Continuous authorization over HTTP using Verifiable Credentials and OAuth 2.0

Nikos Fotiou, Evgenia Faltaka, Vasilis Kalos, Anna Kefala, Iakovos Pittaras, Vasilios A. Siris, George C. Polyzos[1]

Abstract:

We design, implement, and evaluate a solution for achieving continuous authorization of HTTP requests exploiting Verifiable Credentials (VCs) and OAuth 2.0. Specifically, we develop a VC issuer that acts as an OAuth 2.0 authorization server, a VC verifier that transparently protects HTTP-based resources, and a VC wallet implemented as a browser extension capable of injecting the necessary authentication data in HTTP requests without needing user intervention. Our approach is motivated by recent security paradigms, such as the Zero Trust architecture, that require authentication and authorization of every request and it is tailored for HTTP-based services, accessed using a web browser. Our solution leverages JSON Web Tokens and JSON Web Signatures for encoding VCs and protecting their integrity, achieving this way interoperability and security. VCs in our system are bound to a user-controlled public key or a Decentralized Identifier, and mechanisms for proving possession are provided. Finally, VCs can be easily revoked.

Keywords: Access control; Authentication; Zero Trust

1 Introduction

In the recent years, the global pandemic made remote working a necessity rather than an option. Nevertheless, this came at a cost: according to a recent research 74% of organizations attribute business-impacting cyber attacks to vulnerabilities in technology put in place during the pandemic.[2] For this reason, more and more enterprises embrace security approaches such as the *Zero Trust* paradigm. The main concept of Zero Trust is "never trust, always verify", which means, among other things, that every request should be authenticated and authorized. This architecture begs for new, secure, lightweight access control solutions, with increased interoperability and without adding privacy threats. In this paper, we propose a security solution that can be used for providing authorization for every HTTP request. Our solution leverages *Verifiable Credentials* (VC) [Ma19] and provides efficient VC management, improves interoperability, and enhances user security and privacy.

[1] Athens University of Economics and Business, Mobile Multimedia Laboratory {fotiou,eugeniafaltaka,kalos20, kefala,pittaras,vsiris,polyzos}@aueb.gr

[2] https://www.tenable.com/press-releases/seventy-four-percent-of-organizations-attribute-damaging-cyberattacks-to

cc by sa

Our solution considers users of an enterprise wishing to access an HTTP-based protected resource using a web browser. Both users and the protected resource may be located in networks outside the administrative realm of the enterprise. From a high-level perspective our solution operates as follows: Users interact with an authorization server, owned or controlled by the enterprise, using a *wallet*, implemented as a browser extension. The authorization server responds with a Verifiable Credential (VC) that contains the capabilities of the user, which is stored in the user's wallet. Then, the user interacts with a protected resource (through the web browser) and includes in the corresponding requests: (i) the received VC and (ii) a Proof of VC Possession. A VC *verifier*, acting as a transparent HTTP proxy, intercepts the communication between the web browser and the protected resource, validates the VC based on pre-configured rules and confirms the Proof of Possession. If all checks succeed the verifier forwards the HTTP request to the resource.

Our solution also enables authorization servers to provide an efficient revocation mechanism. This revocation mechanism includes a compact list of revoked VCs encoded in a self-verifiable data structure. The revocation list can be received directly from the issuer, or it can be provided indirectly; it can be even included in a resource access request.

Compared to legacy OAuth 2.0 solutions, our system provides the following advantages:

- The generated VCs are bound to a user-controlled identity, therefore they can be stored for a longer interval and they cannot be used by entities that have intercepted them (unlike, for example, mere "bearer tokens"). Hence, client applications do not have to interact often with an authorization server.
- Our system uses VCs as "access tokens". VCs support richer semantics, can be used for evaluating complex access control policies, and facilitate interoperability.
- Our system provides an efficient mechanism for checking the revocation status of an access token/VC.

