Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 22, 2016

R. Barnes
Mozilla
J. Hoffman-Andrews
EFF
J. Kasten
University of Michigan

March 21, 2016

Automatic Certificate Management Environment (ACME)
draft-ietf-acme-acme-02

Abstract

Certificates in the Web's X.509 PKI (PKIX) are used for a number of purposes, the most significant of which is the authentication of domain names. Thus, certificate authorities in the Web PKI are trusted to verify that an applicant for a certificate legitimately represents the domain name(s) in the certificate. Today, this verification is done through a collection of ad hoc mechanisms. This document describes a protocol that a certificate authority (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.

DISCLAIMER: This is a work in progress draft of ACME and has not yet had a thorough security analysis.

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH: The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/ietf-wg-acme/acme . Instructions are on that page as well. Editorial changes can be managed in GitHub, but any substantive change should be discussed on the ACME mailing list (acme@ietf.org).

time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 22, 2016.

Table of Contents

1. Introduction 3
2. Deployment Model and Operator Experience 4
3. Terminology 6
4. Protocol Overview 6
5. Message Transport 9
5.1. HTTPS Requests 9
5.2. Request Authentication 9
5.3. Request URI Type Integrity 10
5.4. Replay protection 11
5.4.1. Replay-Nonce 12
5.4.2. "nonce" (Nonce) JWS header parameter 12
5.5. Errors 12
6. Certificate Management 14
6.1. Resources 14
6.1.1. Registration Objects 16
6.1.2. Authorization Objects 17
6.2. Directory 18
6.3. Registration 20
6.3.1. Account Key Roll-over 22
6.3.2. Deleting an Account 23
6.4. Identifier Authorization 24
6.4.1. Responding to Challenges 26
6.4.2. Deleting an Authorization 28
6.5. Certificate Issuance 29
6.6. Certificate Revocation 32
7. Identifier Validation Challenges 33

7.1. Key Authorizations 35
7.2. HTTP 35
7.3. TLS with Server Name Indication (TLS SNI) 38
7.4. DNS 39
8. IANA Considerations 41
9. Well-Known URI for the HTTP Challenge 41
9.1. Replay-Nonce HTTP Header 41
9.2. "nonce" JWS Header Parameter 41
9.3. URN Sub-namespace for ACME (urn:ietf:params:acme) 42
9.4. New Registries 42
9.4.1. Error Codes 42
9.4.2. Identifier Types 43
9.4.3. Challenge Types 43
10. Security Considerations 44
10.1. Threat model 44
10.2. Integrity of Authorizations 45
10.3. Denial-of-Service Considerations 48
10.4. CA Policy Considerations 49
11. Operational Considerations 49
11.1. Default Virtual Hosts 49
11.2. Use of DNSSEC Resolvers 50
12. Acknowledgements 50
13. References 51
13.1. Normative References 51
13.2. Informative References 53
Authors' Addresses 54

1. Introduction

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

Existing Web PKI certificate authorities tend to run on a set of ad hoc protocols for certificate issuance and identity verification. A typical user experience is something like:

- Generate a PKCS#10 [RFC2314] Certificate Signing Request (CSR).
- Cut-and-paste the CSR into a CA web page.
- Prove ownership of the domain by one of the following methods:
  - Put a CA-provided challenge at a specific place on the web server.

    * Put a CA-provided challenge at a DNS location corresponding to the target domain.

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

- Download the issued certificate and install it on their 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 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 infrastructural software to obtain certificates without user interaction. Use of this protocol should radically simplify the deployment of HTTPS and the practicality of PKIX authentication for other protocols based on TLS [RFC5246].

2. Deployment Model and Operator Experience

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

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. For DV validation, the CA merely verifies that the requester has effective control of the web server and/or DNS server for the domain, but does not explicitly attempt to verify their 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.)

DV certificate validation commonly checks claims about properties related to control of a domain name - properties that can be observed by the issuing authority 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.

When deploying a current HTTPS server, an operator generally gets a prompt to generate a self-signed certificate. When an operator deploys an ACME-compatible web server, the experience would be something like this:

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

- 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.

- The operator selects a CA.

- In the background, the ACME client contacts the CA and requests that a certificate be issued for the intended domain name(s).

- Once the CA is satisfied, the certificate is issued and the ACME client automatically downloads and installs it, potentially notifying the operator via e-mail, SMS, etc.

- The ACME client periodically contacts the CA to get updated certificates, stapled OCSP responses, or whatever else would be required to keep the server functional and its credentials up-to-date.

The overall idea is that it's nearly as easy to deploy with a CA-issued certificate as a self-signed certificate, and that once the operator has done so, the process is self-sustaining with minimal manual intervention. Close integration of ACME with HTTPS servers, for example, can allow the immediate and automated deployment of certificates as they are issued, optionally sparing the human administrator from additional configuration work.

3. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

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 therefore typically runs on a web server, mail server, or some other server system which requires valid TLS certificates. The ACME server runs at a certificate authority, and responds to client requests, performing the requested actions if the client is authorized.

An ACME client is represented by 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 JSON messages carried over HTTPS. In some ways, ACME functions much like a traditional CA, in which a user creates an account, adds identifiers to that account (proving control of the domains), and requests certificate issuance for those domains while logged in to the account.

In ACME, the account is represented by an account key pair. The "add a domain" function is accomplished by authorizing the key pair for a given domain. Certificate issuance and revocation are authorized by a signature with the key pair.

The first phase of ACME is for the client to register with the ACME server. The client generates an asymmetric key pair and associates this key pair with a set of contact information by signing the contact information. The server acknowledges the registration by replying with a registration object echoing the client's input.

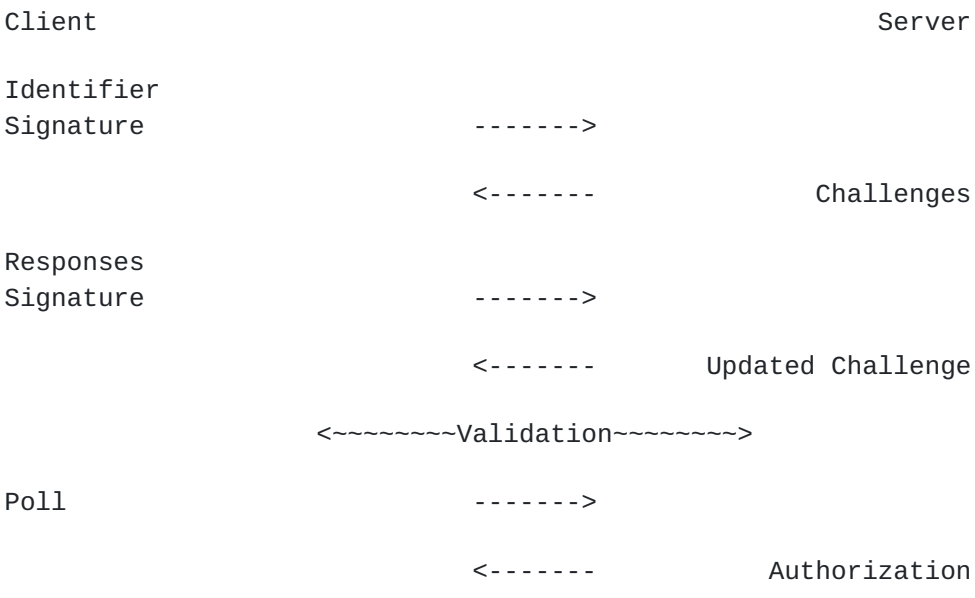
Before a client can issue certificates, it must establish an authorization with the server for an account key pair to act for the

identifier(s) that it wishes to include in the certificate. To do this, the client must demonstrate to the server both (1) that it holds the private key of the account key pair, and (2) that it has authority over the identifier being claimed.

