Automatic Certificate Management Environment (ACME)
draft-ietf-acme-acme-08

Abstract

   Certificates in PKI using X.509 (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 certification authority (CA) and
   an applicant can use to automate the process of verification and
   certificate issuance. The protocol also provides facilities for
   other certificate management functions, such as certificate
   revocation.

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

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

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

This Internet-Draft will expire on May 3, 2018.

Copyright Notice

Table of Contents

1. Introduction

   Certificates [RFC5280] in the Web PKI 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.

   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.)

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

   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
         email 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 Transport Layer Security (TLS) [RFC5246].

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

2. Deployment Model and Operator Experience

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

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

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

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

o The ACME client presents the operator with a list of CAs from
   which it could get a certificate. (This list will change over
   time based on the capabilities of CAs and updates to ACME

      configuration.) The ACME client might prompt the operator for
      payment information at this point.

   o The operator selects a CA.

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

   o The CA verifies that the client controls the requested domain
      name(s).

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

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

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

3. Terminology

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "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 may run on a web
   server, mail server, or some other server system which requires valid
   TLS certificates. Or, it may run on a separate server that does not
   consume the certificate, but is authorized to respond to a CA-
   provided challenge. The ACME server runs at a certification
   authority, and responds to client requests, performing the requested
   actions if the client is authorized.

   An ACME client 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 JavaScript Object Notation (JSON) messages carried over
   HTTPS. In many ways, ACME functions much like a traditional CA, in
   which a user creates an account, requests a certificate, and proves
   control of the domains in that certificate in order for the CA to
   sign the requested certificate.

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

```
      Client                                                Server

      Contact Information
      ToS Agreement
      Additional Data
      Signature                        ------->

                                       <-------               Account
```

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

   1. Submit an order for a certificate to be issued

   2. Prove control of any identifiers requested in the certificate

   3. Await issuance and download the issued certificate

   The client's order 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 them.

Once the validation process is complete and the server is satisfied
that the client has met its requirements, the server will issue the
requested certificate and make it available to the client.

```
      Order
      Signature                      ------->
                                                      Required
                               <-------            Authorizations

      Responses
      Signature                      ------->

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

                               <-------              Certificate
```

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

```
      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 8 below address validation of domain
names. The use of ACME for other identifiers will require further
specification in order to describe how these identifiers are encoded
in the protocol and what types of validation challenges the server
might require.

5. Character Encoding

All requests and responses sent via HTTP by ACME clients, ACME
servers, and validation servers as well as any inputs for digest
computations MUST be encoded using the UTF-8 [RFC3629] character set.

6. Message Transport

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

6.1. HTTPS Requests

   Each ACME function is accomplished by the client sending a sequence
   of HTTPS requests to the server, carrying JSON messages
   [RFC2818][RFC7159]. Use of HTTPS is REQUIRED. Each subsection of
   Section 7 below describes the message formats used by the function
   and the order in which messages are sent.

   In most HTTPS transactions used by ACME, the ACME client is the HTTPS
   client and the ACME server is the HTTPS server. The ACME server acts
   as an HTTP and HTTPS client when validating challenges via HTTP.

   ACME servers SHOULD follow the recommendations of [RFC7525] when
   configuring their TLS implementations. ACME servers that support TLS
   1.3 MAY allow clients to send early data (0xRTT). This is safe
   because the ACME protocol itself includes anti-replay protections.

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

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

   ACME servers that are intended to be generally accessible need to use
   Cross-Origin Resource Sharing (CORS) in order to be accessible from
   browser-based clients [W3C.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.

6.2. Request Authentication

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

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

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

   o The JWS MUST NOT have a Message Authentication Code (MAC)-based
      algorithm in its "alg" field

   o The JWS Protected Header MUST include the following fields:

      * "alg" (Algorithm)

      * "jwk" (JSON Web Key, only for requests to new-account and
         revoke-cert resources)

      * "kid" (Key ID, for all other requests)

      * "nonce" (defined in Section 6.4 below)

      * "url" (defined in Section 6.3 below)

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

   For new-account requests, and for revoke-cert requests authenticated
   by certificate key, there MUST be a "jwk" field.

   For all other requests, there MUST be a "kid" field. This field must
   contain the account URL received by POSTing to the new-account
   resource.

   Note that authentication via signed JWS request bodies implies that
   GET requests are not authenticated. Servers MUST NOT respond to GET
   requests for resources that might be considered sensitive. Account
   resources are the only sensitive resources defined in this
   specification.

   If the client sends a JWS signed with an algorithm that the server
   does not support, then the server MUST return an error with status
   code 400 (Bad Request) and type

"urn:ietf:params:acme:error:badSignatureAlgorithm". The problem
document returned with the error MUST include an "algorithms" field
with an array of supported "alg" values.

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.

6.3. Request URL Integrity

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

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

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

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

6.3.1. "url" (URL) JWS header parameter

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

6.4. 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 provides nonces to clients using the Replay-Nonce
   header field, as specified in Section 6.4.1 below. The server MUST
   include a Replay-Nonce header field in every successful response to a
   POST request and SHOULD provide it in error responses as well.

   Every JWS sent by an ACME client MUST include, in its protected
   header, the "nonce" header parameter, with contents as defined in
   Section 6.4.2 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 MUST provide HTTP status code 400
   (Bad Request), and indicate the ACME error type
   "urn:ietf:params:acme:error:badNonce". An error response with the
   "badNonce" error type MUST include a Replay-Nonce header with a fresh
   nonce. On receiving such a response, a client SHOULD retry the
   request using the new nonce.

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

6.4.1. 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 MUST generate the value provided in
   Replay-Nonce in such a way that they are unique to each message, with
   high probability. For instance, it is acceptable to generate Replay-
   Nonces randomly.

   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.

6.4.2. "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.

6.5. 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 type
"urn:ietf:params:acme:error: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 "urn:ietf:params:acme:documentation" relation.

6.6. Errors

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

When the server responds with an error status, it SHOULD provide
additional information using a problem document [RFC7807]. To
facilitate automatic response to errors, this document defines the

following standard tokens for use in the "type" field (within the
"urn:ietf:params:acme:error:" namespace):

