Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 28, 2015

R. Barnes
E. Rescorla
Mozilla
P. Eckersley
S. Schoen
EFF
A. Halderman
J. Kasten
University of Michigan

January 28, 2015

Automatic Certificate Management Environment (ACME)
draft-barnes-acme-00

Abstract

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

Table of Contents

1. Introduction

Certificates 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.

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

- Generate a PKCS#10 [RFC2314] Certificate Signing Request (CSR).
- Cut-and-paste the CSR into a CA web page.
- Prove ownership of the domain by one of the following methods:
  - Put a CA-provided challenge at a specific place on the web server.
  - Put a CA-provided challenge at a DNS location corresponding to the target domain.
  - Receive CA challenge at a (hopefully) administrator-controlled e-mail address corresponding to the domain and then respond to it on the CA's web page.
- Download the issued certificate and install it on their Web Server.

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

This document describes an extensible framework for automating the issuance and domain validation procedure, thereby allowing servers and infrastructural software to obtain certificates without user interaction. Use of this protocol should radically simplify the deployment of HTTPS and the practicality of PKIX authentication for other TLS based protocols.

2. Deployment Model and Operator Experience

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

Different types of certificates reflect different kinds of CA verification of information about the certificate subject. "Domain Validation" (DV) certificates are by far the most common type. For DV validation, the CA merely verifies that the requester has effective control of the web server and/or DNS server for the domain, but does not explicitly attempt to verify their real-world identity. (This is as opposed to "Organization Validation" (OV) and "Extended Validation" (EV) certificates, where the process is intended to also verify the real-world identity of the requester.)

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

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

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

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

- The operator selects a CA.

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

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

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

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

3. Terminology

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

The two main roles in ACME are "client" and "server". The ACME client uses the protocol to request certificate management actions, such as issuance or revocation. An ACME client therefore typically runs on a web server, mail server, or some other server system which requires valid TLS certificates. The ACME server runs at a certificate authority, and responds to client requests, performing the requested actions if the client is authorized.

For simplicity, in the HTTPS transactions used by ACME, the ACME client is the HTTPS client and the ACME server is the HTTPS server.

In the discussion below, we will refer to three different types of keys / key pairs:

Subject Public Key: A public key to be included in a certificate.

Authorized Key Pair: A key pair for which the ACME server considers the holder of the private key authorized to manage certificates for a given identifier. The same key pair may be authorized for multiple identifiers.

Recovery Token: A secret value that can be used to demonstrate prior authorization for an identifier, in a situation where all Subject Private Keys and Authorized Keys are lost.

ACME messaging is based on HTTPS [RFC2818] and JSON [RFC7159]. Since JSON is a text-based format, binary fields are Base64-encoded. For Base64 encoding, we use the variant defined in [I-D.ietf-jose-json-web-signature]. The important features of this

encoding are (1) that it uses the URL-safe character set, and (2) that "=" padding characters are stripped.

4. Protocol Overview

ACME allows a client to request certificate management actions using a set of JSON messages carried over HTTPS. It is a prerequisite for this process that the client be configured with the HTTPS URI for the server. ACME messages MUST NOT be carried over "plain" HTTP, without HTTPS semantics.

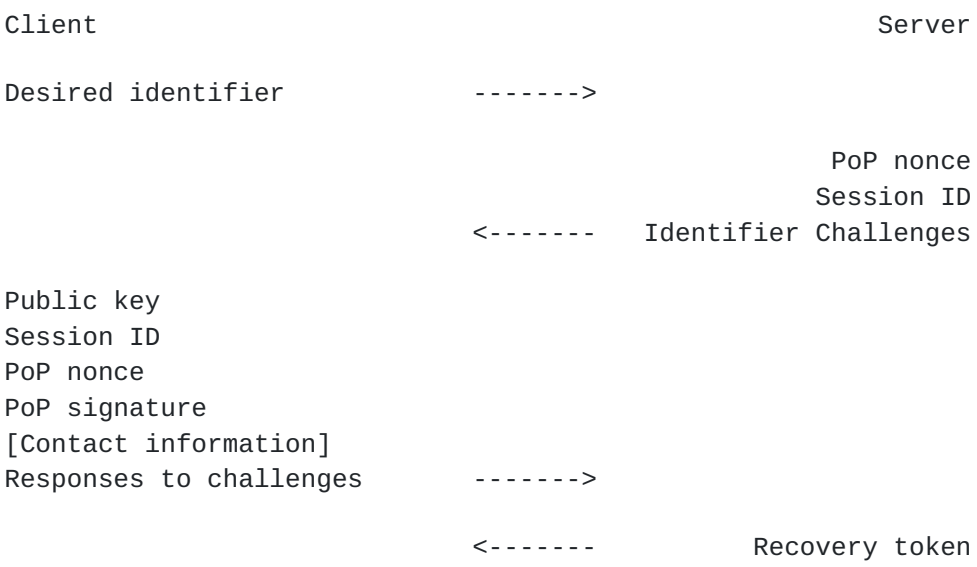
In some ways, ACME functions much like a traditional CA, in which a user creates an account, adds domains to that account (proving control of the domains), and requests certificate issuance for those domains while logged in to the account. In ACME, the account is represented by a key pair. The "add a domain" function is accomplished by authorizing the key pair for a given domain, and certificate issuance is authorized by a signature with the key pair.

