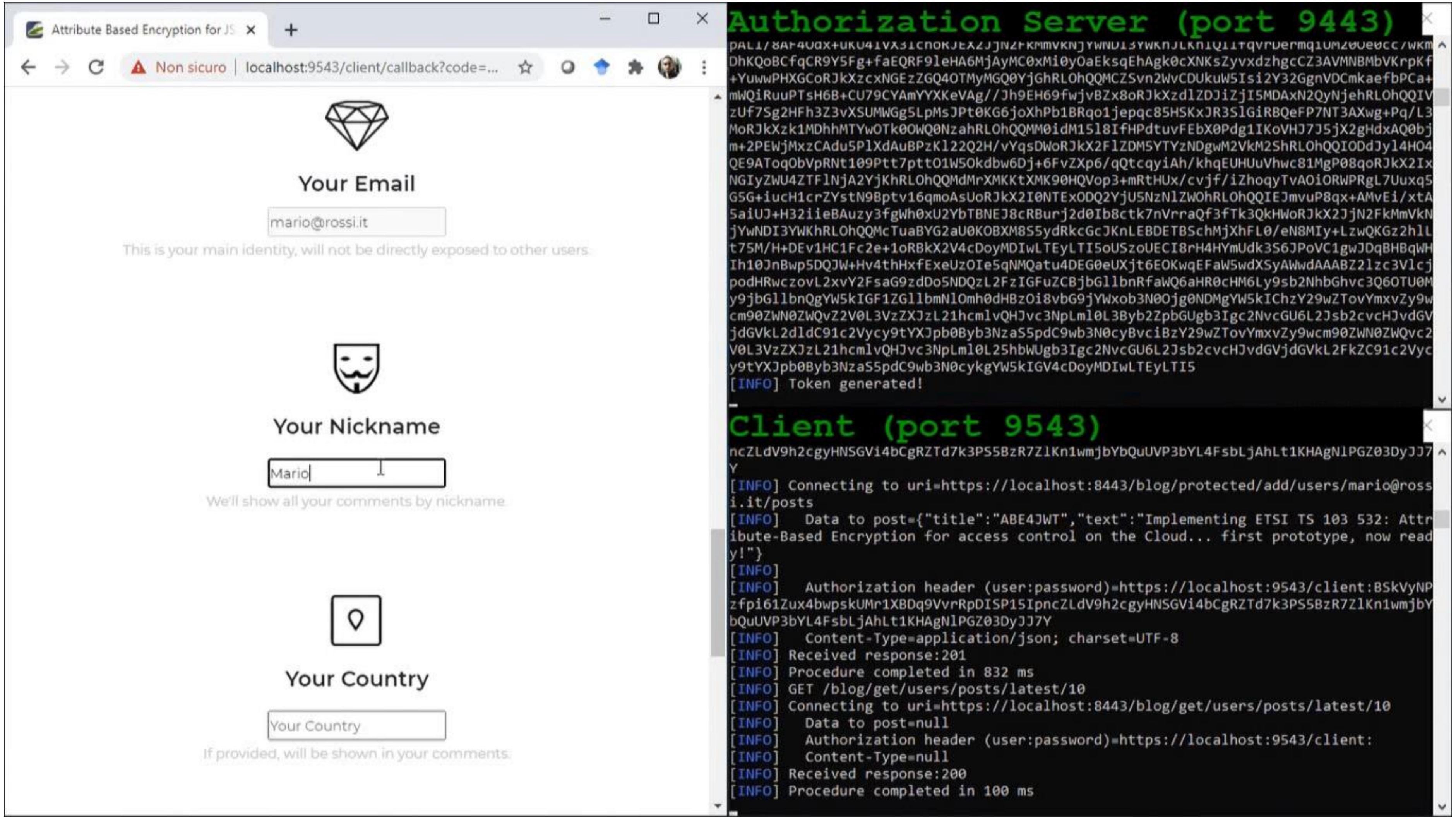
…policy can be shaped to minimize the needed knowledge →



Demo Time…

ABE4JWT.NET

Source code on https://github.com/netgroup/abe4jwt



Takeaway

1. Token signature, authorization code, Client's object request signature, PKCE and DPoP create a distributed session between Client, Authorization Server and Resource Server.
2. To achieve the same result, using a different design, we leverage on predicate encryption. ABE generates randomized encryption keys from a chosen set of attributes and ciphertext from regular expressions over them.
3. To implement an access decision, tokens based on digital signature require a coordination of signature verification and software components. Using ABE policy, the access decision is "automatically" achieved, by solving a cryptographic challenge.