Proof of possession of the account key is built into the ACME protocol. All messages from the client to the server are signed by the client, and the server verifies them using the public key of the account key pair.

To verify that the client controls the identifier being claimed, the server issues the client a set of challenges. 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 the challenges to check that the client has accomplished the challenge.

For example, if the client requests a domain name, the server might challenge the client to provision a record in the DNS under that name, or to provision a file on a web server referenced by an A or AAAA record under that name. The server would then query the DNS for the record in question, or send an HTTP request for the file. If the client provisioned the DNS or the web server as expected, then the server considers the client authorized for the domain name.

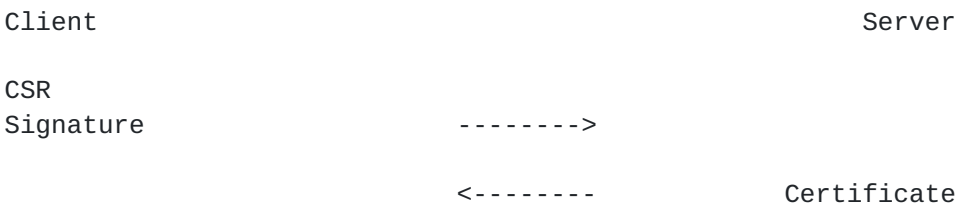


Once the client has authorized an account key pair for an identifier, it can use the key pair to authorize the issuance of certificates for

the identifier. To do this, the client sends a PKCS#10 Certificate Signing Request (CSR) to the server (indicating the identifier(s) to be included in the issued certificate) and a signature over the CSR by the private key of the account key pair.

Note that as a result, the CSR is signed twice: One by the private key corresponding to the public key in the CSR, and once by the private key of the account key pair. The former signature indicates that the holder of the key in the CSR is willing to act for the indicated identifiers, and the latter signature indicates to the server that the issuance of the certificate is authorized by the client (i.e., the domain holder).

If the server agrees to issue the certificate, then it creates the certificate and provides it in its response. The certificate is assigned a URI, which the client can use to fetch updated versions of the certificate.



To revoke a certificate, the client simply sends a revocation request indicating the certificate to be revoked, signed with an authorized key pair. The server indicates whether the request has succeeded.

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 7 below address validation of domain names. The use of ACME for other protocols 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. Message Transport

ACME uses a combination of HTTPS and JWS to create a messaging layer with a few important security properties.

Communications between an ACME client and an ACME server are done over HTTPS, using JWS to provide som 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 a degree of integrity for the HTTPS request URI.

5.1. HTTPS Requests

Each ACME function is accomplished by the client sending a sequence of HTTPS requests to the server, carrying JSON messages [RFC2818][RFC7159]. Use of HTTPS is REQUIRED. Clients SHOULD support HTTP public key pinning [RFC7469], and servers SHOULD emit pinning headers. Each subsection of Section 6 below describes the message formats used by the function, and the order in which messages are sent.

In all HTTPS transactions used by ACME, the ACME client is the HTTPS client and the ACME server is the HTTPS server.

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.CR-cors-20130129]. 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 [RFC4648] Section 5, according to the profile specified in JSON Web Signature [RFC7515] Section 2. This encoding uses a URL safe character set. Trailing '=' characters MUST be stripped.

5.2. Request Authentication

All ACME requests with a non-empty body MUST encapsulate the body in a JWS object, signed using the account key pair. The server MUST verify the JWS before processing the request. (For readability, however, the examples below omit this encapsulation.) Encapsulating request bodies in JWS provides a simple authentication of requests by way of key continuity.

JWS objects sent in ACME requests MUST meet the following additional criteria:

- The JWS MUST use the Flattened JSON Serialization
- The JWS MUST be encoded using UTF-8
- The JWS Header or Protected Header MUST include "alg" and "jwk" fields
- The JWS MUST NOT have the value "none" in its "alg" field
- The JWS Protected Header MUST include the "nonce" field (defined below)

Note that this implies that GET requests are not authenticated. Servers MUST NOT respond to GET requests for resources that might be considered sensitive.

5.3. Request URI Type Integrity

It is common in deployment 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 protection.

These intermediaries can also change values in the request that are not signed in the HTTPS request, e.g., the request URI and headers. ACME uses JWS to provides a limited integrity mechanism, which protects against an intermediary changing the request URI to anothe ACME URI of a different type. (It does not protect against changing between URIs of the same type, e.g., from one authorization URI to another).

An ACME request carries a JSON dictionary that provides the details of the client's request to the server. Each request object MUST have a "resource" field that indicates what type of resource the request is addressed to, as defined in the below table:

Resource type	"resource" value
New registration	new-reg
New authorization	new-authz
New certificate	new-cert
Revoke certificate	revoke-cert
Registration	reg
Authorization	authz
Challenge	challenge
Certificate	cert

Other fields in ACME request bodies are described below.

5.4. Replay protection

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

An ACME server MUST include a Replay-Nonce header field in each successful response it provides to a client, with contents as specified below. In particular, the ACME server MUST provide a Replay-Nonce header field in response to a HEAD request for any valid resource. (This allows clients to easily obtain a fresh nonce.) It MAY also provide nonces in error responses.

Every JWS sent by an ACME client MUST include, in its protected header, the "nonce" header parameter, with contents as defined below. 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 SHOULD provide HTTP status code 400

(Bad Request), and indicate the ACME error code "urn:ietf:params:acme:error:badNonce".

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.

5.4.1. Replay-Nonce

The "Replay-Nonce" header field includes a server-generated value that the server can use to detect unauthorized replay in future client requests. The server should generate the value provided in Replay-Nonce in such a way that they are unique to each message, with high probability.

The value of the Replay-Nonce 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.

```
base64url = [A-Z] / [a-z] / [0-9] / "-" / "_"

Replay-Nonce = *base64url
```

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

5.4.2. "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.

5.5. Errors

Errors can be reported in ACME both at the HTTP layer and within ACME payloads. 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 problem document [I-D.ietf-appsawg-http-problem]. To facilitate automatic response to errors, this document defines the following standard tokens for use in the "type" field (within the "urn:ietf:params:acme:error:" namespace):

Code	Description
badCSR	The CSR is unacceptable (e.g., due to a short key)
badNonce	The client sent an unacceptable anti-replay nonce
connection	The server could not connect to the client for validation
dnssec	The server could not validate a DNSSEC signed domain
malformed	The request message was malformed
serverInternal	The server experienced an internal error
tls	The server experienced a TLS error during validation
unauthorized	The client lacks sufficient authorization
unknownHost	The server could not resolve a domain name
rateLimited	The request exceeds a rate limit
invalidContact	The provided contact URI for a registration was invalid

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 other than the standard types. Clients SHOULD display the "detail" field of such errors.

Authorization and challenge objects can also contain error information to indicate why the server was unable to validate authorization.

