QWACs in Automated Environments

Leonard Auer, Stefan Genchev*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: auerle@in.tum.de, genchev@net.in.tum.de

***Abstract*—The European Union's eIDAS 2.0 regulation introduces Qualified Website Authentication Certificates (QWACs) to enhance web authentication through verified organizational identity. To address criticism regarding regulatory influence over the global root trust stores, the ETSI recently proposed a novel approach, 2-QWAC, which binds a QWAC to a standard TLS certificate via a digital signature. This binding process, however, introduces operational complexity, especially in environments where TLS certificates are frequently rotated using the ACME protocol. In this work, we survey common ACME implementations and analyze the feasibility of automating 2-QWAC binding renewal. We present a proof-of-concept automation that extends the ACME flow in a popular and open-source web server.**

***Index Terms*—qualified web authentication certificates, 2-qwac, eidas 2.0, acme, pki, automation**

1. Introduction

The European Union's eIDAS 2.0 regulation introduces Qualified Website Authentication Certificates (QWACs) to add verifiable organizational identities to web authentication [1]. Unlike the Extended Validation certificates, which have been removed from the user interfaces of major web browsers [2], QWACs are issued by Qualified Trust Service Providers (QTSPs) in the EU [3].

Initial criticism from browser vendors and academia focused on the regulatory requirement for browsers to accept TLS certificates from those QTSPs. This would have required including all EU-qualified Certificate Authorities (CAs) into global root stores. Browser vendors and academia raised two main concerns: First, this requirement would circumvent the existing restrictions and processes of global root stores, possibly introducing security risks. Second, government intervention in the web trust model would set a dangerous geopolitical precedent. [4], [5]

In response, the European Telecommunications Standards Institute (ETSI) revised the specification and introduced a novel approach called 2-QWAC. It does not provide backwards compatibility, but co-exists along the original approach now called 1-QWAC. This new framework decouples identity assertion from transport layer security by binding the QWAC to a separate, conventionally issued Transport Layer Security (TLS) certificate. Importantly, the QWAC is not used directly for the TLS handshake but is validated separately via a signed binding. While the specification still mandates visual indicators in web browsers when a website uses a valid QWAC, 2-QWAC provides a compromise between introducing website identity verification under eIDAS 2.0 and recognizing the autonomy of browser vendors since it allows for a separate QWAC root CA store instead of government-controlled additions to a browser's TLS root CA store. [3]

Major browser vendors have started efforts to implement QWACs [6], however, there do not yet exist solutions to integrate them with the TLS tool chains in modern web environments. With increasing TLS automation – encouraged by organizations like the National Institute of Standards and Technology [7] – via the Automatic Certificate Management Environment (ACME) protocol [8], the added complexity of the 2-QWAC binding presents a new challenge.

This paper investigates how such bindings can be automated and synchronized with TLS certificate issuance. We make the following main contributions:

- We provide an overview of popular ACME v2 implementations.
- We analyze integration opportunities for automating the QWAC bindings in a popular ACME v2 implementation.
- We implement a proof-of-concept binding renewal mechanism for a popular open-source web server.

2. Background

For analyzing opportunities for automating the 2-QWAC binding renewal, an understanding of the two main concepts involved is necessary. This chapter provides an overview of those – the 2-QWAC framework and the ACME protocol.

2.1. 2-QWACs

QWACs are X.509 certificates issued by Qualified Trust Service Providers. The certificates contain validated organizational identity attributes about a website's operator. [3] QTSPs are EU-qualified and -supervised CAs that conform to regularly audited legal requirements. Examples include the implementation of the NIS2 directive [9] and obligations to the identity verification process set by the eIDAS legal framework. [1]

The current ETSI specification [3] defines two co-existing approaches:

- *1-QWAC*: The QWAC is used as TLS certificate, combining the identity assertion and TLS handshake. In this approach, QWACs replace standard

 doi: 10.2313/NET-2025-11-3_10

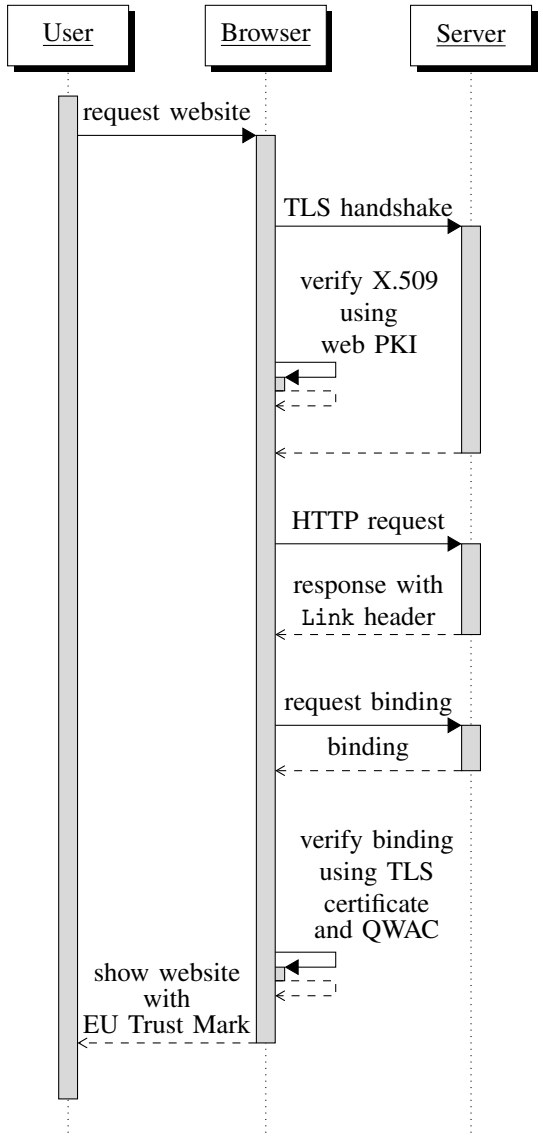

Figure 1: Sequence Diagram for a web request with valid 2-QWAC

TLS certificates. If the certificate received as part of the TLS handshake contains indicators that it is a QWAC, this approach is used.

- *2-QWAC*: The QWAC is digitally bound to a standard TLS certificate. Browsers validate the identity information independently of the TLS handshake.

The 2-QWAC binding is implemented through a JAdES (JSON Advanced Electronic Signature) structure signed using the QWAC private key. The structure references both the QWAC and the TLS certificate fingerprints. Website operators obtain QWACs and create bindings to their TLS certificates. [3]

Figure 1 shows the flow of a typical, successful 2-QWAC binding verification in web browsers. They first validate the TLS certificate using their root trust store as part of the usual TLS handshake. During a TLS session, the web server's response includes an HTTP `Link` header pointing to the binding. Web browsers then fetch the binding and validate it using the separate EU trust store. If validation succeeds, the browser is supposed to show the EU Trust Mark and the identity data of the QWAC. [3]

2.2. ACME

The Automatic Certificate Management Environment protocol standardizes automated certificate issuance. Manually obtaining a TLS certificate usually involves generating a Certificate Signing Request (CSR), sending it to a CA, proving domain ownership, and deploying the issued X.509 certificate on a web server. ACME allows web servers to automatically request TLS certificates, prove domain control, and deploy the certificates with minimal website operator input. [8]

The ACME specification defines the following flow for the automated process: [8]

1) *Account registration*: An ACME client – the web server – requests an account with an ACME server – the CA. The client sends optional contact information or other optional data along with a signature.
2) The client submits a *certificate order*.
3) *Proving domain ownership*: The ACME server sends the client a list of challenges to prove domain control. The client solves those challenges, for example, by creating a domain record containing a unique value (`dns-01`) or returning a unique value in an HTTP response (`http-01`). The ACME server validates the challenge results.
4) The client submits a *Certificate Signing Request* (CSR) to the ACME server.
5) *Deployment*: After the server issues the certificate, the client downloads and installs it.