Takeaway

4. Existing or ad-hoc invented additional signature schemes are being progressively introduced in OAuth2/OIDC to fit Zero Knowledge requirements (essentially to turn a two-party relationship signer/verifier into a three party one: issuer, prover, verifier).
5. ABE natively implements this three-party relationship: featured with a native policy definition language, an ABE-based challenge/response protocol may well support several ZK schemes (incl. not only selective disclosure of attributes, but also: proof of membership, range proof, complex predicate proof…)
6. Several features (from biometric authentication to revocation) may be reliably achieved without changing the schema, just by adding proper attributes to keys

More info about the crypto we used

→

…but not for this talk ☺

THANKS!

The Standards People

ETSI Security Week 2020 goes virtual!

Even More Advanced Cryptography

ETSI Standardization in Advanced Cryptography

Presented by: François Ambrosini, Umlaut
Christoph Striecks, AIT Austrian Institute of Technology

https://www.brighttalk.com/webcast/12761/409316

Some Backup Slides…

Signing request

In OIDC, this can be done using the Request Object (a signed JWT passed in the authz req).

However care must be taken to not implement naïve vulnerabilities.

openid.net/specs/openid-connect-core-1_0.html#JWTRequestValidation

TOC

6.1. Passing a Request Object by Value

The `request` Authorization Request parameter enables OpenID Connect requests to be passed in a single, self-contained parameter and to be optionally signed and/or encrypted. It represents the request as a JWT whose Claims are the request parameters specified in **Section 3.1.2**. This JWT is called a Request Object.

Support for the `request` parameter is OPTIONAL. The `request_parameter_supported` Discovery result indicates whether the OP supports this parameter. Should an OP not support this parameter and an RP uses it, the OP MUST return the `request_not_supported` error.

When the `request` parameter is used, the OpenID Connect request parameter values contained in the JWT supersede those passed using the OAuth 2.0 request syntax. However, parameters MAY also be passed using the OAuth 2.0 request syntax even when a Request Object is used; this would typically be done to enable a cached, pre-signed (and possibly pre-encrypted) Request Object value to be used containing the fixed request parameters, while parameters that can vary with each request, such as `state` and `nonce`, are passed as OAuth 2.0 parameters.

Summarizing…

Issue #1

Create a distributed session between C, AS and RS ensuring the semantics inside a JWT is commonly understood and correctly enforced

Issue #2

Provide confidentiality in token transmission

(will avoid code4token)