Compared to related VC-based solutions, our design provides the following advantages:

- Our system builds on the widely used and well supported OAuth 2.0 flows for managing the lifecycle of a VC. These flows are implemented in a browser based wallet achieving continuous and secure authorization over HTTP without user intervention.
- Our system leverages JSON Web Tokens (JWT) [JBS15b] and JSON Web Signatures (JWS) [JBS15a] for representing and protecting VCs. These are widely used and standardized solutions (as opposed to, for example, linked-data proofs).
- Our system supports user authentication using both public public keys, as well as "Decentralized Identifiers" (DIDs) [W321]. DIDs allow users to rotate their secret keys without having to receive a new VC.

The remainder of this paper is organized as follows. In Section 2 we detail the design of our solution. In Section 3 we present the implementation and evaluation of our solution. We discuss related work in Section 4 and we conclude our paper in Section 5.

2 Design

In this section we detail the components of our system and their interactions (also illustrated in Fig. 1).

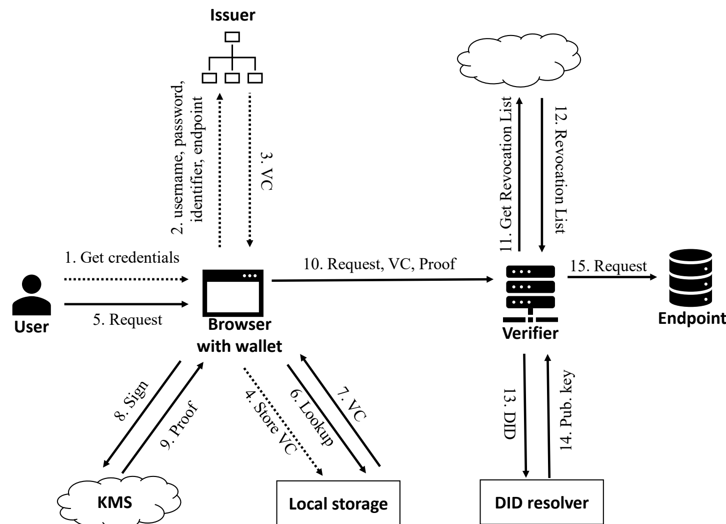

Fig. 1: An instance of our system

2.1 Components

Our system is composed of a VC *issuer*, a VC *verifier*, and a *wallet*. The goal of our system is to allow authorized users to invoke *operations* on *resources* stored in an particular *endpoint*, for example, perform a “list” operation, on a “folder”, stored in a Cloud storage system. An endpoint is identified by a URL, denoted by $URL_{Endpoint}$, and it can be oblivious to our system. Our system supports the use of *Decentralized Identifiers* (DID) as a mean for identifying various entities. DIDs are URIs which resolve to a DID document that contains information related to the DID, such as ways to cryptographically authenticate the DID owner. The structure of a DID document and the DID resolution mechanism are specific to each DID *method*. We provide more details about how DIDs are implemented in our system n section 3.1

The VC issuer is an OAuth 2.0 authorization server extended with VC issuing capabilities. Each VC issuer is identified by an ID_{issuer}, which can be a URL or a DID. Furthermore,

each issuer owns a public-private key pair and we assume a secure method for resolving an ID_{issuer} to the corresponding public key (e.g., through static configuration, using the web PKI, or by performing a DID resolution). Issued VCs are encoded as JWTs, and signed using a JSON Web Signature (JWS) and the private key of the issuer. VCs in our system "describe" the capabilities of a VC *subject* over a protected endpoint. Additionally, a VC issuer maintains a VC revocation list.

The VC verifier is an HTTP proxy that intercepts HTTP requests towards a protected endpoint. The VC verifier is able to verify the validity, the status, and the ownership of VCs included in the intercepted requests. Additionally, the VC verifier acts as a *policy enforcement point* by validating whether or not a VC can be used for executing the requested operation over a resource.

The wallet is a web browser extension that interacts with the VC issuer and verifier using standard OAuth 2.0 flows. The wallet is responsible for storing the received VCs, and for including them in the corresponding HTTP requests. Additionally, the wallet generates and manages user owned identifiers, which can be public keys or DIDs. Such an identifier is included in a VC and the associated secret key is used by the wallet for generating a VC *proof of possession*. Users may have multiple identifiers, as well as multiple wallets.

