Internet Engineering Task Force (IETF)
Request for Comments: 8555
Category: Standards Track
ISSN: 2070-1721

R. Barnes
Cisco
J. Hoffman-Andrews
EFF
D. McCarney
Let's Encrypt
J. Kasten
University of Michigan

March 2019

# Automatic Certificate Management Environment (ACME)

## Abstract

Public Key Infrastructure using X.509 (PKIX) certificates are used for a number of purposes, the most significant of which is the authentication of domain names. Thus, certification authorities (CAs) in the Web PKI are trusted to verify that an applicant for a certificate legitimately represents the domain name(s) in the certificate. As of this writing, this verification is done through a collection of ad hoc mechanisms. This document describes a protocol that a CA and an applicant can use to automate the process of verification and certificate issuance. The protocol also provides facilities for other certificate management functions, such as certificate revocation.

## Status of This Memo

## Table of Contents

## 1. Introduction

Certificates [RFC5280] in the Web PKI are most commonly used to authenticate domain names. Thus, certification authorities (CAs) in the Web PKI are trusted to verify that an applicant for a certificate legitimately represents the domain name(s) in the certificate.

Different types of certificates reflect different kinds of CA verification of information about the certificate subject. "Domain Validation" (DV) certificates are by far the most common type. The only validation the CA is required to perform in the DV issuance process is to verify that the requester has effective control of the domain [CABFBR]. The CA is not required to attempt to verify the requester's real-world identity. (This is as opposed to "Organization Validation" (OV) and "Extended Validation" (EV) certificates, where the process is intended to also verify the real-world identity of the requester.)

Existing Web PKI certification authorities tend to use a set of ad hoc protocols for certificate issuance and identity verification. In the case of DV certificates, a typical user experience is something like:

- Generate a PKCS#10 [RFC2986] Certificate Signing Request (CSR).

- Cut and paste the CSR into a CA's web page.

- Prove ownership of the domain(s) in the CSR by one of the following methods:

  * Put a CA-provided challenge at a specific place on the web server.

  * Put a CA-provided challenge in a DNS record corresponding to the target domain.

  * Receive a CA-provided challenge at (hopefully) an administrator-controlled email address corresponding to the domain, and then respond to it on the CA's web page.

- Download the issued certificate and install it on the user's Web Server.

With the exception of the CSR itself and the certificates that are issued, these are all completely ad hoc procedures and are accomplished by getting the human user to follow interactive natural-language instructions from the CA rather than by machine-implemented published protocols. In many cases, the instructions are difficult to follow and cause significant frustration and confusion. Informal usability tests by the authors indicate that webmasters often need 1-3 hours to obtain and install a certificate for a domain. Even in the best case, the lack of published, standardized mechanisms presents an obstacle to the wide deployment of HTTPS and other PKIX-dependent systems because it inhibits mechanization of tasks related to certificate issuance, deployment, and revocation.

This document describes an extensible framework for automating the issuance and domain validation procedure, thereby allowing servers and infrastructure software to obtain certificates without user interaction. Use of this protocol should radically simplify the deployment of HTTPS and the practicality of PKIX-based authentication for other protocols based on Transport Layer Security (TLS) [RFC8446].

It should be noted that while the focus of this document is on validating domain names for purposes of issuing certificates in the Web PKI, ACME supports extensions for uses with other identifiers in other PKI contexts. For example, as of this writing, there is ongoing work to use ACME for issuance of Web PKI certificates attesting to IP addresses [ACME-IP] and Secure Telephone Identity Revisited (STIR) certificates attesting to telephone numbers [ACME-TELEPHONE].

ACME can also be used to automate some aspects of certificate management even where non-automated processes are still needed. For example, the external account binding feature (see Section 7.3.4) can allow an ACME account to use authorizations that have been granted to an external, non-ACME account. This allows ACME to address issuance scenarios that cannot yet be fully automated, such as the issuance of "Extended Validation" certificates.

## 2. Deployment Model and Operator Experience

The guiding use case for ACME is obtaining certificates for websites (HTTPS [RFC2818]). In this case, a web server is intended to speak for one or more domains, and the process of certificate issuance is intended to verify that this web server actually speaks for the domain(s).

DV certificate validation commonly checks claims about properties related to control of a domain name -- properties that can be observed by the certificate issuer in an interactive process that can be conducted purely online. That means that under typical circumstances, all steps in the request, verification, and issuance process can be represented and performed by Internet protocols with no out-of-band human intervention.

Prior to ACME, when deploying an HTTPS server, a server operator typically gets a prompt to generate a self-signed certificate. If the operator were instead deploying an HTTPS server using ACME, the experience would be something like this:

o The operator's ACME client prompts the operator for the intended domain name(s) that the web server is to stand for.

o The ACME client presents the operator with a list of CAs from which it could get a certificate. (This list will change over time based on the capabilities of CAs and updates to ACME configuration.) The ACME client might prompt the operator for payment information at this point.

o The operator selects a CA.

o In the background, the ACME client contacts the CA and requests that it issue a certificate for the intended domain name(s).

o The CA verifies that the client controls the requested domain name(s) by having the ACME client perform some action(s) that can only be done with control of the domain name(s). For example, the CA might require a client requesting example.com to provision a DNS record under example.com or an HTTP resource under http://example.com.

o Once the CA is satisfied, it issues the certificate and the ACME client automatically downloads and installs it, potentially notifying the operator via email, SMS, etc.

o The ACME client periodically contacts the CA to get updated certificates, stapled Online Certificate Status Protocol (OCSP) responses [RFC6960], or whatever else would be required to keep the web server functional and its credentials up to date.

In this way, it would be nearly as easy to deploy with a CA-issued certificate as with a self-signed certificate. Furthermore, the maintenance of that CA-issued certificate would require minimal manual intervention. Such close integration of ACME with HTTPS servers allows the immediate and automated deployment of certificates as they are issued, sparing the human administrator from much of the time-consuming work described in the previous section.

## 3. Terminology

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.

The two main roles in ACME are "client" and "server". The ACME client uses the protocol to request certificate management actions, such as issuance or revocation. An ACME client may run on a web server, mail server, or some other server system that requires valid X.509 certificates. Or, it may run on a separate server that does not consume the certificate but is authorized to respond to a CA-provided challenge. The ACME server runs at a certification authority and responds to client requests, performing the requested actions if the client is authorized.

An ACME client authenticates to the server by means of an "account key pair". The client uses the private key of this key pair to sign all messages sent to the server. The server uses the public key to verify the authenticity and integrity of messages from the client.

## 4. Protocol Overview

ACME allows a client to request certificate management actions using a set of JavaScript Object Notation (JSON) messages [RFC8259] carried over HTTPS [RFC2818]. Issuance using ACME resembles a traditional CA's issuance process, in which a user creates an account, requests a certificate, and proves control of the domain(s) in that certificate in order for the CA to issue the requested certificate.

The first phase of ACME is for the client to request an account with the ACME server. The client generates an asymmetric key pair and requests a new account, optionally providing contact information, agreeing to terms of service (ToS), and/or associating the account with an existing account in another system. The creation request is signed with the generated private key to prove that the client controls it.

```
      Client                                                  Server

      [Contact Information]
      [ToS Agreement]
      [Additional Data]
      Signature                     ------->
                                                         Account URL
                                    <-------          Account Object


                [] Information covered by request signatures

                              Account Creation
```

Once an account is registered, there are four major steps the client needs to take to get a certificate:

1. Submit an order for a certificate to be issued

2. Prove control of any identifiers requested in the certificate

3. Finalize the order by submitting a CSR

4. Await issuance and download the issued certificate

The client's order for a certificate describes the desired identifiers plus a few additional fields that capture semantics that are not supported in the CSR format. If the server is willing to consider issuing such a certificate, it responds with a list of requirements that the client must satisfy before the certificate will be issued.

For example, in most cases, the server will require the client to demonstrate that it controls the identifiers in the requested certificate. Because there are many different ways to validate possession of different types of identifiers, the server will choose from an extensible set of challenges that are appropriate for the identifier being claimed. The client responds with a set of responses that tell the server which challenges the client has completed. The server then validates that the client has completed the challenges.

Once the validation process is complete and the server is satisfied that the client has met its requirements, the client finalizes the order by submitting a PKCS#10 Certificate Signing Request (CSR). The server will issue the requested certificate and make it available to the client.

```
         Client                                                   Server

         [Order]
         Signature                     ------->
                                       <-------  Required Authorizations

         [Responses]
         Signature                     ------->

                             <~~~~~~~~Validation~~~~~~~~>

         [CSR]
         Signature                     ------->
                                       <-------          Acknowledgement

                             <~~~~~~Await issuance~~~~~~>

         [POST-as-GET request]
         Signature                     ------->
                                       <-------              Certificate

                   [] Information covered by request signatures

                               Certificate Issuance
```

To revoke a certificate, the client sends a signed revocation request indicating the certificate to be revoked:

```
         Client                                                 Server

         [Revocation request]
         Signature                    -------->

                                      <--------                 Result

                   [] Information covered by request signatures

                              Certificate Revocation
```

Note that while ACME is defined with enough flexibility to handle different types of identifiers in principle, the primary use case addressed by this document is the case where domain names are used as identifiers.  For example, all of the identifier validation challenges described in Section 8 address validation of domain names. The use of ACME for other identifiers will require further specification in order to describe how these identifiers are encoded in the protocol and what types of validation challenges the server might require.

## 5. Character Encoding

All requests and responses sent via HTTP by ACME clients, ACME servers, and validation servers as well as any inputs for digest computations MUST be encoded using the UTF-8 character set [RFC3629]. Note that identifiers that appear in certificates may have their own encoding considerations (e.g., DNS names containing non-ASCII characters are expressed as A-labels rather than U-labels). Any such encoding considerations are to be applied prior to the aforementioned UTF-8 encoding.

## 6. Message Transport

Communications between an ACME client and an ACME server are done over HTTPS, using JSON Web Signature (JWS) [RFC7515] to provide some additional security properties for messages sent from the client to the server. HTTPS provides server authentication and confidentiality. With some ACME-specific extensions, JWS provides authentication of the client's request payloads, anti-replay protection, and integrity for the HTTPS request URL.

### 6.1. HTTPS Requests

Each ACME function is accomplished by the client sending a sequence of HTTPS requests to the server [RFC2818], carrying JSON messages [RFC8259]. Use of HTTPS is REQUIRED. Each subsection of Section 7 below describes the message formats used by the function and the order in which messages are sent.

In most HTTPS transactions used by ACME, the ACME client is the HTTPS client and the ACME server is the HTTPS server. The ACME server acts as a client when validating challenges: an HTTP client when validating an 'http-01' challenge, a DNS client with 'dns-01', etc.

ACME servers SHOULD follow the recommendations of [RFC7525] when configuring their TLS implementations. ACME servers that support TLS 1.3 MAY allow clients to send early data (0-RTT). This is safe because the ACME protocol itself includes anti-replay protections (see Section 6.5) in all cases where they are required. For this reason, there are no restrictions on what ACME data can be carried in 0-RTT.

ACME clients MUST send a User-Agent header field, in accordance with [RFC7231]. This header field SHOULD include the name and version of the ACME software in addition to the name and version of the underlying HTTP client software.

ACME clients SHOULD send an Accept-Language header field in accordance with [RFC7231] to enable localization of error messages.

ACME servers that are intended to be generally accessible need to use Cross-Origin Resource Sharing (CORS) in order to be accessible from browser-based clients [W3C.REC-cors-20140116]. Such servers SHOULD set the Access-Control-Allow-Origin header field to the value "*".

Binary fields in the JSON objects used by ACME are encoded using base64url encoding described in Section 5 of [RFC4648] according to the profile specified in JSON Web Signature in Section 2 of [RFC7515]. This encoding uses a URL safe character set. Trailing '=' characters MUST be stripped. Encoded values that include trailing '=' characters MUST be rejected as improperly encoded.

## 6.2. Request Authentication

All ACME requests with a non-empty body MUST encapsulate their payload in a JSON Web Signature (JWS) [RFC7515] object, signed using the account's private key unless otherwise specified. The server MUST verify the JWS before processing the request. Encapsulating request bodies in JWS provides authentication of requests.

A JWS object sent as the body of an ACME request MUST meet the following additional criteria:

- The JWS MUST be in the Flattened JSON Serialization [RFC7515]
- The JWS MUST NOT have multiple signatures
- The JWS Unencoded Payload Option [RFC7797] MUST NOT be used
- The JWS Unprotected Header [RFC7515] MUST NOT be used
- The JWS Payload MUST NOT be detached
- The JWS Protected Header MUST include the following fields:
    * "alg" (Algorithm)
        + This field MUST NOT contain "none" or a Message Authentication Code (MAC) algorithm (e.g. one in which the algorithm registry description mentions MAC/HMAC).
    * "nonce" (defined in Section 6.5)
    * "url" (defined in Section 6.4)

   * Either "jwk" (JSON Web Key) or "kid" (Key ID) as specified below

An ACME server MUST implement the "ES256" signature algorithm [RFC7518] and SHOULD implement the "EdDSA" signature algorithm using the "Ed25519" variant (indicated by "crv") [RFC8037].

The "jwk" and "kid" fields are mutually exclusive.  Servers MUST reject requests that contain both.

For newAccount requests, and for revokeCert requests authenticated by a certificate key, there MUST be a "jwk" field.  This field MUST contain the public key corresponding to the private key used to sign the JWS.

For all other requests, the request is signed using an existing account, and there MUST be a "kid" field.  This field MUST contain the account URL received by POSTing to the newAccount resource.

If the client sends a JWS signed with an algorithm that the server does not support, then the server MUST return an error with status code 400 (Bad Request) and type "urn:ietf:params:acme:error:badSignatureAlgorithm".  The problem document returned with the error MUST include an "algorithms" field with an array of supported "alg" values.  See Section 6.7 for more details on the structure of error responses.

If the server supports the signature algorithm "alg" but either does not support or chooses to reject the public key "jwk", then the server MUST return an error with status code 400 (Bad Request) and type "urn:ietf:params:acme:error:badPublicKey".  The problem document detail SHOULD describe the reason for rejecting the public key; some example reasons are:

o  "alg" is "RS256" but the modulus "n" is too small (e.g., 512-bit)

o  "alg" is "ES256" but "jwk" does not contain a valid P-256 public key

o  "alg" is "EdDSA" and "crv" is "Ed448", but the server only supports "EdDSA" with "Ed25519"

o  the corresponding private key is known to have been compromised

Because client requests in ACME carry JWS objects in the Flattened JSON Serialization, they must have the Content-Type header field set to "application/jose+json". If a request does not meet this requirement, then the server MUST return a response with status code 415 (Unsupported Media Type).

## 6.3. GET and POST-as-GET Requests

Note that authentication via signed JWS request bodies implies that requests without an entity body are not authenticated, in particular GET requests. Except for the cases described in this section, if the server receives a GET request, it MUST return an error with status code 405 (Method Not Allowed) and type "malformed".