```
+------------------------+-------------------------------------------+
| Type                   | Description                               |
+------------------------+-------------------------------------------+
| badCSR                 | The CSR is unacceptable (e.g., due to a   |
|                        | short key)                                |
|                        |                                           |
| badNonce               | The client sent an unacceptable anti-     |
|                        | replay nonce                              |
|                        |                                           |
| badSignatureAlgorithm  | The JWS was signed with an algorithm the  |
|                        | server does not support                   |
|                        |                                           |
| invalidContact         | A contact URL for an account was invalid  |
|                        |                                           |
| unsupportedContact     | A contact URL for an account used an      |
|                        | unsupported protocol scheme               |
|                        |                                           |
| accountDoesNotExist    | The request specified an account that     |
|                        | does not exist                            |
|                        |                                           |
| malformed              | The request message was malformed         |
|                        |                                           |
| rateLimited            | The request exceeds a rate limit          |
|                        |                                           |
| rejectedIdentifier     | The server will not issue for the         |
|                        | identifier                                |
|                        |                                           |
| serverInternal         | The server experienced an internal error  |
|                        |                                           |
| unauthorized           | The client lacks sufficient authorization |
|                        |                                           |
| unsupportedIdentifier  | Identifier is not supported, but may be   |
|                        | in future                                 |
|                        |                                           |
| userActionRequired     | Visit the "instance" URL and take actions |
|                        | specified there                           |
|                        |                                           |
| badRevocationReason    | The revocation reason provided is not     |
|                        | allowed by the server                     |
|                        |                                           |
| caa                    | Certification Authority Authorization     |
|                        | (CAA) records forbid the CA from issuing  |
|                        |                                           |
| dns                    | There was a problem with a DNS query      |
|                        |                                           |
```

```
| connection            | The server could not connect to          |
|                       | validation target                        |
|                       |                                          |
| tls                   | The server received a TLS error during   |
|                       | validation                               |
|                       |                                          |
| incorrectResponse     | Response received didn't match the       |
|                       | challenge's requirements                 |
+-----------------------+------------------------------------------+
```

   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 [RFC3553] namespace for errors other than
   the standard types. Clients SHOULD display the "detail" field of all
   errors.

7. Certificate Management

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

   o Account Creation

   o Ordering a Certificate

   o Identifier Authorization

   o Certificate Issuance

   o Certificate Revocation

7.1. Resources

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

   o Account resources, representing information about an account
      (Section 7.1.2, Section 7.3)

   o Order resources, representing an account's requests to issue
      certificates (Section 7.1.3)

   o Authorization resources, representing an account's authorization
      to act for an identifier (Section 7.1.4)

   o Challenge resources, representing a challenge to prove control of
      an identifier (Section 7.5, Section 8)

o Certificate resources, representing issued certificates
   (Section 7.4.2)

o A "directory" resource (Section 7.1.1)

o A "new-nonce" resource (Section 7.2)

o A "new-account" resource (Section 7.3)

o A "new-order" resource (Section 7.4)

o A "revoke-cert" resource (Section 7.6)

o A "key-change" resource (Section 7.3.6)

The server MUST provide "directory" and "new-nonce" resources.

ACME uses different URLs for different management functions. Each
function is listed in a directory along with its corresponding URL,
so clients only need to be configured with the directory URL. These
URLs are connected by a few different link relations [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 "index" link relation is present on all resources other than the
directory and indicates the URL of the directory.

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

```
                          directory
                              |
                              |--> new-nonce
                              |
        -------------------------------+
        |          |          |        |
        |          |          |        |
        V          V          V        V
   new-account new-authz  new-order  revoke-cert
        |          |          |
        |          |          |
        V          |          V
     account       |        order -----> cert
                   |          |
                   |          |
                   |          V
              +------> authz
                      | ^
                      | | "up"
                      V |
                    challenge
```
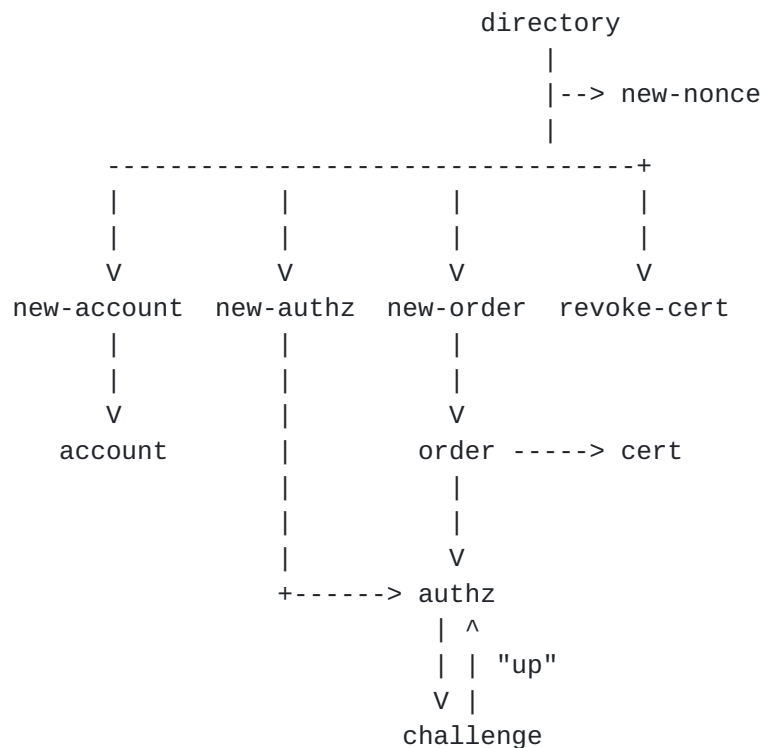
   The following table illustrates a typical sequence of requests
   required to establish a new account with the server, prove control of
   an identifier, issue a certificate, and fetch an updated certificate
   some time after issuance. The "->" is a mnemonic for a Location
   header pointing to a created resource.

Action	Request	Response
Get a nonce	HEAD new-nonce	204
Create account	POST new-account	201 -> account
Submit an order	POST new-order	201 -> order
Fetch challenges	GET authz	200
Respond to challenge	POST challenge	200
Poll for status	GET authz	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.

7.1.1. Directory

   In order to help clients configure themselves with the right URLs for
   each ACME operation, ACME servers provide a directory object. This
   should be the only URL needed to configure clients. It is a JSON
   object, whose fields names are drawn from the following table and
   whose values are the corresponding URLs.

