Automated Certificate Management Environment Working Group
Internet-Draft
Intended status: Standards Track
Expires: 20 October 2026

F. Geng
P. Wu
L. Xia
Huawei Technologies
X. Chen
TrustAsia

18 April 2026

Automated Certificate Management Environment (ACME) Extension for Public Key Challenges

draft-geng-acme-public-key-06

Abstract

The current ACME protocol [RFC8555] requires applicants to submit a PKCS#10 Certificate Signing Request (CSR) during the finalization phase. The construction, ASN.1 encoding, and transmission of the CSR impose additional implementation burdens on both the client (especially resource-constrained devices) and the server. Moreover, the CSR cannot prevent a public key from being replaced by an intermediary at the protocol level.

This document introduces the "pk-01" challenge extension based on the ACME protocol. Its core mechanism is as follows: the applicant declares the public key to be authenticated during the "newOrder" phase and completes the Proof of Possession (PoP) by signing with the private key during the challenge phase. Since the public key is declared when the order is created and verified during the challenge phase, there is no need to submit a CSR during the finalization phase; the ACME server can issue the certificate directly based on the verified public key, thereby eliminating the CSR at the protocol level.

The "pk-01" challenge supports two verification modes via the pop_mode field:

1. *Asynchronous Mode (Async)*: The applicant pre-deploys a signature proof to a designated location (such as a DNS TXT record, HTTP path, or email reply). The ACME server then performs the verification query asynchronously, thereby completing dual verification of both control over the identifier and proof of private key ownership.

2. *Synchronous Mode (Sync)*: The ACME server issues a random number (nonce) in the challenge object and performs dual verification—confirming control of the identifier and proof of private key ownership—through a real-time Lightweight TLS handshake with the applicant, who must remain online during the verification process.

The challenge object declares the available delivery methods via the supported_delivery field, and the client selects one of them to choose an authentication method and resource deployment.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

1.1. Background and Motivation

In the current ACME process, applicants must separately generate and submit a PKCS#10 CSR during the "finalize" phase, which presents the following issues:

* *Implementation Burden*: CSR requires clients to implement ASN.1 encoding, DER format construction, and PKCS#10 encapsulation. For standard Domain Validation (DV) certificates and resource-constrained devices, this introduces unnecessary complexity.

* *Semantic Redundancy*: In the ACME certificate scenario, the certificate's subject information can be provided through existing methods; the CSR serves solely as a carrier for transmitting the public key, and other fields (such as the Subject DN) are typically ignored by the CA.

* *Post-Quantum Migration*: The signature sizes of post-quantum signature algorithms (such as ML-DSA and SLH-DSA) are significantly larger than those of traditional algorithms, and ASN.1 toolchains do not yet fully support the identifiers for these algorithms; as a result, CSR-based processes face higher compatibility risks in post-quantum scenarios. This extension provides a more streamlined migration path for the introduction of post-quantum cryptography by moving the public key declaration to newOrder, transmitting it in the original SPKI format, and using the declared key directly to complete the PoP signature.

The solution proposed in this document is to move the public key declaration to the "newOrder" phase and perform ownership verification during the challenge phase using methods such as private key signatures. As a result, the "finalize" phase does not require a CSR; instead, the ACME server issues the certificate directly using the verified public key and the identifiers in the order.

This extension minimizes changes to the existing ACME infrastructure, adding only three top-level fields — public_key, pop_mode, and csr_less — to the "newOrder" request and introducing a new challenge type, "pk-01". No modifications are required to the behavior during the finalization phase.

2. Conventions and Definitions

The key terms used in this document are defined as follows:

* *End Entity (EE, Applicant)*: The entity that initiates the ACME certificate request and holds the private key to be verified.

* *ACME Server (AS)*: A Certificate Authority (CA) or its delegated service that implements the ACME protocol.

* *Claimed Public Key*: The public key to be authenticated, declared by the applicant in the public_key field of the "newOrder" request, in Base64URL-encoded Subject Public Key Information (SPKI) format [RFC5480]. The claimed public key *MUST* correspond to the private key actually held by the applicant; the ACME server *MUST* perform signature verification using the claimed public key and *MUST* perform a byte-by-byte comparison with the public key in the final certificate before issuing it.

* *Proof of Possession (PoP)*: A mechanism whereby an applicant demonstrates, through cryptographic signature operations, that they actually possess the private key corresponding to the declared public key.

* *Extended Key Authorization Value (keyAuthorization)*: An extended version of the challenge-response value defined in this document. It builds upon the standard ACME keyAuthorization (token + "." + base64url (JWK_Thumbprint (accountKey))). It incorporates the normalized string of the order’s full identifier (identifiers_canonical) to bind the order scope, and is covered by a signature using the declared private key, serving to simultaneously verify both resource control and private key ownership (see Section 4 for details).

* *public_key*: A new top-level extension field added to the "newOrder" request, containing the declared public key (SPKI encoded in Base64URL).

* *pop_mode*: A top-level field added to the newOrder request in this document, used by the client to specify the selected PoP validation mode (async or sync, or other extensible mode strings).

* *csr_less*: A new top-level boolean field added to the "newOrder" request, the default value is false. Controls whether the CSR submission is omitted during the finalization phase: "true" indicates that the certificate is issued directly using the declared public key; "false" indicates that a standard PKCS#10 CSR must still be submitted during the finalize phase, and the "pk-01" challenge is executed as an additional public key pre-check step.

* *usage context*: A fixed prefix consisting of 64 bytes of 0x20 (ASCII space), followed by the fixed ASCII string "ACME-pk-01", and a null byte (\x00), shall serve as the domain-separation prefix for all "pk-01" signature messages. It is denoted as (SP × 64) || "ACME-pk-01\x00", where SP represents a single 0x20 byte.

* *identifiers_canonical*: The canonical string representation of all identifiers in an order, used to bind the full order scope in the signature message. Construction method: sort all entries in the order' s identifiers array in lexicographical order by (type, value) in ascending order, format each entry as "type:value", and concatenate the entries with "," without whitespace. Example: For an order containing [{dns, www.example.com}, {dns, example.com}], identifiers_canonical = "dns:example.com,dns:www.example.com"; For a single-identifier order [{dns, example.com}], identifiers_canonical = "dns:example.com".

* *supported_delivery*: A list (array of strings) of available delivery methods declared by the AS in the challenge object. In asynchronous mode, this may include "dns", "http", and "email"; in synchronous mode, it may include "tls-handshake".

* *delivery*: A field included by the client in the body of the challenge-response POST request, specifying the selected delivery method.