If a client wishes to fetch a resource from the server (which would otherwise be done with a GET), then it MUST send a POST request with a JWS body as described above, where the payload of the JWS is a zero-length octet string. In other words, the "payload" field of the JWS object MUST be present and set to the empty string ("").

We will refer to these as "POST-as-GET" requests. On receiving a request with a zero-length (and thus non-JSON) payload, the server MUST authenticate the sender and verify any access control rules. Otherwise, the server MUST treat this request as having the same semantics as a GET request for the same resource.

The server MUST allow GET requests for the directory and newNonce resources (see Section 7.1), in addition to POST-as-GET requests for these resources. This enables clients to bootstrap into the ACME authentication system.

## 6.4. Request URL Integrity

It is common in deployment for the entity terminating TLS for HTTPS to be different from the entity operating the logical HTTPS server, with a "request routing" layer in the middle. For example, an ACME CA might have a content delivery network terminate TLS connections from clients so that it can inspect client requests for denial-of-service (DoS) protection.

These intermediaries can also change values in the request that are not signed in the HTTPS request, e.g., the request URL and header fields. ACME uses JWS to provide an integrity mechanism, which protects against an intermediary changing the request URL to another ACME URL.

As noted in Section 6.2, all ACME request objects carry a "url" header parameter in their protected header. This header parameter encodes the URL to which the client is directing the request. On receiving such an object in an HTTP request, the server MUST compare the "url" header parameter to the request URL. If the two do not match, then the server MUST reject the request as unauthorized.

Except for the directory resource, all ACME resources are addressed with URLs provided to the client by the server. In POST requests sent to these resources, the client MUST set the "url" header parameter to the exact string provided by the server (rather than performing any re-encoding on the URL). The server SHOULD perform the corresponding string equality check, configuring each resource with the URL string provided to clients and having the resource check that requests have the same string in their "url" header parameter. The server MUST reject the request as unauthorized if the string equality check fails.

#### 6.4.1. "url" (URL) JWS Header Parameter

The "url" header parameter specifies the URL [RFC3986] to which this JWS object is directed. The "url" header parameter MUST be carried in the protected header of the JWS. The value of the "url" header parameter MUST be a string representing the target URL.

### 6.5. Replay Protection

In order to protect ACME resources from any possible replay attacks, ACME POST requests have a mandatory anti-replay mechanism. This mechanism is based on the server maintaining a list of nonces that it has issued, and requiring any signed request from the client to carry such a nonce.

An ACME server provides nonces to clients using the HTTP Replay-Nonce header field, as specified in Section 6.5.1. The server MUST include a Replay-Nonce header field in every successful response to a POST request and SHOULD provide it in error responses as well.

Every JWS sent by an ACME client MUST include, in its protected header, the "nonce" header parameter, with contents as defined in Section 6.5.2. As part of JWS verification, the ACME server MUST verify that the value of the "nonce" header is a value that the server previously provided in a Replay-Nonce header field. Once a nonce value has appeared in an ACME request, the server MUST consider it invalid, in the same way as a value it had never issued.

When a server rejects a request because its nonce value was unacceptable (or not present), it MUST provide HTTP status code 400 (Bad Request), and indicate the ACME error type "urn:ietf:params:acme:error:badNonce". An error response with the "badNonce" error type MUST include a Replay-Nonce header field with a fresh nonce that the server will accept in a retry of the original query (and possibly in other requests, according to the server's nonce scoping policy). On receiving such a response, a client SHOULD retry the request using the new nonce.

The precise method used to generate and track nonces is up to the server. For example, the server could generate a random 128-bit value for each response, keep a list of issued nonces, and strike nonces from this list as they are used.

Other than the constraint above with regard to nonces issued in "badNonce" responses, ACME does not constrain how servers scope nonces. Clients MAY assume that nonces have broad scope, e.g., by having a single pool of nonces used for all requests. However, when retrying in response to a "badNonce" error, the client MUST use the nonce provided in the error response. Servers should scope nonces broadly enough that retries are not needed very often.

### 6.5.1. Replay-Nonce

The Replay-Nonce HTTP header field includes a server-generated value that the server can use to detect unauthorized replay in future client requests. The server MUST generate the values provided in Replay-Nonce header fields in such a way that they are unique to each message, with high probability, and unpredictable to anyone besides the server. For instance, it is acceptable to generate Replay-Nonces randomly.

The value of the Replay-Nonce header field MUST be an octet string encoded according to the base64url encoding described in Section 2 of [RFC7515]. Clients MUST ignore invalid Replay-Nonce values. The ABNF [RFC5234] for the Replay-Nonce header field follows:

```
base64url = ALPHA / DIGIT / "-" / "_"

Replay-Nonce = 1*base64url
```

The Replay-Nonce header field SHOULD NOT be included in HTTP request messages.

#### 6.5.2. "nonce" (Nonce) JWS Header Parameter

The "nonce" header parameter provides a unique value that enables the verifier of a JWS to recognize when replay has occurred. The "nonce" header parameter MUST be carried in the protected header of the JWS.

The value of the "nonce" header parameter MUST be an octet string, encoded according to the base64url encoding described in Section 2 of [RFC7515]. If the value of a "nonce" header parameter is not valid according to this encoding, then the verifier MUST reject the JWS as malformed.

### 6.6. Rate Limits

Creation of resources can be rate limited by ACME servers to ensure fair usage and prevent abuse. Once the rate limit is exceeded, the server MUST respond with an error with the type "urn:ietf:params:acme:error:rateLimited". Additionally, the server SHOULD send a Retry-After header field [RFC7231] indicating when the current request may succeed again. If multiple rate limits are in place, that is the time where all rate limits allow access again for the current request with exactly the same parameters.

In addition to the human-readable "detail" field of the error response, the server MAY send one or multiple link relations in the Link header field [RFC8288] pointing to documentation about the specific rate limit that was hit, using the "help" link relation type.

### 6.7. Errors

Errors can be reported in ACME both at the HTTP layer and within challenge objects as defined in Section 8. ACME servers can return responses with an HTTP error response code (4XX or 5XX). For example, if the client submits a request using a method not allowed in this document, then the server MAY return status code 405 (Method Not Allowed).

When the server responds with an error status, it SHOULD provide additional information using a problem document [RFC7807]. To facilitate automatic response to errors, this document defines the following standard tokens for use in the "type" field (within the ACME URN namespace "urn:ietf:params:acme:error:"):

| Type | Description |
|---|---|
| accountDoesNotExist | The request specified an account that does not exist |
| alreadyRevoked | The request specified a certificate to be revoked that has already been revoked |
| badCSR | The CSR is unacceptable (e.g., due to a short key) |
| badNonce | The client sent an unacceptable anti-replay nonce |
| badPublicKey | The JWS was signed by a public key the server does not support |
| badRevocationReason | The revocation reason provided is not allowed by the server |
| badSignatureAlgorithm | The JWS was signed with an algorithm the server does not support |
| caa | Certification Authority Authorization (CAA) records forbid the CA from issuing a certificate |
| compound | Specific error conditions are indicated in the "subproblems" array |
| connection | The server could not connect to validation target |
| dns | There was a problem with a DNS query during identifier validation |
| externalAccountRequired | The request must include a value for the "externalAccountBinding" field |
| incorrectResponse | Response received didn't match the challenge's requirements |
| invalidContact | A contact URL for an account was invalid |
| malformed | The request message was malformed |

| | |
|---|---|
| orderNotReady | The request attempted to finalize an order that is not ready to be finalized |
| rateLimited | The request exceeds a rate limit |
| rejectedIdentifier | The server will not issue certificates for the identifier |
| serverInternal | The server experienced an internal error |
| tls | The server received a TLS error during validation |
| unauthorized | The client lacks sufficient authorization |
| unsupportedContact | A contact URL for an account used an unsupported protocol scheme |
| unsupportedIdentifier | An identifier is of an unsupported type |
| userActionRequired | Visit the "instance" URL and take actions specified there |

This list is not exhaustive. The server MAY return errors whose "type" field is set to a URI other than those defined above. Servers MUST NOT use the ACME URN namespace for errors not listed in the appropriate IANA registry (see Section 9.6). Clients SHOULD display the "detail" field of all errors.

In the remainder of this document, we use the tokens in the table above to refer to error types, rather than the full URNs. For example, an "error of type 'badCSR'" refers to an error document with "type" value "urn:ietf:params:acme:error:badCSR".

### 6.7.1. Subproblems

Sometimes a CA may need to return multiple errors in response to a request. Additionally, the CA may need to attribute errors to specific identifiers. For instance, a newOrder request may contain multiple identifiers for which the CA cannot issue certificates. In this situation, an ACME problem document MAY contain the "subproblems" field, containing a JSON array of problem documents, each of which MAY contain an "identifier" field. If present, the "identifier" field MUST contain an ACME identifier (Section 9.7.7).

The "identifier" field MUST NOT be present at the top level in ACME problem documents. It can only be present in subproblems. Subproblems need not all have the same type, and they do not need to match the top level type.

ACME clients may choose to use the "identifier" field of a subproblem as a hint that an operation would succeed if that identifier were omitted. For instance, if an order contains ten DNS identifiers, and the newOrder request returns a problem document with two subproblems (referencing two of those identifiers), the ACME client may choose to submit another order containing only the eight identifiers not listed in the problem document.

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Link: <https://example.com/acme/directory>;rel="index"

{
    "type": "urn:ietf:params:acme:error:malformed",
    "detail": "Some of the identifiers requested were rejected",
    "subproblems": [
        {
            "type": "urn:ietf:params:acme:error:malformed",
            "detail": "Invalid underscore in DNS name \"_example.org\"",
            "identifier": {
                "type": "dns",
                "value": "_example.org"
            }
        },
        {
            "type": "urn:ietf:params:acme:error:rejectedIdentifier",
            "detail": "This CA will not issue for \"example.net\"",
            "identifier": {
                "type": "dns",
                "value": "example.net"
            }
        }
    ]
}
```

## 7. Certificate Management

In this section, we describe the certificate management functions that ACME enables:

- o Account Creation
- o Ordering a Certificate
- o Identifier Authorization
- o Certificate Issuance
- o Certificate Revocation

### 7.1. Resources

ACME is structured as an HTTP-based application with the following types of resources:

- o Account resources, representing information about an account (Section 7.1.2, Section 7.3)
- o Order resources, representing an account's requests to issue certificates (Section 7.1.3)
- o Authorization resources, representing an account's authorization to act for an identifier (Section 7.1.4)
- o Challenge resources, representing a challenge to prove control of an identifier (Section 7.5, Section 8)
- o Certificate resources, representing issued certificates (Section 7.4.2)
- o A "directory" resource (Section 7.1.1)
- o A "newNonce" resource (Section 7.2)
- o A "newAccount" resource (Section 7.3)
- o A "newOrder" resource (Section 7.4)
- o A "revokeCert" resource (Section 7.6)
- o A "keyChange" resource (Section 7.3.5)

The server MUST provide "directory" and "newNonce" resources.

ACME uses different URLs for different management functions.  Each function is listed in a directory along with its corresponding URL, so clients only need to be configured with the directory URL.  These URLs are connected by a few different link relations [RFC8288].

The "up" link relation is used with challenge resources to indicate the authorization resource to which a challenge belongs.  It is also used, with some media types, from certificate resources to indicate a resource from which the client may fetch a chain of CA certificates that could be used to validate the certificate in the original resource.

The "index" link relation is present on all resources other than the directory and indicates the URL of the directory.

The following diagram illustrates the relations between resources on an ACME server.  For the most part, these relations are expressed by URLs provided as strings in the resources' JSON representations. Lines with labels in quotes indicate HTTP link relations.

```
                                  directory
                                      |
                                      +--> newNonce
                                      |
          +----------+----------+-----+-----+------------+
          |          |          |           |            |
          |          |          |           |            |
          V          V          V           V            V
     newAccount   newAuthz   newOrder   revokeCert   keyChange
          |          |          |
          |          |          |
          V          |          V
       account       |        order --+--> finalize
                     |          |     |
                     |          |     +--> cert
                     |          V
                     +---> authorization
                               | ^
                               | | "up"
                               V |
                             challenge
```

ACME Resources and Relationships

The following table illustrates a typical sequence of requests required to establish a new account with the server, prove control of an identifier, issue a certificate, and fetch an updated certificate some time after issuance. The "->" is a mnemonic for a Location header field pointing to a created resource.

| Action | Request | Response |
|---|---|---|
| Get directory | GET  directory | 200 |
| Get nonce | HEAD newNonce | 200 |
| Create account | POST newAccount | 201 -> account |
| Submit order | POST newOrder | 201 -> order |
| Fetch challenges | POST-as-GET order's authorization urls | 200 |
| Respond to challenges | POST authorization challenge urls | 200 |
| Poll for status | POST-as-GET order | 200 |
| Finalize order | POST order's finalize url | 200 |
| Poll for status | POST-as-GET order | 200 |
| Download certificate | POST-as-GET order's certificate url | 200 |

The remainder of this section provides the details of how these resources are structured and how the ACME protocol makes use of them.

### 7.1.1. Directory

In order to help clients configure themselves with the right URLs for each ACME operation, ACME servers provide a directory object. This should be the only URL needed to configure clients. It is a JSON object, whose field names are drawn from the resource registry (Section 9.7.5) and whose values are the corresponding URLs.

| Field | URL in Value |
|---|---|
| newNonce | New nonce |
| newAccount | New account |
| newOrder | New order |
| newAuthz | New authorization |
| revokeCert | Revoke certificate |
| keyChange | Key change |

There is no constraint on the URL of the directory except that it should be different from the other ACME server resources' URLs, and that it should not clash with other services. For instance:

o a host that functions as both an ACME and a Web server may want to keep the root path "/" for an HTML "front page" and place the ACME directory under the path "/acme".

o a host that only functions as an ACME server could place the directory under the path "/".

If the ACME server does not implement pre-authorization (Section 7.4.1), it MUST omit the "newAuthz" field of the directory.

The object MAY additionally contain a "meta" field. If present, it MUST be a JSON object; each field in the object is an item of metadata relating to the service provided by the ACME server.

The following metadata items are defined (Section 9.7.6), all of which are OPTIONAL:

termsOfService (optional, string): A URL identifying the current terms of service.

website (optional, string): An HTTP or HTTPS URL locating a website providing more information about the ACME server.

caaIdentities (optional, array of string): The hostnames that the ACME server recognizes as referring to itself for the purposes of CAA record validation as defined in [RFC6844]. Each string MUST represent the same sequence of ASCII code points that the server will expect to see as the "Issuer Domain Name" in a CAA issue or issuewild property tag. This allows clients to determine the correct issuer domain name to use when configuring CAA records.

externalAccountRequired (optional, boolean): If this field is present and set to "true", then the CA requires that all newAccount requests include an "externalAccountBinding" field associating the new account with an external account.

Clients access the directory by sending a GET request to the directory URL.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "newNonce": "https://example.com/acme/new-nonce",
  "newAccount": "https://example.com/acme/new-account",
  "newOrder": "https://example.com/acme/new-order",
  "newAuthz": "https://example.com/acme/new-authz",
  "revokeCert": "https://example.com/acme/revoke-cert",
  "keyChange": "https://example.com/acme/key-change",
  "meta": {
    "termsOfService": "https://example.com/acme/terms/2017-5-30",
    "website": "https://www.example.com/",
    "caaIdentities": ["example.com"],
    "externalAccountRequired": false
  }
}
```