```
               +-------------+--------------------+
               | Field       | URL in value       |
               +-------------+--------------------+
               | new-nonce   | New nonce          |
               |             |                    |
               | new-account | New account        |
               |             |                    |
               | new-order   | New order          |
               |             |                    |
               | new-authz   | New authorization  |
               |             |                    |
               | revoke-cert | Revoke certificate |
               |             |                    |
               | key-change  | Key change         |
               +-------------+--------------------+
```

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

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

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

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

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

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

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

"caa-identities" (optional, array of string): Each string MUST be a
   lowercase hostname which the ACME server recognizes 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 records.

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

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

{
  "new-nonce": "https://example.com/acme/new-nonce",
  "new-account": "https://example.com/acme/new-account",
  "new-order": "https://example.com/acme/new-order",
  "new-authz": "https://example.com/acme/new-authz",
  "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/2017-5-30",
    "website": "https://www.example.com/",
    "caa-identities": ["example.com"]
  }
}
```

7.1.2. Account Objects

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

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

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

terms-of-service-agreed (optional, boolean): Including this field in
   a new-account request, with a value of true, indicates the
   client's agreement with the terms of service. This field is not
   updateable by the client.

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

```
   {
     "status": "valid",
     "contact": [
       "mailto:cert-admin@example.com",
       "tel:+12025551212"
     ],
     "terms-of-service-agreed": true,
     "orders": "https://example.com/acme/acct/1/orders"
   }
```

7.1.2.1. Orders List

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

```
   HTTP/1.1 200 OK
   Content-Type: application/json
   Link: href="/acme/acct/1/orders?cursor=2", rel="next"

   {
     "orders": [
       "https://example.com/acme/acct/1/order/1",
       "https://example.com/acme/acct/1/order/2",
       /* 47 more URLs not shown for example brevity */
       "https://example.com/acme/acct/1/order/50"
     ]
   }
```

7.1.3. Order Objects

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

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

expires (optional, string): The timestamp after which the server
   will consider this order 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: Because this
   field uses base64url, and does not include headers, 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].

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

authorizations (required, array of string): For pending orders, the
   authorizations that the client needs to complete before the
   requested certificate can be issued (see Section 7.5). For final
   orders, the authorizations that were completed. Each entry is a
   URL from which an authorization can be fetched with a GET request.

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

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

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

  "authorizations": [
    "https://example.com/acme/authz/1234",
    "https://example.com/acme/authz/2345"
  ],

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

The elements of the "authorizations" array are immutable once set.
The server MUST NOT change the contents of the "authorizations" array

after it is created. If a client observes a change in the contents
of the "authorizations" array, then it SHOULD consider the order
invalid.

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

7.1.4. Authorization Objects

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

The structure of an ACME authorization resource is as follows:

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

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

   value (required, string): The identifier itself.

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

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 URL for an order 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 orders until
   the authorization expires.

challenges (required, array of objects): For pending authorizations,
   the challenges that the client can fulfill in order to prove
   possession of the identifier. For final authorizations, the
   challenges that were used. Each array entry is an object with

parameters required to validate the challenge. A client should
attempt to fulfill one of these challenges, and a server should
consider any one of the challenges sufficient to make the
authorization valid. For final authorizations, it contains the
challenges that were successfully completed.

The only type of identifier defined by this specification is a fully-
qualified domain name (type: "dns"). If a domain name contains non-
ASCII Unicode characters it MUST be encoded using the rules defined
in [RFC3492]. Servers MUST verify any identifier values that begin
with the ASCII Compatible Encoding prefix "xn-" as defined in
[RFC5890] are properly encoded. Wildcard domain names (with "*" as
the first label) MUST NOT be included in authorization objects.

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

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

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

  "challenges": [
    {
      "url": "https://example.com/authz/1234/0",
      "type": "http-01",
      "status": "valid",
      "token": "DGyRejmCefe7v4NfDGDKfA"
      "validated": "2014-12-01T12:05:00Z",
      "keyAuthorization": "SXQe-2XODaDxNR...vb29HhjjLPSggwiE"
    }
  ]
}
```

7.2. Getting a Nonce

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

To get a fresh nonce, the client sends a HEAD request to the new-
nonce resource on the server. The server's response MUST include a

   Replay-Nonce header field containing a fresh nonce, and SHOULD have
   status code 204 (No Content). The server SHOULD also respond to GET
   requests for this resource, returning an empty body (while still
   providing a Replay-Nonce header) with a 204 (No Content) status.

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

   HTTP/1.1 204 No Content
   Replay-Nonce: oFvnlFP1wIhRlYS2jTaXbA
   Cache-Control: no-store

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

7.3. Account Creation

   A client creates a new account with the server by sending a POST
   request to the server's new-account URL. The body of the request is
   a stub account object containing the "contact" field and optionally
   the "terms-of-service-agreed" field.

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

   terms-of-service-agreed (optional, boolean): Same meaning as the
      corresponding server field defined in Section 7.1.2

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

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

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

The server MUST ignore any values provided in the "orders" fields in
account 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 can be
provided by the client.

In general, the server MUST ignore any fields in the request object
that it does not recognize. In particular, it MUST NOT reflect
unrecognized fields in the resulting account object. This allows
clients to detect when servers do not support an extension field.

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

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

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

   The server creates an account and stores the public key used to
   verify the JWS (i.e., the "jwk" element of the JWS header) to
   authenticate future requests from the account. The server returns
   this account object in a 201 (Created) response, with the account URL
   in a Location header field.

   HTTP/1.1 201 Created
   Content-Type: application/json
   Replay-Nonce: D8s4D2mLs8Vn-goWuPQeKA
   Location: https://example.com/acme/acct/1
   Link: <https://example.com/acme/some-directory>;rel="index"

   {
     "status": "valid",

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

7.3.1. Finding an Account URL Given a Key

   If the server already has an account registered with the provided
   account key, then it MUST return a response with a 200 (OK) status
   code and provide the URL of that account in the Location header
   field. This allows a client that has an account key but not the
   corresponding account URL to recover the account URL.

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

7.3.2. Account Update

   If the client wishes to update this information in the future, it
   sends a POST request with updated information to the account URL.
   The server MUST ignore any updates to "order" fields or any other
   fields it does not recognize. If the server accepts the update, it
   MUST return a response with a 200 (OK) status code and the resulting
   account object.

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