* *acme-pk/1*: A TLS Application Layer Protocol Negotiation [RFC7301] identifier defined in this document, intended exclusively for TLS-HANDSHAKE delivery in the "pk-01" challenge-synchronization mode. When the EE receives a connection on port 443 that carries this identifier, it performs the TLS handshake using its *existing TLS certificate for the domain* (without constructing a new certificate), and returns the raw proof bytes directly as TLS application data upon completion of the handshake. During the handshake, the AS *MUST* skip certificate chain trust validation, and *SHOULD* verify that the server certificate public key matches newOrder.public_key on a byte-by-byte basis. "acme-pk/1" and "acme-tls/1", as defined in [RFC8737], are two distinct application-layer protocols (they have different formats and cannot be used interchangeably): "acme-tls/1" requires the

construction of a self-signed certificate containing OID extensions, whereas "acme-pk/1" directly transmits the raw signature bytes.

The key words "*MUST*", "*MUST NOT*", "*REQUIRED*", "*SHALL*", "*SHALL NOT*", "*SHOULD*", "*SHOULD NOT*", "*RECOMMENDED*", "*NOT RECOMMENDED*", "*MAY*", and "*OPTIONAL*" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Protocol Extension: newOrder Fields

This document introduces minimal extensions to the "newOrder" request: the identifiers field retains its standard ACME semantics and carries the certificate subject information; a new top-level field, public_key, is added to carry the declared public key; a new top-level field, pop_mode, is added for the client to declare the selected validation mode; and a new top-level boolean field, csr_less, is added to control whether the CSR submission is omitted during the finalization phase.

After the ACME server receives a "newOrder" request containing the public_key field, it will issue a "pk-01" challenge for each identifier. When csr_less is set to "true", the server issues the certificate directly during the finalize phase based on the verified declared public key and the identifiers; when csr_less is set to "false" (default), the applicant must still submit a CSR during the finalize phase, and the "pk-01" challenge is executed as an additional public key pre-check step.

If the public_key field does not exist, the ACME server *SHOULD* process the request according to the standard ACME procedure; this extension does not apply.

The pk-01 challenge *MUST NOT* support the authorization reuse mechanism defined in [RFC8555]. Every "newOrder" request containing the public_key field *MUST* trigger a full new challenge flow. The server *MUST NOT* reuse any existing authorization, regardless of its validity status or whether the public key matches. This restriction eliminates binding ambiguity between authorization state and order public keys, and prevents proxy clients from constructing proofs for different orders using existing authorizations. By the same reasoning, this extension *MUST NOT* support the pre-authorization mechanism.

3.1. Description of New Fields

Field name	Type	Existence	Description
public_key	String	OPTIONAL	Claimed public key, SPKI [RFC5480] encoded in Base64URL. If present, triggers the "pk-01" challenge and the CSR-less issuance process.
pop_mode	String	OPTIONAL	PoP verification mode declared by the client: "async" (the applicant pre-deploys the proof, and the AS verifies it independently) or "sync" (requires online interaction with the applicant). This can be extended to other third-party modes.
csr_less	Boolean	OPTIONAL	Whether to skip the CSR submission during the finalization phase. "true": Issue directly using the declared public key; "false" (default): A CSR must still be submitted, with "pk-01" serving as a pre-check step.

Table 1

The rules for the pop_mode field are as follows:

* "async" (Asynchronous mode): The applicant pre-deploys the PoP signature proof to a specified resource location, and the AS independently queries and verifies it at any time; the applicant does not need to be online during verification. Supported delivery methods are declared by the supported_delivery field of the challenge target and can be "dns", "http", or "email".

* "sync" (Synchronous mode): AS performs authentication by interacting directly with the applicant's server via a real-time protocol; the applicant *MUST* remain online during authentication. Supported delivery methods are specified by the supported_delivery declaration; a typical implementation is a simplified "tls-handshake" handshake.

* If pop_mode is omitted, CA *SHOULD* use the "async" by default.

* If the CA does not support the scheme declared by the client, it *SHOULD* return an error in the response.

3.2. newOrder Request Example

3.2.1. Asynchronous Mode— Multi-identifier DNS Order

```
{
  "identifiers": [
    { "type": "dns", "value": "example.com" }
    { "type": "dns", "value": "www.example.com" }
  ],
  "public_key": "<Claimed Public Key, Base64URL-encoded SPKI>",
  "pop_mode": "async",
  "csr_less": true
}
```

3.2.2. Synchronization Mode — DNS Identifier

```
{
  "identifiers": [
    { "type": "dns", "value": "example.com" }
  ],
  "public_key": "<Claimed Public Key, Base64URL-encoded SPKI>",
  "pop_mode": "sync",
  "csr_less": true
}
```

3.2.3. Asynchronous Mode — Email Certificate Scenario

```
{
  "identifiers": [
    { "type": "rfc822name", "value": "user@example.com" }
  ],
  "public_key": "<Claimed Public Key, Base64URL-encoded SPKI>",
  "pop_mode": "async",
  "csr_less": true
}
```

4. Extended KeyAuthorization

The "pk-01" extends the standard ACME certificate format by introducing domain identifier binding and private key signature verification, thereby validating both control over the resource and ownership of the private key.

4.1. Base KeyAuthorization Value

The base keyAuthorization value is consistent with [RFC8555]:

```
keyAuthorization = token || "." || base64url(JWK_Thumbprint(accountKey))
```

4.2. Signature Construction in Asynchronous Mode

In asynchronous mode, the signature message (to_sign) is constructed by appending the usage context prefix and the canonical string of all order identifiers to the keyAuthorization. The usage context prefix (see definition in §2 usage context) provides domain separation for the signature message, while binding the full set of order identifiers prevents proxies from using single-domain proofs to request multi-domain certificates.

```
to_sign_async = (SP × 64) || "ACME-pk-01\x00" || keyAuthorization || "." || identifiers_canonical
```

where SP × 64 denotes 64 consecutive ASCII space bytes (0x20), and the construction of identifiers_canonical is defined in § 2.

Proof of Possession Value:

```
proof = base64url(Sign(claimedPrivateKey, to_sign_async))
```

For the semantics of Sign(key, message), see §4.4.

Multi-Identifier Order Notes: For an order containing multiple identifiers, the server issues a separate "pk-01" challenge for each identifier, with a corresponding token and validation URL for each challenge. The client constructs a keyAuthorization (using the token of each challenge) and a proof individually for each challenge. However, the signature message for all challenges *MUST* use the same identifiers_canonical (covering all identifiers in the order). This ensures that each single-domain proof is cryptographically bound to the full scope of the order, so that an individual proof cannot be reused to request an order with a different set of identifiers.