### 7.1.2. Account Objects

An ACME account resource represents a set of metadata associated with an account. Account resources have the following structure:

status (required, string): The status of this account. Possible values are "valid", "deactivated", and "revoked". The value "deactivated" should be used to indicate client-initiated deactivation whereas "revoked" should be used to indicate server-initiated deactivation. See Section 7.1.6.

contact (optional, array of string): An array of URLs that the server can use to contact the client for issues related to this account. For example, the server may wish to notify the client about server-initiated revocation or certificate expiration. For information on supported URL schemes, see Section 7.3.

termsOfServiceAgreed (optional, boolean): Including this field in a newAccount request, with a value of true, indicates the client's agreement with the terms of service. This field cannot be updated by the client.

externalAccountBinding (optional, object): Including this field in a newAccount request indicates approval by the holder of an existing non-ACME account to bind that account to this ACME account. This field is not updateable by the client (see Section 7.3.4).

orders (required, string): A URL from which a list of orders submitted by this account can be fetched via a POST-as-GET request, as described in Section 7.1.2.1.

```
{
  "status": "valid",
  "contact": [
    "mailto:cert-admin@example.org",
    "mailto:admin@example.org"
  ],
  "termsOfServiceAgreed": true,
  "orders": "https://example.com/acme/orders/rzGoeA"
}
```

#### 7.1.2.1. Orders List

Each account object includes an "orders" URL from which a list of orders created by the account can be fetched via POST-as-GET request. The result of the request MUST be a JSON object whose "orders" field is an array of URLs, each identifying an order belonging to the account. The server SHOULD include pending orders and SHOULD NOT include orders that are invalid in the array of URLs. The server MAY return an incomplete list, along with a Link header field with a "next" link relation indicating where further entries can be acquired.

```
   HTTP/1.1 200 OK
   Content-Type: application/json
   Link: <https://example.com/acme/directory>;rel="index"
   Link: <https://example.com/acme/orders/rzGoeA?cursor=2>;rel="next"

   {
     "orders": [
       "https://example.com/acme/order/TOlocE8rfgo",
       "https://example.com/acme/order/4E16bbL5iSw",
       /* more URLs not shown for example brevity */
       "https://example.com/acme/order/neBHYLfw0mg"
     ]
   }
```

### 7.1.3. Order Objects

An ACME order object represents a client's request for a certificate and is used to track the progress of that order through to issuance. Thus, the object contains information about the requested certificate, the authorizations that the server requires the client to complete, and any certificates that have resulted from this order.

status (required, string): The status of this order. Possible values are "pending", "ready", "processing", "valid", and "invalid". See Section 7.1.6.

expires (optional, string): The timestamp after which the server will consider this order invalid, encoded in the format specified in [RFC3339]. This field is REQUIRED for objects with "pending" or "valid" in the status field.

identifiers (required, array of object): An array of identifier objects that the order pertains to.

   type (required, string): The type of identifier. This document defines the "dns" identifier type. See the registry defined in Section 9.7.7 for any others.

   value (required, string): The identifier itself.

notBefore (optional, string): The requested value of the notBefore field in the certificate, in the date format defined in [RFC3339].

notAfter (optional, string): The requested value of the notAfter field in the certificate, in the date format defined in [RFC3339].

error (optional, object): The error that occurred while processing the order, if any. This field is structured as a problem document [RFC7807].

authorizations (required, array of string): For pending orders, the authorizations that the client needs to complete before the requested certificate can be issued (see Section 7.5), including unexpired authorizations that the client has completed in the past for identifiers specified in the order. The authorizations required are dictated by server policy; there may not be a 1:1 relationship between the order identifiers and the authorizations required. For final orders (in the "valid" or "invalid" state), the authorizations that were completed. Each entry is a URL from which an authorization can be fetched with a POST-as-GET request.

finalize (required, string): A URL that a CSR must be POSTed to once all of the order's authorizations are satisfied to finalize the order. The result of a successful finalization will be the population of the certificate URL for the order.

certificate (optional, string): A URL for the certificate that has been issued in response to this order.

```
{
  "status": "valid",
  "expires": "2016-01-20T14:09:07.99Z",

  "identifiers": [
    { "type": "dns", "value": "www.example.org" },
    { "type": "dns", "value": "example.org" }
  ],

  "notBefore": "2016-01-01T00:00:00Z",
  "notAfter": "2016-01-08T00:00:00Z",

  "authorizations": [
    "https://example.com/acme/authz/PAniVnsZcis",
    "https://example.com/acme/authz/r4HqLzrSrpI"
  ],

  "finalize": "https://example.com/acme/order/TOlocE8rfgo/finalize",

  "certificate": "https://example.com/acme/cert/mAt3xBGaobw"
}
```

Any identifier of type "dns" in a newOrder request MAY have a wildcard domain name as its value. A wildcard domain name consists of a single asterisk character followed by a single full stop character ("*.") followed by a domain name as defined for use in the Subject Alternate Name Extension by [RFC5280]. An authorization returned by the server for a wildcard domain name identifier MUST NOT include the asterisk and full stop ("*.") prefix in the authorization identifier value. The returned authorization MUST include the optional "wildcard" field, with a value of true.

The elements of the "authorizations" and "identifiers" arrays are immutable once set. The server MUST NOT change the contents of either array after they are created. If a client observes a change in the contents of either array, then it SHOULD consider the order invalid.

The "authorizations" array of the order SHOULD reflect all authorizations that the CA takes into account in deciding to issue, even if some authorizations were fulfilled in earlier orders or in pre-authorization transactions. For example, if a CA allows multiple orders to be fulfilled based on a single authorization transaction, then it SHOULD reflect that authorization in all of the orders.

Note that just because an authorization URL is listed in the "authorizations" array of an order object doesn't mean that the client is required to take action. There are several reasons that the referenced authorizations may already be valid:

- The client completed the authorization as part of a previous order

- The client previously pre-authorized the identifier (see Section 7.4.1)

- The server granted the client authorization based on an external account

Clients SHOULD check the "status" field of an order to determine whether they need to take any action.

#### 7.1.4. Authorization Objects

An ACME authorization object represents a server's authorization for an account to represent an identifier. In addition to the identifier, an authorization includes several metadata fields, such as the status of the authorization (e.g., "pending", "valid", or "revoked") and which challenges were used to validate possession of the identifier.

The structure of an ACME authorization resource is as follows:

identifier (required, object): The identifier that the account is authorized to represent.

type (required, string): The type of identifier (see below and Section 9.7.7).

value (required, string): The identifier itself.

status (required, string): The status of this authorization. Possible values are "pending", "valid", "invalid", "deactivated", "expired", and "revoked". See Section 7.1.6.

expires (optional, string): The timestamp after which the server will consider this authorization invalid, encoded in the format specified in [RFC3339]. This field is REQUIRED for objects with "valid" in the "status" field.

challenges (required, array of objects): For pending authorizations, the challenges that the client can fulfill in order to prove possession of the identifier. For valid authorizations, the challenge that was validated. For invalid authorizations, the challenge that was attempted and failed. Each array entry is an object with parameters required to validate the challenge. A client should attempt to fulfill one of these challenges, and a server should consider any one of the challenges sufficient to make the authorization valid.

wildcard (optional, boolean): This field MUST be present and true for authorizations created as a result of a newOrder request containing a DNS identifier with a value that was a wildcard domain name. For other authorizations, it MUST be absent. Wildcard domain names are described in Section 7.1.3.

The only type of identifier defined by this specification is a fully qualified domain name (type: "dns"). The domain name MUST be encoded in the form in which it would appear in a certificate. That is, it MUST be encoded according to the rules in Section 7 of [RFC5280]. Servers MUST verify any identifier values that begin with the ASCII-Compatible Encoding prefix "xn--" as defined in [RFC5890] are properly encoded. Wildcard domain names (with "*" as the first label) MUST NOT be included in authorization objects. If an authorization object conveys authorization for the base domain of a newOrder DNS identifier containing a wildcard domain name, then the optional authorizations "wildcard" field MUST be present with a value of true.

Section 8 describes a set of challenges for domain name validation.

```
{
  "status": "valid",
  "expires": "2015-03-01T14:09:07.99Z",

  "identifier": {
    "type": "dns",
    "value": "www.example.org"
  },

  "challenges": [
    {
      "url": "https://example.com/acme/chall/prV_B7yEyA4",
      "type": "http-01",
      "status": "valid",
      "token": "DGyRejmCefe7v4NfDGDKfA",
      "validated": "2014-12-01T12:05:58.16Z"
    }
  ],

  "wildcard": false
}
```

### 7.1.5. Challenge Objects

An ACME challenge object represents a server's offer to validate a client's possession of an identifier in a specific way. Unlike the other objects listed above, there is not a single standard structure for a challenge object. The contents of a challenge object depend on the validation method being used. The general structure of challenge objects and an initial set of validation methods are described in Section 8.

### 7.1.6. Status Changes

Each ACME object type goes through a simple state machine over its lifetime. The "status" field of the object indicates which state the object is currently in.

Challenge objects are created in the "pending" state. They transition to the "processing" state when the client responds to the challenge (see Section 7.5.1) and the server begins attempting to validate that the client has completed the challenge. Note that within the "processing" state, the server may attempt to validate the challenge multiple times (see Section 8.2). Likewise, client requests for retries do not cause a state change. If validation is successful, the challenge moves to the "valid" state; if there is an error, the challenge moves to the "invalid" state.

```
            pending
               |
               | Receive
               | response
               V
           processing <-+
               |   |    | Server retry or
               |   |    | client retry request
               |   +----+
               |
               |
   Successful  |   Failed
   validation  |   validation
     +---------+---------+
     |                   |
     V                   V
   valid              invalid
```

State Transitions for Challenge Objects

Authorization objects are created in the "pending" state. If one of the challenges listed in the authorization transitions to the "valid" state, then the authorization also changes to the "valid" state. If the client attempts to fulfill a challenge and fails, or if there is an error while the authorization is still pending, then the authorization transitions to the "invalid" state. Once the authorization is in the "valid" state, it can expire ("expired"), be deactivated by the client ("deactivated", see Section 7.5.2), or revoked by the server ("revoked").

```
                      pending --------------------+
                         |                        |
       Challenge failure |                        |
              or         |                        |
             Error       |  Challenge valid       |
               +---------+---------+              |
               |                   |              |
               V                   V              |
            invalid              valid            |
                                   |              |
                                   |              |
                                   |              |
                    +--------------+--------------+
                    |              |              |
                    |              |              |
             Server |       Client |   Time after |
             revoke |   deactivate |    "expires" |
                    V              V              V
                 revoked      deactivated      expired
```

State Transitions for Authorization Objects

Order objects are created in the "pending" state. Once all of the authorizations listed in the order object are in the "valid" state, the order transitions to the "ready" state. The order moves to the "processing" state after the client submits a request to the order's "finalize" URL and the CA begins the issuance process for the certificate. Once the certificate is issued, the order enters the "valid" state. If an error occurs at any of these stages, the order moves to the "invalid" state. The order also moves to the "invalid" state if it expires or one of its authorizations enters a final state other than "valid" ("expired", "revoked", or "deactivated").

```
    pending --------------+
       |                  |
       | All authz        |
       | "valid"          |
       V                  |
     ready ---------------+
       |                  |
       | Receive          |
       | finalize         |
       | request          |
       V                  |
   processing ------------+
       |                  |
       | Certificate      | Error or
       | issued           | Authorization failure
       V                  V
     valid             invalid
```

State Transitions for Order Objects

Account objects are created in the "valid" state, since no further action is required to create an account after a successful newAccount request. If the account is deactivated by the client or revoked by the server, it moves to the corresponding state.

```
                     valid
                       |
                       |
           +-----------+-----------+
    Client |                Server |
   deactiv.|                revoke |
           V                       V
      deactivated               revoked
```

State Transitions for Account Objects

Note that some of these states may not ever appear in a "status" field, depending on server behavior. For example, a server that issues synchronously will never show an order in the "processing" state. A server that deletes expired authorizations immediately will never show an authorization in the "expired" state.

## 7.2. Getting a Nonce

Before sending a POST request to the server, an ACME client needs to have a fresh anti-replay nonce to put in the "nonce" header of the JWS. In most cases, the client will have gotten a nonce from a previous request. However, the client might sometimes need to get a new nonce, e.g., on its first request to the server or if an existing nonce is no longer valid.

To get a fresh nonce, the client sends a HEAD request to the newNonce resource on the server. The server's response MUST include a Replay-Nonce header field containing a fresh nonce and SHOULD have status code 200 (OK). The server MUST also respond to GET requests for this resource, returning an empty body (while still providing a Replay-Nonce header) with a status code of 204 (No Content).

```
HEAD /acme/new-nonce HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
Replay-Nonce: oFvnlFP1wIhRlYS2jTaXbA
Cache-Control: no-store
Link: <https://example.com/acme/directory>;rel="index"
```

Proxy caching of responses from the newNonce resource can cause clients to receive the same nonce repeatedly, leading to "badNonce" errors. The server MUST include a Cache-Control header field with the "no-store" directive in responses for the newNonce resource, in order to prevent caching of this resource.

## 7.3. Account Management

In this section, we describe how an ACME client can create an account on an ACME server and perform some modifications to the account after it has been created.