The first phase of ACME is for the client to establish an authorized key pair with the server for the identifier(s) it wishes to include in the certificate. To do this, the client must demonstrate to the server both (1) that it holds the private key of the key pair being authorized, and (2) that it has authority over the identifier being claimed.

In the key authorization process, then, the server presents the client with two tests. First, a task to demonstrate that the client holds the private key of key pair being authorized, and second, a set of challenges that the client can perform to demonstrate its authority over the domain in question.

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. For example, if the client requests a domain name, the server might challenge the client to provision a record in the DNS under that name, or to provision a file on a web server reference by an A or AAAA record under that name.

After the client has prepared responses to the server's challenges, it sends a second request with its responses to these challenges. The server's response indicates whether the request for authorization has succeeded or failed. If the authorization request succeeded, the server also provides a recovery token, which the client can use in a later authorization transaction to show that it is the same as the entity that participated in this authorization.



Once the client has established an authorized key pair for an identifier, it can use the key pair to authorize the issuance of certificates for the identifier. To do this, the client sends a PKCS#10 Certificate Signing Request (CSR) to the server (indicating the identifier(s) to be included in the issued certificate), and a signature over the CSR by the private key of the authorized key pair.

If the server agrees to issue the certificate, then it creates the certificate and provides it in its response. The server may also provide a URI that can be used to renew the certificate, if it allows renewal without re-validation.

To revoke a certificate, the client simply sends a revocation request, signed with an authorized key pair, and the server indicates whether the request has succeeded.

Note that while ACME is defined with enough flexibility to handle different types of identifiers in principle, the primary use case addressed by this document is the case where domain names are used as identifiers. For example, all of the identifier validation challenges described in Section Section 6 below address validation of domain names. The use of ACME for other protocols will require further specification, in order to describe how these identifiers are encoded in the protocol, and what types of validation challenges the server might require.

5. Certificate Management

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

- Key Authorization
- Certificate Issuance
- Certificate Revocation

Each of these functions is accomplished by the client sending a sequence of HTTPS requests to the server, carrying JSON messages. Each subsection below describes the message formats used by the function, and the order in which messages are sent.

All ACME messages share some common structure. At base, each ACME message is a JSON dictionary, and MUST include a "type" field to indicate which type of message it is.

type (required, string): The type of ACME message encoded in this JSON dictionary.

All other fields in an ACME message are defined by the type, as described below. Unrecognized fields in ACME messages MUST be ignored. Creators of ACME messages MUST NOT create messages with duplicate fields. Parsers of ACME messages MAY be tolerant of duplicate fields, but the behavior of the protocol in this case is undefined.

5.1. General Request/Response Lifecycle

Client-server interactions in ACME are logically request/response transactions, corresponding directly to HTTPS requests and responses. The client sends a request message of a particular type, and the server sends response of a corresponding type.

All requests for a given ACME server are sent to the same HTTPS URI. It is assumed that clients are configured with this URI out of band. ACME requests MUST use the POST method, and since they carry JSON payloads, SHOULD set the Content-Type header field to "application/json". ACME responses MUST be carried in HTTP responses with the status code 200. ACME clients SHOULD follow HTTP redirects (301 or 302 responses), in case an ACME server is relocated.

ACME provides three general message types - "error", "defer", and "statusRequest" - to cover cases where the server is not able to return a successful result immediately. If there is a problem that prevents the request from succeeding, then the server sends an error message. The fields in an error message are as follows:

type (required, string): "error"

error (required, string): A token from the below list indicating what type of error occurred.

message (optional, string): A human-readable string describing the error.

moreInfo (optional, string): A URL of a resource containing additional human-readable documentation about the error, such as advice on how to revise the request or adjust the client configuration to allow the request to succeed, or documentation of CA issuance policies that describe why the request cannot be fulfilled.

```
{
  "type": "error",
  "error": "badCSR",
  "message": "RSA keys must be at least 2048 bits long",
  "moreInfo": "https://ca.example.com/documentation/csr-requirements"
}
```

The possible error codes are as follows:

Code	Semantic
malformed	The request message was malformed
unauthorized	The client lacks sufficient authorization
serverInternal	The server experienced an internal error
notSupported	The request type is not supported
unknown	The server does not recognize an ID/token in the request
badCSR	The CSR is unacceptable (e.g., due to a short key)

The server may also defer providing a response by sending a defer message. For example, in the key authorization process, the server may need additional time to validate the client's responses to its challenges. Or in the issuance process, there may be some delay due to batch signing. The fields in a defer message are as follows:

type (required, string): "defer"

token (required, string): An opaque value that the client uses to check on the status of the request (using a statusRequest message).

interval (optional, number): The amount of time, in seconds, that the client should wait before checking on the status of the request. (This is a recommendation only, and clients SHOULD enforce minimum and maximum deferral times.)

message (optional, string): A human-readable string describing the reason for the deferral.

For example, a deferral due to batch signing might be indicated with a message of the following form:

```
{
  "type": "defer",
  "token": "O7-s9MNq1siZHlgrMzi9_A",
  "interval": 60,
  "message": "Warming up the HSM"
}
```

When a client receives a defer message, it periodically sends a statusRequest message to the server, with the token provided in the defer message.