```
POST /acme/acct/1 HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "ax5RnthDqp_Yf4_HZnFLmA",
    "url": "https://example.com/acme/acct/1"
  }),
  "payload": base64url({
    "contact": [
      "mailto:certificates@example.com",
      "tel:+12125551212"
    ]
  }),
  "signature": "hDXzvcj8T6fbFbmn...rDzXzzvzpRy64N0o"
}
```

7.3.3. Account Information

   Servers SHOULD NOT respond to GET requests for account 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 an empty object ({}).

7.3.4. Changes of Terms of Service

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

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

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

```
HTTP/1.1 403 Forbidden
Replay-Nonce: IXVHDyxIRGcTE0VSblhPzw
Link: <https://example.com/acme/terms/2017-6-02>;rel="terms-of-service"
Content-Type: application/problem+json
Content-Language: en

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

7.3.5. External Account Binding

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

To enable ACME account binding, a CA needs to provision the ACME
client with a MAC key and a key identifier. The key identifier MUST
be an ASCII string. The MAC key SHOULD be provided in base64url-
encoded form, to maximize compatibility between provisioning systems
and ACME clients.

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

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

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

o The "nonce" field MUST NOT be present

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

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

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

{
  "protected": base64url({
    "alg": "ES256",
    "jwk": /* account key */,
    "nonce": "K60BWPrMQG9SDxBDS_xtSw",
    "url": "https://example.com/acme/new-account"
  }),
  "payload": base64url({
    "contact": ["mailto:example@anonymous.invalid"],
    "terms-of-service-agreed": true,

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

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

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

   2. Verify that the JWS protected meets the above criteria

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

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

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

If all of these checks pass and the CA creates a new account, then
the CA may consider the new account associated with the external
account corresponding to the MAC key and MUST reflect value of the
"external-account-binding" field in the resulting account object. If
any of these checks fail, then the CA MUST reject the new-account
request.

7.3.6. Account Key Roll-over

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

To change the key associated with an account, the client first
constructs a key-change object describing the change that it would
like the server to make:

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

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

The client then encapsulates the key-change object in a JWS, signed
with the requested new account key (i.e., the key matching the
"newKey" value).

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

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

o The inner JWS is NOT REQUIRED to have a "nonce" header parameter.
   The server MUST ignore any value provided for the "nonce" header
   parameter.

This transaction has signatures from both the old and new keys so
that the server can verify that the holders of the two keys both
agree to the change. The signatures are nested to preserve the
property that all signatures on POST messages are signed by exactly
one key.

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

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "K60BWPrMQG9SDxBDS_xtSw",
    "url": "https://example.com/acme/key-change"
  }),
  "payload": base64url({
    "protected": base64url({
      "alg": "ES256",
      "jwk": /* new key */,
      "url": "https://example.com/acme/key-change"
    }),
    "payload": base64url({
      "account": "https://example.com/acme/acct/1",
      "newKey": /* new key */
    }),
    "signature": "Xe8B94RD30Azj2ea...8BmZIRtcSKPSd8gU"
  }),
  "signature": "5TWiqIYQfIDfALQv...x9C2mg8JGPxl5bI4"
}
```

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

1. Validate the POST request belongs to a currently active account,
    as described in [Section 6](#).

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

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

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

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

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

7. Check that the "account" field of the key-change object contains
    the URL for the account matching the old key

8. Check that the "newKey" field of the key-change object also
    verifies the inner JWS.

9. Check that no account exists whose account key is the same as the
    key in the "newKey" field.

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

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

7.3.7. Account Deactivation

A client can 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 or decommissioned.

```
POST /acme/acct/1 HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "ntuJWWSic4WVNSqeUmshgg",
    "url": "https://example.com/acme/acct/1"
  }),
  "payload": base64url({
    "status": "deactivated"
  }),
  "signature": "earzVLd3m5M4xJzR...bVTqn7R08AKOVf3Y"
}
```

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

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

7.4. Applying for Certificate Issuance

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

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

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

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

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

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "5XJ1L3lEkMG7tR6pA00clA",
    "url": "https://example.com/acme/new-order"
  }),
  "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 fulfill 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 type and description.

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

```
HTTP/1.1 201 Created
Replay-Nonce: MYAuvOpaoIiywTezizk5vw
Location: https://example.com/acme/order/asdf