A client creates a new account with the server by sending a POST request to the server's newAccount URL. The body of the request is a stub account object containing some subset of the following fields:

contact (optional, array of string): Same meaning as the corresponding server field defined in Section 7.1.2.

termsOfServiceAgreed (optional, boolean): Same meaning as the corresponding server field defined in Section 7.1.2.

onlyReturnExisting (optional, boolean): If this field is present with the value "true", then the server MUST NOT create a new account if one does not already exist. This allows a client to look up an account URL based on an account key (see Section 7.3.1).

externalAccountBinding (optional, object): Same meaning as the corresponding server field defined in Section 7.1.2

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

{
  "protected": base64url({
    "alg": "ES256",
    "jwk": {...},
    "nonce": "6S8IqOGY7eL2lsGoTZYifg",
    "url": "https://example.com/acme/new-account"
  }),
  "payload": base64url({
    "termsOfServiceAgreed": true,
    "contact": [
      "mailto:cert-admin@example.org",
      "mailto:admin@example.org"
    ]
  }),
  "signature": "RZPOnYoPs1PhjszF...-nh6X1qtOFPB519I"
}
```

The server MUST ignore any values provided in the "orders" fields in account objects sent by the client, as well as any other fields that it does not recognize. If new fields are specified in the future, the specification of those fields MUST describe whether they can be provided by the client. The server MUST NOT reflect the "onlyReturnExisting" field or any unrecognized fields in the resulting account object. This allows clients to detect when servers do not support an extension field.

The server SHOULD validate that the contact URLs in the "contact" field are valid and supported by the server. If the server validates contact URLs, it MUST support the "mailto" scheme. Clients MUST NOT provide a "mailto" URL in the "contact" field that contains "hfields" [RFC6068] or more than one "addr-spec" in the "to" component. If a server encounters a "mailto" contact URL that does not meet these criteria, then it SHOULD reject it as invalid.

If the server rejects a contact URL for using an unsupported scheme, it MUST return an error of type "unsupportedContact", with a description of the error and what types of contact URLs the server considers acceptable.  If the server rejects a contact URL for using a supported scheme but an invalid value, then the server MUST return an error of type "invalidContact".

If the server wishes to require the client to agree to terms under which the ACME service is to be used, it MUST indicate the URL where such terms can be accessed in the "termsOfService" subfield of the "meta" field in the directory object, and the server MUST reject newAccount requests that do not have the "termsOfServiceAgreed" field set to "true".  Clients SHOULD NOT automatically agree to terms by default.  Rather, they SHOULD require some user interaction for agreement to terms.

The server creates an account and stores the public key used to verify the JWS (i.e., the "jwk" element of the JWS header) to authenticate future requests from the account.  The server returns this account object in a 201 (Created) response, with the account URL in a Location header field.  The account URL is used as the "kid" value in the JWS authenticating subsequent requests by this account (see Section 6.2).  The account URL is also used for requests for management actions on this account, as described below.

```
HTTP/1.1 201 Created
Content-Type: application/json
Replay-Nonce: D8s4D2mLs8Vn-goWuPQeKA
Link: <https://example.com/acme/directory>;rel="index"
Location: https://example.com/acme/acct/evOfKhNU60wg

{
  "status": "valid",

  "contact": [
    "mailto:cert-admin@example.org",
    "mailto:admin@example.org"
  ],

  "orders": "https://example.com/acme/acct/evOfKhNU60wg/orders"
}
```

### 7.3.1.  Finding an Account URL Given a Key

If the server receives a newAccount request signed with a key for which it already has an account registered with the provided account key, then it MUST return a response with status code 200 (OK) and provide the URL of that account in the Location header field.  The body of this response represents the account object as it existed on the server before this request; any fields in the request object MUST be ignored. This allows a client that has an account key but not the corresponding account URL to recover the account URL.

If a client wishes to find the URL for an existing account and does not want an account to be created if one does not already exist, then it SHOULD do so by sending a POST request to the newAccount URL with a JWS whose payload has an "onlyReturnExisting" field set to "true" ({"onlyReturnExisting": true}). If a client sends such a request and an account does not exist, then the server MUST return an error response with status code 400 (Bad Request) and type "urn:ietf:params:acme:error:accountDoesNotExist".

## 7.3.2. Account Update

If the client wishes to update this information in the future, it sends a POST request with updated information to the account URL. The server MUST ignore any updates to the "orders" field, "termsOfServiceAgreed" field (see Section 7.3.3), the "status" field (except as allowed by Section 7.3.6), or any other fields it does not recognize. If the server accepts the update, it MUST return a response with a 200 (OK) status code and the resulting account object.

For example, to update the contact information in the above account, the client could send the following request:

```
POST /acme/acct/evOfKhNU60wg HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/evOfKhNU60wg",
    "nonce": "ax5RnthDqp_Yf4_HZnFLmA",
    "url": "https://example.com/acme/acct/evOfKhNU60wg"
  }),
  "payload": base64url({
    "contact": [
      "mailto:certificates@example.org",
      "mailto:admin@example.org"
    ]
  }),
  "signature": "hDXzvcj8T6fbFbmn...rDzXzzvzpRy64N0o"
}
```

### 7.3.3. Changes of Terms of Service

As described above, a client can indicate its agreement with the CA's terms of service by setting the "termsOfServiceAgreed" field in its account object to "true".

If the server has changed its terms of service since a client initially agreed, and the server is unwilling to process a request without explicit agreement to the new terms, then it MUST return an error response with status code 403 (Forbidden) and type "urn:ietf:params:acme:error:userActionRequired". This response MUST include a Link header field with link relation "terms-of-service" and the latest terms-of-service URL.

The problem document returned with the error MUST also include an "instance" field, indicating a URL that the client should direct a human user to visit in order for instructions on how to agree to the terms.

```
 HTTP/1.1 403 Forbidden
 Replay-Nonce: T81bdZroZ2ITWSondpTmAw
 Link: <https://example.com/acme/directory>;rel="index"
 Link: <https://example.com/acme/terms/2017-6-02>;rel="terms-of-service"
 Content-Type: application/problem+json
 Content-Language: en

 {
   "type": "urn:ietf:params:acme:error:userActionRequired",
   "detail": "Terms of service have changed",
   "instance": "https://example.com/acme/agreement/?token=W8Ih3PswD-8"
 }
```

### 7.3.4. External Account Binding

The server MAY require a value for the "externalAccountBinding" field to be present in "newAccount" requests. This can be used to associate an ACME account with an existing account in a non-ACME system, such as a CA customer database.

To enable ACME account binding, the CA operating the ACME server needs to provide the ACME client with a MAC key and a key identifier, using some mechanism outside of ACME. The key identifier MUST be an ASCII string. The MAC key SHOULD be provided in base64url-encoded form, to maximize compatibility between non-ACME provisioning systems and ACME clients.

The ACME client then computes a binding JWS to indicate the external account holder's approval of the ACME account key.  The payload of this JWS is the ACME account key being registered, in JWK form.  The protected header of the JWS MUST meet the following criteria:

- The "alg" field MUST indicate a MAC-based algorithm

- The "kid" field MUST contain the key identifier provided by the CA

- The "nonce" field MUST NOT be present

- The "url" field MUST be set to the same value as the outer JWS

The "signature" field of the JWS will contain the MAC value computed with the MAC key provided by the CA.

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

{
  "protected": base64url({
    "alg": "ES256",
    "jwk": /* account key */,
    "nonce": "K60BWPrMQG9SDxBDS_xtSw",
    "url": "https://example.com/acme/new-account"
  }),
  "payload": base64url({
    "contact": [
      "mailto:cert-admin@example.org",
      "mailto:admin@example.org"
    ],
    "termsOfServiceAgreed": true,

    "externalAccountBinding": {
      "protected": base64url({
        "alg": "HS256",
        "kid": /* key identifier from CA */,
        "url": "https://example.com/acme/new-account"
      }),
      "payload": base64url(/* same as in "jwk" above */),
      "signature": /* MAC using MAC key from CA */
    }
  }),
  "signature": "5TWiqIYQfIDfALQv...x9C2mg8JGPxl5bI4"
}
```

If such a CA requires that newAccount requests contain an "externalAccountBinding" field, then it MUST provide the value "true" in the "externalAccountRequired" subfield of the "meta" field in the directory object. If the CA receives a newAccount request without an "externalAccountBinding" field, then it SHOULD reply with an error of type "externalAccountRequired".

When a CA receives a newAccount request containing an "externalAccountBinding" field, it decides whether or not to verify the binding. If the CA does not verify the binding, then it MUST NOT reflect the "externalAccountBinding" field in the resulting account object (if any). To verify the account binding, the CA MUST take the following steps:

1. Verify that the value of the field is a well-formed JWS

2. Verify that the JWS protected field meets the above criteria

3. Retrieve the MAC key corresponding to the key identifier in the "kid" field

4. Verify that the MAC on the JWS verifies using that MAC key

5. Verify that the payload of the JWS represents the same key as was used to verify the outer JWS (i.e., the "jwk" field of the outer JWS)

If all of these checks pass and the CA creates a new account, then the CA may consider the new account associated with the external account corresponding to the MAC key. The account object the CA returns MUST include an "externalAccountBinding" field with the same value as the field in the request. If any of these checks fail, then the CA MUST reject the newAccount request.

#### 7.3.5. Account Key Rollover

A client may wish to change the public key that is associated with an account in order to recover from a key compromise or proactively mitigate the impact of an unnoticed key compromise.

To change the key associated with an account, the client sends a request to the server containing signatures by both the old and new keys. The signature by the new key covers the account URL and the old key, signifying a request by the new key holder to take over the account from the old key holder. The signature by the old key covers this request and its signature, and indicates the old key holder's assent to the rollover request.

To create this request object, the client first constructs a keyChange object describing the account to be updated and its account key:

account (required, string): The URL for the account being modified. The content of this field MUST be the exact string provided in the Location header field in response to the newAccount request that created the account.

oldKey (required, JWK): The JWK representation of the old key.

The client then encapsulates the keyChange object in an "inner" JWS, signed with the requested new account key. This "inner" JWS becomes the payload for the "outer" JWS that is the body of the ACME request.

The outer JWS MUST meet the normal requirements for an ACME JWS request body (see Section 6.2). The inner JWS MUST meet the normal requirements, with the following differences:

o The inner JWS MUST have a "jwk" header parameter, containing the public key of the new key pair.

o The inner JWS MUST have the same "url" header parameter as the outer JWS.

o The inner JWS MUST omit the "nonce" header parameter.

This transaction has signatures from both the old and new keys so that the server can verify that the holders of the two keys both agree to the change. The signatures are nested to preserve the property that all signatures on POST messages are signed by exactly one key. The "inner" JWS effectively represents a request by the holder of the new key to take over the account form the holder of the old key. The "outer" JWS represents the current account holder's assent to this request.

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

   {
     "protected": base64url({
       "alg": "ES256",
       "kid": "https://example.com/acme/acct/evOfKhNU60wg",
       "nonce": "S9XaOcxP5McpnTcWPIhYuB",
       "url": "https://example.com/acme/key-change"
     }),
     "payload": base64url({
       "protected": base64url({
         "alg": "ES256",
         "jwk": /* new key */,
         "url": "https://example.com/acme/key-change"
       }),
       "payload": base64url({
         "account": "https://example.com/acme/acct/evOfKhNU60wg",
         "oldKey": /* old key */
       }),
       "signature": "Xe8B94RD30Azj2ea...8BmZIRtcSKPSd8gU"
     }),
     "signature": "5TWiqIYQfIDfALQv...x9C2mg8JGPxl5bI4"
   }
```

On receiving a keyChange request, the server MUST perform the following steps in addition to the typical JWS validation:

1. Validate the POST request belongs to a currently active account, as described in Section 6.

2. Check that the payload of the JWS is a well-formed JWS object (the "inner JWS").

3. Check that the JWS protected header of the inner JWS has a "jwk" field.

4. Check that the inner JWS verifies using the key in its "jwk" field.

5. Check that the payload of the inner JWS is a well-formed keyChange object (as described above).

6. Check that the "url" parameters of the inner and outer JWSs are the same.

7. Check that the "account" field of the keyChange object contains the URL for the account matching the old key (i.e., the "kid" field in the outer JWS).

8. Check that the "oldKey" field of the keyChange object is the same as the account key for the account in question.

9. Check that no account exists whose account key is the same as the key in the "jwk" header parameter of the inner JWS.

If all of these checks pass, then the server updates the corresponding account by replacing the old account key with the new public key and returns status code 200 (OK). Otherwise, the server responds with an error status code and a problem document describing the error. If there is an existing account with the new key provided, then the server SHOULD use status code 409 (Conflict) and provide the URL of that account in the Location header field.

Note that changing the account key for an account SHOULD NOT have any other impact on the account. For example, the server MUST NOT invalidate pending orders or authorization transactions based on a change of account key.

#### 7.3.6. Account Deactivation

A client can deactivate an account by posting a signed update to the account URL with a status field of "deactivated". Clients may wish to do this when the account key is compromised or decommissioned. A deactivated account can no longer request certificate issuance or access resources related to the account, such as orders or authorizations. If a server receives a POST or POST-as-GET from a deactivated account, it MUST return an error response with status code 401 (Unauthorized) and type "urn:ietf:params:acme:error:unauthorized".

```
POST /acme/acct/evOfKhNU60wg HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/evOfKhNU60wg",
    "nonce": "ntuJWWSic4WVNSqeUmshgg",
    "url": "https://example.com/acme/acct/evOfKhNU60wg"
  }),
  "payload": base64url({
    "status": "deactivated"
  }),
  "signature": "earzVLd3m5M4xJzR...bVTqn7R08AKOVf3Y"
}
```

The server MUST verify that the request is signed by the account key. If the server accepts the deactivation request, it replies with a 200 (OK) status code and the current contents of the account object.

Once an account is deactivated, the server MUST NOT accept further requests authorized by that account's key. The server SHOULD cancel any pending operations authorized by the account's key, such as certificate orders. A server may take a variety of actions in response to an account deactivation, e.g., deleting data related to that account or sending mail to the account's contacts. Servers SHOULD NOT revoke certificates issued by the deactivated account, since this could cause operational disruption for servers using these certificates. ACME does not provide a way to reactivate a deactivated account.

## 7.4. Applying for Certificate Issuance

The client begins the certificate issuance process by sending a POST request to the server's newOrder resource. The body of the POST is a JWS object whose JSON payload is a subset of the order object defined in Section 7.1.3, containing the fields that describe the certificate to be issued:

identifiers (required, array of object): An array of identifier objects that the client wishes to submit an order for.

   type (required, string): The type of identifier.

   value (required, string): The identifier itself.

notBefore (optional, string): The requested value of the notBefore field in the certificate, in the date format defined in [RFC3339].

notAfter (optional, string): The requested value of the notAfter field in the certificate, in the date format defined in [RFC3339].

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

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/evOfKhNU60wg",
    "nonce": "5XJ1L3lEkMG7tR6pA00clA",
    "url": "https://example.com/acme/new-order"
  }),
  "payload": base64url({
    "identifiers": [
      { "type": "dns", "value": "www.example.org" },
      { "type": "dns", "value": "example.org" }
    ],
    "notBefore": "2016-01-01T00:04:00+04:00",
    "notAfter": "2016-01-08T00:04:00+04:00"
  }),
  "signature": "H6ZXtGjTZyUnPeKn...wEA4TklBdh3e454g"
}
```

The server MUST return an error if it cannot fulfill the request as specified, and it MUST NOT issue a certificate with contents other than those requested. If the server requires the request to be modified in a certain way, it should indicate the required changes using an appropriate error type and description.