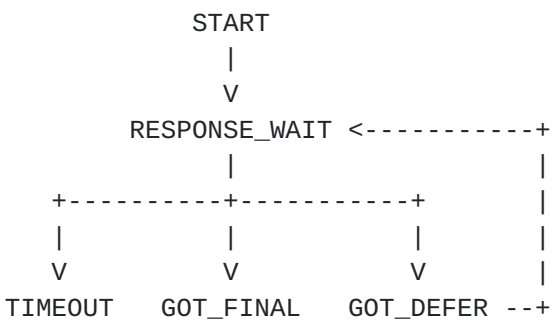
type (required, string): "statusRequest"

token (required, string): An opaque value that was provided in a defer message.

```
{
  "type": "statusRequest",
  "token": "O7-s9MNq1siZHlgrMzi9_A"
}
```

If the server responds with another defer message, then the server still does not have a final response. The client MUST ignore the "token" value in defer responses provided in responses to status requests, and continue polling with the original token. Any non-defer response (error or success) is considered final, and the client MUST cease polling.

In summary, the client goes through the following state machine to perform an ACME transaction:



The client begins by sending a request and awaiting the response. If the response contains an ACME message of any type besides "defer", then the request is completed, and if no response arrives, the request times out. If a defer request arrives, then the client waits some time and sends a polling request, whose response is handled in the same way as the original request.

The following table summarizes the request and response types defined in this document. If the server provides the client with a non-error response of a type that does not match the request message type, then the client MUST treat it as an error message with code "serverInternal".

Request	Response
challengeRequest	challenge
authorizationRequest	authorization
certificateRequest	certificate
revocationRequest	revocation
statusRequest	(any)

5.2. Signatures

ACME uses a simple JSON-based structure for encoding signatures, based on the JSON Web Signature structure. An ACME signature is a JSON object, with the following fields:

alg (required, string): A token indicating the cryptographic algorithm used to compute the signature [I-D.ietf-jose-json-web-algorithms]. (MAC algorithms such as "HS*" MUST NOT be used.)

sig (required, string): The signature, base64-encoded.

nonce (required, string): A signer-provided random nonce of at least 16 bytes, base64-encoded. (For anti-replay.)

jwk (required, object): A JSON Web Key object describing the key used to verify the signature [I-D.ietf-jose-json-web-key].

Each usage of a signature object must specify the content being signed. To avoid replay risk, the input to the signature algorithm is the concatenation of the nonce with the content to be signed.

```
signature-input = nonce || content
```

A verifier computes the same input before verifying the signature. Note that while an signature object contains all of the information required to verify the signature, the verifier must also check that the public key encoded in the JWK object is the correct key for a given context.

5.3. Key Authorization

The key authorization process establishes a key pair as an authorized key pair for a given identifier. This process must assure the server of two things: First, that the client controls the private key of the key pair, and second, that the client holds the identifier in question. This process may be repeated to associate multiple identifiers to a key pair (e.g., to request certificates with multiple identifiers), or to associate multiple key pairs with an identifier (e.g., for load balancing).

As illustrated by the figure in the overview section above, the authorization process proceeds in two transactions. The client first requests a list of challenges from the server, and then requests authorization based on its answers to those challenges.

The first request in the key authorization process is a "challengeRequest" message, specifying the identifier for which the client will be requesting authorization. The fields in a "challengeRequest" message are as follows:

type (required, string): "challengeRequest"

identifier (required, string): The identifier for which authorization is being sought. For implementations of this specification, this identifier MUST be a domain name. (If other types of identifier are supported, then an extension to this protocol will need to add a field to distinguish types of identifier.)

```
{
  "type": "challengeRequest",
  "identifier": "example.com"
}
```

On receiving a "challengeRequest" message, the server determines what sorts of challenges it will accept as proof that the client holds the identifier. (The server could also decide that a particular identifier is invalid or that the server cannot possibly issue certificates related to that identifier, in which case the server may return an error.) The set of challenges may be limited by the server's capabilities, and the server may require different challenges to be completed for different identifiers (e.g., requiring a higher standard for higher-value names). In all cases, however, the server provides a nonce as a proof-of-possession challenge for the key pair being authorized. The server returns this policy to the client in a "challenge" message:

type (required, string): "challenge"

sessionID (required, string): An opaque string that allows the server to correlate transactions related to this challenge request.

nonce (required, string): A base64-encoded octet string that the client is expected to sign with the private key of the key pair being authorized.

challenges (required, array): A list of challenges to be fulfilled by the client in order to prove possession of the identifier. The syntax for challenges is described in Section Section 6.

combinations (optional, array of arrays): A collection of sets of challenges, each of which would be sufficient to prove possession of the identifier. Clients SHOULD complete a set of challenges that that covers at least one set in this array. Challenges are represented by their associated zero-based index in the challenges array.

For example, if the server wants to have the client demonstrate both that the client controls the domain name in question, and that this client is the same client that previously requested authorization for the domain name, it might issue the following request. The client is expected to provide "simpleHttps" and "recoveryToken" responses ("[0,2]"), or else "dns" and "recoveryToken" responses ("[1,2]"), or all three.

```
{
  "type": "challenge",
  "sessionID": "aefoGaavieG9Wihuk2aufai3aeZ5EeW4",
  "nonce": "czpsrF0KMH6dgajig3TGHw",
  "challenges": [
    {
      "type": "simpleHttps",
      "token": "IlirfxKKXAsHtmzK29Pj8A"
    },
    {
      "type": "dns",
      "token": "DGyRejmCefe7v4NfDGDKfA"
    },
    {
      "type": "recoveryToken"
    }
  ],
  "combinations": [
    [0, 2], [1, 2]
  ]
}
```

In order to avoid replay attacks, the server MUST generate a fresh nonce of at least 128 bits for each authorization transaction, and MUST NOT accept more than one authorizationRequest with the same nonce.