{
  "status": "pending",
  "expires": "2016-01-01T00:00:00Z",

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

  "authorizations": [
    "https://example.com/acme/authz/1234",
    "https://example.com/acme/authz/2345"
  ]
}
```

The order object returned by the server represents a promise that if
the client fulfills the server's requirements before the "expires"
time, then the server will issue the requested certificate. In the
order object, any authorization referenced in the "authorizations"
array whose status is "pending" represents an authorization
transaction that the client must complete before the server will
issue the certificate (see Section 7.5). If the client fails to
complete the required actions before the "expires" time, then the
server SHOULD change the status of the order to "invalid" and MAY
delete the order resource.

The server MUST begin the issuance process for the requested
certificate and update the order resource with a URL for the
certificate once 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 MUST proactively issue the requested certificate and provide a
URL for it in the "certificate" field of the order. The server MUST,
however, still list the completed authorizations in the
"authorizations" array.

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

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

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

o "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.

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

7.4.1. Pre-Authorization

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

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

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

To request authorization for an identifier, the client sends a POST
request to the new-authorization resource specifying the identifier
for which authorization is being requested and how the server should
behave with respect to existing authorizations for this identifier.

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

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

   value (required, string): The identifier itself.

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

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "uQpSjlRb4vQVCjVYAyyUWg",
    "url": "https://example.com/acme/new-authz"
  }),
  "payload": base64url({
    "identifier": {
      "type": "dns",
      "value": "example.net"
    }
  }),
  "signature": "nuSDISbWG8mMgE7H...QyVUL68yzf3Zawps"
}
```

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

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

o "identifier" the identifier submitted by the client

o "status" MUST be "pending" unless the server has out-of-band
   information about the client's authorization status

o "challenges" as selected by the server's policy for this
   identifier

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

7.4.2. Downloading the Certificate

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

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

   The server MAY provide one or more link relation header fields
   [RFC5988] with relation "alternate". Each such field SHOULD express
   an alternative certificate chain starting with the same end-entity
   certificate. This can be used to express paths to various trust
   anchors. Clients can fetch these alternates and use their own
   heuristics to decide which is optimal.

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

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

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

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

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

   The ACME client MAY request other formats by including an Accept
   header in its request. For example, the client could use the media
   type "application/pkix-cert" [RFC2585] to request the end-entity
   certificate in DER format. Server support for alternate formats is
   OPTIONAL. For formats that can only express a single certificate,

the server SHOULD provide one or more "Link: rel="up"" headers
pointing to an issuer or issuers so that ACME clients can build a
certificate chain as defined in TLS.

7.5. Identifier Authorization

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

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

2. That the client controls the identifier in question.

This process may be repeated to associate multiple identifiers 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 order by setting
the "scope" field of the authorization to the URL for that order.

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

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

```
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="index"

{
  "status": "pending",
  "expires": "2018-03-03T14:09:00Z",

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

  "challenges": [
    {
      "type": "http-01",
      "url": "https://example.com/authz/1234/0",
      "token": "DGyRejmCefe7v4NfDGDKfA"
    },
    {
      "type": "tls-sni-02",
      "url": "https://example.com/authz/1234/1",
      "token": "DGyRejmCefe7v4NfDGDKfA"
    },
    {
      "type": "dns-01",
      "url": "https://example.com/authz/1234/2",
      "token": "DGyRejmCefe7v4NfDGDKfA"
    }
  ]
}
```

7.5.1. 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.

   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 URL (not authorization
   URL) once it is ready for the server to attempt validation.

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/1234/0 HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "Q_s3MWoqT05TrdkM2MTDcw",
    "url": "https://example.com/acme/authz/1234/0"
  }),
  "payload": base64url({
    "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 any reason or does not
provide the server with appropriate information to validate the
challenge, then the server MUST return an HTTP error. On receiving
such an error, the client SHOULD undo any actions that have been
taken to fulfill the challenge, e.g., removing files that have been
provisioned to a web server.

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

Usually, the validation process will take some time, so the client
will need to poll the authorization resource to see when it is
finalized. For challenges where the client can tell when the server
has validated the challenge (e.g., by seeing an HTTP or DNS request

from the server), the client SHOULD NOT begin polling until it has
seen the validation request from the server.

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

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

HTTP/1.1 200 OK

{
  "status": "valid",
  "expires": "2018-09-09T14:09:00Z",

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

  "challenges": [
    {
      "type": "http-01"
      "url": "https://example.com/authz/1234/0",
      "status": "valid",
      "validated": "2014-12-01T12:05:00Z",
      "token": "IlirfxKKXAsHtmzK29Pj8A",
      "keyAuthorization": "IlirfxKKXA...vb29HhjjLPSggwiE"
    }
  ]
}
```

7.5.2. Deactivating an Authorization

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

```
POST /acme/authz/1234 HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "xWCM9lGbIyCgue8di6ueWQ",
    "url": "https://example.com/acme/authz/1234"
  }),
  "payload": base64url({
    "status": "deactivated"
  }),
  "signature": "srX9Ji7Le9bjszhu...WTFdtujObzMtZcx4"
}
```

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

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

7.6. Certificate Revocation

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

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

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

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

{
  "protected": base64url({
    "alg": "ES256",
    "jwk": /* account key */,
    "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 requests 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 MUST consider at least the following
accounts authorized for a given certificate:

o the account that issued the certificate.

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

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

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

```
HTTP/1.1 200 OK
Replay-Nonce: IXVHDyxIRGcTE0VSblhPzw
Content-Length: 0

--- or ---

HTTP/1.1 403 Forbidden
Replay-Nonce: IXVHDyxIRGcTE0VSblhPzw
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"
}
```

8. Identifier Validation Challenges

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

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

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

o Control the identifier in question

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

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

This section describes an initial set of challenge types. 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.

url (required, string): The URL 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".

errors (optional, array of object): Errors that occurred while the
    server was validating the challenge, if any, structured as problem
    documents [RFC7807]. The server MUST NOT modify the array except
    by appending entries onto the end. The server can limit the size
    of this object by limiting the number of times it will retry a
    challenge.

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

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

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

8.1. Key Authorizations

   Several of the challenges in this document make 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 [FIPS180-4]. As noted in JWA
   [RFC7518] any prepended zero octets in the JWK object MUST be
   stripped before doing the computation.

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

8.2. Retrying Challenges

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

   Clients SHOULD NOT respond to challenges until they believe that the
   server's queries will succeed. If a server's initial validation
   query fails, the server SHOULD retry the query after some time.
   While the server is still trying, the status of the challenge remains
   "pending"; it is only marked "invalid" once the server has given up.

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

   Clients can explicitly request a retry by re-sending their response
   to a challenge in a new POST request (with a new nonce, etc.). This
   allows clients to request a retry when the state has changed (e.g.,
   after firewall rules have been updated). Servers SHOULD retry a

request immediately on receiving such a POST request. In order to
avoid denial-of-service attacks via client-initiated retries, servers
SHOULD rate-limit such requests.

8.3. HTTP Challenge

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

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

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

token (required, string): A random value that uniquely identifies
   the challenge. This value MUST have at least 128 bits of entropy.
   It MUST NOT contain any characters outside the base64url alphabet,
   including padding characters ("=").