If the server is willing to issue the requested certificate, it responds with a 201 (Created) response. The body of this response is an order object reflecting the client's request and any authorizations the client must complete before the certificate will be issued.

```
   HTTP/1.1 201 Created
   Replay-Nonce: MYAuvOpaoIiywTezizk5vw
   Link: <https://example.com/acme/directory>;rel="index"
   Location: https://example.com/acme/order/TOlocE8rfgo

   {
     "status": "pending",
     "expires": "2016-01-05T14:09:07.99Z",

     "notBefore": "2016-01-01T00:00:00Z",
     "notAfter": "2016-01-08T00:00:00Z",

     "identifiers": [
       { "type": "dns", "value": "www.example.org" },
       { "type": "dns", "value": "example.org" }
     ],

     "authorizations": [
       "https://example.com/acme/authz/PAniVnsZcis",
       "https://example.com/acme/authz/r4HqLzrSrpI"
     ],

     "finalize": "https://example.com/acme/order/TOlocE8rfgo/finalize"
   }
```

The order object returned by the server represents a promise that if the client fulfills the server's requirements before the "expires" time, then the server will be willing to finalize the order upon request and issue the requested certificate.  In the order object, any authorization referenced in the "authorizations" array whose status is "pending" represents an authorization transaction that the client must complete before the server will issue the certificate (see Section 7.5).  If the client fails to complete the required actions before the "expires" time, then the server SHOULD change the status of the order to "invalid" and MAY delete the order resource. Clients MUST NOT make any assumptions about the sort order of "identifiers" or "authorizations" elements in the returned order object.

Once the client believes it has fulfilled the server's requirements, it should send a POST request to the order resource's finalize URL. The POST body MUST include a CSR:

csr (required, string): A CSR encoding the parameters for the certificate being requested [RFC2986]. The CSR is sent in the base64url-encoded version of the DER format. (Note: Because this field uses base64url, and does not include headers, it is different from PEM.)

```
POST /acme/order/TOlocE8rfgo/finalize HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/evOfKhNU60wg",
    "nonce": "MSF2j2nawWHPxxkE3ZJtKQ",
    "url": "https://example.com/acme/order/TOlocE8rfgo/finalize"
  }),
  "payload": base64url({
    "csr": "MIIBPTCBxAIBADBFMQ...FS6aKdZeGsysoCo4H9P",
  }),
  "signature": "uOrUfIIk5RyQ...nw62Ay1cl6AB"
}
```

The CSR encodes the client's requests with regard to the content of the certificate to be issued. The CSR MUST indicate the exact same set of requested identifiers as the initial newOrder request. Identifiers of type "dns" MUST appear either in the commonName portion of the requested subject name or in an extensionRequest attribute [RFC2985] requesting a subjectAltName extension, or both. (These identifiers may appear in any sort order.) Specifications that define new identifier types must specify where in the certificate signing request these identifiers can appear.

A request to finalize an order will result in an error if the CA is unwilling to issue a certificate corresponding to the submitted CSR. For example:

- If the CSR and order identifiers differ

- If the account is not authorized for the identifiers indicated in the CSR

- If the CSR requests extensions that the CA is not willing to include

In such cases, the problem document returned by the server SHOULD use error code "badCSR" and describe specific reasons the CSR was rejected in its "detail" field. After returning such an error, the server SHOULD leave the order in the "ready" state, to allow the client to submit a new finalize request with an amended CSR.

A request to finalize an order will result in error if the order is not in the "ready" state. In such cases, the server MUST return a 403 (Forbidden) error with a problem document of type "orderNotReady". The client should then send a POST-as-GET request to the order resource to obtain its current state. The status of the order will indicate what action the client should take (see below).

If a request to finalize an order is successful, the server will return a 200 (OK) with an updated order object. The status of the order will indicate what action the client should take:

- "invalid": The certificate will not be issued. Consider this order process abandoned.

- "pending": The server does not believe that the client has fulfilled the requirements. Check the "authorizations" array for entries that are still pending.

- "ready": The server agrees that the requirements have been fulfilled, and is awaiting finalization. Submit a finalization request.

- "processing": The certificate is being issued. Send a POST-as-GET request after the time given in the Retry-After header field of the response, if any.

- "valid": The server has issued the certificate and provisioned its URL to the "certificate" field of the order. Download the certificate.

```
   HTTP/1.1 200 OK
   Replay-Nonce: CGf81JWBsq8QyIgPCi9Q9X
   Link: <https://example.com/acme/directory>;rel="index"
   Location: https://example.com/acme/order/TOlocE8rfgo

   {
     "status": "valid",
     "expires": "2016-01-20T14:09:07.99Z",

     "notBefore": "2016-01-01T00:00:00Z",
     "notAfter": "2016-01-08T00:00:00Z",

     "identifiers": [
       { "type": "dns", "value": "www.example.org" },
       { "type": "dns", "value": "example.org" }
     ],

     "authorizations": [
       "https://example.com/acme/authz/PAniVnsZcis",
       "https://example.com/acme/authz/r4HqLzrSrpI"
     ],

     "finalize": "https://example.com/acme/order/TOlocE8rfgo/finalize",

     "certificate": "https://example.com/acme/cert/mAt3xBGaobw"
   }
```

### 7.4.1. Pre-authorization

The order process described above presumes that authorization objects are created reactively, in response to a certificate order. Some servers may also wish to enable clients to obtain authorization for an identifier proactively, outside of the context of a specific issuance. For example, a client hosting virtual servers for a collection of names might wish to obtain authorization before any virtual servers are created and only create a certificate when a virtual server starts up.

In some cases, a CA running an ACME server might have a completely external, non-ACME process for authorizing a client to issue certificates for an identifier. In these cases, the CA should provision its ACME server with authorization objects corresponding to these authorizations and reflect them as already valid in any orders submitted by the client.

If a CA wishes to allow pre-authorization within ACME, it can offer a "new authorization" resource in its directory by adding the field "newAuthz" with a URL for the newAuthz resource.

To request authorization for an identifier, the client sends a POST request to the newAuthz resource specifying the identifier for which authorization is being requested.

identifier (required, object): The identifier to appear in the resulting authorization object (see Section 7.1.4).

type (required, string): The type of identifier.

value (required, string): The identifier itself.

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

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/evOfKhNU60wg",
    "nonce": "uQpSjlRb4vQVCjVYAyyUWg",
    "url": "https://example.com/acme/new-authz"
  }),
  "payload": base64url({
    "identifier": {
      "type": "dns",
      "value": "example.org"
    }
  }),
  "signature": "nuSDISbWG8mMgE7H...QyVUL68yzf3Zawps"
}
```

Note that because the identifier in a pre-authorization request is the exact identifier to be included in the authorization object, pre-authorization cannot be used to authorize issuance of certificates containing wildcard domain names.

Before processing the authorization request, the server SHOULD determine whether it is willing to issue certificates for the identifier. For example, the server should check that the identifier is of a supported type. Servers might also check names against a blacklist of known high-value identifiers. If the server is unwilling to issue for the identifier, it SHOULD return an error with status code 403 (Forbidden), with a problem document describing the reason for the rejection.

If the server is willing to proceed, it builds a pending authorization object from the inputs submitted by the client:

- "identifier" the identifier submitted by the client
- "status" MUST be "pending" unless the server has out-of-band information about the client's authorization status
- "challenges" as selected by the server's policy for this identifier

The server allocates a new URL for this authorization and returns a 201 (Created) response with the authorization URL in the Location header field and the JSON authorization object in the body. The client then follows the process described in Section 7.5 to complete the authorization process.

### 7.4.2. Downloading the Certificate

To download the issued certificate, the client simply sends a POST-as-GET request to the certificate URL.

The default format of the certificate is application/pem-certificate-chain (see Section 9).

The server MAY provide one or more link relation header fields [RFC8288] with relation "alternate". Each such field SHOULD express an alternative certificate chain starting with the same end-entity certificate. This can be used to express paths to various trust anchors. Clients can fetch these alternates and use their own heuristics to decide which is optimal.

```
   POST /acme/cert/mAt3xBGaobw HTTP/1.1
   Host: example.com
   Content-Type: application/jose+json
   Accept: application/pem-certificate-chain

   {
     "protected": base64url({
       "alg": "ES256",
       "kid": "https://example.com/acme/acct/evOfKhNU60wg",
       "nonce": "uQpSjlRb4vQVCjVYAyyUWg",
       "url": "https://example.com/acme/cert/mAt3xBGaobw"
     }),
     "payload": "",
     "signature": "nuSDISbWG8mMgE7H...QyVUL68yzf3Zawps"
   }

   HTTP/1.1 200 OK
   Content-Type: application/pem-certificate-chain
   Link: <https://example.com/acme/directory>;rel="index"

   -----BEGIN CERTIFICATE-----
   [End-entity certificate contents]
   -----END CERTIFICATE-----
   -----BEGIN CERTIFICATE-----
   [Issuer certificate contents]
   -----END CERTIFICATE-----
   -----BEGIN CERTIFICATE-----
   [Other certificate contents]
   -----END CERTIFICATE-----
```

A certificate resource represents a single, immutable certificate. If the client wishes to obtain a renewed certificate, the client initiates a new order process to request one.

Because certificate resources are immutable once issuance is complete, the server MAY enable the caching of the resource by adding Expires and Cache-Control header fields specifying a point in time in the distant future.  These header fields have no relation to the certificate's period of validity.

The ACME client MAY request other formats by including an Accept header field [RFC7231] in its request.  For example, the client could use the media type "application/pkix-cert" [RFC2585] or "application/pkcs7-mime" [RFC5751] to request the end-entity certificate in DER format.  Server support for alternate formats is OPTIONAL.  For formats that can only express a single certificate, the server SHOULD provide one or more "Link: rel="up"" header fields pointing to an issuer or issuers so that ACME clients can build a certificate chain as defined in TLS (see Section 4.4.2 of [RFC8446]).

## 7.5. Identifier Authorization

The identifier authorization process establishes the authorization of an account to manage certificates for a given identifier. This process assures the server of two things:

1. That the client controls the private key of the account key pair, and

2. That the client controls the identifier in question.

This process may be repeated to associate multiple identifiers with an account (e.g., to request certificates with multiple identifiers) or to associate multiple accounts with an identifier (e.g., to allow multiple entities to manage certificates).

Authorization resources are created by the server in response to newOrder or newAuthz requests submitted by an account key holder; their URLs are provided to the client in the responses to these requests. The authorization object is implicitly tied to the account key used to sign the request.

When a client receives an order from the server in reply to a newOrder request, it downloads the authorization resources by sending POST-as-GET requests to the indicated URLs. If the client initiates authorization using a request to the newAuthz resource, it will have already received the pending authorization object in the response to that request.

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

   {
     "protected": base64url({
       "alg": "ES256",
       "kid": "https://example.com/acme/acct/evOfKhNU60wg",
       "nonce": "uQpSjlRb4vQVCjVYAyyUWg",
       "url": "https://example.com/acme/authz/PAniVnsZcis"
     }),
     "payload": "",
     "signature": "nuSDISbWG8mMgE7H...QyVUL68yzf3Zawps"
   }

   HTTP/1.1 200 OK
   Content-Type: application/json
   Link: <https://example.com/acme/directory>;rel="index"

   {
     "status": "pending",
     "expires": "2016-01-02T14:09:30Z",

     "identifier": {
       "type": "dns",
       "value": "www.example.org"
     },

     "challenges": [
       {
         "type": "http-01",
         "url": "https://example.com/acme/chall/prV_B7yEyA4",
         "token": "DGyRejmCefe7v4NfDGDKfA"
       },
       {
         "type": "dns-01",
         "url": "https://example.com/acme/chall/Rg5dV14Gh1Q",
         "token": "DGyRejmCefe7v4NfDGDKfA"
       }
     ]
   }
```

### 7.5.1.  Responding to Challenges

To prove control of the identifier and receive authorization, the client needs to provision the required challenge response based on the challenge type and indicate to the server that it is ready for the challenge validation to be attempted.

The client indicates to the server that it is ready for the challenge validation by sending an empty JSON body ("{}") carried in a POST request to the challenge URL (not the authorization URL).

For example, if the client were to respond to the "http-01" challenge in the above authorization, it would send the following request:

```
POST /acme/chall/prV_B7yEyA4 HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/evOfKhNU60wg",
    "nonce": "Q_s3MWoqT05TrdkM2MTDcw",
    "url": "https://example.com/acme/chall/prV_B7yEyA4"
  }),
  "payload": base64url({}),
  "signature": "9cbg5JO1Gf5YLjjz...SpkUfcdPai9uVYYQ"
}
```

The server updates the authorization document by updating its representation of the challenge with the response object provided by the client.  The server MUST ignore any fields in the response object that are not specified as response fields for this type of challenge. Note that the challenges in this document do not define any response fields, but future specifications might define them.  The server provides a 200 (OK) response with the updated challenge object as its body.

If the client's response is invalid for any reason or does not provide the server with appropriate information to validate the challenge, then the server MUST return an HTTP error.  On receiving such an error, the client SHOULD undo any actions that have been taken to fulfill the challenge, e.g., removing files that have been provisioned to a web server.

The server is said to "finalize" the authorization when it has completed one of the validations.  This is done by assigning the authorization a status of "valid" or "invalid", corresponding to whether it considers the account authorized for the identifier.  If the final state is "valid", then the server MUST include an "expires" field.  When finalizing an authorization, the server MAY remove challenges other than the one that was completed, and it may modify the "expires" field.  The server SHOULD NOT remove challenges with status "invalid".

Usually, the validation process will take some time, so the client will need to poll the authorization resource to see when it is finalized. For challenges where the client can tell when the server has validated the challenge (e.g., by seeing an HTTP or DNS request from the server), the client SHOULD NOT begin polling until it has seen the validation request from the server.

To check on the status of an authorization, the client sends a POST-as-GET request to the authorization URL, and the server responds with the current authorization object. In responding to poll requests while the validation is still in progress, the server MUST return a 200 (OK) response and MAY include a Retry-After header field to suggest a polling interval to the client.

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

   {
     "protected": base64url({
       "alg": "ES256",
       "kid": "https://example.com/acme/acct/evOfKhNU60wg",
       "nonce": "uQpSjlRb4vQVCjVYAyyUWg",
       "url": "https://example.com/acme/authz/PAniVnsZcis"
     }),
     "payload": "",
     "signature": "nuSDISbWG8mMgE7H...QyVUL68yzf3Zawps"
   }

   HTTP/1.1 200 OK
   Content-Type: application/json
   Link: <https://example.com/acme/directory>;rel="index"

   {
     "status": "valid",
     "expires": "2018-09-09T14:09:01.13Z",

     "identifier": {
       "type": "dns",
       "value": "www.example.org"
     },

     "challenges": [
       {
         "type": "http-01",
         "url": "https://example.com/acme/chall/prV_B7yEyA4",
         "status": "valid",
         "validated": "2014-12-01T12:05:13.72Z",
         "token": "IlirfxKKXAsHtmzK29Pj8A"
       }
     ]
   }
```

### 7.5.2.  Deactivating an Authorization

If a client wishes to relinquish its authorization to issue certificates for an identifier, then it may request that the server deactivate each authorization associated with it by sending POST requests with the static object {"status": "deactivated"} to each authorization URL.

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

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/evOfKhNU60wg",
    "nonce": "xWCM9lGbIyCgue8di6ueWQ",
    "url": "https://example.com/acme/authz/PAniVnsZcis"
  }),
  "payload": base64url({
    "status": "deactivated"
  }),
  "signature": "srX9Ji7Le9bjszhu...WTFdtujObzMtZcx4"
}
```

The server MUST verify that the request is signed by the account key corresponding to the account that owns the authorization. If the server accepts the deactivation, it should reply with a 200 (OK) status code and the updated contents of the authorization object.

