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

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

July 21, 2015

Automatic Certificate Management Environment (ACME)
draft-barnes-acme-04

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.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

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

Copyright Notice

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 protocols based on TLS [RFC5246].

2. Deployment Model and Operator Experience

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

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 all 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.

Account 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 Key: A MAC key that a client can use to demonstrate that it participated in a prior registration transaction.

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 [RFC7515]. The important features of this encoding are (1) that it uses the URL-safe character set, and (2) that "=" padding characters are stripped.

Some HTTPS bodies in ACME are authenticated and integrity-protected by being encapsulated in a JSON Web Signature (JWS) object [RFC7515]. ACME uses a profile of JWS, with the following restrictions:

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

Additionally, JWS objects used in ACME MUST include the "nonce" header parameter, defined below.

4. Protocol Overview

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

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

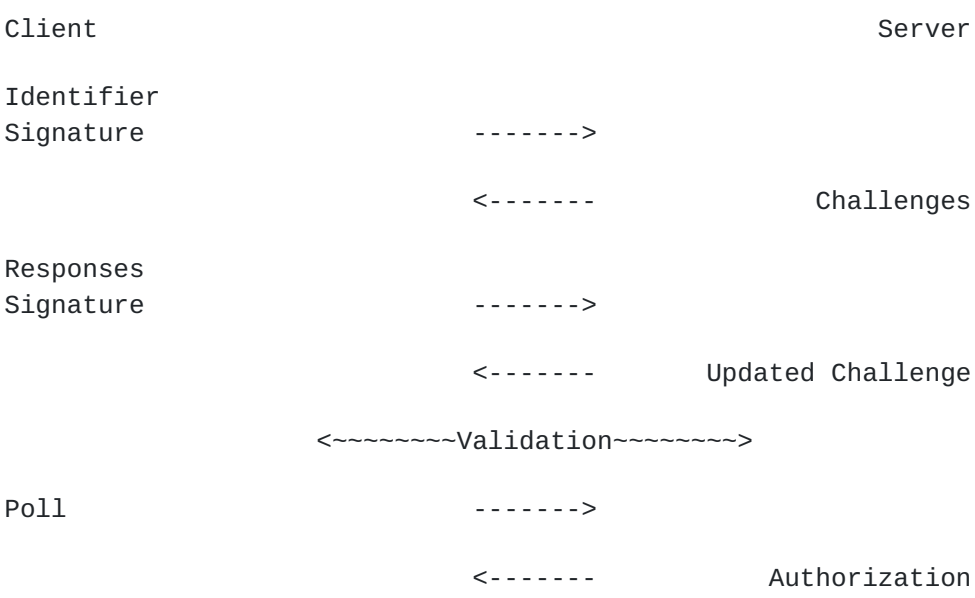
The first phase of ACME is for the client to register with the ACME server. The client generates an asymmetric key pair and associates this key pair with a set of contact information by signing the contact information. The server acknowledges the registration by replying with a registration object echoing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

Before a client can issue certificates, it must establish an authorization with the server for an account key pair to act for the identifier(s) that it wishes to include in the certificate. To do this, the client must demonstrate to the server both (1) that it holds the private key of the account key pair, and (2) that it has authority over the identifier being claimed.

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

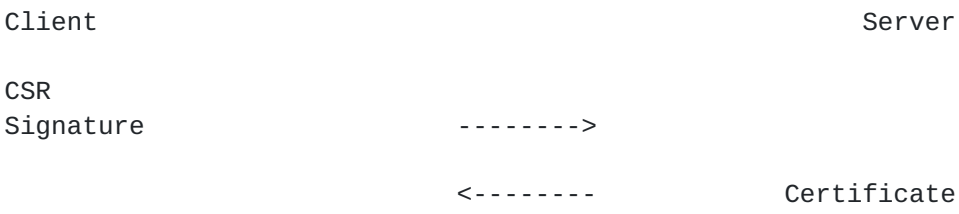
To verify that the client controls the identifier being claimed, the server issues the client a set of challenges. Because there are many different ways to validate possession of different types of identifiers, the server will choose from an extensible set of challenges that are appropriate for the identifier being claimed. The client responds with a set of responses that tell the server which challenges the client has completed. The server then validates the challenges to check that the client has accomplished the challenge.

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



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

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



To revoke a certificate, the client simply sends a revocation request, 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 7 below address validation of domain names. The use of ACME for other protocols will require further specification, in order to describe how these identifiers are encoded in the protocol, and what types of validation challenges the server might require.

5. Protocol Elements

This section describes several components that are used by ACME, and general rules that apply to ACME transactions.

5.1. HTTPS Requests

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

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

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

An ACME request carries a JSON dictionary that provides the details of the client's request to the server. In order to avoid attacks that might arise from sending a request object to a resource of the wrong type, each request object MUST have a "resource" field that indicates what type of resource the request is addressed to, as defined in the below table:

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

Other fields in ACME request bodies are described below.

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 "*".

5.2. Registration Objects

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

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

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

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

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

certificates (optional, string): A URI from which a list of certificates issued for this account can be fetched via a GET request. The result of the GET request MUST be a JSON object whose "certificates" field is an array of strings, where each string is the URI of a certificate. The server SHOULD NOT include expired certificates.