6. Certificate Management

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

- Account Key Registration
- Account Key Authorization
- Certificate Issuance
- Certificate Renewal
- Certificate Revocation

6.1. Resources

ACME is structured as a REST application with a few types of resources:

- Registration resources, representing information about an account
- Authorization resources, representing an account's authorization to act for an identifier
- Challenge resources, representing a challenge to prove control of an identifier
- Certificate resources, representing issued certificates
- A "directory" resource
- A "new-registration" resource
- A "new-authorization" resource
- A "new-certificate" resource
- A "revoke-certificate" resource

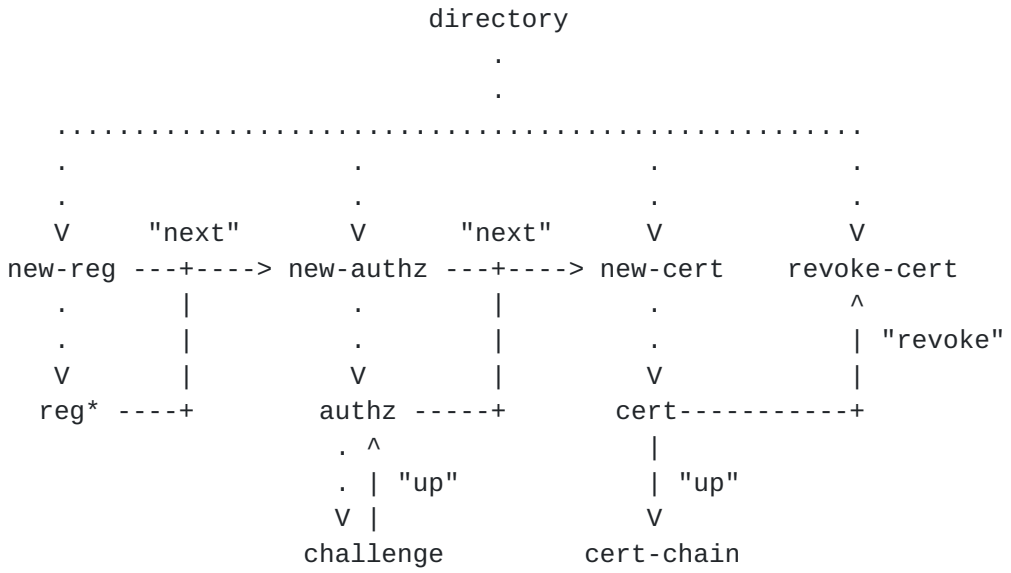
For the "new-X" resources above, the server MUST have exactly one resource for each function. This resource may be addressed by multiple URIs, but all must provide equivalent functionality.

ACME uses different URIs for different management functions. Each function is listed in a directory along with its corresponding URI, so clients only need to be configured with the directory URI. These URIs are connected by a few different link relations [RFC5988].

The "up" link relation is used with challenge resources to indicate the authorization resource to which a challenge belongs. It is also used 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 "directory" link relation is present on all resources other than the directory and indicates the directory URL.

The following diagram illustrates the relations between resources on an ACME server. The solid lines indicate link relations, and the dotted lines correspond to relationships expressed in other ways, e.g., the Location header in a 201 (Created) response.



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 pointing to a created resource.

Action	Request	Response
Register	POST new-reg	201 -> reg
Request challenges	POST new-authz	201 -> authz
Answer challenges	POST challenge	200
Poll for status	GET authz	200
Request issuance	POST new-cert	201 -> cert
Check for new cert	GET cert	200

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

6.1.1. Registration Objects

An ACME registration resource represents a set of metadata associated to an account key pair. Registration resources have the following structure:

key (required, dictionary): The public key of the account key pair, encoded as a JSON Web Key object [RFC7517].

contact (optional, array of string): An array of URIs that the server can use to contact the client for issues related to this authorization. For example, the server may wish to notify the client about server-initiated revocation.

agreement (optional, string): A URI referring to a subscriber agreement or terms of service provided by the server (see below). Including this field indicates the client's agreement with the referenced terms.

authorizations (required, string): A URI from which a list of authorizations granted to this account can be fetched via a GET request. The result of the GET request MUST be a JSON object whose "authorizations" field is an array of strings, where each string is the URI of an authorization belonging to this registration. The server SHOULD include pending authorizations, and SHOULD NOT include authorizations that are invalid or expired. The server MAY return an incomplete list, along with a Link header with link relation "next" indicating a URL to retrieve further entries.

certificates (required, string): A URI from which a list of certificates issued for this account can be fetched via a GET request. The result of the GET request MUST be a JSON object whose "certificates" field is an array of strings, where each string is the URI of a certificate. The server SHOULD NOT include expired or revoked certificates. The server MAY return an incomplete list, along with a Link header with link relation "next" indicating a URL to retrieve further entries.

```
{
  "resource": "new-reg",
  "contact": [
    "mailto:cert-admin@example.com",
    "tel:+12025551212"
  ],
  "agreement": "https://example.com/acme/terms",
  "authorizations": "https://example.com/acme/reg/1/authz",
  "certificates": "https://example.com/acme/reg/1/cert",
}
```

6.1.2. Authorization Objects

An ACME authorization object represents 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, dictionary of string): The identifier that the account is authorized to represent

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

value (required, string): The identifier itself.

status (required, string): The status of this authorization. Possible values are: "unknown", "pending", "processing", "valid", "invalid" and "revoked". If this field is missing, then the default value is "pending".

expires (optional, string): The timestamp after which the server will consider this authorization invalid, encoded in the format specified in RFC 3339 [RFC3339]. This field is REQUIRED for objects with "valid" in the "status field.

challenges (required, array): The challenges that the client needs to fulfill in order to prove possession of the identifier (for pending authorizations). For final authorizations, the challenges that were used. Each array entry is a dictionary with parameters required to validate the challenge, as specified in Section 7.

combinations (optional, array of arrays of integers): A collection of sets of challenges, each of which would be sufficient to prove possession of the identifier. Clients complete a set of challenges that covers at least one set in this array. Challenges are identified by their indices in the challenges array. If no "combinations" element is included in an authorization object, the client completes all challenges.

The only type of identifier defined by this specification is a fully-qualified domain name (type: "dns"). The value of the identifier MUST be the ASCII representation of the domain name. Wildcard domain names (with "*" as the first label) MUST NOT be included in authorization requests. See Section 6.5 below for more information about wildcard domains.

```
{
  "status": "valid",
  "expires": "2015-03-01T14:09:00Z",

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

  "challenges": [
    {
      "type": "http-01",
      "status": "valid",
      "validated": "2014-12-01T12:05:00Z",
      "keyAuthorization": "SXQe-2XODaDxNR...vb29HhjjLPSggwiE"
    }
  ],
}
```

6.2. Directory

In order to help clients configure themselves with the right URIs for each ACME operation, ACME servers provide a directory object. This should be the only URL needed to configure clients. It is a JSON dictionary, whose keys are the "resource" values listed in Section 5.1, and whose values are the URIs used to accomplish the corresponding function.

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

- a host which function as both an ACME and Web server may want to keep the root path "/" for an HTML "front page", and and place the ACME directory under path "/acme".

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

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

The following metadata items are defined, all of which are OPTIONAL:

"terms-of-service" (optional, string): A URI identifying the current terms of service.