The server MUST NOT treat deactivated authorization objects as sufficient for issuing certificates.

## 7.6. Certificate Revocation

To request that a certificate be revoked, the client sends a POST request to the ACME server's revokeCert URL. The body of the POST is a JWS object whose JSON payload contains the certificate to be revoked:

certificate (required, string): The certificate to be revoked, in the base64url-encoded version of the DER format. (Note: Because this field uses base64url, and does not include headers, it is different from PEM.)

reason (optional, int): One of the revocation reasonCodes defined in Section 5.3.1 of [RFC5280] to be used when generating OCSP responses and CRLs. If this field is not set, the server SHOULD omit the reasonCode CRL entry extension when generating OCSP responses and CRLs. The server MAY disallow a subset of reasonCodes from being used by the user. If a request contains a disallowed reasonCode, then the server MUST reject it with the error type "urn:ietf:params:acme:error:badRevocationReason". The problem document detail SHOULD indicate which reasonCodes are allowed.

Revocation requests are different from other ACME requests in that they can be signed with either an account key pair or the key pair in the certificate.

Example using an account key pair for the signature:

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

   {
     "protected": base64url({
       "alg": "ES256",
       "kid": "https://example.com/acme/acct/evOfKhNU60wg",
       "nonce": "JHb54aT_KTXBWQOzGYkt9A",
       "url": "https://example.com/acme/revoke-cert"
     }),
     "payload": base64url({
       "certificate": "MIIEDTCCAvegAwIBAgIRAP8...",
       "reason": 4
     }),
     "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
   }
```

Example using the certificate key pair for the signature:

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

   {
     "protected": base64url({
       "alg": "RS256",
       "jwk": /* certificate's public key */,
       "nonce": "JHb54aT_KTXBWQOzGYkt9A",
       "url": "https://example.com/acme/revoke-cert"
     }),
     "payload": base64url({
       "certificate": "MIIEDTCCAvegAwIBAgIRAP8...",
       "reason": 1
     }),
     "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
   }
```

Before revoking a certificate, the server MUST verify that the key used to sign the request is authorized to revoke the certificate. The server MUST consider at least the following accounts authorized for a given certificate:

- o the account that issued the certificate.
- o an account that holds authorizations for all of the identifiers in the certificate.

The server MUST also consider a revocation request valid if it is signed with the private key corresponding to the public key in the certificate.

If the revocation succeeds, the server responds with status code 200 (OK). If the revocation fails, the server returns an error. For example, if the certificate has already been revoked, the server returns an error response with status code 400 (Bad Request) and type "urn:ietf:params:acme:error:alreadyRevoked".

```
HTTP/1.1 200 OK
Replay-Nonce: IXVHDyxIRGcTE0VSblhPzw
Content-Length: 0
Link: <https://example.com/acme/directory>;rel="index"

--- or ---

HTTP/1.1 403 Forbidden
Replay-Nonce: lXfyFzi6238tfPQRwgfmPU
Content-Type: application/problem+json
Content-Language: en
Link: <https://example.com/acme/directory>;rel="index"

{
  "type": "urn:ietf:params:acme:error:unauthorized",
  "detail": "No authorization provided for name example.org"
}
```

## 8. Identifier Validation Challenges

There are few types of identifiers in the world for which there is a standardized mechanism to prove possession of a given identifier. In all practical cases, CAs rely on a variety of means to test whether an entity applying for a certificate with a given identifier actually controls that identifier.

Challenges provide the server with assurance that an account holder is also the entity that controls an identifier. For each type of challenge, it must be the case that, in order for an entity to successfully complete the challenge, the entity must both:

- Hold the private key of the account key pair used to respond to the challenge, and
- Control the identifier in question.

Section 10 documents how the challenges defined in this document meet these requirements. New challenges will need to document how they do.

ACME uses an extensible challenge/response framework for identifier validation. The server presents a set of challenges in the authorization object it sends to a client (as objects in the "challenges" array), and the client responds by sending a response object in a POST request to a challenge URL.

This section describes an initial set of challenge types. The definition of a challenge type includes:

1. Content of challenge objects
2. Content of response objects
3. How the server uses the challenge and response to verify control of an identifier

Challenge objects all contain the following basic fields:

type (required, string): The type of challenge encoded in the object.

url (required, string): The URL to which a response can be posted.

status (required, string): The status of this challenge. Possible values are "pending", "processing", "valid", and "invalid" (see Section 7.1.6).

validated (optional, string): The time at which the server validated this challenge, encoded in the format specified in [RFC3339]. This field is REQUIRED if the "status" field is "valid".

error (optional, object): Error that occurred while the server was validating the challenge, if any, structured as a problem document [RFC7807]. Multiple errors can be indicated by using subproblems Section 6.7.1. A challenge object with an error MUST have status equal to "invalid".

All additional fields are specified by the challenge type. If the server sets a challenge's "status" to "invalid", it SHOULD also include the "error" field to help the client diagnose why the challenge failed.

Different challenges allow the server to obtain proof of different aspects of control over an identifier. In some challenges, like HTTP and DNS, the client directly proves its ability to do certain things related to the identifier. The choice of which challenges to offer to a client under which circumstances is a matter of server policy.

The identifier validation challenges described in this section all relate to validation of domain names. If ACME is extended in the future to support other types of identifiers, there will need to be new challenge types, and they will need to specify which types of identifier they apply to.

## 8.1. Key Authorizations

All challenges defined in this document make use of a key authorization string. A key authorization is a string that concatenates the token for the challenge with a key fingerprint, separated by a "." character:

```
keyAuthorization = token || '.' || base64url(Thumbprint(accountKey))
```

The "Thumbprint" step indicates the computation specified in [RFC7638], using the SHA-256 digest [FIPS180-4]. As noted in [RFC7518] any prepended zero octets in the fields of a JWK object MUST be stripped before doing the computation.

As specified in the individual challenges below, the token for a challenge is a string comprised entirely of characters in the URL-safe base64 alphabet. The "||" operator indicates concatenation of strings.

## 8.2. Retrying Challenges

ACME challenges typically require the client to set up some network-accessible resource that the server can query in order to validate that the client controls an identifier. In practice, it is not uncommon for the server's queries to fail while a resource is being set up, e.g., due to information propagating across a cluster or firewall rules not being in place.

Clients SHOULD NOT respond to challenges until they believe that the server's queries will succeed. If a server's initial validation query fails, the server SHOULD retry the query after some time, in order to account for delay in setting up responses such as DNS records or HTTP resources. The precise retry schedule is up to the server, but server operators should keep in mind the operational scenarios that the schedule is trying to accommodate. Given that retries are intended to address things like propagation delays in HTTP or DNS provisioning, there should not usually be any reason to retry more often than every 5 or 10 seconds. While the server is still trying, the status of the challenge remains "processing"; it is only marked "invalid" once the server has given up.

The server MUST provide information about its retry state to the client via the "error" field in the challenge and the Retry-After HTTP header field in response to requests to the challenge resource. The server MUST add an entry to the "error" field in the challenge after each failed validation query. The server SHOULD set the Retry-After header field to a time after the server's next validation query, since the status of the challenge will not change until that time.

Clients can explicitly request a retry by re-sending their response to a challenge in a new POST request (with a new nonce, etc.). This allows clients to request a retry when the state has changed (e.g., after firewall rules have been updated). Servers SHOULD retry a request immediately on receiving such a POST request. In order to avoid denial-of-service attacks via client-initiated retries, servers SHOULD rate-limit such requests.

## 8.3. HTTP Challenge

With HTTP validation, the client in an ACME transaction proves its control over a domain name by proving that it can provision HTTP resources on a server accessible under that domain name. The ACME server challenges the client to provision a file at a specific path, with a specific string as its content.

As a domain may resolve to multiple IPv4 and IPv6 addresses, the server will connect to at least one of the hosts found in the DNS A and AAAA records, at its discretion.  Because many web servers allocate a default HTTPS virtual host to a particular low-privilege tenant user in a subtle and non-intuitive manner, the challenge must be completed over HTTP, not HTTPS.

type (required, string):  The string "http-01".

token (required, string):  A random value that uniquely identifies the challenge.  This value MUST have at least 128 bits of entropy. It MUST NOT contain any characters outside the base64url alphabet and MUST NOT include base64 padding characters ("=").  See [RFC4086] for additional information on randomness requirements.

```
{
  "type": "http-01",
  "url": "https://example.com/acme/chall/prV_B7yEyA4",
  "status": "pending",
  "token": "LoqXcYV8q5ONbJQxbmR7SCTNo3tiAXDfowyjxAjEuX0"
}
```

A client fulfills this challenge by constructing a key authorization from the "token" value provided in the challenge and the client's account key.  The client then provisions the key authorization as a resource on the HTTP server for the domain in question.

The path at which the resource is provisioned is comprised of the fixed prefix "/.well-known/acme-challenge/", followed by the "token" value in the challenge.  The value of the resource MUST be the ASCII representation of the key authorization.

```
GET /.well-known/acme-challenge/LoqXcYV8...jxAjEuX0
Host: example.org

HTTP/1.1 200 OK
Content-Type: application/octet-stream

LoqXcYV8...jxAjEuX0.9jg46WB3...fm21mqTI
```

(In the above, "..." indicates that the token and the JWK thumbprint in the key authorization have been truncated to fit on the page.)

A client responds with an empty object ({}) to acknowledge that the challenge can be validated by the server.

```
POST /acme/chall/prV_B7yEyA4
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/evOfKhNU60wg",
    "nonce": "UQI1PoRi5OuXzxuX7V7wL0",
    "url": "https://example.com/acme/chall/prV_B7yEyA4"
  }),
  "payload": base64url({}),
  "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
}
```

On receiving a response, the server constructs and stores the key authorization from the challenge "token" value and the current client account key.

Given a challenge/response pair, the server verifies the client's control of the domain by verifying that the resource was provisioned as expected.

1. Construct a URL by populating the URL template [RFC6570] "http://{domain}/.well-known/acme-challenge/{token}", where:

   * the domain field is set to the domain name being verified; and

   * the token field is set to the token in the challenge.

2. Verify that the resulting URL is well-formed.

3. Dereference the URL using an HTTP GET request. This request MUST be sent to TCP port 80 on the HTTP server.

4. Verify that the body of the response is a well-formed key authorization. The server SHOULD ignore whitespace characters at the end of the body.

5. Verify that key authorization provided by the HTTP server matches the key authorization stored by the server.

The server SHOULD follow redirects when dereferencing the URL. Clients might use redirects, for example, so that the response can be provided by a centralized certificate management server. See Section 10.2 for security considerations related to redirects.

If all of the above verifications succeed, then the validation is successful. If the request fails, or the body does not pass these checks, then it has failed.

The client SHOULD de-provision the resource provisioned for this challenge once the challenge is complete, i.e., once the "status" field of the challenge has the value "valid" or "invalid".

Note that because the token appears both in the request sent by the ACME server and in the key authorization in the response, it is possible to build clients that copy the token from request to response. Clients should avoid this behavior because it can lead to cross-site scripting vulnerabilities; instead, clients should be explicitly configured on a per-challenge basis. A client that does copy tokens from requests to responses MUST validate that the token in the request matches the token syntax above (e.g., that it includes only characters from the base64url alphabet).

## 8.4. DNS Challenge

When the identifier being validated is a domain name, the client can prove control of that domain by provisioning a TXT resource record containing a designated value for a specific validation domain name.

type (required, string): The string "dns-01".

token (required, string): A random value that uniquely identifies the challenge. This value MUST have at least 128 bits of entropy. It MUST NOT contain any characters outside the base64url alphabet, including padding characters ("="). See [RFC4086] for additional information on randomness requirements.

```
{
  "type": "dns-01",
  "url": "https://example.com/acme/chall/Rg5dV14Gh1Q",
  "status": "pending",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA"
}
```

A client fulfills this challenge by constructing a key authorization from the "token" value provided in the challenge and the client's account key. The client then computes the SHA-256 digest [FIPS180-4] of the key authorization.

The record provisioned to the DNS contains the base64url encoding of this digest. The client constructs the validation domain name by prepending the label "_acme-challenge" to the domain name being validated, then provisions a TXT record with the digest value under that name.  For example, if the domain name being validated is "www.example.org", then the client would provision the following DNS record:

```
_acme-challenge.www.example.org. 300 IN TXT "gfj9Xq...Rg85nM"
```

A client responds with an empty object ({}) to acknowledge that the challenge can be validated by the server.

```
POST /acme/chall/Rg5dV14Gh1Q
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/evOfKhNU60wg",
    "nonce": "SS2sSl1PtspvFZ08kNtzKd",
    "url": "https://example.com/acme/chall/Rg5dV14Gh1Q"
  }),
  "payload": base64url({}),
  "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
}
```

On receiving a response, the server constructs and stores the key authorization from the challenge "token" value and the current client account key.

To validate a DNS challenge, the server performs the following steps:

1.  Compute the SHA-256 digest [FIPS180-4] of the stored key authorization

2.  Query for TXT records for the validation domain name

3.  Verify that the contents of one of the TXT records match the digest value

If all of the above verifications succeed, then the validation is successful.  If no DNS record is found, or DNS record and response payload do not pass these checks, then the validation fails.

The client SHOULD de-provision the resource record(s) provisioned for this challenge once the challenge is complete, i.e., once the "status" field of the challenge has the value "valid" or "invalid".

## 9. IANA Considerations

### 9.1. Media Type: application/pem-certificate-chain

A file of this type contains one or more certificates encoded with the PEM textual encoding, according to [RFC7468]. The textual encoding of certificates in this file MUST use the strict encoding and MUST NOT include explanatory text. The ABNF for this format is as follows, where "stricttextualmsg" and "eol" are as defined in Section 3 of RFC 7468:

```
certchain = stricttextualmsg *(eol stricttextualmsg)
```

In order to provide easy interoperation with TLS, the first certificate MUST be an end-entity certificate. Each following certificate SHOULD directly certify the one preceding it. Because certificate validation requires that trust anchors be distributed independently, a certificate that represents a trust anchor MAY be omitted from the chain, provided that supported peers are known to possess any omitted certificates.

The following has been registered in the "Media Types" registry:
Type name: application

Subtype name: pem-certificate-chain

Required parameters: None

Optional parameters: None

Encoding considerations: 7bit

Security considerations: Carries a cryptographic certificate and its associated certificate chain. This media type carries no active content.

Interoperability considerations: None

Published specification: RFC 8555

Applications that use this media type: ACME clients and servers, HTTP servers, other applications that need to be configured with a certificate chain

Additional information:

Deprecated alias names for this type: n/a
Magic number(s): n/a

File extension(s): .pem
Macintosh file type code(s): n/a

Person & email address to contact for further information: See Authors' Addresses section.