The client SHOULD satisfy all challenges in one of the sets expressed in the "combinations" array. If a "combinations" field is not specified, the client SHOULD attempt to fulfill as many challenges as possible.

Once the client believes that it has fulfilled enough challenges, it creates an authorizationRequest object requesting authorization of a key pair for this identifier based on its responses. The authorizationRequest also contains the public key to be authorized, and the signature by the corresponding private key over the nonce in the challenge.

type (required, string): "authorizationRequest"

sessionID (required, string): The session ID provided by the server in the challenge message (to allow the server to correlate the two transactions).

nonce (required, string): The nonce provided by the server in the challenge message.

signature (required, object): A signature object reflecting a signature over the identifier being authorized and the nonce provided by the server. Thus, for this authorization:

```
signature-input = signature-nonce || identifier || server-nonce
```

responses (required, array): The client's responses to the server's challenges, in the same order as the challenges. If the client chooses not to respond to a given challenge, then the corresponding entry in the response array is set to null. Otherwise, it is set to a value defined by the challenge type.

contact (optional, array): An array of URIs that the server can use to contact the client for issues related to this authorization. For example, the server may wish to notify the client about server-initiated revocation, or check with the client on future authorizations (see the "recoveryContact" challenge type).

```
{
  "type": "authorizationRequest",
  "sessionID": "aefoGaavieG9Wihuk2aufai3aeZ5EeW4",
  "nonce": "czpsrF0KMH6dgajig3TGHw",
  "signature": {
    "nonce": "Aenb3DvfvOPImdXdnxHMlp7Jh4qsgYeTEM-dFgFOGxU",
    "alg": "ES256",
    "jwk": {
      "kty": "EC",
      "crv": "P-256",
      "x": "NJ15BoXput18sSwnXA3gJEEnqIAzxSEl9ga8wGM4mEU",
      "y": "6l_U9mals_dwt77tIxSiQ6oL_CyLVey4baa8wCn0V9k"
    },
    "sig": "lxj0Ucdo4r5s1c1cuY2R7oKqWi4QuNJzdwe5/4m9zWQ"
  },
  "responses": [
    {
      "type": "simpleHttps",
      "path": "Hf5GrX4Q7EBax9hc2jJnfw"
    },
    null,
    {
      "type": "recoveryToken",
      "token": "23029d88d9e123e"
    }
  ],
  "contact": [
    "mailto:cert-admin@example.com",
    "tel:+12025551212"
  ]
}
```

Once it has received the client's responses, the server verifies them according to procedures specific to each challenge type. Because some of these procedures take time to verify, it is likely that the server will respond to an authorizationRequest message with a defer message.

If there is a problem with the authorizationRequest (e.g., the signature object does not verify), or if the available responses are not sufficient to convince the server that the client controls the identifier, then the server responds with an error message. The server should use the "unauthorized" error code for cases where the client's responses were insufficient. If the server is satisfied that the client controls the private key and identifier in question, then it sends an authorization message indicating the success of the authorization request, and providing a recovery token that the client

can use to help recover authorization if the private key of the authorized key pair is lost.

type (required, string): "authorization"

recoveryToken (optional, string): An arbitrary server-generated string. If the server provides a recovery token, it MUST generate a unique value for every authorization transaction, and this value MUST NOT be predictable or guessable by a third party.

identifier (optional, string): The identifier for which authorization has been granted.

jwk (optional, object): A JSON Web Key object describing the authorized public key.

5.3.1. Recovery Tokens

A recovery token is a fallback authentication mechanism. In the event that a client loses all other state, including authorized key pairs and key pairs bound to certificates, the client can use the recovery token to prove that it was previously authorized for the identifier in question.

This mechanism is necessary because once an ACME server has issued an Authorization Key for a given identifier, that identifier enters a higher-security state, at least with respect the ACME server. That state exists to protect against attacks such as DNS hijacking and router compromise which tend to inherently defeat all forms of Domain Validation. So once a domain has begun using ACME, new DV-only authorization will not be performed without proof of continuity via possession of an Authorized Private Key or potentially a Subject Private Key for that domain.

This higher state of security poses some risks. From time to time, the administrators and owners of domains may lose access to keys they have previously had issued or certified, including Authorized private keys and Subject private keys. For instance, the disks on which this key material is stored may suffer failures, or passphrases for these keys may be forgotten. In some cases, the security measures that are taken to protect this sensitive data may contribute to its loss.

Recovery Tokens and Recovery Challenges exist to provide a fallback mechanism to restore the state of the domain to the server-side administrative security state it was in prior to the use of ACME, such that fresh Domain Validation is sufficient for reauthorization.

Recovery tokens are therefore only useful to an attacker who can also perform Domain Validation against a target domain, and as a result client administrators may choose to handle them with somewhat fewer security precautions than Authorized and Subject private keys, decreasing the risk of their loss.

Recovery tokens come in several types, including high-entropy passcodes (which need to be safely preserved by the client admin) and email addresses (which are inherently hard to lose, and which can be used for verification, though they may be a little less secure).

Recovery tokens are employed in response to Recovery Challenges. Such challenges will be available if the server has issued Recovery Tokens for a given domain, and the combination of a Recovery Challenge and a domain validation Challenge is a plausible alternative to other challenge sets for domains that already have extant Authorized keys.