3. ACME Client Survey

Automating the 2-QWAC binding requires awareness of when TLS certificates are issued or renewed. Thus, we aim to extend the ACME flow with logic to create the binding between the QWAC and the TLS certificate. Additionally, we need to extend the web server, for example, to serve the binding. Thus, an extensible ACME client with high integration with a web server is ideal for our use case.

In this chapter, we highlight three conceptually different, popular ACME v2 implementations. We evaluate them based on two criteria:

- *Integration Level*: This describes how tightly coupled the ACME client is with the HTTP server. Clients that have a high integration level are embedded within the HTTP server runtime and can directly influence the certificate lifecycle and modify web requests. Standalone tools that operate outside the web server have a low integration level. For our purpose, a high integration is ideal.
- *Extensibility*: This refers to how easily the ACME client can be extended. A low extensibility means that the client provides no or a limited plugin API, while a highly extensible client has a modular architecture. Our approach benefits from high extensibility.

Certbot [10] is a widely used command-line ACME client maintained by the Electronic Frontier Foundation. It

 doi: 10.2313/NET-2025-11-3_10

is written in Python and can be used both as a standalone application as well as in combination with an existing web server like *Apache* or *nginx*. Other, not officially supported web servers can integrate Certbot through plugins. [11]

Certbot stores obtained certificates at predefined paths (`/etc/letsencrypt/live/$domain`) and relies on external configuration in the web server to deploy them [11]. Thus, it is loosely coupled with the actual web server. Working with predefined locations should not pose an issue when creating the binding: we know the location of the TLS certificates and can define a path where website operators should place QWACs. However, we would need to additionally extend the web server to serve the binding and modify the HTTP response headers.

cert-manager [12] is a Kubernetes certificate controller written in Go. It automates the TLS certificate issuance and management in a Kubernetes cluster. As part of a modular issuer framework, it also incorporates an ACME client. cert-manager consists of three Kubernetes pods – a controller, a webhook, and a CA injector –, making extension for binding logic more complex. Its integration is specific to Kubernetes and less applicable to general-purpose HTTP server deployments.

Caddy [13] is a modular and extensible HTTP server written in Go. It has built-in modules with different implementations to obtain TLS certificates, one of them being an ACME v2 client. Since the ACME module is implemented directly in the web server, it has a very high integration. This means we should be able to effectively modify the HTTP header and add endpoints. Caddy's modularity should aid in extending the ACME client to control the certificate obtaining process.

4. Automated QWAC Binding Renewal

From the options we discussed, Caddy provides the most promising foundation for automating the QWAC binding renewal process due to its high extensibility and high integration of the ACME module with the web server. Thus, in the following, we analyze how QWAC binding renewal can be achieved in Caddy.

4.1. Architectural Overview of Caddy

Caddy is a statically linked Go binary, but follows a modular and extensible architecture by offering a plugin system. Architecturally, Caddy can be divided into three parts: [14]

- The *command* is Caddy's command line interface.
- Caddy's *core* reads the configuration file and is responsible for module orchestration.
- A set of *modules*, both built-in and by external developers. They contain most of Caddy's functionality and implement server features.

Modules define their own configuration options. They follow a well-defined lifecycle: The load phase loads the module into memory. The provision phase contains the module setup code. The use phase executes the module logic and the cleanup phase unloads the module. [14]

4.2. Implementation Approaches

In the following, we describe three strategies to hook into certificate issuance in order to implement automated binding renewal in Caddy.

Storage module. By implementing the interface `certmagic.Storage`, it is possible to detect file changes, including certificate file changes. Compared to a naive file polling approach, this method should be able to detect certificate changes with more precision. However, while precise in detecting changes, this method lacks context: we cannot immediately distinguish TLS certificate from other certificates, nor determine whether the event corresponds to issuance or renewal.