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

5.3. Authorization Objects

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

The structure of an ACME authorization resource is as follows:

identifier (required, dictionary of string): The identifier that the account is authorized to represent

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

value (required, string): The identifier itself.

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

expires (optional, string): The date after which the server will consider this authorization invalid, encoded in the format specified in RFC 3339 [RFC3339].

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

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

The only type of identifier defined by this specification is a fully-qualified domain name (type: "dns"). The value of the identifier MUST be the ASCII representation of the domain name. Wildcard domain names (with "*" as the first label) MUST NOT be included in authorization requests. See Section 6.6 below for more information about wildcard domains.

```
{
  "status": "valid",
  "expires": "2015-03-01",

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

  "challenges": [
    {
      "type": "simpleHttp",
      "status": "valid",
      "validated": "2014-12-01T12:05Z",
      "token": "IlirfxKKXAsHtmzK29Pj8A"
      "path": "Hf5GrX4Q7EBax9hc2jJnfw"
    }
  ],
}
```

5.4. Errors

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

When the server responds with an error status, it SHOULD provide additional information using problem document [I-D.ietf-appsawg-http-problem]. The "type" and "detail" fields MUST be populated. To facilitate automatic response to errors, this document defines the following standard tokens for use in the "type" field (within the "urn:acme:" namespace):

Code	Semantic
badCSR	The CSR is unacceptable (e.g., due to a short key)
badNonce	The client sent an unacceptable anti-replay nonce
connection	The server could not connect to the client for DV
dnssec	The server could not validate a DNSSEC signed domain
malformed	The request message was malformed
serverInternal	The server experienced an internal error
tls	The server experienced a TLS error during DV
unauthorized	The client lacks sufficient authorization
unknownHost	The server could not resolve a domain name

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

TODO: Flesh out errors and syntax for them

5.5. Replay protection

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

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

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

When a server rejects a request because its nonce value was unacceptable (or not present), it SHOULD provide HTTP status code 400 (Bad Request), and indicate the ACME error code "urn:acme:badNonce".

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

5.5.1. Replay-Nonce

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

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

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

Replay-Nonce = *base64url
```

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

5.5.2. "nonce" (Nonce) JWS header parameter

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

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

5.6. Key Agreement

Certain elements of the protocol will require the establishment of a shared secret between the client and the server, in such a way that an entity observing the ACME protocol cannot derive the secret. In these cases, we use a simple ECDH key exchange, based on the system used by CMS [RFC5753]:

- Inputs:
  - Client-generated key pair
  - Server-generated key pair
  - Length of the shared secret to be derived
  - Label
- Perform the ECDH primitive operation to obtain Z (Section 3.3.1 of [SEC1])
- Select a hash algorithm according to the curve being used:
  - For "P-256", use SHA-256
  - For "P-384", use SHA-384
  - For "P-521", use SHA-512
- Derive the shared secret value using the KDF in Section 3.6.1 of [SEC1] using Z and the selected hash algorithm, and with the UTF-8 encoding of the label as the SharedInfo value

In cases where the length of the derived secret is shorter than the output length of the chosen hash algorithm, the KDF referenced above reduces to a single hash invocation. The shared secret is equal to the leftmost octets of the following:

H(Z || 00000001 || label)

6. Certificate Management

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

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

6.1. Resources

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

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

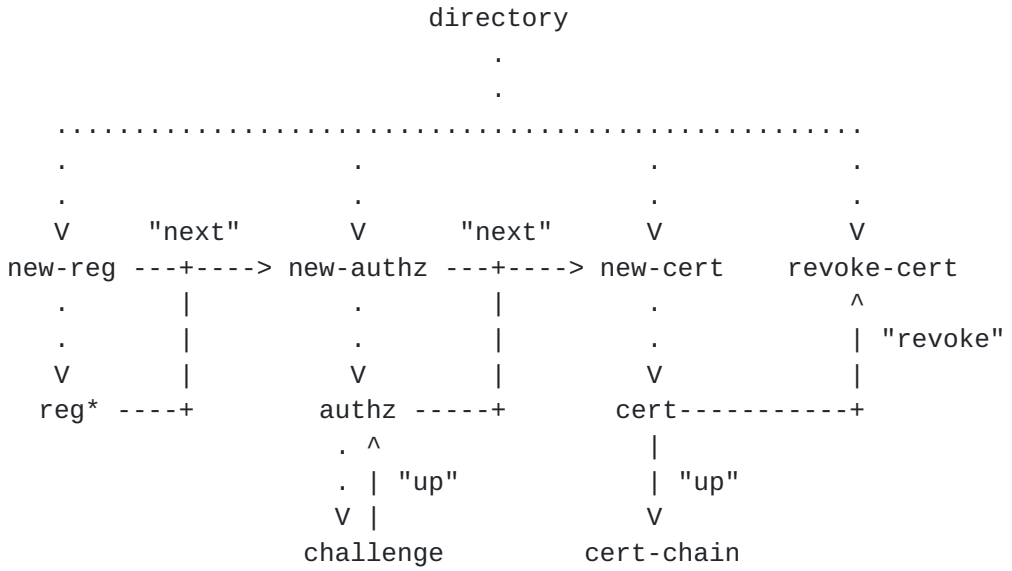
For the "new-X" resources above, the server MUST have exactly one resource for each function. This resource may be addressed by multiple URIs, but all must provide equivalent functionality.

In general, the intent is for authorization and certificate resources to contain only public information, so that CAs may publish these resources to document what certificates have been issued and how they were authorized. Non-public information, such as contact information, is stored in registration resources.

ACME uses different URIs for different management functions. Each function is listed in a directory along with its corresponding URI, so clients only need to be configured with the directory URI.

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



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

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

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

6.2. Directory

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

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

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

{
  "new-reg": "https://example.com/acme/new-reg",
  "recover-reg": "https://example.com/acme/recover-reg",
  "new-authz": "https://example.com/acme/new-authz",
  "new-cert": "https://example.com/acme/new-cert",
  "revoke-cert": "https://example.com/acme/revoke-cert"
}
```

