Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 9, 2017

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

July 08, 2016

Automatic Certificate Management Environment (ACME)
draft-ietf-acme-acme-03

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).

Status of This Memo

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 January 9, 2017.

Copyright Notice

Table of Contents

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:

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

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

- o 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.

- o 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. The server can also provide terms of service at this stage, which the client can present to a human user.

```
      Client                                                 Server

      Contact Information
      Signature                     ------->

                                    <-------           Registration
                                                   Terms of Service
```

Once the client is registered, there are three major steps it needs to take to get a certificate:

1. Apply for a certificate to be issued

2. Fulfill the server's requirements for issuance

3. Finalize the application and request issuance

The client's application for a certificate describes the desired certificate using a PKCS#10 Certificate Signing Request (CSR) plus a few additional fields that capture semantics that are not supported in the CSR format. If the server is willing to consider issuing such a certificate, it responds with a list of requirements that the client must satisfy before the certificate will be issued.

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

Once the validation process is complete and the server is satisfied that the client has met its requirements, the server can either proactively issue the requested certificate or wait for the client to request that the application be "finalized", at which point the certificate will be issued and provided to the client.

```
      Application
      Signature                     ------->
                                    <-------            Requirements
                                                    (e.g., Challenges)

      Responses
      Signature                     ------->

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

      Finalize application
      Signature                     ------->
                                    <-------             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.

```
      Client                                                  Server

      Revocation request
      Signature                    -------->

                                   <--------                  Result
```

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

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

o The JWS MUST be encoded using UTF-8

o The JWS MUST NOT have the value "none" in its "alg" field

o The JWS Protected Header MUST include the following fields:

   * "alg"

   * "jwk"

   * "nonce" (defined below)

* "url" (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.

In the examples below, JWS objects are shown in the JSON or flattened JSON serialization, with the protected header and payload expressed as base64url(content) instead of the actual base64-encoded value, so that the content is readable. Some fields are omitted for brevity, marked with "...".

5.3. Request URI 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 provide a limited integrity mechanism, which protects against an intermediary changing the request URI to another 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).

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

Except for the directory resource, all ACME resources are addressed with URLs provided to the client by the server. In such cases, the client MUST set the "url" field to the exact string provided by the server (rather than performing any re-encoding on the URL).

5.3.1. "url" (URL) JWS header parameter

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

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. Rate limits

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

In addition to the human readable "detail" field of the error response, the server MAY send one or multiple tokens in the "Link" header pointing to documentation about the specific hit rate limits using the "rate-limit" relation.

5.6. 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 [RFC7807]. 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 validation target
dnssec	DNSSEC validation failed
caa	CAA records forbid the CA from issuing
malformed	The request message was malformed
serverInternal	The server experienced an internal error
tls	The server received 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 contact URI for a registration was invalid
rejectedIdentifier	The server will not issue for the identifier
unsupportedIdentifier	Identifier is not supported, but may be in future

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
- Application for a Certificate
- Account Key Authorization
- Certificate Issuance
- 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
- Application resources, represnting an account's requests to issue certificates
- 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-application" resource
- A "revoke-certificate" resource
- A "key-change" 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. For the most part, these relations are expressed by URLs provided as strings in the resources' JSON representations. Lines with labels in quotes indicate HTTP link relations

```
                               directory
                                   |
                                   |
    --------------------------------------------------------
    |                  |                                |
    |                  |                                |
    V                  V                                V
 new-reg            new-app                        revoke-cert
    |                  |                                ^
    |                  |                                | "revoke"
    V                  V                                |
   reg -------------> app -------------> cert ---------+
                      | ^                  |
                      | | "up"             | "up"
                      V |                  V
                     authz            cert-chain
                      | ^
                      | | "up"
                      V |
                    challenge
```

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
Apply for a cert	POST new-app	201 -> app
Fetch challenges	GET authz	200
Answer challenges	POST challenge	200
Poll for status	GET authz	200
Request issuance	POST app	200
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. 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 drawn from the following table and whose values are the corresponding URLs.

Key	URL in value
new-reg	New registration
new-app	New application
revoke-cert	Revoke certificate
key-change	Key change

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:

o 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".

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

6.1.2. 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].

status (required, string): "good" or "deactivated"

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.

applications (required, string): A URI from which a list of authorizations submitted by this account can be fetched via a GET request. The result of the GET request MUST be a JSON object whose "applications" field is an array of strings, where each string is the URI of an authorization belonging to this registration. The server SHOULD include pending applications, and SHOULD NOT include applications that are invalid. 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.