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

"caa-identities" (optional, array of string): Each string MUST be a lowercase hostname which the ACME server recognises as referring to itself for the purposes of CAA record validation as defined in [RFC6844]. This allows clients to determine the correct issuer domain name to use when configuring CAA record.

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

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

{
  "new-reg": "https://example.com/acme/new-reg",
  "new-authz": "https://example.com/acme/new-authz",
  "new-cert": "https://example.com/acme/new-cert",
  "revoke-cert": "https://example.com/acme/revoke-cert",
  "meta": {
    "terms-of-service": "https://example.com/acme/terms",
    "website": "https://www.example.com/",
    "caa-identities": ["example.com"]
  }
}
```

6.3. Registration

A client creates a new account with the server by sending a POST request to the server's new-registration URI. The body of the request is a stub registration object containing only the "contact" field (along with the required "resource" field).

```
POST /acme/new-registration HTTP/1.1
Host: example.com

{
  "resource": "new-reg",
  "contact": [
    "mailto:cert-admin@example.com",
    "tel:+12025551212"
  ],
}
/* Signed as JWS */
```

The server MUST ignore any values provided in the "key", "authorizations", and "certificates" fields in registration bodies 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 may be provided by the client.

The server creates a registration object with the included contact information. The "key" element of the registration is set to the public key used to verify the JWS (i.e., the "jwk" element of the JWS header). The server returns this registration object in a 201 (Created) response, with the registration URI in a Location header field. The server SHOULD also indicate its new-authorization URI using the "next" link relation.

If the server already has a registration object with the provided account key, then it MUST return a 409 (Conflict) response and provide the URI of that registration in a Location header field. This allows a client that has an account key but not the corresponding registration URI to recover the registration URI.

If the server wishes to present the client with terms under which the ACME service is to be used, it MUST indicate the URI where such terms can be accessed in a Link header with link relation "terms-of-service". As noted above, the client may indicate its agreement with these terms by updating its registration to include the "agreement" field, with the terms URI as its value. When these terms change in a way that requires an agreement update, the server MUST use a different URI in the Link header.

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: https://example.com/acme/reg/asdf
Link: <https://example.com/acme/new-authz>;rel="next"
Link: <https://example.com/acme/terms>;rel="terms-of-service"
Link: <https://example.com/acme/some-directory>;rel="directory"

{
  "key": { /* JWK from JWS header */ },

  "contact": [
    "mailto:cert-admin@example.com",
    "tel:+12025551212"
  ]
}
```

If the client wishes to update this information in the future, it sends a POST request with updated information to the registration URI. The server MUST ignore any updates to the "key", "authorizations, or "certificates" fields, and MUST verify that the request is signed with the private key corresponding to the "key" field of the request before updating the registration.

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

```
POST /acme/reg/asdf HTTP/1.1
Host: example.com

{
  "resource": "reg",
  "contact": [
    "mailto:certificates@example.com",
    "tel:+12125551212"
  ],
}
/* Signed as JWS */
```

Servers SHOULD NOT respond to GET requests for registration resources as these requests are not authenticated. If a client wishes to query the server for information about its account (e.g., to examine the "contact" or "certificates" fields), then it SHOULD do so by sending a POST request with an empty update. That is, it should send a JWS whose payload is trivial ({"resource":"reg"}). In this case the server reply MUST contain the same link headers sent for a new registration, to allow a client to retrieve the "new-authorization" and "terms-of-service" URI

6.3.1. Account Key Roll-over

A client may wish to change the public key that is associated with a registration, e.g., in order to mitigate the risk of key compromise. To do this, the client first constructs a JSON object representing a request to update the registration:

resource (required, string): The string "reg", indicating an update to the registration.

oldKey (required, string): The JWK thumbprint of the old key [RFC7638], base64url-encoded

```
{
  "resource": "reg",
  "oldKey": "D7J9RL1f-RWUl68JP-gW1KSl2TkIrJB7hK6rLFFeYMU"
}
```

The client signs this object with the new key pair and encodes the object and signature as a JWS. The client then sends this JWS to the server in the "newKey" field of a request to update the registration.

```
POST /acme/reg/asdf HTTP/1.1
Host: example.com

{
  "resource": "reg",
  "newKey": /* JSON object signed as JWS with new key */
}
/* Signed as JWS with original key */
```

On receiving a request to the registration URL with the "newKey" attribute set, the server MUST perform the following steps:

1. Check that the contents of the "newKey" attribute are a valid JWS

2. Check that the "newKey" JWS verifies using the key in the "jwk" header parameter of the JWS

3. Check that the payload of the JWS is a valid JSON object

4. Check that the "resource" field of the object has the value "reg"

5. Check that the "oldKey" field of the object contains the JWK thumbprint of the account key for this registration

If all of these checks pass, then the server updates the registration by replacing the old account key with the public key carried in the "jwk" header parameter of the "newKey" JWS object.

If the update was successful, then the server sends a response with status code 200 (OK) and the updated registration object as its body. If the update was not successful, then the server responds with an error status code and a problem document describing the error.

6.3.2. Deleting an Account

If a client no longer wishes to have an account key registered with the server, it may request that the server delete its account by sending a POST request to the account URI containing the "delete" field.

delete (required, boolean): The boolean value "true".

The request object MUST contain the "resource" field as required above (with the value "reg"). It MUST NOT contain any fields besides "resource" and "delete".

Note that although this object is very simple, the risk of replay or fraudulent generation via signing oracles is mitigated by the need for an anti-replay token in the protected header of the JWS.

```
POST /acme/reg/asdf HTTP/1.1
Host: example.com

{
  "resource": "reg",
  "delete": true,
}
/* Signed as JWS */
```

On receiving a POST to an account URI containing a "delete" field, the server MUST verify that no other fields were sent in the object (other than "resource"), and it MUST verify that the value of the "delete" field is "true" (as a boolean, not a string). If either of these checks fails, then the server MUST reject the request with status code 400 (Bad Request).

If the server accepts the deletion request, then it MUST delete the account and all related objects and send a response with a 200 (OK) status code and an empty body. The server SHOULD delete any authorization objects related to the deleted account, since they can no longer be used. The server SHOULD NOT delete certificate objects

related to the account, since certificates issued under the account continue to be valid until they expire or are revoked.

6.4. Identifier Authorization

The identifier authorization process establishes the authorization of an account to manage certificates for a given identifier. This process must assure the server of two things: First, that the client controls the private key of the account key pair, and second, that the client holds the identifier in question. This process may be repeated to associate multiple identifiers to a key pair (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).

As illustrated by the figure in the overview section above, the authorization process proceeds in two phases. The client first requests a new authorization, and the server issues challenges, then the client responds to those challenges and the server validates the client's responses.

To begin the key authorization process, the client sends a POST request to the server's new-authorization resource. The body of the POST request MUST contain a JWS object, whose payload is a partial authorization object. This JWS object MUST contain only the "identifier" field, so that the server knows what identifier is being authorized. The server MUST ignore any other fields present in the client's request object.