6.3. Registration

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

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

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

The server MUST ignore any values provided in the "key", "authorizations", and "certificates" fields in registration bodies sent by the client, as well as any other fields that it does not recognize. If new fields are specified in the future, the specification of those fields MUST describe whether they may be provided by the client.

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

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

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

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

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

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

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

Servers SHOULD NOT respond to GET requests for registration resources as these requests are not authenticated. If a client wishes to query the server for information about its account (e.g., to examine the "contact" or "certificates" fields), then it SHOULD do so by sending a POST request with an empty update. That is, it should send a JWS whose payload is trivial ({"resource":"reg"}).

6.3.1. Recovery Keys

If the client wishes to establish a secret key with the server that it can use to recover this account later (a "recovery key"), then it must perform a simple key agreement protocol as part of the new-registration transaction. The client and server perform an ECDH exchange through the new-registration transaction (using the technique in Section 5.6), and the result is the recovery key.

To request a recovery key, the client includes a "recoveryKey" field in its new-registration request. The value of this field is a JSON object.

client (required, JWK): The client's ECDH public key

length (required, number): The length of the derived secret, in octets.

In the client's request, this object contains a JWK for a random ECDH public key generated by the client and the client-selected length value. Clients need to choose length values that balance security and usability. On the one hand, a longer secret makes it more difficult for an attacker to recover the secret when it is used for recovery (see Section 6.4.1). On the other hand, clients may wish to make the recovery key short enough for a user to easily write it down.

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

{
  "resource": "new-reg",
  "contact": [
    "mailto:cert-admin@example.com",
    "tel:+12025551212"
  ],
  "recoveryKey": {
    "client": { "kty": "EC", ... },
    "length": 128
  }
}
/* Signed as JWS */
```

The server MUST validate that the elliptic curve ("crv") and length value chosen by the client are acceptable, and that it is otherwise willing to create a recovery key. If not, then it MUST reject the new-registration request.

If the server agrees to create a recovery key, then it generates its own random ECDH key pair and combines it with with the client's public key as described in Section 5.6 above, using the label "recovery". The derived secret value is the recovery key. The server then returns to the client the ECDH key that it generated. The server MUST generate a fresh key pair for every transaction.

server (required, JWK): The server's ECDH public key

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: https://example.com/acme/reg/asdf

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

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

  "recoveryKey": {
    "server": { "kty": "EC", ... }
  }
}
```

On receiving the server's response, the client can compute the recovery key by combining the server's public key together with the private key corresponding to the public key that it sent to the server.

Clients may refresh the recovery key associated with a registration by sending a POST request with a new recoveryKey object. If the server agrees to refresh the recovery key, then it responds in the same way as to a new registration request that asks for a recovery key.

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

{
  "resource": "reg",
  "recoveryKey": {
    "client": { "kty": "EC", ... }
  }
}
/* Signed as JWS */
```

6.4. Account Recovery

Once a client has created an account with an ACME server, it is possible that the private key for the account will be lost. The recovery contacts included in the registration allows the client to recover from this situation, as long as it still has access to these contacts.

By "recovery", we mean that the information associated with an old account key is bound to a new account key. When a recovery process succeeds, the server provides the client with a new registration whose contents are the same as base registration object - except for the "key" field, which is set to the new account key. The server reassigns resources associated with the base registration to the new registration (e.g., authorizations and certificates). The server SHOULD delete the old registration resource after it has been used as a base for recovery.

In addition to the recovery mechanisms defined by ACME, individual client implementations may also offer implementation-specific recovery mechanisms. For example, if a client creates account keys deterministically from a seed value, then this seed could be used to recover the account key by re-generating it. Or an implementation could escrow an encrypted copy of the account key with a cloud storage provider, and give the encryption key to the user as a recovery value.

6.4.1. MAC-Based Recovery

With MAC-based recovery, the client proves to the server that it holds a secret value established in the initial registration transaction. The client requests MAC-based recovery by sending a MAC over the new account key, using the recovery key from the initial registration.

method (required, string): The string "mac"

base (required, string): The URI for the registration to be recovered.

mac (required, string): A JSON-formatted JWS object using an HMAC algorithm, whose payload is the JWK representation of the public key of the new account key pair.

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

{
  "resource": "recover-reg",
  "method": "mac",
  "base": "https://example.com/acme/reg/asdf",
  "mac": {
    "header": { "alg": "HS256" },
    "payload": base64(JWK(newAccountKey)),
    "signature": "5wUrDI3eAaV4wl2Rfj3aC0Pp--XB3t4YYuNgacv_D3U"
  }
}
/* Signed as JWS, with new account key */
```