```
{
  "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"
}
```

6.1.3. Application Objects

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

status (required, string): The status of this authorization. Possible values are: "unknown", "pending", "processing", "valid", and "invalid".

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

csr (required, string): A CSR encoding the parameters for the certificate being requested [RFC2986]. The CSR is sent in the Base64url-encoded version of the DER format. (Note: 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]

requirements (required, array): The requirements that the client needs to fulfill before the requested certificate can be granted (for pending applications). For final applications, the requirements that were met. Each entry is a dictionary with parameters describing the requirement (see below).

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

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

  "csr": "jcRf4uXra7FGYW5ZMewvV...rhlnznwy8YbpMGqwidEXfE",
  "notBefore": "2016-01-01T00:00:00Z",
  "notAfter": "2016-01-08T00:00:00Z",

  "requirements": [
    {
      "type": "authorization",
      "status": "valid",
      "url": "https://example.com/acme/authz/1234"
    },
    {
      "type": "out-of-band",
      "status": "pending",
      "url": "https://example.com/acme/payment/1234"
    }
  ]

  "certificate": "https://example.com/acme/cert/1234"
}
```


[[Open issue: There are two possible behaviors for the CA here. Either (a) the CA automatically issues once all the requirements are fulfilled, or (b) the CA waits for confirmation from the client that it should issue. If we allow both, we will need a signal in the application object of whether confirmation is required. I would prefer that auto-issue be the default, which would imply a syntax like "confirm": true]]

[[Open issue: Should this syntax allow multiple certificates? That would support reissuance / renewal in a straightforward way, especially if the CSR / notBefore / notAfter could be updated.]]

The elements of the "requirements" array are immutable once set, except for their "status" fields. If any other part of the object changes after the object is created, the client MUST consider the application invalid.

The "requirements" array in the challenge SHOULD reflect everything that the CA required the client to do before issuance, even if some requirements were fulfilled in earlier applications. For example, if a CA allows multiple applications to be fufilled based on a single authorization transaction, then it must reflect that authorization in all of the applications.

Each entry in the "requirements" array expresses a requirement from the CA for the client to takek a particular action. All requirements objects have the following basic fields:

type (required, string): The type of requirement (see below for defined types)

status (required, string): The status of this requirement. Possible values are: "pending", "valid", and "invalid".

All additional fields are specified by the requirement type.

6.1.3.1. Authorization Requirement

A requirement with type "authorization" requests that the ACME client complete an authorization transaction. The server specifies the authorization by pre-provisioning a pending authorization resource and providing the URI for this resource in the requirement.

url (required, string): The URL for the authorization resource

To fulfill this requirement, the ACME client should fetch the authorization object from the indicated URL, then follow the process for obtaining authorization as specified in Section 6.4.

6.1.3.2. Out-of-Band Requirement

A requirement with type "out-of-band" requests that the ACME client have a human user visit a web page in order to receive further instructions for how to fulfill the requirement. The requirement object provides a URI for the web page to be visited.

url (required, string): The URL to be visited. The scheme of this URL MUST be "http" or "https"

To fulfill this requirement, the ACME client should direct the user to the indicated web page.

6.1.4. 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.

scope (optional, string): If this field is present, then it MUST contain a URI for an application resource, such that this authorization is only valid for that resource. If this field is absent, then the CA MUST consider this authorization valid for all applications until the authorization expires. [[Open issue: More flexible scoping?]]

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.

```
{
  "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. 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.

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

{
  "protected": base64url({
    "alg": "ES256",
    "jwk": {...},
    "nonce": "6S8IqOGY7eL2lsGoTZYifg",
    "url": "https://example.com/acme/new-reg"
  })
  "payload": base64url({
    "contact": [
      "mailto:cert-admin@example.com",
      "tel:+12025551212"
    ]
  }),
  "signature": "RZPOnYoPs1PhjszF...-nh6X1qtOFPB519I"
}
```

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.