Intended usage: COMMON

Restrictions on usage: n/a

Author: See Authors' Addresses section.

Change controller: IETF <iesg@ietf.org>

## 9.2. Well-Known URI for the HTTP Challenge

The following value has been registered in the "Well-Known URIs" registry (using the template from [RFC5785]):

URI suffix: acme-challenge

Change controller: IETF

Specification document(s): RFC 8555, Section 8.3

Related information: N/A

## 9.3. Replay-Nonce HTTP Header

The following value has been registered in the "Message Headers" registry:

| Header Field Name | Protocol | Status | Reference |
|---|---|---|---|
| Replay-Nonce | http | standard | RFC 8555, Section 6.5.1 |

## 9.4. "url" JWS Header Parameter

The following value has been registered in the "JSON Web Signature and Encryption Header Parameters" registry:

- Header Parameter Name: "url"
- Header Parameter Description: URL
- Header Parameter Usage Location(s): JWE, JWS
- Change Controller: IESG
- Specification Document(s): RFC 8555, Section 6.4.1

## 9.5. "nonce" JWS Header Parameter

The following value has been registered in the "JSON Web Signature and Encryption Header Parameters" registry:

- Header Parameter Name: "nonce"
- Header Parameter Description: Nonce
- Header Parameter Usage Location(s): JWE, JWS
- Change Controller: IESG
- Specification Document(s): RFC 8555, Section 6.5.2

## 9.6. URN Sub-namespace for ACME (urn:ietf:params:acme)

The following value has been registered in the "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry, following the template in [RFC3553]:

Registry name: acme

Specification: RFC 8555

Repository: http://www.iana.org/assignments/acme

Index value: No transformation needed.

## 9.7. New Registries

IANA has created the following registries:

1. ACME Account Object Fields (Section 9.7.1)

2. ACME Order Object Fields (Section 9.7.2)

3. ACME Authorization Object Fields (Section 9.7.3)

4. ACME Error Types (Section 9.7.4)

5. ACME Resource Types (Section 9.7.5)

6. ACME Directory Metadata Fields (Section 9.7.6)

7. ACME Identifier Types (Section 9.7.7)

8. ACME Validation Methods (Section 9.7.8)

All of these registries are under a heading of "Automated Certificate Management Environment (ACME) Protocol" and are administered under a Specification Required policy [RFC8126].

### 9.7.1. Fields in Account Objects

The "ACME Account Object Fields" registry lists field names that are defined for use in ACME account objects. Fields marked as "configurable" may be included in a newAccount request.

Template:

- Field name: The string to be used as a field name in the JSON object

- Field type: The type of value to be provided, e.g., string, boolean, array of string

- Requests: Either the value "none" or a list of types of requests where the field is allowed in a request object, taken from the following values:

  * "new" - Requests to the "newAccount" URL

  * "account" - Requests to an account URL

- Reference: Where this field is defined

Initial contents: The fields and descriptions defined in Section 7.1.2.

| Field Name | Field Type | Requests | Reference |
|---|---|---|---|
| status | string | new, account | RFC 8555 |
| contact | array of string | new, account | RFC 8555 |
| externalAccountBinding | object | new | RFC 8555 |
| termsOfServiceAgreed | boolean | new | RFC 8555 |
| onlyReturnExisting | boolean | new | RFC 8555 |
| orders | string | none | RFC 8555 |

## 9.7.2. Fields in Order Objects

The "ACME Order Object Fields" registry lists field names that are defined for use in ACME order objects. Fields marked as "configurable" may be included in a newOrder request.

Template:

- o Field name: The string to be used as a field name in the JSON object
- o Field type: The type of value to be provided, e.g., string, boolean, array of string
- o Configurable: Boolean indicating whether the server should accept values provided by the client
- o Reference: Where this field is defined

Initial contents: The fields and descriptions defined in Section 7.1.3.

| Field Name | Field Type | Configurable | Reference |
|---|---|---|---|
| status | string | false | RFC 8555 |
| expires | string | false | RFC 8555 |
| identifiers | array of object | true | RFC 8555 |
| notBefore | string | true | RFC 8555 |
| notAfter | string | true | RFC 8555 |
| error | string | false | RFC 8555 |
| authorizations | array of string | false | RFC 8555 |
| finalize | string | false | RFC 8555 |
| certificate | string | false | RFC 8555 |

### 9.7.3. Fields in Authorization Objects

The "ACME Authorization Object Fields" registry lists field names that are defined for use in ACME authorization objects. Fields marked as "configurable" may be included in a newAuthz request.

Template:

o Field name: The string to be used as a field name in the JSON object

o Field type: The type of value to be provided, e.g., string, boolean, array of string

o Configurable: Boolean indicating whether the server should accept values provided by the client

o Reference: Where this field is defined

Initial contents: The fields and descriptions defined in Section 7.1.4.

| Field Name | Field Type | Configurable | Reference |
|---|---|---|---|
| identifier | object | true | RFC 8555 |
| status | string | false | RFC 8555 |
| expires | string | false | RFC 8555 |
| challenges | array of object | false | RFC 8555 |
| wildcard | boolean | false | RFC 8555 |

### 9.7.4. Error Types

The "ACME Error Types" registry lists values that are used within URN values that are provided in the "type" field of problem documents in ACME.

Template:

o Type: The label to be included in the URN for this error, following "urn:ietf:params:acme:error:"

o Description: A human-readable description of the error

o Reference: Where the error is defined

Initial contents: The types and descriptions in the table in Section 6.7 above, with the Reference field set to point to this specification.

### 9.7.5. Resource Types

The "ACME Resource Types" registry lists the types of resources that ACME servers may list in their directory objects.

Template:

o Field name: The value to be used as a field name in the directory object

o Resource type: The type of resource labeled by the field o Reference: Where the resource type is defined

Initial contents:

| Field Name | Resource Type | Reference |
|---|---|---|
| newNonce | New nonce | RFC 8555 |
| newAccount | New account | RFC 8555 |
| newOrder | New order | RFC 8555 |
| newAuthz | New authorization | RFC 8555 |
| revokeCert | Revoke certificate | RFC 8555 |
| keyChange | Key change | RFC 8555 |
| meta | Metadata object | RFC 8555 |

### 9.7.6. Fields in the "meta" Object within a Directory Object

The "ACME Directory Metadata Fields" registry lists field names that are defined for use in the JSON object included in the "meta" field of an ACME directory object.

Template:

o Field name: The string to be used as a field name in the JSON object

o Field type: The type of value to be provided, e.g., string, boolean, array of string

o Reference: Where this field is defined

Initial contents: The fields and descriptions defined in Section 7.1.1.

| Field Name | Field Type | Reference |
|---|---|---|
| termsOfService | string | RFC 8555 |
| website | string | RFC 8555 |
| caaIdentities | array of string | RFC 8555 |
| externalAccountRequired | boolean | RFC 8555 |

### 9.7.7. Identifier Types

The "ACME Identifier Types" registry lists the types of identifiers that can be present in ACME authorization objects.

Template:

- o Label: The value to be put in the "type" field of the identifier object
- o Reference: Where the identifier type is defined

Initial contents:

| Label | Reference |
|---|---|
| dns | RFC 8555 |

### 9.7.8. Validation Methods

The "ACME Validation Methods" registry lists identifiers for the ways that CAs can validate control of identifiers. Each method's entry must specify whether it corresponds to an ACME challenge type. The "Identifier Type" field must be contained in the Label column of the "ACME Identifier Types" registry.

Template:

- o Label: The identifier for this validation method
- o Identifier Type: The type of identifier that this method applies to
- o ACME: "Y" if the validation method corresponds to an ACME challenge type; "N" otherwise
- o Reference: Where the validation method is defined

This registry may also contain reserved entries (e.g., to avoid collisions). Such entries should have the "ACME" field set to "N" and the "Identifier Type" set to "RESERVED".

Initial Contents

| Label | Identifier Type | ACME | Reference |
|---|---|---|---|
| http-01 | dns | Y | RFC 8555 |
| dns-01 | dns | Y | RFC 8555 |
| tls-sni-01 | RESERVED | N | RFC 8555 |
| tls-sni-02 | RESERVED | N | RFC 8555 |

When evaluating a request for an assignment in this registry, the designated expert should ensure that the method being registered has a clear, interoperable definition and does not overlap with existing validation methods. That is, it should not be possible for a client and server to follow the same set of actions to fulfill two different validation methods.

The values "tls-sni-01" and "tls-sni-02" are reserved because they were used in pre-RFC versions of this specification to denote validation methods that were removed because they were found not to be secure in some cases.

Validation methods do not have to be compatible with ACME in order to be registered. For example, a CA might wish to register a validation method to support its use with the ACME extensions to CAA [ACME-CAA].

## 10. Security Considerations

ACME is a protocol for managing certificates that attest to identifier/key bindings. Thus, the foremost security goal of ACME is to ensure the integrity of this process, i.e., to ensure that the bindings attested by certificates are correct and that only authorized entities can manage certificates. ACME identifies clients by their account keys, so this overall goal breaks down into two more precise goals:

1. Only an entity that controls an identifier can get an authorization for that identifier

2. Once authorized, an account key's authorizations cannot be improperly used by another account

In this section, we discuss the threat model that underlies ACME and the ways that ACME achieves these security goals within that threat model. We also discuss the denial-of-service risks that ACME servers face, and a few other miscellaneous considerations.

### 10.1. Threat Model

As a service on the Internet, ACME broadly exists within the Internet threat model [RFC3552]. In analyzing ACME, it is useful to think of an ACME server interacting with other Internet hosts along two "channels":

- An ACME channel, over which the ACME HTTPS requests are exchanged

- A validation channel, over which the ACME server performs additional requests to validate a client's control of an identifier

```
   +------------+
   |    ACME    |     ACME Channel
   |   Client   |--------------------+
   +------------+                    |
                                     V
                               +------------+
                               |    ACME    |
                               |   Server   |
                               +------------+
   +------------+                    |
   | Validation |<-------------------+
   |   Server   |  Validation Channel
   +------------+
```

Communications Channels Used by ACME

In practice, the risks to these channels are not entirely separate, but they are different in most cases. Each channel, for example, uses a different communications pattern: the ACME channel will comprise inbound HTTPS connections to the ACME server and the validation channel outbound HTTP or DNS requests.

Broadly speaking, ACME aims to be secure against active and passive attackers on any individual channel. Some vulnerabilities arise (noted below) when an attacker can exploit both the ACME channel and one of the others.

On the ACME channel, in addition to network-layer attackers, we also need to account for man-in-the-middle (MitM) attacks at the application layer and for abusive use of the protocol itself. Protection against application-layer MitM addresses potential attackers such as Content Distribution Networks (CDNs) and middleboxes with a TLS MitM function. Preventing abusive use of ACME means ensuring that an attacker with access to the validation channel can't obtain illegitimate authorization by acting as an ACME client (legitimately, in terms of the protocol).

ACME does not protect against other types of abuse by a MitM on the ACME channel. For example, such an attacker could send a bogus "badSignatureAlgorithm" error response to downgrade a client to the lowest-quality signature algorithm that the server supports. A MitM that is present on all connections (such as a CDN) can cause denial-of-service conditions in a variety of ways.

## 10.2. Integrity of Authorizations

ACME allows anyone to request challenges for an identifier by registering an account key and sending a newOrder request using that account key. The integrity of the authorization process thus depends on the identifier validation challenges to ensure that the challenge can only be completed by someone who both (1) holds the private key of the account key pair and (2) controls the identifier in question.

Validation responses need to be bound to an account key pair in order to avoid situations where a MitM on ACME HTTPS requests can switch out a legitimate domain holder's account key for one of his choosing. Such MitMs can arise, for example, if a CA uses a CDN or third-party reverse proxy in front of its ACME interface. An attack by such an MitM could have the following form:

1. Legitimate domain holder registers account key pair A

2. MitM registers account key pair B

3. Legitimate domain holder sends a newOrder request signed using account key A

4. MitM suppresses the legitimate request but sends the same request signed using account key B

5. ACME server issues challenges and MitM forwards them to the legitimate domain holder

6. Legitimate domain holder provisions the validation response

7. ACME server performs validation query and sees the response provisioned by the legitimate domain holder

8. Because the challenges were issued in response to a message signed with account key B, the ACME server grants authorization to account key B (the MitM) instead of account key A (the legitimate domain holder)

```
   Domain                                         ACME
   Holder                  MitM                  Server
     |                      |                      |
     | newAccount(A)        |                      |
     |--------------------->|--------------------->|
     |                      |                      |
     |                      | newAccount(B)        |
     |                      |--------------------->|
     | newOrder(domain, A)  |                      |
     |--------------------->|                      |
     |                      | newOrder(domain, B)  |
     |                      |--------------------->|
     |                      |                      |
     |   authz, challenges  |   authz, challenges  |
     |<---------------------|<---------------------|
     |                      |                      |
     | response(chall, A)   | response(chall, B)   |
     |--------------------->|--------------------->|
     |                      |                      |
     |  validation request  |                      |
     |<--------------------------------------------|
     |                      |                      |
     | validation response  |                      |
     |-------------------------------------------->|
     |                      |                      |
     |                      |                      | Considers challenge
     |                      |                      | fulfilled by B
     |                      |                      |
```

Man-in-the-Middle Attack Exploiting a Validation
Method without Account Key Binding

All of the challenges defined in this document have a binding between the account private key and the validation query made by the server, via the key authorization. The key authorization reflects the account public key and is provided to the server in the validation response over the validation channel.

The association of challenges to identifiers is typically done by requiring the client to perform some action that only someone who effectively controls the identifier can perform. For the challenges in this document, the actions are as follows:

o  HTTP: Provision files under .well-known on a web server for the domain

o  DNS: Provision DNS resource records for the domain

There are several ways that these assumptions can be violated, both by misconfiguration and by attacks. For example, on a web server that allows non-administrative users to write to .well-known, any user can claim to own the web server's hostname by responding to an HTTP challenge. Similarly, if a server that can be used for ACME validation is compromised by a malicious actor, then that malicious actor can use that access to obtain certificates via ACME.

The use of hosting providers is a particular risk for ACME validation. If the owner of the domain has outsourced operation of DNS or web services to a hosting provider, there is nothing that can be done against tampering by the hosting provider. As far as the outside world is concerned, the zone or website provided by the hosting provider is the real thing.

More limited forms of delegation can also lead to an unintended party gaining the ability to successfully complete a validation transaction. For example, suppose an ACME server follows HTTP redirects in HTTP validation and a website operator provisions a catch-all redirect rule that redirects requests for unknown resources to a different domain. Then the target of the redirect could use that to get a certificate through HTTP validation since the validation path will not be known to the primary server.