On receiving such a request the server MUST verify that:

- The base registration has a recovery key associated with it
- The "alg" value in the "mac" JWS represents a MAC algorithm
- The "mac" JWS is valid according to the validation rules in [RFC7515], using the recovery key as the MAC key
- The JWK in the payload represents the new account key (i.e. the key used to verify the ACME message)

If those conditions are met, and the recovery request is otherwise acceptable to the server, then the recovery process has succeeded. The server creates a new registration resource based on the base registration and the new account key, and returns it on a 201 (Created) response, together with a Location header indicating a URI for the new registration. If the recovery request is unsuccessful, the server returns an error response, such as 403 (Forbidden).

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

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

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

  "authorizations": "...",
  "certificate": "..."
}
```

6.4.2. Contact-Based Recovery

In the contact-based recovery process, the client requests that the server send a message to one of the contact URIs registered for the account. That message indicates some action that the server requires the client's user to perform, e.g., clicking a link in an email. If the user successfully completes the server's required actions, then the server will bind the account to the new account key.

(Note that this process is almost entirely out of band with respect to ACME. ACME only allows the client to initiate the process, and the server to indicate the result.)

To initiate contact-based recovery, the client sends a POST request to the server's recover-registration URI, with a body specifying which registration is to be recovered. The body of the request MUST be signed by the client's new account key pair.

method (required, string): The string "contact"

base (required, string): The URI for the registration to be recovered.

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

{
  "resource": "recover-reg",
  "method": "contact",
  "base": "https://example.com/acme/reg/asdf",
  "contact": [
    "mailto:forgetful@example.net"
  ]
}
/* Signed as JWS, with new account key */
```

If the server agrees to attempt contact-based recovery, then it creates a new registration resource containing a stub registration object. The stub registration has the client's new account key and contacts, but no authorizations or certificates associated. The server returns the stub contact in a 201 (Created) response, along with a Location header field indicating the URI for the new registration resource (which will be the registration URI if the recovery succeeds).

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: https://example.com/acme/reg/qwer

{
  "key": { /* new account key from JWS header */ },

  "contact": [
    "mailto:forgetful@example.net"
  ]
}
```

After recovery has been initiated, the server follows its chosen recovery process, out-of-band to ACME. While the recovery process is ongoing, the client may poll the registration resource's URI for status, by sending a POST request with a trivial body ({"resource":"reg"}). If the recovery process is still pending, the server sends a 202 (Accepted) status code, and a Retry-After header field. If the recovery process has failed, the server sends an error code (e.g., 404), and SHOULD delete the stub registration resource.

If the recovery process has succeeded, then the server will send a 200 (OK) response, containing the full registration object, with any necessary information copied from the old registration). The client may now use this in the same way as if he had gotten it from a new-registration transaction.

6.5. Identifier Authorization

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

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

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

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

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

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

Before processing the authorization further, the server SHOULD determine whether it is willing to issue certificates for the identifier. For example, the server should check that the identifier is of a supported type. Servers might also check names against a blacklist of known high-value identifiers. If the server is unwilling to issue for the identifier, it SHOULD return a 403

(Forbidden) error, with a problem document describing the reason for the rejection.

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

- "identifier" the identifier submitted by the client.
- "status": MUST be "pending"
- "challenges" and "combinations": As selected by the server's policy for this identifier
- The "expires" field MUST be absent.

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

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

{
  "status": "pending",

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

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

  "combinations": [
    [0, 2],
    [1, 2]
  ]
}
```

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. For example, if the client wishes to complete the "simpleHttp" challenge, it needs to provide the "path" component. (This is also the stage where the client should perform any actions required by the challenge.)

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

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

```
POST /acme/authz/asdf/0 HTTP/1.1
Host: example.com

{
  "resource": "challenge",
  "type": "simpleHttp",
  "path": "Hf5GrX4Q7EBax9hc2jJnfw"
}
/* Signed as JWS */
```

The server updates the authorization document by updating its representation of the challenge with the response fields provided by the client. The server MUST ignore any fields in the response object that are not specified as response fields for this type of challenge. The server provides a 200 (OK) response with the updated challenge object as its body.

Presumably, the client's responses provide the server with enough information to validate one or more challenges. The server is said to "finalize" the authorization when it has completed all the validations it is going to complete, and assigns the authorization a status of "valid" or "invalid", corresponding to whether it considers the account authorized for the identifier. If the final state is "valid", the server MUST add an "expires" field to the authorization. When finalizing an authorization, the server MAY remove the "combinations" field (if present) or remove any challenges still pending. The server SHOULD NOT remove challenges with status "invalid".