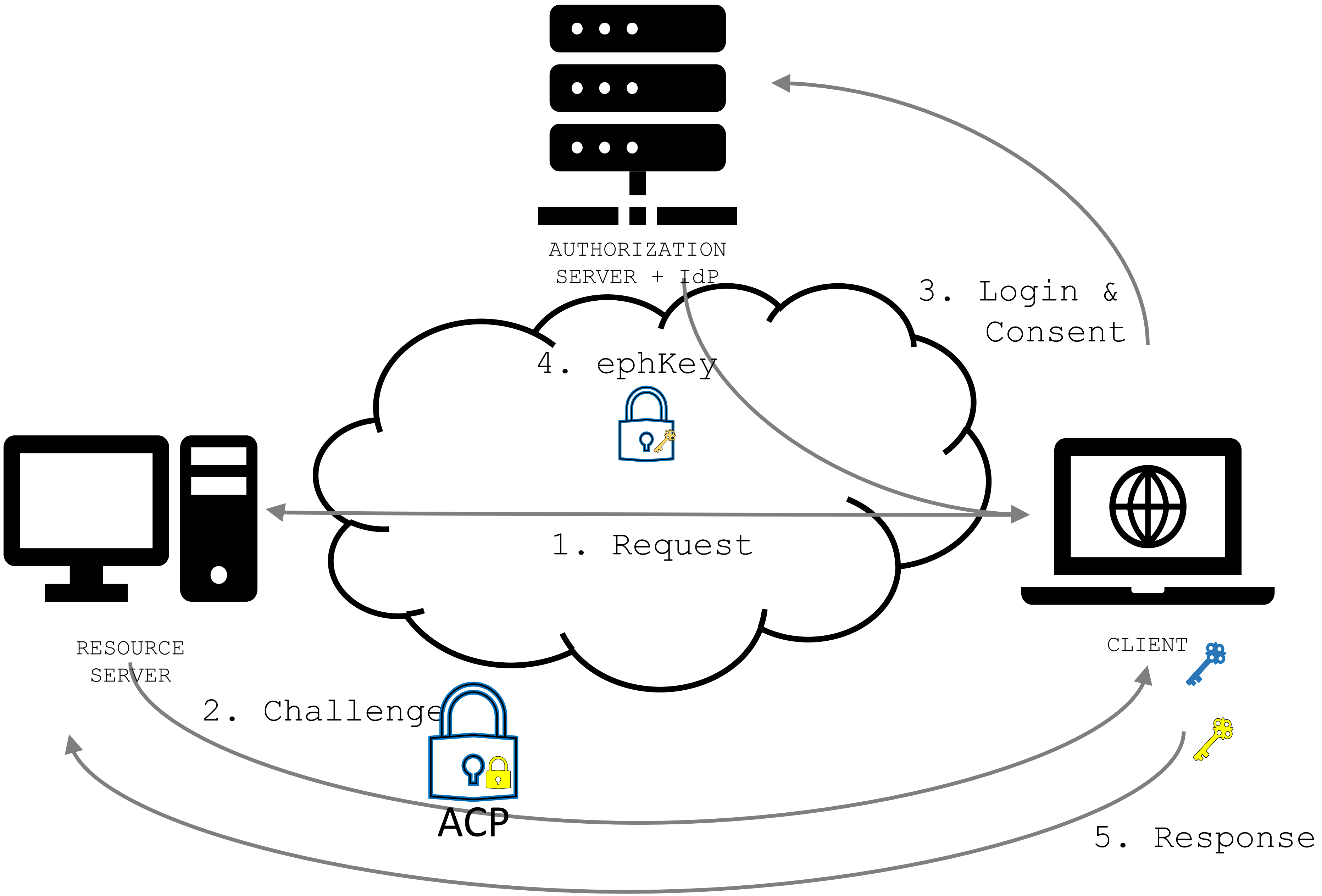
Issue #3

Guarantee Client's proof of possession

Our **proposed flow** is a slightly modified implementation of OpenID Connect 1.0 Implicit Grant + HTTP challenge-response authentication

→

Problems 1, 2 and 3 wholly solved by crypto



Zero Knowledge

A Zero Knowledge schema guarantees that no verifier learns anything other than the fact that a true statement is true.

Need: minimize disclosed information to preserve privacy

```
SK        = KeyGen(IKM, keyInfo);
PK        = SkToPk(SK);
signature = Sign(SK, PK, header, messages);
result    = Verify(PK, signature, header, messages);
proof     = ProofGen(PK, signature, header, ph, messages,
       disclosedIndexes);
result    = ProofVerify(PK, proof, messages.length, header, ph,
            disclosedMessages, disclosedIndexes);
```

An example: BBS+ signature (draft-looker-cfrg-bbs-signatures-01, July 2022; based on Boneh, Boyen and Shacham, 2004) →

Pairing-based ECC signature that signs multiple messages (i.e., claims in a token). The signature and messages can be used to create signature proofs of knowledge in zero-knowledge proofs in which the signature is not revealed, and messages can be selectively disclosed

Suggested use in Oauth2/OIDC: the access token features a BBS signature. The Client generates a unique proof from the original token and includes the proof in the request instead of the token ("non-correlating Security Token" – Appendix B.1)

The Resource Server can detect a replay attack by ensuring the proof presented is unique (Appendix B.2)

Demo explained...

<u>Once only</u>: set it up by getting MPK by the Authorization Server.

Then register your own secret key*.

The AS will post your secret key → to your chosen endpoint (`redirect_uri`).