The authorization object is implicitly tied to the account key used to sign the request. Once created, the authorization may only be updated by that account.

```
POST /acme/new-authorization HTTP/1.1
Host: example.com

{
  "resource": "new-authz",
  "identifier": {
    "type": "dns",
    "value": "example.org"
  }
}
/* Signed as JWS */
```

Before processing the authorization further, 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 a 403 (Forbidden) error, 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 initial authorization object 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" and "combinations": As selected by the server's policy for this identifier

The server allocates a new URI for this authorization, and returns a 201 (Created) response, with the authorization URI in a Location header field, and the JSON authorization object in the body.

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: https://example.com/authz/asdf
Link: <https://example.com/acme/new-cert>;rel="next"
Link: <https://example.com/acme/some-directory>;rel="directory"

{
  "status": "pending",

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

  "challenges": [
    {
      "type": "http-01",
      "uri": "https://example.com/authz/asdf/0",
      "token": "IlirfxKKXAsHtmzK29Pj8A"
    },
    {
      "type": "dns-01",
      "uri": "https://example.com/authz/asdf/1",
      "token": "DGyRejmCefe7v4NfDGDKfA"
    }
  },

  "combinations": [[0], [1]]
}
```

6.4.1. Responding to Challenges

To prove control of the identifer and receive authorization, the client needs to respond with information to complete the challenges. To do this, the client updates the authorization object received from the server by filling in any required information in the elements of the "challenges" dictionary. (This is also the stage where the client should perform any actions required by the challenge.)

The client sends these updates back to the server in the form of a JSON object with the response fields required by the challenge type, carried in a POST request to the challenge URI (not authorization URI or the new-authorization URI). This allows the client to send information only for challenges it is responding to.

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/authz/asdf/0 HTTP/1.1
Host: example.com

{
  "resource": "challenge",
  "type": "http-01",
  "keyAuthorization": "IlirfxKKXA...vb29HhjjLPSggwiE"
}
/* Signed as JWS */
```

The server updates the authorization document by updating its representation of the challenge with the response fields 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. The server provides a 200 (OK) response with the updated challenge object as its body.

If the client's response is invalid for some 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.

Presumably, the client's responses provide the server with enough information to validate one or more challenges. The server is said to "finalize" the authorization when it has completed all the validations it is going to complete, and assigns 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", the server MUST add an "expires" field to the authorization. When finalizing an authorization, the server MAY remove the "combinations" field (if present) or remove any challenges still pending. 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 GET request to the authorization URI, 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 202

(Accepted) response, and MAY include a Retry-After header field to suggest a polling interval to the client.

```
GET /acme/authz/asdf HTTP/1.1
Host: example.com

HTTP/1.1 200 OK

{
  "status": "valid",
  "expires": "2015-03-01T14:09:00Z",

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

  "challenges": [
    {
      "type": "http-01"
      "status": "valid",
      "validated": "2014-12-01T12:05:00Z",
      "token": "IlirfxKKXAsHtmzK29Pj8A",
      "keyAuthorization": "IlirfxKKXA...vb29HhjjLPSggwiE"
    }
  ]
}
```

6.4.2. Deleting an Authorization

If a client wishes to relinquish its authorization to issue certificates for an identifier, then it may request that the server delete the authorization. The client makes this request by sending a POST request to the authorization URI containing a payload in the same format as in Section 6.3.2. The only difference is that the value of the "resource" field is "authz".

```
POST /acme/authz/asdf HTTP/1.1
Host: example.com

{
  "resource": "authz",
  "delete": true,
}
/* Signed as JWS */
```

The server MUST perform the same validity checks as in Section 6.3.2 and reject the request if they fail. If the server deletes the

account then it MUST send a response with a 200 (OK) status code and an empty body.

6.5. Certificate Issuance

The holder of an account key pair authorized for one or more identifiers may use ACME to request that a certificate be issued for any subset of those identifiers. The client makes this request by sending a POST request to the server's new-certificate resource. The body of the POST is a JWS object whose JSON payload contains a Certificate Signing Request (CSR) [RFC2986]. The CSR encodes the parameters of the requested certificate; authority to issue is demonstrated by the JWS signature by an account key, from which the server can look up related authorizations. Some attributes which cannot be reflected in a CSR are placed directly in the certificate request.

csr (required, string): A CSR encoding the parameters for the certificate being requested. The CSR is sent in the Base64url-encoded version of the DER format. (Note: This field uses the same modified Base64 encoding rules used elsewhere in this document, so it is different from PEM.)

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-cert HTTP/1.1
Host: example.com
Accept: application/pkix-cert

{
  "resource": "new-cert",
  "csr": "5jNudRx6Ye4HzKEqT5...FS6aKdZeGsysoCo4H9P",
  "notBefore": "2016-01-01T00:00:00Z",
  "notAfter": "2016-01-08T00:00:00Z"
}
/* Signed as JWS */
```

The CSR encodes the client's requests with regard to the content of the certificate to be issued. The CSR MUST indicate the requested identifiers, either in the commonName portion of the requested subject name, or in an extensionRequest attribute [RFC2985] requesting a subjectAltName extension.

The values provided in the CSR are only a request, and are not guaranteed. The server SHOULD return an error if it cannot fulfil the request as specified, but MAY issue a certificate with contents other than those requested, according to its local policy (e.g., removing identifiers for which the client is not authorized).

It is up to the server's local policy to decide which names are acceptable in a certificate, given the authorizations that the server associates with the client's account key. A server MAY consider a client authorized for a wildcard domain if it is authorized for the underlying domain name (without the "*" label). Servers SHOULD NOT extend authorization across identifier types. For example, if a client is authorized for "example.com", then the server should not allow the client to issue a certificate with an iPAddress subjectAltName, even if it contains an IP address to which example.com resolves.

If the CA decides to issue a certificate, then the server creates a new certificate resource and returns a URI for it in the Location header field of a 201 (Created) response.

```
HTTP/1.1 201 Created
Location: https://example.com/acme/cert/asdf
```

If the certificate is available at the time of the response, it is provided in the body of the response. If the CA has not yet issued the certificate, the body of this response will be empty. The client should then send a GET request to the certificate URI to poll for the certificate. As long as the certificate is unavailable, the server MUST provide a 202 (Accepted) response and include a Retry-After header to indicate when the server believes the certificate will be issued (as in the example above).

```
GET /acme/cert/asdf HTTP/1.1
Host: example.com
Accept: application/pkix-cert
```

```
HTTP/1.1 202 Accepted
Retry-After: 120
```

The default format of the certificate is DER (application/pkix-cert). The client may request other formats by including an Accept header in its request.

The server provides metadata about the certificate in HTTP headers. In particular, the server MUST include a Link relation header field [RFC5988] with relation "up" to provide a certificate under which

this certificate was issued, and one with relation "author" to indicate the registration under which this certificate was issued.

The server MAY include an Expires header as a hint to the client about when to renew the certificate. (Of course, the real expiration of the certificate is controlled by the notAfter time in the certificate itself.)

If the CA participates in Certificate Transparency (CT) [RFC6962], then they may want to provide the client with a Signed Certificate Timestamp (SCT) that can be used to prove that a certificate was submitted to a CT log. An SCT can be included as an extension in the certificate or as an extension to OCSP responses for the certificate. The server can also provide the client with direct access to an SCT for a certificate using a Link relation header field with relation "ct-sct".