2.2 Interactions

Our system involves a configuration step, after which the system components can interact with each other using OAuth 2.0 flows.

2.2.1 System configuration

This is a step usually executed during a set-up phase. With this step an issuer is configured with policies that specify the capabilities that correspond to a user. Additionally, users *register* with the issuer (at least) a wallet. With this registration process, users create a username and password for their wallet and assigns to it a subset of their capabilities. The generated username and password will be later used by the wallet in order to retrieve VCs from the issuer. Finally, verifiers are configured with a list of trusted ID_{issuer} identifiers and (if required) with their corresponding public keys.

2.2.2 VC request and issuance

With this step, a user's wallet requests from the issuer a VC that can be used for accessing a particular endpoint. A VC request is in essence an OAuth 2.0 access token request using the client credentials grant (section 4.4 of [He12]); in our system the corresponding "client

credentials grant" is the wallet's username and the password registered to the issuer during the configuration phase. The wallet includes in this request an identifier (which can be either a public key, or a DID). The wallet may re-use an existing identifier or it may generate a new one, specific to that particular VC request. The issuer verifies the provided username and password, and retrieves the capabilities associated with them. Then it creates a VC that includes these capabilities and the provided identifier, encodes it as a JWT, and signs it. An example of a VC as used in our system follows. As it can be seen, the standard *iss* and *aud* JWT claims are used for denoting the issuer and the target endpoint of the VC. If the provided identifier is a DID it is included in a *sub* claim; if it is a public key the *cnf* claim as defined in RFC7800 is used. The VC may include additional JWT claims that control its validity period. Finally, the *vc* claim includes information that can be used for determine the VC's revocation status, as well as a list of "resources" and allowed "operations".

```
1  {
2      "iss": ID_issuer,
3      "aud": URL_Endpoint,
4      "sub": User owned DID,
5      "vc":{
6        "@context": [...],
7        "id": "credential 1",
8        "credentialStatus": {...},
9        "credentialSubject": {
10         "type": ["CapabilitiesCredential"],
11         "Resource1": [ "Operation 1", "Operation 2" ]
12       }
13     }
14   }
```

2.2.3 VC revocation

Our revocation mechanism is based on the system described in [SDS20]; a similar approach for VC revocation is followed by a recent W3C draft [Gr20]. In order to support revocation, an issuer maintains a revocation list that covers all *not expired* VCs it has issued. This list is a simple bitstring and each VC is associated with a position in the list. In particular, each revocable VC includes a property named *revocationListIndex* that specifies the position of the VC in the revocation list. Revoking a VC means setting the bit corresponding to the VC to 1. Since the list includes only non-expired VCs, its size is tolerable for most use cases. For example, an issuer that issues on average 100 VCs per day with lifetime equal to one month, would only need 30×100 bits to store its revocation list. Any entity can verify the status of a non-expired VC that supports this revocation mechanism, by examining the value of the bit of the corresponding revocation list.

A revocation list is included in a JWT, signed and timestamped by the issuer. This JWT can be retrieved directly by the issuer, or indirectly, e.g., the issuer can store it in an online location, or even in a blockchain. Each revocable VC includes a property named *statusListCredential* which is a "pointer" (e.g., a URL) to the revocation list location. A verifier can retrieve the revocation list by itself, or require from users to include it in their requests. In all cases, the verifier has to validate the signature of the JWT and determine its "freshness".