```
GET /acme/authz/1234/0 HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
{
  "type": "http-01",
  "url": "https://example.com/acme/authz/0",
  "status": "pending",
  "token": "LoqXcYV8q5ONbJQxbmR7SCTNo3tiAXDfowyjxAjEuX0"
}
```

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.

```
GET /.well-known/acme-challenge/LoqXcYV8q5ONbJQxbmR7SCTNo3tiAXDfowyjxAjEuX0
Host: example.org

HTTP/1.1 200 OK
LoqXcYV8q5ONbJQxbmR7SCTNo3tiAXDfowyjxAjEuX0.9jg46WB3rR_AHD-
EBXdN7cBkH1WOu0tA3M9fm21mqTI
```

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.

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

```
POST /acme/authz/1234/0
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "JHb54aT_KTXBWQOzGYkt9A",
    "url": "https://example.com/acme/authz/1234/0"
  }),
  "payload": base64url({
    "keyAuthorization": "evaGxfADs...62jcerQ"
  }),
  "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
}
```

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. Construct a URL by populating the URL template [RFC6570]
    "http://{domain}/.well-known/acme-challenge/{token}", where: *
    the domain field is set to the domain name being verified; and *
    the token field is set to the token in the challenge.

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

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

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

5. Verify that key authorization provided by the HTTP server matches
    the key authorization provided by the client in its response to
    the challenge.

The server SHOULD follow redirects when dereferencing the URL.

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.

8.4. TLS with Server Name Indication (TLS SNI) Challenge

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 the DNS A and AAAA resource
records for 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 TLS
server and verifying a particular 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.
   It MUST NOT contain any characters outside the base64url alphabet,
   including padding characters ("=").

```
GET /acme/authz/1234/1 HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
{
  "type": "tls-sni-02",
  "url": "https://example.com/acme/authz/1234/1",
  "status": "pending",
  "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 [RFC5280].

SAN A MUST be constructed as follows: compute the SHA-256 digest
[FIPS180-4] of the 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 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.

```
POST /acme/authz/1234/1
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "JHb54aT_KTXBWQOzGYkt9A",
    "url": "https://example.com/acme/authz/1234/1"
  }),
  "payload": base64url({
    "keyAuthorization": "evaGxfADs...62jcerQ"
  }),
  "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
}
```

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,
       presenting SAN A in the SNI field. This connection MUST be sent
       to TCP port 443 on the TLS server. 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.

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

8.5. DNS Challenge

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

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

   token (required, string): A random value that uniquely identifies
      the challenge. This value MUST have at least 128 bits of entropy.
      It MUST NOT contain any characters outside the base64url alphabet,
      including padding characters ("=").

   GET /acme/authz/1234/2 HTTP/1.1
   Host: example.com

   HTTP/1.1 200 OK
   {
     "type": "dns-01",
     "url": "https://example.com/acme/authz/1234/2",
     "status": "pending",
     "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 [FIPS180-4] 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.org", then the client would provision the following DNS
record:

_acme-challenge.example.org. 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.

```
POST /acme/authz/1234/2
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "JHb54aT_KTXBWQOzGYkt9A",
    "url": "https://example.com/acme/authz/1234/2"
  }),
  "payload": base64url({
    "keyAuthorization": "evaGxfADs...62jcerQ"
  }),
  "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
}
```

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 [FIPS180-4] of the key authorization

2. Query for TXT records for the validation domain name

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

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

8.6. Out-of-Band Challenge

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"

href (required, string): The URL to be visited. The scheme of this
   URL MUST be "http" or "https". Note that this field is distinct
   from the "url" field of the challenge, which identifies the
   challenge itself.

```
GET /acme/authz/1234/3 HTTP/1.1
Host: example.com

HTTP/1.1 200 OK
{
  "type": "oob-01",
  "url": "https://example.com/acme/authz/1234/3",
  "status": "pending",
  "href": "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 chooses to complete
this challenge (by visiting 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"

```
POST /acme/authz/1234/3
Host: example.com
Content-Type: application/jose+json

{
  "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/1",
    "nonce": "JHb54aT_KTXBWQOzGYkt9A",
    "url": "https://example.com/acme/authz/1234/3"
  }),
  "payload": base64url({
    "type": "oob-01"
  }),
  "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
}
```

On receiving a response, the server MUST verify that the value of the
"type" field is "oob-01". Otherwise, the steps the server takes to
validate identifier possession are determined by the server's local
policy.

9. IANA Considerations

9.1. MIME Type: application/pem-certificate-chain

The "Media Types" registry should be updated with the following
additional value:

MIME media type name: application

MIME subtype name: pem-certificate-chain

Required parameters: None

Optional parameters: None

Encoding considerations: None

Security considerations: Carries a cryptographic certificate and its
associated certificate chain

Interoperability considerations: None

Published specification: draft-ietf-acme-acme [[RFC EDITOR: Please
replace draft-ietf-acme-acme above with the RFC number assigned to
this]]

Applications which use this media type: Any MIME-compliant transport

Additional information:

File should contain one or more certificates encoded as PEM according
to RFC 7468 [RFC7468]. In order to provide easy interoperation with
TLS, the first certificate MUST be an end-entity certificate. Each
following certificate SHOULD directly certify one preceding it.
Because certificate validation requires that trust anchors be
distributed independently, a certificate that specifies a trust
anchor MAY be omitted from the chain, provided that supported peers
are known to possess any omitted certificates.

9.2. 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 8.3

Related information: N/A

9.3. 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 6.4.1 |
    +-------------------+----------+----------+---------------+
```

9.4. "url" JWS Header Parameter

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

o Header Parameter Name: "url"

o Header Parameter Description: URL

o Header Parameter Usage Location(s): JWE, JWS

o Change Controller: IESG

o Specification Document(s): Section 6.3.1 of RFC XXXX

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

9.5. "nonce" JWS Header Parameter

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

o Header Parameter Name: "nonce"

o Header Parameter Description: Nonce

o Header Parameter Usage Location(s): JWE, JWS

o Change Controller: IESG

o Specification Document(s): Section 6.4.2 of RFC XXXX

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

9.6. 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.