The DNS is a common point of vulnerability for all of these challenges. An entity that can provision false DNS records for a domain can attack the DNS challenge directly and can provision false A/AAAA records to direct the ACME server to send its HTTP validation query to a remote server of the attacker's choosing. There are a few different mitigations that ACME servers can apply:

- Always querying the DNS using a DNSSEC-validating resolver (enhancing security for zones that are DNSSEC-enabled)

- Querying the DNS from multiple vantage points to address local attackers

- Applying mitigations against DNS off-path attackers, e.g., adding entropy to requests [DNS0x20] or only using TCP

Given these considerations, the ACME validation process makes it impossible for any attacker on the ACME channel or a passive attacker on the validation channel to hijack the authorization process to authorize a key of the attacker's choice.

An attacker that can only see the ACME channel would need to convince the validation server to provide a response that would authorize the attacker's account key, but this is prevented by binding the validation response to the account key used to request challenges. A passive attacker on the validation channel can observe the correct validation response and even replay it, but that response can only be used with the account key for which it was generated.

An active attacker on the validation channel can subvert the ACME process, by performing normal ACME transactions and providing a validation response for his own account key. The risks due to hosting providers noted above are a particular case.

Attackers can also exploit vulnerabilities in Internet routing protocols to gain access to the validation channel (see, e.g., [RFC7132]). In order to make such attacks more difficult, it is RECOMMENDED that the server perform DNS queries and make HTTP connections from multiple points in the network. Since routing attacks are often localized or dependent on the position of the attacker, forcing the attacker to attack multiple points (the server's validation vantage points) or a specific point (the DNS / HTTP server) makes it more difficult to subvert ACME validation using attacks on routing.

## 10.3. Denial-of-Service Considerations

As a protocol run over HTTPS, standard considerations for TCP-based and HTTP-based DoS mitigation also apply to ACME.

At the application layer, ACME requires the server to perform a few potentially expensive operations. Identifier validation transactions require the ACME server to make outbound connections to potentially attacker-controlled servers, and certificate issuance can require interactions with cryptographic hardware.

In addition, an attacker can also cause the ACME server to send validation requests to a domain of its choosing by submitting authorization requests for the victim domain.

All of these attacks can be mitigated by the application of appropriate rate limits. Issues closer to the front end, like POST body validation, can be addressed using HTTP request limiting. For validation and certificate requests, there are other identifiers on which rate limits can be keyed. For example, the server might limit the rate at which any individual account key can issue certificates or the rate at which validation can be requested within a given subtree of the DNS. And in order to prevent attackers from circumventing these limits simply by minting new accounts, servers would need to limit the rate at which accounts can be registered.

## 10.4. Server-Side Request Forgery

Server-Side Request Forgery (SSRF) attacks can arise when an attacker can cause a server to perform HTTP requests to an attacker-chosen URL. In the ACME HTTP challenge validation process, the ACME server performs an HTTP GET request to a URL in which the attacker can choose the domain. This request is made before the server has verified that the client controls the domain, so any client can cause a query to any domain.

Some ACME server implementations include information from the validation server's response (in order to facilitate debugging). Such implementations enable an attacker to extract this information from any web server that is accessible to the ACME server, even if it is not accessible to the ACME client. For example, the ACME server might be able to access servers behind a firewall that would prevent access by the ACME client.

It might seem that the risk of SSRF through this channel is limited by the fact that the attacker can only control the domain of the URL, not the path. However, if the attacker first sets the domain to one they control, then they can send the server an HTTP redirect (e.g., a 302 response) which will cause the server to query an arbitrary URL.

In order to further limit the SSRF risk, ACME server operators should ensure that validation queries can only be sent to servers on the public Internet, and not, say, web services within the server operator's internal network. Since the attacker could make requests to these public servers himself, he can't gain anything extra through an SSRF attack on ACME aside from a layer of anonymization.

## 10.5. CA Policy Considerations

The controls on issuance enabled by ACME are focused on validating that a certificate applicant controls the identifier he claims. Before issuing a certificate, however, there are many other checks that a CA might need to perform, for example:

- Has the client agreed to a subscriber agreement?

- Is the claimed identifier syntactically valid?

- For domain names:

  - If the leftmost label is a '*', then have the appropriate checks been applied?

  - Is the name on the Public Suffix List?

   * Is the name a high-value name?

   * Is the name a known phishing domain?

- Is the key in the CSR sufficiently strong?

- Is the CSR signed with an acceptable algorithm?

- Has issuance been authorized or forbidden by a Certification Authority Authorization (CAA) record ([RFC6844])?

CAs that use ACME to automate issuance will need to ensure that their servers perform all necessary checks before issuing.

CAs using ACME to allow clients to agree to terms of service should keep in mind that ACME clients can automate this agreement, possibly not involving a human user.

ACME does not specify how the server constructs the URLs that it uses to address resources. If the server operator uses URLs that are predictable to third parties, this can leak information about what URLs exist on the server, since an attacker can probe for whether a POST-as-GET request to the URL returns 404 (Not Found) or 401 (Unauthorized).

For example, suppose that the CA uses highly structured URLs with guessable fields:

- Accounts: https://example.com/:accountID

- Orders: https://example.com/:accountID/:domainName

- Authorizations: https://example.com/:accountID/:domainName

- Certificates: https://example.com/:accountID/:domainName

Under that scheme, an attacker could probe for which domain names are associated with which accounts, which may allow correlation of ownership between domain names, if the CA does not otherwise permit it.

To avoid leaking these correlations, CAs SHOULD assign URLs with an unpredictable component. For example, a CA might assign URLs for each resource type from an independent namespace, using unpredictable IDs for each resource:

- Accounts: https://example.com/acct/:accountID
- Orders: https://example.com/order/:orderID
- Authorizations: https://example.com/authz/:authorizationID
- Certificates: https://example.com/cert/:certID

Such a scheme would leak only the type of resource, hiding the additional correlations revealed in the example above.

## 11. Operational Considerations

There are certain factors that arise in operational reality that operators of ACME-based CAs will need to keep in mind when configuring their services. See the subsections below for examples.

### 11.1. Key Selection

ACME relies on two different classes of key pair:

- Account key pairs, which are used to authenticate account holders
- Certificate key pairs, which are used to sign and verify CSRs (and whose public keys are included in certificates)

Compromise of the private key of an account key pair has more serious consequences than compromise of a private key corresponding to a certificate. While the compromise of a certificate key pair allows the attacker to impersonate the entities named in the certificate for the lifetime of the certificate, the compromise of an account key pair allows the attacker to take full control of the victim's ACME account and take any action that the legitimate account holder could take within the scope of ACME:

1. Issuing certificates using existing authorizations
2. Revoking existing certificates
3. Accessing and changing account information (e.g., contacts)
4. Changing the account key pair for the account, locking out the legitimate account holder

For this reason, it is RECOMMENDED that each account key pair be used only for authentication of a single ACME account. For example, the public key of an account key pair MUST NOT be included in a certificate. If an ACME client receives a request from a user for account creation or key rollover using an account key that the client knows to be used elsewhere, then the client MUST return an error. Clients MUST generate a fresh account key for every account creation or rollover operation. Note that given the requirements of Section 7.3.1, servers will not create accounts with reused keys anyway.

ACME clients and servers MUST verify that a CSR submitted in a finalize request does not contain a public key for any known account key pair. In particular, when a server receives a finalize request, it MUST verify that the public key in a CSR is not the same as the public key of the account key pair used to authenticate that request. This assures that vulnerabilities in the protocols with which the certificate is used (e.g., signing oracles in TLS [JSS15]) do not result in compromise of the ACME account. Because ACME accounts are uniquely identified by their account key pair (see Section 7.3.1), the server MUST not allow account key pair reuse across multiple accounts.

## 11.2. DNS Security

As noted above, DNS forgery attacks against the ACME server can result in the server making incorrect decisions about domain control and thus mis-issuing certificates. Servers SHOULD perform DNS queries over TCP, which provides better resistance to some forgery attacks than DNS over UDP.

An ACME-based CA will often need to make DNS queries, e.g., to validate control of DNS names. Because the security of such validations ultimately depends on the authenticity of DNS data, every possible precaution should be taken to secure DNS queries done by the CA. Therefore, it is RECOMMENDED that ACME-based CAs make all DNS queries via DNSSEC-validating stub or recursive resolvers. This provides additional protection to domains that choose to make use of DNSSEC.

An ACME-based CA must only use a resolver if it trusts the resolver and every component of the network route by which it is accessed. Therefore, it is RECOMMENDED that ACME-based CAs operate their own DNSSEC-validating resolvers within their trusted network and use these resolvers both for CAA record lookups and all record lookups in furtherance of a challenge scheme (A, AAAA, TXT, etc.).

### 11.3. Token Entropy

The http-01 and dns-01 validation methods mandate the use of a random token value to uniquely identify the challenge. The value of the token is required to contain at least 128 bits of entropy for the following security properties. First, the ACME client should not be able to influence the ACME server's choice of token as this may allow an attacker to reuse a domain owner's previous challenge responses for a new validation request. Second, the entropy requirement makes it more difficult for ACME clients to implement a "naive" validation server that automatically replies to challenges without being configured per challenge.

### 11.4. Malformed Certificate Chains

ACME provides certificate chains in the widely used format known colloquially as PEM (though it may diverge from the actual Privacy Enhanced Mail specification [RFC1421], as noted in [RFC7468]). Some current software will allow the configuration of a private key and a certificate in one PEM file by concatenating the textual encodings of the two objects. In the context of ACME, such software might be vulnerable to key replacement attacks. A malicious ACME server could cause a client to use a private key of its choosing by including the key in the PEM file returned in response to a query for a certificate URL.

When processing a file of type "application/pem-certificate-chain", a client SHOULD verify that the file contains only encoded certificates. If anything other than a certificate is found (i.e., if the string "-----BEGIN" is ever followed by anything other than "CERTIFICATE"), then the client MUST reject the file as invalid.

## 12. References

### 12.1. Normative References

[FIPS180-4]
National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", FIPS PUB 180-4, DOI 10.6028/NIST.FIPS.180-4, August 2015, <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>.

[JSS15] Somorovsky, J., Schwenk, J., and J. Somorovsky, "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption", CSS '15 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security Pages 1185-1196, DOI 10.1145/2810103.2813657, <https://dl.acm.org/citation.cfm?id=2813657>.

[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>.

[RFC2585] Housley, R. and P. Hoffman, "Internet X.509 Public Key Infrastructure Operational Protocols: FTP and HTTP", RFC 2585, DOI 10.17487/RFC2585, May 1999, <https://www.rfc-editor.org/info/rfc2585>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/info/rfc2818>.

[RFC2985] Nystrom, M. and B. Kaliski, "PKCS #9: Selected Object Classes and Attribute Types Version 2.0", RFC 2985, DOI 10.17487/RFC2985, November 2000, <https://www.rfc-editor.org/info/rfc2985>.

[RFC2986] Nystrom, M. and B. Kaliski, "PKCS #10: Certification Request Syntax Specification Version 1.7", RFC 2986, DOI 10.17487/RFC2986, November 2000, <https://www.rfc-editor.org/info/rfc2986>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[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>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, DOI 10.17487/RFC5751, January 2010, <https://www.rfc-editor.org/info/rfc5751>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[RFC6068] Duerst, M., Masinter, L., and J. Zawinski, "The 'mailto' URI Scheme", RFC 6068, DOI 10.17487/RFC6068, October 2010, <https://www.rfc-editor.org/info/rfc6068>.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <https://www.rfc-editor.org/info/rfc6570>.

[RFC6844] Hallam-Baker, P. and R. Stradling, "DNS Certification Authority Authorization (CAA) Resource Record", RFC 6844, DOI 10.17487/RFC6844, January 2013, <https://www.rfc-editor.org/info/rfc6844>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <https://www.rfc-editor.org/info/rfc7468>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

[RFC7638] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <https://www.rfc-editor.org/info/rfc7638>.

[RFC7797] Jones, M., "JSON Web Signature (JWS) Unencoded Payload Option", RFC 7797, DOI 10.17487/RFC7797, February 2016, <https://www.rfc-editor.org/info/rfc7797>.

[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>.

[RFC8037] Liusvaara, I., "CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE)", RFC 8037, DOI 10.17487/RFC8037, January 2017, <https://www.rfc-editor.org/info/rfc8037>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[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>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc8288>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

### 12.2. Informative References

[ACME-CAA]
Landau, H., "CAA Record Extensions for Account URI and ACME Method Binding", Work in Progress, draft-ietf-acme-caa-06, January 2019.

[ACME-IP] Shoemaker, R., "ACME IP Identifier Validation Extension", Work in Progress, draft-ietf-acme-ip-05, February 2019.

[ACME-TELEPHONE]
Peterson, J. and R. Barnes, "ACME Identifiers and Challenges for Telephone Numbers", Work in Progress, draft-ietf-acme-telephone-01, October 2017.

[CABFBR] CA/Browser Forum, "CA/Browser Forum Baseline Requirements", September 2018, <https://cabforum.org/baseline-requirements-documents/>.

[DNS0x20] Vixie, P. and D. Dagon, "Use of Bit 0x20 in DNS Labels to Improve Transaction Identity", Work in Progress, draft-vixie-dnsext-dns0x20-00, March 2008.

[RFC1421] Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures", RFC 1421, DOI 10.17487/RFC1421, February 1993, <https://www.rfc-editor.org/info/rfc1421>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, DOI 10.17487/RFC3553, June 2003, <https://www.rfc-editor.org/info/rfc3553>.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <https://www.rfc-editor.org/info/rfc5785>.

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/info/rfc6960>.

[RFC7132] Kent, S. and A. Chi, "Threat Model for BGP Path Security", RFC 7132, DOI 10.17487/RFC7132, February 2014, <https://www.rfc-editor.org/info/rfc7132>.

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <https://www.rfc-editor.org/info/rfc7525>.

[W3C.REC-cors-20140116] Kesteren, A., Ed., "Cross-Origin Resource Sharing", W3C Recommendation REC-cors-20140116, January 2014, <http://www.w3.org/TR/2014/REC-cors-20140116>.

## Acknowledgements

In addition to the editors listed on the front page, this document has benefited from contributions from a broad set of contributors, all the way back to its inception.

- Andrew Ayer, SSLMate
- Karthik Bhargavan, INRIA
- Peter Eckersley, EFF
- Alex Halderman, University of Michigan
- Sophie Herold, Hemio
- Tim Hollebeek, DigiCert
- Eric Rescorla, Mozilla
- Seth Schoen, EFF
- Roland Shoemaker, Let's Encrypt
- Rob Stradling, Sectigo
- Martin Thomson, Mozilla
- Jakub Warmuz, University of Oxford

This document draws on many concepts established by Eric Rescorla's "Automated Certificate Issuance Protocol" draft. Martin Thomson provided helpful guidance in the use of HTTP.

## Authors' Addresses

Richard Barnes
Cisco

Email: rlb@ipv.sx

Jacob Hoffman-Andrews
EFF

Email: jsha@eff.org

Daniel McCarney
Let's Encrypt

Email: cpu@letsencrypt.org

James Kasten
University of Michigan

Email: jdkasten@umich.edu