2.2.4 Endpoint access

A user can request from an endpoint to perform an operation over a resource. This request is transmitted over HTTP using the user's web browser. If the $URL_{Endpoint}$ is included in the *aud* claim of a stored VC, the wallet retrieves this VC from its local storage and prepares a *proof of possession*. This proof is generated according to "Demonstration of Proof-of-Possession at the Application Layer" (DPoP) [D.20] OAuth 2.0 extension. DPoP has been designed for HTTP communication and achieves PoP in a single message. In particular, with DPoP the wallet creates JWS signed using the key the corresponds to the user identifier included in the VC. The DPoP payload includes at least a unique, sufficiently large random number, the HTTP *method* of the request, the HTTP *URI* of the request, and the *time* when the proof was created. Then, the wallet includes the VC in the *Authorization* HTTP header of the request and the generated proof in a *DPoP* HTTP header. The request is received by the verifier that acts as an HTTP proxy. The verifier initially validates the included VC. In particular, it examines if the VC is signed by a trusted issuer and if the value of *aud* claim equals to the the $URL_{Endpoint}$. Additionally, if the VC includes claims that control its validity period, it examines if the VC is valid. Then, it extracts the user identifier included in the VC. If that identifier is a DID, the verifier performs a DID document resolution and retrieves the corresponding public key. Then, it verifies the signature of the provided DPoP using the public key associated with the user identifier, and it examines if the DPoP is "sufficiently fresh", if it includes the correct HTTP method and URI, as well as if the included random number has not been "recently" used. If the VC is revocable, it examines the status of the VC by retrieving the revocation list from the VC issuer. Finally, it verifies if provided VC includes the capabilities that are necessary for invoking the requested operation. If all checks succeed, the verifier forwards the request to the endpoint.

3 Implementation and evaluation

We have implemented[3] our issuer as a .net core web application, and our verifier using Python3 and the jwcrypto library[4]. Moreover, we integrated DIDs in our system using DIF's

[3] Pointers to GitHub repositories of our implementations can be found in `https://mm.aueb.gr/projects/zerotrustvc`

[4] `https://jwcrypto.readthedocs.io`

Universal Resolver [DI21], and we have implemented our wallet as a Firefox extension. We provide more details about the use of the Universal Resolver and about our browser-based wallet in the following subsections. Then we present our evaluation scenario and we discuss the performance and the security properties of our solution.

3.1 Support for DIDs

Our system supports DIDs, which is common practice in most VC systems. The DID standard allows each DID method to define its own way for resolving a DID to the corresponding document. To avoid any further complexity and to contribute to the interoperability of our project, we rely on DIF's Universal Resolver for DID document resolution. The Universal Resolver performs DID resolution across many different DID methods by providing a universal API. Internally, this is accomplished through an architecture consisting of drivers for each supported method.

Our current implementation supports the did:web method [Mi21]. This is a DID method that bootstraps trust by leveraging an existing web domain's reputation. A did:web DID is constructed based on a URL which when resolved results in the corresponding DID document. Our verifier implementation uses a local instance of the Universal Resolver to resolve did:web DIDs. It interacts with it through an API that receives as input a did:web DID and responds with the corresponding public key.

3.2 Browser-based wallet

User's wallet is implemented as a browser extension. The extension is tasked with requesting the credentials from the issuer and presenting them to the verifier. Internally, each credential is stored to the browser's supplied storage and is indexed based on the URL included in the *aud* claim. This not only allows for fast lookups, but also for syncing those credentials between browsers in different devices. Furthermore, users will also have the ability to back up their credentials to some other source (i.e., the cloud, the file system etc.).

Users provide to the extension an ID_{issuer} and the appropriate username and password. Then, the extension either creates a new cryptographic key pair, or re-uses an existing did:web DID and communicates with the issuer. If successful, the extension will read the credential from the issuer's response, parse it and update its internal state. This is the only process that involves user intervention. To present a credential, the extension in the background listens for any HTTP request made to a URL for which there is a saved credential that includes that URL in the *aud* claim. When such a request is made, the extension will retrieve that credential and create a fresh DPoP value using the appropriate cryptographic key material. Then, the extension injects the DPoP and the VC as new HTTP headers, in the original HTTP request.

The extension is implemented for the Firefox browser. To manage the user's cryptographic key material we have adopted 2 different strategies; in-browser key management and using external key management systems (KMS). In the former case, the extension saves private keys encrypted in the browser's storage, while public keys are saved in clear. To encrypt a private key we use AES with a key derived from a user supplied password using PBKDF2. In the latter case, we rely on a Cloud-based KMS (but any external KMS can be trivially supported). In that case, private keys are stored in the Cloud and when the wallet needs to sign something (i.e., to create a DPoP), the cryptographic hash of the raw data is sent to the KMS. This method, although it adds an additional round trip, it alleviates the need for the extension to manage private keys itself.