Issuer wrapping. `certmagic.Issuer` is an interface implemented by certificate issuers. By wrapping the existing `ACMEIssuer` and registering it as new `Issuer`, we can intercept calls to the `Issue()` function. This provides full access to the CSR and the issued certificate, enabling precise and controlled execution of binding logic. However, our second approach is not as general as our first approach since it is specific to the issuer we wrap. Additionally, this modification of the internal issuer pipeline risks compatibility issues with future versions of Caddy.

Event-based hook. Our third approach uses Caddy's built-in event module. Caddy emits a `cert_obtained` event when a TLS certificate is successfully obtained. We can register a listener for this event and trigger the binding logic. Since the event only contains the certificate path, we need to parse the raw certificate bytes, which adds additional complexity. However, we can reuse existing libraries for this task. This approach is non-intrusive and generic across issuers, while offering us the control over the issuance process we require.

4.3. Proof-of-Concept Implementation

To show the feasibility of automated binding renewal for 2-QWAC, we implement a proof-of-concept binding renewal for Caddy. We choose the event-based approach for its precision and robust compatibility. It comprises two Caddy modules.

Middleware module. We implement a middleware that modifies the `ServeHTTP()` function. It adds the `Link` header to HTTP responses pointing to a configurable endpoint. Additionally, it serves the corresponding configurable endpoint which returns a static dummy JWT.
Event listener module. The event handler subscribes to the `cert_obtained` event. Upon a newly created or issued TLS certificate, it resolves the certificate paths and parses the raw certificate bytes using Go's `crypto/x509` package. Then, it computes a SHA-256 thumbprint and prints it as proof that the certificate renewal has been detected. The event handler then generates an HMAC-SHA-256 signature using a static secret as a placeholder for the QWAC private key to simulate a digital binding operation.

4.4. Evaluation

We evaluate the implementation using Pebble, a lightweight and local ACME test server [15]. We configure our test setup with 15 seconds certificate lifetime in Pebble, ten seconds renewal interval in Caddy and disable

the challenge validation in Pebble for rapid testing. Then, using those configurations, we start a local Pebble server and a local Caddy server including our modules.

Using curl [16], we send a GET request to the Caddy server. We inspect the HTTP response and validate the presence and correctness of the `Link` header. We then verify that requests to the `Link` header URL return the static dummy JWT. Thus, we conclude that the middleware module is working correctly. Next, we observe the certificate renewals. Each certificate renewal triggers an event within the Caddy server once Caddy's polling has detected it. The event listener module creates a log entry including the certificate thumbprint and the HMAC-SHA-256 signature. In conclusion, the results confirm that the automation functions correctly under continuous rotation.

5. Conclusion and Future Work

The European Union's eIDAS 2.0 regulation has initially received criticism from both browser and academia, partly due to the introduction of QWACs [4], [5]. A recent change to the QWAC specification introduces 2-QWAC, an approach which does not require the addition of EU-mandated certificate authorities to global root trust stores [3]. 2-QWAC offers a viable compromise between verified organizational identity and browser trust autonomy.

However, the 2-QWAC approach introduces more complexity; by separating identity verification and TLS handshake, a binding between QWAC and TLS certificate is necessary. Manual management of bindings would not be feasible for websites with frequent certificate rotation.

It is our belief that the adoption of QWACs depends on the ability to integrate with modern, automated TLS workflows. Our work demonstrates that automation of 2-QWAC binding generation and renewal is both feasible and practical. By leveraging Caddy's modular architecture and integrated ACME client, we implemented a proof-of-concept that extends the ACME flow and is able to dynamically serve bindings. We aim to make our implementation fully ETSI-compliant by implementing JAdES binding generation.

For the adoption of QWACs to succeed, however, implementations for other popular ACME clients like Certbot are necessary. We identified multiple approaches to automate the 2-QWAC binding, which future work could generalize for other ACME clients. Furthermore, we discussed characteristics of other popular ACME clients hindering their extension with QWAC binding logic; future work should find ways to address these challenges.

A comprehensive ACME client survey is left for future work. While there exist lists of ACME implementations [17], [18], a comprehensive overview and feature comparison of ACME clients, evaluating support for hooks, plugin APIs, and web server integration depth, has – to our knowledge – not yet been published.