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

To check on the status of an authorization, the client sends a GET request to the authorization URI, and the server responds with the current authorization object. In responding to poll requests while the validation is still in progress, the server MUST return a 202 (Accepted) response with a Retry-After header field.

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

HTTP/1.1 200 OK

{
  "status": "valid",
  "expires": "2015-03-01",

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

  "challenges": [
    {
      "type": "simpleHttp"
      "status": "valid",
      "validated": "2014-12-01T12:05Z",
      "token": "IlirfxKKXAsHtmzK29Pj8A"
      "path": "Hf5GrX4Q7EBax9hc2jJnfw"
    }
  ]
}
```

6.6. 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 by sending a POST request to the server's new-certificate resource. The body of the POST is a JWS object whose JSON payload contains a Certificate Signing Request (CSR) [RFC2986]. The CSR encodes the parameters of the requested certificate; authority to issue is demonstrated by the JWS signature by an account key, from which the server can look up related authorizations.

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

```
POST /acme/new-cert HTTP/1.1
Host: example.com
Accept: application/pkix-cert

{
  "resource": "new-cert",
  "csr": "5jNudRx6Ye4HzKEqT5...FS6aKdZeGsysoCo4H9P",
}
/* Signed as JWS */
```

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

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

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

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

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

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

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

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

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

The server provides metadata about the certificate in HTTP headers. In particular, the server MUST include a Link relation header field [RFC5988] with relation "up" to provide a certificate under which this certificate was issued, and one with relation "author" to indicate the registration under which this certicate was issued. The server MAY also include an Expires header as a hint to the client about when to renew the certificate. (Of course, the real expiration of the certificate is controlled by the notAfter time in the certificate itself.)

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

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

[DER-encoded certificate]
```

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

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

To avoid unnecessary renewals, the CA may choose not to issue a renewed certificate until it receives such a request (if it even allows renewal at all). In such cases, if the CA requires some time to generate the new certificate, the CA MUST return a 202 (Accepted) response, with a Retry-After header field that indicates when the new certificate will be available. The CA MAY include the current (non-renewed) certificate as the body of the response.

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

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

6.7. Certificate Revocation

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

certificate (required, string): The certificate to be revoked, in the Base64-encoded version of the DER format. (Note: This field uses the same modified Base64-encoding rules used elsewhere in this document, so it is different from PEM.)

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

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

Revocation requests are different from other ACME request in that they can be signed either with an account key pair or the key pair in the certificate. Before revoking a certificate, the server MUST verify at least one of these conditions applies:

- the public key of the key pair signing the request matches the public key in the certificate.

- the key pair signing the request is an account key, and the corresponding account is authorized to act for all of the identifier(s) in the certificate.

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

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

--- or ---

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

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

7. Identifier Validation Challenges

There are few types of 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.

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

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

- Control the identifier in question

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

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 general structure of Challenge and Response payloads is as follows:

type (required, string): The type of Challenge or Response encoded in the object.

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

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

validated (optional, string): : The time at which this challenge was completed by the server, encoded in the format specified in RFC 3339 [RFC3339].

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

All additional fields are specified by the Challenge type. The server MUST ignore any values provided in the "uri", "status", "validated", and "error" fields of a Response payload. If the server sets a Challenge's "status" to "invalid", it SHOULD also include the "error" field to help the client diagnose why they failed the challenge.

Different challenges allow the server to obtain proof of different aspects of control over an identifier. In some challenges, like Simple HTTP and DVSNI, the client directly proves its ability to do certain things related to the identifier. In the Proof of Possession challenge, 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 CA may choose different sets of challenges depending on whether it has interacted with a domain before, and how. For example:

- New domain with no known certificates: Domain Validation (DVSNI or Simple HTTP)
- Domain for which known certs exist from other CAs: DV + Proof of Possession of previous CA-signed key
- Domain with a cert from this CA, lost account key: DV + PoP of ACME-certified Subject key
- Domain with a cert from this CA, all keys and recovery mechanisms lost: Out of band proof of authority for the domain

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.

7.1. Simple HTTP

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

As a domain may resolve to multiple IPv4 and IPv6 addresses, the server will connect to at least one of the hosts found in A and AAAA records, at its discretion. The HTTP server may be made available over either HTTPS or unencrypted HTTP; the client tells the server in its response which to check.

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

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": "simpleHttp",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ+PCt92wr+oA"
}
```

A client responds to this challenge by signing a JWS object and provisioning it as a resource on the HTTP server for the domain in question. The payload of the JWS MUST be a JSON dictionary containing the fields "type", "token", "path", and "tls" from the

ACME challenge and response (see below), and no other fields. If the "tls" field is not included in the response, then validation object MUST have its "tls" field set to "true". The JWS MUST be signed with the client's account key pair. This JWS is NOT REQUIRED to have a "nonce" header parameter (as with the JWS objects that carry ACME request objects), but MUST otherwise meet the guidelines laid out in Section 3.

```
{
  "type": "simpleHttp",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA",
  "tls": false
}
```

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.

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

The client's response to this challenge indicates whether it would prefer for the validation request to be sent over TLS:

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

tls (optional, boolean, default true): If this attribute is present and set to "false", the server will perform its validation check over unencrypted HTTP (on port 80) rather than over HTTPS. Otherwise the check will be done over HTTPS, on port 443.