3.3 Cloud Storage scenario

A Cloud Storage access scenario is implemented in order to evaluate the proposed architecture. In this use case, the protected resource is a Google Cloud Storage Bucket. Buckets are the basic containers in Google Cloud Storage and are used in order to organize data and control access to them.

According to our use case an employee of the enterprise wishes to access some data from a bucket via a web interface. The user must first request a VC through the browser extension acting as her VC wallet. The issued VC specifies certain capabilities such as *read*, *write*, *upload*, or *list*. The web interface is provided by a Python3 script, which implements the Google Storage API: this script is the protected endpoint. This script interacts with the Cloud storage using a "service" account, hence both the script, as well as the Cloud storage provider are oblivious to the used VCs, as well as to the user management system and the access control policies of the enterprise.

3.4 Overhead

We have measured the VC issuing processes, DPoP generation, and access request verification in a desktop PC equipped with an Intel i5 5540 CPU and 8GB RAM, running Windows 10 using EdDSA and ES256 signature algorithms for JWS. All operations require less than $0.1ms$.

Since VCs and DPoPs are transmitted in HTTP headers they are encoded using base64. The base64 encoding of a VC that includes two resources and two operations is 656 bytes. Similarly, the base64 encoding of a DPoP is 440 bytes.

A revocation list is stored in a signed JWT. This JWT also includes the *iss* claim, which defines the issuer, and the *iat* claim, which defines the date and time at which the token was issued. Such a signed token, which is encoded using base64 encoding, generated using

ES256 JWS algorithm, including the verification key in the JWS header and a revocation list with 4000 entries, is 1431 bytes long.

3.5 Security properties

Our solution leverages OAuth 2.0, whose security properties have been formally verified [FKS16], for managing the lifecycle of VCs and it provides proof of possession, preventing this way many security attacks. Additionally, our system achieves the following security properties:

Increased availability. Non-revocable VCs can be verified without needing the issuer to be online. When it comes to revocable VCs various optimizations can be considered for decreasing the dependence on the availability of the issuer, e.g., cache revocation lists for some time, store revocation lists in alternative locations. Similarly, verifiers do not have to maintain any user specific state since all the information required to make an access control decision is included in each request; verifiers are only required to maintain for a limited time the nonce included in a DPoP in order to prevent replay attacks.

Efficient access control management. User and access control policy management is implemented independently of the protected endpoint, since granting or revoking an access right does not involve any communication with the endpoint (or the verifier). Furthermore, by implementing the VC verifier as an HTTP proxy, we allow transparent protection of any HTTP-based resource.

Attack surface reduction. In our solution the amount of verifications a verifier needs to perform is less compared to a system that relies on Access Control Lists (ACLs), which are inflexible, do not scale well, and are difficult to use and upgrade [Ka06]. In our system, a verifier has only to verify the validity and the possession of the VC included in an access request. Furthermore, verifiers are not required to store any additional secret information to implement our protocols, neither do they have to maintain user accounts. Moreover, a user is allowed to use multiple wallets and assign to each wallet different capabilities. For example, a wallet used in a "travel laptop" may have less capabilities compare to a wallet used in a secured, well-administrated PC. Finally, our wallet selects the appropriate VC by itself, by matching the requested URL with the URL included in the "aud" claim of each VC: this approach is less prone to security attacks compared to most of the existing approaches that require user intervention, e.g., they require from users to scan a QRcode.

Resilience to attacks. Our system is resilient to many types of attacks. Since the VCs are bound to a user owned identifier, our system is not affected by attackers-in-the-middle that intercept the communication towards a protected endpoint. These attackers, can neither modify the transmitted VCs without being detected, nor re-use the captured VCs to their own purposes. Similarly, our system allows different user identifiers per VC, hence, even if the private key that corresponds to an identifier is breached, the captured VCs can only

be used for accessing a specific endpoint. Moreover, by including a DID such as did:web as a user identifier in a VC, a user can rotate the private key that corresponds to that DID without having to receive a new VC. Therefore, users can even proactively rotate their keys.