Deployment locations for proofs of each identifier type:

Identifier Types	Delivery type	Deployment Location	Search Method
dns	dns	DNS TXT record _acme-challenge.<domain>	DNS Query
dns	http	HTTP path /.well-known/acme-challenge/<token>	HTTP Query
rfc822name	email	Body of the S/MIME email reply	Email Receipt

Table 2

4.3. Signature Construction in Synchronous Mode

In synchronous mode, the token in a signed message is replaced with a new random number (nonce) issued by the CA to ensure the timeliness of the signature, and the message is similarly appended with a usage context prefix and the canonical string of all order identifiers.

```
keyAuthorization_sync = nonce || "." || base64url(JWK_Thumbprint(accountKey))
to_sign_sync = (SP × 64) || "ACME-pk-01\x00" || keyAuthorization_sync || "." || identifi
ers_canonical
```

Proof of Possession Value:

```
proof = base64url(Sign(claimedPrivateKey, to_sign_sync))
```

The nonce is issued by the CA in the challenge object (see §5.4.1) and has at least 128 bits of entropy. The semantics of Sign(key, message) are described in §4.4.

A typical implementation of the synchronous mode is the TLS-HANDSHAKE handshake: the CA initiates a TLS connection to the applicant's domain (using the protocol identifier "acme-pk/1"), and the applicant's server returns a proof as application data during the handshake. The CA simultaneously verifies both domain reachability (control) and the signature (proof of private key possession). The time-sensitive nature of the nonce ensures that the signature cannot be precomputed.

4.4. Signature Algorithm Convention

In this document, Sign(key, message) denotes performing a complete signing operation on the raw bytes of message (the algorithm handles the hashing internally; the caller must not perform any additional hashing on message beforehand). The specific algorithm is determined by the key type of the declared public key:

Key type	Signature Algorithm
EC P-256	ECDSA with SHA-256
EC P-384	ECDSA with SHA-384
Ed25519	Ed25519 (PureEdDSA, RFC 8032)
RSA	RSASSA-PSS with SHA-256

Table 3

The server *SHOULD* reject signature algorithms that are not included in the table above or do not comply with the current security baseline requirements, and explicitly declare the set of supported algorithms in the metadata of the Directory resource. Clients *MUST* use algorithms that correspond to the declared public key types; algorithm mismatches will result in signature verification failure.

4.4.1. Explanation of Post-Quantum Algorithms

The set of algorithms currently specified in this document consists primarily of traditional cryptographic algorithms. For post-quantum migration scenarios, implementers may refer to NIST post-quantum standardized algorithms (such as ML-DSA/CRYSTALS-Dilithium [FIPS-204] and SLH-DSA/SPHINCS+ [FIPS-205]), whose algorithm identifiers and SPKI formats have been defined by relevant standards. Servers may negotiate the set of supported post-quantum algorithms through Directory metadata.

Post-quantum signature algorithms produce larger signatures than traditional algorithms, but they still fit within the capacity of DNS TXT records. Taking ML-DSA-87 as an example, its signature is approximately 4627 bytes in length, or about 6170 bytes after Base64 encoding—well below the practical maximum size of a single DNS TXT record ([RFC1035] defines a maximum of 255 bytes per character string, but a single TXT resource record may contain multiple strings, supporting tens of KB in practice). DNS TXT records also

support direct storage of binary data without Base64 encoding, which further reduces storage overhead. For large signatures, the client MUST split the data into multiple character strings and write them into the same TXT record as described in § 5.1.1.

All delivery methods support post-quantum algorithms, and implementers may choose freely based on their deployment environment.

* *DNS Delivery*: Suitable for scenarios where DNS write permissions are available. Large signatures *MUST* be split into multiple strings for recording. If the DNS UDP response exceeds the EDNS0 payload size (typically around 4096 bytes), it will fall back to TCP. Support for TCP fallback varies across some DNS infrastructures, which implementers should take into consideration during deployment.

* *HTTP Delivery*: There are no artificial size limits on HTTP response bodies, allowing post-quantum signatures to be transmitted directly without additional constraints.

* *TLS-HANDSHAKE Delivery (Synchronous Mode)*: The application data layer imposes no message size restrictions, and the nonce mechanism provides built-in freshness guarantees, making this a viable option as well.

The server *SHOULD* declare the set of supported post-quantum algorithms and their corresponding signature lengths in the Directory metadata, for the client to reference when selecting a delivery method.

5. The pk-01 Challenge

The "pk-01" challenge distinguishes between asynchronous and synchronous modes via the pop_mode field in "newOrder". The challenge object declares the list of delivery methods supported by the AS via the supported_delivery field, and the client specifies the selected method via the delivery field in the POST response. All challenge objects adhere to the structure defined in [RFC8555] § 8, and the semantics of the standard fields (url, status, validated, error) remain unchanged.

5.1. Asynchronous mode of DNS

5.1.1. Challenge Object

* *type*: "pk-01"

* *token*: Unpredictable random challenge token (Base64URL-encoded, with an entropy of at least 128 bits) [RFC4086].

```
{
  "type": "pk-01",
  "url": "https://acme.example.com/acme/chall/abc123",
  "status": "pending",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA",
  "supported_delivery": ["dns", "http"]
}
```

Note (DNS TXT Record Length): The maximum length of a single DNS TXT record is 255 bytes ([RFC1035]). The length of the Base64URL-encoded signature value varies depending on the key type: EC P-256/ Ed25519 is approximately 86 characters (well below the limit); RSA-2048 is approximately 342 characters, and RSA-4096 is approximately 683 characters (both exceed the single-string limit). If the signature value exceeds 255 bytes, the client *MUST* split it into multiple strings, each no longer than 255 bytes, and write them into the same TXT resource record ([RFC4408] §3.1.3), and the server *MUST* concatenate the multiple strings in order to reconstruct the original string before performing signature verification. Servers *SHOULD* declare the supported key types and corresponding signature lengths in the Directory metadata to guide clients in selecting the appropriate key type. For DNS delivery of post-quantum algorithm signatures, large signatures *MUST* be split into multiple chunks and written according to the rules described above. If a DNS UDP response exceeds the EDNS0 payload limit (approximately 4096 bytes), the server will fall back to TCP. Support for TCP fallback varies across some DNS infrastructures, which implementers *MUST* evaluate during deployment. All delivery methods support post-quantum algorithms; for details, see Section 4.4 (Post-Quantum Algorithms).