```
{
  "type": "simpleHttp",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJ-PCt92wr-oA",
  "tls": false
}
/* Signed as JWS */
```

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

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

   * the scheme field is set to "http" if the "tls" field in the response is present and set to false, and "https" otherwise;

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

   * the token field is the token provided in the challenge.

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

3. Dereference the URI using an HTTP or HTTPS GET request. If using HTTPS, the ACME server MUST ignore the certificate provided by the HTTPS server.

4. Verify that the Content-Type header of the response is either absent, or has the value "application/jose+json"

5. Verify that the body of the response is a valid JWS of the type indicated by the Content-Type header (if present), signed with the client's account key

6. Verify that the payload of the JWS meets the following criteria:

   * It is a valid JSON dictionary

   * It has exactly four fields

   * Its "type" field is set to "simpleHttp"

   * Its "token" field is equal to the "token" field in the challenge

   * Its "path" field is equal to the "path" field in the response

   * Its "tls" field is equal to the "tls" field in the response, or "true" if the "tls" field was absent

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

7.2. Domain Validation with Server Name Indication (DVSNI)

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

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

token (required, string): A random value with at least 128 bits of entropy, base64-encoded

```
{
  "type": "dvsni",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJyPCt92wrDoA",
}
```

In response to the challenge, the client uses its account private key to sign a JWS over a JSON object describing the challenge. The validation object covered by the signature MUST have the following fields and no others:

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

token (required, string): A random value with at least 128 bits of entropy, base64-encoded

```
{
  "type": "dvsni",
  "token": "evaGxfADs6pSRb2LAv9IZf17Dt3juxGJyPCt92wrDoA",
}
```

The client serializes the validation object to UTF-8, then uses its account private key to sign a JWS with the serialized JSON object as its payload. This JWS is NOT REQUIRED to have the "nonce" header parameter.

The client will compute Z, the SHA-256 of the "signature" value from the JWS. The hash is calculated over the base64-encoded signature string. Z is encoded in hexadecimal form.

The client will generate a self-signed certificate with the subjectAlternativeName extension containing the dNSName "<Z[0:32]>.<Z[32:64]>.acme.invalid". The client will then configure the TLS server at the domain such that when a handshake is initiated with the Server Name Indication extension set to "<Z[0:32]>.<Z[32:64]>.acme.invalid", the generated test certificate is presented.

The response to the DVSNI challenge provides the validation JWS to the server.

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

validation (required, string): The JWS object computed with the validation object and the account key

```
{
  "type": "dvsni",
  "validation": {
    "header": { "alg": "RS256" },
    "payload": "qzu9...6bjn",
    "signature": "gfj9XqFv07e1wU66hSLYkiFqYakPSjAu8TsyXRg85nM"
  }
}
```

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.

1. Verify the validation JWS using the account key for which the challenge was issued.

2. Decode the payload of the JWS as UTF-8 encoded JSON.

3. Verify that there are exactly two fields in the decoded object, and that:

   * The "type" field is set to "dvsni"

   * The "token" field matches the "token" value in the challenge

4. Open a TLS connection to the domain name being validated on port 443, presenting the value "<Z[0:32]>.<Z[32:64]>.acme.invalid" in the SNI field.

5. Verify that the certificate contains a subjectAltName extension with the dNSName of "<Z[0:32]>.<Z[32:64]>.acme.invalid".

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

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

7.3. 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 sign with that key. This challenge is meant to be used when the server knows of a public key that is

already associated with the identifier being claimed, and wishes for new authorizations to be authorized by the holder of the corresponding private key. For DNS identifiers, for example, this can help guard against domain hijacking.

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"

certs (optional, array of string): An array of certificates, in Base64-encoded DER format, that contain acceptable public keys.

```
{
  "type": "proofOfPossession",
  "certs": ["MIIF7z...bYVQLY"]
}
```

In response to this challenge, the client uses the private key corresponding to one of the acceptable public keys to sign a JWS object including data related to the challenge. The validation object covered by the signature has the following fields:

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

identifiers (required, identifier): A list of identifiers for which the holder of the prior key authorizes the new key

accountKey (required, JWK): The client's account public key

```
{
  "type": "proofOfPossession",
  "identifiers: [{"type": "dns", "value": "example.com"}],
  "accountKey": { "kty": "RSA", ... }
}
```

This JWS is NOT REQUIRED to have a "nonce" header parameter (as with the JWS objects that carry ACME request objects). This allows proof-of-possession response objects to be computed off-line. For example, as part of a domain transfer, the new domain owner might require the old domain owner to sign a proof-of-possession validation object, so that the new domain owner can present that in an ACME transaction later.

The validation JWS MUST contain a "jwk" header parameter indicating the public key under which the server should verify the JWS.

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"

authorization (required, JWS): The validation JWS