5.4. Certificate Issuance

The holder of an authorized key pair for an identifier may use ACME to request that a certificate be issued for that identifier. The client makes this request using a "certificateRequest" message, which contains a Certificate Signing Request (CSR) [RFC2986] and a signature by the authorized key pair.

type (required, string): "certificateRequest"

csr (required, string): A CSR encoding the parameters for the certificate being requested. The CSR is sent in base64-encoded version the DER format. (Note: This field uses the same modified base64-encoding rules used elsewhere in this document, so it is different from PEM.)

signature (required, object): A signature object reflecting a signature by an authorized key pair over the CSR.

```
{
  "type": "certificateRequest",
  "csr": "5jNudRx6Ye4HzKEqT5...FS6aKdZeGsysoCo4H9P",
  "signature": {
    "alg": "RS256",
    "nonce": "h5aYpWVkq-xlJh6cpR-3cw",
    "sig": "KxITJ0rNlfDMAtfDr8eAw...fSSoehDFNZKQKzTZPtQ",
    "jwk": {
      "kty":"RSA",
      "e":"AQAB",
      "n":"KxITJ0rNlfDMAtfDr8eAw...fSSoehDFNZKQKzTZPtQ"
    }
  }
}
```

The CSR encodes the client's requests with regard to the content of the certificate to be issued. The CSR MUST contain at least one extensionRequest attribute [RFC2985] requesting a subjectAltName extension, containing the requested identifiers.

The values provided in the CSR are only a request, and are not guaranteed. The server or CA may alter any fields in the certificate before issuance. For example, the CA may remove identifiers that are not authorized for the key indicated in the "authorization" field.

If the CA decides to issue a certificate, then the server responds with a certificate message. (Of course, the server may also respond with an error message if issuance is denied, or a defer message if there may be some delay in issuance.)

type (required, string): "certificate"

certificate (required, string): The issued certificate, as a base64-encoded DER certificate.

chain (optional, array of string): A chain of CA certificates which are parents of the issued certificate. Each certificate is in base64-encoded DER form (not PEM, as for CSRs above). This array MUST be presented in the same order as would be required in a TLS handshake [RFC5246].

refresh (optional, string): An HTTP or HTTPS URI from which updated versions of this certificate can be fetched.

```
{
  "type": "certificate",
  "certificate": "Zmzdx7UKvwDJ6bk...YBX22NPGQZyYcg",
  "chain": [
    "WUn8L2vLT553pIWJ2...gJ574o2anls1k2p",
    "y3O4puZa9r5KBk1LX...Ya7jlaAZUfuYZGZ"
  ],
  "refresh": "https://example.com/refresh/Dr8eAwTVQfSS/"
}
```

The certificate message allows the server to provide the certificate itself, as well as some associated management information. The chain of CA certificates can simplify TLS server configuration, by allowing the server to suggest the certificate chain that a TLS server using the issued certificate should present.

The refresh URI allows the client to download updated versions of the issued certificate, in the sense of certificates with different validity intervals, but otherwise the same contents (in particular, the same names and public key). This can be useful in cases where a CA wishes to issue short-lived certificates, but is still willing to vouch for an identifier-key binding over a longer period of time. To download an updated certificate, the client simply sends a GET request to the refresh URI.

5.5. Certificate Revocation

To request that a certificate be revoked, the client sends a revocationRequest message that indicates the certificate to be revoked, with a signature by an authorized key:

type (required, string): "revocationRequest"

certificate (required, string): The certificate to be revoked.

signature (required, object): A signature object reflecting a signature by an authorized key pair over the certificate.

```
{
  "type": "revocationRequest",
  "certificate": "Zmzdx7UKvwDJ6bk...YBX22NPGQZyYcg",
  "signature": {
    "alg": "RS256",
    "nonce": "OQqU4VlhXhvZW9FIqNW-jg",
    "sig": "KxITJ0rNlfDMAtfDr8eAw...fSSoehDFNZKQKzTZPtQ",
    "jwk": {
      "kty":"RSA",
      "e":"AQAB",
      "n":"KxITJ0rNlfDMAtfDr8eAw...fSSoehDFNZKQKzTZPtQ"
    }
  }
}
```

Before revoking a certificate, the server MUST verify that the public key indicated in the signature object is authorized to act for all of the identifier(s) in the certificate. The server MAY also accept a signature by the private key corresponding to the public key in the certificate.

If the revocation fails, the server returns an error message, e.g., an "unauthorized" error if the signing key was not authorized to revoke this certificate. If the revocation succeeds, then the server confirms with a "revocation" message, which has no payload.

type (required, string): "revocation"

```
{
  "type": "revocation"
}
```

6. Identifier Validation Challenges

There are few types of identifier 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.

To accommodate this reality, ACME includes an extensible challenge/response framework for identifier validation. This section describes an initial set of Challenge types. Each challenge must describe:

- Content of Challenge payloads (in Challenge messages)

- Content of Response payloads (in authorizationRequest messages)

- How the server uses the Challenge and Response to verify control of an identifier

The only general requirement for Challenge and Response payloads is that they MUST be structured as a JSON object, and they MUST contain a parameter "type" that specifies the type of Challenge or Response encoded in the object.

Different challenges allow the server to obtain proof of different aspects of control over an identifier. In some challenges, like Simple HTTPS and DVSNI, the client directly proves control of an identifier. In other challenges, such as Recovery or Proof of Possession, the client proves historical control of the identifier, by reference to a prior authorization transaction or certificate.