5.1.2. Client Preparation Steps

1) Construct the base keyAuthorization using the standard ACME format:

```
keyAuthorization = token || "." || base64url(JWK_Thumbprint(accountKey))
```

2) Construct the signed message (see §4.4 for the Sign semantics):

```
identifiers_canonical = All identifiers are sorted lexicographically by (type, value), fo
rmatted as "type:value" and concatenated with ",".
to_sign = (SP × 64) || "ACME-pk-01\x00" || keyAuthorization || "." || identifiers_canoni
cal
```

3) Calculate the proof using the claimed private key:

```
proof = base64url(Sign(claimedPrivateKey, to_sign))
```

4) Write proof to the _acme-challenge.<domain> DNS TXT record for the domain:

_acme-challenge.example.com. 120 IN TXT "<proof>"

5) Send a POST request to "url" with the payload {"delivery": "dns"} to notify the server that the allocation is complete and specify that DNS delivery is selected.

5.1.3. Server Validation Steps

1) Look up the _acme-challenge.<domain> TXT resource record to retrieve the proof.

2) Rebuild local signature message:

```
keyAuthorization = token || "." || base64url(JWK_Thumbprint(accountKey))
identifiers_canonical = Reconstructed from the order record as defined in Section 2.
to_sign = (SP × 64) || "ACME-pk-01\x00" || keyAuthorization || "." || identifiers_canoni
cal
```

3) Verify the signature of proof using public_key (the declared public key) (see §4.4 for the semantics of Sign).

4) If validation passes, set the challenge status to "valid"; if validation fails, set it to "invalid".

The DNS query itself confirms the applicant’s control over the domain’s DNS zone, while the signature verification verifies ownership of the private key; both verifications are completed simultaneously in a single operation.

5.2. Asynchronous mode of HTTP

The asynchronous mode supports deploying PoP proofs to an HTTP path on a domain, which the AS retrieves independently via a GET request. Unlike DNS delivery, this method does not require DNS write permissions, but the applicant’ s HTTP server *MUST* be accessible from the outside. The applicant does not need to remain online while the AS performs verification.

5.2.1. Challenge Object

```
{
  "type": "pk-01",
  "url": "https://acme.example.com/acme/chall/def456",
  "status": "pending",
  "token": "DGyRejmCefe7v4NfDGDKfA",
  "supported_delivery": ["dns", "http"]
}
```

5.2.2. Client Preparation Steps

1) Construct a base keyAuthorization value using the standard ACME method (using token):

```
keyAuthorization = token || "." || base64url(JWK_Thumbprint(accountKey))
```

2) Construct a signed message (same as the asynchronous DNS mode, using token):

```
identifiers_canonical = Constructed as defined in Section 2.
to_sign = (SP × 64) || "ACME-pk-01\x00" || keyAuthorization || "." || identifiers_canoni
cal
```

3) Calculate the proof using the claimed private key:

```
proof = base64url(Sign(claimedPrivateKey, to_sign))
```

4) Deploy proof to the following HTTP path (using token as part of the path):

```
http://<domain>/.well-known/acme-challenge/<token>
```

5) Send a POST request to "url" with {"delivery": "http"} in the request body to notify the server that it can proceed with verification.

5.2.3. Server Validation Steps

1) Send an HTTP GET request to http://<domain>/.well-known/acme-challenge/<token> and retrieve the response body as proof. A successful HTTP GET request confirms the applicant’s control over the domain’s HTTP service.

2) Reconstruct the signature message locally:

```
keyAuthorization = token || "." || base64url(JWK_Thumbprint(accountKey))
identifiers_canonical = Reconstructed from the order record as defined in Section 2.
to_sign = (SP × 64) || "ACME-pk-01\x00" || keyAuthorization || "." || identifiers_canoni
cal
```

3) Verify the signature of proof using public_key (the claimed public key) (see §4.4 for the semantics of Sign).

4) If validation passes, set the challenge status to "valid"; if validation fails, set it to "invalid".

The HTTP GET request verifies domain ownership (HTTP reachability), and the signature verification confirms possession of the private key.

5.3. Email Scenario

The "pk-01" email scenario is based on the "email-reply-00" challenge mechanism defined in [RFC8823] and is suitable for email certificate application scenarios such as S/MIME (using the rfc822name type identifier); it always uses the asynchronous mode.

5.3.1. Challenge Object

* *type*: "pk-01"

* *token*: Unpredictable random challenge token (Base64URL-encoded, with an entropy of at least 128 bits) [RFC4086].

5.3.2. Client Preparation Steps

1) The ACME server sends a challenge email containing a token to the email address associated with the order.

2) The applicant constructs a signed message in accordance with §4.2 (including the prefix (SP × 64) || "ACME-pk-01\x00", where identifiers_canonical includes email address identifiers, such as "rfc822name:user@example.com") and computes proof.

3) Send the proof as the body of a reply to the server’s specified address in an S/MIME email.

4) Send a POST request to "url" with the payload { “delivery” : “email” } to notify the server that the email has been sent.

5.3.3. Server Validation Steps

The server receives the applicant’s reply email, extracts the proof from the email body, and performs signature verification using the same logic as in §5.1.3 (identifiers_canonical includes email address identifiers). If the verification passes, the challenge status is set to "valid".

5.4. Synchronous mode of TLS-HANDSHAKE

In synchronous mode, the AS performs real-time verification through a direct TLS handshake with the applicant's server; the applicant *MUST* remain online during the verification process. This document defines a new protocol identifier, "acme-pk/1" (see §2 and §8.3) for this handshake negotiation.

The applicant listens for connections using the token "acme-pk/1" on port 443 of the domain name, completes the handshake using the domain's existing TLS certificate (without constructing a new certificate), and returns the raw proof bytes directly as TLS application data upon handshake completion. When initiating the connection, the AS *MUST* skip certificate chain trust validation (since the EE may use a self-signed certificate or a certificate not yet issued), and *SHOULD* verify that the server certificate public key is byte-for-byte identical to newOrder.public_key for additional binding assurance. This scheme is compatible with TLS 1.2 and TLS 1.3, where the extension is natively supported in both versions.

"acme-pk/1" is a different application-layer protocol from "acme-tls/1" as defined in [RFC8737]: "acme-tls/1" requires the construction of a self-signed X.509 certificate containing OID extensions; in the "acme-pk/1" handshake, the server directly returns the raw proof bytes in the TLS application data, resulting in a simpler implementation with no additional restrictions on post-quantum large-size signatures.

5.4.1. Challenge Object

The sync mode challenge object includes the nonce and supported_delivery fields:

* *type*: "pk-01"

* *nonce*: A new random number generated by CA specifically for this challenge (Base64URL-encoded, with an entropy of at least 128 bits).

* *supported_delivery*: ["tls-handshake"]

```
{
  "type": "pk-01",
  "url": "https://acme.example.com/acme/chall/sync789",
  "status": "pending",
  "nonce": "Kz3mVpQeRd9fLwYbN5hXuT6oJsIc0vAg2nEp1yMrFqZ",
  "supported_delivery": ["tls-handshake"]
}
```

5.4.2. Client Preparation Steps

1) Retrieve the nonce from the challenge source. If it is missing, the client *MUST* terminate and report an error.