4 Related work

The problem of designing efficient authorization and access control solutions for Zero Trust Architectures (ZTA) is well known and there are many efforts that try to address it [Ya20, LHK20]. Lukaseder et al. [LHK20] discuss how the Zero Trust Model can be applied to open networks, such as in a network of a university. Furthermore, they implement and present a Zero Trust network framework, called Alekto that authenticates and authorizes users in order to take access control decisions and compute trust scores for an eLearning system. Yao et al. [Ya20] propose a dynamic and fine-grained access control and authorization solution for ZTA, which is composed of an access control agent, a user identity authentication module, an access control engine, and a trust evaluation engine. The main differences between our work and these works are that our solution uses VCs as access tokens, which include the client capabilities. The VCs can have longer lifetimes and they can be stored in secure wallets as opposed to mere bearer tokens.

Similarly to this work, Lagutin et al. [La19] try to integrate VCs and DIDs into the OAuth 2.0 protocol. In their solution, which is designed for constrained devices, they use VCs and DIDs as authentication grants. Clients use these grants to obtain access tokens from the authorization server. Our solution follows a reverse approach: clients use a username and password as a grant to obtain VCs. This has the advantage that authorization is enforced when requesting access to a resource. The solution in [FSP21] also combines VCs with OAuth 2.0 in order to provide capabilities-based access control. Our approach, improves the solution presented in [FSP21] by allowing users to use multiple identifiers and even use different identifier per VC, by adding support for Decentralized Identifiers, and by considering a wallet. In our system users can have multiple wallets and each wallet can be assigned different access rights: this property has many security advantages.

Our system leverages VCs for expressing capabilities because VCs are well understood techniques being standardized. Additionally, supporting a specific VC type is straightforward, hence interoperability can be supported with low effort. Related approaches that can be used instead of VCs in a system similar to ours are Macaroons [Bi14], and Authorization Capabilities for Linked Data (ZCAP-LD) [C.20]. Similarly, our system builds on OAuth 2.0, which is a widely used standard, hence our solution can be easily integrated in existing systems. Additionally, OAuth 2.0 has been designed specifically for HTTP services. Other related works propose new protocols such as the Credential Handler API [Ls21], and the Presentation Exchange protocol [BZR22].

5 Conclusion

In this paper we proposed a security solution that allows continuous authorization over HTTP. Our solution uses Verifiable Credentials (VCs) to store user capabilities. Additionally, it leverages OAuth 2.0 and a browser-based wallet to provide fast VC lifecycle management, without requiring any user intervention. Our solution implements proofs of possession preventing this way VC sharing. Additionally, our solution supports Decentralized Identifiers, allowing users to rotate their private keys without having to receive a new VC.

Our solution provides additional advantages, which are not highlighted by the use case considered in our paper. For example, our solution allows multiple issuers, it supports re-using the same VC for accessing different endpoints of the same type, and the considered revocation mechanism does not reveal to an issuer information about the user that tries to access an endpoint. Similarly, our solution allows wallets to include in user request a "fresh" copy of the revocation list, enabling this way offline verifiers, which can be of particular importance in cases such as IoT systems. Future work in this area involves the replacement of DPoP with Webauthn assertions, the application of our solution in the IoT by adding support for more efficient VC encodings (e.g., using CBOR), as well as for IoT specific protocols (e.g., CoAP), and the integration of VC verifier into the endpoints themselves.

Acknowledgments

The work reported in this paper has been funded in part by European Union's Horizon 2020 research and innovation programme through subgrant *Enabling Zero Trust Architectures using OAuth2.0 and Verifiable Credentials (ZeroTrustVC)* of project *eSSIF-Lab*, under grant agreement No 871932 and by the *Research Center of the Athens University of Economics and Business*.

Bibliography

[Bi14] Birgisson, Arnar; Politz, Joe Gibbs; Úlfar Erlingsson; Taly, Ankur; Vrable, Michael; Lentczner, Mark: Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In: Network and Distributed System Security Symposium. 2014.