The choice of which Challenges to offer to a client under which circumstances is a matter of server policy. A server may choose different sets of challenges depending on whether it has interacted with a domain before, and how. For example:

Domain status	Challenges typically sufficient for (re)Authorization
No known prior certificates or ACME usage	Domain Validation (DVSNI or Simple HTTPS)
Existing valid certs, first use of ACME	DV + Proof of Possession of previous CA-signed key
Ongoing ACME usage	PoP of previous Authorized key
Ongoing ACME usage, lost Authorized key	DV + (Recovery or PoP of ACME-certified Subject key)
ACME usage, all keys and recovery tokens lost	Recertification by another CA + PoP of that key

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

6.1. Simple HTTPS

With Simple HTTPS validation, the client in an ACME transaction proves its control over a domain name by proving that it can provision resources on an HTTPS server that responds for that domain name. The ACME server challenges the client to provision a file with a specific string as its contents.

type (required, string): The string "simpleHttps"

token (required, string): The value to be provisioned in the file. This value MUST have at least 128 bits of entropy, in order to prevent an attacker from guessing it. It MUST NOT contain any non-ASCII characters.

```
{
  "type": "simpleHttps",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ+PCt92wr+oA"
}
```

A client responds to this Challenge by provisioning the nonce as a resource on the HTTPS server for the domain in question. The path at which the resource is provisioned is determined by the client, but MUST begin with ".well-known/acme-challenge/". The content type of the resource MUST be "text/plain". The client returns the part of the path coming after that prefix in its Response message.

type (required, string): The string "simpleHttps"

path (required, string): The string to be appended to the standard prefix ".well-known/acme-challenge" in order to form the path at which the nonce resource is provisioned. The result of concatenating the prefix with this value MUST match the "path" production in the standard URI format [RFC3986]

```
{
  "type": "simpleHttps",
  "path": "6tbIMBC5Anhl5bOlWT5ZFA"
}
```

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

1. Form a URI by populating the URI template "https://{domain}/.well-known/acme-challenge/{path}", where the

domain field is set to the domain name being verified and the path field is the path provided in the challenge [RFC6570].

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

3. Dereference the URI using an HTTPS GET request.

4. Verify that the certificate presented by the HTTPS server is a valid self-signed certificate, and contains the domain name being validated as well as the public key of the key pair being authorized.

5. Verify that the Content-Type header of the response is either absent, or has the value "text/plain"

6. Compare the entity body of the response with the nonce. This comparison MUST be performed in terms of Unicode code points, taking into account the encodings of the stored nonce and the body of the request.

If the GET request succeeds and the entity body is equal to the nonce, then the validation is successful. If the request fails, or the body does not match the nonce, then it has failed.

6.2. Domain Validation with Server Name Indication

The Domain Validation with Server Name Indication (DVSNI) validation method aims to ensure that the ACME client has administrative access to the web server at the domain name being validated, and possession of the private key being authorized. The ACME server verifies that the operator can reconfigure the web server by having the client create a new self-signed challenge certificate and respond to TLS connections from the ACME server with it.

The challenge proceeds as follows: The ACME server sends the client a random value R and a nonce used to identify the transaction. The client responds with another random value S. The server initiates a TLS connection on port 443 to a host with the domain name being validated. In the handshake, the ACME server sets the Server Name Indication extension set to "<nonce>.acme.invalid". The TLS server (i.e., the ACME client) should respond with a valid self-signed certificate containing both the domain name being validated and the domain name "<Z>.acme.invalid", where Z = SHA-256(R || S).

The ACME server's Challenge provides its random value R, and a random nonce used to identify the transaction:

type (required, string): The string "dvsni"

r (required, string): A random 32-byte octet, base64-encoded

nonce (required, string): A random 16-byte octet string, hex-encoded (so that it can be used as a DNS label)

```
{
  "type": "dvsni",
  "r": "Tyq0La3slT7tqQ0wlOiXnCY2vyez7Zo5blgPJ1xt5xI",
  "nonce": "a82d5ff8ef740d12881f6d3c2277ab2e"
}
```

The ACME server MAY re-use nonce values, but SHOULD periodically refresh them. ACME clients MUST NOT rely on nonce values being stable over time.

The client responds to this Challenge by configuring a TLS server on port 443 of a server with the domain name being validated:

1. Decode the server's random value R

2. Generate a random 32-byte octet string S

3. Compute Z = SHA-256(R || S) (where || denotes concatenation of octet strings)

4. Generate a self-signed certificate with a subjectAltName extension containing two dNSName values:

5. The domain name being validated

6. A name formed by hex-encoding Z and appending the suffix ".acme.invalid"

7. Compute a nonce domain name by appending the suffix ".acme.invalid" to the nonce provided by the server.

8. Configure the TLS server such that when a client presents the nonce domain name in the SNI field, the server presents the generated certificate.

The client's response provides its random value S:

type (required, string): The string "dvsni"

s (required, string): A random 32-byte secret octet string, base64-encoded

```
{
  "type": "dvsni",
  "s": "9dbjsl3gTAtOnEtKFEmhS6Mj-ajNjDcOmRkp3Lfzm3c"
}
```

Given a Challenge/Response pair, the ACME server verifies the client's control of the domain by verifying that the TLS server was configured as expected:

1. Compute the value Z = SHA-256(R || S)

2. Open a TLS connection to the domain name being validated on port 443, presenting the value "<nonce>.acme.invalid" in the SNI field.