2) Construct a synchronous signed message (see §4.4 for the Sign semantics):

```
keyAuthorization_sync = nonce || "." || base64url(JWK_Thumbprint(accountKey))
identifiers_canonical = Constructed as defined in Section 2.
to_sign = (SP × 64) || "ACME-pk-01\x00" || keyAuthorization_sync || "." || identifiers_c
anonical
```

3) Calculate the proof using the claimed private key:

```
proof = base64url(Sign(claimedPrivateKey, to_sign))
```

4) Configure a TLS listener on port 443 of the domain using the domain's existing TLS certificate (no new certificate shall be constructed). When the AS initiates a connection using the identifier "acme-pk/1", return the proof bytes as TLS application data after the handshake is completed, then close the connection immediately.

5) Send a POST request to "url" with {"delivery": "tls-handshake"} in the request body to notify the server that it can proceed with verification.

5.4.3. Server Validation Steps

1) Initiate a TLS connection to <domain>:443 corresponding to the current identifier of the order, using the ALPN protocol identifier "acme-pk/1". The AS *MUST* skip trust validation of the server certificate chain, and *SHOULD* verify that the server certificate public key is byte-for-byte identical to newOrder.public_key. Successful TLS handshake confirms the applicant's control over the TLS service for the domain. After handshake completion, read the application data as the proof.

2) Reconstruct the signed message using the locally stored nonce (*MUST NOT* trust any nonce value provided by the client):

```
keyAuthorization_sync = nonce || "." || base64url(JWK_Thumbprint(accountKey))
identifiers_canonical = Reconstructed from the order record as defined in Section 2.
to_sign = (SP × 64) || "ACME-pk-01\x00" || keyAuthorization_sync || "." || identifiers_c
anonical
```

3) Verify the signature of proof using public_key (the claimed public key) (see §4.4 for the semantics of Sign).

4) If validation passes, set the challenge status to "valid"; if validation fails, set it to "invalid".

A TLS connection verifies domain ownership (TLS reachability), and signature verification confirms possession of the private key.

5.5. Protocol Interaction Process

5.5.1. Asynchronous mode (DNS identifier, csr_less: true)

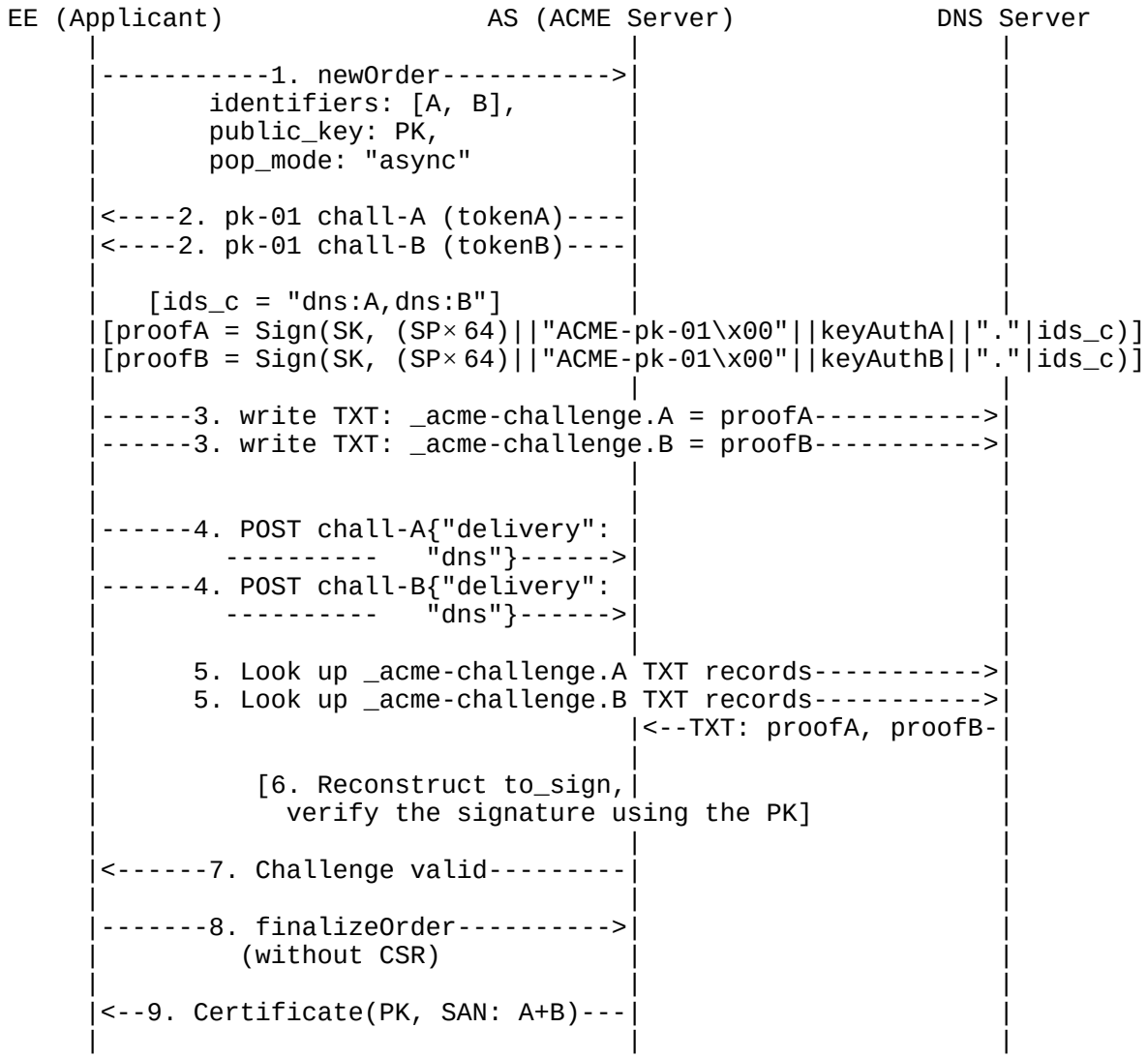

Figure 1: Asynchronous mode (DNS identifier, csr_less: true)