```
GET /acme/cert/asdf HTTP/1.1
Host: example.com
Accept: application/pkix-cert

HTTP/1.1 200 OK
Content-Type: application/pkix-cert
Link: <https://example.com/acme/ca-cert>;rel="up";title="issuer"
Link: <https://example.com/acme/revoke-cert>;rel="revoke"
Link: <https://example.com/acme/reg/asdf>;rel="author"
Link: <https://example.com/acme/sct/asdf>;rel="ct-sct"
Link: <https://example.com/acme/some-directory>;rel="directory"
Location: https://example.com/acme/cert/asdf
Content-Location: https://example.com/acme/cert-seq/12345

[DER-encoded certificate]
```

A certificate resource always represents the most recent certificate issued for the name/key binding expressed in the CSR. If the CA allows a certificate to be renewed, then it publishes renewed versions of the certificate through the same certificate URI.

Clients retrieve renewed versions of the certificate using a GET query to the certificate URI, which the server should then return in a 200 (OK) response. The server SHOULD provide a stable URI for each specific certificate in the Content-Location header field, as shown above. Requests to stable certificate URIs MUST always result in the same certificate.

To avoid unnecessary renewals, the CA may choose not to issue a renewed certificate until it receives such a request (if it even allows renewal at all). In such cases, if the CA requires some time

to generate the new certificate, the CA MUST return a 202 (Accepted) response, with a Retry-After header field that indicates when the new certificate will be available. The CA MAY include the current (non-renewed) certificate as the body of the response.

Likewise, in order to prevent unnecessary renewal due to queries by parties other than the account key holder, certificate URIs should be structured as capability URLs [W3C.WD-capability-urls-20140218].

From the client's perspective, there is no difference between a certificate URI that allows renewal and one that does not. If the client wishes to obtain a renewed certificate, and a GET request to the certificate URI does not yield one, then the client may initiate a new-certificate transaction to request one.

6.6. Certificate Revocation

To request that a certificate be revoked, the client sends a POST request to the ACME server's revoke-cert URI. 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: This field uses the same modified Base64 encoding rules used elsewhere in this document, so it is different from PEM.)

```
POST /acme/revoke-cert HTTP/1.1
Host: example.com

{
  "resource": "revoke-cert",
  "certificate": "MIIEDTCCAvegAwIBAgIRAP8..."
}
/* Signed as JWS */
```

Revocation requests are different from other ACME request in that they can be signed either with an account key pair or the key pair in the certificate. Before revoking a certificate, the server MUST verify that the key used to sign the request is authorized to revoke the certificate. The server SHOULD consider at least the following keys authorized for a given certificate:

- the public key in the certificate.
- an account key that is authorized to act for all of the identifier(s) in the certificate.

If the revocation succeeds, the server responds with status code 200 (OK). If the revocation fails, the server returns an error.

```
HTTP/1.1 200 OK
Content-Length: 0

--- or ---

HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
  "type": "urn:ietf:params:acme:error:unauthorized"
  "detail": "No authorization provided for name example.net"
  "instance": "http://example.com/doc/unauthorized"
}
```

7. 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 key 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
- 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 challenge 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 URI.

This section describes an initial set of challenge types. Each challenge must describe:

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.

uri (required, string): The URI to which a response can be posted.

status (required, string): The status of this authorization. Possible values are: "pending", "valid", and "invalid". If this field is missing, then the default value is "pending".

validated (optional, string): The time at which this challenge was completed by the server, encoded in the format specified in RFC 3339 [RFC3339]. This field is REQUIRED if the "status" field is "valid".

error (optional, dictionary of string): The error that occurred while the server was validating the challenge, if any. This field is structured as a problem document [I-D.ietf-appsawg-http-problem].

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 they failed the challenge.

Different challenges allow the server to obtain proof of different aspects of control over an identifier. In some challenges, like HTTP and TLS SNI, 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 identifier, there will need to be new challenge types, and they will need to specify which types of identifier they apply to.

[[Editor's Note: In pre-RFC versions of this specification, challenges are labeled by type, and with the version of the draft in

which they were introduced. For example, if an HTTP challenge were introduced in version -03 and a breaking change made in version -05, then there would be a challenge labeled "http-03" and one labeled "http-05" - but not one labeled "http-04", since challenge in version -04 was compatible with one in version -04.]]

[[Editor's Note: Operators SHOULD NOT issue "combinations" arrays in authorization objects that require the client to perform multiple challenges over the same type, e.g., ["http-03", "http-05"]. Challenges within a type are testing the same capability of the domain owner, and it may not be possible to satisfy both at once.]]

7.1. Key Authorizations

Several of the challenges in this document makes use of a key authorization string. A key authorization is a string that expresses a domain holder's authorization for a specified key to satisfy a specified challenge, by concatenating the token for the challenge with a key fingerprint, separated by a "." character:

key-authz = token || '.' || base64url(JWK_Thumbprint(accountKey))

The "JWK_Thumbprint" step indicates the computation specified in [RFC7638], using the SHA-256 digest. As specified in the individual challenges below, the token for a challenge is a JSON string comprised entirely of characters in the URL-safe Base64 alphabet. The "||" operator indicates concatenation of strings.

In computations involving key authorizations, such as the digest computations required for the DNS and TLS SNI challenges, the key authorization string MUST be represented in UTF-8 form (or, equivalently, ASCII).

An example of how to compute a JWK thumbprint can be found in Section 3.1 of [RFC7638]. Note that some cryptographic libraries prepend a zero octet to the representation of the RSA public key parameters N and E, in order to avoid ambiguity with regard to the sign of the number. As noted in JWA [RFC7518], a JWK object MUST NOT include this zero octet. That is, any initial zero octets MUST be stripped before the values are base64url-encoded.

7.2. HTTP

With HTTP validation, the client in an ACME transaction proves its control over a domain name by proving that it can provision resources on an HTTP server that responds for 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 A and AAAA records. 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, in order to prevent an attacker from guessing it. It MUST NOT contain any characters outside the URL-safe Base64 alphabet and MUST NOT contain any padding characters ("=").

```
{
  "type": "http-01",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA",
}
```

A client responds to 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.

```
.well-known/acme-challenge/evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA
```

The client's response to this challenge indicates its agreement to this challenge by sending the server the key authorization covering the challenge's token and the client's account key. In addition, the client MAY advise the server at which IP the challenge is provisioned.

keyAuthorization (required, string): The key authorization for this challenge. This value MUST match the token from the challenge and the client's account key.

address (optional, string): An IPv4 or IPv6 address, in dotted decimal form or [RFC4291] form, respectively. If given, this address MUST be included in the set of IP addresses to which the domain name resolves when the server attempts validation. If given, the server SHOULD connect to that specific IP address

instead of arbitrarily choosing an IP from the set of A and AAAA records to which the domain name resolves.