```
{
  "type": "proofOfPossession",
  "authorization": {
    "header": {
      "alg": "RS256",
      "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "n": "AMswMT...3aVtjE"
      }
    },
    "payload": "SfiR1...gSAl7A",
    "signature": "XcQLfL...cW5beg"
  }
}
```

To validate a proof-of-possession challenge, the server performs the following steps:

1. Verify that the public key in the "jwk" header of the "authorization" JWS corresponds to one of the certificates in the "certs" field of the challenge

2. Verify the "authorization" JWS using the key indicated in its "jwk" header

3. Decode the payload of the JWS as UTF-8 encoded JSON

4. Verify that there are exactly three fields in the decoded object, and that:

   * The "type" field is set to "proofOfPossession"

   * The "identifier" field contains the identifier for which authorization is being validated

   * The "accountKey" field matches the account key for which the challenge was issued

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

7.4. DNS

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

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

token (required, string): A random value with at least 128 bits of entropy. It MUST NOT contain any characters outside the URL-safe Base64 alphabet.

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

In response to this challenge, the client uses its account private key to sign a JWS over a JSON object describing the challenge. The validation object covered by the signature MUST have the following fields and no others:

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

token (required, string): The token value in the challenge

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

The client serializes the validation object to UTF-8, then uses its account private key to sign a JWS with the serialized JSON object as its payload. This JWS is NOT REQUIRED to have the "nonce" header parameter.

The record provisioned to the DNS is the "signature" value from the JWS, i.e., the base64-encoded signature value. The client constructs the validation domain name by appending the label "_acme-challenge" to the domain name being validated, then provisions a TXT record with the signature value under that name. For example, if the domain name being validated is "example.com", then the client would provision the following DNS record:

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

The response to a DNS challenge provides the validation JWS to the server.

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

validation (required, JWS): The JWS object computed with the validation object and the account key