5.5.2. Synchronous mode (TLS-HANDSHAKE delivery, csr_less: true)

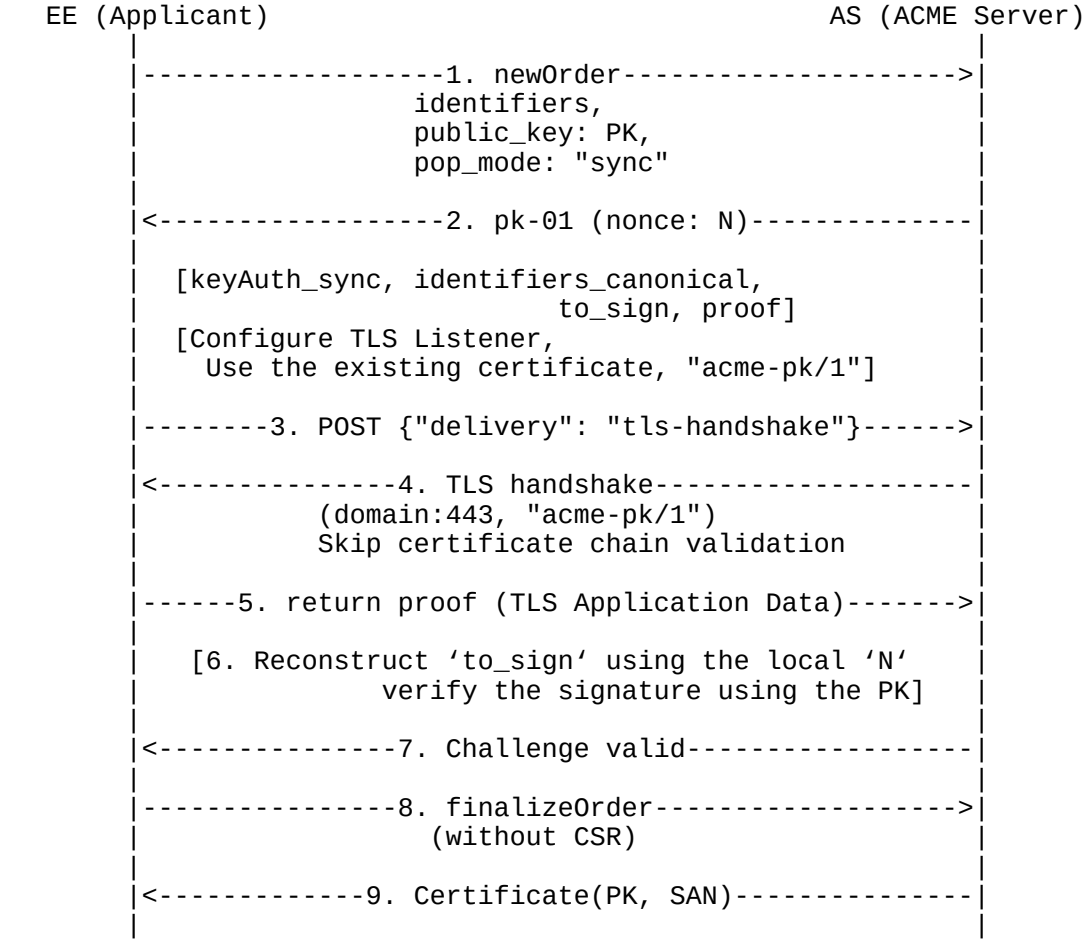

Figure 2: Synchronous mode (TLS-HANDSHAKE delivery, csr_less: true)

6. Finalization

After the "pk-01" challenge was validated, the ACME server has confirmed the following:

* *Resource Control*: Verified via DNS TXT records (asynchronous/DNS), HTTP paths (asynchronous/HTTP), TLS handshakes (synchronous/TLS-HANDSHAKE), or email replies (email scenarios).

* *Proof of Private Key Ownership*: Verified via the proof signature.

* *Claimed public key*: Already claimed in the public_key field during the "newOrder" phase and bound to the order.

The behavior in the finalization phase is determined by the csr_less field in "newOrder":

6.1. csr_less: true（No CSR）

"finalize" requests do not require a CSR, and the body may be an empty object. The server constructs and issues the certificate directly based on the following sources:

* *Public Key*: Taken from the public_key field in "newOrder" (the declared public key, verified via a challenge).

* *Subject Alternative Name (SAN)*: Taken from the identifiers field in "newOrder".

```
POST /acme/order/xyz/finalize HTTP/1.1
Host: acme.example.com
Content-Type: application/jose+json

{}
```

The server issues a certificate using the public_key (claimed public key) as the public key and the domain name in identifiers as the SAN, and returns the certificate download URL.

6.2. csr_less: false（Compatibility mode, default value）

"finalize" requests *MUST* still include a standard PKCS#10 CSR; the process is consistent with the standard ACME [RFC8555] procedure. At this point, the "pk-01" challenge is performed as an additional public key pre-validation step: after verifying the proof of ownership of the claimed public key during the challenge phase, the server performs an additional byte-by-byte comparison during the "finalize" phase to ensure that the public key in the CSR matches the public_key field exactly before issuing the certificate. If the two do not match, the server *MUST* reject the request and return an error.

```
POST /acme/order/xyz/finalize HTTP/1.1
Host: acme.example.com
Content-Type: application/jose+json

{
  "csr": "<DER-encoded PKCS#10 CSR, Base64URL-encoded>"
}
```

6.3. Public Key Consistency Validation and Byte Normalization

Regardless of the value of csr_less, the server *MUST*:

1. Confirm that the order status is "ready" (all challenges have been passed).

2. Perform a strict byte-by-byte comparison between the public key to be written to the certificate and the public_key declared in "newOrder" to ensure they are exactly the same.

To ensure the reliability of byte-level comparison, the server *MUST* treat the raw bytes of the public_key received in the "newOrder" request as the sole authoritative source and *MUST NOT* perform any form of DER normalization, re-encoding, or attribute pruning on it. A single cryptographic key may have multiple valid DER encodings (for example, the ECParameters field of an EC public key can be in OID format or implicit format). If the server normalizes the data during storage or comparison, this can result in false negatives (valid requests being rejected) or false positives (keys with different encodings being mistakenly identified as identical).

7. Security Considerations

7.1. Proof of Key Possession

"pk-01" requires applicants to possess the private key corresponding to the public key they declare. The server *MUST* verify the signature using the public_key in "newOrder" and *MUST NOT* rely on indirect methods to infer ownership of the public key. When issuing a certificate, the server *MUST* compare the public key bytes again to ensure consistency.

7.2. Replay Attack Prevention

In asynchronous mode, the token *MUST* meet the unpredictability requirement (entropy of at least 128 bits) [RFC4086]. In synchronous mode, the nonce *MUST* also have an entropy of at least 128 bits. The server accepts each nonce only once; after use, it *MUST* immediately mark it as consumed. Any subsequent authentication requests carrying the same nonce *MUST* be rejected to prevent replay attacks.