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

9.7. New Registries

This document requests that IANA create the following new registries:

1. ACME Account Object Fields (Section 9.7.1)

2. ACME Order Object Fields ([Section 9.7.2](#))

3. ACME Error Types ([Section 9.7.3](#))

4. ACME Resource Types ([Section 9.7.4](#))

5. ACME Identifier Types ([Section 9.7.5](#))

6. ACME Validation Methods ([Section 9.7.6](#))

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

9.7.1. Fields in Account Objects

This registry lists field names that are defined for use in ACME account objects. Fields marked as "configurable" may be included in a new-account request.

Template:

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

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

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

o Reference: Where this field is defined

Initial contents: The fields and descriptions defined in [Section 7.1.2](#).

Field Name	Field Type	Configurable	Reference
status	string	false	RFC XXXX
contact	array of string	true	RFC XXXX
external-account-binding	object	true	RFC XXXX
terms-of-service-agreed	boolean	true	RFC XXXX
orders	array of string	false	RFC XXXX

9.7.2. Fields in Order Objects

This registry lists field names that are defined for use in ACME
order objects. Fields marked as "configurable" may be included in a
new-order request.

Template:

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

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

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

o Reference: Where this field is defined

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

```
+----------------+----------------+-------------+-----------+
| Field Name     | Field Type     | Configurable | Reference |
+----------------+----------------+-------------+-----------+
| status         | string         | false       | RFC XXXX  |
|                |                |             |           |
| expires        | string         | false       | RFC XXXX  |
|                |                |             |           |
| csr            | string         | true        | RFC XXXX  |
|                |                |             |           |
| notBefore      | string         | true        | RFC XXXX  |
|                |                |             |           |
| notAfter       | string         | true        | RFC XXXX  |
|                |                |             |           |
| authorizations | array of string | false      | RFC XXXX  |
|                |                |             |           |
| certificate    | string         | false       | RFC XXXX  |
+----------------+----------------+-------------+-----------+
```

9.7.3. Error Types

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

Template:

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

o Description: A human-readable description of the error

o Reference: Where the error is defined

Initial contents: The types and descriptions in the table in
Section 6.6 above, with the Reference field set to point to this
specification.

9.7.4. Resource Types

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

Template:

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

o Resource type: The type of resource labeled by the field

   o Reference: Where the resource type is defined

   Initial contents:

```
          +-------------+--------------------+-----------+
          | Field Name  | Resource Type      | Reference |
          +-------------+--------------------+-----------+
          | new-account | New account        | RFC XXXX  |
          |             |                    |           |
          | new-order   | New order          | 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]]

9.7.5. Identifier Types

   This registry lists the types of identifiers that can be present in
   ACME authorization objects.

   Template:

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

   o Reference: Where the identifier type is defined

   Initial contents:

```
                 +-------+-----------+
                 | Label | Reference |
                 +-------+-----------+
                 | dns   | RFC XXXX  |
                 +-------+-----------+
```

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

9.7.6. Validation Methods

   This registry lists identifiers for the ways that CAs can validate
   control of identifiers. Each method's entry must specify whether it
   corresponds to an ACME challenge type. The "Identifier Type" field

   must be contained in the Label column of the ACME Identifier Types
   registry.

   Template:

   o Label: The identifier for this validation method

   o Identifier Type: The type of identifier that this method applies
      to

   o ACME: "Y" if the validation method corresponds to an ACME
      challenge type; "N" otherwise.

   o Reference: Where the validation method is defined

   Initial Contents

```
        +------------+-----------------+------+-----------+
        | Label      | Identifier Type | ACME | Reference |
        +------------+-----------------+------+-----------+
        | http-01    | dns             | Y    | RFC XXXX  |
        |            |                 |      |           |
        | tls-sni-02 | dns             | Y    | RFC XXXX  |
        |            |                 |      |           |
        | dns-01     | dns             | Y    | RFC XXXX  |
        |            |                 |      |           |
        | oob-01     | dns             | Y    | RFC XXXX  |
        +------------+-----------------+------+-----------+
```

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

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

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

10. Security Considerations

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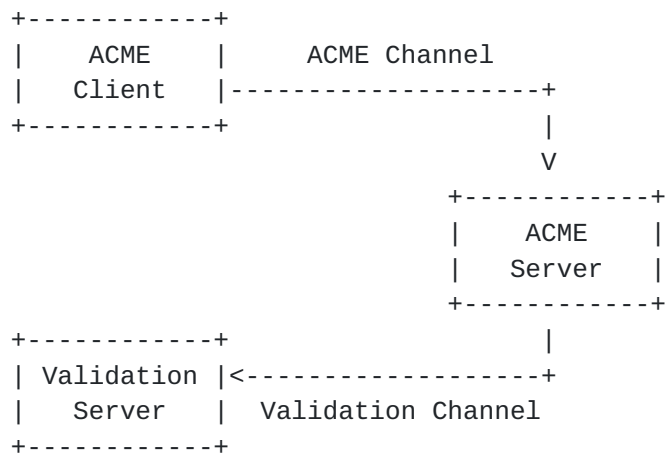   precise goals:

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

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

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

10.1. Threat model

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

   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

```
    +------------+
    |    ACME    |      ACME Channel
    |   Client   |--------------------+
    +------------+                    |
                                      V
                            +------------+
                            |    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 channel, for example, uses a different communications pattern: the ACME channel will comprise inbound HTTPS connections to the ACME server and the validation channel outbound HTTP or DNS requests.