```
{
  "keyAuthorization": "evaGxfADs...62jcerQ"
}
/* Signed as JWS */
```

On receiving a response, the server MUST verify that the key authorization in the response matches the "token" value in the challenge and the client's account key. If they do not match, then the server MUST return an HTTP error in response to the POST request in which the client sent the challenge.

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. Form a URI by populating the URI 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 URI is well-formed.

3. If the client has supplied an address to use, verify that the address is included in the A or AAAA records to which the domain name resolves. If the address is not included in the result, the validation fails.

4. Dereference the URI using an HTTP GET request. If an address was supplied by the client, use that address to establish the HTTP connection.

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

6. Verify that key authorization provided by the server matches the token for this challenge and the client's account key.

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.

7.3. TLS with Server Name Indication (TLS SNI)

The TLS with Server Name Indication (TLS SNI) validation method proves control over a domain name by requiring the client to configure a TLS server referenced by an A/AAAA record under the domain name to respond to specific connection attempts utilizing the Server Name Indication extension [RFC6066]. The server verifies the client's challenge by accessing the reconfigured server and verifying a particular challenge certificate is presented.

type (required, string): The string "tls-sni-02"

token (required, string): A random value that uniquely identifies the challenge. This value MUST have at least 128 bits of entropy, in order to prevent an attacker from guessing it. It MUST NOT contain any characters outside the URL-safe Base64 alphabet and MUST NOT contain any padding characters ("=").

```
{
  "type": "tls-sni-02",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA"
}
```

A client responds to this challenge by constructing a self-signed certificate which the client MUST provision at the domain name concerned in order to pass the challenge.

The certificate may be constructed arbitrarily, except that each certificate MUST have exactly two subjectAlternativeNames, SAN A and SAN B. Both MUST be dNSNames.

SAN A MUST be constructed as follows: compute the SHA-256 digest of the UTF-8-encoded challenge token and encode it in lowercase hexadecimal form. The dNSName is "x.y.token.acme.invalid", where x is the first half of the hexadecimal representation and y is the second half.

SAN B MUST be constructed as follows: compute the SHA-256 digest of the UTF-8 encoded key authorization and encode it in lowercase hexadecimal form. The dNSName is "x.y.ka.acme.invalid" where x is the first half of the hexadecimal representation and y is the second half.

The client MUST ensure that the certificate is served to TLS connections specifying a Server Name Indication (SNI) value of SAN A.

The response to the TLS-SNI challenge simply acknowledges that the client is ready to fulfill this challenge.

keyAuthorization (required, string): The key authorization for this challenge. This value MUST match the token from the challenge and the client's account key.

```
{
  "keyAuthorization": "evaGxfADs...62jcerQ",
}
/* Signed as JWS */
```

On receiving a response, the server MUST verify that the key authorization in the response matches the "token" value in the challenge and the client's account key. If they do not match, then the server MUST return an HTTP error in response to the POST request in which the client sent the challenge.

Given a challenge/response pair, the ACME server verifies the client's control of the domain by verifying that the TLS server was configured appropriately, using these steps:

1. Compute SAN A and SAN B in the same way as the client.

2. Open a TLS connection to the domain name being validated on the requested port, presenting SAN A in the SNI field. In the ClientHello initiating the TLS handshake, the server MUST include a server_name extension (i.e., SNI) containing SAN A. The server SHOULD ensure that it does not reveal SAN B in any way when making the TLS connection, such that the presentation of SAN B in the returned certificate proves association with the client.

3. Verify that the certificate contains a subjectAltName extension containing dNSName entries of SAN A and SAN B and no other entries. The comparison MUST be insensitive to case and ordering of names.

It is RECOMMENDED that the ACME server validation TLS connections from multiple vantage points to reduce the risk of DNS hijacking attacks.

If all of the above verifications succeed, then the validation is successful. Otherwise, the validation fails.

7.4. DNS

When the identifier being validated is a domain name, the client can prove control of that domain by provisioning a resource record under it. The DNS challenge requires the client to provision a TXT record containing a designated value under 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, in order to prevent an attacker from guessing it. It MUST NOT contain any characters outside the URL-safe Base64 alphabet and MUST NOT contain any padding characters ("=").

```
{
  "type": "dns-01",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA"
}
```

A client responds to 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 of the key authorization.

The record provisioned to the DNS is 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 "example.com", then the client would provision the following DNS record:

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

The response to the DNS challenge provides the computed key authorization to acknowledge that the client is ready to fulfill this challenge.

keyAuthorization (required, string): The key authorization for this challenge. This value MUST match the token from the challenge and the client's account key.