7.3. DNS Control Dependency

In asynchronous mode (DNS identifier), security relies on the applicant having actual control over the corresponding DNS zone. If DNS control is compromised (e.g., through DNS hijacking), an attacker could write a forged signature into the TXT record. Implementers *SHOULD* use this in conjunction with DNSSEC.

7.4. Security Notes for csr_less Mode

After enabling csr_less: true, the finalization phase no longer passes the public key via the CSR; the CA relies entirely on the public_key declared in "newOrder" and validated by the "pk-01" challenge. In this scenario, the proof of private key ownership from the "pk-01" challenge is the sole cryptographic basis for the public key’s legitimacy. The server *MUST* perform byte-level locking on the order’s public key after the challenge verification passes to prevent subsequent requests from replacing the public key.

8. IANA Considerations

8.1. ACME Validation Methods

This document requests that IANA add the following entry to the ACME Validation Methods registry.

Label	Note	Reference
pk-01	Public key challenge with async (DNS/HTTP/email) and sync (TLS-HANDSHAKE) PoP modes	RFC XXX

Table 4

8.2. IANA ACME Message Fields

This document requests that IANA add the following entry to the ACME Validation Fields registry.

newOrder request fields:

Properties	Value
Field Name	public_key
Message Type	newOrder Request
Data Type	String
Presence	OPTIONAL（This must be included only when using the public key challenge extension）
Description	A public key awaiting certification, encoded as a Base64URL-encoded SPKI [RFC5480]. If present, triggers the pk-01 challenge and the CSR-less issuance process.
Reference	RFC XXX

Table 5

Properties	Value
Field Name	pop_mode
Message Type	newOrder Request
Data Type	String
Presence	OPTIONAL（Default value: "async"）
Description	PoP verification mode declared by the client: async (the applicant pre-deploys the proof, and the AS verifies it independently) or “sync” (requires real-time interaction). Extensible.
Reference	RFC XXX

Table 6

Properties	Value
Field Name	csr_less
Message Type	newOrder Request
Data Type	Boolean
Presence	OPTIONAL（Default value:false）
Description	Controls whether the CSR submission is skipped during the finalization phase. true indicates that the certificate is issued directly using the declared public key; false indicates that a PKCS#10 CSR must still be submitted, with the "pk-01" challenge serving as an additional pre-validation step.
Reference	RFC XXX

Table 7

Challenge Target Field:

Properties	Value
Field Name	supported_delivery
Message Type	pk-01 Challenge Object
Data Type	Array of String
Presence	OPTIONAL
Description	A list of available delivery methods declared by AS. Asynchronous modes may include "dns", "http", and "email"; synchronous modes may include "tls-handshake". The client selects one from this list and declares it in the challenge-response.
Reference	RFC XXX

Table 8

Properties	Value
Field Name	delivery
Message Type	pk-01 Challenge Response (POST body)
Data Type	String
Presence	REQUIRED（when supported_delivery is present）
Description	The client must specify the selected delivery method in the POST body of the challenge-response request; this must be one of the values in the supported_delivery list.
Reference	RFC XXX

Table 9

8.3. TLS Protocol Identifier Registration

This document requests that IANA add the following entry to the TLS Application-Layer Protocol Negotiation (ALPN) Protocol IDs registry [RFC7301] with the IETF as the Change Controller:

Protocol	Identification Sequence	Reference
acme-pk/1	0x61 0x63 0x6d 0x65 0x2d 0x70 0x6b 0x2f 0x31 ("acme-pk/1")	RFC XXX

Table 10

The "acme-pk/1" identifier is reserved for TLS-HANDSHAKE delivery in the "pk-01" challenge synchronization mode; its operational semantics differ from those of [RFC8737]: In TLS sessions using "acme-pk/1", the server returns the raw proof bytes directly in the TLS application data after the handshake is complete, without constructing an X.509 certificate or embedding any challenge data in the certificate extension fields.

9. Implementation Considerations

9.1. ACME Server

* When processing a "newOrder" request, the server must perform strict format validation on the public_key field to verify that it is a valid DER-encoded SPKI; for unsupported key types, it *SHOULD* return an error. The server *MUST* store public_key in the raw bytes received and must not perform any DER normalization or re-encoding (see § 6.3).

* The server must verify the uniqueness of the token in asynchronous mode and the nonce in synchronous mode to prevent token reuse.

* Before issuing the certificate, the server must verify the public key bytes once again to ensure they match exactly, byte-for-byte, with the public key declared in "newOrder".

* The server *MUST* mark the nonce as used immediately after it is consumed for the first time and reject any subsequent attempts to reuse it. The server should limit the validity window of the nonce (*RECOMMENDED* to be no longer than the challenge token’s validity period).

* When executing an asynchronous HTTP GET request, the server *MUST NOT* follow HTTP redirects that change the target domain to a different domain (cross-domain redirects), as following such redirects would invalidate the domain control verification. The server *SHOULD* allow same-domain redirects (such as HTTP → HTTPS redirects within the same host) and adhere to the standard ACME http-01 HTTP request specifications (timeouts, maximum number of redirects, etc.) to prevent server-side request forgery (SSRF).

* The server *SHOULD* include a supported_delivery field in the challenge object to declare all supported delivery methods for the client to choose from.

* After the server receives the Challenge-Response POST request, it must read the delivery field to determine which verification method to use; if the delivery field is missing or contains an unsupported value, the server *SHOULD* return an error.

* When the server returns an error response related to the "pk-01" challenge, it *SHOULD* indicate the delivery method that triggered the error in the detail field of the [RFC7807] "Problem Details" format, following the format: [<delivery>] <error description>. If the client has not declared a delivery (i.e., the delivery field is missing), it shall be marked as [pk-01].

```
{
  "type": "urn:ietf:params:acme:error:rejectedIdentifier",
  "detail": "public key delivery field missing or unsupported value",
  "status": 400
}
```

```
{
  "type": "urn:ietf:params:acme:error:invalidDeliveryMethod",
  "detail": "invalid delivery method for current pk-01 challenge",
  "status": 403
}
```

* For orders containing multiple identifiers, the server issues and validates challenges independently for each individual identifier. When reconstructing to_sign for signature verification, the server *MUST* construct identifiers_canonical from all identifiers of the order as defined in § 2, rather than only the single identifier corresponding to the current challenge.

* During validation in synchronous mode (TLS-HANDSHAKE), the server *MUST* skip trust validation of the TLS certificate chain presented by the client, and *SHOULD* verify that the public key in the server certificate is byte-for-byte identical to newOrder.public_key to prevent man-in-the-middle relay attacks.

9.2. ACME Client

* The client *MUST* ensure that the private key used for signing strictly matches the public_key declared in "newOrder"; a mismatch in key types will cause the server to fail the signature verification.