** Slightly more complex than this in our implementation.*

```
//GET MPK

GET https://localhost:9443/as/jwk
```

AAAAIqpvyjuP8CFnxAvHGt15TwhmtDJleGFtcGxlLm9yZy5tcGsAAAGooQFZsgEEtLIBAAEBzE
ozi7JRPkRz/SO3dy/wO2bnrV0fbtzJvbzzF0jBIU66RtdPAPyPEVQiq8dXRu0t0z4Wxu+cobDH
8yQv1lEUMChNdxNhqeO9FeUhZsC+Jx47IB7Nxy/a7gHSTQI/4xV3VUzjyLAUn9OKMnGCEMBthy
eN29Rkc1dcFUPhKghwA0TR/NBIrH8f1hczg+3p8XtXJM5N+dXGcDmi/F8LhAYKGX69P2EmsIqz
UEf31BuV5s7ITu7V6fvDCzJMHLvIAxx3Wr4Jr1WQundGLoP/1F3qV3f9T0Wv2cNc/CAm82m/EY
RB9TYGczWm5GHo1m1jYisFjLmu7wX1lK2WCiLOb/6hAmcxoSSyoSECIdvwkMTNv6Sq9CqndJwH
3dG0CnnMlKLwdSHARcvB06GhAmcyoUSzoUEDADbOvvphPMbPW56lfUZNbTN0fIKTGFdmCnqii/
wQZ9wV4hgTtFAWMNa1JvfXew0Lu4tnIihXnO5MQ1XMQbnq0qEBa6ElHQAAACC4JZzeAHhPG5Qd
NVu8lFeANvDfgOeh86qUlflF74w/MA==

```
//REGISTER YOUR OWN SECRET KEY

GET
https://localhost:9443/as/sk?redirect_uri=https://localhost:9543/client/callback&state=14c5ed6e-d184-b1a2-5ad1-af978cf79bcf

//AS will respond you with Client's secret key:

POST https://localhost:9543/client/callback&state=14c5ed6e-d184-b1a2-5ad1-af978cf79bcf
```

AAAAGapvylA44LubhSEN523PsjNXSctkZWNLZXkAAADJoRJEXzcxNGEzZGQ4OTMyMGQ0YjGhJL
KhIQMkZM9FrxW0toROc52TDA8jXZ4/YDTnoy3b8jhHOSb2WaESZF83MTRhM2RkODkzMjBkNGIx
oUSzoUECBjGJdYdMhnah/N94MLaixTA058KUNxvSY8butWq+FTUTTK5SlMsfDh2evOytWHXAFq
ptsTOtXiu7jfuwf3q4q6EFaW5wdXShLB0AAAAnY2xpZW50X2lkOmh0dHBzOi8vbG9jYWxob3N0
Ojk1NDMvY2xpZW50

Demo.
Resource server offers publicly accessible resources and private resources.

Access to a private resource is denied.

But what is this?→

```
//Simple REST API
GET https://localhost:8443/blog/get/users/posts/latest/3

[{title:"My first post", text:"Love this one"},
{title:"My Second post", text:"Not so much"},
{title:"My Third post", text:"Definitively hate it"}]


//Try now access to a protected resource
GET
https://localhost:8443/blog/protected/get/users/johndoe@example.com
/profile
//You'll get a challenge:
401 UNAUTHORIZED
WWW-Authenticate: Basic realm =
zoROqb8rcmqGFCG7u481SpodMuslrsgEJo[…]RJDXzcxNGEzZGQ4O_TMyMGQ0YjGhJL
KhIQ[…]MCqHFHX9W5ehsz1d0g9WluUxvsxYk3fKxDuEXlVptuZyXlSMYTZFYlPKwdfP
wLSG8BKOhA19FRKFFHQAAAEBnKeNH1bajqqra3IZhM5HJoBJCnYg7xR1Ho92@localh
ost
```

Demo.
Login and Consent happens as in a traditional OpenIDConnect flow.

```
//Authorization Request: redirect to AS for Login & Consent
GET https://localhost:9443/as/authorize
      ?response_type=code
      &client=https://localhost:9543/client
      &redirect_uri=https://localhost:9543/client/callback
      &audience_uri=https://localhost:8443
      &scope=/blog/protected/get/users/{id}/profile
             /blog/protected/get/users/{id}/posts
             /blog/protected/set/users/{id}/name
             /blog/protected/set/users/{id}/country
             /blog/protected/add/users/{id}/posts
      &state=c42aed6e-dd84-41a2-96d1-1f9c8cf79bcf
      &nonce=0394852-3190485-2490358
```

Demo.
The returned code is an encrypted JWT.