```
{
  "type": "dns"
  "clientPublicKey": { "kty": "EC", ... },
  "validation": {
    "header": { "alg": "HS256" },
    "payload": "qzu9...6bjn",
    "signature": "gfj9XqFv07e1wU66hSLYkiFqYakPSjAu8TsyXRg85nM"
  }
}
```

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

1. Verify the validation JWS using the account key for which this challenge was issued

2. Decode the payload of the JWS as UTF-8 encoded JSON

3. Verify that there are exactly two fields in the decoded object, and that:

   * The "type" field is set to "dns"

   * The "token" field matches the "token" value in the challenge

4. Query for TXT records under the validation domain name

5. Verify that the contents of one of the TXT records match the "signature" value in the "validation" JWS

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

8. IANA Considerations

TODO

- Register .well-known path
- Register Replay-Nonce HTTP header
- Register "nonce" JWS header parameter
- Register "urn:acme" namespace
- Create identifier validation method registry
- Registries of syntax tokens, e.g., message types / error types?

9. Security Considerations

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

1. Only an entity that controls a identifier can get an account key authorized for that identifier

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

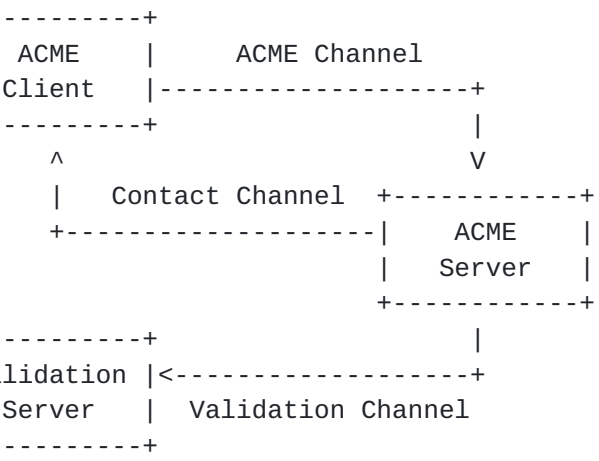
In this section, we discuss the threat model that underlies ACME and the ways that ACME achieves these security goals within that threat

model. We also discuss the denial-of-service risks that ACME servers face, and a few other miscellaneous considerations.

9.1. Threat model

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

- An ACME channel, over which the ACME HTTPS requests are exchanged
- A validation channel, over which the ACME server performs additional requests to validate a client's control of an identifier
- A contact channel, over which the ACME server sends messages to the registered contacts for ACME clients



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

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

On the ACME channel, in addition to network-layer attackers, we also need to account for application-layer man in the middle attacks, and for abusive use of the protocol itself. Protection against

application-layer MitM addresses potential attackers such as Content Distribution Networks (CDNs) and middleboxes with a TLS MitM function. Preventing abusive use of ACME means ensuring that an attacker with access to the validation or contact channels can't obtain illegitimate authorization by acting as an ACME client (legitimately, in terms of the protocol).

9.2. Integrity of Authorizations

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

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

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

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

- Simple HTTP: The value provided in the validation request is signed by the account private key.

- DVSNI: The validation TLS request uses the account key pair as the server's key pair.

- DNS: The MAC covers the account key, and the MAC key is derived from an ECDH public key signed with the account private key.

- Proof of possession of a prior key: The signature by the prior key covers the account public key.

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

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

- DVSNI: Configure a TLS server for the domain

- DNS: Provision DNS resource records for the domain

- Proof of possession of a prior key: Sign using the private key specified by the server

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

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

More limited forms of delegation can also lead to an unintended party gaining the ability to successfully complete a validation transaction. For example, suppose an ACME server follows HTTP redirects in Simple HTTP validation and a web site operator provisions a catch-all redirect rule that redirects requests for unknown resources to different domain. Then the target of the redirect could use that to get a certificate through Simple 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 DVSNI or Simple HTTP validation query to a server of the attacker's choosing. There are a few different mitigations that ACME servers can apply:

- Checking the DNSSEC status of DNS records used in ACME validation (for zones that are DNSSEC-enabled)
- Querying the DNS from multiple vantage points to address local attackers
- Applying mitigations against DNS off-path attackers, e.g., adding entropy to requests [I-D.vixie-dnsext-dns0x20] or only using TCP

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

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

An active attacker on the validation channel can subvert the ACME process, by performing normal ACME transactions and providing a validation response for his own account key. The risks due to hosting providers noted above are a particular case. For identifiers where the server already has some credential associated with the domain this attack can be prevented by requiring the client to complete a proof-of-possession challenge.

9.3. Preventing Authorization Hijacking

The account recovery processes described in Section 6.4 allow authorization to be transferred from one account key to another, in case the former account key pair's private key is lost. ACME needs to prevent these processes from being exploited by an attacker to hijack the authorizations attached to one key and assign them to a key of the attacker's choosing.

Recovery takes place in two steps: 1. Provisioning recovery information (contact or recovery key) 2. Using recovery information to recover an account

The provisioning process needs to ensure that only the account key holder ends up with information that is useful for recovery. The recovery process needs to assure that only the (now former) account key holder can successfully execute recovery, i.e., that this entity is the only one that can choose the new account key that receives the capabilities held by the account being recovered.

MAC-based recovery can be performed if the attacker knows the account key and registration URI for the account being recovered. Both of these are difficult to obtain for a network attacker, because ACME uses HTTPS, though if the recovery key and registration URI are sufficiently predictable, the attacker might be able to guess them. An ACME MitM can see the registration URI, but still has to guess the recovery key, since neither the ECDH in the provisioning phase nor HMAC in the recovery phase will reveal it to him.

ACME clients can thus mitigate problems with MAC-based recovery by using long recovery keys. ACME servers should enforce a minimum recovery key length, and impose rate limits on recovery to limit an attacker's ability to test different guesses about the recovery key.

Contact-based recovery uses both the ACME channel and the contact channel. The provisioning process is only visible to an ACME MitM, and even then, the MitM can only observe the contact information provided. If the ACME attacker does not also have access to the contact channel, there is no risk.

The security of the contact-based recovery process is entirely dependent on the security of the contact channel. The details of this will depend on the specific out-of-band technique used by the server. For example:

- If the server requires a user to click a link in a message sent to a contact address, then the contact channel will need to ensure that the message is only available to the legitimate owner of the contact address. Otherwise, a passive attacker could see the link and click it first, or an active attacker could redirect the message.

- If the server requires a user to respond to a message sent to a contact address containing a secret value, then the contact channel will need to ensure that an attacker cannot observe the secret value and spoof a message from the contact address.

In practice, many contact channels that can be used to reach many clients do not provide strong assurances of the types noted above. In designing and deploying contact-based recovery schemes, ACME servers operators will need to find an appropriate balance between using contact channels that can reach many clients and using contact-based recovery schemes that achieve an appropriate level of risk using those contact channels.

9.4. Denial-of-Service Considerations

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

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

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

All of these attacks can be mitigated by the application of appropriate rate limits. Issues closer to the front end, like POST body validation, can be addressed using HTTP request limiting. For validation and certificate requests, there are other identifiers on which rate limits can be keyed. For example, the server might limit the rate at which any individual account key can issue certificates, or the rate at which validation can be requested within a given subtree of the DNS.

9.5. CA Policy Considerations

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

- Has the client agreed to a subscriber agreement?
- Is the claimed identifier syntactically valid?
- For domain names:
  - If the leftmost label is a '*', then have the appropriate checks been applied?

    * Is the name on the Public Suffix List?

    * Is the name a high-value name?

    * Is the name a known phishing domain?

  o Is the key in the CSR sufficiently strong?

  o Is the CSR signed with an acceptable algorithm?

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

10. 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.

11. References

11.1. Normative References

[I-D.ietf-appsawg-http-problem]
Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", draft-ietf-appsawg-http-problem-00 (work in progress), September 2014.

[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.

[RFC3339] Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.

[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.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 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.

[RFC5753] Turner, S. and D. Brown, "Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)", RFC 5753, January 2010.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.

[RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

[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.

[RFC7469] Evans, C., Palmer, C., and R. Sleevi, "Public Key Pinning Extension for HTTP", RFC 7469, April 2015.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, May 2015.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, May 2015.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, May 2015.

[SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", May 2009, <http://www.secg.org/sec1-v2.pdf>.

11.2. Informative References

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

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

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, July 2003.

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

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

Authors' Addresses

Richard Barnes
Mozilla

Email: rlb@ipv.sx

Jacob Hoffman-Andrews
EFF

Email: jsha@eff.org

James Kasten
University of Michigan

Email: jdkasten@umich.edu