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/terms>;rel="terms-of-service"
Link: <https://example.com/acme/some-directory>;rel="directory"

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

  "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
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "jwk": {...},
    "nonce": "ax5RnthDqp_Yf4_HZnFLmA",
    "url": "https://example.com/acme/reg/asdf"
  })
  "payload": base64url({
    "contact": [
      "mailto:certificates@example.com",
      "tel:+12125551212"
    ]
  }),
  "signature": "hDXzvcj8T6fbFbmn...rDzXzzvzpRy64N0o"
}
```

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 ({}).

6.2.1. Account Key Roll-over

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

To change the key associate with an account, the client sends a POST request containing a key-change object with the following fields:

oldKey (required, JWK): The JWK representation of the original key (i.e., the client's current account key)

newKey (requrired, JWK): The JWK representation of the new key

The JWS of this POST must have two signatures: one signature from the existing key on the account, and one signature from the new key that the client proposes to use. This demonstrates that the client actually has control of the private key corresponding to the new public key. The protected header must contain a JWK field containing the current account key.

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

{
  "payload": base64url({
    "oldKey": /* Old key in JWK form */
    "newKey": /* New key in JWK form */
  }),
  "signatures": [{
    "protected": base64url({
      "alg": "ES256",
      "jwk": /* old key */,
      "nonce": "pq00v-D1KB0sReG4jFfqVg",
      "url": "https://example.com/acme/key-change"
    }),
    "signature": "XFvVbo9diBlIBvhE...UI62sNT6MZsCJpQo"
  }, {
    "protected": base64url({
      "alg": "ES256",
      "jwk": /* new key */,
      "nonce": "vYjyueEYhMjpVQHe_unw4g",
      "url": "https://example.com/acme/key-change"
    }),
    "signature": "q20gG1f1r9cD6tBM...a48h0CkP11tl5Doo"
  }]
}
```

On receiving key-change request, the server MUST perform the following steps in addition to the typical JWS validation:

1. Check that the JWS protected header container a "jwk" field containing a key that matches a currently active account.

2. Check that there are exactly two signatures on the JWS.

3. Check that one of the signatures validates using the account key from (1).

4. Check that the "key" field contains a well-formed JWK that meets key strength requirements.

5. Check that the "key" field is not equivalent to the current account key or any other currently active account key.

6. Check that one of the two signatures on the JWS validates using the JWK from the "key" field.

If all of these checks pass, then the server updates the corresponding registration by replacing the old account key with the new public key and returns status code 200. Otherwise, the server responds with an error status code and a problem document describing the error.

6.2.2. Account deactivation

A client may deactivate an account by posting a signed update to the server with a status field of "deactivated." Clients may wish to do this when the account key is compromised.

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

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

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

Once an account is deactivated, the server MUST NOT accept further requests authorized by that account's key. It is up to server policy how long to retain data related to that account, whether to revoke certificates issued by that account, and whether to send email to that account's contacts. ACME does not provide a way to reactivate a deactivated account.

6.3. Applying for Certificate Issuance

The holder of an account key pair may use ACME to submit an application for a certificate to be issued. The client makes this request by sending a POST request to the server's new-application resource. The body of the POST is a JWS object whose JSON payload is a subset of the application object defined in Section 6.1.3, containing the fields that describe the certificate to be issued:

csr (required, string): A CSR encoding the parameters for the certificate being requested [RFC2986]. The CSR is sent in the Base64url-encoded version of the DER format. (Note: 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-app HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "jwk": {...},
    "nonce": "5XJ1L3lEkMG7tR6pA00clA",
    "url": "https://example.com/acme/new-app"
  })
  "payload": base64url({
    "csr": "5jNudRx6Ye4HzKEqT5...FS6aKdZeGsysoCo4H9P",
    "notBefore": "2016-01-01T00:00:00Z",
    "notAfter": "2016-01-08T00:00:00Z"
  }),
  "signature": "H6ZXtGjTZyUnPeKn...wEA4TklBdh3e454g"
}
```

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 server MUST return an error if it cannot fulfil the request as specified, and MUST NOT issue a certificate with contents other than

those requested. If the server requires the request to be modified in a certain way, it should indicate the required changes using an appropriate error code and description.

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

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

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

  "csr": "jcRf4uXra7FGYW5ZMewvV...rhlnznwy8YbpMGqwidEXfE",
  "notBefore": "2016-01-01T00:00:00Z",
  "notAfter": "2016-01-08T00:00:00Z",

  "requirements": [
    {
      "type": "authorization",
      "status": "valid",
      "url": "https://example.com/acme/authz/1234"
    },
    {
      "type": "out-of-band",
      "status": "pending",
      "url": "https://example.com/acme/payment/1234"
    }
  ]
}
```

The application object returned by the server represents a promise that if the client fulfills the server's requirements before the "expires" time, then the server will issue the requested certificate. In the application object, any object in the "requirements" array whose status is "pending" represents an action that the client must perform before the server will issue the certificate. If the client fails to complete the required actions before the "expires" time, then the server SHOULD change the status of the application to "invalid" and MAY delete the application resource.