Decrypt the returned JWT → using your own secret key

You'll discover an ephemeral key → inside your JWT. The ephemeral key **encodes attributes corresponding to JWT claims**

```
//Authorization response: AS replies with
https://localhost:9543/client/callback?code=eyJlcobDH8yQv1lEUMChNdxNhqeO9Fe
UhZsC+Jx47IB7Nxy/a7gHSTQI/4xV3VUzjyLAUn9OKMnGCEMBthyeN29Rkc1dcFUPhKghwA0TRN
BIrH8f1hczg+3p8XtXJM5N+dXGcDmi/F8LhAYKGX69P2EmsIqzUEf31BuV5s7ITu7V6fvDCzJMH
LvIAxx3Wr4Jr1WQundGLoP[…]24f8

//"code" is a JWT encrypted to your Client. Decrypt it using your Client's
key:
EncryptedJWT.parse(code).decrypt(new KPABEDecrypter(new
Base64URL(clientKey)))

//After decryption you will discover ephKey (user has authorized 2 of 4
items):
{"sub":"johndoe@example.com",
"aud":"https://localhost:8443",
"nbf":1611142470,
"ephkey":"AAAAGX69P2EmLTIF8LhAYKGX69P2Em[…]sIqzUEf31BuV5s7ITu7V6fvDCzJMHLvI
Axx3Wr4Jr1WQundGLoP/1F3qV3f9T0Wv2cNc/CAm82m/"scope"EYRB9TYGczWm5GHo1m1jYisF
jLmu7wX1lK2WCiLOw",
"scope":"/blog/protected/get/users/pippo@pippo.it/profile
        /blog/protected/get/users/pippo@pippo.it/posts",
"iss":"https://localhost:9443/as",
"exp":1611187199,
"iat":1611142470,
"client_id":"https://localhost:9543/client",
"nonce":0394852-3190485-2490358
}
```

Demo.
Decrypt the ciphertext using your secret key and your JWT ephemeral key.

Present the secret as a response to the RS challenge.

Finally get the requested resource →

```
//Finally decrypt the challenge using both clientKey & ephKey

plaintext=abeProvider.decrypt(new Base64URL(clientKey), new
Base64URL(ciphertext.parts[0]), new Base64URL(ciphertext.parts[1]))

abeProvider.decrypt(new Base64URL(ephkey), new
Base64URL(plaintext.parts[0]), new Base64URL(plaintext.parts[1]))

//You will get:
GHXMPFDR1Q5FSTMsc29QaOhYJAwLZA5KtB3Hy1QwBrTFTJIcY0NtjFmwwQTlKia7onlwz9vgSqL
NAusTceCKCTHSumR8ubGUmfTmelMuGBc2hD89q4SA1m4mn8g1gGmD

//Repeat your request to the RS

GET
https://localhost:8443/blog/protected/get/users/johndoe@example.com/profile

Authentication: <new
Base64URL("https://localhost:9543/client:GHXMPFDR1Q5FSTMsc29QaOhYJAwLZA5KtB
3Hy1QwBrTFTJIcY0NtjFmwwQTlKia7onlwz9vgSqLNAusTceCKCTHSumR8ubGUmfTmelMuGBc2h
D89q4SA1m4mn8g1gGmD")>

//Finally you'll get
200 OK
{name:"John Doe",
country:"Italy"}
```

Compliance with ARF functional reqs (ARF chapt. 4)

- 1. Perform electronic identification and store and manage qualified electronic attestation of attributes (QEAA) and electronic attestation of attributes (EAA) locally [or remote]**:** **natively satisfied**
- 2. Request and obtain attestations from providers, qualified electronic attestation of attributes (QEAA) and electronic attestation of attributes (EAA): **natively satisfied**
- 3. Provide or access cryptographic functions: **natively satisfied**
- 4. Mutual authentication between the EUDI Wallet and external entities: **natively satisfied**
- 5. Selecting, combining and sharing with relying parties PID, QEAA and EAA: **natively satisfied**
- 6. Privacy by design and selective disclosure of attributes: **natively satisfied (by the intrinsic ABE capability to fulfil an ACP without disclosing unnecessary attributes/their value)**
- 7. Provisioning of interfaces to external parties: **natively satisfied**
- 8. Authentication of (Q)EAA and PID when [and only when] those are linked to the EUDI Wallet: **natively satisfied**
- 9. Online and offline Wallet authentication with third party: **natively satisfied**
- 10. very strong crypto: **natively satisfied**
- 11. User interface supporting user awareness and explicit authorization mechanism: **natively satisfied**
- 12. Signing data by means of qualified electronic signature/seal (QES): **signature module on a different interface**