3. Verify the following properties of the certificate provided by the TLS server:

   * It is a valid self-signed certificate

   * The public key is the public key for the key pair being authorized

   * It contains the domain name being validated as a subjectAltName

   * It contains a subjectAltName matching the hex-encoding of Z, with the suffix ".acme.invalid"

It is RECOMMENDED that the ACME server verify the challenge certificate using multi-path probing techniques to reduce the risk of DNS hijacking attacks.

If the server presents a certificate matching all of the above criteria, then the validation is successful. Otherwise, the validation fails.

6.3. Recovery Contact

A server may issue a recovery contact challenge to verify that the client is the same as the entity that previously requested authorization, using contact information provided by the client in a prior authorizationRequest message.

The server's message to the client may request action in-band or out-of-band to ACME. The server can provide a token in the message that the client provides in its response. Or the server could provide

some out-of-band response channel in its message, such as a URL to click in an email.

type (required, string): The string "recoveryContact"

activationURL (optional, string): A URL the client can visit to cause a recovery message to be sent to client's contact address.

successURL (optional, string): A URL the client may poll to determine if the user has successfully clicked a link or completed other tasks specified by the recovery message. This URL will return a 200 success code if the required tasks have been completed. The client SHOULD NOT poll the URL more than once every three seconds.

contact (optional, string) A full or partly obfuscated version of the contact URI that the server will use to contact the client. Client software may present this to a user in order to suggest what contact point the user should check (e.g., an email address).

```
{
  "type": "recoveryContact",
  "activationURL" : "https://example.ca/sendrecovery/a5bd99383fb0",
  "successURL" : "https://example.ca/confirmrecovery/bb1b9928932",
  "contact" : "c********n@example.com"
}
```

type (required, string): The string "recoveryContact"

token (optional, string): If the user transferred a token from a contact email or call into the client software, the client sends it here. If it the client has received a 200 success response while polling the RecoveryContact Challenge's successURL, this field SHOULD be omitted.

```
{
  "type": "recoveryContact",
  "token": "23029d88d9e123e"
}
```

If the value of the "token" field matches the value provided in the out-of-band message to the client, or if the client has completed the required out-of-band action, then the validation succeeds. Otherwise, the validation fails.

6.4. Recovery Token

A recovery token is a simple way for the server to verify that the client was previously authorized for a domain. The client simply provides the recovery token that was provided in the authorize message.

type (required, string): The string "recoveryToken"

```
{
  "type": "recoveryToken"
}
```

The response to a recovery token challenge is simple; the client sends the requested token that it was provided by the server earlier.

type (required, string): The string "recoveryToken"

token (optional, string): The recovery token provided by the server.

```
{
  "type": "recoveryToken",
  "token": "23029d88d9e123e"
}
```

If the value of the "token" field matches a recovery token that the server previously provided for this domain, then the validation succeeds. Otherwise, the validation fails.

6.5. Proof of Possession of a Prior Key

The Proof of Possession challenge verifies that a client possesses a private key corresponding to a server-specified public key, as demonstrated by its ability to correctly sign server-provided data with that key.

This method is useful if a server policy calls for issuing a certificate only to an entity that already possesses the subject private key of a particular prior related certificate (perhaps issued by a different CA). It may also help enable other kinds of server policy that are related to authenticating a client's identity using digital signatures.

This challenge proceeds in much the same way as the proof of possession of the authorized key pair in the main ACME flow (challenge + authorizationRequest). The server provides a nonce and

the client signs over the nonce. The main difference is that rather than signing with the private key of the key pair being authorized, the client signs with a private key specified by the server. The server can specify which key pair(s) are acceptable directly (by indicating a public key), or by asking for the key corresponding to a certificate.

The server provides the following fields as part of the challenge:

type (required, string): The string "proofOfPossession"

alg (required, string): A token indicating the cryptographic algorithm that should be used by the client to compute the signature [I-D.ietf-jose-json-web-algorithms]. (MAC algorithms such as "HS*" MUST NOT be used.) The client MUST verify that this algorithm is supported for the indicated key before responding to this challenge.

nonce (required, string): A random 16-byte octet string, base64-encoded

hints (required, object): A JSON object that contains various clues for the client about what the requested key is, such that the client can find it. Entries in the hints object may include:

jwk (required, object): A JSON Web Key object describing the public key whose corresponding private key should be used to generate the signature [I-D.ietf-jose-json-web-key]

certFingerprints (optional, array): An array of certificate fingerprints, hex-encoded SHA1 hashes of DER-encoded certificates that are known to contain this key

certs (optional, array): An array of certificates, in PEM encoding, that contain acceptable public keys.

subjectKeyIdentifiers (optional, array): An array of hex-encoded Subject Key Identifiers (SKIDs) from certificate(s) that contain the key. Because of divergences in the way that SKIDs are calculated [RFC5280], there may conceivably be more than one of these.