[BZR22] Buchder, D.; Zundel, B.; Reidel, M.: Presentation Exchange v2.0.0. Working Group Draft, DIF, 2022. `https://identity.foundation/presentation-exchange/`.

[C.20] C. L. Webber, M. Sporny Eds: Authorization Capabilities for Linked Data. Draft Community Group Report, W3C, 2020. `https://w3c-ccg.github.io/zcap-ld/`.

[D.20] D. Fett et al.: OAuth 2.0 Demonstration of Proof-of-Possession at the Application Layer (DPoP). RFC draft, 2020.

[DI21] DIF Identifiers and Discovery Working Group: , DID Universal Resolver, 2021. `https://github.com/decentralized-identity/universal-resolver`.

[FKS16] Fett, Daniel; Küsters, Ralf; Schmitz, Guido: A Comprehensive Formal Security Analysis of OAuth 2.0. CCS '16, Association for Computing Machinery, New York, NY, USA, p. 1204–1215, 2016.

[FSP21] Fotiou, Nikos; Siris, Vasilios A.; Polyzos, George C.: Capability-based access control for multi-tenant systems using OAuth 2.0 and Verifiable Credentials. In: 2021 International Conference on Computer Communications and Networks (ICCCN). pp. 1–9, 2021.

[Gr20] Group, W3C Credentials Community: Revocation List 2020. Draft community group report, W3C, 2020. https://w3c-ccg.github.io/vc-status-rl-2020/.

[He12] Hardt (ed.), D: The OAuth 2.0 Authorization Framework. RFC 6749, IETF, 2012.

[JBS15a] Jones, M.; Bradley, J.; Sakimura, N.: JSON Web Signature (JWS). RFC 7515, IETF, May 2015.

[JBS15b] Jones, M.; Bradley, J.; Sakimura, N.: JSON Web Token (JWT). RFC 7519, IETF, 2015.

[Ka06] Karp, Alan H.: Authorization-Based Access Control for the Services Oriented Architecture. In: Fourth International Conference on Creating, Connecting and Collaborating through Computing (C5'06). pp. 160–167, 2006.

[La19] Lagutin, Dmitrij; Kortesniemi, Yki; Fotiou, Nikos; Siris, Vasilios A.: Enabling Decentralised Identifiers and Verifiable Credentials for Constrained IoT Devices using OAuth-based Delegation. Proceedings of DISS 2019 Workshop on Decentralized IoT Systems and Security, p. 6, 2019.

[LHK20] Lukaseder, Thomas; Halter, Maya; Kargl, Frank: Context-based Access Control and Trust Scores in Zero Trust Campus Networks. In: SICHERHEIT 2020. Gesellschaft für Informatik e.V., Bonn, pp. 53–66, 2020.

[Ls21] Longley, D.; sporny, M.: Credential Handler API 1.0. Draft Community Group Report, W3C, 2021. https://w3c-ccg.github.io/credential-handler-api/.

[Ma19] Manu Sporny et al.: Verifiable Credentials Data Model 1.0. W3C Recommendation, W3C, 2019. https://www.w3.org/TR/verifiable-claims-data-model/.

[Mi21] Michael Prorock et al.: did:web Method Specification. Editor's draft, W3C, December 2021. https://w3c-ccg.github.io/did-method-web/.

[SDS20] Smith, Trevor; Dickinson, Luke; Seamons, Kent: Let's Revoke: Scalable Global Certificate Revocation. In: Network and Distributed System Security Symposium. 2020.

[W321] W3C Credentials Community Group: Decentralized Identifiers (DIDs) v1.0. W3C Proposed Recommendation, W3C, 2021. https://www.w3.org/TR/did-core/.

[Ya20] Yao, Qigui; Wang, Qi; Zhang, Xiaojian; Fei, Jiaxuan: Dynamic Access Control and Authorization System Based on Zero-Trust Architecture. In: 2020 International Conference on Control, Robotics and Intelligent System. Association for Computing Machinery, New York, NY, USA, pp. 123–127, 2020.