The server SHOULD issue the requested certificate and update the application resource with a URL for the certificate as soon as the client has fulfilled the server's requirements. If the client has

already satisfied the server's requirements at the time of this request (e.g., by obtaining authorization for all of the identifiers in the certificate in previous transactions), then the server MAY proactively issue the requested certificate and provide a URL for it in the "certificate" field of the application. The server MUST, however, still list the satisfied requirements in the "requirements" array, with the state "valid".

Once the client believes it has fulfilled the server's requirements, it should send a GET request to the application resource to obtain its current state. The status of the application will indicate what action the client should take:

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

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

- "processing": The server agrees that the requirements have been fulfilled, and is in the process of generating the certificate. Retry after the time given in the "Retry-After" header field of the response, if any.

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

6.3.1. Downloading the Certificate

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

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. For example, the client may use the media type application/x-pem-file to request the certificate in PEM format.

The server provides metadata about the certificate in HTTP headers. In particular, the server MUST send one or more link relation header fields [RFC5988] with relation "up", each indicating a single certificate resource for the issuer of this certificate. The server MAY also include the "up" links from these resources to enable the client to build a full certificate chain.

The server MUST also provide a link relation header field with relation "author" to indicate the application under which this certificate was issued.

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/app/asdf>;rel="author"
Link: <https://example.com/acme/sct/asdf>;rel="ct-sct"
Link: <https://example.com/acme/some-directory>;rel="directory"

[DER-encoded certificate]
```

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

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

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). The server may declare that an

authorization is only valid for a specific application by setting the "scope" field of the authorization to the URI for that application.

Authorization resources are created by the server in response to certificate applications submitted by an account key holder; their URLs are provided to the client in "authorization" requirement objects. The authorization object is implicitly tied to the account key used to sign the new-application request.

When a client receives an application from the server with an "authorization" requirement, it downloads the authorization resource by sending a GET request to the indicated URL.

```
GET /acme/authz/1234 HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
Content-Type: application/json
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 identifier 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). 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
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "jwk": {...},
    "nonce": "Q_s3MWoqT05TrdkM2MTDcw",
    "url": "https://example.com/acme/authz/asdf/0"
  })
  "payload": base64url({
    "type": "http-01",
    "keyAuthorization": "IlirfxKKXA...vb29HhjjLPSggwiE"
  }),
  "signature": "9cbg5JO1Gf5YLjjz...SpkUfcdPai9uVYYQ"
}
```

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. Deactivating an Authorization

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

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

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

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

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

6.5. 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.)

reason (optional, int): One of the revocation reasonCodes defined in RFC 5280 [RFC5280] Section 5.3.1 to be used when generating OCSP responses and CRLs. If this field is not set the server SHOULD use the unspecified (0) reasonCode value when generating OCSP responses and CRLs. The server MAY disallow a subset of reasonCodes from being used by the user.

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

{
  "protected": base64url({
    "alg": "ES256",
    "jwk": {...},
    "nonce": "JHb54aT_KTXBWQOzGYkt9A",
    "url": "https://example.com/acme/revoke-cert"
  })
  "payload": base64url({
    "certificate": "MIIEDTCCAvegAwIBAgIRAP8...",
    "reason": 1
  }),
  "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
}
```

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:

- o Hold the private key of the account key pair used to respond to the challenge

- o Control the identifier in question

Section 9 documents how the challenges defined in this document meet these requirements. New challenges will need to document how they do.

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

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 [RFC7807].

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, at its discretion. Because many webservers 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.

```
/* BEGIN JWS-signed content */
{
  "keyAuthorization": "evaGxfADs...62jcerQ"
}
/* END JWS-signed content */
```

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. Dereference the URI using an HTTP GET request.

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

5. 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.

```
/* BEGIN JWS-signed content */
{
  "keyAuthorization": "evaGxfADs...62jcerQ"
}
/* END JWS-signed content */
```

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.

```
/* BEGIN JWS-signed content */
{
  "keyAuthorization": "evaGxfADs...62jcerQ"
}
/* END JWS-signed content */
```

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.