serialNumbers (optional, array of numbers): An array of serial numbers of certificates that are known to contain the requested key

issuers (optional, array): An array of X.509 Distinguished Names [RFC5280] of CAs that have been observed to issue certificates for this key, in text form [RFC4514]

authorizedFor (optional, array): An array of domain names, if any, for which this server regards the key as an ACME Authorized key.

```
{
  "type": "proofOfPossession",
  "alg": "RS256",
  "nonce": "eET5udtV7aoX8Xl8gYiZIA",
  "hints" : {
    "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "n": "KxITJ0rNlfDMAtfDr8eAw...fSSoehDFNZKQKzTZPtQ"
    },
    "certFingerprints": [
      "93416768eb85e33adc4277f4c9acd63e7418fcfe",
      "16d95b7b63f1972b980b14c20291f3c0d1855d95",
      "48b46570d9fc6358108af43ad1649484def0debf"
    ],
    "subjectKeyIdentifiers": ["d0083162dcc4c8a23ecb8aecbd86120e56fd24e5"],
    "serialNumbers": [34234239832, 23993939911, 17],
    "issuers": [
      "C=US, O=SuperT LLC, CN=SuperTrustworthy Public CA",
      "O=LessTrustworthy CA Inc, CN=LessTrustworthy But StillSecure"
    ],
    "authorizedFor": ["www.example.com", "example.net"]
  }
}
```

In this case the server is challenging the client to prove its control over the private key that corresponds to the public key specified in the jwk object. The signing algorithm is specified by the alg field. The nonce value is used by the server to identify this challenge and is also used, also with a client-provided signature nonce, as part of the signature input.

```
signature-input = signature-nonce || server-nonce
```

The client's response includes the server-provided nonce, together with a signature over that nonce by one of the private keys requested by the server.

type (required, string): The string "proofOfPossession"

nonce (required, string): The server nonce that the server previously associated with this challenge

signature (required, object): The ACME signature computed over the signature-input using the server-specified algorithm

```
{
  "type": "proofOfPossession",
  "nonce": "eET5udtV7aoX8Xl8gYiZIA",
  "signature": {
    "alg": "RS256",
    "nonce": "eET5udtV7aoX8Xl8gYiZIA",
    "sig": "KxITJ0rNlfDMAtfDr8eAw...fSSoehDFNZKQKzTZPtQ",
    "jwk": {
      "kty": "RSA",
      "e": "AQAB",
      "n": "KxITJ0rNlfDMAtfDr8eAw...fSSoehDFNZKQKzTZPtQ"
    }
  }
}
```

Note that just as in the authorizationRequest message, there are two nonces here, once provided by the client (inside the signature object) and one provided by the server in its challenge (outside the signature object). The signature covers the concatenation of these two nonces (as specified in the signature-input above).

If the server is able to validate the signature and confirm that the jwk and alg objects are unchanged from the original challenge, the server can conclude that the client is in control of the private key that corresponds to the specified public key. The server can use this evidence in support of its authorization and certificate issuance policies.

6.6. DNS

When the identifier being validated is a domain name, the client can prove control of that domain by provisioning records under it. The DNS challenge requires the client to provision a TXT record containing a validation token under a specific validation domain name.

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

token (required, string): An ASCII string that is to be provisioned in the TXT record. This string SHOULD be randomly generated, with

at least 128 bits of entropy (e.g., a hex-encoded random octet string).

```
{
  "type": "dns",
  "token": "17817c66b60ce2e4012dfad92657527a"
}
```

In response to this challenge, the client first MUST verify that the token contains only ASCII characters. If so, the client constructs the validation domain name by appending the label "_acme-challenge" to the domain name being validated. For example, if the domain name being validated is "example.com", then the client would provision the following DNS record:

```
_acme-challenge.example.com. IN TXT "17817c66b60ce2e4012dfad92657527a"
```

The response to a DNS challenge is simply an acknowledgement that the relevant record has been provisioned.

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

```
{
  "type": "dns"
}
```

To validate a DNS challenge, the server queries for TXT records under the validation domain name. If it receives a record whose contents match the token in the challenge, then the validation succeeds. Otherwise, the validation fails.

6.7. Other possibilities

For future work:

- Email
- DNSSEC
- WHOIS

7. IANA Considerations

TODO

- Register .well-known path
- Create identifier validation method registry
- Registries of syntax tokens, e.g., message types / error types?

8. Security Considerations

TODO

- General authorization story
- PoP nonce entropy
- ToC/ToU; duration of key authorization
- Clients need to protect recovery key
- CA needs to perform a very wide range of issuance policy enforcement and sanity-check steps
- Parser safety (for JSON, JWK, ASN.1, and any other formats that can be parsed by the ACME server)

9. References

9.1. Normative References

[I-D.ietf-jose-json-web-algorithms]
Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms-40 (work in progress), January 2015.

[I-D.ietf-jose-json-web-key]
Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key-41 (work in progress), January 2015.

[I-D.ietf-jose-json-web-signature]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature-41 (work in progress), January 2015.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2314] Kaliski, B., "PKCS #10: Certification Request Syntax Version 1.5", RFC 2314, March 1998.

[RFC2985] Nystrom, M. and B. Kaliski, "PKCS #9: Selected Object Classes and Attribute Types Version 2.0", RFC 2985, November 2000.

[RFC2986] Nystrom, M. and B. Kaliski, "PKCS #10: Certification Request Syntax Specification Version 1.7", RFC 2986, November 2000.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4514] Zeilenga, K., "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names", RFC 4514, June 2006.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[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, May 2008.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, March 2012.

[RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

9.2. Informative References

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

Authors' Addresses

Richard Barnes
Mozilla

Email: rlb@ipv.sx

Eric Rescorla
Mozilla

Email: ekr@rtfm.com

Peter Eckersley
EFF

Email: pde@eff.org

Seth Schoen
EFF

Email: schoen@eff.org

Alex Halderman
University of Michigan

Email: jhalderm@eecs.umich.edu

James Kasten
University of Michigan

Email: jdkasten@umich.edu