QWACs themselves offer opportunities for further research, as well. Since the current specification is still very recent, it has not yet been implemented in major web browsers. How the parallel specification of two QWAC variants, 1-QWAC and 2-QWAC, will affect adoption and whether website operators will prefer one of the approaches is an interesting question that requires further observation.

As QWACs continue to evolve, automation will be one key factor to their adoption. We believe our results provide a foundational step toward fully integrated, secure, and scalable QWAC usage in real-world environments.

References

[1] European Union, "Regulation (EU) No 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing Directive 1999/93/EC," Oct. 2024.

[2] Chromium Developers, "EV UI Moving to Page Info," Available: https://chromium.googlesource.com/chromium/src/+/HEAD/docs/security/ev-to-page-info.md, [Accessed: Jun. 22, 2025].

[3] ETSI, "Etsi ts 119 411-5, v2.1.1," Available: https://www.etsi.org/deliver/etsi_ts/119400_119499/11941105/02.01.01_60/ts_11941105v020101p.pdf, Feb. 2025, [Accessed: May 1, 2025].

[4] Mozilla, "November 2021 position paper on the European Commission's legislative proposal to revise the eIDAS Regulation," Available: https://blog.mozilla.org/netpolicy/files/2021/11/eIDAS-Position-paper-Mozilla-.pdf, Nov. 2021, [Accessed: Jun. 10, 2025].

[5] "Global website security ecosystem at risk from EU Digital Identity framework's new website authentication provisions," Available: https://www.eff.org/files/2022/03/02/eidas_cybersecurity_community_open_letter_1_1.pdf, Mar. 2022, [Accessed: Jun. 10, 2025].

[6] Chromium Developers, "Implement support for QWACs," Available: https://issuetracker.google.com/issues/392931065, 2025, [Accessed: Jun. 19, 2025].

[7] M. Akram, W. Barker, R. Clatterbuck, D. Dodson, B. Everhart, J. Gilbert, W. Haag, B. Johnson, A. Kapasouris, D. Lam, B. Pleasant, M. Raguso, M. Souppaya, S. Symington, P. Turner, and C. Wilson, "Securing Web Transactions: TLS Server Certificate Management," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 1800-16, Jun. 2020.

[8] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten, "Automatic Certificate Management Environment (ACME)," Internet Engineering Task Force, Request for Comments RFC 8555, Mar. 2019.

[9] European Union, "Directive (EU) 2022/2555 of the European Parliament and of the Council of 14 December 2022 on measures for a high common level of cybersecurity across the Union, amending Regulation (EU) No 910/2014 and Directive (EU) 2018/1972, and repealing Directive (EU) 2016/1148 (NIS 2 Directive)," Dec. 2022.

[10] Certbot Developers, "Certbot," Available: https://certbot.eff.org/, [Accessed: May 26, 2025].

[11] ——, "User Guide," Available: https://eff-certbot.readthedocs.io/en/stable/using.html, 2018, [Accessed: Jun. 15, 2025].

[12] cert-manager Developers, "Cert-manager," Available: https://cert-manager.io/, 2025, [Accessed: May 26, 2025].

[13] ZeroSSL, "Caddy," Available: https://caddyserver.com/, 2025, [Accessed: May 26, 2025].

[14] ——, "Architecture," Available: https://caddyserver.com/docs/architecture, 2025, [Accessed: May 21, 2025].

[15] Pebble Developers, "Letsencrypt/pebble," Available: https://github.com/letsencrypt/pebble, Jun. 2025, [Accessed: Jun. 9, 2025].

[16] "Curl," Available: https://curl.se/, [Accessed: Jun. 9, 2025].

[17] Developers of Certify The Web, "ACME Clients," Available: https://acmeclients.com/, [Accessed: May 26, 2025].

[18] Let's Encrypt (ISRG), "ACME Client Implementations," Available: https://letsencrypt.org/docs/client-options/, Feb. 2025, [Accessed: May 21, 2025].