7.5. Out-of-Band

There may be cases where a server cannot perform automated validation of an identifier, for example if validation requires some manual steps. In such cases, the server may provide an "out of band" (OOB) challenge to request that the client perform some action outside of ACME in order to validate possession of the identifier.

The OOB challenge requests that the client have a human user visit a web page to receive instructions on how to validate possession of the identifier, by providing a URL for that web page.

type (required, string): The string "oob-01"

url (required, string): The URL to be visited. The scheme of this URL MUST be "http" or "https"

```
{
  "type": "oob-01",
  "url": "https://example.com/validate/evaGxfADs6pSRb2LAv9IZ"
}
```

A client responds to this challenge by presenting the indicated URL for a human user to navigate to. If the user choses to complete this challege (by vising the website and completing its instructions), the client indicates this by sending a simple acknowledgement response to the server.

type (required, string): The string "oob-01"

```
/* BEGIN JWS-signed content */
{
  "type": "oob-01"
}
/* END JWS-signed content */
```

On receiving a response, the server MUST verify that the value of the "type" field is as required. Otherwise, the steps the server takes to validate identifier possession are determined by the server's local policy.

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?]]

8.1. 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

8.2. 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 |

8.3. "url" JWS Header Parameter

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

- Header Parameter Name: "url"
- Header Parameter Description: URL
- Header Parameter Usage Location(s): JWE, JWS
- Change Controller: IESG
- Specification Document(s): Section 5.3.1 of RFC XXXX

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

8.4. "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]]

8.5. 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.]]

8.6. New Registries

This document requests that IANA create the following new registries:

1. ACME Error Codes

2. ACME Resource Types

3. ACME Identifier Types

4. ACME Challenge Types

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

8.6.1. 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 5.6 above, with the Reference field set to point to this specification.

8.6.2. Resource Types

This registry lists the types of resources that ACME servers may list in their directory objects.

Template:

- Key: The value to be used as a dictionary key in the directory object

- Resource type: The type of resource labeled by the key

- Reference: Where the identifier type is defined

Initial contents:

Key	Resource type	Reference
new-reg	New registration	RFC XXXX
new-app	New application	RFC XXXX
revoke-cert	Revoke certificate	RFC XXXX
key-change	Key change	RFC XXXX

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

8.6.3. 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]]

8.6.4. 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:

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

9. 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.

9.1. 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

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

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

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

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).

9.2. Integrity of Authorizations

ACME allows anyone to request challenges for an identifier by registering an account key and sending a new-application 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-application 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

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

o 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 have a binding between the account private key and the validation query made by the server, via the key authorization. The key authorization is signed by the account private key, reflects the corresponding public key, and is provided to the server in the validation response.

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:

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

o TLS SNI: Configure a TLS server for the domain

o DNS: Provision DNS resource records for the domain

There are several ways that these assumptions can be violated, both by misconfiguration and by 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 a different domain. Then the target of the redirect could use that to get a certificate through HTTP validation, since the validation path will not be known to the primary server.

The DNS is a common point of vulnerability for all of these challenges. An entity that can provision false DNS records for a domain can attack the DNS challenge directly, and can provision false A/AAAA records to direct the ACME server to send its 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.

9.3. 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.

9.4. Server-Side Request Forgery

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

Some server implementations include information from the validation server's response (in order to facilitate debugging). Such implementations enable an attacker to extract this information from any web server that is accessible to the ACME server, even if it is not accessible to the ACME client.

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

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

9.5. CA Policy Considerations

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

- o Has the client agreed to a subscriber agreement?
- o Is the claimed identifier syntactically valid?
- o 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?
- o Is the key in the CSR sufficiently strong?
- o 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.

10. 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:

10.1. DNS over TCP

As noted above, DNS forgery attacks against the ACME server can result in the server making incorrect decisions about domain control and thus mis-issuing certificates. Servers SHOULD verify DNSSEC when it is available for a domain. When DNSSEC is not available, servers SHOULD perform DNS queries over TCP, which provides better resistance to some forgery attacks than DNS over UDP.

10.2. 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 comes 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").

10.3. 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.).

11. 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.

o Peter Eckersley, EFF

o Eric Rescorla, Mozilla

o Seth Schoen, EFF

o Alex Halderman, University of Michigan

o Martin Thomson, Mozilla

o 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.

12. References

12.1. Normative References

[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>.

[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>.

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

[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>.

[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>.

[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>.

[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>.

[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>.

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

12.2. 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>.

[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>.

[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>.

[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>.

[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>.

[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>.

[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>.

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