* In synchronous mode, the client *MUST* send a response to the challenge URL only after the TLS listening configuration is complete, to ensure it is reachable when the server initiates the TLS handshake.

* In asynchronous mode, the client *SHOULD* verify that the resource (DNS TXT record or HTTP path) has taken effect before notifying the server, taking into account DNS propagation delays or HTTP service availability.

* When csr_less: true, the body of the "finalize" request may be an empty object {}, the client does not need to construct a CSR; when csr_less: false (default), the client *MUST* still submit a standard PKCS#10 CSR during the "finalize" phase, and the public key in the CSR *MUST* match the public_key declared in "newOrder".

* When sending a challenge-response POST request, the client *MUST* include a delivery field in the request body, and its value must be one of the items in the supported_delivery list of the challenge object.

* For orders with multiple identifiers, when constructing the proof for each challenge, the client *MUST* generate identifiers_canonical from *all identifiers of the order* in accordance with § 2. All challenges use the identical value of identifiers_canonical.

10. Examples

10.1. Asynchronous Mode: pk-01 (DNS) Complete Interaction, Single identifier

Prerequisites: The applicant *MUST* have DNS control over the domain example.com and a P-256 key pair (private key d, public key Q, SPKI-encoded as pk_spki).

Step 1: newOrder Request

```
POST /acme/new-order HTTP/1.1
Host: acme.example.com
Content-Type: application/jose+json

{
  "identifiers": [
    { "type": "dns", "value": "example.com" }
  ],
  "public_key": "pk_spki",
  "pop_mode": "async",
  "csr_less": true
}
```

Step 2: The server returns the pk-01 challenge

```
{
  "type": "pk-01",
  "url": "https://acme.example.com/acme/chall/abc123",
  "status": "pending",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA"
  "supported_delivery": ["dns", "http"]
}
```

Step 3: The client generates a signature and writes it to the DNS TXT record

```
keyAuthorization = "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA.pswg5_v_JaVFRXrHxGfJkg"
identifiers_canonical = "dns:example.com"
to_sign = (SP × 64) || "ACME-pk-01\x00" || keyAuthorization || "." || identifiers_canoni
cal
proof = base64url(Sign(d, to_sign))  →  "<proof_value>"
# The Sign() function uses ECDSA-with-SHA256 (P-256 key) and internally signs the ‘to_sig
n‘ value after applying SHA-256 to it.

_acme-challenge.example.com.  120  IN  TXT  "<proof_value>"
```

Step 4: The client notifies the server, and the server queries and verifies

```
POST /acme/chall/abc123 HTTP/1.1
Content-Type: application/jose+json

{"delivery": "dns"}
```

The server queries the TXT record for _acme-challenge.example.com, reconstructs to_sign with identifiers_canonical = "dns:example.com", verifies the signature using pk_spki, and sets the challenge status to valid upon successful verification.

Step 5: finalizeOrder (without CSR)

```
POST /acme/order/xyz/finalize HTTP/1.1
Content-Type: application/jose+json

{}
```

The server issues a certificate using pk_spki as the public key and example.com as the subject alternative name (SAN).

10.2. Asynchronous Mode: pk-01 (HTTP) Complete Interaction, Without Extensions

Prerequisites: The applicant *MUST* have HTTP control over the domain example.com and a P-256 key pair.

Step 1: newOrder Request

```
POST /acme/new-order HTTP/1.1
Content-Type: application/jose+json

{
  "identifiers": [
    { "type": "dns", "value": "example.com" }
  ],
  "public_key": "pk_spki",
  "pop_mode": "async",
  "csr_less": true
}
```

Step 2: The server returns the pk-01 challenge

```
{
  "type": "pk-01",
  "url": "https://acme.example.com/acme/chall/def456",
  "status": "pending",
  "token": "DGyRejmCefe7v4NfDGDKfA",
  "supported_delivery": ["dns", "http"]
}
```

Step 3: The client generates a signature and deploys it to the HTTP path

```
keyAuthorization = "DGyRejmCefe7v4NfDGDKfA.pswg5_v_JaVFRXrHxGfJkg"
identifiers_canonical = "dns:example.com"
to_sign = (SP × 64) || "ACME-pk-01\x00" || keyAuthorization || "." || identifiers_canoni
cal
proof = base64url(Sign(d, to_sign))
# The Sign() function uses ECDSA-with-SHA256 (P-256 key) and internally signs the ‘to_sig
n‘ value after applying SHA-256 to it.
```

Deploy to: http://example.com/.well-known/acme-challenge/
DGyRejmCefe7v4NfDGDKfA

Content: <proof>

Step 4: The client notifies the server, and the server initiates an HTTP GET authentication request

```
POST /acme/chall/def456 HTTP/1.1
Content-Type: application/jose+json

{"delivery": "http"}
```

The server sends an HTTP GET request to http://example.com/.well-known/acme-challenge/DGyRejmCefe7v4NfDGDKfA to verify domain ownership. After validating the signature, it sets the challenge status to "valid".

Step 5: finalizeOrder (without CSR)

The server issues a certificate using pk_spki as the public key and example.com as the subject alternative name (SAN).

11. Normative References

[RFC8555] Barnes, R., Hoffman-Andrews, J., McCarney, D., and J. Kasten, "Automatic Certificate Management Environment (ACME)", RFC 8555, DOI 10.17487/RFC8555, March 2019, <https://www.rfc-editor.org/info/rfc8555>.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <https://www.rfc-editor.org/info/rfc5480>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC8737] Shoemaker, R.B., "Automated Certificate Management Environment (ACME) TLS Application-Layer Protocol Negotiation (ALPN) Challenge Extension", RFC 8737, DOI 10.17487/RFC8737, February 2020, <https://www.rfc-editor.org/info/rfc8737>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4408] Wong, M. and W. Schlitt, "Sender Policy Framework (SPF) for Authorizing Use of Domains in E-Mail, Version 1", RFC 4408, DOI 10.17487/RFC4408, April 2006, <https://www.rfc-editor.org/info/rfc4408>.

[RFC8823] Melnikov, A., "Extensions to Automatic Certificate Management Environment for End-User S/MIME Certificates", RFC 8823, DOI 10.17487/RFC8823, April 2021, <https://www.rfc-editor.org/info/rfc8823>.

[RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016, <https://www.rfc-editor.org/info/rfc7807>.

Authors' Addresses

Feng Geng
Huawei Technologies
Email: gengfeng@huawei.com

Panyu Wu
Huawei Technologies
Email: wupanyu3@huawei.com

Liang Xia
Huawei Technologies
Email: frank.xialiang@huawei.com

Xin Chen
TrustAsia
Email: palos.chen@trustasia.com