```
{
  "keyAuthorization": "evaGxfADs...62jcerQ",
}
/* Signed as JWS */
```

On receiving a response, the server MUST verify that the key authorization in the response matches the "token" value in the challenge and the client's account key. If they do not match, then the server MUST return an HTTP error in response to the POST request in which the client sent the challenge.

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

1. Compute the SHA-256 digest of the key authorization

2. Query for TXT records under the validation domain name

3. Verify that the contents of one of the TXT records matches 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.

8. IANA Considerations

[[Editor's Note: Should we create a registry for tokens that go into the various JSON objects used by this protocol, i.e., the field names in the JSON objects?]]

9. Well-Known URI for the HTTP Challenge

The "Well-Known URIs" registry should be updated with the following additional value (using the template from [RFC5785]):

URI suffix: acme-challenge

Change controller: IETF

Specification document(s): This document, Section Section 7.2

Related information: N/A

9.1. Replay-Nonce HTTP Header

The "Message Headers" registry should be updated with the following additional value:

| Header Field Name | Protocol | Status | Reference |
+:------------+:------+:------+:-----------+ | Replay-Nonce | http | standard | Section 5.4.1 |

9.2. "nonce" JWS Header Parameter

The "JSON Web Signature and Encryption Header Parameters" registry should be updated with the following additional value:

- Header Parameter Name: "nonce"

- Header Parameter Description: Nonce

- Header Parameter Usage Location(s): JWE, JWS
- Change Controller: IESG
- Specification Document(s): Section 5.4.2 of RFC XXXX

[[RFC EDITOR: Please replace XXXX above with the RFC number assigned to this document]]

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

The "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry should be updated with the following additional value, following the template in [RFC3553]:

Registry name: acme

Specification: RFC XXXX

Repository: URL-TBD

Index value: No transformation needed. The

[[RFC EDITOR: Please replace XXXX above with the RFC number assigned to this document, and replace URL-TBD with the URL assigned by IANA for registries of ACME parameters.]]

9.4. New Registries

This document requests that IANA create three new registries:

1. ACME Error Codes
2. ACME Identifier Types
3. ACME Challenge Types

All of these registries should be administered under a Specification Required policy [RFC5226].

9.4.1. Error Codes

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

Template:

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

- Description: A human-readable description of the error

- Reference: Where the error is defined

Initial contents: The codes and descriptions in the table in Section 5.5 above, with the Reference field set to point to this specification.

9.4.2. Identifier Types

This registry lists the types of identifiers that ACME clients may request authorization to issue in certificates.

Template:

- Label: The value to be put in the "type" field of the identifier object

- Reference: Where the identifier type is defined

Initial contents:

Label	Reference
dns	RFC XXXX

[[RFC EDITOR: Please replace XXXX above with the RFC number assigned to this document]]

9.4.3. Challenge Types

This registry lists the ways that ACME servers can offer to validate control of an identifier. The "Identifier Type" field in template MUST be contained in the Label column of the ACME Identifier Types registry.

Template:

- Label: The value to be put in the "type" field of challenge objects using this validation mechanism

- Identifier Type: The type of identifier that this mechanism applies to

o Reference: Where the challenge type is defined

Initial Contents

Label	Identifier Type	Reference
http	dns	RFC XXXX
tls-sni	dns	RFC XXXX
dns	dns	RFC XXXX

[[RFC EDITOR: Please replace XXXX above with the RFC number assigned to this document]]

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 account key authorized for that identifier

2. Once authorized, an account key's authorizations cannot be improperly transferred to another account key

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.

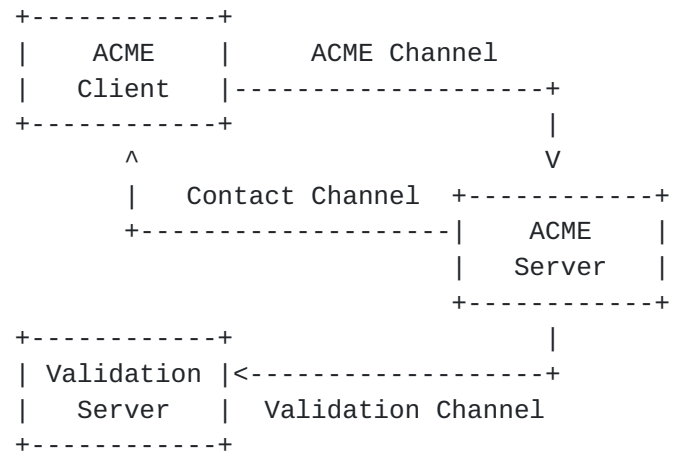
10.1. Threat model

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 three "channels":

o 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

- A contact channel, over which the ACME server sends messages to the registered contacts for ACME clients



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

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 application-layer man in the middle attacks, 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 or contact channels can't obtain illegitimate authorization by acting as an ACME client (legitimately, in terms of the protocol).

10.2. Integrity of Authorizations

ACME allows anyone to request challenges for an identifier by registering an account key and sending a new-authorization request under 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 an ACME MitM can switch out a legitimate domain holder's account key for one of his choosing, e.g.:

- Legitimate domain holder registers account key pair A
- MitM registers account key pair B
- Legitimate domain holder sends a new-authorization request signed under account key A
- MitM suppresses the legitimate request, but sends the same request signed under account key B
- ACME server issues challenges and MitM forwards them to the legitimate domain holder
- Legitimate domain holder provisions the validation response
- ACME server performs validation query and sees the response provisioned by the legitimate domain holder
- Because the challenges were issued in response to a message signed account key B, the ACME server grants authorization to account key B (the MitM) instead of account key A (the legitimate domain holder)

All of the challenges above that require an out-of-band query by the server have a binding to the account private key, such that only the account private key holder can successfully respond to the validation query:

- HTTP: The value provided in the validation request is signed by the account private key.
- TLS SNI: The validation TLS request uses the account key pair as the server's key pair.
- DNS: The MAC covers the account key, and the MAC key is derived from an ECDH public key signed with the account private key.

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:

- HTTP: Provision files under .well-known on a web server for the domain
- TLS SNI: Configure a TLS server for the domain
- DNS: Provision DNS resource records for the domain

There are several ways that these assumptions can be violated, both by misconfiguration and by attack. For example, on a web server that allows non-administrative users to write to .well-known, any user can claim to own the server's hostname by responding to an HTTP challenge, and likewise for TLS configuration and TLS SNI.

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 web site 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 web site operator provisions a catch-all redirect rule that redirects requests for unknown resources to 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 TLS SNI or HTTP validation query to a 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 [I-D.vixie-dnsext-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. For identifiers where the server already has some public key associated with the domain this attack can be prevented by requiring the client to prove control of the corresponding private key.

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.

10.4. 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?

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

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. For example:

- It is advisable to perform DNS queries via TCP to mitigate DNS forgery attacks over UDP

[[TODO: Other operational considerations]]

11.1. Default Virtual Hosts

In many cases, TLS-based services are deployed on hosted platforms that use the Server Name Indication (SNI) TLS extension to distinguish between different hosted services or "virtual hosts". When a client initiates a TLS connection with an SNI value indicating a provisioned host, the hosting platform routes the connection to that host.

When a connection come in with an unknown SNI value, one might expect the hosting platform to terminate the TLS connection. However, some hosting platforms will choose a virtual host to be the "default", and route connections with unknown SNI values to that host.

In such cases, the owner of the default virtual host can complete a TLS-based challenge (e.g., "tls-sni-02") for any domain with an A record that points to the hosting platform. This could result in mis-issuance in cases where there are multiple hosts with different owners resident on the hosting platform.

A CA that accepts TLS-based proof of domain control should attempt to check whether a domain is hosted on a domain with a default virtual host before allowing an authorization request for this host to use a TLS-based challenge. A default virtual host can be detected by initiating TLS connections to the host with random SNI values within the namespace used for the TLS-based challenge (the "acme.invalid" namespace for "tls-sni-02").

11.2. Use of DNSSEC Resolvers

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. It is therefore RECOMMENDED that ACME-based CAs make all DNS queries via DNSSEC-validating stub or recursive resolvers. This provides additional protection to domains which choose to make use of DNSSEC.

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

12. 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.

- Peter Eckersley, EFF
- Eric Rescorla, Mozilla
- Seth Schoen, EFF

- Alex Halderman, University of Michigan
- 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.

13. References

13.1. Normative References

[I-D.ietf-appsawg-http-problem]
mnot, m. and E. Wilde, "Problem Details for HTTP APIs", draft-ietf-appsawg-http-problem-03 (work in progress), January 2016.

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

[RFC2314] Kaliski, B., "PKCS #10: Certification Request Syntax Version 1.5", RFC 2314, DOI 10.17487/RFC2314, March 1998, <http://www.rfc-editor.org/info/rfc2314>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <http://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, <http://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, <http://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, <http://www.rfc-editor.org/info/rfc3339>.

[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, <http://www.rfc-editor.org/info/rfc3553>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <http://www.rfc-editor.org/info/rfc4291>.

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

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[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, <http://www.rfc-editor.org/info/rfc5280>.

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

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <http://www.rfc-editor.org/info/rfc5988>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <http://www.rfc-editor.org/info/rfc6066>.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <http://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, <http://www.rfc-editor.org/info/rfc6844>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <http://www.rfc-editor.org/info/rfc6962>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

[RFC7469] Evans, C., Palmer, C., and R. Sleevi, "Public Key Pinning Extension for HTTP", RFC 7469, DOI 10.17487/RFC7469, April 2015, <http://www.rfc-editor.org/info/rfc7469>.

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

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <http://www.rfc-editor.org/info/rfc7517>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <http://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, <http://www.rfc-editor.org/info/rfc7638>.

13.2. Informative References

[I-D.vixie-dnsext-dns0x20] Vixie, P. and D. Dagon, "Use of Bit 0x20 in DNS Labels to Improve Transaction Identity", draft-vixie-dnsext-dns0x20-00 (work in progress), March 2008.

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

[W3C.CR-cors-20130129]
Kesteren, A., "Cross-Origin Resource Sharing", World Wide Web Consortium CR CR-cors-20130129, January 2013, <http://www.w3.org/TR/2013/CR-cors-20130129>.

[W3C.WD-capability-urls-20140218]
Tennison, J., "Good Practices for Capability URLs", World Wide Web Consortium WD WD-capability-urls-20140218, February 2014, <http://www.w3.org/TR/2014/WD-capability-urls-20140218>.

Authors' Addresses

Richard Barnes
Mozilla

Email: rlb@ipv.sx

Jacob Hoffman-Andrews
EFF

Email: jsha@eff.org

James Kasten
University of Michigan

Email: jdkasten@umich.edu