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

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

10.2. Integrity of Authorizations

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

Validation responses need to be bound to an account key pair in order to avoid situations where an ACME MitM can switch out a legitimate domain holder's account key for one of his choosing, e.g.:

o Legitimate domain holder registers account key pair A

o MitM registers account key pair B

o Legitimate domain holder sends a new-order request signed using account key A

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

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

o 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 reflects the account public
key, is provided to the server in the validation response over the
validation channel and signed afterwards by the corresponding private
key in the challenge response over the ACME channel.

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

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 attacks. For example, on a web server
that allows non-administrative users to write to .well-known, any
user can claim to own the web server's hostname by responding to an
HTTP challenge, and likewise for TLS configuration and TLS SNI.
Similarly, if a server that can be used for ACME validation is
compromised by a malicious actor, then that malicious actor can use
that access to obtain certificates via ACME.

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

More limited forms of delegation can also lead to an unintended party
gaining the ability to successfully complete a validation
transaction. For example, suppose an ACME server follows HTTP
redirects in HTTP validation and a website operator provisions a
catch-all redirect rule that redirects requests for unknown resources

to a different domain. Then the target of the redirect could use
that to get a certificate through HTTP validation since the
validation path will not be known to the primary server.

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

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

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

o 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.

It is RECOMMENDED that the server perform DNS queries and make HTTP
and TLS connections from various network perspectives, in order to
make MitM attacks harder.

10.3. Denial-of-Service Considerations

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

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

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

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

10.4. Server-Side Request Forgery

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

Some 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 URL.

In order to further limit the SSRF risk, ACME server operators should
ensure that validation queries can only be sent to servers on the
public Internet, and not, say, web services within the server
operator's internal network. Since the attacker could make requests

to these public servers himself, he can't gain anything extra through
an SSRF attack on ACME aside from a layer of anonymization.

10.5. CA Policy Considerations

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

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?

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

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

CAs using ACME to allow clients to agree to terms of service should
keep in mind that ACME clients can automate this agreement, possibly
not involving a human user. If a CA wishes to have stronger evidence
of user consent, it may present an out-of-band requirement or
challenge to require human involvement.

11. Operational Considerations

There are certain factors that arise in operational reality that
operators of ACME-based CAs will need to keep in mind when
configuring their services. For example:

11.1. DNS security

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

An ACME-based CA will often need to make DNS queries, e.g., to validate control of DNS names. Because the security of such validations ultimately depends on the authenticity of DNS data, every possible precaution should be taken to secure DNS queries done by the CA. 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.).

11.2. 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. Typically, systems with default virtual hosts
do not allow the holder of the default virtual host to control what
certificates are presented on a request-by-request basis. Rather,
the default virtual host can configure which certificate is presented
in TLS on a fairly static basis, so that the certificate presented
should be stable over small intervals.

A CA can detect such a bounded default vhost 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"). If it receives the same certificate on two different
connections, then it is very likely that the server is in a default
virtual host configuration. Conversely, if the TLS server returns an
unrecognized_name alert, then this is an indication that the server
is not in a default virtual host configuration.

11.3. Token Entropy

The http-01, tls-sni-02 and dns-01 validation methods mandate the
usage of a random token value to uniquely identify the challenge.
The value of the token is required to contain at least 128 bits of
entropy for the following security properties. First, the ACME
client should not be able to influence the ACME server's choice of
token as this may allow an attacker to reuse a domain owner's
previous challenge responses for a new validation request. Secondly,
the entropy requirement prevents ACME clients from implementing a
"naive" validation server that automatically replies to challenges
without participating in the creation of the initial authorization
request.

11.4. Malformed Certificate Chains

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

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

12. 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.

13. References

13.1. Normative References

[FIPS180-4]
          Department of Commerce, National., "NIST FIPS 180-4,
          Secure Hash Standard", March 2012,
          <http://csrc.nist.gov/publications/fips/fips180-4/
          fips-180-4.pdf>.

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

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

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

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

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

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

[RFC3492] Costello, A., "Punycode: A Bootstring encoding of Unicode
           for Internationalized Domain Names in Applications
           (IDNA)", RFC 3492, DOI 10.17487/RFC3492, March 2003,
           <https://www.rfc-editor.org/info/rfc3492>.

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

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

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

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
           (TLS) Protocol Version 1.2", RFC 5246,
           DOI 10.17487/RFC5246, August 2008, <https://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,
           <https://www.rfc-editor.org/info/rfc5280>.

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

[RFC5988] Nottingham, M., "Web Linking", RFC 5988,
           DOI 10.17487/RFC5988, October 2010, <https://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, <https://www.rfc-
           editor.org/info/rfc6066>.

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

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

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

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

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

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

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

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

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

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

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

13.2. Informative References

   [I-D.ietf-acme-caa]
              Landau, H., "CAA Record Extensions for Account URI and
              ACME Method Binding", draft-ietf-acme-caa-03 (work in
              progress), August 2017.

   [I-D.ietf-acme-ip]
              Shoemaker, R., "ACME IP Identifier Validation Extension",
              draft-ietf-acme-ip-01 (work in progress), September 2017.

   [I-D.ietf-acme-telephone]
              Peterson, J. and R. Barnes, "ACME Identifiers and
              Challenges for Telephone Numbers", draft-ietf-acme-
              telephone-00 (work in progress), July 2017.

   [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.

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

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

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

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

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

   [W3C.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
   Cisco

   Email: rlb@ipv.sx

   Jacob Hoffman-Andrews
   EFF

   Email: jsha@eff.org

   James Kasten
   University of Michigan

   Email: jdkasten@umich.edu